Joint HP-SEE/LinkSCEEM/PRACE HPC Summer Training Optimization and Benchmarking

Chris Dahnken
Intel SSG EMEA HPCTC

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Optimization Notice

Optimization Notice

Intel® compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel® and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the "Intel® Compiler User and Reference Guides" under "Compiler Options." Many library routines that are part of Intel® compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel® compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel® compilers, associated libraries and associated development tools 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 Intel® Streaming SIMD Extensions 2 (Intel® SSE2), Intel® Streaming SIMD Extensions 3 (Intel® SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel® 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.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel® and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Notice revision #20101101

Agenda

Introduction

Processor technology

- Processor Core Technologies
- Memory Subsystem

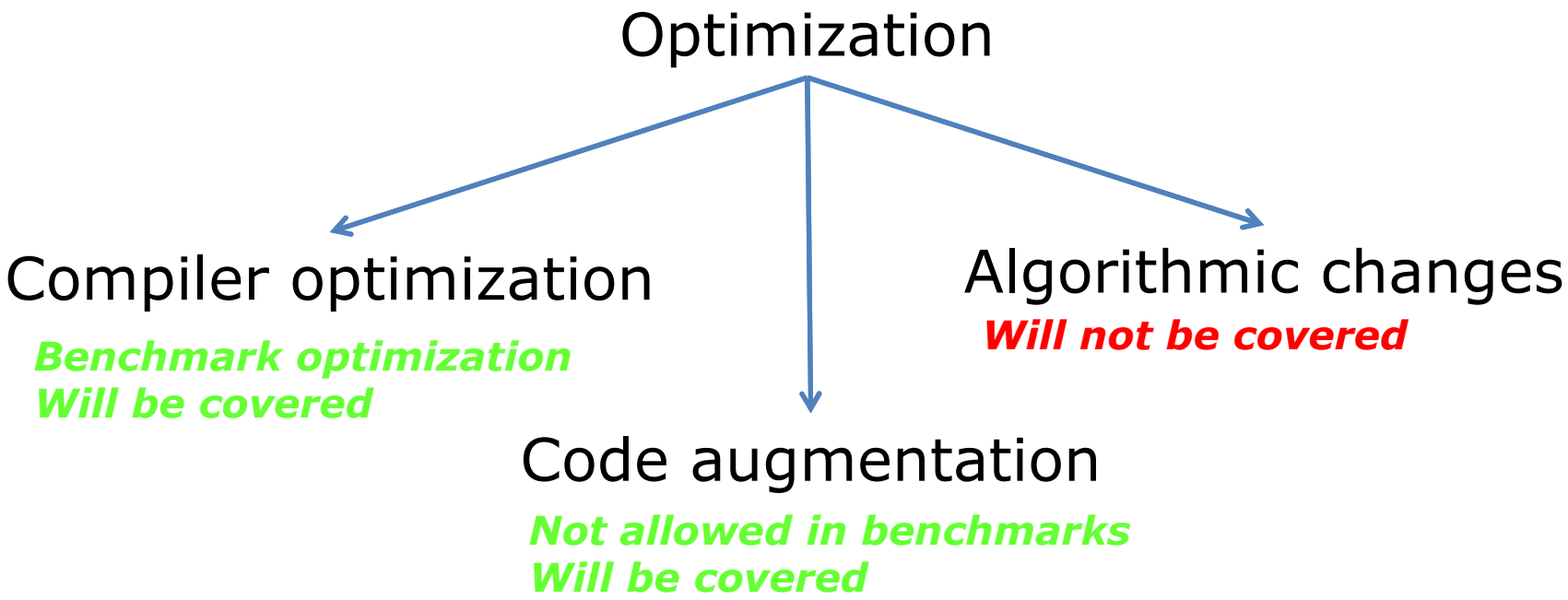
Measuring performance and finding bottlenecks

- Performance Monitoring Unit
- Performance Events

Optimization with the compiler

- Vectorization with the compiler
- Overcomming vectorization problems

Optimization and Benchmarking

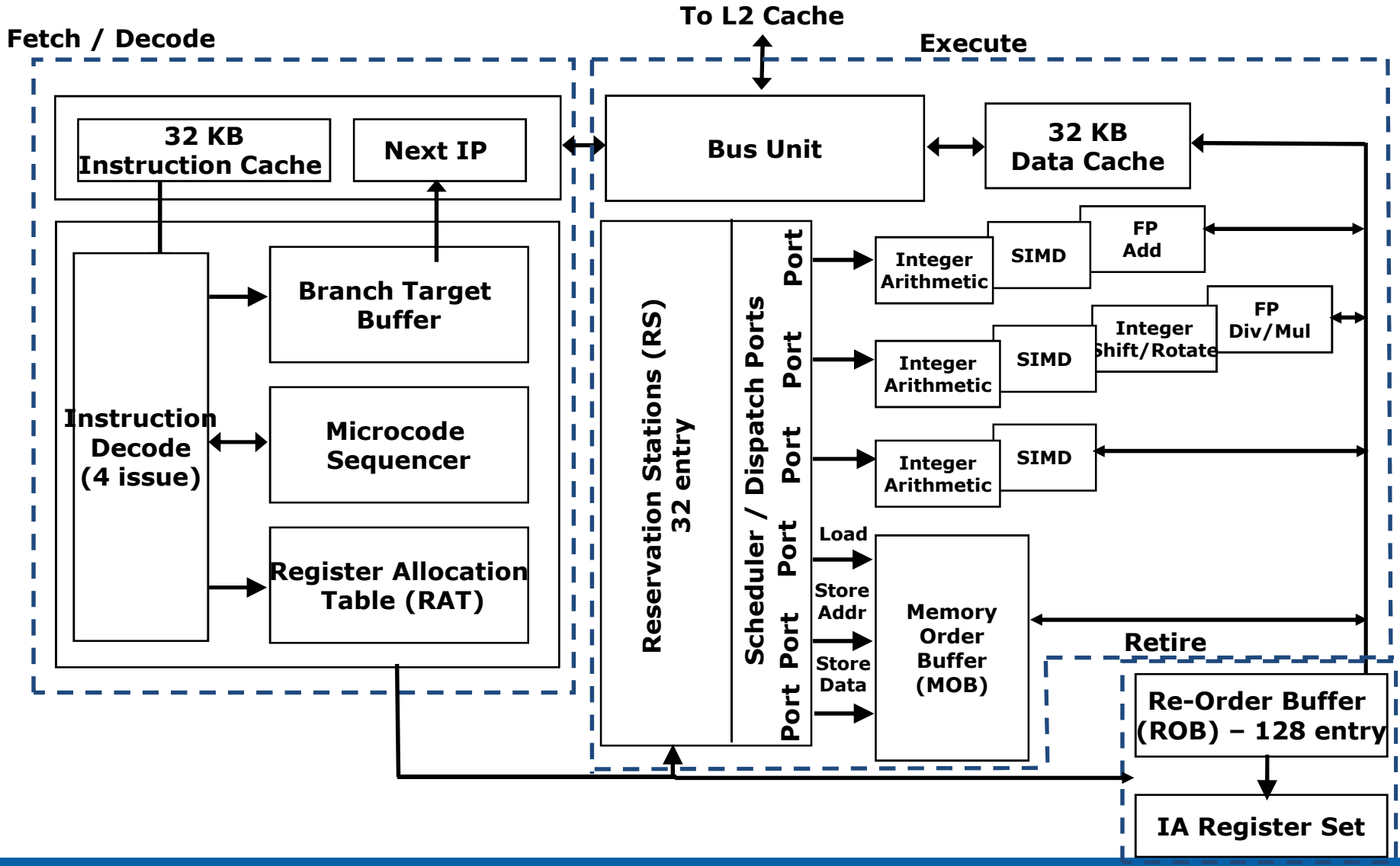


Generally, in benchmarking code changes are not accepted. Improvements can only be reached through compilers and libraries

Understanding the Platform

Processor Technology Crash Course
(Very simplified)

Core i7 Block Diagram



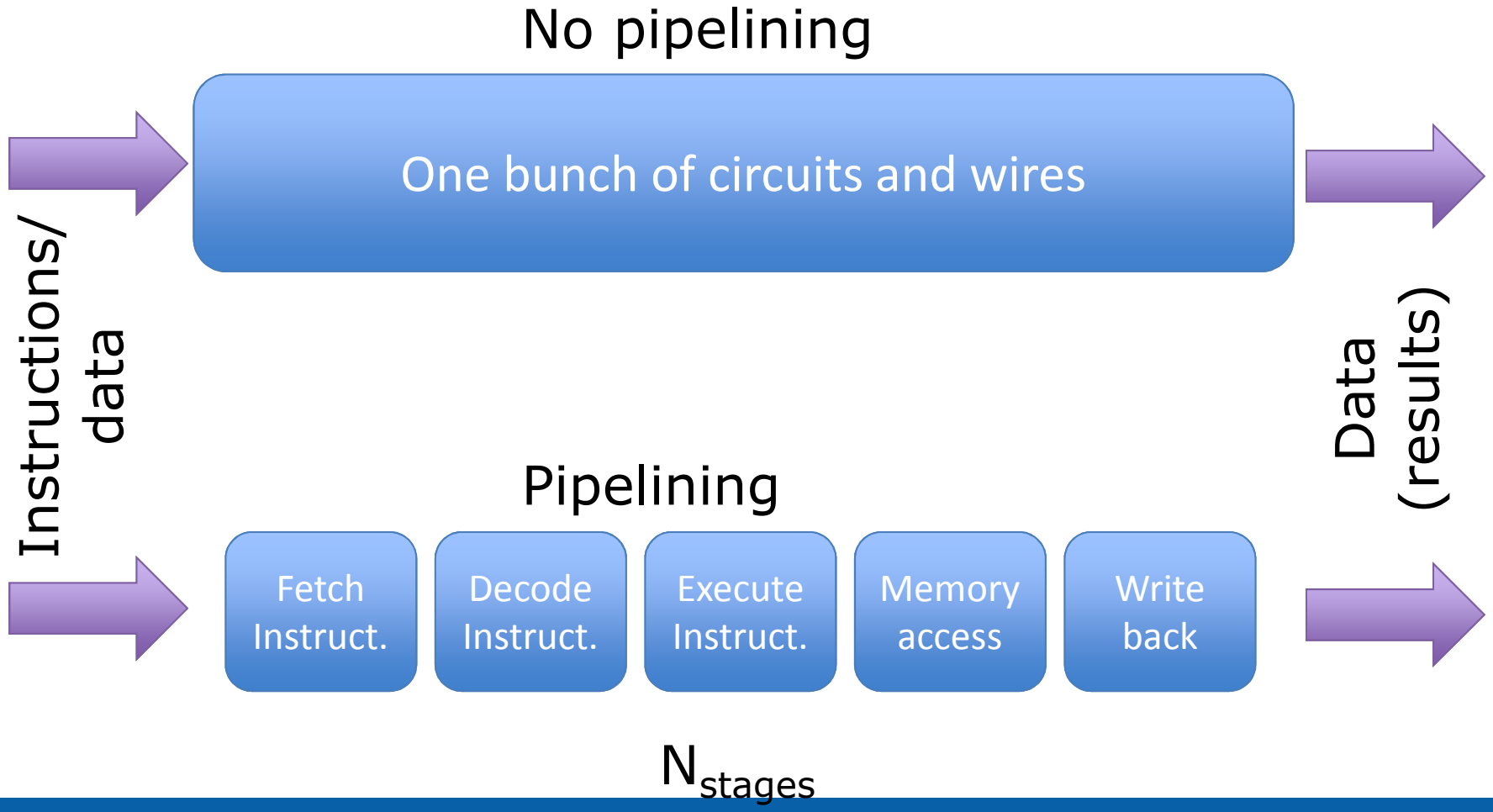
Core i7 / Sandy Bridge implement

- Pipelining
- Branch prediction
- Out-of-order execution
- Super-Scalarity
- SIMD execution

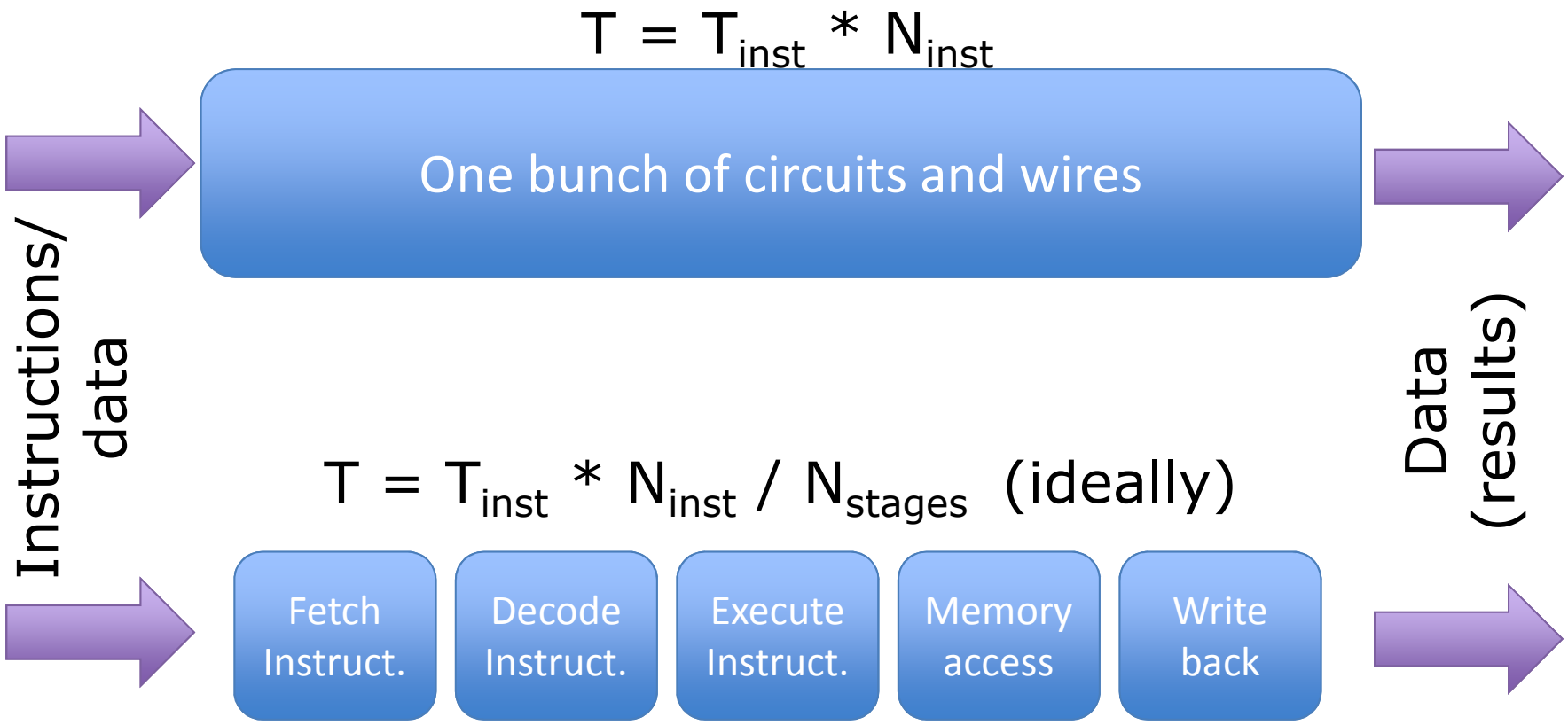
The optimal execution of a program depends on all of these technologies and their interplay working properly with a given code.

Pipelining

Patterson/Hennessy
"Computer Organization and Design – The Hardware/Software Interface"



Pipelining

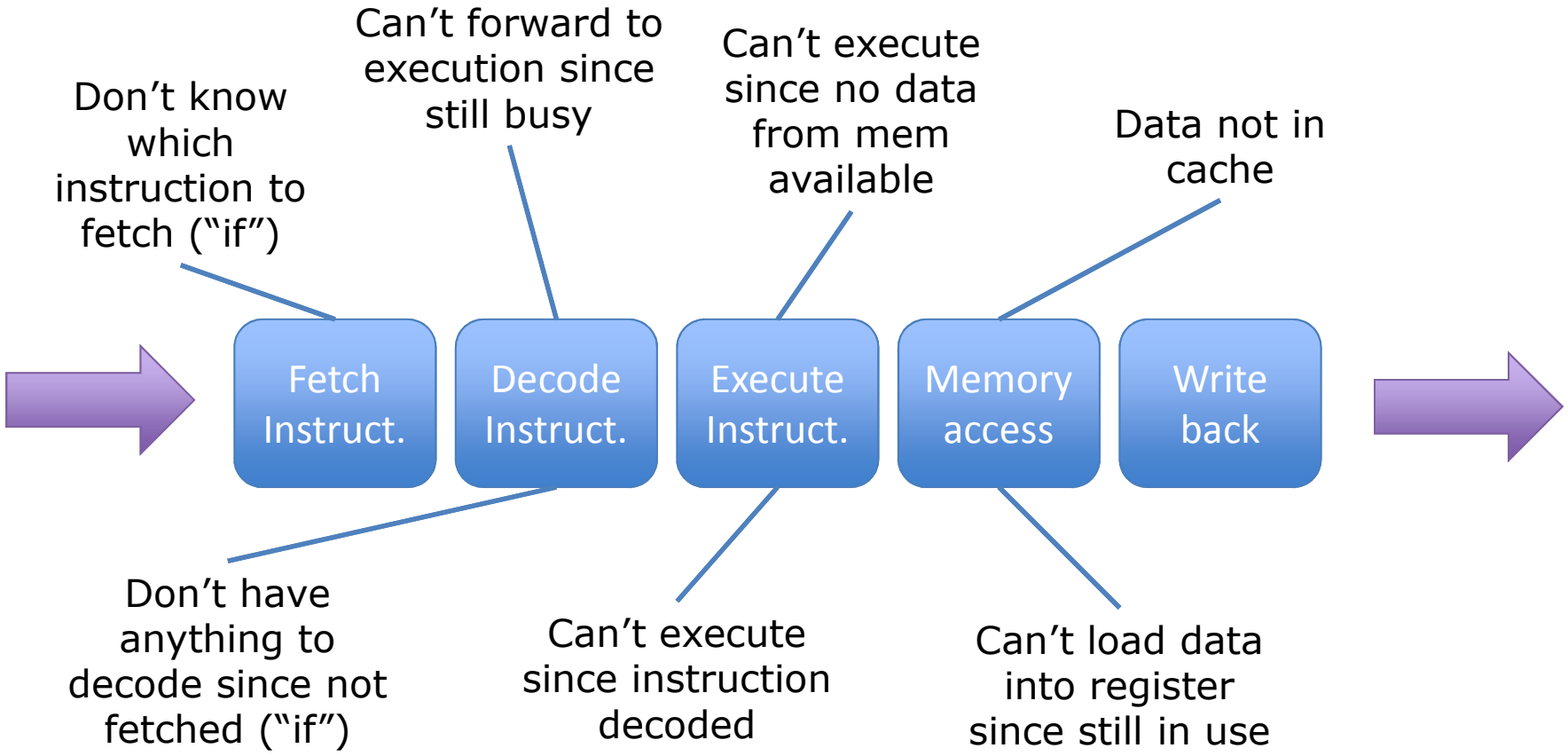

$T = T_{inst} * N_{inst}$
One bunch of circuits and wires
Instructions/ data
Data (results)
$T = T_{inst} * N_{inst} / N_{stages}$ (ideally)
Fetch Instruct.
Decode Instruct.
Execute Instruct.
Memory access
Write back

Pipelining

- Pipelining is a great method to improve performance
- All following techniques are designed to further improve the performance, but will also interact with the pipeline functionality
- All analysis and optimization methods hereafter are centered around keeping up or re-establishing the pipeline flow

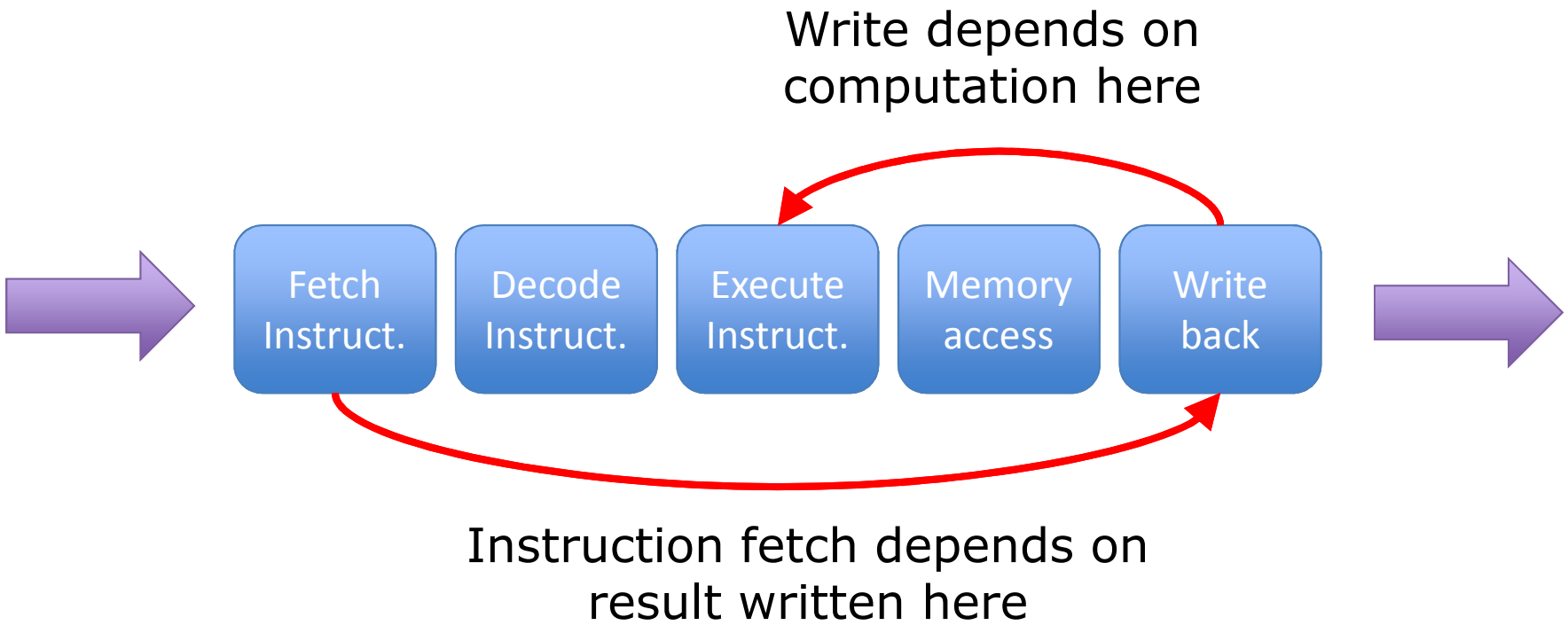
Pipeline Stalls

- Pipeline stalls appear when one stage waits for the previous or next to finish, e.g.



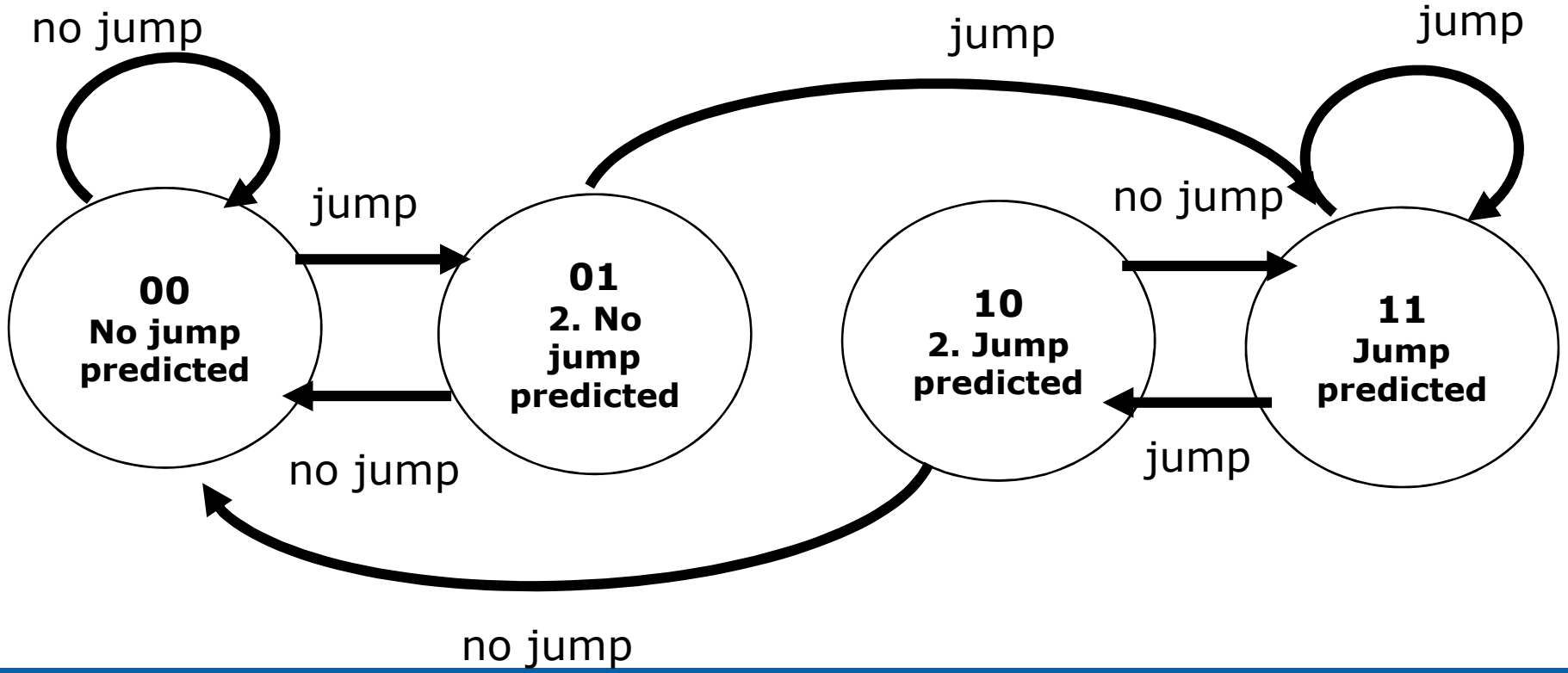
Branch Prediction

- When the code branches conditionally (at an "if" condition), the "fetch instruction" does not know which code path to fetch



Branch Prediction

- The processor has a buffer storing the last branches and will "guess" and speculatively execute this path, e.g.



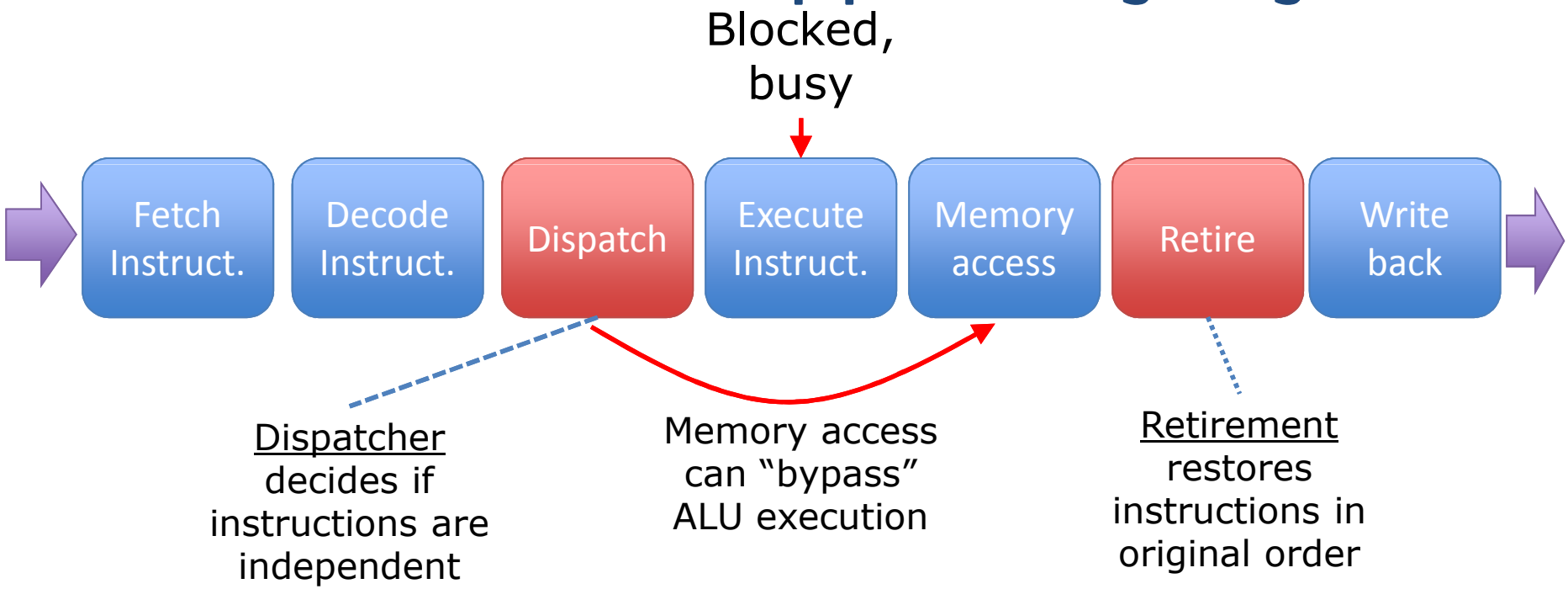
Branch Prediction

What happens if the prediction is wrong?

- The CPU will notice when the branch it predicted was wrong by watching the condition on which the branch depends
- If the wrong codepath was executed, the pipeline is cleared out ("pipeline flush") and the computation begins again at the point of the branch, now with the right codepath.

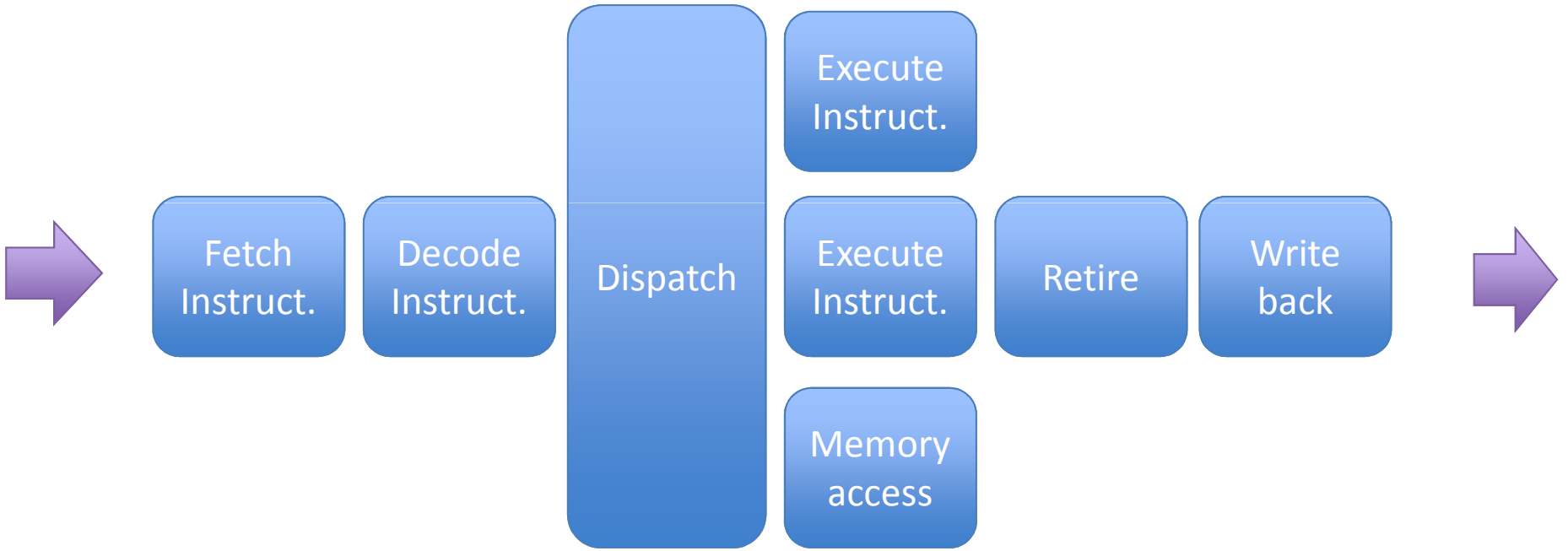
Out of order execution

- While one (decoded) instruction still executes, another independent instruction can be executed in a different pipeline stage, e.g.



SuperScalar execution

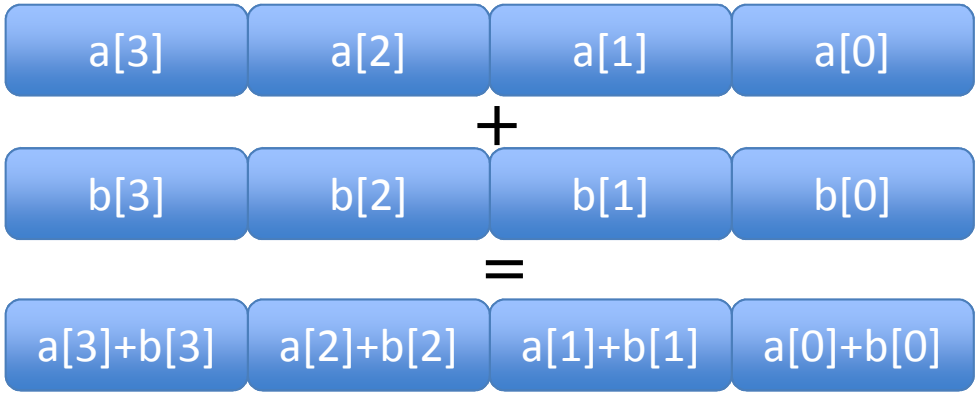
- No reason why some out-order execution can't be done simultaneously, e.g.



SIMD instructions

- A simple method to get more (arithmetic) operations done in the same timeframe using SIMD (Single Instruction, Multiple Data)
- SIMDization is done with a special type of registers that can execute N-many identical operations on all elements, e.g.

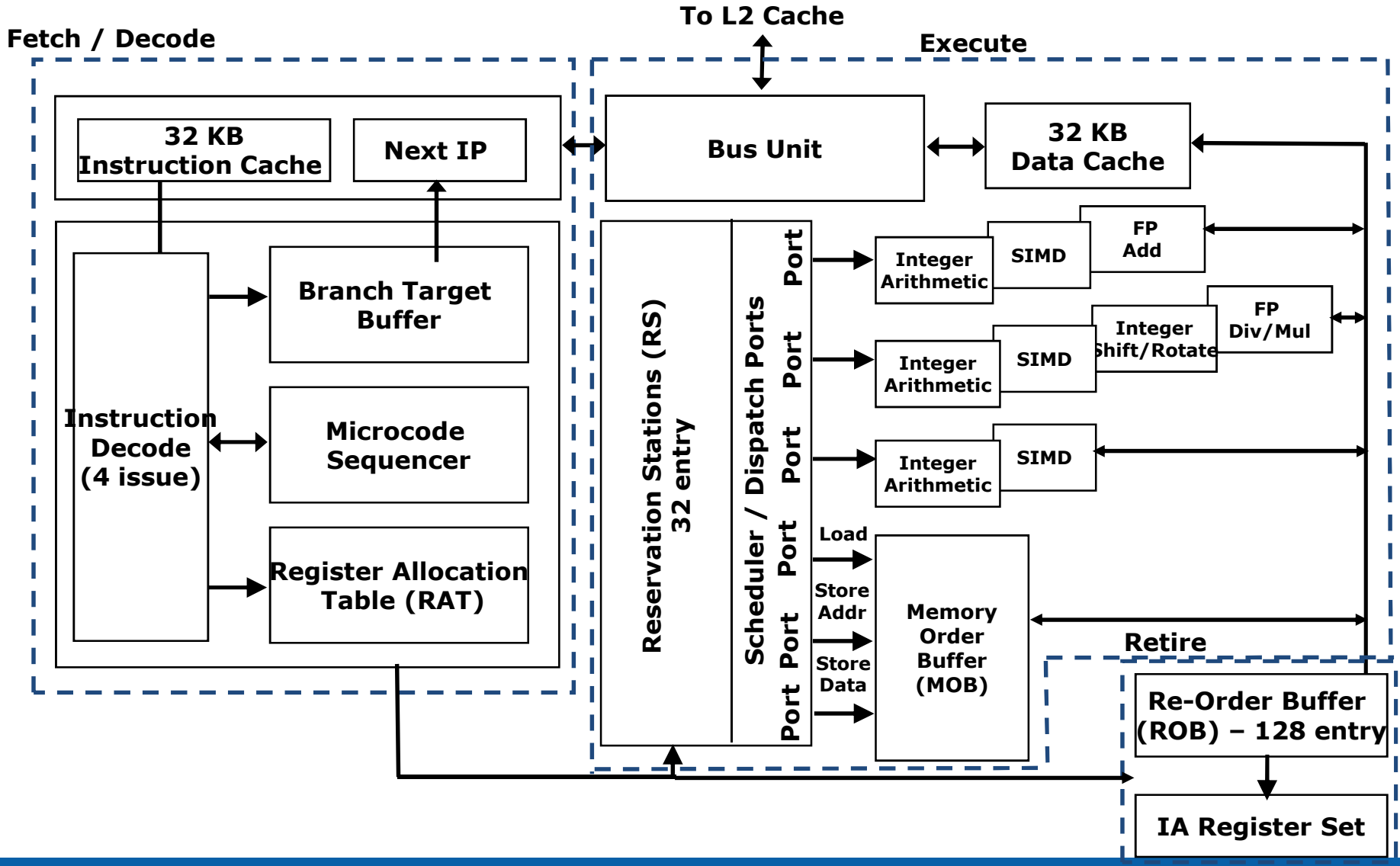
```
for(int i=0;i<4;i++){
        a[i]=b[i]+c[i]
}
```



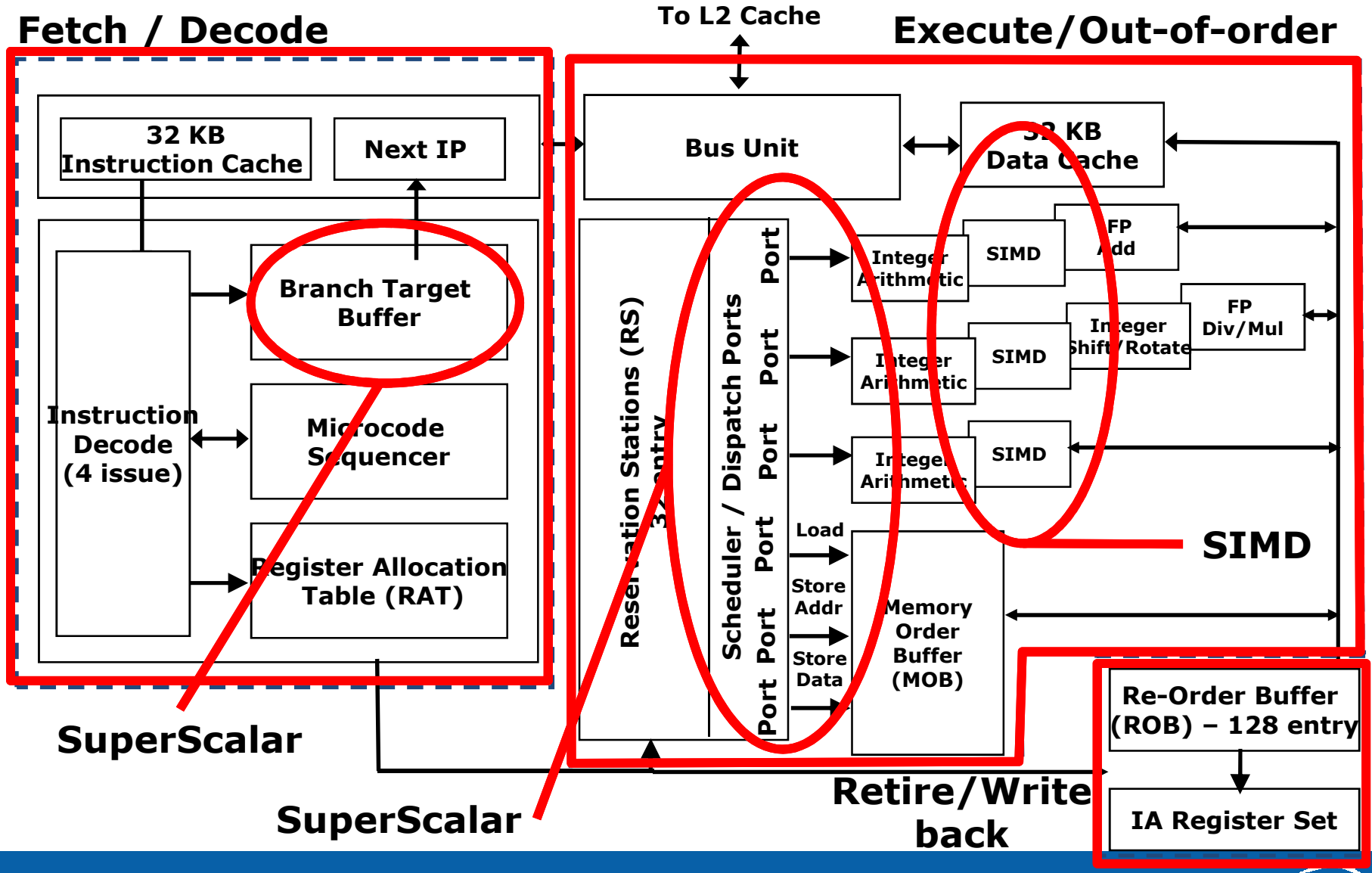
SIMD Instructions

- When talking about Peak Performance, this is always packed SIMD performance!
- In order to reach a considerable fraction of peak performance, your algorithms must vectorize!
- Current Intel SIMD instruction sets are
  - SSE (128bit, Streaming SIMD Instructions)
  - AVX (256bit, Advanced Vector Instructions)

Core i7 Block Diagram



Core i7 Block Diagram



Processor Technology Crash Course
Summary

- The central element of a CPU is the pipeline
- Focal point of optimization is to keep the pipeline busy!
- Important concepts to understand
  - Superscalar pipeline
  - Out-Of-Order execution
  - SIMD
  - Branch prediction

Understanding the Platform

Memory Subsystem

Understanding the memory subsystem

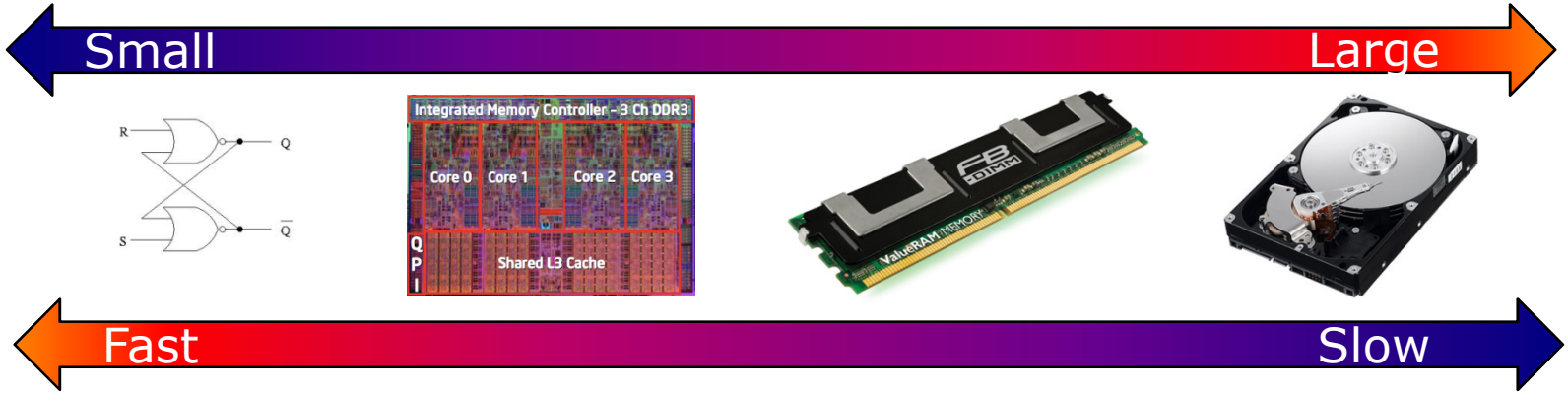
- Caches
- Cache lines
- Cache levels
- Main Memory

So far we know …

- … the main technologies a modern processing core is constructed with.
- Based on this knowledge, we could write a program that uses the core efficiently if all necessary data would be available there.
- But most programs mainly process data that is stored in main memory.
- For such programs it is likely that the bottleneck of the computation is found in the memory accesses, not in the actual computation.

Principle of locality

- Programmers want an unlimited amount of fast memory! Impossible!
- But: Not all data is accessed uniformly
- **Principle of locality**: Data that is closer together (spatial locality) is more often access at the same time (temporal locality).
- This, plus the fact that smaller hardware can be constructed faster, leads to the idea of a **memory hierarchy based of different sizes and speeds.**



Cache types

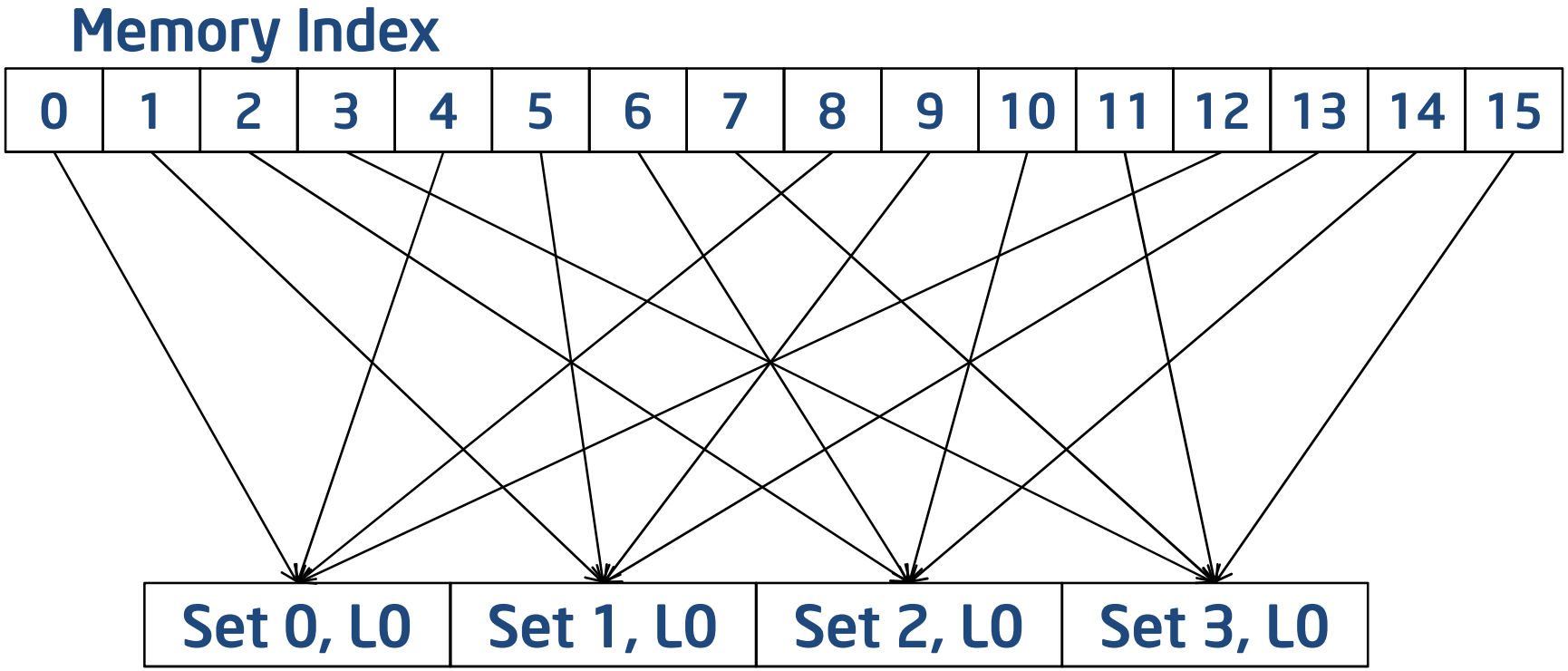
Caches store memory in contiguous chunks called Cache Line
On Intel64 the cache line is 64 byte

Two general strategies

Fully Associative
Any memory address can be stored in any cache line

Direct Mapped
Each memory address can stored in only one cache line

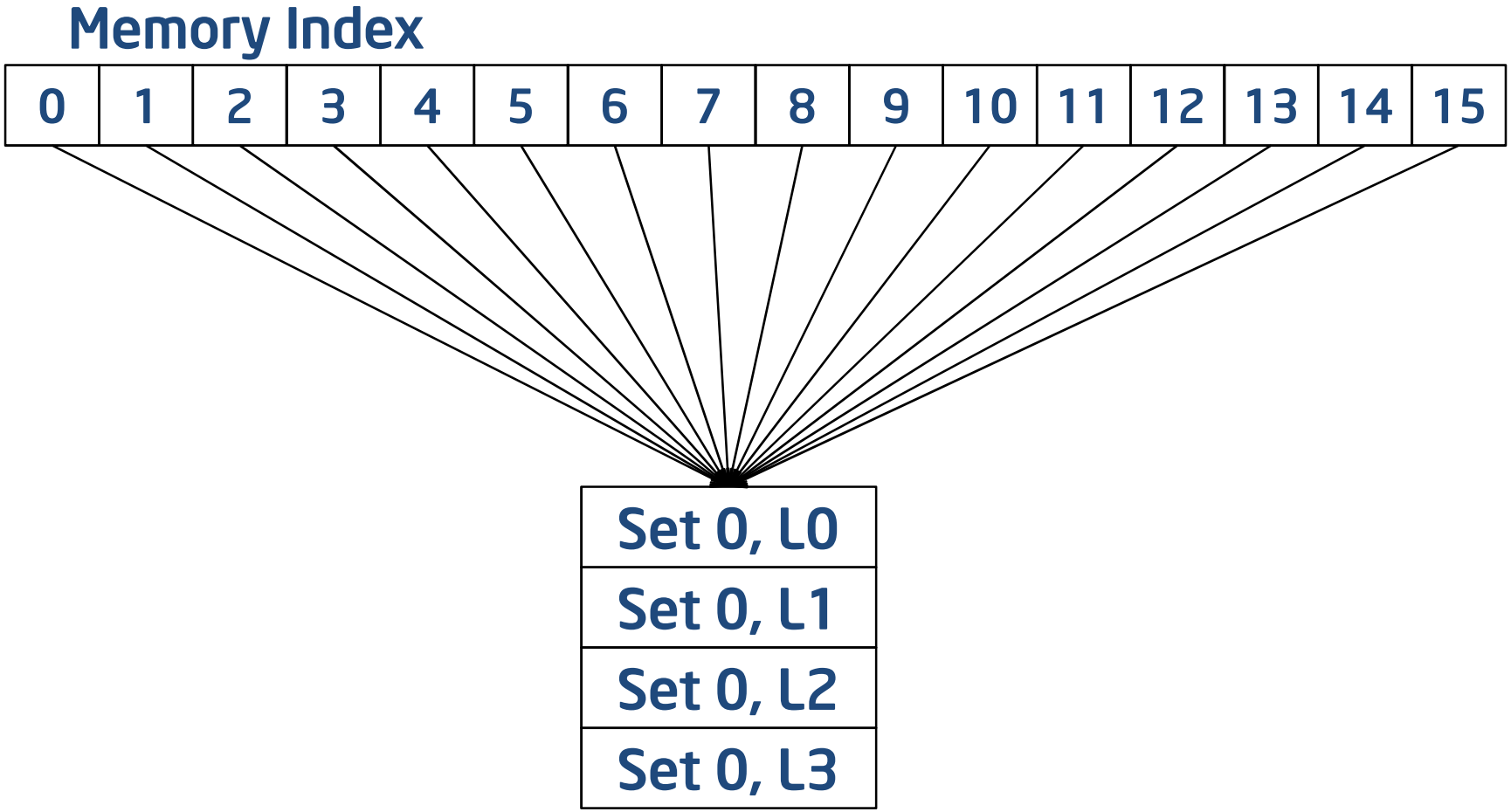
Direct Mapped Cache



Set number = Memory index % Number of sets

Each line in a set will be immediatelly replace

Fully Associative



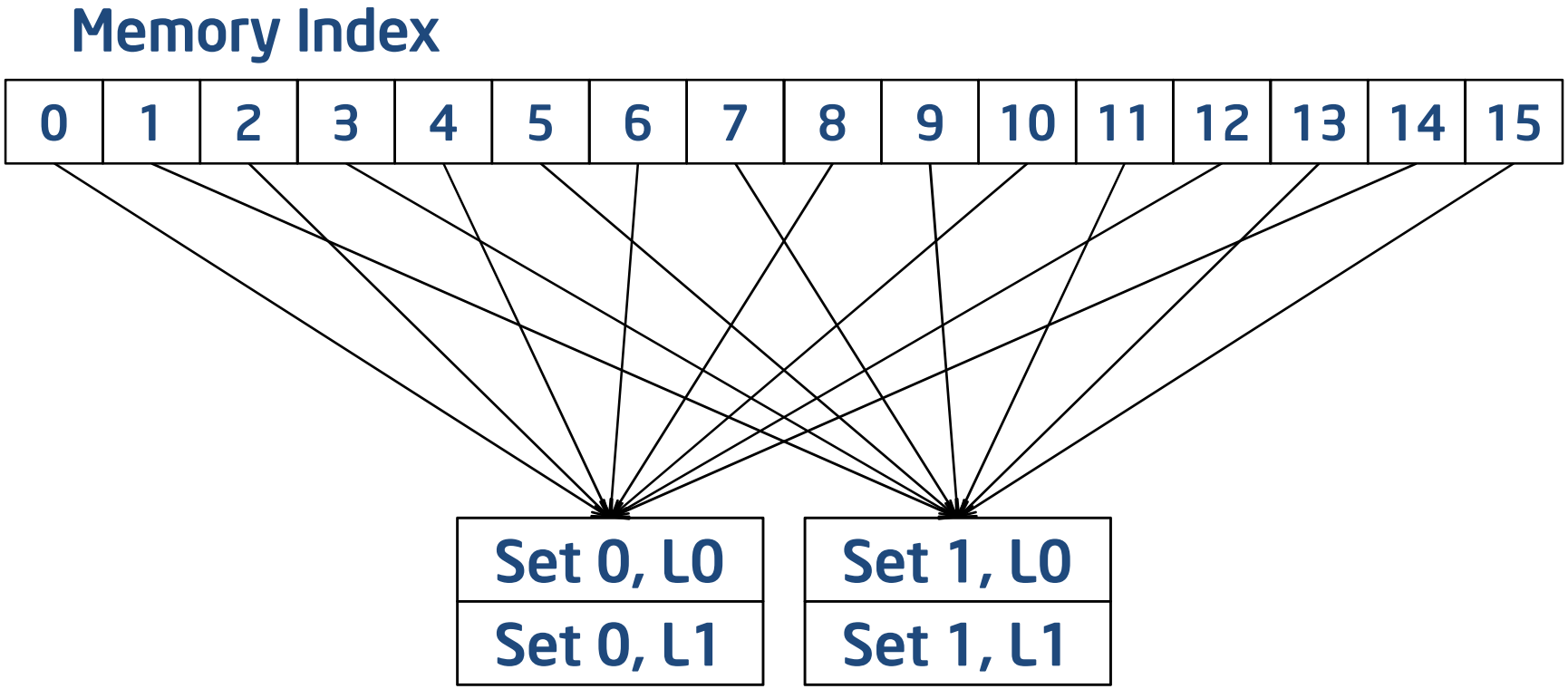
Replacement policy! Usually LRU (last recently used)!

Pros and cons of direct and fully accoiative caches

- Direct mapped
  - Less chip space
  - Fast
  - Lower hit rates due to conflicts
- Fully associative
  - More chip space
  - Slow
  - High hit rates

- Need a compromise here!

Set Associative (2 Way)

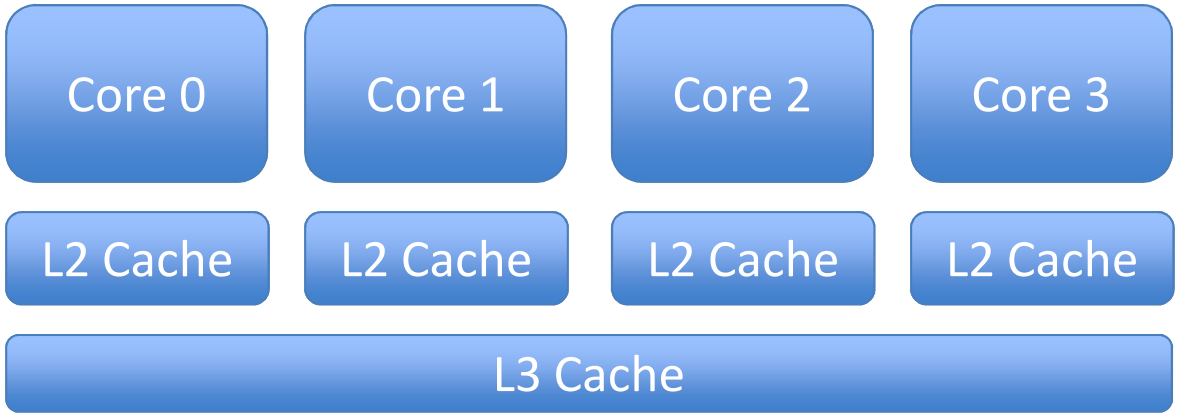


Multi Level Cache

- In order keep memory access up with the increases in CPU speed, modern CPUs feature multi-level caches.
- Caches are number after their levels L1 (Closest to the core, fastest, smallest) to LN (furthest away, slowest access, largest).
- In current Intel CPUs three caches levels can be found
- L1 or Data Cache Unit (DCU) 8 way set associative
- L2 or Mid Level cache (MLC) 8 way set associative
- L3 or Last Level Cache (LLC) 16 way associative

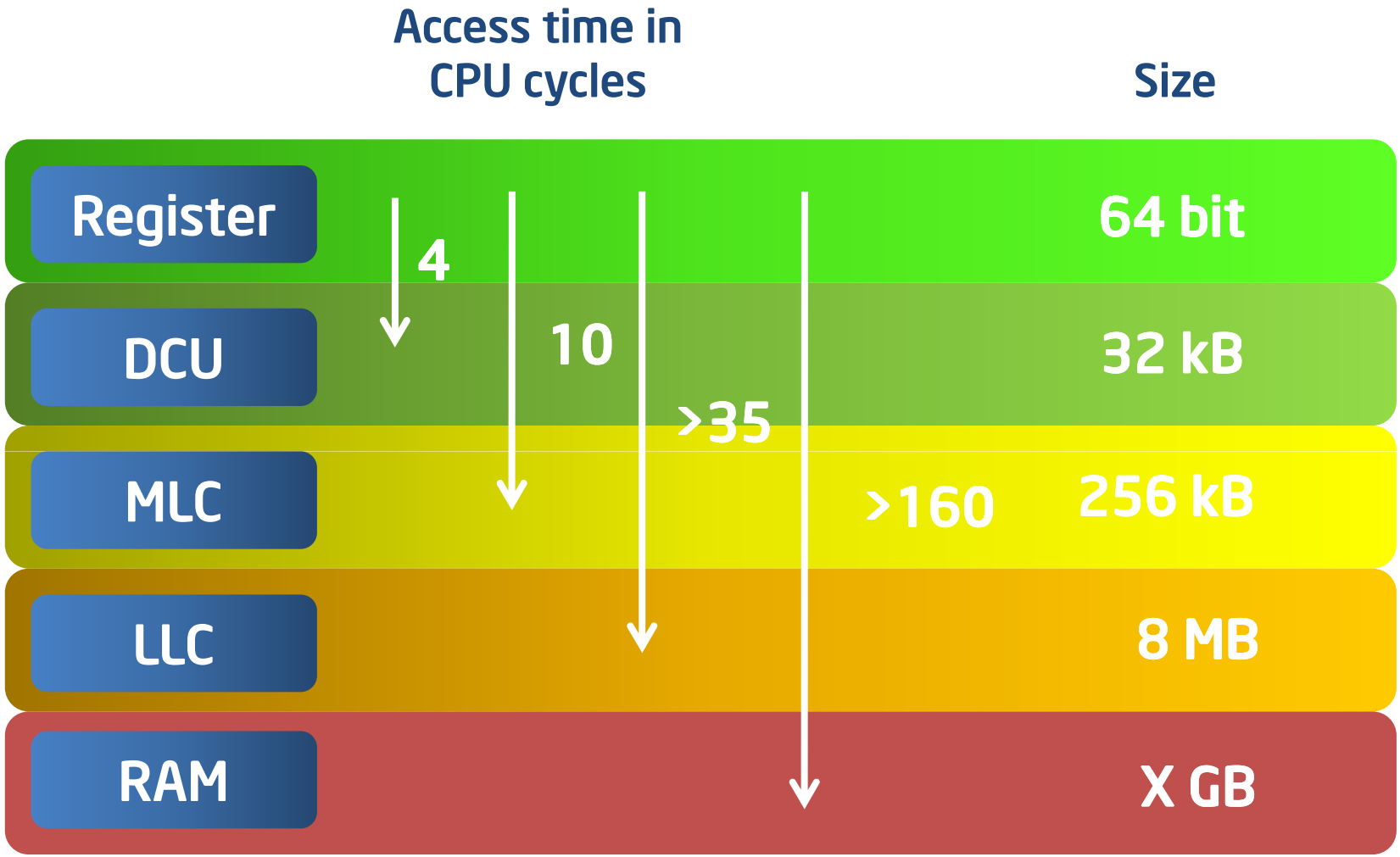
Multi-Core Caches

- In multi-core chip designs the cache also needs to serve as a communication layer



- This ensures fast data exchange, but introduces more complications

Cache sizes and access time on Nehalem



Data Prefetching

- The Nehalem architecture features a number of data prefetchers that attempt to have the data readily in cache when the user needs it
- Most importantly:
  - Adjacent cache lines are prefetched
  - Equal strides are prefetched

Cache Issues

Compulsory Misses

Let's identify a number of standard cache issues

First of all, an initial access to a data element can never be cached will lead to a cache miss

Cache Issues
Compulsory Misses

Compulsory Cache Miss

- Issue: An initial access to a data element can never be cached will lead to a cache miss
- Reason: Program touches data elements only once
- Remedy: Data prefetching, either by software (intrinsics) or HW eases the pain. Organize data in a way that will allow the prefetcher to work efficiently. Don't access data randomly.

Cache Issues
Capacity Misses

Capacity Cache Miss

- Issue: a data element is evicted before it can be reused
- Reason: data set too large for the cache
- Remedy: Reduce working set size. Implement cache blocking. Organize data linearly to help the prefetcher.

Cache Issues
Conflict Misses

Conflict Cache Miss

- Issue: Data elements are evicted from the cache although the cache capacity is not evicted.
- Reason: Mapping strategy or access pattern conflicts with cache organization, e.g. the associativity (remember Set number = Memory index % Number of sets).
- Remedy: Change the data access pattern. Change the memory allignement

Cache Issues
Coherency Misses

Coherency Cache Miss

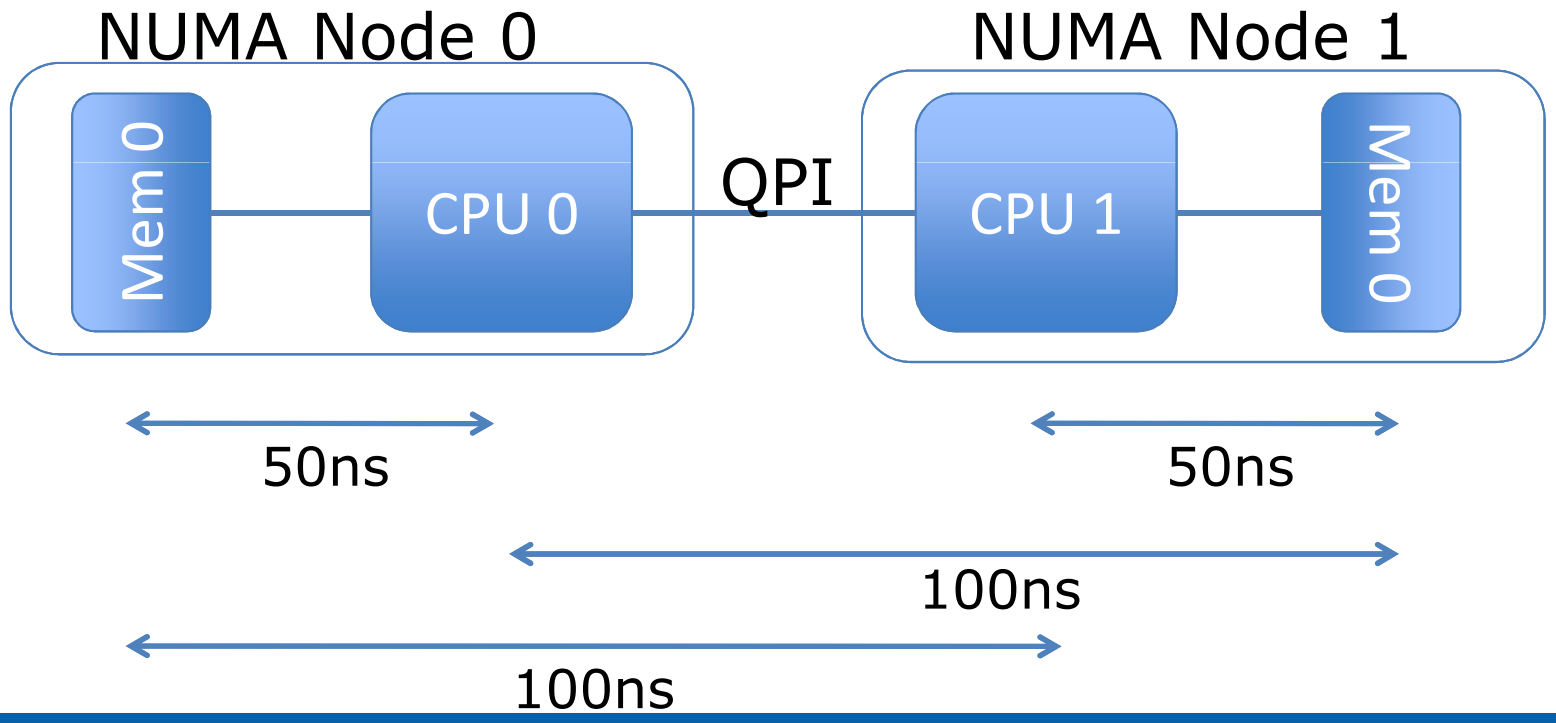
- Issue: Coherency misses only appear in multi-core and multi-processor systems. When a data element is present in multiple caches a modification of the data in one cache will invalidate the data in the other cache.
- Reason: Two many threads working on the same data, e.g. a sum or reduction
- Remedy: store data thread-localy (separate valariable) as long as possible

Cache Misses
Summary

- **Compulsory** – The very first access to a block cannot be in cache
- **Capacity** – If the number of blocks loaded is larger than the number of blocks that can be stored in the cache.
- **Conflict** – if a mapping strategy evicts blocks in a set that will be retrieved in a following step, e.g. if data maps only to one set in a set associative cache and the set capacity is exceeded.
- **Coherency** – if to concurrent processes on different cores evict mutually cache lines from the respective core, e.g. by writing to the same value or a value in the same cache line

NUMA Systems

- In multi-processor systems with integtrated memory controllers an additional complication appears:



NUMA Systems

- Generally memory is placed at a particular NUMA node at “first touch”
- the allocation (via new or malloc) doesn’t result in the creation of a memory page
- Only the first write of the memory area under consideration will result in the placement of one particular NUMA node

Thread-local or thread-private memory allocation is important here. Use processor pinning!

Performance Monitoring

The Performance Monitoring Unit (PMU)

Ways to measure performance

- time – the ultimate measure!
- gprof – break down execution time on a functional level. Careful, changes execution time!
- counter monitoring – hardware supported measurement that allows to determine which hardware component or functional unit is under stress or starves. Doesn't change exec time!

Performance Counter Monitoring

- Performance counter monitoring is a HW supported method count events appearing in the hardware
- Events are situations the HW designers allow us to see. E.g. the CPU could experience "I tried to load data, but it was not in the last level cache" (aka a LLC cache miss). This is an event that can be measured.

Command line perf monitoring

Architectural perf events

Table A-1. Architectural Performance Events

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
3CH	UnHalted Core Cycles	00H	Unhalted core cycles	
3CH	UnHalted Reference Cycles	01H	Unhalted reference cycles	Measures bus cycle[1]
C0H	Instruction Retired	00H	Instruction retired	
2EH	LLC Reference	4FH	LL cache references	
2EH	LLC Misses	41H	LL cache misses	
C4H	Branch Instruction Retired	00H	Branch instruction retired	
C5H	Branch Misses Retired	00H	Mispredicted Branch Instruction retired	

There are another ~1000 non-architectural performance events!

Performance Counter Monitoring

- There are
  - 4 freely programmable
  - 3 fix function

  performance counter registers on current Intel CPUs
- There are 4 configuration registers, that let us specify which events are counted in the programmable counters
- There is 1 configuration register that lets us enable and disable performance monitoring.

Performance Counter Monitoring

Just in case: What is a register?

- A register is a series of latches (HW bits) with 64 bit width (some might differ, e.g. SSE) the CPU can access very quickly

- A Model Specific Register (MSR) is a register that is not guaranteed to be there in the next CPU!

Performance Counter Monitoring

How can I influence/change/write/read MSRs?

- MSRs can only be written in Ring 0 (supervisor mode). You can read MSRs as user.

You need to be

- root/sudoer to write

or

- a driver that does the access for you

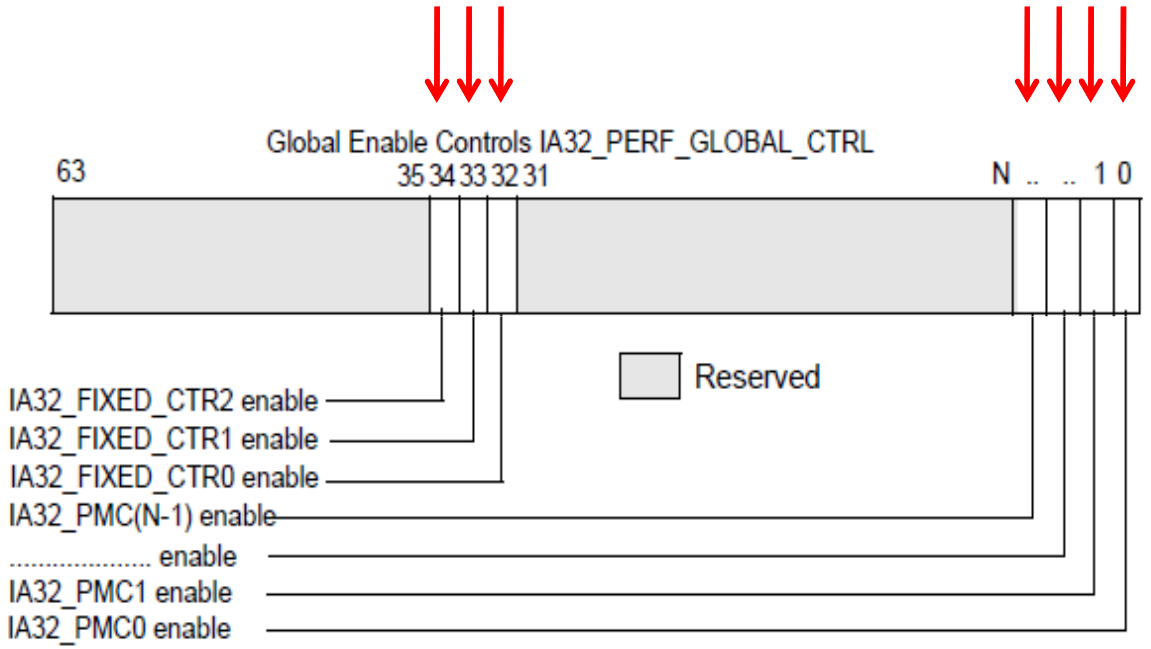
Performance Counter Monitoring

Software to read and write MSRs (linux)

- msr-tools package: rdmsr, wrmsr, msr.ko
- perf (linux system tool)
- Intel Vtune, Amplifier XE, PTU (http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/)
- Oprofile (http://oprofile.sourceforge.net/news/)
- Likwid (http://code.google.com/p/likwid/)

PMU MSRs
Control Registers

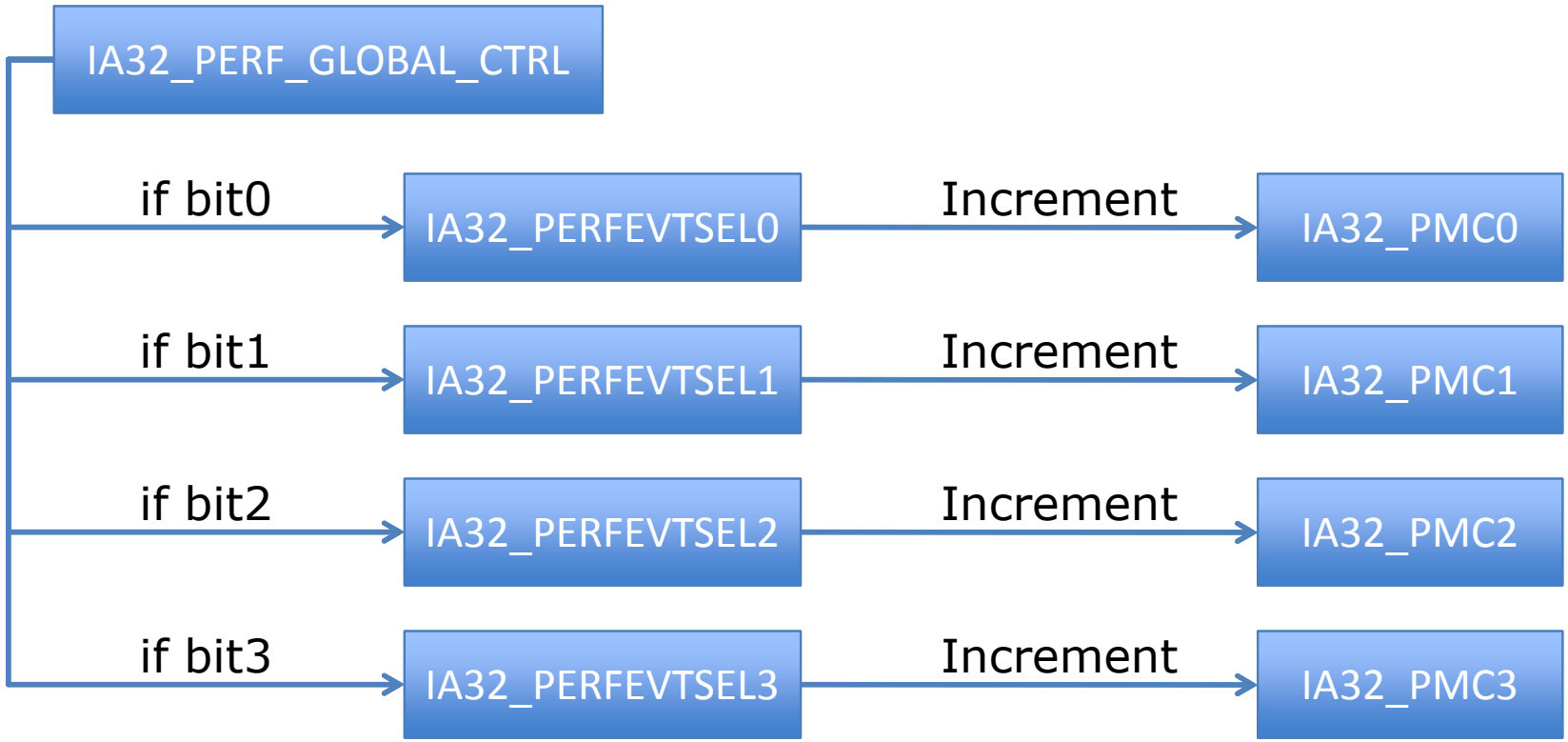
- IA32_PERF_GLOBAL_CTRL (0x38F/911)



wrmsr 911 –d 30064771087

PMU MSRs

General purpose performance counters



IA32_PERFEVTSELx specifies the number of the event to be obeserved (and a number of modifiers)

PMU MSRs

Selecting events

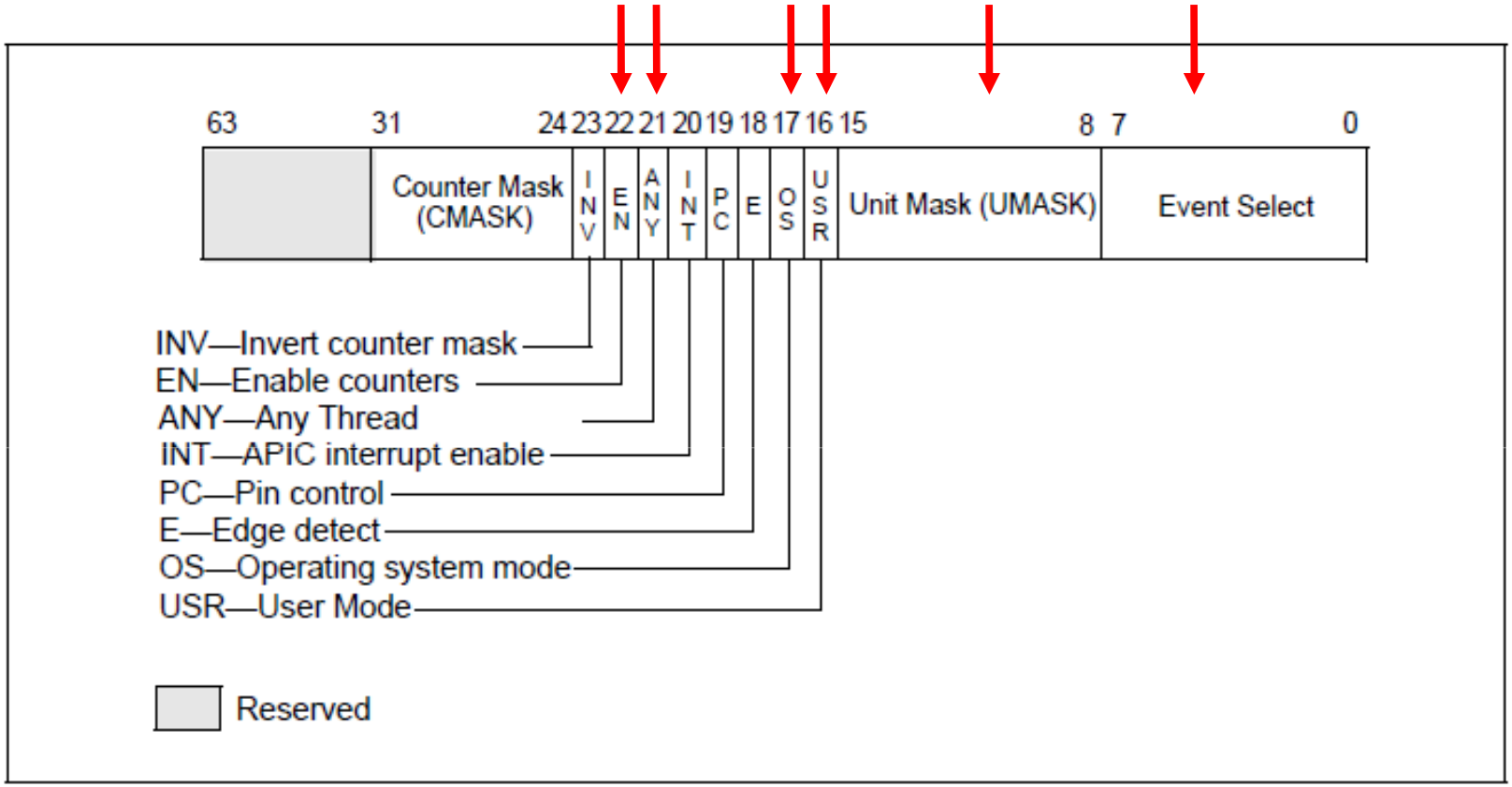


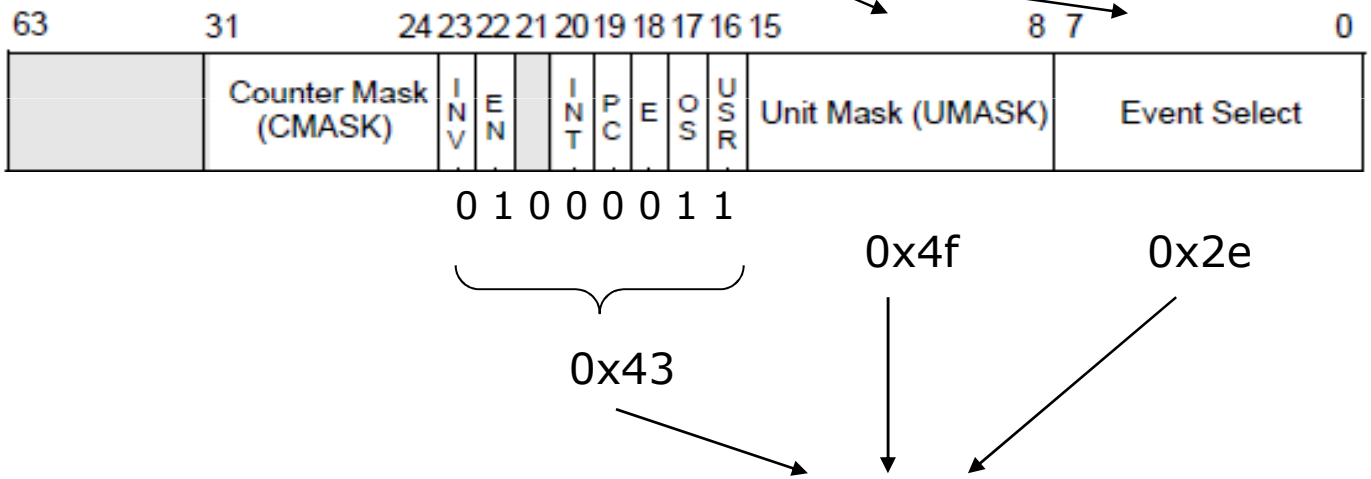
Figure 30-6. Layout of IA32_PERFEVTSELx MSRs Supporting Architectural Performance Monitoring Version 3

PMU MSRs

Selecting general purpose events

Intel® 64 and IA-32 Architectures Software Developer's Manual
Volume 3B: System Programming Guide, Part 2

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
2EH	LLC Reference	4FH	LL cache references	



Set IA32_PERFEVTSELx to this value

PMU MSRs

Fixed function performance counters

Additionally to the 4 freely programmable performance counters Nehalem (and later CPUs) offer three fixed function performance counters that will observe

- Instructions Retired
- CPU Cycles
- Reference CPU Cycles

PMU MSRs

Fixed function performance counters

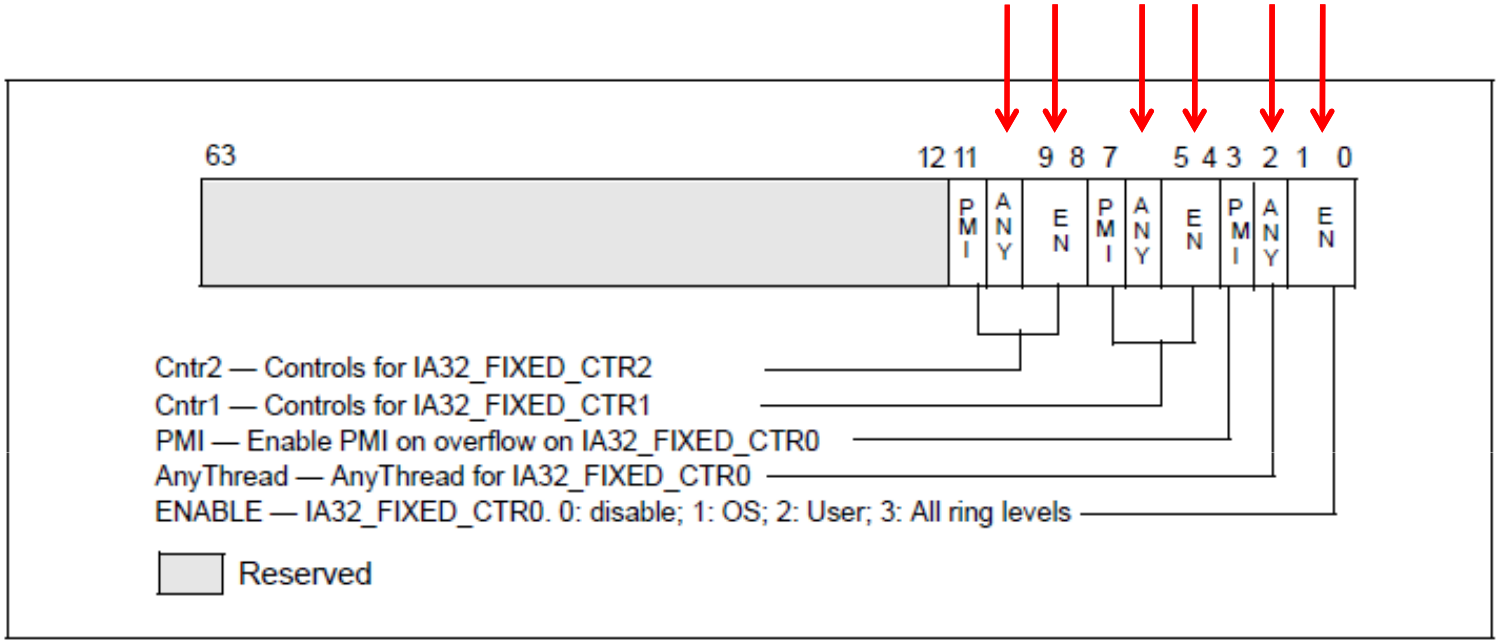


Figure 30-7. Layout of IA32_FIXED_CTR_CTRL MSR Supporting Architectural Performance Monitoring Version 3

PMU MSRs

Fixed function performance counters

- Event counts can now be seen in
- IA32_FIXED_CTR0: Instructions Retired
- IA32_FIXED_CTR1: Unhalted CPU Cycles
- IA32_FIXED_CTR2: Reference CPU Cycles

309H	777	IA32_FIXED_CTR0 (MSR_PERF_FIXED_CTR0)	Fixed-Function Performance Counter 0 (R/W): Counts Instr_Retired.Any	If CPUID.0AH: EDX[4:0] > 0
30AH	778	IA32_FIXED_CTR1 (MSR_PERF_FIXED_CTR1)	Fixed-Function Performance Counter 1 0 (R/W): Counts CPU_CLK_Unhalted.Core	If CPUID.0AH: EDX[4:0] > 1
30BH	779	IA32_FIXED_CTR2 (MSR_PERF_FIXED_CTR2)	Fixed-Function Performance Counter 0 0 (R/W): Counts CPU_CLK_Unhalted.Ref	If CPUID.0AH: EDX[4:0] > 2

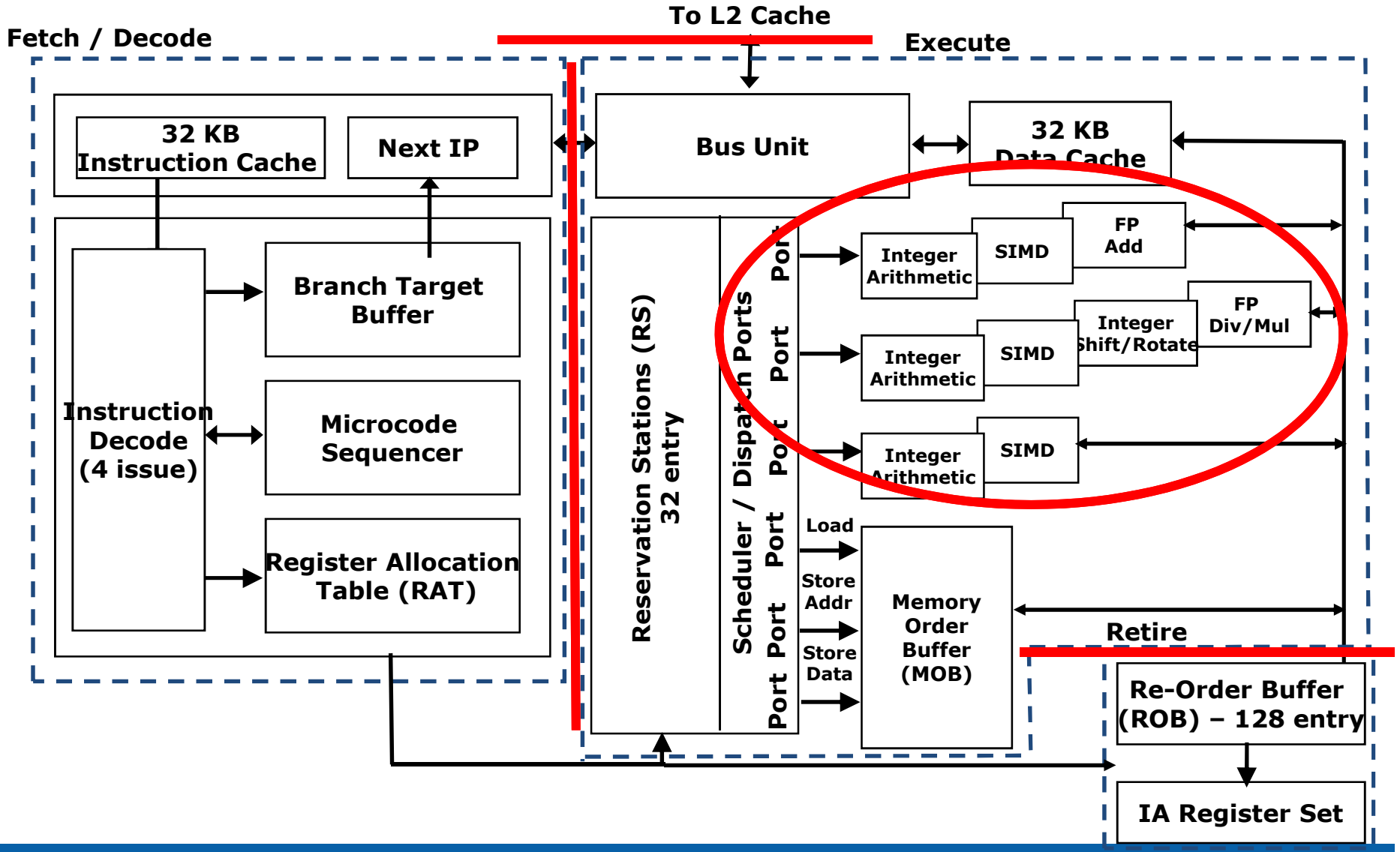
PMU MSRs
Summary

- Intel CPUs can monitor particular events during execution
- 4 freely programmable counters and 3 fixed function counters measure the occurrence of the given events at the same time
- In order observe event counts
  - Enable event monitoring in the control register
  - Program the event into the event select register
  - Reset/Read the count from the counter register
- READ: 253669 - Intel 64 and IA-32 Architectures Software Developers Manual Volume 3B System Programming Guide Part 2

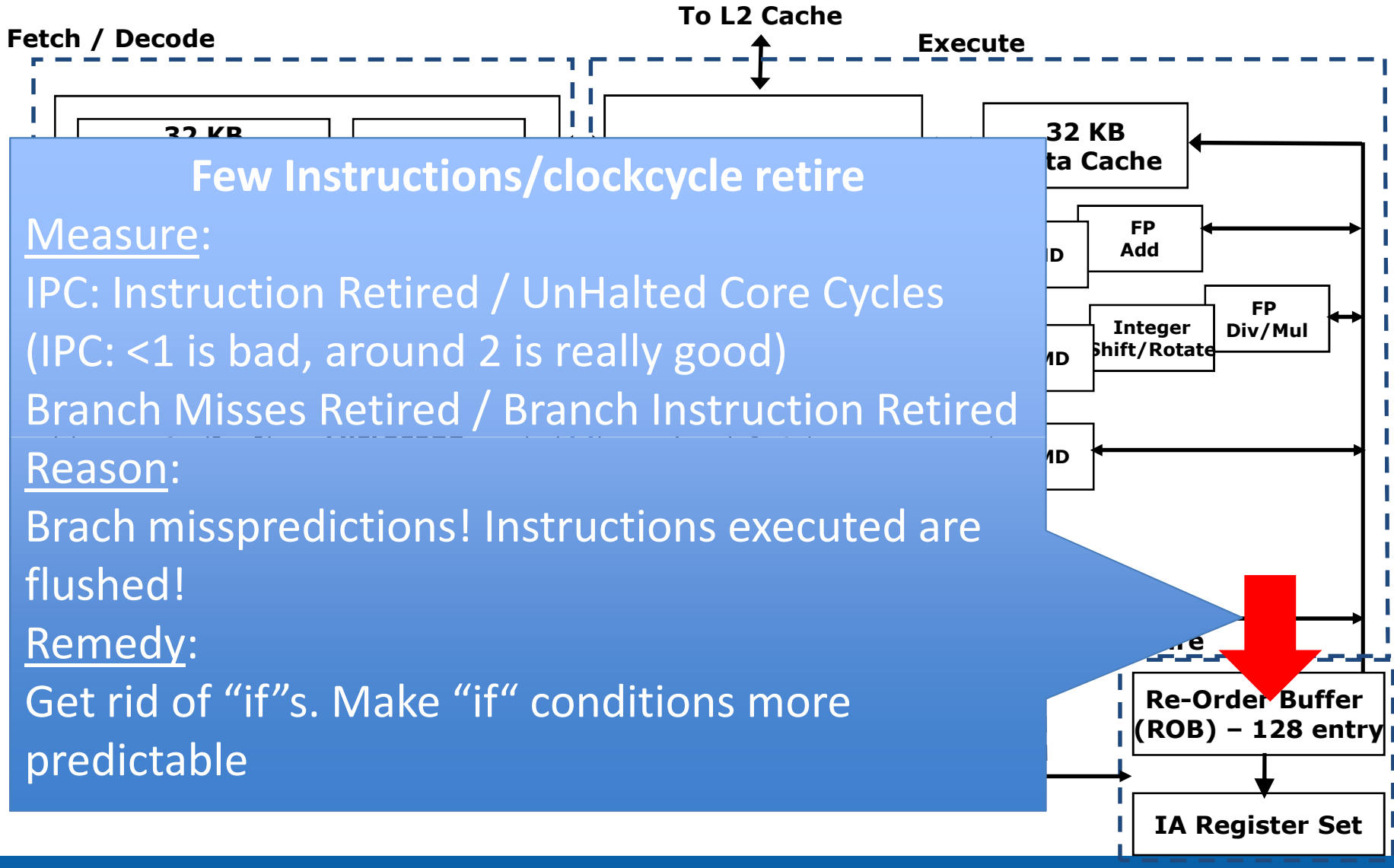
Performance Monitoring

Events to measure – a minimal set

Core i7 Block Diagram



Core i7 Block Diagram



Few Instructions/clockcycle retire

Measure:

IPC: Instruction Retired / UnHalted Core Cycles

(IPC: <1 is bad, around 2 is really good)

Branch Misses Retired / Branch Instruction Retired

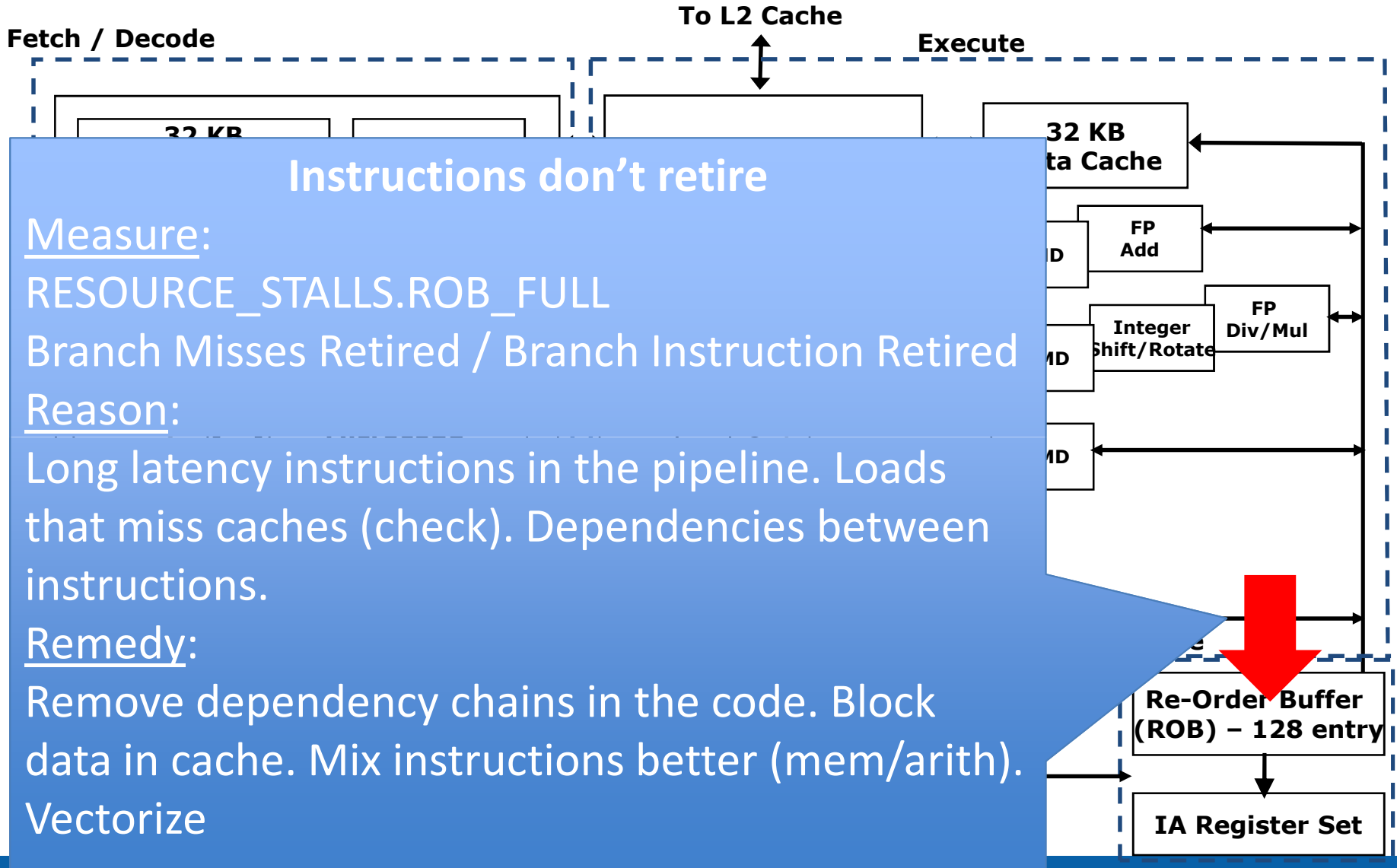
Reason:

Brach misspredictions! Instructions executed are flushed!

Remedy:

Get rid of “if”s. Make “if“ conditions more predictable

Core i7 Block Diagram



Instructions don't retire

Measure:

RESOURCE_STALLS.ROB_FULL

Branch Misses Retired / Branch Instruction Retired

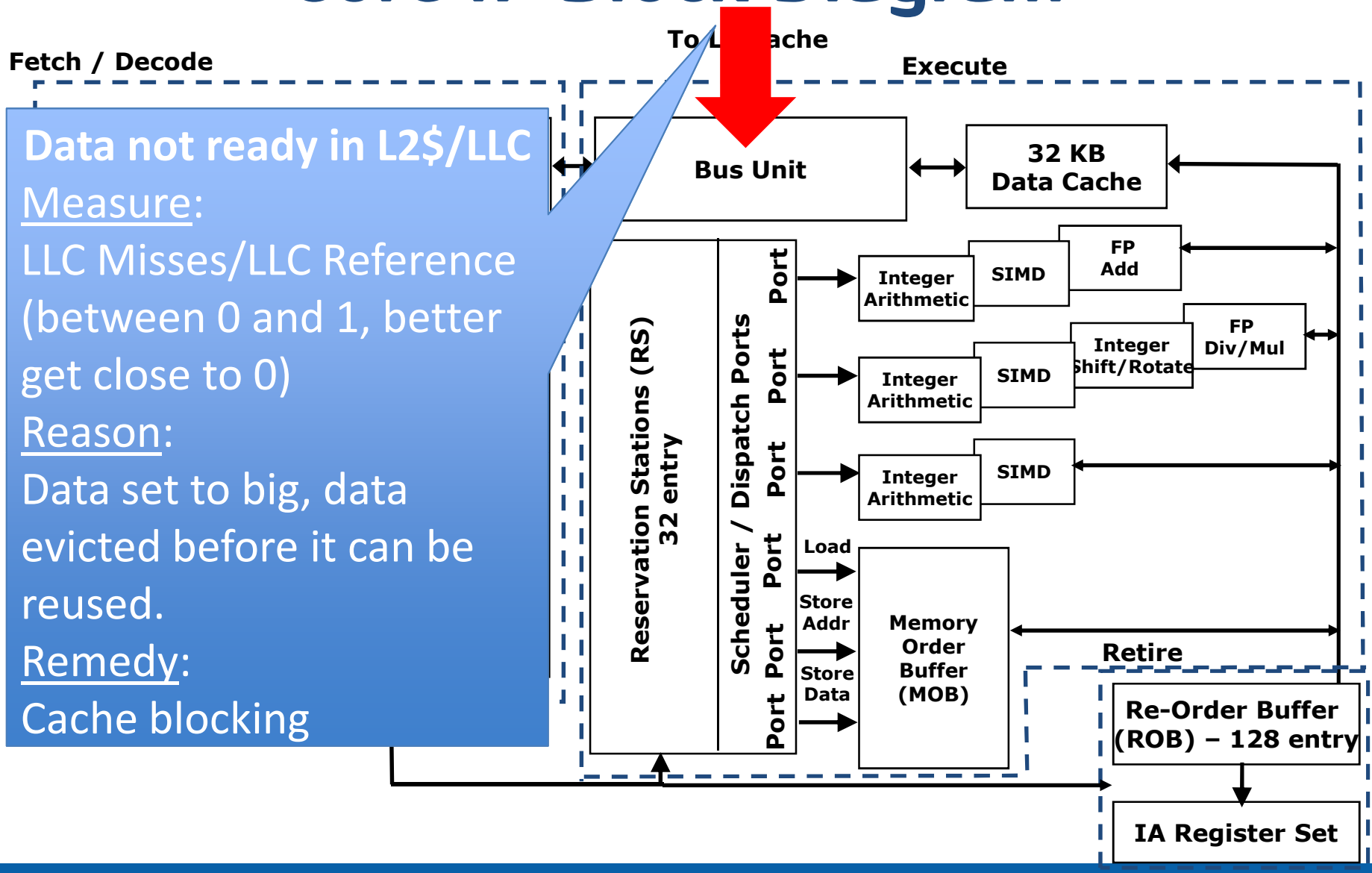
Reason:

Long latency instructions in the pipeline. Loads that miss caches (check). Dependencies between instructions.

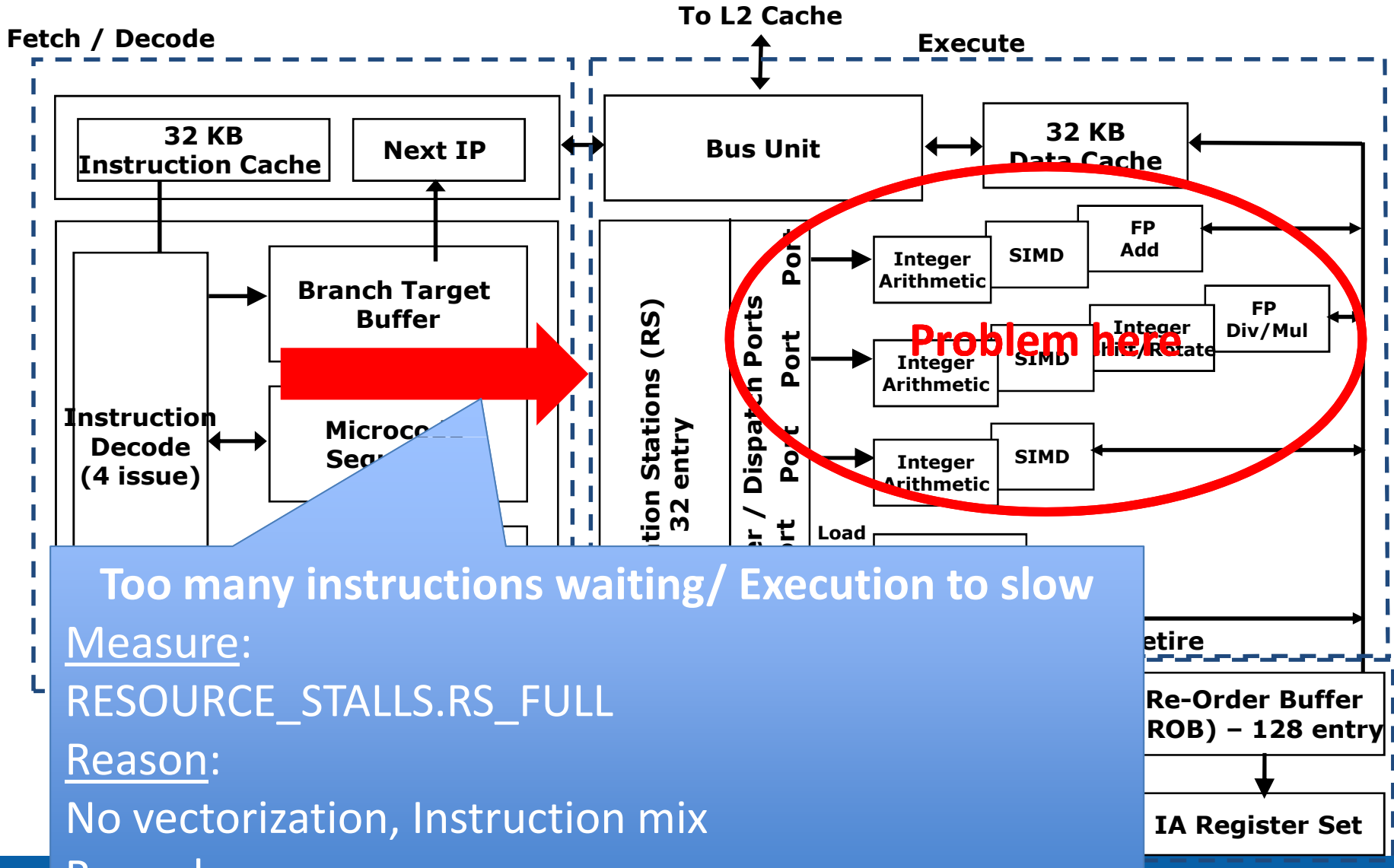
Remedy:

Remove dependency chains in the code. Block data in cache. Mix instructions better (mem/arith). Vectorize

Core i7 Block Diagram



Core i7 Block Diagram



Others to measure

How do we actually know what fraction of the peak performance we are running at?

Event Num.	Umask Value	Event Mask Mnemonic	Description	Comment
C7H	01H	SSEX_UOPS_RETIRED.PACKED_SINGLE	Counts SIMD packed single-precision floating point Uops retired.	
C7H	02H	SSEX_UOPS_RETIRED.SCALAR_SINGLE	Counts SIMD calar single-precision floating point Uops retired.	
C7H	04H	SSEX_UOPS_RETIRED.PACKED_DOUBLE	Counts SIMD packed double-precision floating point Uops retired.	
C7H	08H	SSEX_UOPS_RETIRED.SCALAR_DOUBLE	Counts SIMD scalar double-precision floating point Uops retired.	

This is specific for Nehalem/Westmere (X55xx/X56xx). Can be done similarly on Sandy Bridge.

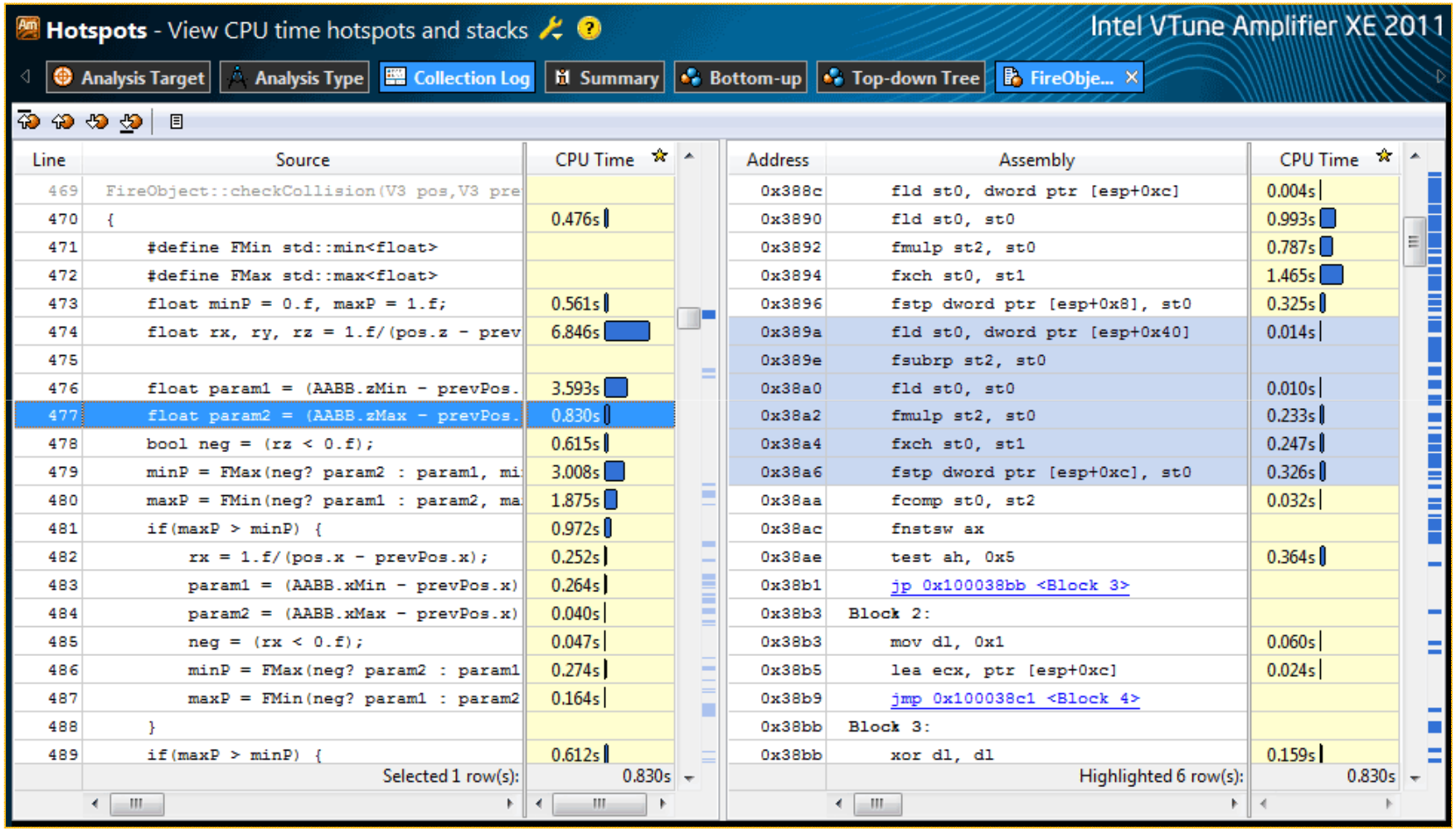
Bash script to measure FLOPS

```
# enable all general and fixed counters - can't hurt
IA32_PERF_GLOBAL_CTRL_VAL=`echo 2^34+2^33+2^32+2^3+2^2+2^1+1|bc -l`

# check presence of the msr module
modprobe msr

# see how many processors we have
procnum=`grep CPU /proc/cpuinfo | wc -l`
let procnum=$procnum-1

sacc=0;
dacc=0;

# for every processor (core)
for i in `seq 0 $procnum `; do
# enable counters
   wrmsr -p$i $IA32_PERF_GLOBAL_CTRL $IA32_PERF_GLOBAL_CTRL_VAL
# single precision packed SSE uops
   wrmsr -p$i $IA32_PERFEVTSEL0  0x4101c7
# single precision scalar SSE uops
   wrmsr -p$i $IA32_PERFEVTSEL1  0x4102c7
# double precision packed SSE uops
   wrmsr -p$i $IA32_PERFEVTSEL2  0x4104c7
# double precision scalar SSE uops
   wrmsr -p$i $IA32_PERFEVTSEL3  0x4108c7
# set all counters to to zero
   wrmsr -p$i  $IA32_PMC0 0
   wrmsr -p$i  $IA32_PMC1 0
   wrmsr -p$i  $IA32_PMC2 0
   wrmsr -p$i  $IA32_PMC3 0
done
```

```
while [ 1 ]; do
# for all processors
   for i in `seq 0 $procnum `; do
# ... read out the counters
         spflop=`rdmsr -p$i -d $IA32_PMC0`;
         ssflop=`rdmsr -p$i -d $IA32_PMC1`;
         dpflop=`rdmsr -p$i -d $IA32_PMC2`;
         dsflop=`rdmsr -p$i -d $IA32_PMC3`;
# ... calculated MFLOP/s with bc
         sflops=`echo   "scale=2; ( 4 * $spflop + $ssflop ) /1000000.0 /
         $SAMPLE_TIME" | bc -l`
         dflops=`echo   "scale=2; ( 2 * $dpflop + $dsflop ) /1000000.0 /
         $SAMPLE_TIME" | bc -l`
         sacc=`echo   "scale=2; $sacc+$sflops" | bc -l`
         dacc=`echo   "scale=2; $dacc+$dflops" | bc -l`

# ... and reset the counters again
         wrmsr -p$i  $IA32_PMC0 0 ;
         wrmsr -p$i  $IA32_PMC1 0 ;
         wrmsr -p$i  $IA32_PMC2 0 ;
         wrmsr -p$i  $IA32_PMC3 0 ;
#        echo proc $i $sflops $dflops
   done;
# ... print the results
   clear ;
#    echo $spflop $ssflop $dpflop $dsflop
   echo "SP MFLOPS : "$sacc
   echo "DP MFLOPS : "$dacc
   sacc=0;
   dacc=0;
   sleep $SAMPLE_TIME;
done
```

VTune™ Amplifier XE

GUI Layout



Summary

- Performance counters allow you to measure almost any aspect of an Intel CPU
- IPC, Cache Miss Ratio and Branch Miss Ratio are the most important indicators (architectural events)
- Most often the problem is found in the execution, particularly in the vectorization!

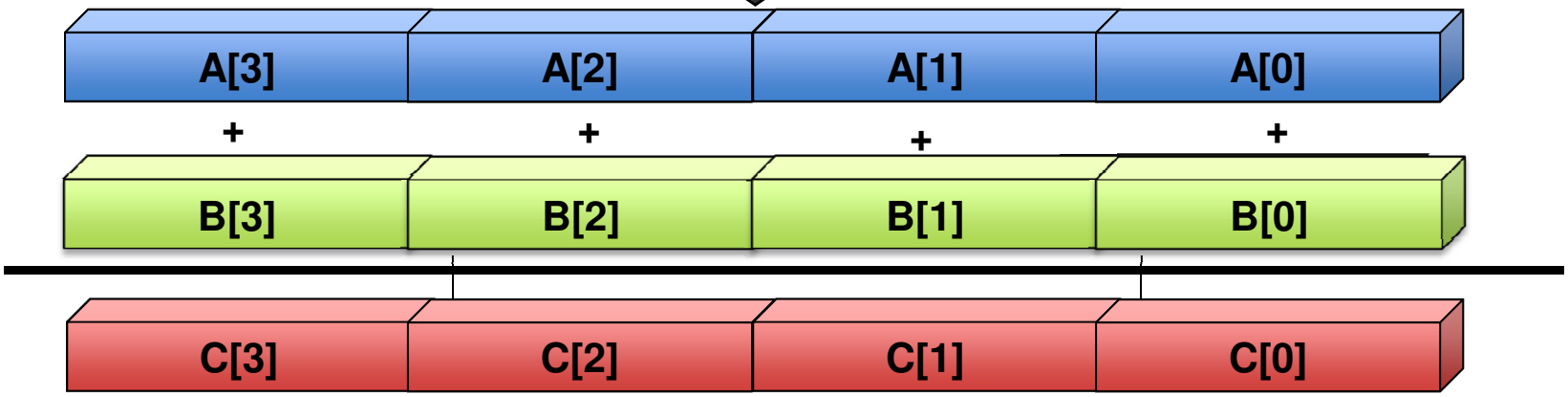
Compiler Vectorization

Vectorization

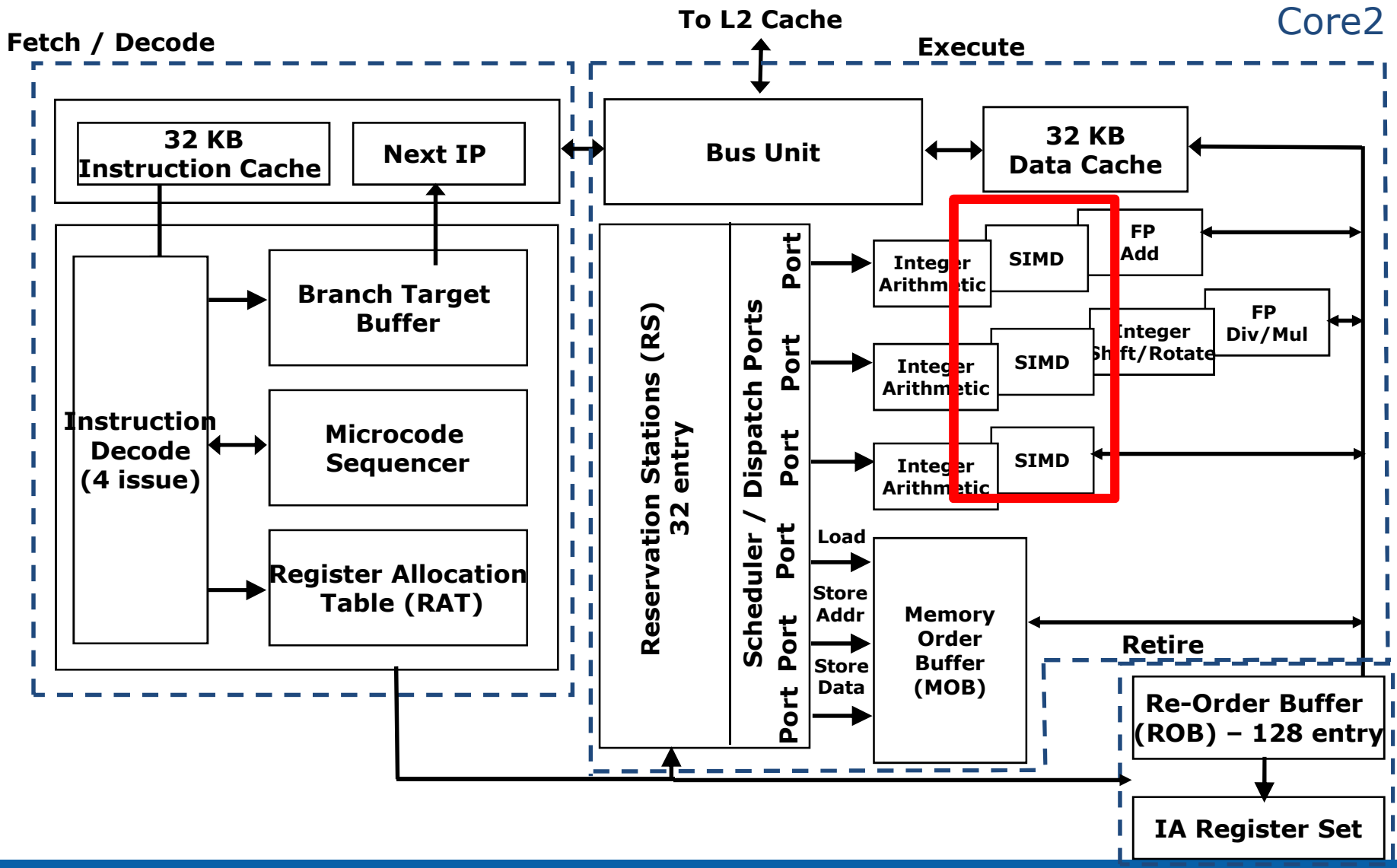
Transforming sequential code to exploit the vector (SIMD, SSE) processing capabilities

- Manually by explicit source code modification
- Automatically by tools like a compiler

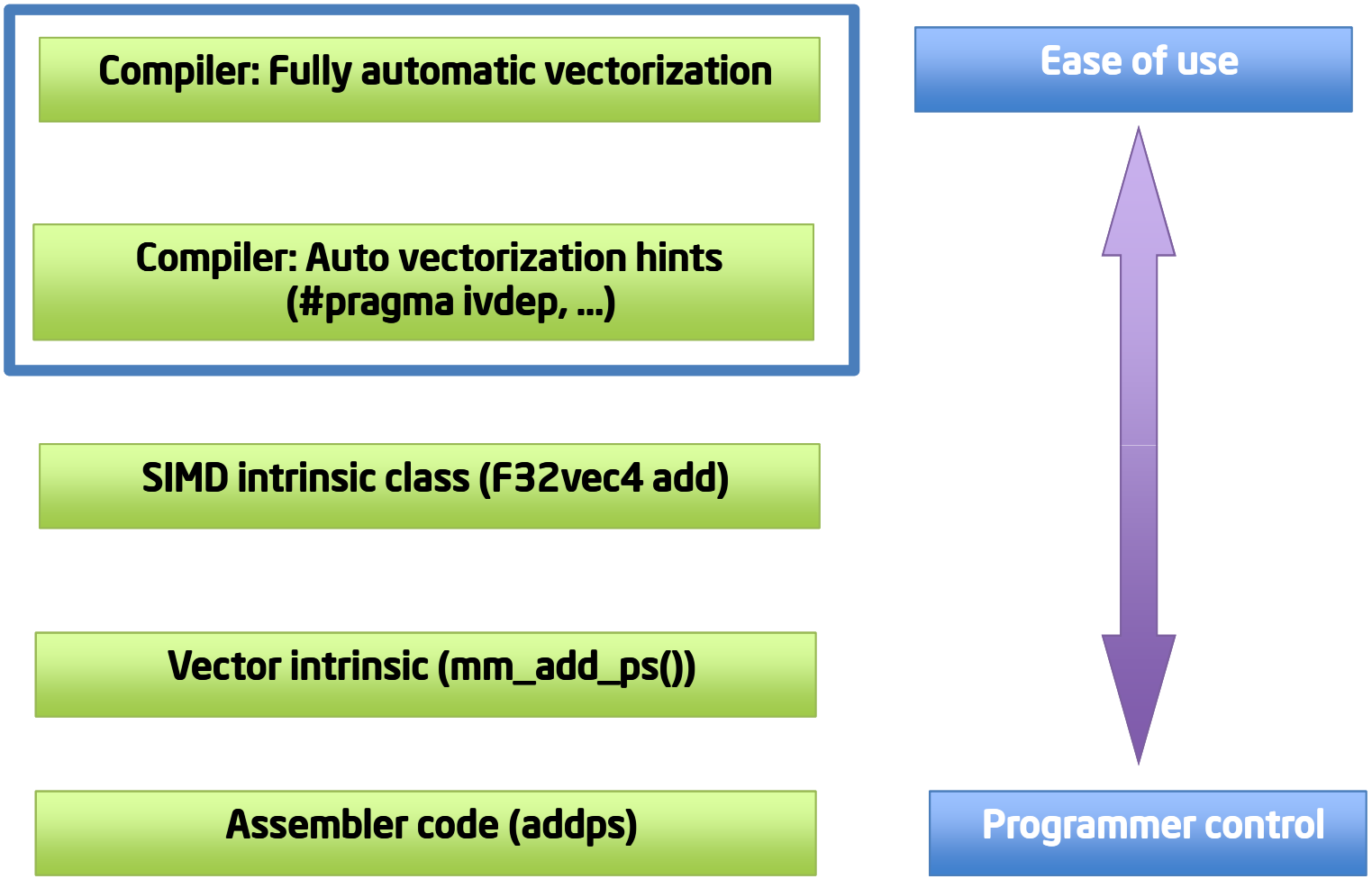
```
for (i=0;i<MAX;i++)
   c[i]=a[i]+b[i];
```



Core i7 Block Diagram



Many Ways for SSE Vectorization



Vectorization

- We don't want to deal with code changes here. Programming SIMD intrinsics or assembly is a full day course.
- Let's focus on what can be achieved with the compiler without code changes or only with hints in the code (like pragmas)

Compiler Based Vectorization Extension Specification

Feature	Extension
Intel® Streaming SIMD Extensions 2 (Intel® SSE2) as available in initial Pentium® 4 or compatible non-Intel processors	SSE2
Intel® Streaming SIMD Extensions 3 (Intel® SSE3) as available in Pentium® 4 or compatible non-Intel processors	SSE3
Intel® Supplemental Streaming SIMD Extensions 3 (Intel® SSSE3) as available in Intel® Core™2 Duo processors	SSSE3
Intel® SSE4.1 as first introduced in Intel® 45nm Hi-K next generation Intel Core™ micro-architecture	SSE4.1
Intel® SSE4.2 Accelerated String and Text Processing instructions supported first by by Intel® Core™ i7 processors	SSE4.2
Extensions offered by Intel® ATOM™ processor : Intel® SSSE3 (!!) and MOVBE instruction	SSE3_ATOM
Intel® Advanced Vector Extensions (Intel® AVX) as available in 2nd generation Intel Core processor family	AVX

Basic Vectorization – Switches [1]

-x<extension>

- Targeting Intel® processors - specific optimizations for Intel® processors
- Compiler will try to make use of all instruction set extensions up to and including <extension>; for Intel® processors only !
- Processor-check added to main-program
- Application will not start (will display message), in case feature is not available

-m<extension>

- No Intel processor check
- Does not perform Intel-specific optimizations
- Application is optimized for and will run on both Intel and non-Intel processors
- Missing check can cause application to fail in case extension not available

-ax<extension>

- Dual-code paths – a 'generic' and 'optimized' path
- 'processor-specific' path for Intel® processors defined by <extension>
- 'default' code path defaults to -msse2 (Windows: /arch:SSE2)
  - The 'default' code path can be modified by -m or -x (/Qx or /arch) switches

Basic Vectorization – Switches [2]

The default now is -msse2

- Activated implicitly for -O2 or higher
- Implies the need for a target processor with Intel® SSE2
- Use -mia32 (Windows /arch:IA32) in case target processor misses SSE2 (Intel® Pentium™ 3 processor for example)

Special switch -xHost

- Compiler checks host processor and makes use of 'latest' instruction set extension available
- Avoid for builds being executed on multiple, unknown platforms

Some support for combination of -x<ext1> and -ax<ext2> switches

- Can result in more than 2 code paths
- Use ext1 = ia32 in case 'generic' code path should support too very early processors not supporting SSE2 (e.g. Intel® Pentium™ 3)

Disable vectorization

- Globally via switch: `-no-vec`
- For a single loop: directive `#pragma novector`
  - Disabling vectorization here means not using packed SSE/AVX instructions. The compiler still might make use of the corresponding instruction set extensions

Enforcing vectorization for a loop - overwriting the compiler heuristics :

```
#pragma vector always
```

  - will enforce vectorization even if the compiler thinks it is not profitable to do so (e.g due to non-unit strides or alignment issues)
  - Will not enforce vectorization if the compiler fails to recognize this as a semantically correct transformation
  - Using directive `#pragma vector always assert` will print error message in case the loop cannot be vectorized and will abort compilation

Validating Vectorization Success

- Assembler code inspection
  - Assembler listing: {L&M}: -S and {W}: /Fa
  - Most reliable way and gives all details of course
  - Check for scalar or packed instructions
    - Assembler listing contains source line numbers mapping generated code to loops in source code
- Optimization report of "High-Performance-Optimizer" (HPO) phase
  - `-opt-report<N>` `-opt-report-phasehpo`
    - N=1,2,3 specifies level of detail, N=2 is default
  - We will come back to the opt-report switch later again
- Vectorization report: -vec-report<N>
- Dynamically counting the number of executed packed SSE instructions using tools like Intel® VTune Amplifier™ profiler
  - E.g. using performance monitoring event `FP_COMP_OPS_EXE.SSE_FP_PACKED` on Intel® Core™ i7 processors

Vectorization Report

- Provides details on vectorization success & failure

  -vec-report<n> ,n=0,1,2,3,4,5

```
35:     subroutine fd( y )
36:     integer :: i
37:     real, dimension(10), intent(inout) :: y
38:     do i=2,10
39:        y(i) = y(i-1) + 1
40:     end do
41:     end subroutine fd
```

```
novec.f90(38): (col. 3) remark: loop was not vectorized: existence of
vector dependence.
novec.f90(39): (col. 5) remark: vector dependence: proven FLOW
dependence between y line 39, and y line 39.
novec.f90(38:3-38:3):VEC:MAIN_:  loop was not vectorized: existence of
vector dependence
```

Compiler Vectorization

Why vectorization fails – and workarounds …

Why Vectorization Fails …

- Most frequent reason: Dependence
  - Simplified: Loop iterations must be independent
- Many other potential reasons
  - Alignment
  - Function calls in loop block
  - Complex control flow / conditional branches
  - Loop not "countable"
    - E.g. upper bound no run time constant
  - Not inner loop
    - Outer loop of nest cannot be vectorized
  - Mixed data types (many cases now handled successfully)
  - Non-unit stride between elements
  - Loop body too complex- register pressure
  - Vectorization seems inefficient
  - Many more … but less likely too occur

Dealing with Dependencies #1

Hints to the Compiler

- Many dependencies assumed by compiler are false dependencies caused by unresolved memory disambiguation
  - The compiler has to be conservative and has to assume the worst case regarding "aliasing"

```
// Sample: Without additional information (like inter-procedural
// knowledge) compiler has to assume 'a' and 'b' to alias
void scale(int *a, int *b)
{
    for (int i=0; i<10000; i++) b[i] = z*a[i];
}
```

  - Many directives, switches and attributes to pass "disambiguation hints" to compiler
    - Programming language and operating system specific
    - Use with care: The compiler might generate incorrect code in case the hints are not fulfilled !

Disambiguation Hints [C/C++]

A few Selected Directives and Switches

IVDEP directive (#pragma ivdep)

- "Ignore Vector Dependencies" - compiler will ignore assumed but not proven dependencies for loop following directive
- In case used together with switch –ivdep-parallel (/Qivdep-parallel), only loop-carried dependencies are ignored

Assume no aliasing at all

-fno-alias

Assume ISO C Standard aliasing rules

-ansi-alias

- A pointer can be de-referenced only to an object of the same type or compatible type

No aliasing for function arguments

-fargument-noalias

- For each given function, the arguments of this function don't refer to a common memory object

Dealing with Dependencies #2

- Dynamic data dependency analysis
  - The compiler can (!) use run-time checks to test for aliasing
    - E.g. Array A[La:Ua], B[Lb:Ub] overlap ⇔ La<Ub && Lb<Ua
  - The outcome of test is used to execute a vectorized or scalar version of the loop (“Loop Versioning”)
  - The heuristic of compiler implements a balance between overhead of testing and performance gains
    - E.g. for an assignment

      $$A[..] = B_1[\ldots] + B_2[\ldots] + \ldots + B_N[\ldots]$$

      the versioning might be done for N=2 but not for N=5
    - Use switch **-opt-multi-version-aggressive** to change heuristic
- Inter-procedural Dependency Analysis
  - Can improve dependence analysis accuracy considerably
  - Activated by “inter-procedural optimization”: **-ipo**
    - For optimization level 2, 3, file-local IPO is on by default
  - Definitions & allocation of function arguments might become visible
  - In case loop body has function call, references in the called function to global variables and actual arguments can be analyzed

Alignment

- In general, the memory accesses in packed SSE instructions require the data to be aligned to 16 byte boundaries
- For packed AVX instructions, it has to be 32 byte alignement
- Unaligned data can be moved to XMM(YMM) registers using "unaligned load/store" instructions
  - However these instruction are very slow except for SSE memory operations on Intel® Core™ i7 processors or processors based on future Sandy Bridge architecture
- The compiler splits expensive unaligned memory operations into 2 partial loads/stores (e.g. two 64byte loads for one 128byte unaligned load) since this is faster – but still much more expensive than the aligned moves
- The compiler can use 'versioning' in case alignment is unclear
  - A run time check tests for alignment controls execution of a fast version of the loop assuming required alignment or a slower one assuming unaligned data

Alignment Hints to Compiler [C/C++]

- Aligned heap memory allocation by intrinsic / library call

```
void* _mm_malloc (int size, int base)
```

- Directive to assert to compiler, that aligned memory operations can be used for all data accesses in loop following directive

```
#pragma vector aligned | unaligned
```

  - Use with care: The assertion must be satisfied not only by start addresses of all arrays used in loop but for all (!!) data accesses

- Align attribute for variable declarations

```
__declspec(align(base)) <array_decl>
<array_decl> __attribute__((aligned(base)))
```

- Assertion to compiler that in the loop following the start address of an array can be assumed to be aligned
  - A language extension (not a directive) for C/C++

```
__assume_aligned(<variable>,base)
```

Alignment can be tricky …

```
void matvec(double a[][COLWIDTH], double b[], double x[])
{
  int i, j;
  for (i = 0; i < size1; i++) {
    b[i] = 0;
    #pragma vector aligned
    for (j = 0;j < size2; j=j++)
      b[i] += a[i][j] * x[j];
  }
}
```

- Let us assume, a, b, c would be declared 16-byte aligned in calling routine
- Would this be correct when compiled for SSE2 ?
- Depends on COLWIDTH
  - In case it is even : All ok !
  - In case it is odd: The generated, vectorized code would fail by alignment error !
- Using `__assume_aligned(a,n)` is legal since this refers to the start address only. It wouldn't change much for the vectorization however

Unsupported Loop Structure

- Unsupported loop structure frequently means, the compiler can't construct a runtime expression for the trip-count
  - E.g. a while-loop where the number of iterations cannot be determined at (run-time) start of loop
  - Upper/lower bound of a for-loop cannot be a determined to be loop-invariant
- Frequently this can fixed by minor modifications:

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
  for (int i=0; I < x->bound; i++)
    a[i] = 0;
}
```

```
struct _x { int d; int bound;};

doit1(int *a, struct _x *x)
{
  int local_ub = x->bound;
  for (int i=0; I < local_ub; i++)
    a[i] = 0;
}
```

Non-Unit Stride Access

Non-unit stride access: Nonconsecutive memory locations are being accessed in the loop

- Vectorization might still be possible (e.g. in case access is regular/linear), the data arrangement operations might be too expensive
  - Vector report: "Loop was not vectorized: vectorization possible but seems inefficient"

Samples:

```
for (I=0;I<=MAX;I++)
  for (J=0;J<=MAX;J++)
  {
    D[I][J]+=1;                    // Unit Stride
    D[J][I]+=1;                    // Non-Unit but linear
    A[J*J]+=1;                     // Non-unit
    A[B[J]]+=1;                    // Non-Unit
    if (A[MAX-J])==1) last=J;      // Non-Unit
  }
```

Avoiding Non-Unit Stride Access

Code transformations like loop interchange can avoid non-unit access frequently in case access is linear

Compiler does this automatically in many cases; popular sample: matrix multiplication loop

- The compiler will swap inner loops to get unit-stride access

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

But in other cases, the exchange has to be done manually: The following loops are not interchanged implicitly:

```
// Non-unit access
for (j = 0; j < N; j++)
 for (i = 0; i <= j; i++)
  c[i][j] = a[i][j]+b[i][j];
```

```
// Unit access
for (i = 0; i < N; i++)
 for (j = i; i <= N; j++)
  c[i][j] = a[i][j]+b[i][j];
```

Function Calls / In-lining

- Function calls prevent vectorization in general
  - Exception #1 : Call of "intrinsic" functions like math routines
  - Exception #2 : Successful in-lining of called routine
    - Inter-procedural optimization enables in-lining of routines defined even in separate source files

Intel Compiler: 15 times faster by using –ipo!*

```
for (i=1;i<nx;i++) {
    x = x0 + i*h;
    sumx = sumx + func(x,y,xp,yp);
}

float func(float x, float y, float xp, float yp)
{
  float denom;
  denom = (x-xp)*(x-xp) + (y-yp)*(y-yp);
  denom = 1./sqrt(denom);
  return denom;
}
```

*: Intel® C++ Compiler 12.0 U1 for Linux, Redhat Enterprise Linux 64bit 6.0, Intel XEON® X5560 processor, 2.8GHz

Function Calls / In-lining [2]

- Success of in-lining can be verified using the optimization report

  `-opt-report -opt-report-phase`ipo_inl
- Intel compilers offer a large set of switches, directives and language extensions to control in-lining globally or locally
  - E.g `#pragma forceinline` which instructs the compiler to ignore the heuristic for in-lining and to inline all calls in the following statements/block (C/C++ only)
  - See compiler manual for details
- Inter-procedural optimization offers additional advantages to vectorization
  - Inter-procedural alignment analysis
  - Improved (more precise) dependence analysis

Vectorizable Mathematical Functions

Calls to most mathematical function in a loop body are "vectorized" too by calling vector versions of the function provided by the "Short Vector Math Library" – libsvml

- Libsvml is optimized for latency compared to the VML library component of Intel® MKL which realizes same functionality but which is optimized for throughput
- Routines in libsvml can be called explicitly too (see manual)

This is the set of mathematical routines which have a vector implementation in libsvml
(Intel® Composer XE 2011)

acos	ceil	fabs	round
acosh	cos	floor	sin
asin	cosh	fmax	sinh
asinh	erf	fmin	sqrt
atan	erfc	log	tan
atan2	erfinv	log10	tanh
atanh	exp	log2	trunc
cbrt	exp2	pow	


intel
Software

Appendix

Command line performance monitoring

Command line perf monitoring

- With rdmsr and wrmsr we can now easily do performance monitoring in a lightweight way.
- Before going on to feature burden apps like Vtune, let's now do a quick performance analysis directly from the command line

Command line perf monitoring
Architectural perf events

- Intel doesn't guarantee the consistency of the performance monitoring unit
- Events can change with each model
- There are a number of events that Intel guarantees always to be present
- These are called Architectural performance events

Command line perf monitoring

Architectural perf events

Table A-1. Architectural Performance Events

Event Num.	Event Mask Mnemonic	Umask Value	Description	Comment
3CH	UnHalted Core Cycles	00H	Unhalted core cycles	
3CH	UnHalted Reference Cycles	01H	Unhalted reference cycles	Measures bus cycle[1]
C0H	Instruction Retired	00H	Instruction retired	
2EH	LLC Reference	4FH	LL cache references	
2EH	LLC Misses	41H	LL cache misses	
C4H	Branch Instruction Retired	00H	Branch instruction retired	
C5H	Branch Misses Retired	00H	Mispredicted Branch Instruction retired	

Command line perf monitoring

Architectural perf events

```
for c in 0 1 2 3; do
#Enable perf mon in IA32_PERF_GLOBAL_CTRL
 wrmsr -p $c 911 30064771087
#Enable fixed perf m. in IA32_FIXED_CTR_CTRL
 wrmsr -p $c 909 819
#IA32_PERFEVTSEL0(390 dec) for total branches
 wrmsr -p $c 390  0x4700c4
 #IA32_PERFEVTSEL1 (391 dec) for branch miss.
 wrmsr -p $c 391  0x4700c5
done
```

Command line perf monitoring

Architectural perf events

```
while[ 1 ]; do
  for c in 0 1 2 3; do
    wrmsr -p $c  193 0  #reset counter 0
    wrmsr -p $c  194 0  #reset counter 1
  done
  sleep 5
  for c in 0 1 2 3; do
   BRREF=`rdmsr -p $c  193` #read counter 0
   BRMISS=`rdmsr -p $c  194` #read counter 1
   echo $c `echo $BRMISS/$BRREF | bc -l`
  done
done
```

Command line perf monitoring

Advanced Example

proc	CPI	LLC misses	Branch misspred
0	1.35079	.30312	.00057
1	1.36037	.29671	.00082
2	1.35890	.30145	.00090
3	1.36869	.29146	.00049
4	1.36068	.29906	.00107
5	1.36653	.29260	.00047
6	1.35879	.30083	.00096
7	1.37438	.29364	.00044

app with cache blocking

proc	CPI	LLC misses	Branch misspred
0	.49865	.99099	.01582
1	.46621	.99247	.01260
2	.49805	.99210	.01281
3	.46907	.98990	.01363
4	.49935	.99038	.01508
5	.46850	.99154	.01374
6	.49610	.99049	.01602
7	.46863	.99231	.01207

app without cache blocking

Command line perf monitoring
Summary

- You can do this for all kind of events you are interested in
- Provides immediate feedback
- Ideal for first view assessments
- Ideal for command line work


intel
Software