Лекция 1
Архитектурно-ориентированная оптимизация программного обеспечения (software optimization introduction)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com
WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)
Осенний семестр, 2015

Архитектура ЭВМ Дж. фон Неймана

Основные черты архитектуры ЭВМ Дж. Фон Неймана

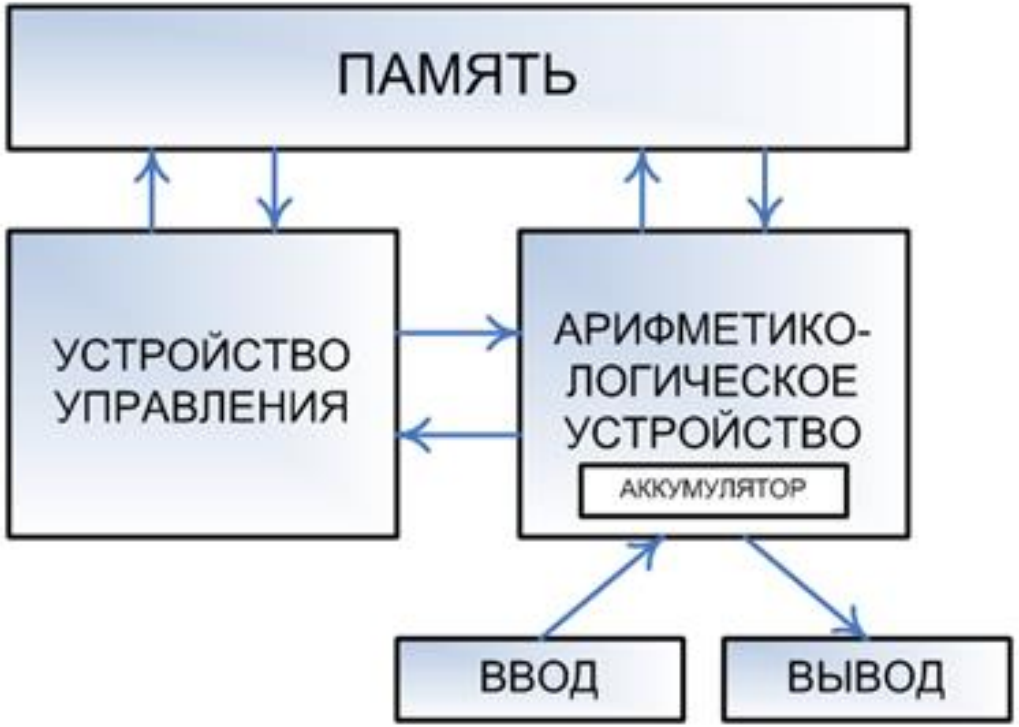
- **Принцип однородности памяти –** команды и данные хранятся в одной и той же памяти (внешне неразличимы)
- **Принцип адресности** – память состоит из пронумерованных ячеек, процессору доступна любая ячейка
- **Принцип программного управления** – вычисления представлены в виде программы, состоящей из последовательности команд
- **Принцип двоичного кодирования** – вся информация, как данные, так и команды, кодируются двоичными цифрами 0 и 1

Первые ЭВМ с хранимой в памяти программой

- **EDVAC** (1944-1951, США):
  Дж. Экерт, Дж. Мокли, Дж. Фон Нейман, Г. Голдсайн

The EDVAC as installed in Building 328 at the Ballistics Research Laboratory, Maryland USA // Wikipedia

Архитектура ЭВМ Дж. фон Неймана



Узкое место архитектуры Дж. фон Неймана –
совместное использование шины для доступа к памяти за данными
и командами программы (это ограничивает пропускную способность
между процессором и памятью)

Пути увеличения производительности ЭВМ

Совершенствование элементной базы

- **Электромеханические реле, вакуумные лампы**

↓

- **Транзисторы**

↓

- **Микросхемы низкой степени интеграции**

↓

- **БИС, СБИС, микропроцессоры**

↓ **?**

- ***Оптические, квантовые, молекулярные процессоры***

Пути увеличения производительности ЭВМ

Совершенствование архитектуры ЭВМ

- **Совершенствование процесса выполнения инструкций**
  - **Совершенствование архитектуры набора команд (Instruction Set Architecture – ISA):** RISC, CISC, MISC, VLIW
  - **Параллелизм уровня инструкций (Instruction Level Parallelism – ILP)**: конвейерная архитектура, суперскалярная обработка, VLIW
  - **Параллелизм уровня потоков (Thread Level Parallelism – TLP)**: многопроцессорные системы, одновременная многопоточность, многоядерные процессоры
  - **Параллелизм данных (Data Parallelism):** векторная обработка данных (векторные процессоры/инструкции)

Пути увеличения производительности ЭВМ

Совершенствование архитектуры ЭВМ

- **Смена парадигмы организации вычислений**

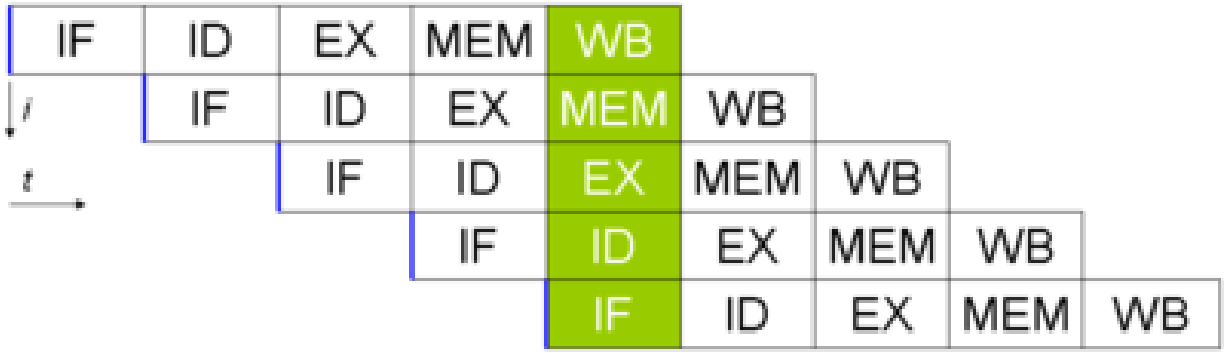
- **Архитектура с управлением потоком команд (Control flow, классическая архитектура фон Неймана)** – последовательность выполнения инструкций задана программой

- **Архитектура с управлением потоком данных (Data flow)** – нет счетчика инструкций, команды выполняются по готовности входных данных (операндов), порядок выполнения операций заранее неизвестен

Бурцев В.С. *Новые принципы организации вычислительных процессов высокого параллелизма* // МСО-2003, URL: http://old.lvk.cs.msu.su/files/mco2003/burtsev.pdf

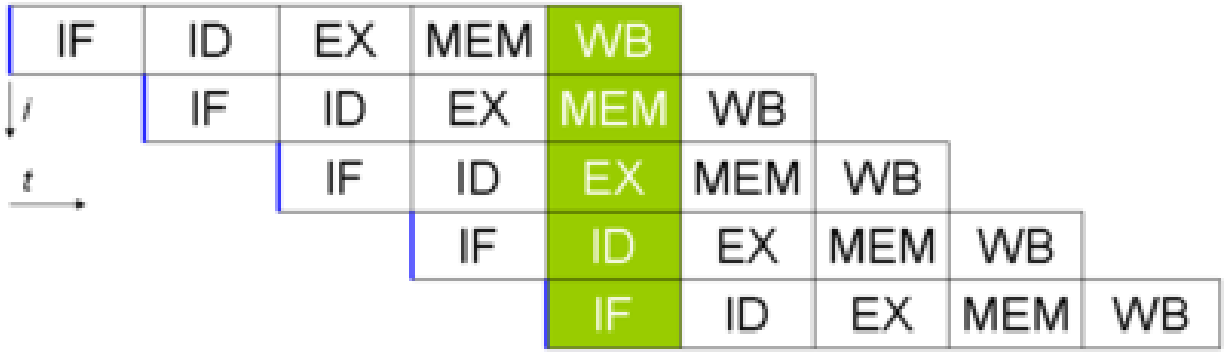
Параллелизм уровня инструкций (Instruction Level Parallelism – ILP)

Вычислительный конвейер (Instruction pipeline)



- Микроконтроллеры **Atmel AVR**, PIC – 2-этапный конвейер
- **Intel 80486** – 5-stage (scalar, CISC)
- **Intel Pentium** – 5-stage (2 integer execution units)
- **Intel Pentium Pro** – 14-stage pipeline
- **Intel Pentium 4** (Cedar Mill) – 31-stage pipeline
- **Intel Core i7 4771** (Haswell) – 14-stage pipeline
- **ARM Cortex-A15** – 15 stage integer/17–25 stage floating point pipeline

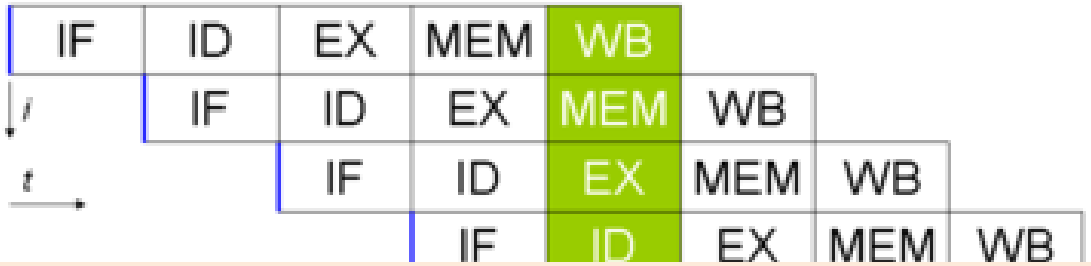
Вычислительный конвейер (Instruction pipeline)



Основные этапы обработки инструкций
(RISC pipeline)

1. **IF** (Instruction Fetch) – загрузка инструкции из памяти (кэша инструкций, 1 такт)
2. **ID** (Instruction Decode) – декодирование инструкции
3. **EX** (Execution) – выполнение
   (Reg-Reg Op. – 1 такт, Mem. ref. – 2 такта; Div, Mul, FP Ops. – >1 такта)
4. **MEM** (Memory access) – доступ к памяти (чтение/запись)
5. **WB** (Register Write Back) – запись в регистры

Вычислительный конвейер (Instruction pipeline)



Максимальное ускорение (Pipeline speedup)
равно числу этапов конвейера

1. **IF** (Instruction Fetch) – загрузка инструкции из памяти (кэша инструкций, 1 такт)
2. **ID** (Instruction Decode) – декодирование инструкции
3. **EX** (Execution) – выполнение
   (Reg-Reg Op. – 1 такт, Mem. ref. – 2 такта; Div, Mul, FP Ops. – >1 такта)
4. **MEM** (Memory access) – доступ к памяти (чтение/запись)
5. **WB** (Register Write Back) – запись в регистры

Pentium II Pipeline (11 stages)

1. Processor fetches instructions from memory in (static) program sequence
2. Each instruction is translated into one or more fixed-length RISC instructions (micro-ops)
3. Processor executes the micro-ops in superscalar pipeline fashion;
   micro-ops may execute out-of-order
4. Processor commits results of each micro-op to the register set in (dynamic) program sequence

IFU	Instruction Fetch Unit
ID	Instruction Decode
RAT	Register Allocation Table (or Allocator)
ROB	Reorder Buffer
DIS	Dispatcher
EX	Execute Stage
RET	Retire Unit

Конфликты данных (Data hazards)

- **Текущий шаг конвейера не может быть выполнен, так как зависит от результатов выполнения предыдущего шага**
- Возможные причины:
  - **Read After Write (RAW)** – True dependency
    i1: R2 = R1 + R3
    i2: R4 = R2 + R3
  - **Write After Read (WAR)** – Anti-dependency
    R4 = R1 + R3
    R3 = R1 + R2
  - **Write After Write (WAW)** – Output dependency
    R2 = R4 + R7
    R2 = R1 + R3

Для разрешения проблемы конвейер приостанавливается (pipeline stall, bubble)

Конфликты данных (Data hazards)

- **Текущий шаг конвейера не может быть выполнен, так как зависит от результатов выполнения предыдущего шага**

- Возможные причины:

  - **Read After Write (RAW)** – True dependency
    i1: R2 = R1 + R3
    i2: R4 = R2 + R3

  - **Write After Read (WAR)** – Anti-dependency
    R4 = R1 + R3
    R3 = R1 + R2

  - **Write After Write (WAW)** – Output dependency
    R2 = R4 + R7
    R2 = R1 + R3

Step	IF	ID	EX	ST
1	i1			
2	i2	i1		
3		i2	i1	
4			i2	i1
5				

Data Hazard – Read After Write (RAW)
Для разрешения проблемы конвейер приостанавливается (pipeline stall, bubble)

Конфликты управления (Control hazards)

```
.text
    movl  %ebx, %eax
    cmpl  $0x10, %eax
    jne   not_equal
    movl %eax, %ecx
    jmp end

not_equal:
    movl $-0x1, %ecx

end:
```

Step	IF	ID	EX	ST
1	movl			
2	cmpl	movl		
3	jne	cmpl	movl	
4	???	jne	cmpl	movl
5				
6				
7				

Какую инструкцию выбирать
из памяти (IF) на шаге 4?
(инструкция jne еще не выполнена)

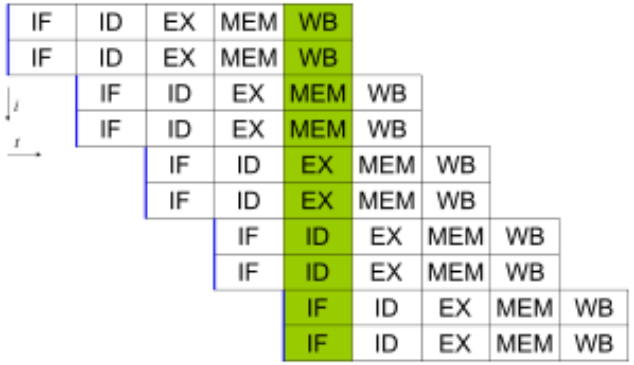
В процессоре присутствует **модуль предсказания переходов**
(Branch Prediction Unit – BPU), который управляет счетчиком команд

Суперскалярные процессоры (Superscalar)

- **Исполняющие модули конвейера присутствуют в нескольких экземплярах** (несколько ALU, FPU, Load/Store-модулей)
- За один такт процессор выполняет параллельно несколько инструкций
- Процессор динамически проверяет входные инструкции на зависимость по данным – динамический планирование выполнения инструкция (идеи dataflow-архитектуры)
- Внеочередное исполнение команд (Out-of-order execution) – переупорядочивание команд для максимально загрузки ALU, FPU, Load/Store (минимизация зависимости по данным между инструкциями, выполнение инструкций по готовности их данных)
- Dynamic scheduling: scoreboarding (CDC 6600, 1965), алгоритм Р. Томасуло (IBM S/360, 1967)

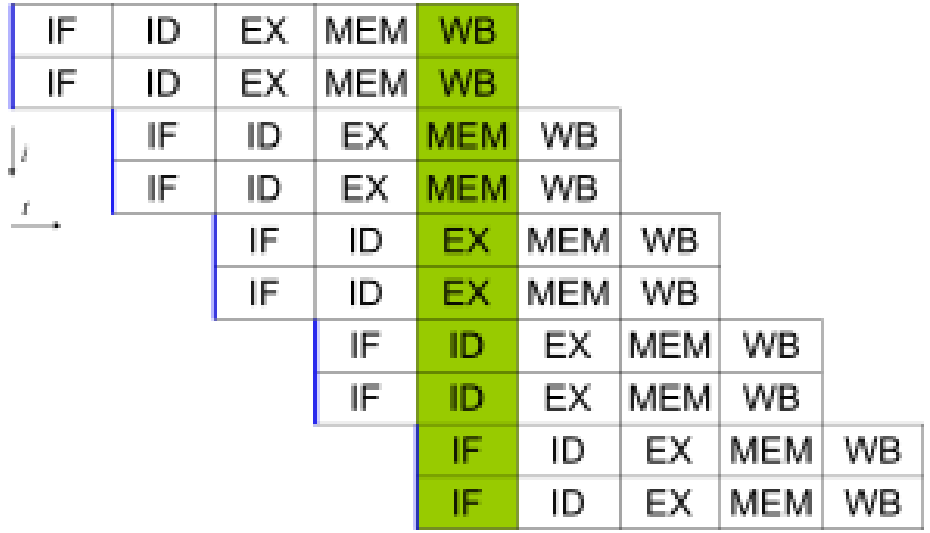
```
a = b + c
d = e + f
```

```
a = b + c
b = e + f
```

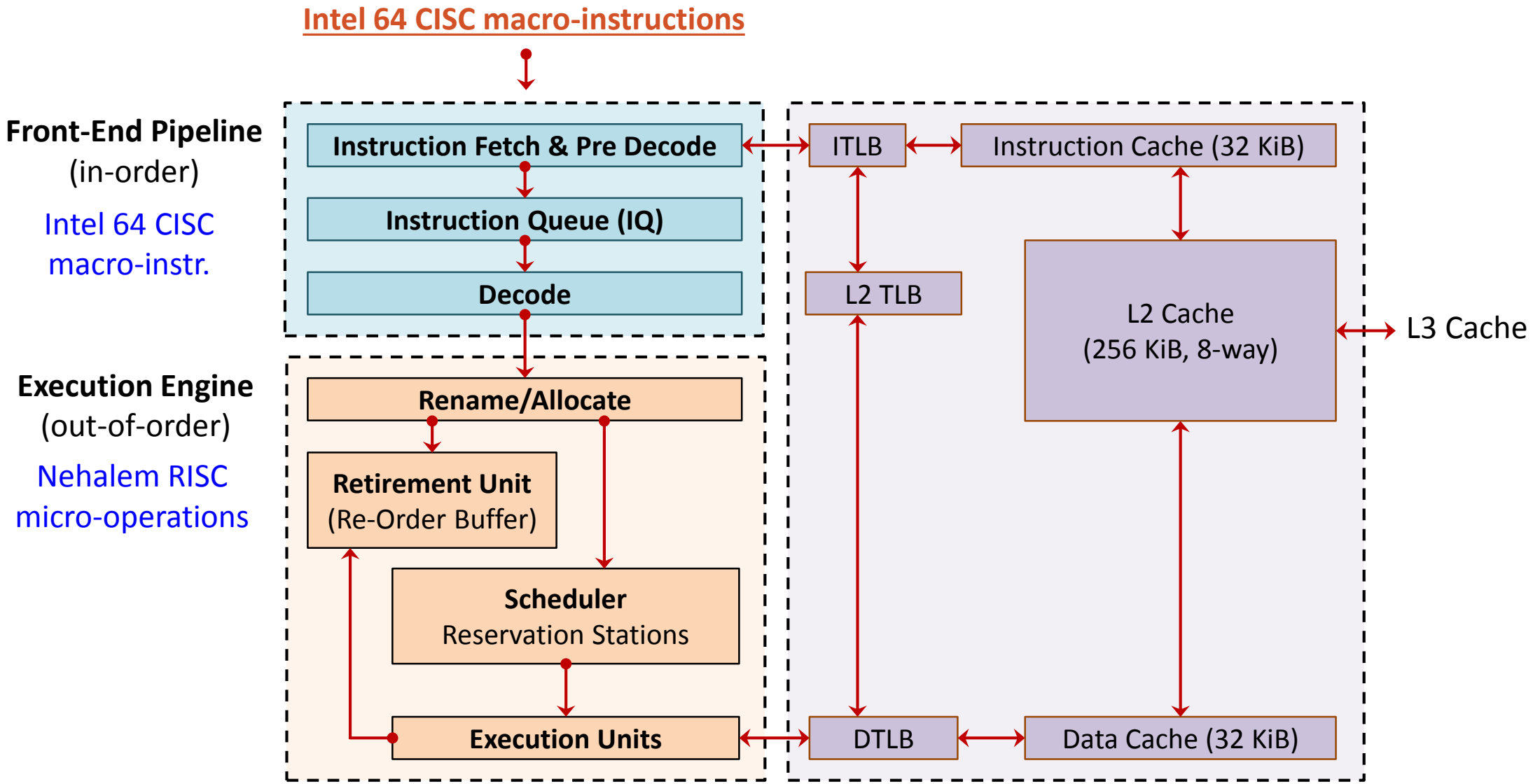


Суперскалярные процессоры (Superscalar)

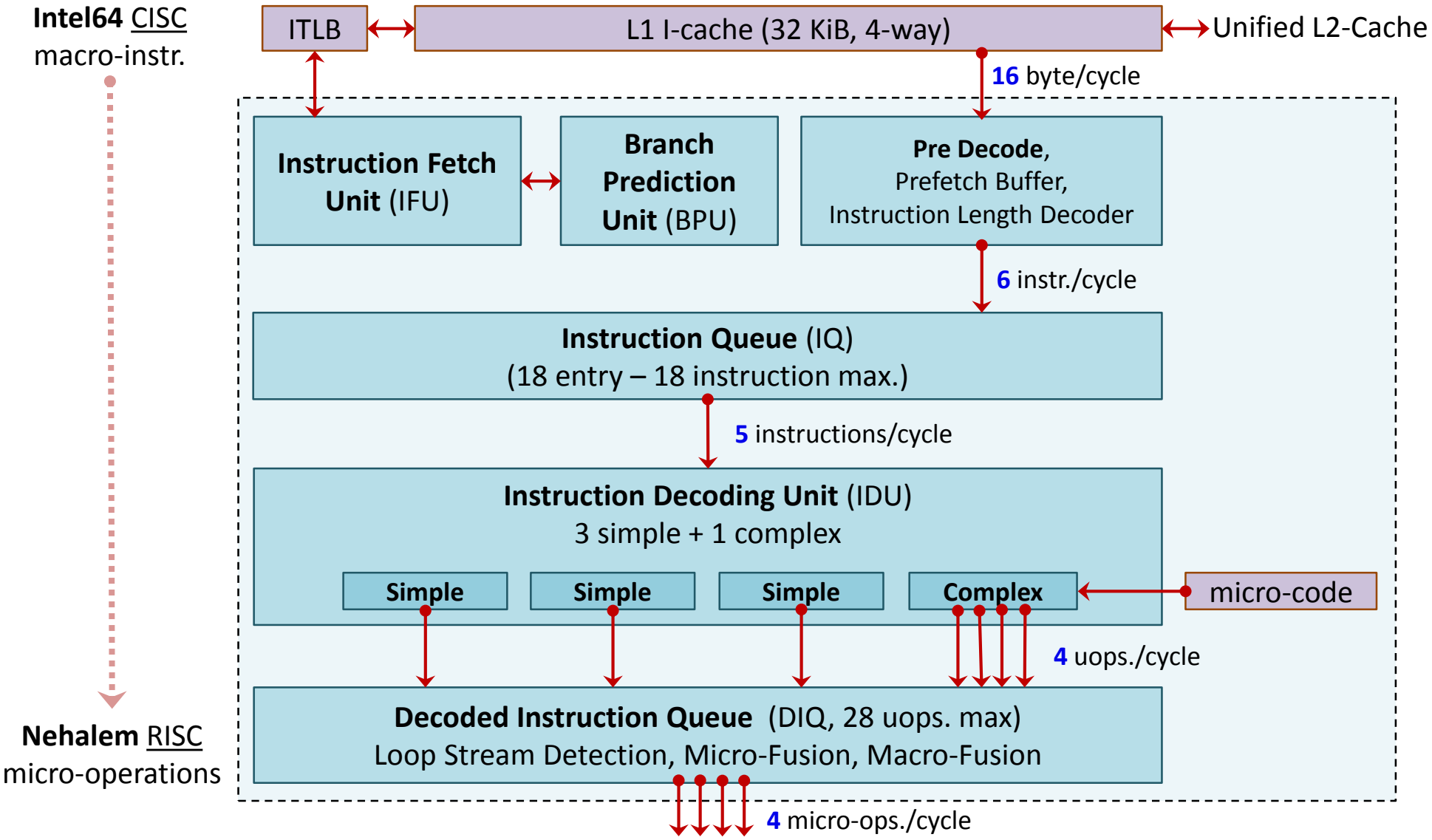
- **CDC 6600** (1965)
- **Intel i960CA** (1988)
- **AMD 29000-series 29050** (1990)
- **Intel Pentium** – первый суперскалярный x86 процессор, 2 datapaths (pipelines)



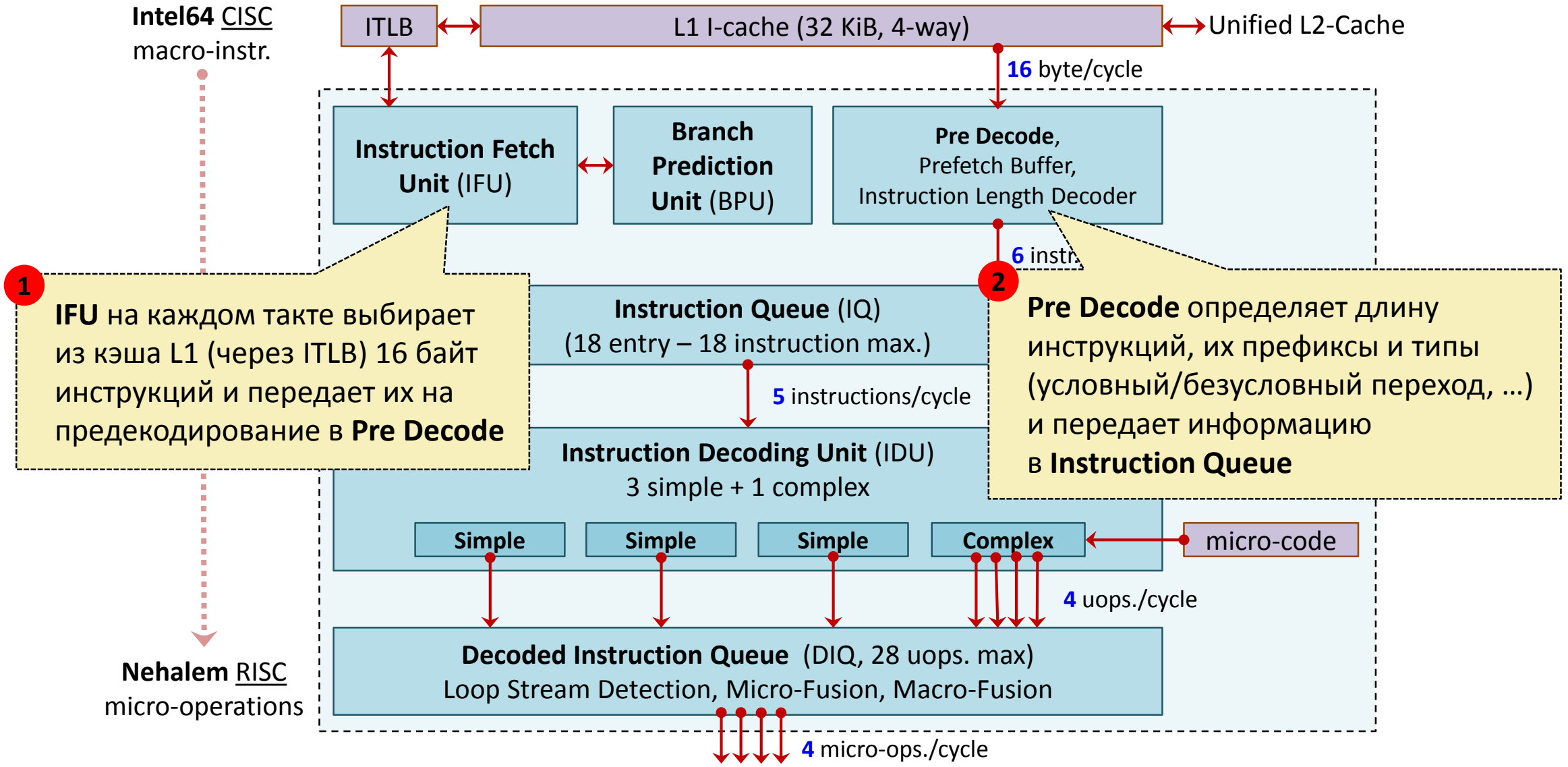
Intel Nehalem Core Pipeline



Intel Nehalem Frontend Pipeline (in-order)



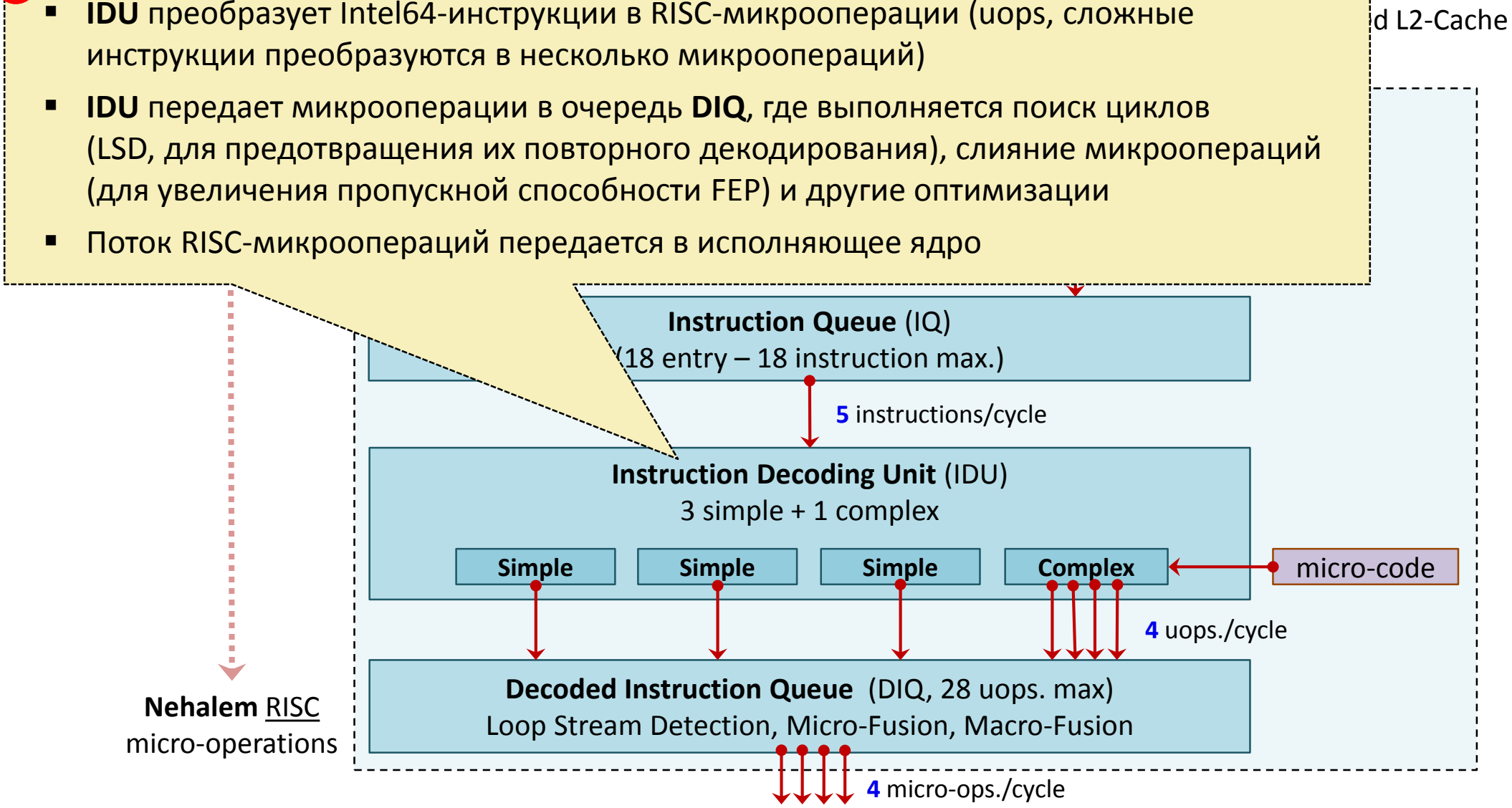
Intel Nehalem Frontend Pipeline (in-order)



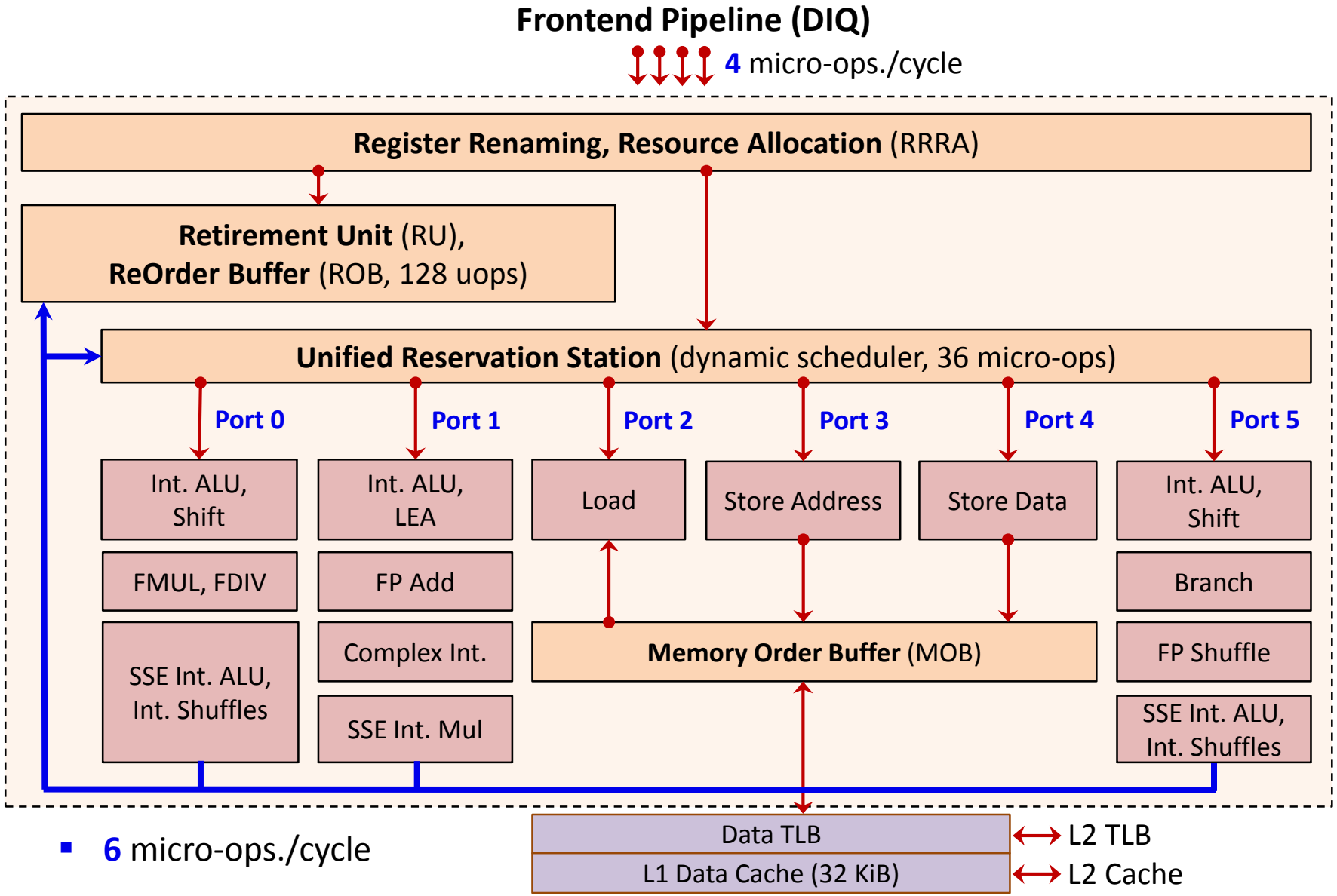
Intel Nehalem Frontend Pipeline (in-order)

3

- **IDU** преобразует Intel64-инструкции в RISC-микрооперации (uops, сложные инструкции преобразуются в несколько микроопераций)
- **IDU** передает микрооперации в очередь **DIQ**, где выполняется поиск циклов (LSD, для предотвращения их повторного декодирования), слияние микроопераций (для увеличения пропускной способности FEP) и другие оптимизации
- Поток RISC-микроопераций передается в исполняющее ядро

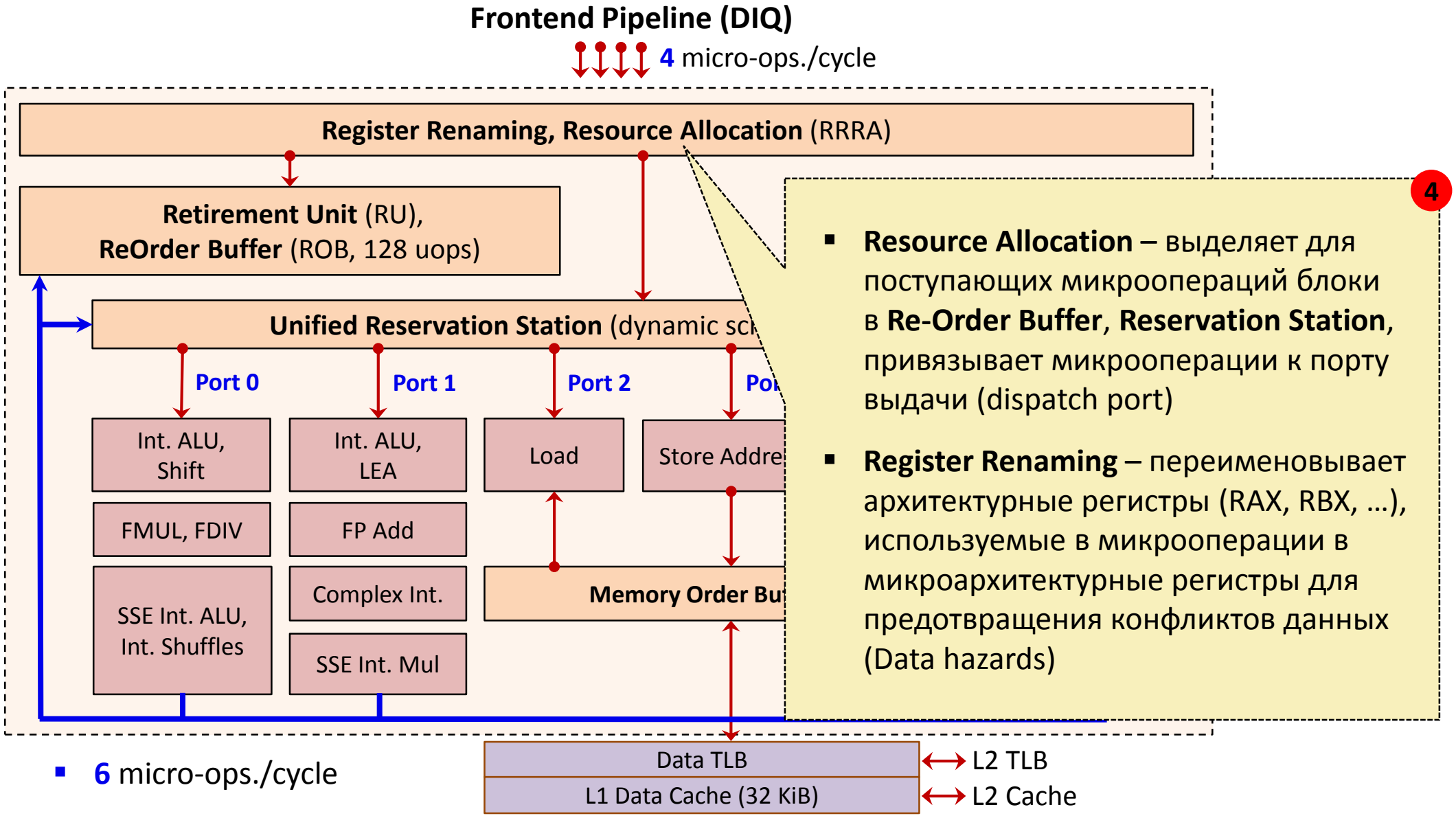


Intel Nehalem Execution Engine



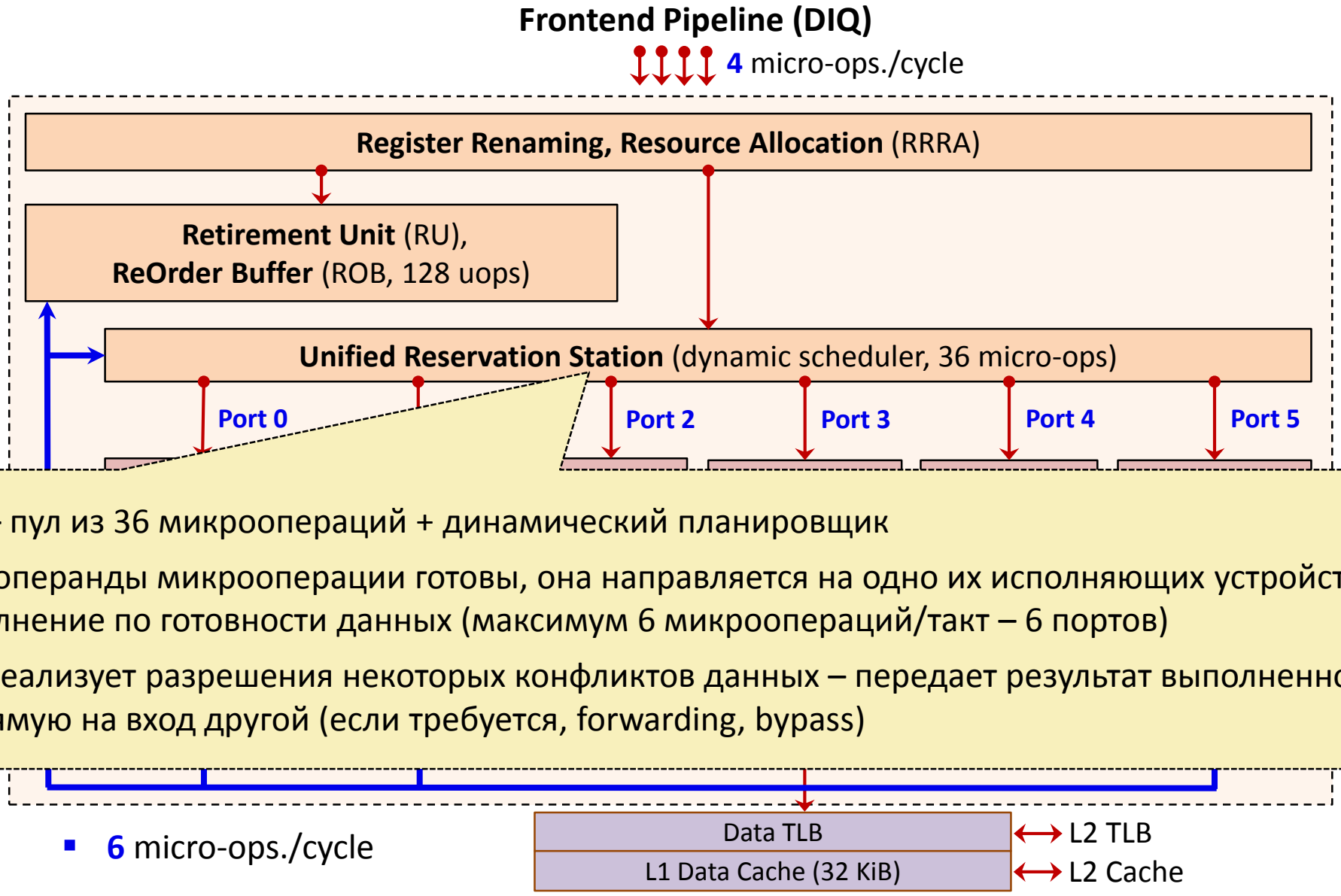
- **6** micro-ops./cycle

Intel Nehalem Execution Engine



- **6** micro-ops./cycle

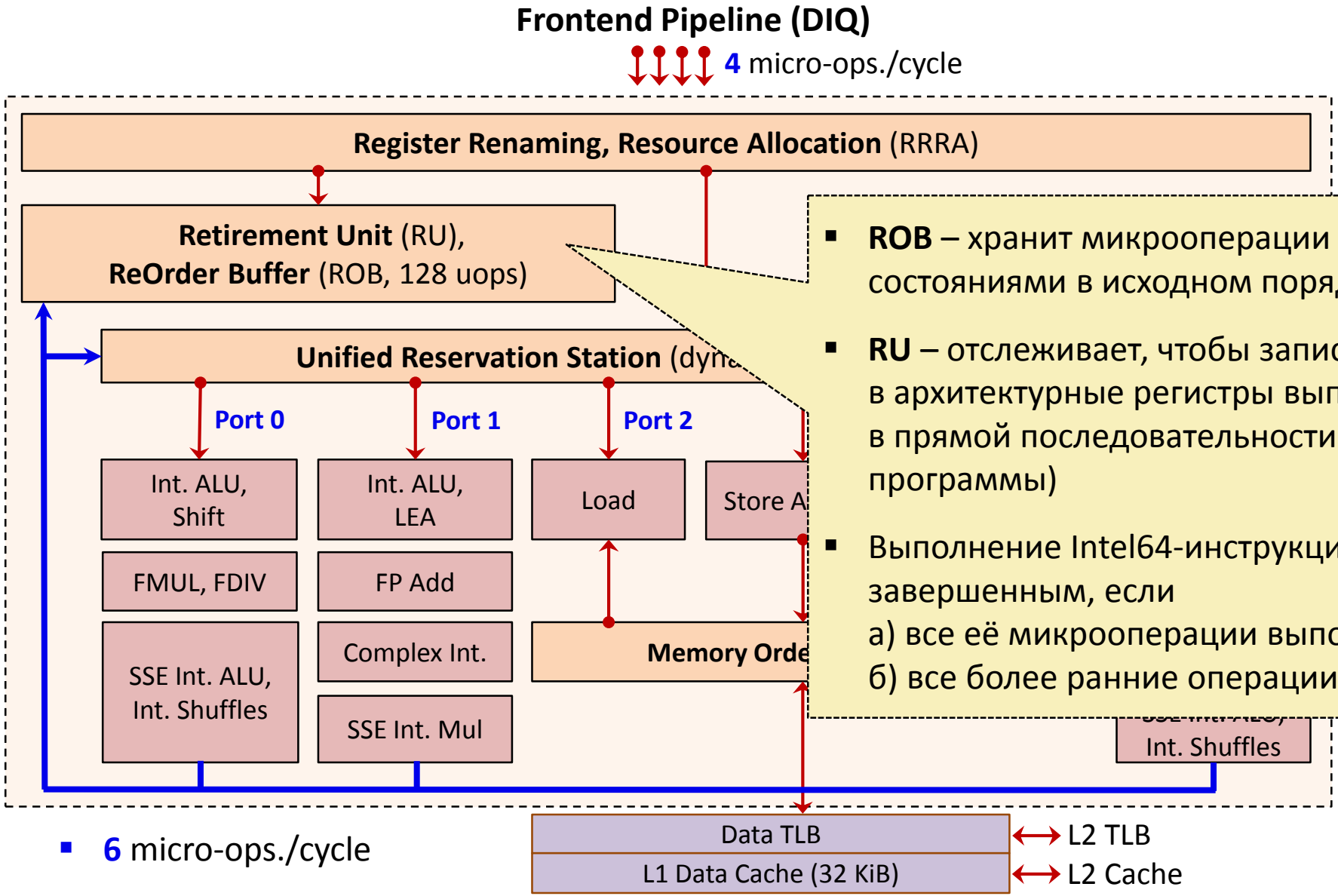
Intel Nehalem Execution Engine



5

- **URS** – пул из 36 микроопераций + динамический планировщик
- Если операнды микрооперации готовы, она направляется на одно их исполняющих устройств – выполнение по готовности данных (максимум 6 микроопераций/такт – 6 портов)
- URS реализует разрешения некоторых конфликтов данных – передает результат выполненной операции напрямую на вход другой (если требуется, forwarding, bypass)

Intel Nehalem Execution Engine



6

- **ROB** – хранит микрооперации с их состояниями в исходном порядке (in-order)
- **RU** – отслеживает, чтобы запись результатов в архитектурные регистры выполнялась в прямой последовательности (исходной программы)
- Выполнение Intel64-инструкции считается завершенным, если
  а) все её микрооперации выполнены,
  б) все более ранние операции завершены

- **6** micro-ops./cycle

Процессоры с широким командным словом (VLIW)

- **VLIW-архитектура** (Very Long Instruction Word) – широкие инструкции содержат в себе несколько команд, которые можно выполнять параллельно на ALU/FPU/Load-Store-модулях
- Статическое планирование (формирование “широких” команд) возложено на компилятор (static scheduling)

- **Примеры**
  - **Эльбрус 2000** – 6 ALU, до 23 операций за такт
  - **Intel Itanium** – 3 команды в 128-битной широкой инструкции, на стадии IF загружается 2 инстр. => 6 команд за такт
  - **Transmeta Efficeon** – 2 ALU, 2 LD/ST, 2 FPU; 8 команд в 256-битной широкой инструкции
  - **Texas Instruments TMS320C6x** – 8 команд в 256-битной широкой инструкции

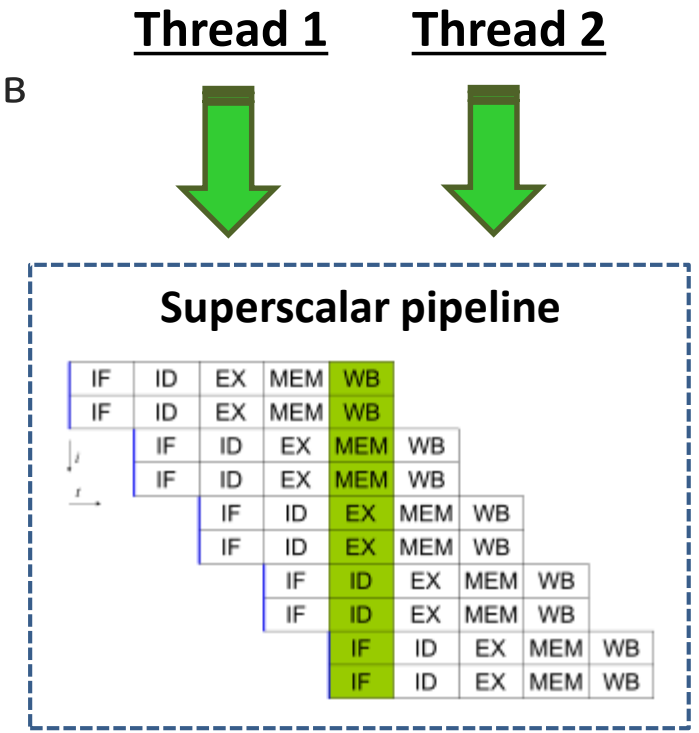
Одновременная многопоточность (Simultaneous multithreading)

- **Одновременная многопоточность** (Simultaneous multithreading – SMT, hardware multithreading) — технология, позволяющая выполнять инструкции из нескольких потоков выполнения (программ) на одном суперскалярном конвейере

- **Потоки разделяют один суперскалярный конвейер процессора (ALU, FPU, Load/Store)**

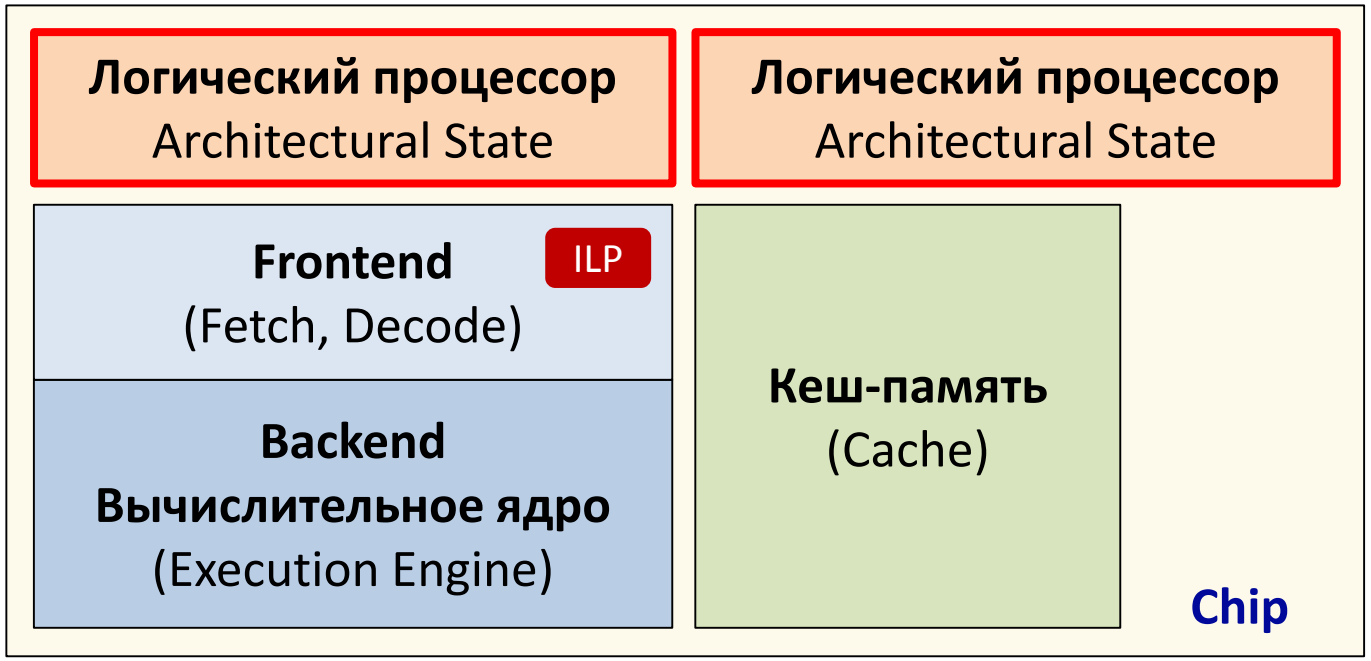
- SMT позволяет повысить эффективность использования модулей суперскалярного процессора (ALU, FPU, Load/Store) за счет наличия большего количества инструкций из разных потоков выполнения (ниже вероятность зависимости по данным)

- Примеры реализации:
  - IBM ACS-360 (1968 г.), DEC Alpha 21464 (1999 г., 4-way SMT)
  - Intel Pentium 4 (2002 г., **Intel Hyper-Threading**, 2-way SMT)
  - Intel Xeon Phi (4-way SMT), Fujitsu Sparc64 VI (2-way SMT), IBM POWER8 (8-way SMT)



Разделение ресурсов ALU, FPU, Load/Store

Intel Hyper-Threading Technology



http://www.intel.ru/content/www/ru/ru/architecture-and-technology/hyper-threading/hyper-threading-technology.html

- Architectural state + Interrupt controller (LAPIC) = **Logical processor**
- **2 потока разделяют суперскалярный конвейер**
- Ускорение (Speedup) ~ 30 %
- Architectural state = {
  - Регистры общего назначения (RAX, RBX, RCX, …)
  - Сегментные регистры (CS, DS, …),
  - Управляющие регистры (RFLAGS, RIP, GDTR)
  - X87 FPU-регистры, MMX/XMM/YMM-регистры
  - MSR-регистры, Time stamp counter

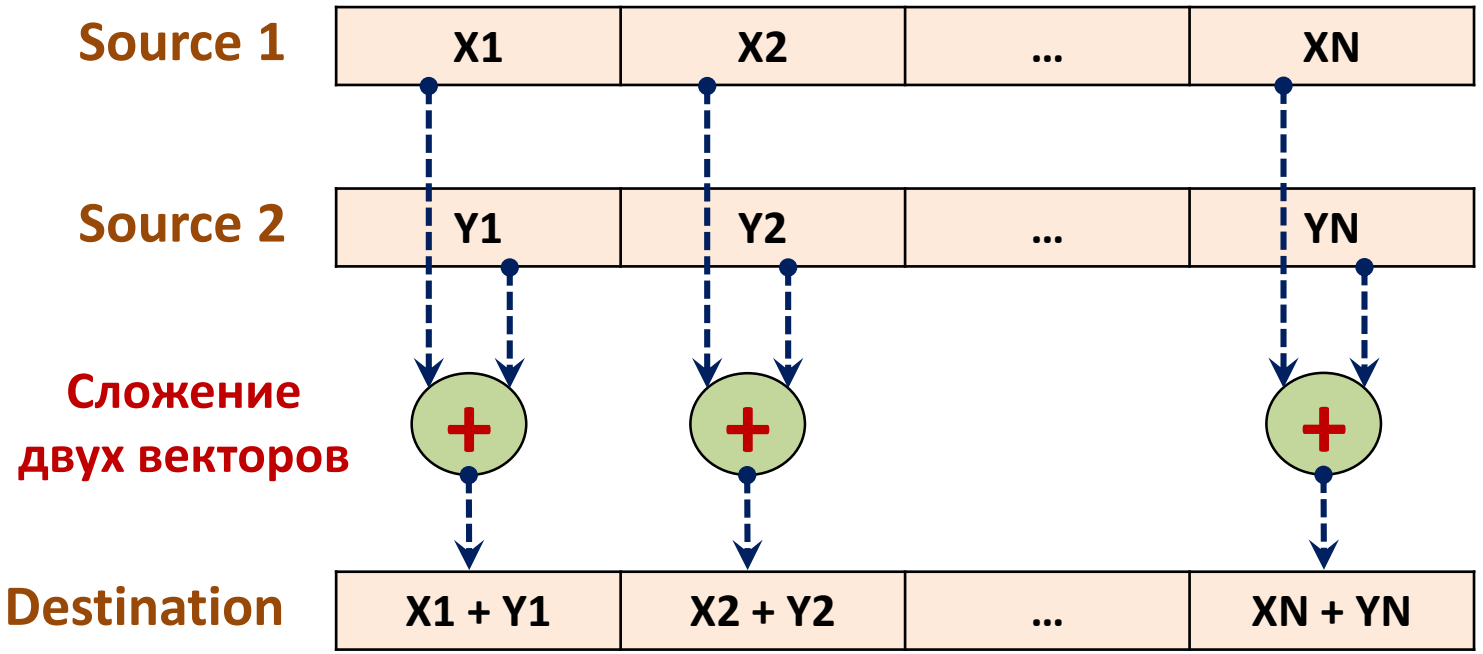
}

Параллелизм уровня данных (Data Parallelism – DP)

Векторные процессоры

- **Векторный процессор (Vector processor) –** процессор поддерживающий на уровне системы команд операции для работы с векторами (SIMD-инструкции)

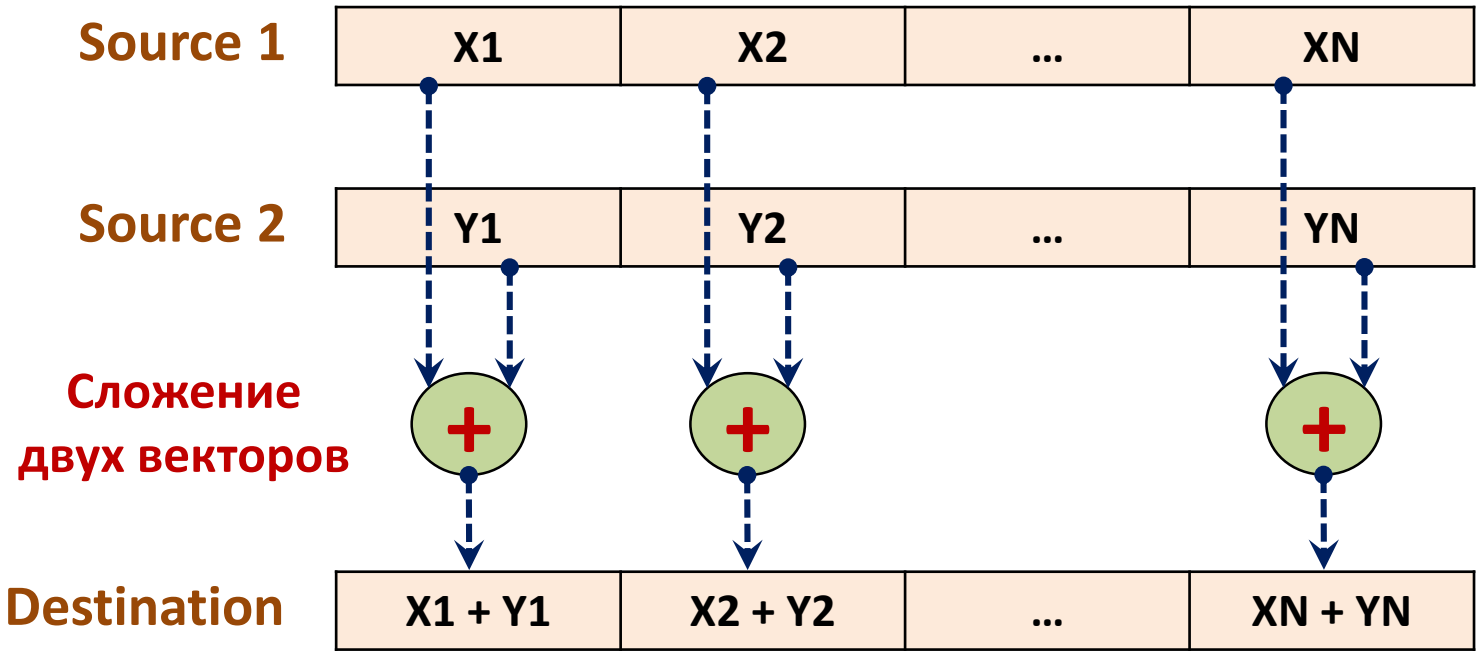
- **Векторные вычислительные системы**
  - **CDC STAR-100** (1972 г., векторы до 65535 элементов)
  - **Cray Research Inc.**: Cray-1 (векторные регистры), Cray-2, Cray X-MP, Cray Y-MP



Векторные процессоры

Максимальное ускорение (Speedup)
линейно зависит от числа элементов в векторном регистре

(использование векторных инструкций может привести к сокращению количества команд в программе, а это может обеспечить более эффективное использование кеш-памяти)



SIMD-инструкции в современных процессорах

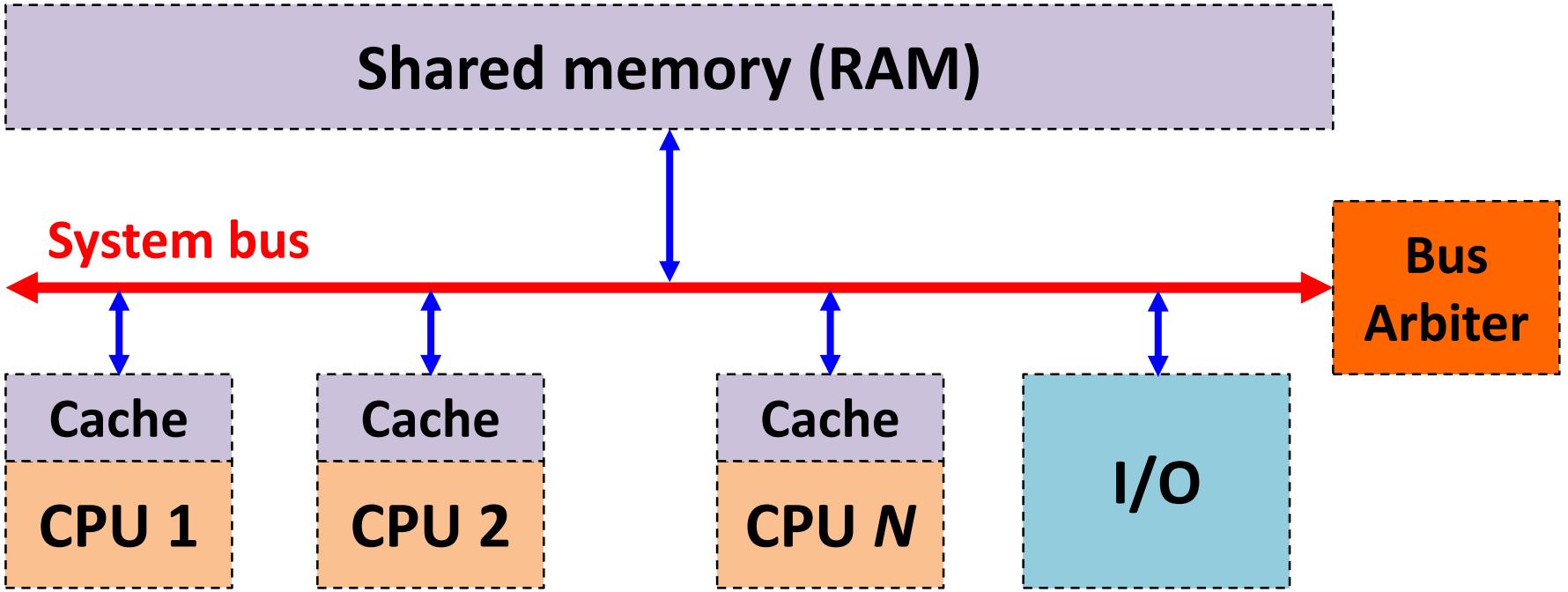
- Intel MMX (1996)
- Motorola PowerPC, IBM POWER **AltiVec** (1999)
- AMD 3DNow! (1998)
- Intel SSE (Intel Pentium III, 1999)
- Intel SSE2, SSE3, SSE4
- **AVX** (Advanced Vector Extension, Intel & AMD, 2008)
- AVX2 (Haswell, 2013)
- AVX-512 (2015)
- ARM Advanced SIMD (**NEON**) – Cortex-A8, 2011

Пример использования SSE-инструкций

```
void add_sse_asm(float *a, float *b, float *c)
{
    __asm__ __volatile__
   (
       "movaps (%[a]), %%xmm0 \n\t"
       "movaps (%[b]), %%xmm1 \n\t"
       "addps %%xmm1, %%xmm0 \n\t"
       "movaps %%xmm0, %[c] \n\t"
       : [c] "=m" (*c)
       : [a] "r" (a), [b] "r" (b)
       : "%xmm0", "%xmm1"
    );
}
```

Параллелизм уровня потоков (Thread Level Parallelism – TLP)

Многопроцессорные SMP-системы (Symmetric multiprocessing)

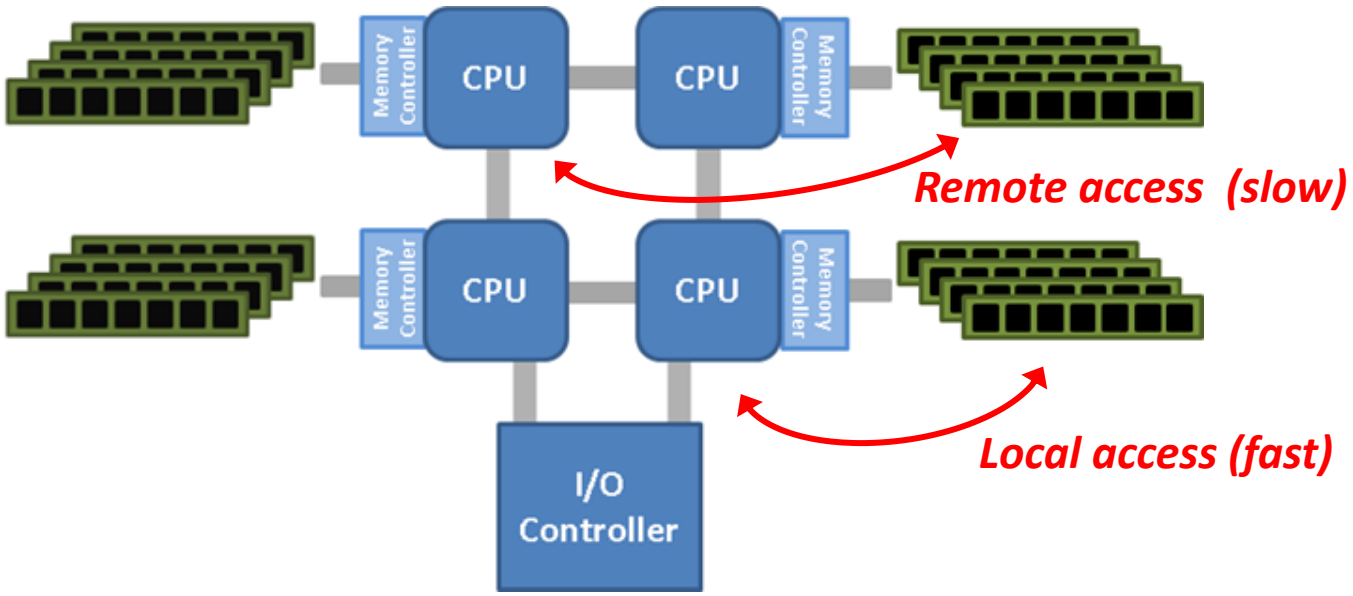


- Процессоры SMP-системы имеют одинаковое время доступа к разделяемой памяти (симметричный доступ)
- Системная шина (System bus) – это узкое место, ограничивающее масштабируемость вычислительного узла

Многопроцессорные SMP-системы (Symmetric multiprocessing)

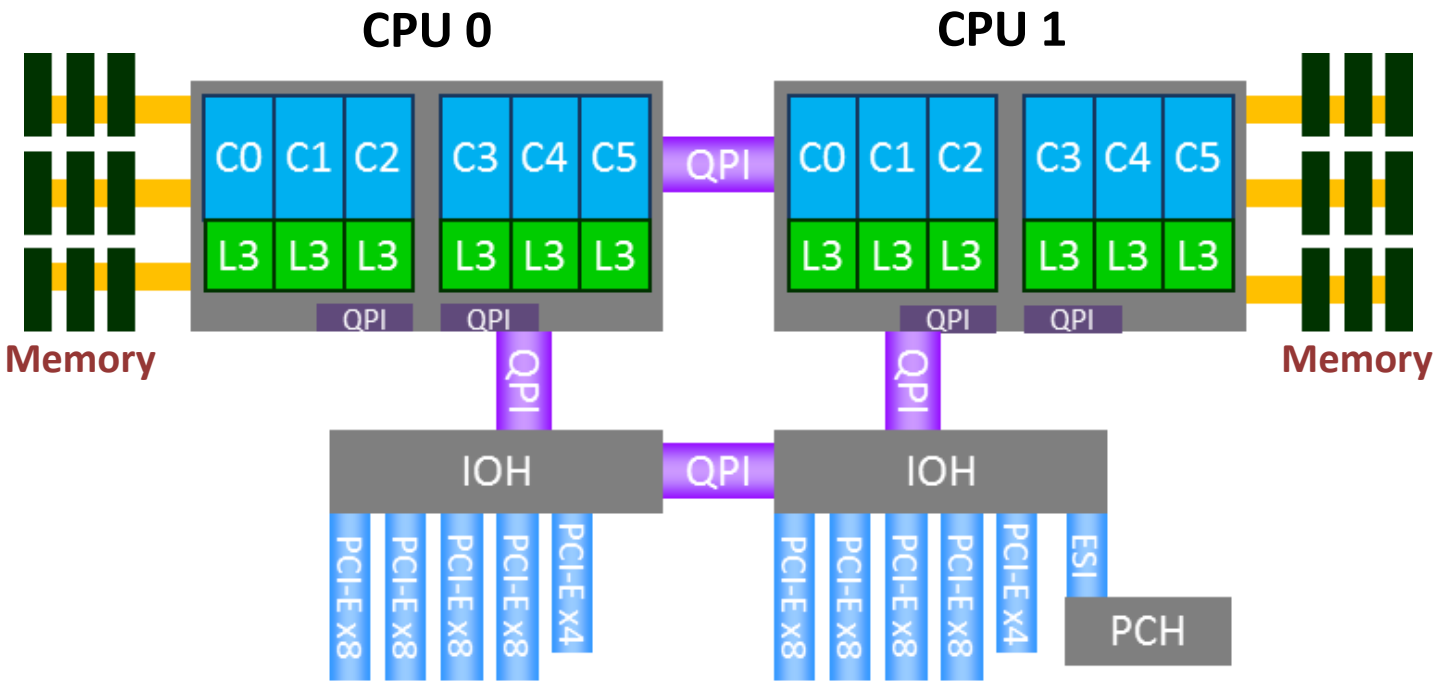
Многопроцессорные NUMA-системы (AMD)

- **NUMA** (Non-Uniform Memory Architecture) – это архитектура вычислительной системы с неоднородным доступом к разделяемой памяти
- Процессоры сгруппированы в NUMA-узлы со своей локальной памятью
- Доступ к локальной памяти NUMA-узла занимает меньше времени по сравнению с временем доступом к памяти удаленных процессоров



- 4-х процессорная NUMA-система
- Каждый процессор имеет интегрированный контроллер и несколько банков памяти
- Процессоры соединены шиной **Hyper-Transport** (системы на базе процессоров AMD)
- Доступ к удаленной памяти занимает больше времени (для Hyper-Transport ~ на 30%, 2006 г.)

Многопроцессорные NUMA-системы (Intel)

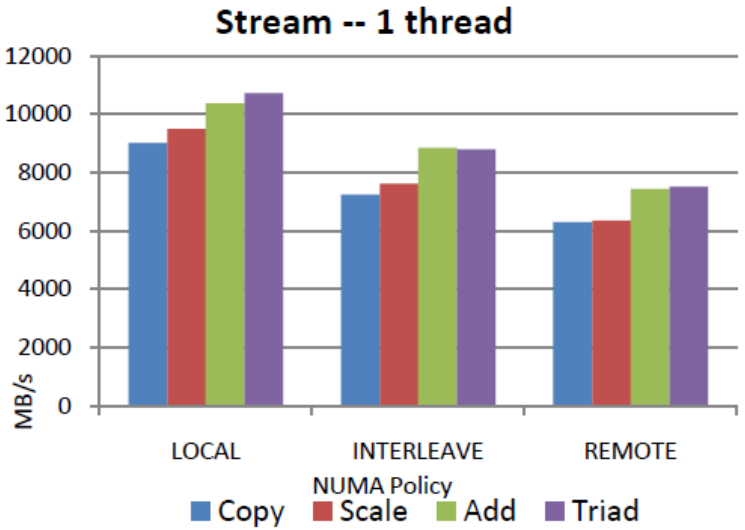


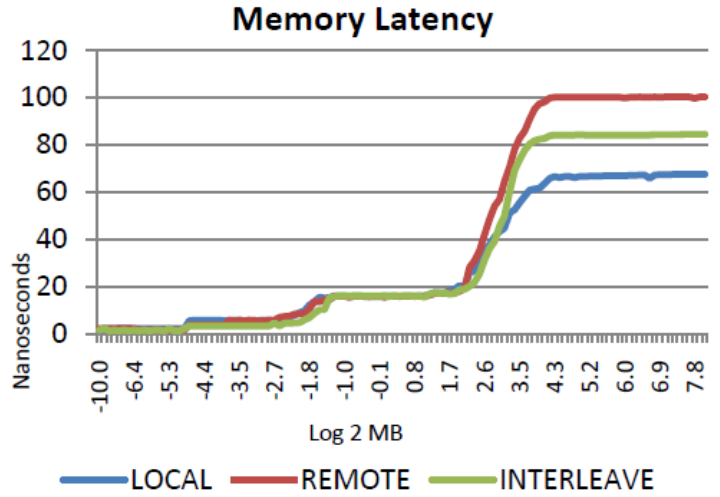
Intel Nehalem based systems with QPI
2-way Xeon 5600 (Westmere) 6-core, 2 IOH

- 4-х процессорная NUMA-система
- Каждый процессор имеет интегрированный контроллер и несколько банков памяти
- Процессоры соединены шиной **Intel QuickPath Interconnect (QPI)** – решения на базе процессоров Intel

Политики управления памятью NUMA-системы

- **Политики управления памятью можно задавать в настройках BIOS/UEFI:**
  - **NUMA Mode** – в системе присутствует несколько NUMA-узлов, у каждого узла имеется своя локальная память (local), операционная система учитывает топологию системы при выделении памяти
  - **Node Interleave** – память циклически выделяется со всех NUMA-узлов (чередование), операционная система “видит” NUMA-систему как SMP-машину

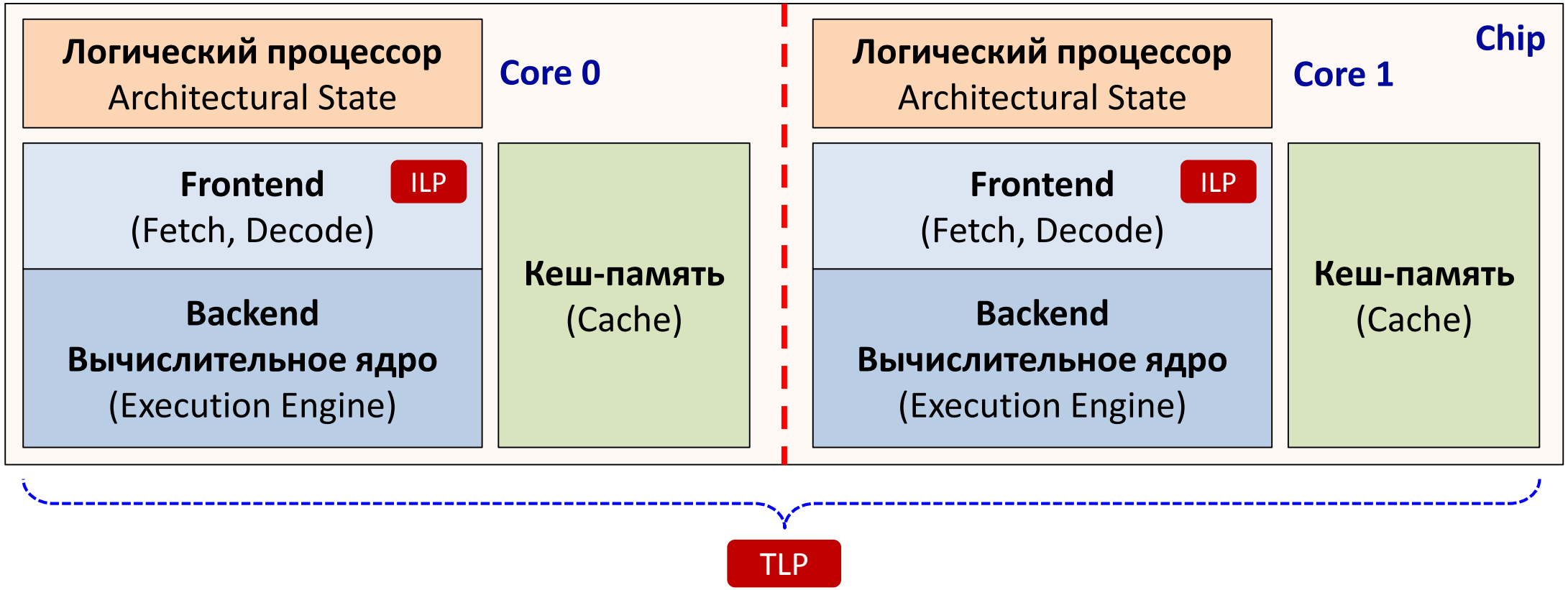




Memory latency and bandwidth accessing local, remote memory for a PowerEdge R610 server (Dual Intel Xeon X5550 Nehalem, 6 x 4GB 1333 MHz RDIMMS)

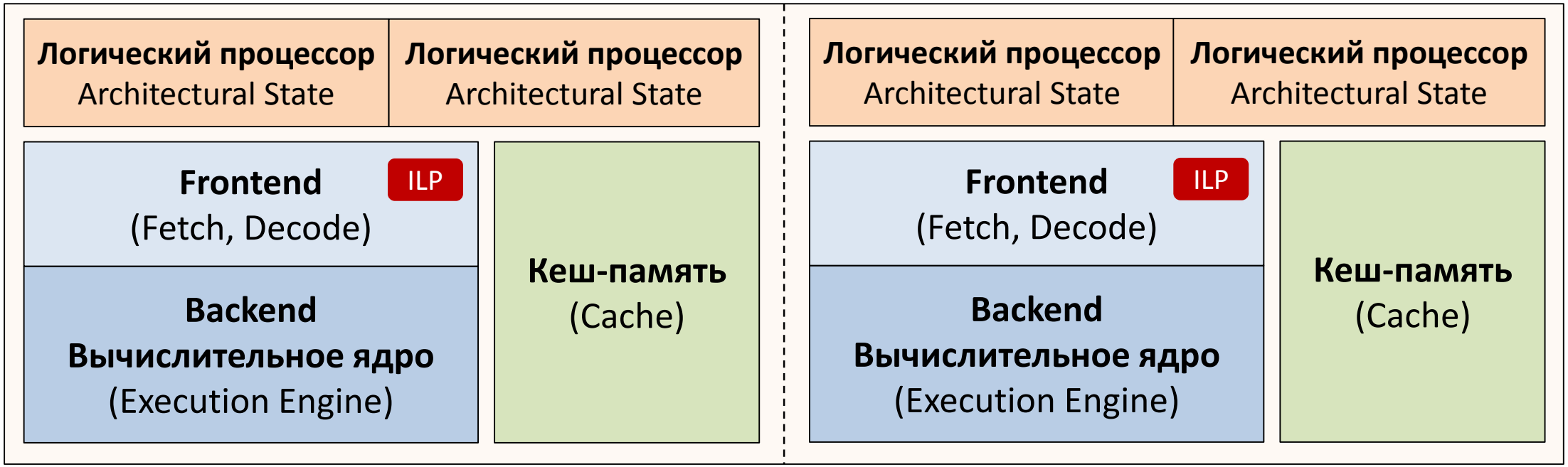
http://i.dell.com/sites/content/business/solutions/whitepapers/ja/Documents/HPC_Dell_11g_BIOS_Options_jp.pdf

Многоядерные процессоры (Multi-core processors)



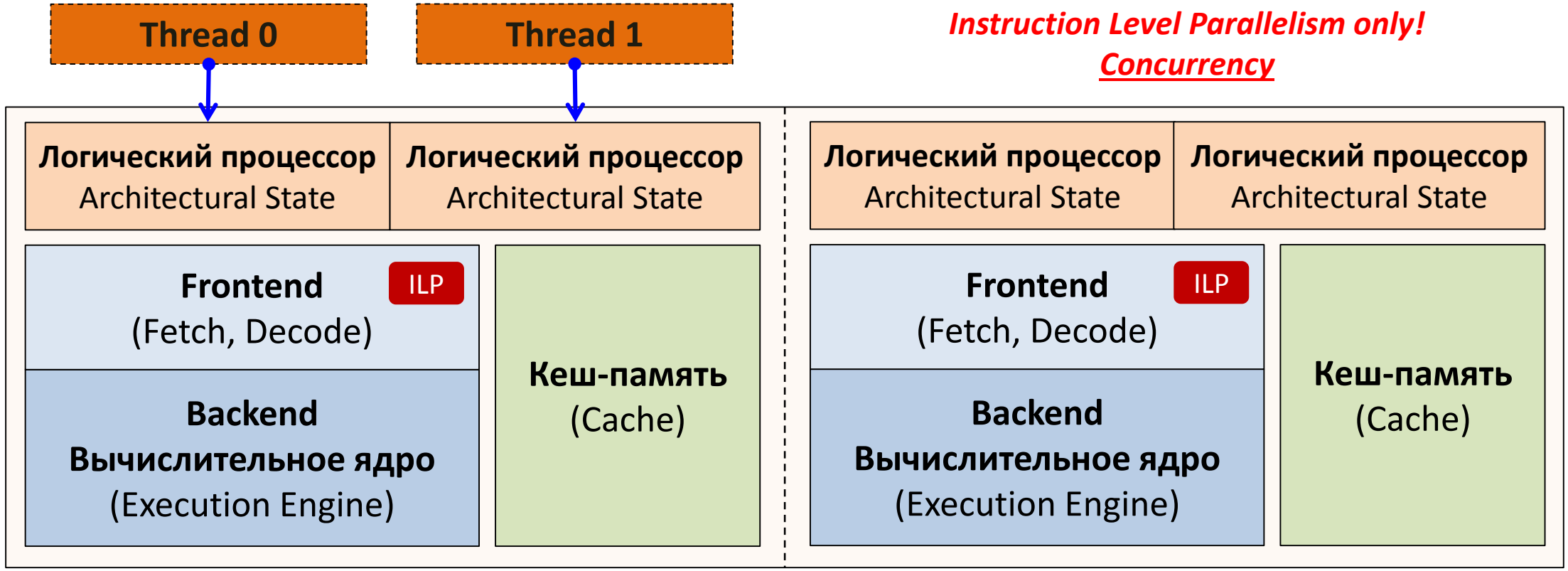
- Процессорные ядра размещены на одном чипе (Processor chip)
- Ядра процессора могу разделять некоторые ресурсы (например, кеш-память)
- Многоядерный процессор реализует параллелизм уровня потоков (Thread level parallelism – TLP)

Многоядерные процессоры с поддержкой SMT



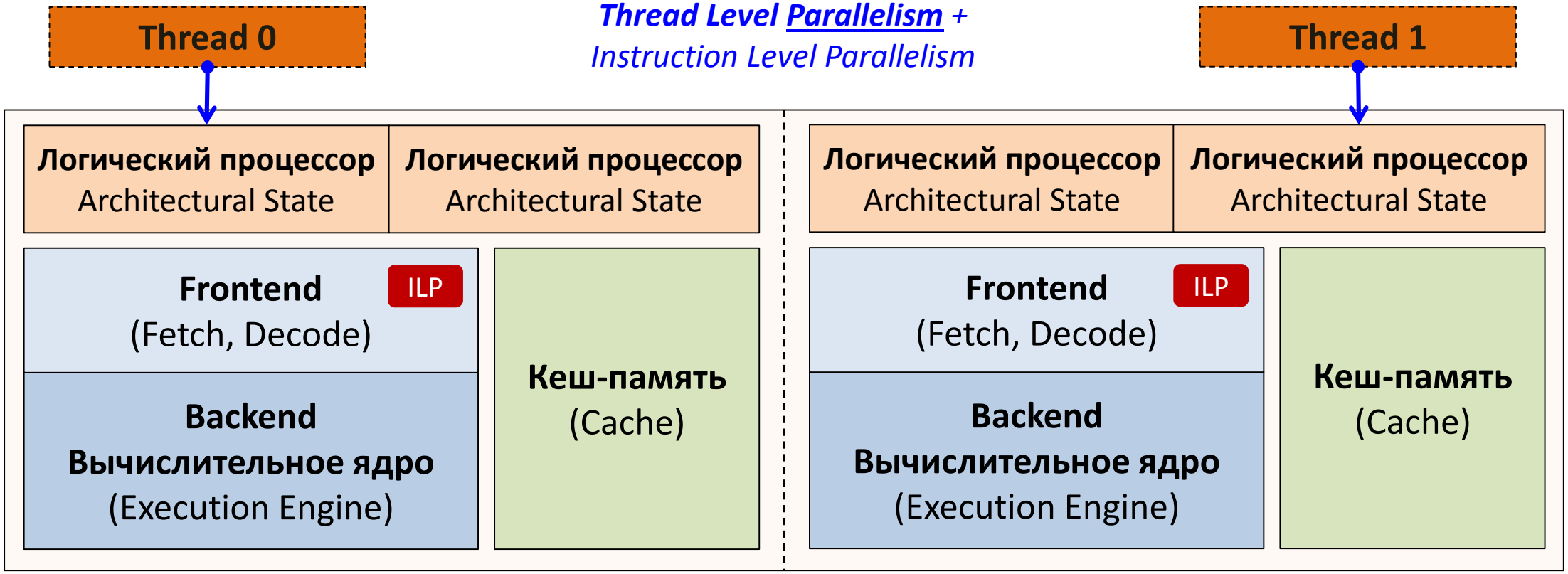
- Многоядерный процессор может поддерживать одновременную многопоточность (Simultaneous multithreading – SMT, Intel Hyper-threading, Fujitsu Vertical Multithreading)
- Каждое ядро может выполнять несколько потоков на своем суперскалярном конвейере (2-way SMT, 4-way SMT, 8-way SMT)
- Операционная система представляет каждый SMT-поток как логический процессор

Многоядерные процессоры с поддержкой SMT



- Операционная система видит 4 логических процессора
- Потоки 0 и 1 выполняются на ресурсах одного ядра – привязаны к логическим процессорам SMT
- Оба потока разделяют ресурсы одного суперскалярного конвейера ядра – конкурируют за ресурсы (только параллелизм уровня инструкций – ILP)

Многоядерные процессоры с поддержкой SMT



- Операционная система видит 4 логических процессора
- Потоки 0 и 1 выполняются на суперскалярных конвейерах разных ядер
- Задействован параллелизм уровня потоков (TLP) и инструкций (ILP)

Современные системы на базе многоядерных процессоров

Apple iPhone 6

- **SoC Apple A8**
  - Dual-core CPU A8 1.4 GHz (64-bit ARMv8-A)
  - Quad-core GPU PowerVR

- **SIMD**: 128-bit wide NEON
- **L1 cache**:
  per core 64 KB L1i, 64 KB L1d
- **L2 cache**: shared 1 MB
- **L3 cache:** 4 MB
- **Technology process**: 20 nm (manufactured by TSMC)

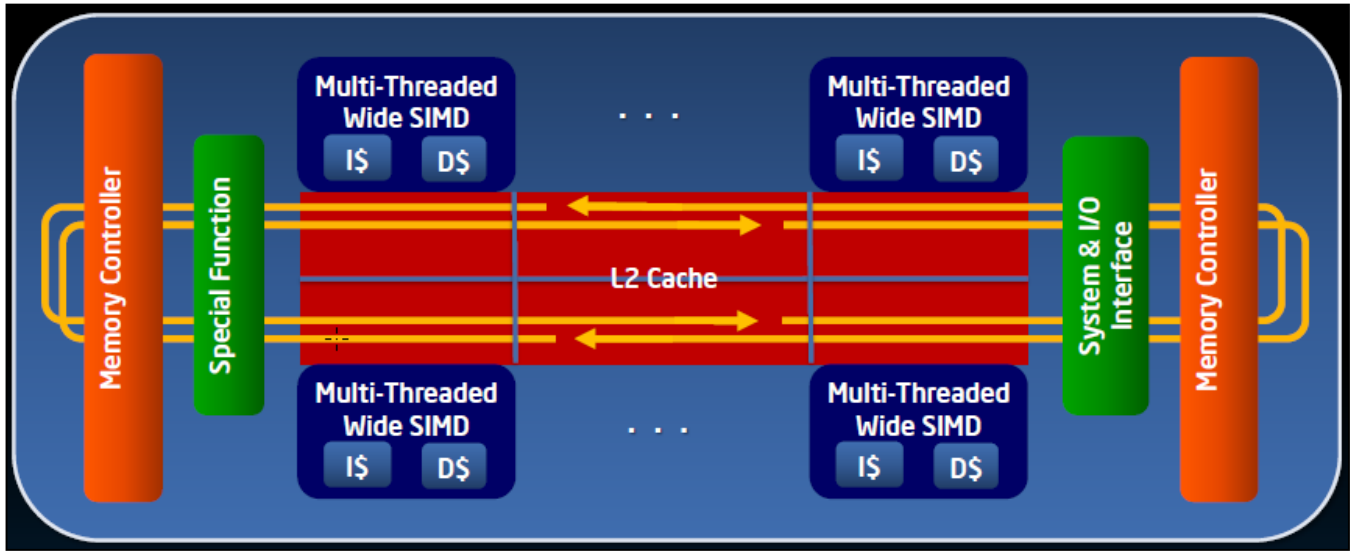
Samsung Galaxy S4 (GT-I9505)

- **Quad-core Qualcomm Snapdragon 600**
  (1.9 GHz with LTE, ARMv7, CPU Krai 300)

- **Конвейер (Pipeline)**: 11 stage integer pipeline
  (3-way decode, 4-way out-of-order speculative issue superscalar)
- **SIMD**: 128-bit wide NEON
- **L0 cache**: 4 KB + 4 KB direct mapped
- **L1 cache**: 16 KB + 16 KB 4-way set associative
- **L2 cache**: 2 MB 8-way set associative
- **Technology process**: 28 nm

http://www.samsung.com/ru/business/business-products/mobile-devices/smartphone/GT-I9505ZRFSER

Специализированные ускорители: Intel Xeon Phi



http://www.intel.ru/content/www/ru/ru/processors/xeon/xeon-phi-detail.html

The Tianhe-2 Xeon Phi drawer in action

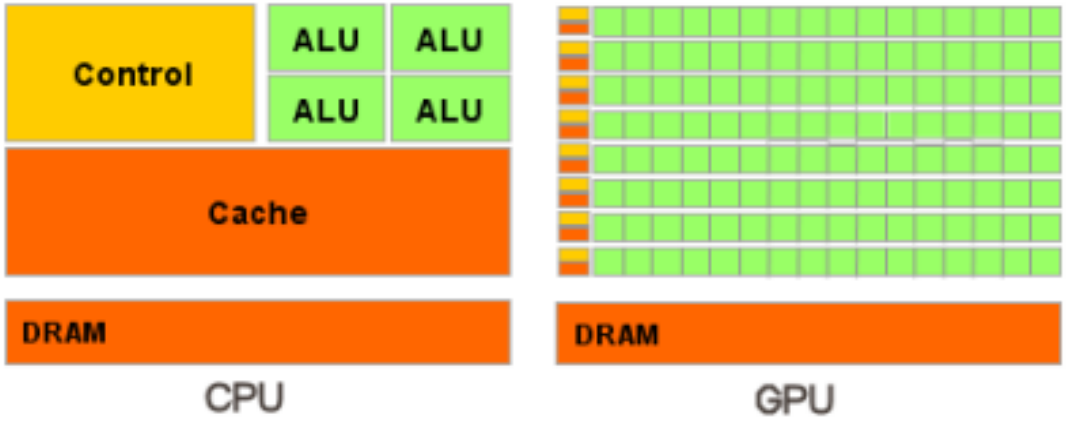
http://www.theregister.co.uk/Print/2013/06/10/inside_chinas_tianhe2_massive_hybrid_supercomputer/

- **Intel Xeon Phi (Intel MIC):** 64 cores Intel P54C (Pentium)
- **Pipeline:** in-order, 4-way SMT, 512-bit SIMD
- Кольцевая шина (1024 бит, ring bus) для связи ядер и контроллера памяти GDDR5
- Устанавливается в PCI Express слот

SMP-система
256 логических процессоров

Специализированные ускорители: GPU – Graphics Processing Unit

- **Graphics Processing Unit** (GPU) – графический процессор, специализированный многопроцессорный ускоритель с общей памятью
- Б*о*льшая часть площади чипа занята элементарными ALU/FPU/Load/Store модулями
- Устройство управления (Control unit) относительно простое по сравнению с CPU



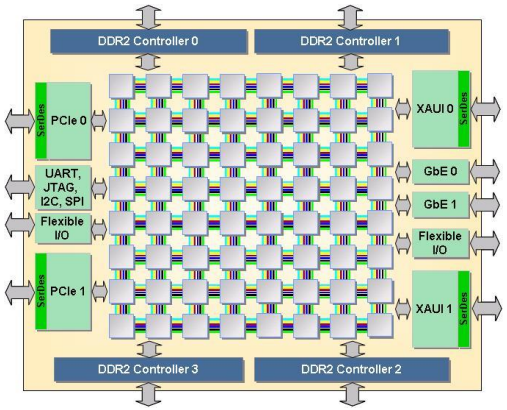
NVIDIA GeForce GTX 780
(Kepler, **2304 cores**, GDDR5 3 GB)

AMD Radeon HD 8970
(**2048 cores**, GDDR5 3 GB)

Специализированные многоядерные процессоры

Sony Playstation 3
IBM Cell
(2-way SMT PowerPC core + 6 SPE)

Microsoft XBox 360
IBM Xenon
(3 cores with 2-way SMT)

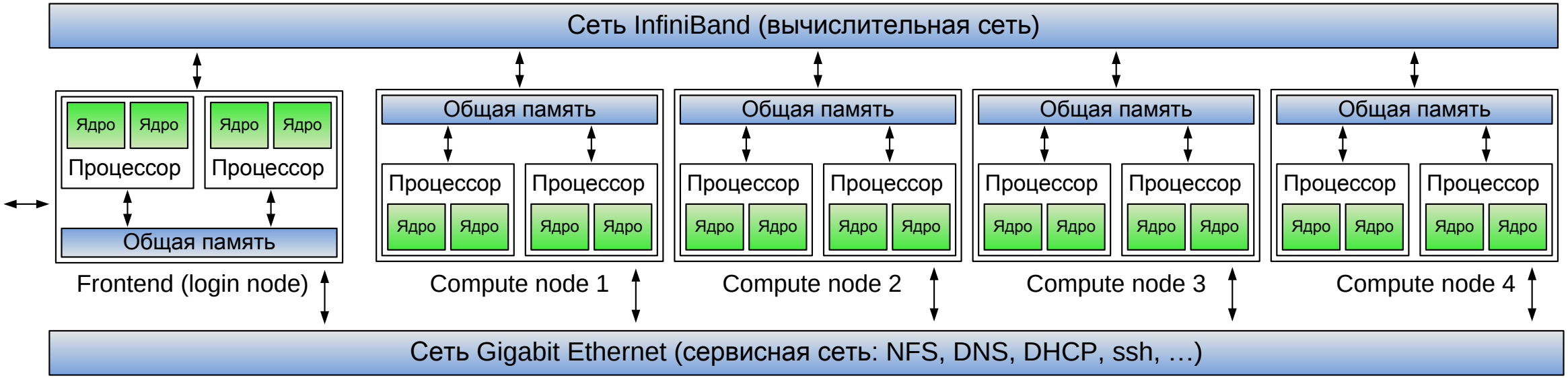


Tilera TILEPro64
(64 cores, VLIW, mesh)

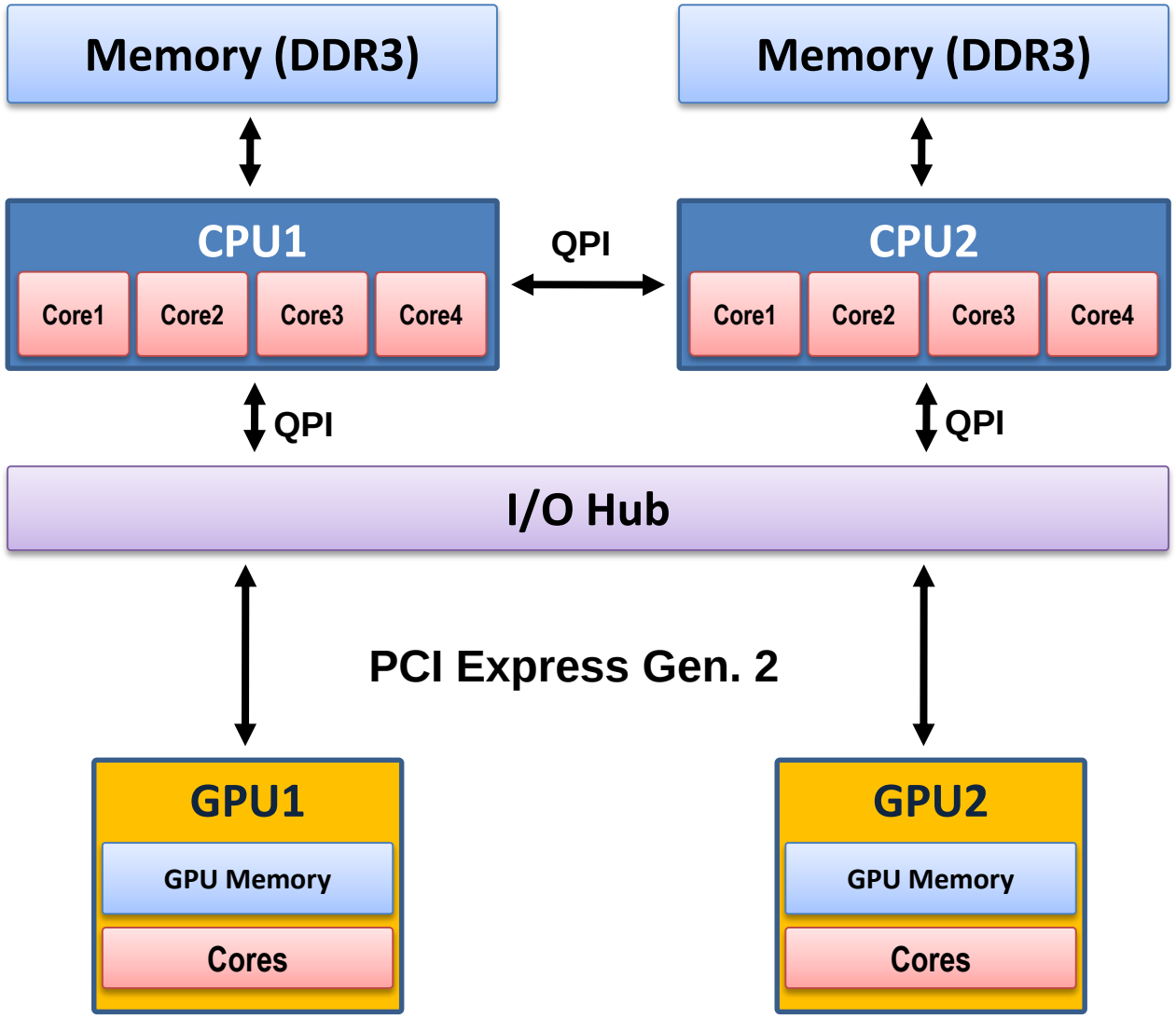
Cisco Routers
MIPS
Multi-core processors

Распределенные вычислительные системы (Distributed Computer Systems)

Вычислительные кластеры



Гибридные вычислительные узлы



Цели оптимизации программного обеспечения

- Минимизация времени выполнения
- Минимизация объема потребляемой памяти
- Минимизация энергопотребления процессором/вычислительным узлом

Этапы оптимизации кода

1. **Алгоритмическая оптимизация кода**

2. **Распараллеливание** – *Thread Level Parallelism*
   (OpenMP/Pthreads, CUDA/OpenCL/OpenACC, MPI)

3. **Оптимизация доступа к памяти**
   (кеш-памяти, ОЗУ, дисковой, сетевым хранилищам)

4. **Векторизация кода** – *Data Parallelism*
   (SSE, AVX, AltiVec, ARM SIMD)

5. **Оптимизация ветвлений, оптимизация загрузки конвейеров** – *Instruction Level Parallelism*

Оптимизация кода

1. Осуществляется **профилирование кода** (проблемных участков) и **выявляются “горячие точки”** (hot spots) – участки программы, на которые приходится больше всего времени выполнения

2. **Горячие точки сортируются** в порядке не убывания времени выполнения – формируется приоритетный список участков программы для последующей оптимизации

3. **Каждая горячая точка оптимизируется**

4. **После оптимизации каждого “хот спота”** проводится **тестирование программы** – проверяется совпадение результатов её работы до модификации и после внесения изменений

Оптимизирующие компиляторы

- **GNU GCC**

```
gcc -march=core2 -O2
```

- **Intel ICC**

```
icc –xT -O2
```

- **PathScale Compiler**

```
pathcc -march=core -O2
```

- **Oracle Solaris Studio**

```
cc –xchip=core2 -O2
```

Profile-Guided Optimization (PGO)

1. **Generate profile**

```
icc -prof-gen -o myprog ./myprog.c
./myprog
```

2. **Use profile**

```
icc -prof-use -o myprogopt ./myprog.c
./myprogopt
```

Профилировщики

- **Intel VTune Amplifier XE**
- **Linux perf**
- AMD CodeAnalyst
- Visual Studio Team System Profiler
- Oprofile
- GNU gprof
- Valgrind – выполняет программу на виртуальной машине

Режимы работы профилировщиков

- **Семплирование (Sampling)** – профилировщик периодически снимает информацию о состоянии программы (значения счетчиков производительности процессора, значение счетчика команд, …) – формирует статистику

- На результаты семплирования значительно влияют другие программы запущенные в системе

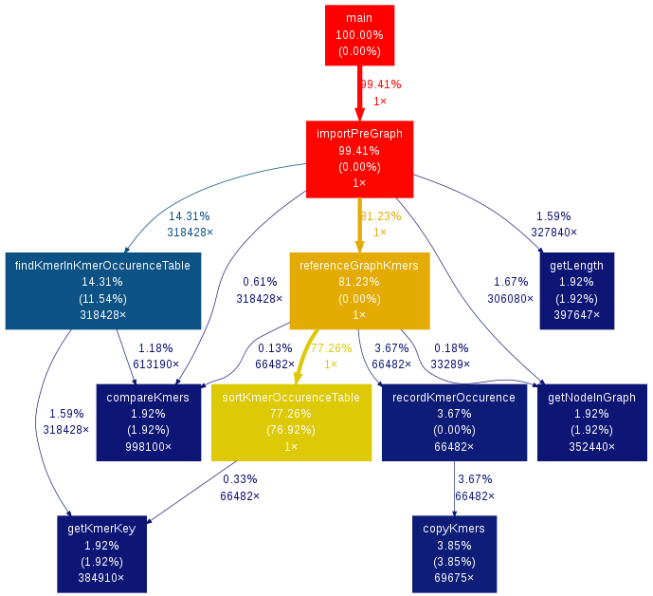
- При семплировании желательно, что бы исследуемая программа работала относительно длительный промежуток времени (профилировщик должен успеть набрать статистику) и в системе было минимум запущенных программ

Linux perf, Oprofile, Intel Vtune, AMD CodeAnalyst, …

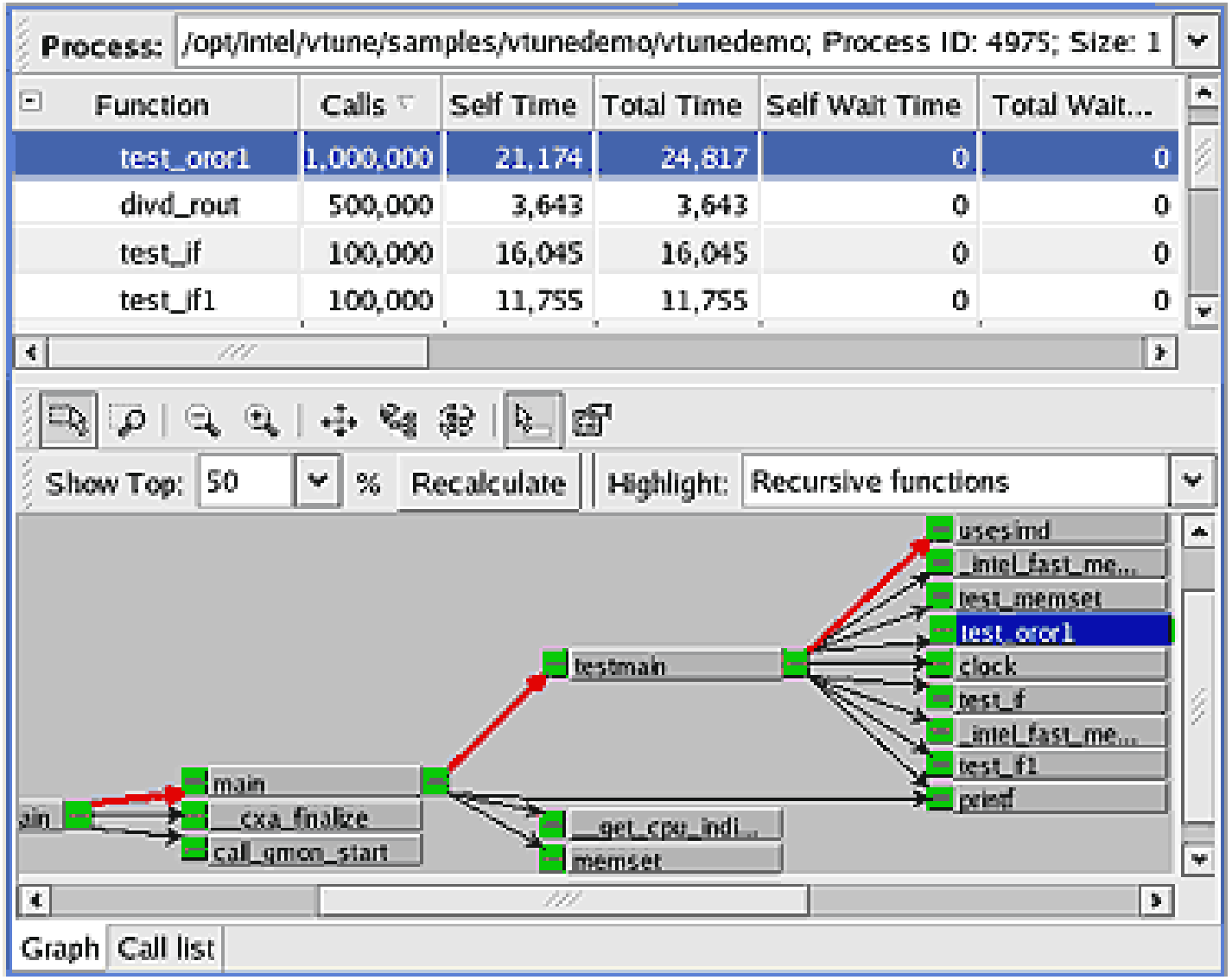
Режимы работы профилировщиков

- **Инструментация (Instrumentation)** – внедрение в программу кода для перехвата интересующих пользователя событий (например, вызова функций)
- Инструментация позволяет формировать граф вызовов программы, подробную информацию о времени выполнения функций и др.

GNU gprof, Intel VTune, Valgrind, AMD CodeAnalyst, …



Intel Vtune


Process: /opt/intel/vtune/samples/vtunedemo/vtunedemo; Process ID: 4975; Size: 1
Function
Calls
Self Time
Total Time
Self Wait Time
Total Wait...
test_oror1 1,000,000 21,174 24,817 0 0
divd_rout 500,000 3,643 3,643 0 0
test_if 100,000 16,045 16,045 0 0
test_if1 100,000 11,755 11,755 0 0
Show Top: 50 % Recalculate
Highlight: Recursive functions
usesimd
_intel_fast_me...
test_memset
test_oror1
clock
test_f
_intel_fast_me...
test_f1
printf
testmain
main
__cxa_finalize
__gmon_start
__get_cpu_indi...
memset
Graph
Call list

CPU Dispatching

```
#include <mmintrin.h>

__declspec(cpu_specific(pentium))
void array_sum(int *r, int const *a, int *b, size_t l))
{
   for (; length > 0; l--) *result++ = *a++ + *b++;
}

__declspec(cpu_specific(pentium_MMX))
void array_sum(int *r, int const *a, int *b, size_t l)
{
    /* Code for MMX */
}

__declspec(cpu_dispatch(pentium, pentium_MMX))
void array_sum(int *r, int const *a, int *b, size_t l))
{

}
```

Измерение времени выполнения кода

Измерение времени выполнения кода (GNU/Linux)

- `int gettimeofday(struct timeval *tv, struct timezone *tz)`
- `int clock_gettime(clockid_t clk_id, struct timespec *tp)`
- `clock_t clock(void)`
- `time_t time(time_t *t);`
- CPU Time Stamp Counter (TSC, 64-bit MSR register)

Функция gettimeofday()

```c
#include <sys/time.h>

int main()
{
    struct timeval t1, t2;
    double t;

    gettimeofday(&t1, NULL);
    /* Measured code */
    gettimeofday(&t2, NULL);

    t = (t2.tv_sec * 1E6 + t2.tv_usec -
         t1.tv_sec * 1E6 - t1.tv_usec) * 1E-6;

    printf("Elapsed time: %.6f sec.\n", t);

    return 0;
}
```

Измерение времени выполнения кода

- Имеется фрагмент кода (функция), требуется измерить продолжительность его выполнения
- Продолжительность выполнения кода (далее, время) может быть выражена в секундах, тактах процессора/системной шины и пр.

```
t0 = get_time();
MEASURED_CODE();
t1 = get_time();

elapsed_time = t1 - t0;
```

- Результаты измерений должны быть воспроизводимыми (повторный запуск измерений должен давать такие же результаты или близкие к ним)
- Без воспроизводимости невозможно осуществлять оптимизацию кода: как понять, что стало причиной сокращения времени выполнения кода - оптимизация или это ошибка измерений?

Измерение времени выполнения кода

- Следует выбирать самый точный таймер имеющийся в системе:
  RTC, HPET, TSC, …
- Необходимо учитывать эффекты кеш-памяти и возможно игнорировать первый запуск теста (прогревочный, warm-up) – не включать его в итоговый результат
- Измерения желательно проводить несколько раз и вычислять оценку мат. ожидания (среднего значения) измеряемой величины (например, времени выполнения) и дисперсию
- Запуск тестов желательно осуществлять всегда в одних и тех же условиях – набор запущенных программ, аппаратная конфигурация, …

Методика измерения времени выполнения кода

1. **Готовим систему к проведению измерений**
   - настраиваем аппаратные подсистемы (настройки BIOS)
   - параметры операционной системы и процесса, в котором будут осуществляться измерения

2. **Выполняем разогревочный вызов измеряемого кода (Warmup)**
   - Регистрируем время выполнения первого вызова и не учитываем его в общей статистике (при первом вызове может осуществляться отложенная инициализация и пр.)

3. **Выполняем многократные запуски и собираем статистику о времени выполнения**
   - Каждый запуск должен осуществляться в одних и тех же условиях (входные массивы заполнены одними и теми же данными, входные данные отсутствуют/присутствуют в кеш-памяти процессора, …)
   - Выполняем измерения пока:
     - ✓ относительная стандартная ошибка среднего времени выполнения (RSE) больше 5% [опционально]
     - ✓ число выполненных измерений меньше максимально допустимого

Методика измерения времени выполнения кода

4. **Проводим статистическую обработку результатов измерений**
   - Находим и **отбрасываем промахи измерений** (выбросы, outliers):
     например, 25% минимальных и максимальных значений результатов измерений [опционально]
   - Вычисляем оценку **математического ожидания** времени выполнения (mean)
     (медиану [опционально])
   - Вычисляем несмещенную оценку **дисперсии** времени выполнения
     (unbiased sample variance - Var)
   - Вычисляем **стандартное отклонение** (corrected sample standard deviation - StdDev)
   - Вычисляем **стандартную ошибку среднего** времени выполнения
     (standard error of the mean - StdErr)
   - Вычисляем **относительную стандартную ошибку среднего** времени выполнения
     (relative standard error of the mean - RSE)
   - Строим доверительные интервалы (confidence interval)

Элементарная обработка результатов измерений

- **Математическое ожидание** времени выполнения (Mean)

$$\bar{t} = \frac{t_1 + t_2 + \cdots + t_n}{n}$$

- **Несмещенная оценка дисперсии** времени выполнения (Unbiased sample variance - Var)

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(t_i - \bar{t}) = \frac{1}{n(n-1)}(n\sum_{i=1}^{n}t_i^2 - (\sum_{i=1}^{n}t_i)^2)$$

- **Стандартное отклонение** (Corrected sample standard deviation - StdDev)

$$s = \sqrt{s^2}$$

- **Стандартная ошибка среднего** времени выполнения (Standard error of the mean - StdErr)
- **Относительная стандартная ошибка среднего** времени выполнения (relative standard error of the mean - RSE)

$$\mathrm{StdErr} = \frac{s}{\sqrt{n}} \qquad \mathrm{RSE}(\bar{t}) = \frac{\mathrm{StdErr}}{\bar{t}} \cdot 100\%$$

Элементарная обработка результатов измерений

- **RSE** показывает на сколько близко вычисленное среднее время выполнения к истинному среднему времени выполнения (среднему генеральной совокупности)
  - На практике хорошая точность RSE <= 5%

- **Оценка погрешности при большом числе измерений**

$$t = \bar{t} \pm 3s$$

  - С вероятностью 0.997 время выполнения лежит в интервале $(\bar{t} - 3s; \bar{t} + 3s)$
  - С вероятностью 0.955 время выполнения лежит в интервале $(\bar{t} - 2s; \bar{t} + 2s)$
  - С вероятностью 0.683 время выполнения лежит в интервале $(\bar{t} - s; \bar{t} + s)$

- **Оценка погрешности при малом числе измерений (*n* < 10)**
  - Задаем требуемый уровень α доверительной вероятности (0.95; 0.99), из таблицы берем значение коэффициента $t_{\alpha,n}$ Стьюдента

$$t = \bar{t} \pm t_{\alpha,n} \cdot \mathrm{StdErr}$$

Вычисление стандартного отклонения (StdEv, StdDev)

- **Стандартный подход:** s^2 может быть < 0, ошибка при вычислении корня

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(t_i - \bar{t}) = \frac{1}{n(n-1)}(n\sum_{i=1}^{n}t_i^2 - (\sum_{i=1}^{n}t_i)^2) \quad \Rightarrow \quad s = \sqrt{s^2}$$

- **Метод B.P. Welford'a**
  Knuth D. The Art of Computer Programming, Vol. 2, 3ed. (p. 232)

$$\begin{aligned} M_1 &= x_1, \qquad M_k = M_{k-1} \oplus (x_k \ominus M_{k-1}) \oslash k, \\ S_1 &= 0, \qquad S_k = S_{k-1} \oplus (x_k \ominus M_{k-1}) \otimes (x_k \ominus M_k), \end{aligned} \quad \Rightarrow \quad \sigma = \sqrt{S_n/(n-1)}.$$

Домашнее чтение

- **How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures** // Intel White Paper, 2010
  http://www.intel.ru/content/www/ru/ru/intelligent-systems/embedded-systems-training/ia-32-ia-64-benchmark-code-execution-paper.html

- **Использование Time-Stamp Counter для измерения времени выполнения кода на процессорах с архитектурой Intel 64** //
  http://www.mkurnosov.net/uploads/Main/mkurnosov-rdtsc-2014.pdf