

Многопоточное программирование (часть 1)

Михаил Георгиевич Курносов

Email: mkurnosov@gmail.com

WWW: <http://www.mkurnosov.net>

Курс «Параллельные и распределённые вычисления»

Школа анализа данных Яндекс (Новосибирск)

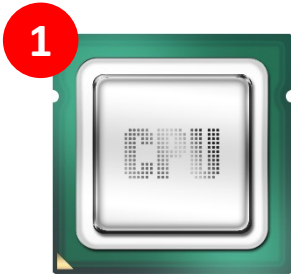
Весенний семестр, 2015

Содержание

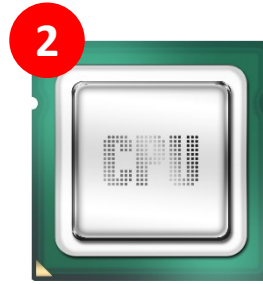
- Многоядерные процессоры и многопроцессорные вычислительные системы (ВС)
- Цели и задачи создания многопоточных программ
- Процессы и потоки
- Многопоточное программирование на C++11

Архитектура многопроцессорных вычислительных систем с общей памятью

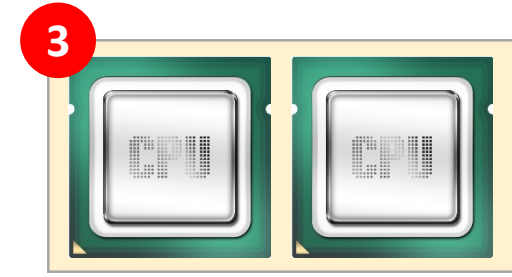
Архитектура многопроцессорных вычислительных систем



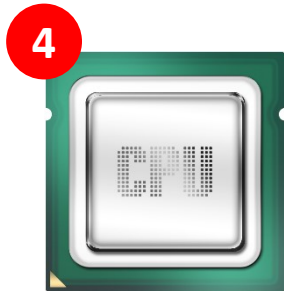
Одноядерный процессор
Параллелизм уровня
инструкций (ILP)



**Одноядерный процессор
с поддержкой аппаратной
многопоточности**
(Intel HyperThreading, параллелизм
уровня инструкций)



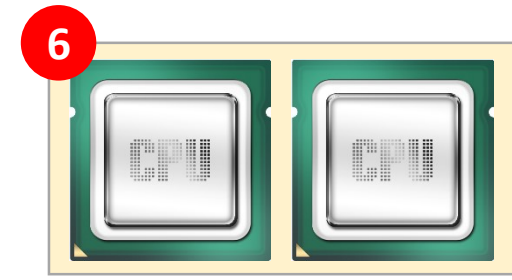
**Многопроцессорные
SMP/NUMA-системы**
Параллелизм уровня потоков (TLP)



Многоядерные процессоры
Параллелизм уровня потоков (TLP)

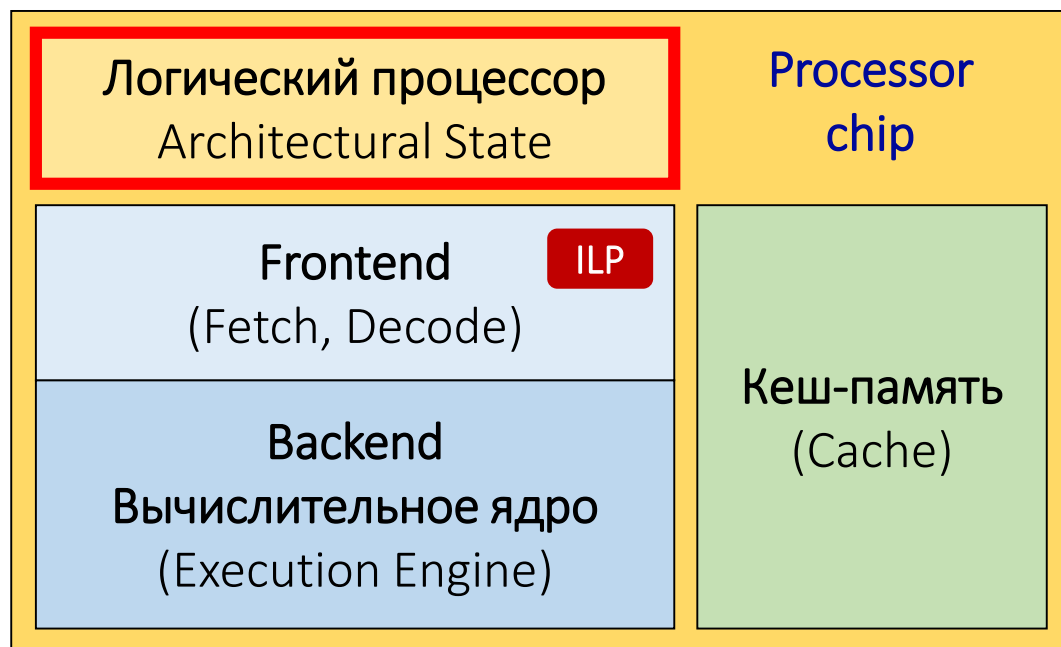


**Многоядерные процессоры
с поддержкой аппаратной
многопоточности**
Параллелизм уровня потоков (TLP)



**Современные
SMP/NUMA-системы**
Параллелизм уровня потоков (TLP)

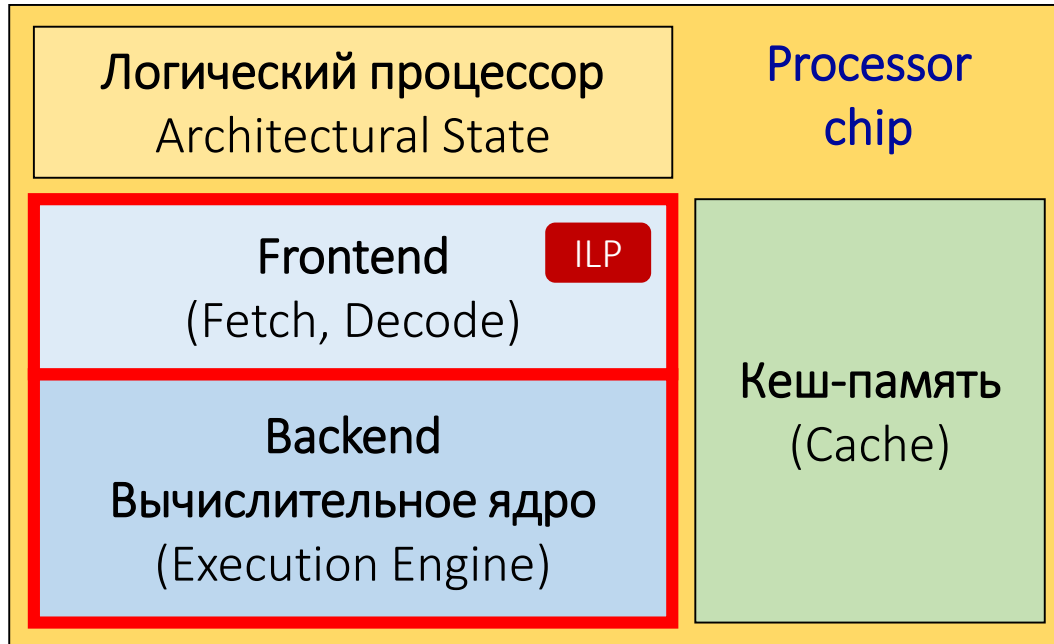
Архитектура ядра процессора Intel 64



- ❑ Intel64 and IA-32 Architectures Software Developer Manuals // <http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>

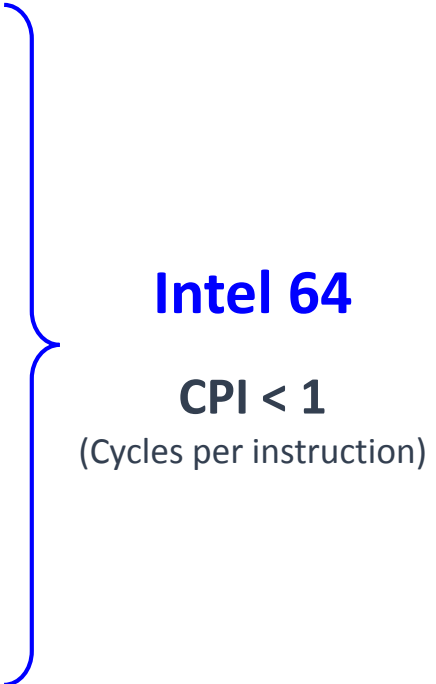
- **Логический процессор (Logical processor)** представлен *архитектурным состоянием* и контроллером прерываний (Interrupt controller, APIC)
- **Архитектурное состояние (Architectural state, AS)** включает:
 - ❑ регистры общего назначения (RAX, RBX, ...)
 - ❑ сегментные регистры (CS, DS, ...),
 - ❑ управляющие регистры (RFLAGS, RIP, GDTR, ...)
 - ❑ X87 FPU-регистры, MMX/XMM/YMM-регистры
 - ❑ MSR-регистры (time stamp counter, ...)
- **Логический процессор** – это то, что “видит” операционная система

Архитектура ядра процессора Intel 64

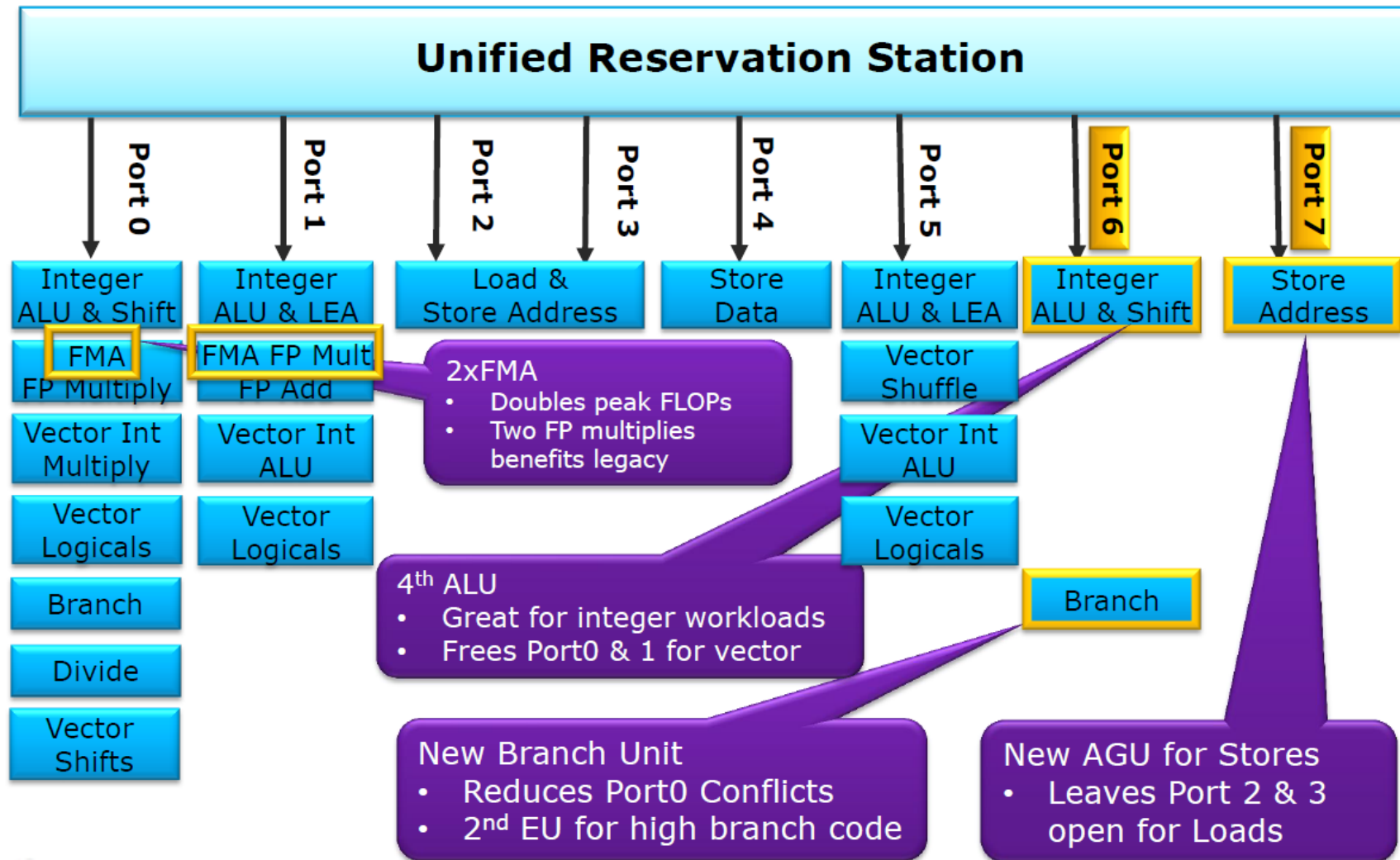


- **Логический процессор** использует ресурсы вычислительного ядра (Execution engine)
- **Frontend** реализует выборку, декодирование инструкций, поддерживает очередь для передачи инструкций в Backend
- **Backend** – это вычислительное ядро; его динамический планировщик распределяет инструкции по исполняющим устройствам: ALU, FPU, Load/Store
- **Backend** реализует *параллельное выполнение инструкций* (Instruction level parallelism – **ILP**)

Параллелизм уровня инструкций (Instruction level parallelism – ILP)

- **Суперскалярный конвейер** (Superscalar pipeline) – исполняющие модули конвейера присутствуют в нескольких экземплярах (несколько ALU, FPU, Load/Store-модулей)
 - **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 – data flow-архитектура)
 - **SIMD-инструкции** – модули ALU, FPU, Load/Store поддерживают операции над векторами (наборы инструкций SSE, AVX, AltiVec, NEON SIMD)
 - **VLIW-архитектура** (Very Long Instruction Word) – процессор с широким командным словом оперирует с инструкциями, содержащими в себе несколько команд, которые можно выполнять параллельно на ALU/FPU/Load-Store (Intel Itanium, Transmeta Efficeon, Texas Instruments TMS320C6x, ЗАО “МЦСТ” Эльбрус)
- 
- Intel 64**
CPI < 1
(Cycles per instruction)

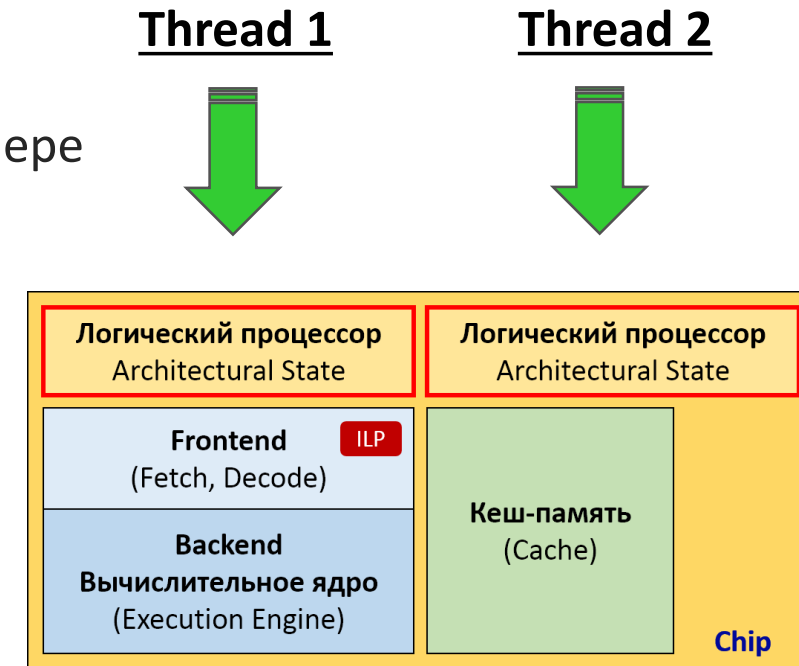
Intel Haswell Execution Engine



<http://arstechnica.com/gadgets/2013/05/a-look-at-haswell/2/>

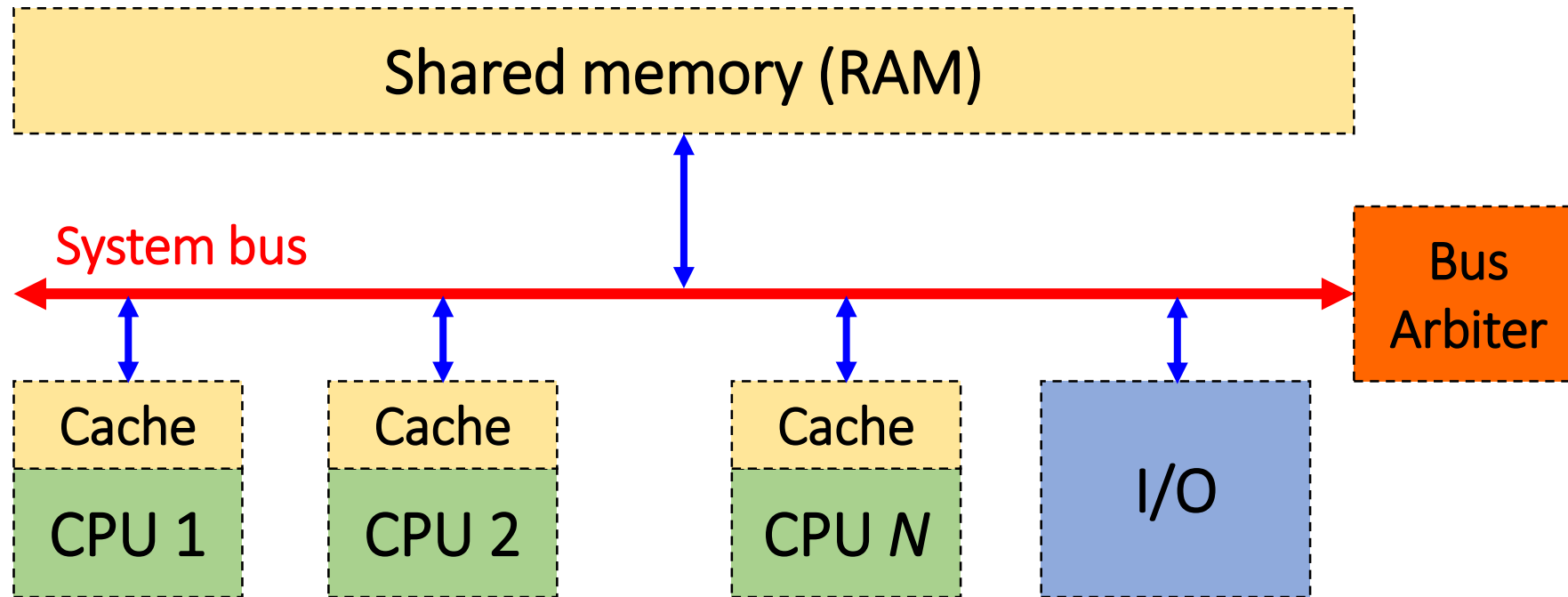
Одновременная многопоточность (Simultaneous multithreading)

- **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**

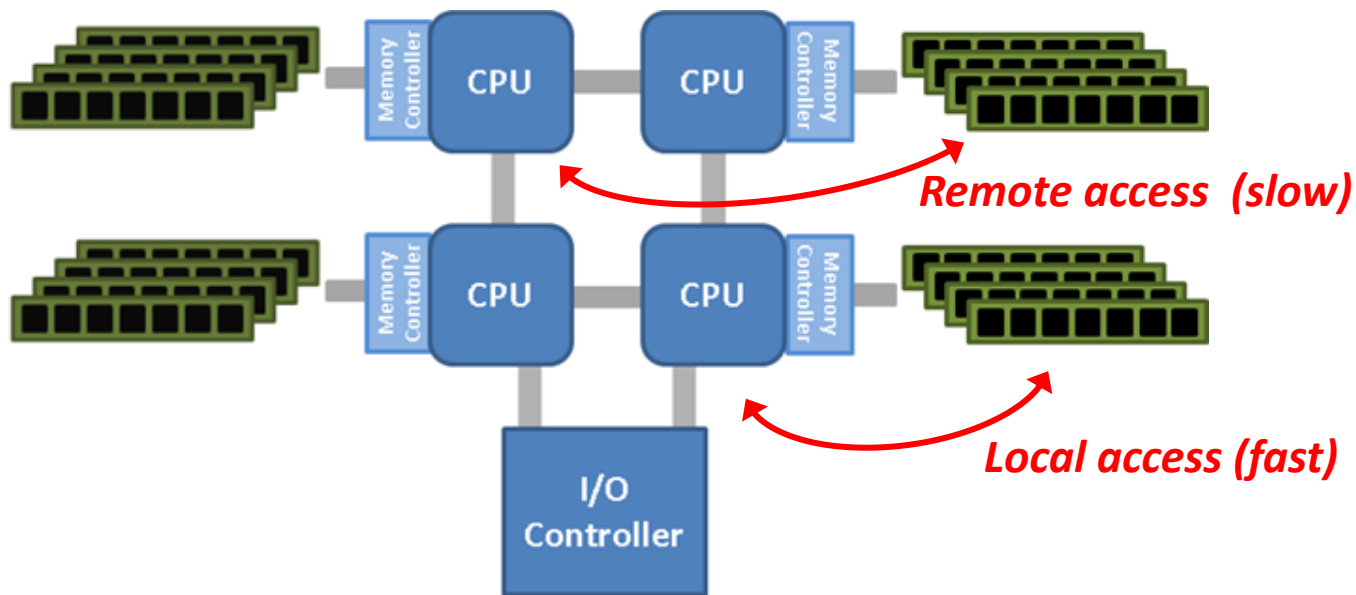
Многопроцессорные SMP-системы (Symmetric multiprocessing)



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

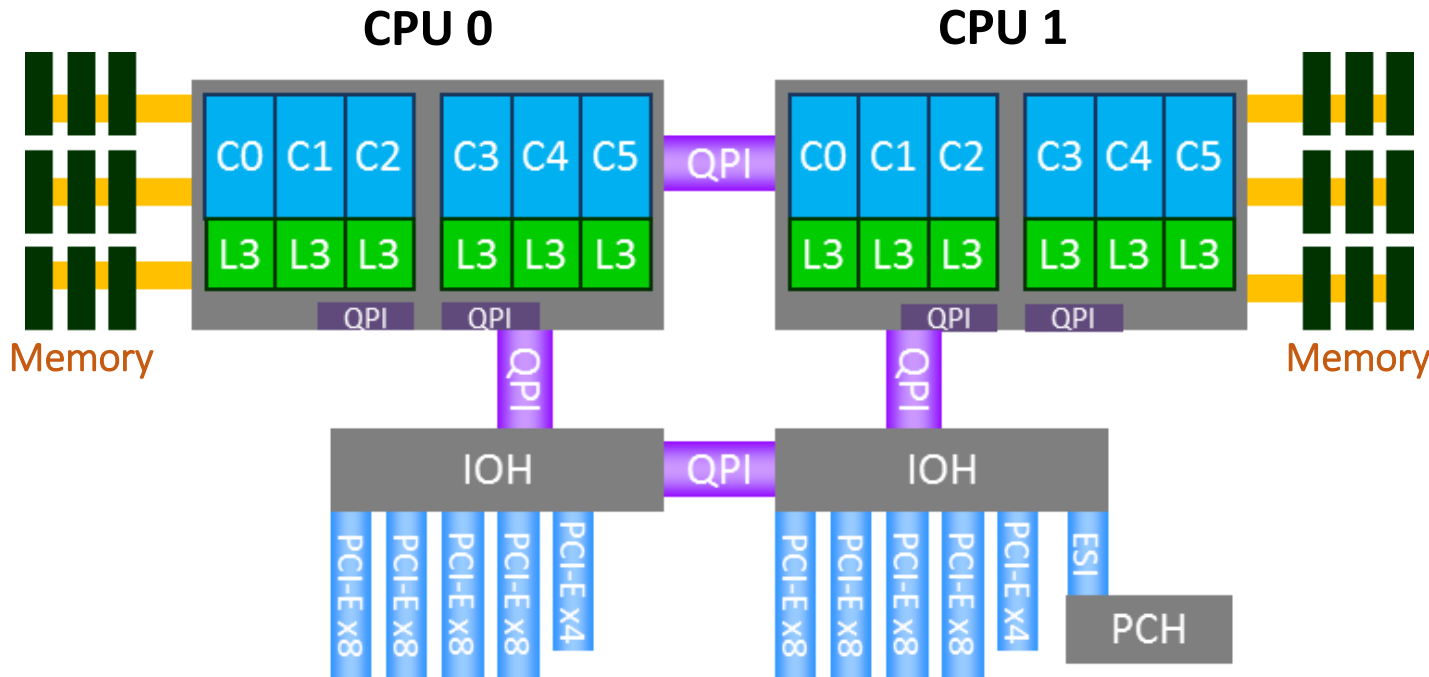
Многопроцессорные NUMA-системы (AMD)

- **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.)

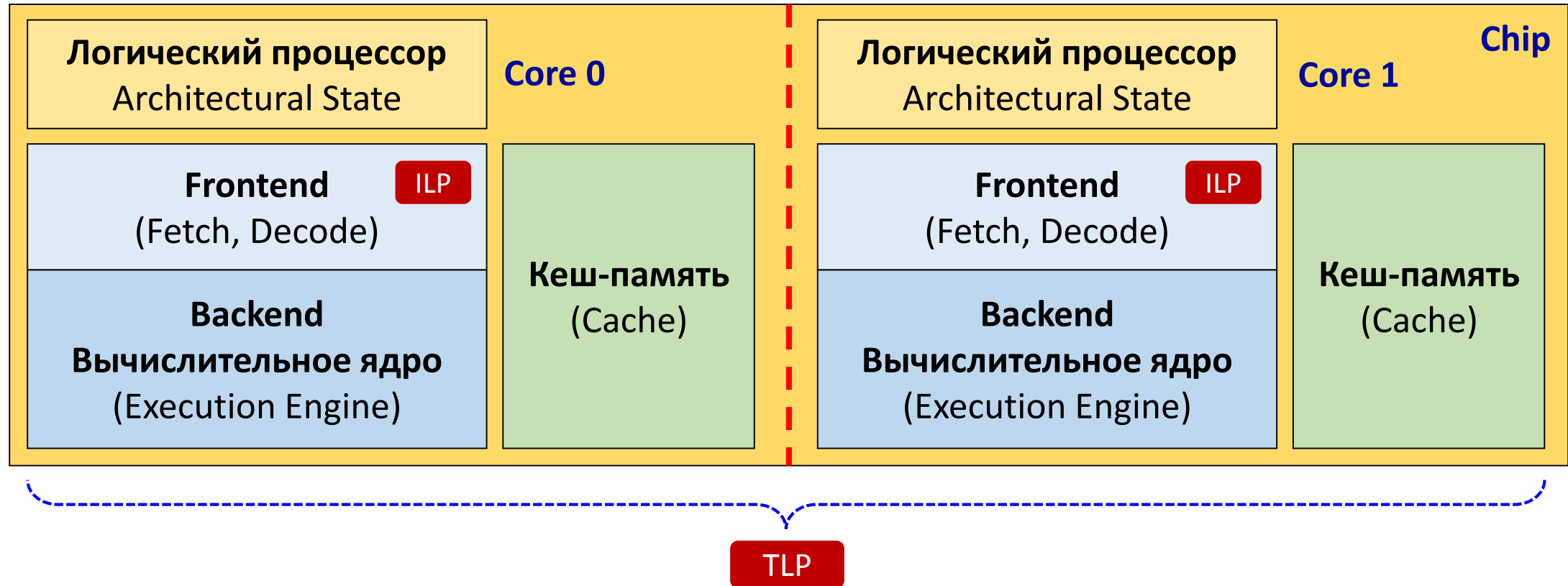
Многопроцессорные NUMA-системы (Intel)



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

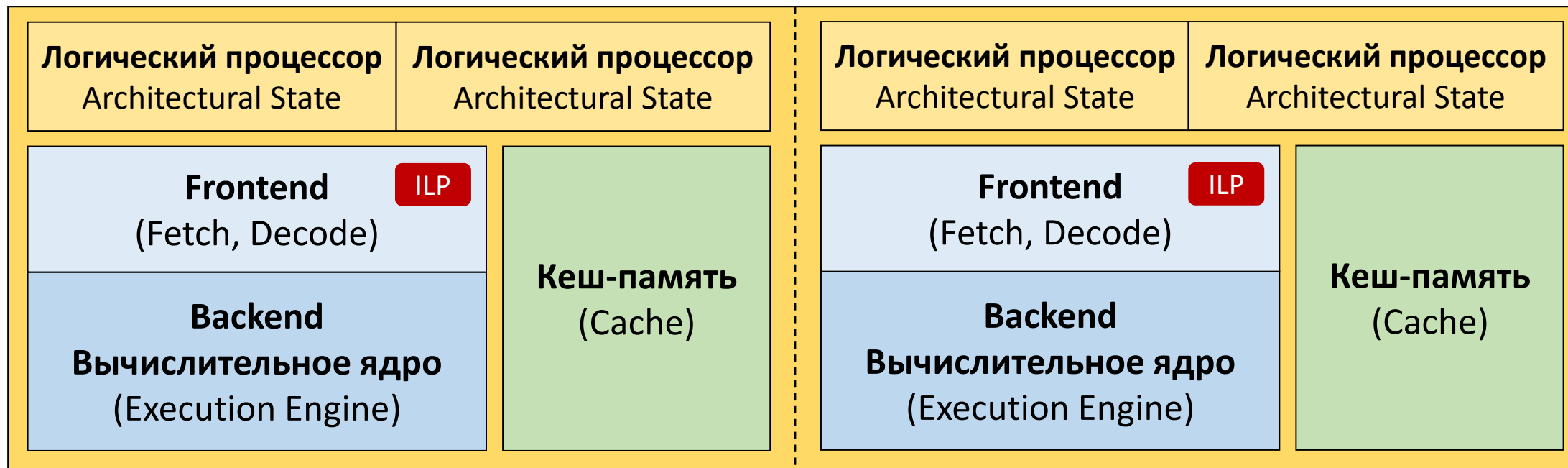
- 4-х процессорная NUMA-система
- Каждый процессор имеет интегрированный контроллер и несколько банков памяти
- Процессоры соединены шиной **Intel QuickPath Interconnect (QPI)** – решения на базе процессоров Intel

Многоядерные процессоры (Multi-core processors)



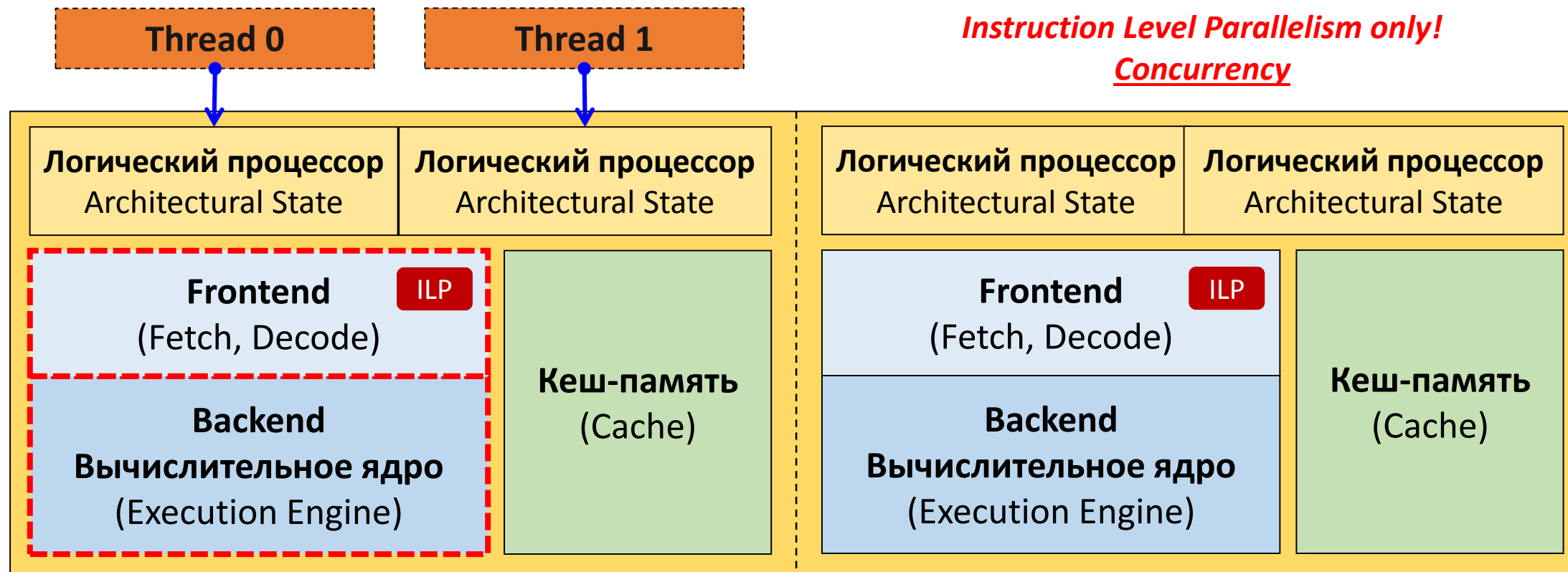
- Процессорные ядра размещены на одном чипе (Processor chip)
- Ядра процессора могут разделять некоторые ресурсы (например, кеш-память)
- Многоядерный процессор реализует параллелизм уровня потоков (Thread level parallelism – TLP)

Многоядерные процессоры с поддержкой SMT



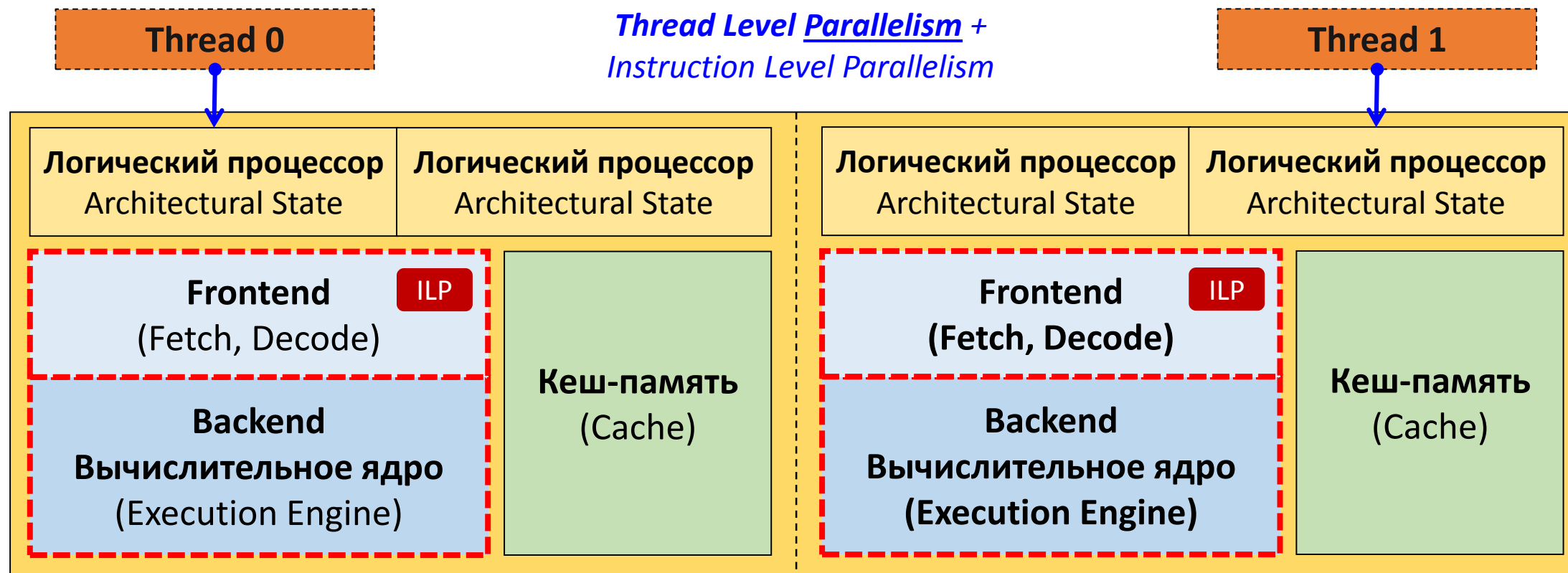
- Многоядерный процессор может поддерживать одновременную многопоточность (Simultaneous multithreading – SMT, Intel Hyper-threading, Fujitsu Vertical Multithreading)
- Каждое ядро может выполнять несколько потоков на своем суперскалярном конвейере (2-way SMT, 4-way SMT, 8-way SMT)
- Операционная система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

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)

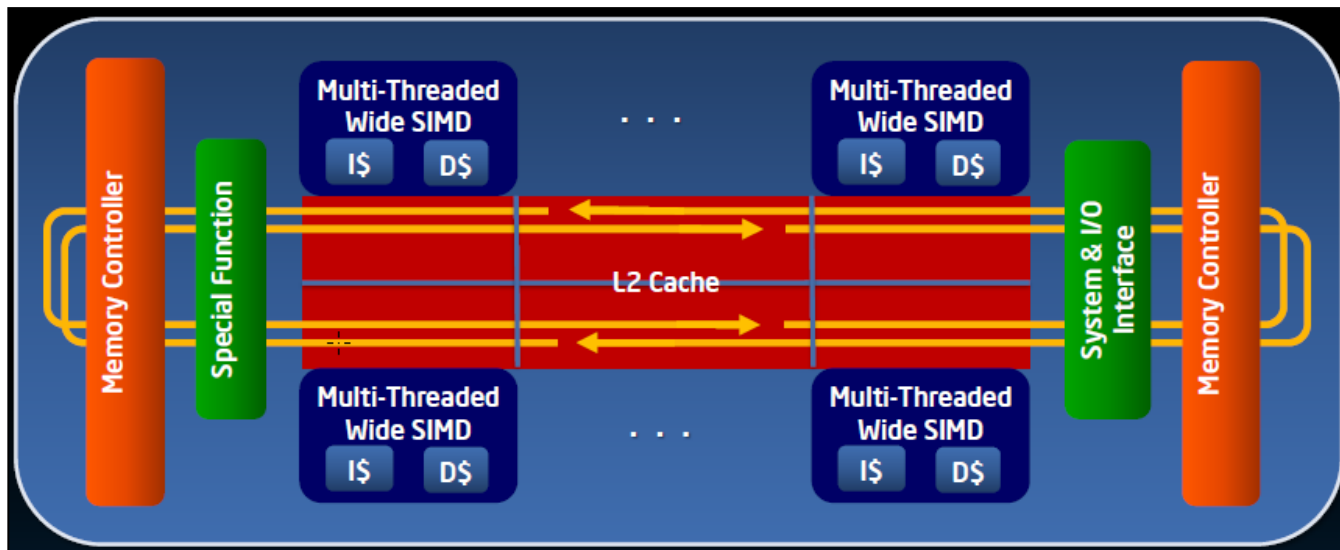


Samsung Galaxy S4 (GT-I9505)

- **Quad-core Qualcomm Snapdragon 600**
(1.9 GHz with LTE, ARMv7, CPU Krait 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



Специализированные ускорители: Intel Xeon Phi



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

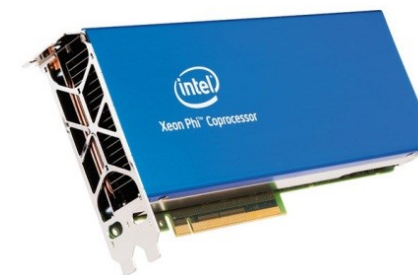
- **Intel Xeon Phi (Intel MIC):** 64 cores Intel P54C (Pentium)
- **Pipeline:** in-order, 4-way SMT, 512-bit SIMD
- Кольцевая шина (1024 бит, ring bus) для связи ядер и контроллера памяти GDDR5
- Устанавливается в PCI Express слот



The Tianhe-2 Xeon Phi drawer in action

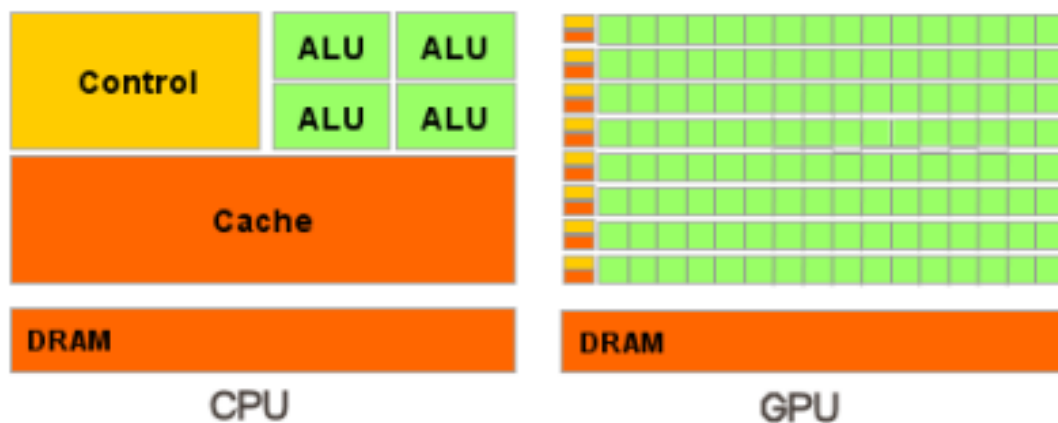
http://www.theregister.co.uk/Print/2013/06/10/inside_chinas_tianhe2_massive_hybrid_supercomputer/

SMP-система
256 логических
процессоров



Специализированные ускорители: GPU – Graphics Processing Unit

- **Graphics Processing Unit (GPU)** – графический процессор, специализированный многопроцессорный ускоритель с общей памятью
- Большая 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)

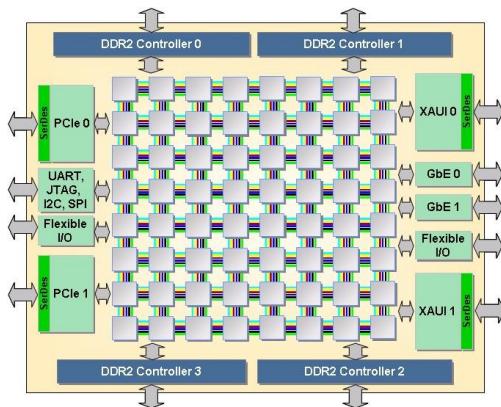
Специализированные многоядерные процессоры



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



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



Tiler TILEPro64
(64 cores, VLIW, mesh)



Cisco Routers
MIPS
Multi-core processors

Многопроцессорные системы с общей памятью

- Как (на чем) разрабатывать программы для такого количества многоядерных архитектур?
- Как быть с переносимостью кода программ между платформами?
- Как быть с переносимостью производительности программ?
- Все ли алгоритмы эффективно распараллеливаются?

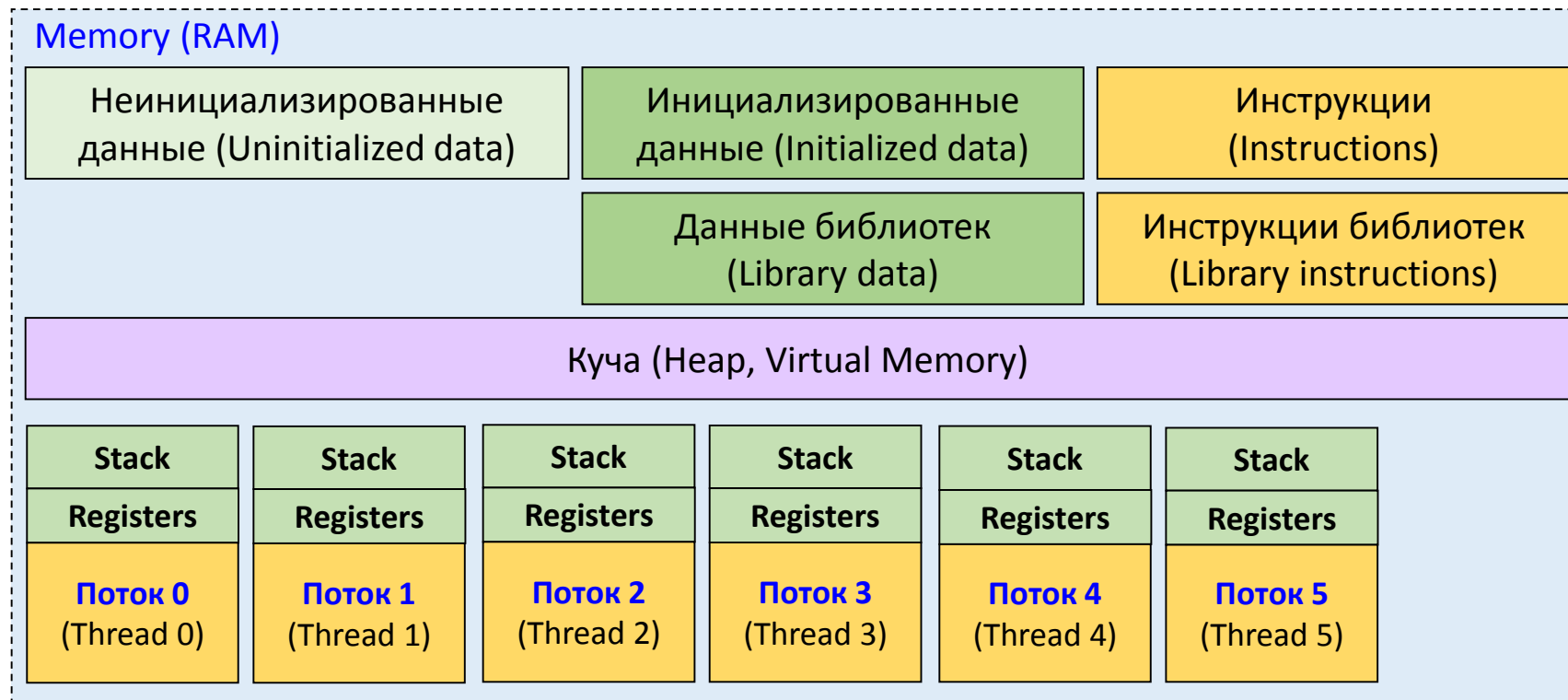
***Concurrency is the next major revolution
in how we write software***

-- Herb Sutter

Herb Sutter. *The Free Lunch Is Over:
A Fundamental Turn Toward Concurrency in Software* // <http://www.gotw.ca/publications/concurrency-ddj.htm>

Процессы и потоки

Многопоточный процесс (Multithreaded process)

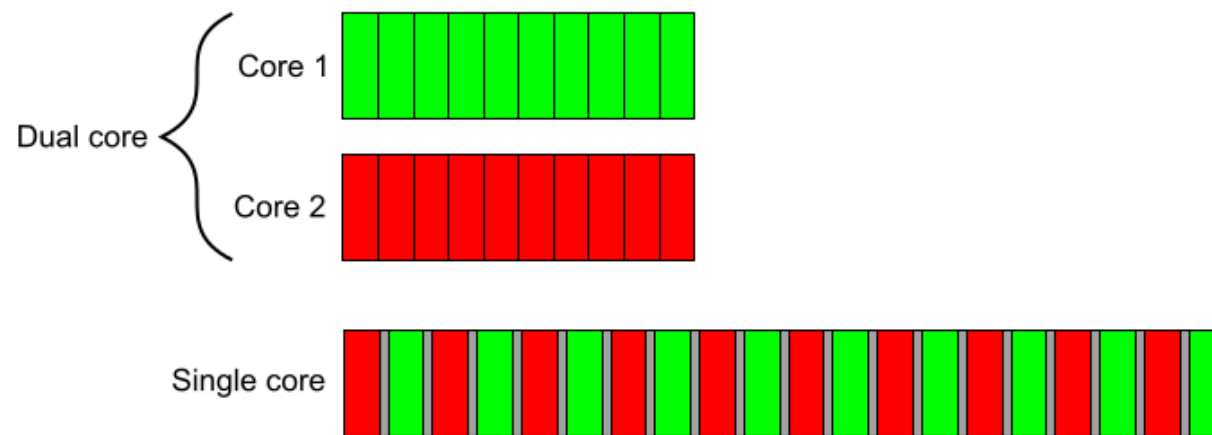


- Каждый поток имеет свой стек, TLS и контекст (context) – память для хранения значения архитектурных регистров при переключении контекстов (context switching) операционной системой
- Куча процесса (heap), инструкции, статические данные (инициализированные) являются общими для всех потоков

Concurrency != Parallelism

- **Concurrency (одновременность)** – несколько потоков разделяют одно процессорное ядро
- Операционная система реализует режим деления времени ядра процессора (time sharing)
- Ускорение вычислений отсутствует
- Зачем?
- Обеспечение отзывчивости интерфейса, совмещение ввода-вывода и вычислений, ...

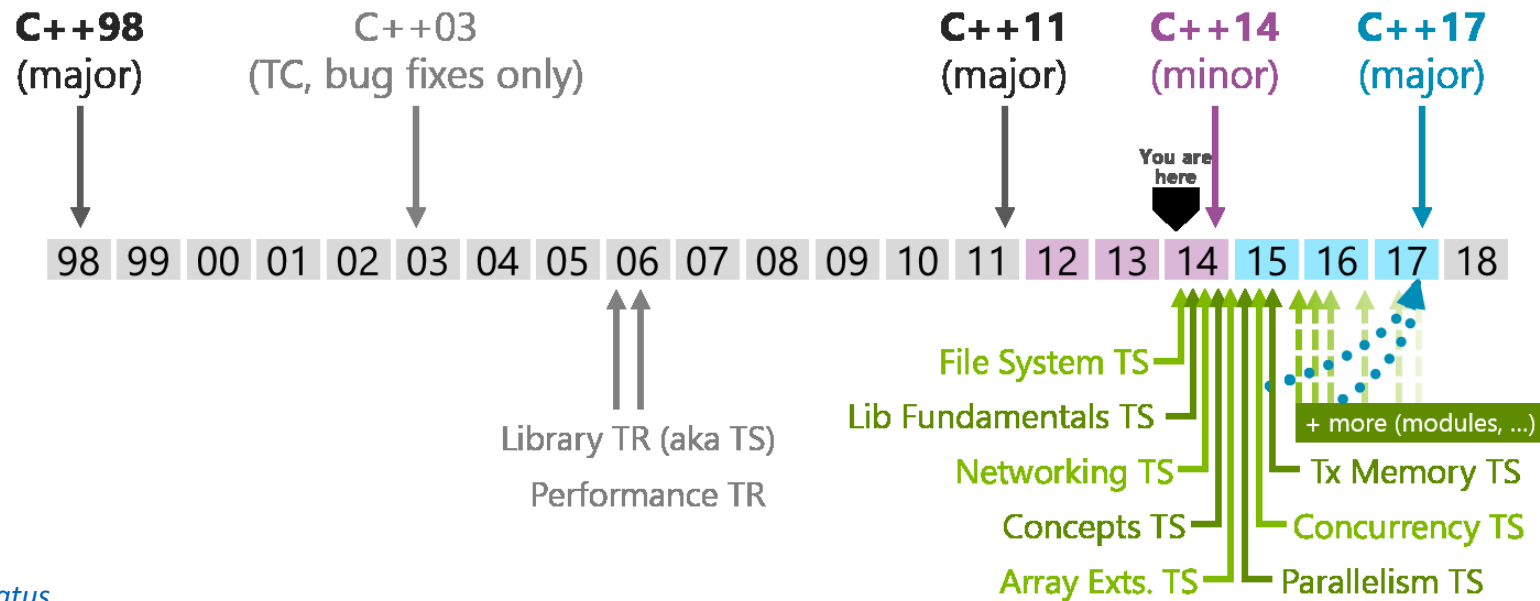
- **Parallelism (параллелизм)** – каждый поток выполняется на отдельном ядре процессора (нет конкуренции за вычислительные ресурсы)
- Вычислений выполняются быстрее



Многопоточность C++

C++11 & C++14

- **Стандарт C++11:** принят в 2011 году, ISO/IEC 14882:2014 (major): многопоточность
- **Стандарт C++14:** принят в 2014 году, ISO/IEC 14882:2014 (minor)
<http://isocpp.org/std/the-standard>
- **Working Draft C++14 (N4296):** <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf>



[*] <https://isocpp.org/std/status>

Поддержка C++14 компиляторами

- **GCC**

<https://gcc.gnu.org/projects/cxx1y.html>

- **Clang**

http://clang.llvm.org/cxx_status.html

- **Intel C++**

<https://software.intel.com/en-us/articles/c14-features-supported-by-intel-c-compiler>

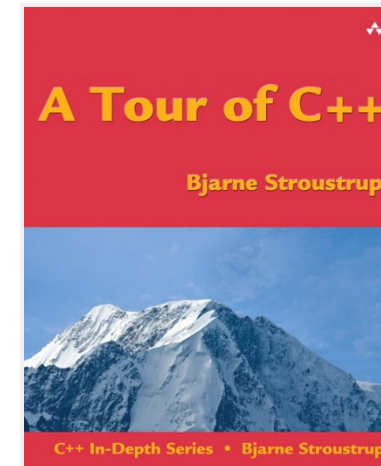
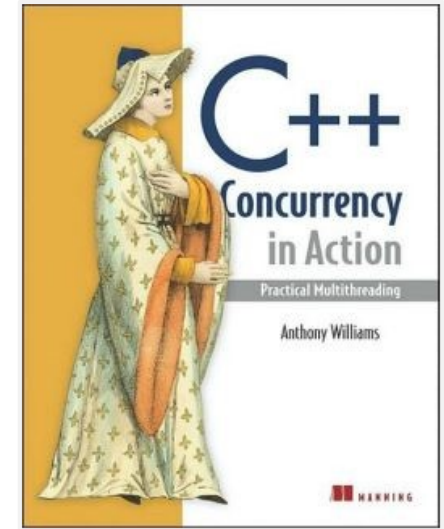
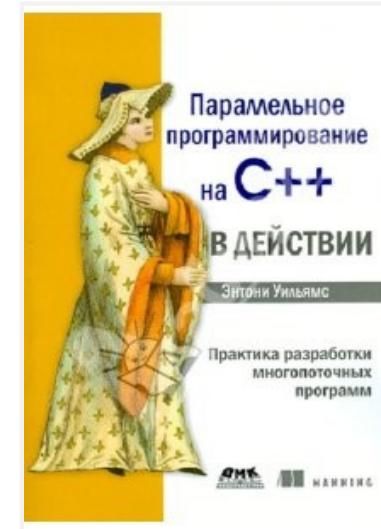
- **Oracle Solaris Studio**

<http://www.oracle.com/technetwork/server-storage/solarisstudio/features/compilers-2332272.html>

- **Visual C++**

<http://blogs.msdn.com/b/vcblog/archive/2014/08/21/c-11-14-features-in-visual-studio-14-ctp3.aspx>

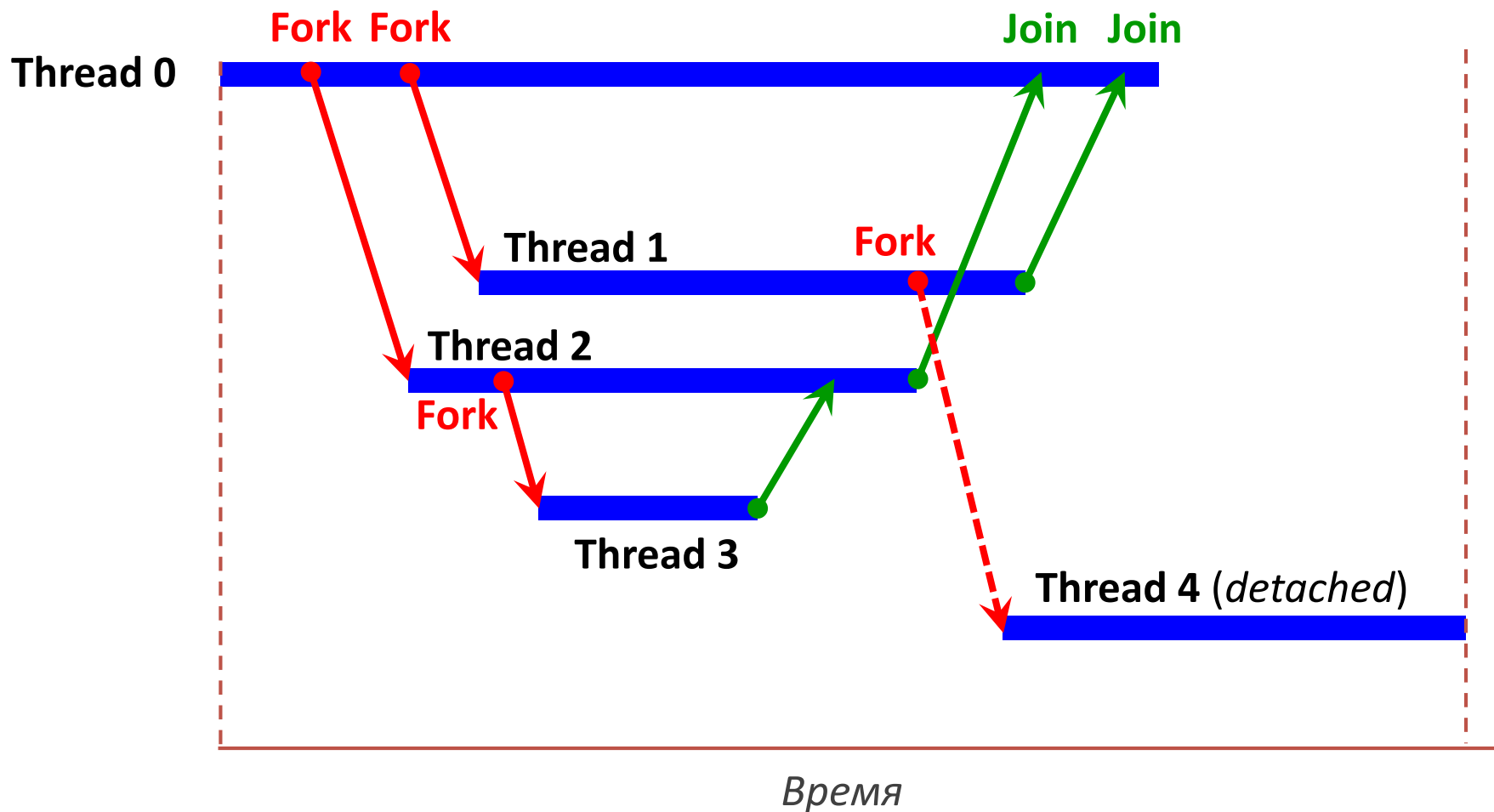
- Уильямс Э. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. - М.: ДМК Пресс, 2012.
- Anthony Williams. **C++ Concurrency in Action. Practical Multithreading**, Manning, 2012
- Bjarne Stroustrup. **A Tour of C++**. The C++ In-Depth Series, Pearson Education, 2013



C++11 Thread support library + Atomic operations library

- **Threads** (class `std::thread`, namespace `std::this_thread`)
- **Mutual exclusion** (`mutex`, `lock_guard`, `lock`, `call_once`, ...)
- **Condition variables** (`condition_variable`)
- **Futures** (`future`, `promise`, `async`, `launch`, `packaged_task`)
- **Atomic operations** (`std::atomic`)
- ...

C++11 Threads – Fork-Join Model



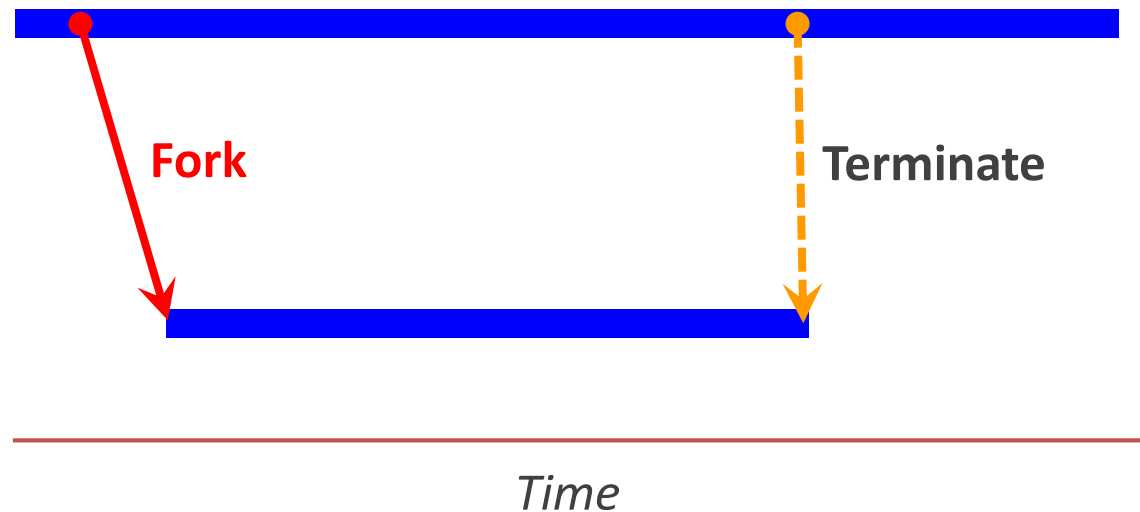
- **Fork** – создание нового потока
- **Join** – ожидание одним потоком завершения другого (объединение потоков управления)

Email Client

(GUI thread + data process thread)

Thread 0
Drawing GUI

Thread 1
Loading emails
in background

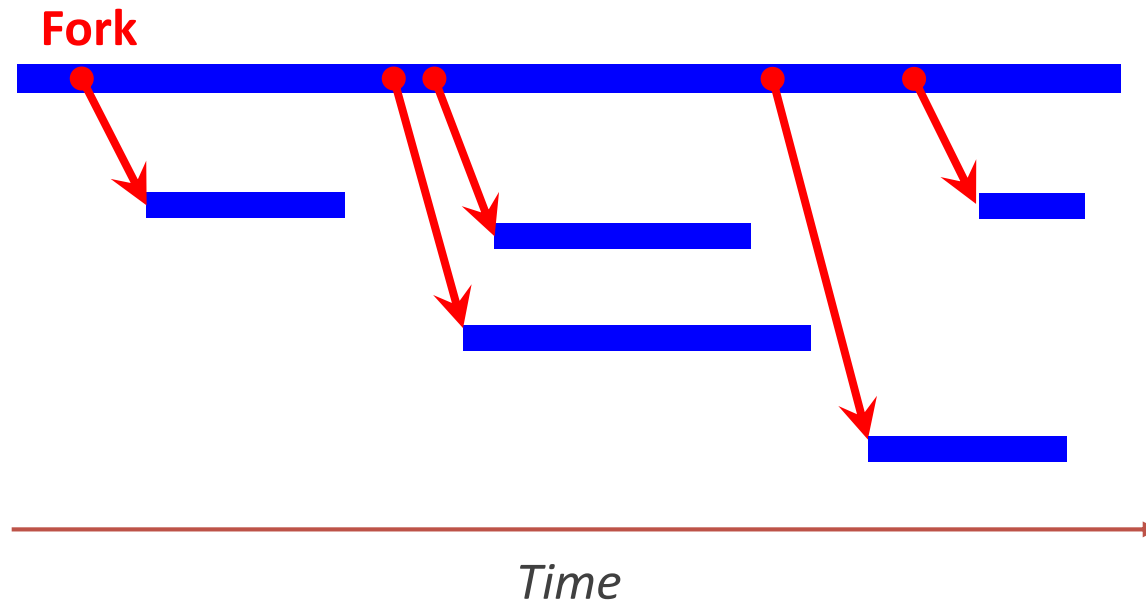


Multithreaded HTTP-server

Master thread + 1 thread per request

Thread 0
Accepting requests

Worker thread
Loading HTML-document
and sending response



Multithreaded HTTP-server

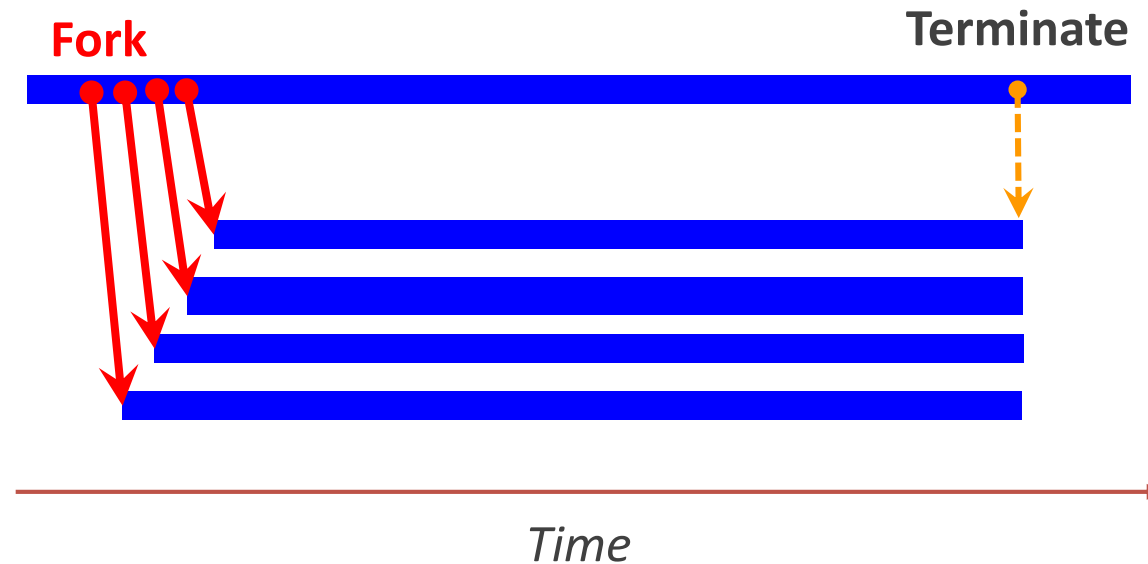
Master thread + **thread pool**

Thread 0

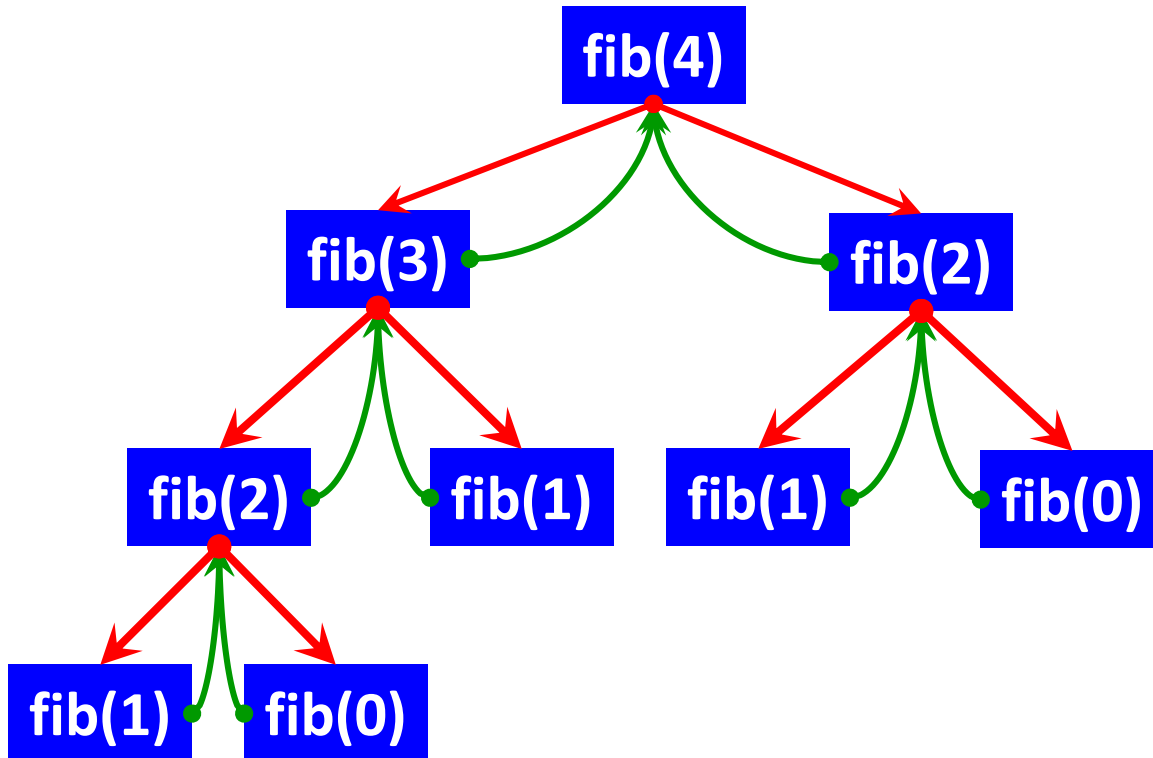
Queue of requests

Worker thread

Processing requests
from the queue



Recursive Parallelism – Divide & Conquer (Parallel QuickSort, Reductions, For loops)



```
function fib(int n)
  if n < 2 then
    return n
  x = fork threadX fib(n - 1)
  y = fork threadY fib(n - 2)
  join threadX
  join threadY
  return x + y
end function
```

Hello, Multithreaded World!

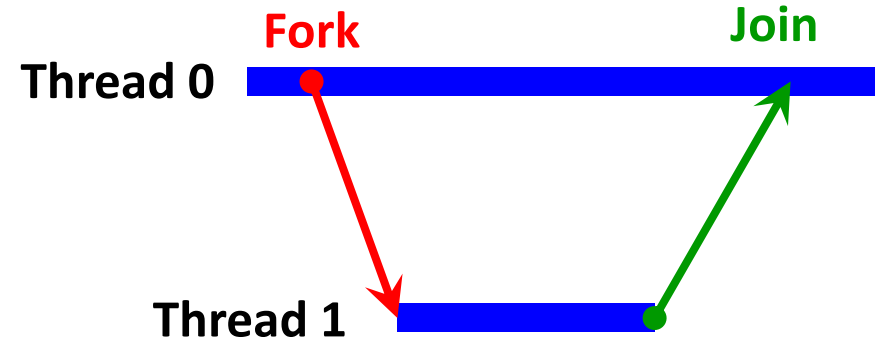
```
#include <iostream>
#include <thread>

void hello()
{
    std::cout << "Hello, Multithreaded World!\n";
}

int main()
{
    // Создаем и запускаем новый поток выполнения
    std::thread mythread(hello);

    // Продолжаем вычисления в главном потоке

    // Ожидаем завершения потока mythread
    mythread.join();
    return 0;
}
```



Компиляция многопоточных программ на C++11

```
# GNU/Linux GCC compiler
```

```
$ g++ -Wall -std=c++11 -pthread -ohello ./hello.cpp
```

```
# GNU/Linux Clang (LLVM)
```

```
$ clang++ -Wall -std=c++11 -pthread -ohello ./hello.cpp
```

```
# GNU/Linux Intel C++ Compiler
```

```
$ icpc -Wall -std=c++11 -pthread -ohello ./hello.cpp
```

- **Oracle Solaris Studio** (GNU/Linux, Oracle Solaris)
- **Microsoft Visual Studio Express 2013**
- **Online C++ Compilers:** liveworkspace.org, coliru.stacked-crooked.com, gcc.godbolt.org, rise4fun.com/vcpp, www.compileonline.com, comeaucomputing.com/tryitout

Класс std::thread

- **Методы класса std::thread**

- std::thread::id `get_id()` const; // ==, !=, <, >, operator<<
- bool `joinable()` const;
- native_handle_type `native_handle()`;
- static unsigned `hardware_concurrency()`;
- void `join()`;
- void `detach()`;
- void `swap(thread& other)`;

- В конструктор класса **std::thread** можно передавать объект любого типа, допускающие вызов (Callable):
 - ❑ функцию возвращающую значение типа void
 - ❑ объект-функцию (function object)
 - ❑ лямбда-выражение (lambda expression)

Использование объекта-функции

```
#include <iostream>
#include <thread>

class background_task {
public:
    void operator()() const
    {
        std::cout << "Hello, Multithreaded World!\n";
    }
};

int main()
{
    background_task bgtask;
    std::thread mythread(bgtask);    // Инициализировали поток объектом-функцией

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```


Использование лямбда-выражения

```
#include <iostream>
#include <thread>

int main()
{
    std::thread mythread([]() -> void {
        std::cout << "Hello, Multithreaded, World\n";
    });

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```

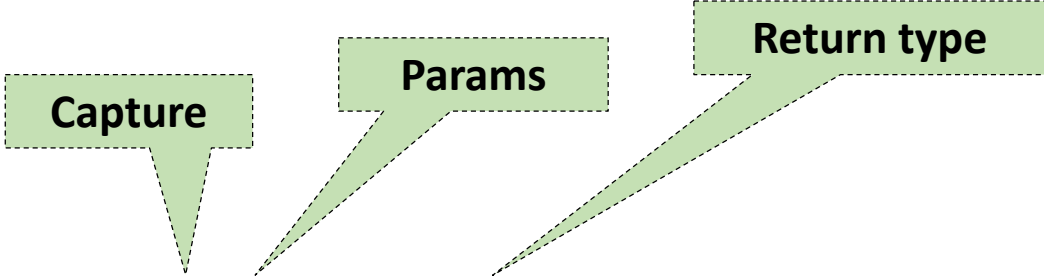
Использование лямбда-выражения

```
#include <iostream>
#include <thread>
```

```
int main()
{
    std::thread mythread([]() -> void {
        std::cout << "Hello, Multithreaded, World\n";
    });

    // Продолжаем вычисления в главном потоке

    mythread.join();
    return 0;
}
```



Capture – определяет какие символы (объекты) будут видны в теле лямбда-функции

- **[a, &b]** – *a* захвачена по значению, *b* захвачена по ссылке
- **[this]** захватывает указатель *this* по значению
- **[&]** захват всех символов по ссылке
- **[=]** захват всех символов по значению
- **[]** ничего не захватывает

Передача данных потоку

- Дополнительные аргументы конструктора `thread::thread()` копируются в память потока, где они становятся доступными новому потоку
- **Сценарии передачи аргументов потоку:**
 - ☐ Передача по значению
 - ☐ Передача по ссылке – поток модифицирует переданный объект
 - ☐ Передача в поток только перемещаемых объектов (movable only)

Передача данных потоку

```
void fun(int i, std::string const& s)
{
    std::cout << "1: i = " << i << "; s = " << s << "\n";
}

int main()
{
    const char *s = "Hello";
    int i = 3;

    std::thread t(fun, i, s); // Поток вызывает fun(i, s)

    t.join();
    return 0;
}
```

- *s* преобразуется в `std::string const&` уже в контексте нового потока

Передача данных потоку по ссылке

```
void load_html doc(html doc& doc) {  
    doc.setContent("Page1");  
}  
  
void process_html doc(html doc& doc) {  
    std::cout << "DOC: " << doc.getContent() << "\n";  
}  
  
int main()  
{  
    html doc("DefaultPage");  
  
    std::thread t(load_html doc, std::ref(doc));  
  
    t.join();  
    process_html doc(doc);  
    return 0;  
}
```

Передача потоку только перемещаемых объектов (move only)

```
void fun(std::unique_ptr<std::string> sptr)
{
    std::cout << "1: *sptr = " << *sptr << "\n";
    std::cout << "1: sptr.get() = " << sptr.get() << "\n";
}

int main()
{
    // Перемещение s
    std::unique_ptr<std::string> sptr(new std::string("Big object"));
    std::cout << "0: sptr.get() = " << sptr.get() << "\n";
    std::thread t(fun, std::move(sptr)); // Передали владение sptr потоку
    std::cout << "0: sptr.get() = " << sptr.get() << "\n";
    t.join();
    return 0;
}
```

```
0: sptr.get() = 0x1a25010
0: sptr.get() = 0
1: *sptr = Big object
1: sptr.get() = 0x1a25010
```

Подсоединяемые (joinable) и отсоединённые (detached) потоки

- **Подсоединяемый поток (joinable thread)** – это поток, завершение которого можно дождаться вызвав метод `thread::join()`
- **Join** – объединить потоки управления (control flows)
- При обращении к методу `thread::join()` выполнение вызывающего потока блокируется
- При запуске нового потока он по умолчанию является подсоединяемым
- После запуска потока можно изменить его тип на отсоединенный (detached)
- **Отсоединенный поток (detached)** – поток, у которого разорвана связь с исходным объектом `std::thread`
- Дождаться завершения отсоединено потока невозможно (“живет своей жизнью”)

Подсоединяемые (joinable) и отсоединённые (detached) потоки

Non-initialized object
(default-constructed)

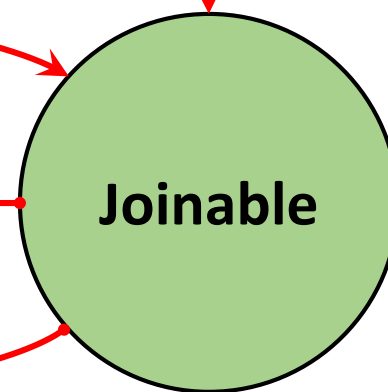
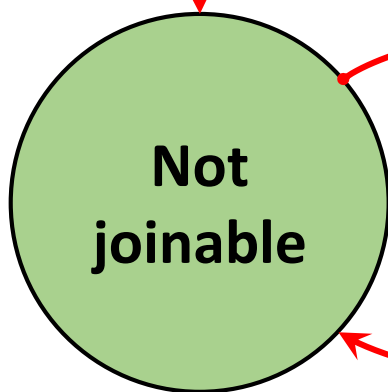
```
std::thread t;
```

Initialized object

```
std::thread t(fun);
```

operator= (thread &&)

```
t = std::thread(fun);
```



join()

detach()

Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread()
{
    std::thread mythread(handler); // Подсоединяемый поток (joinable)
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Вызывается деструктор объекта mythread
    // Объект mythread подсоединяемый (joinable) => деструктор вызывает std::terminate
}

int main()
{
    spawn_thread();
    return 0;
}
```

Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread()
{
    std::thread mythread(handler); // Подсоединяемый поток (joinable)
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Вызывается деструктор объекта mythread
    // Объект mythread подсоединяемый (joinable) => деструктор вызывает std::terminate
}

int main()
{
    spawn_thread();
    return 0;
}
```

```
thread::~~thread()
{
    if (joinable())
        std::terminate();
}
```

Программа завершается с ошибкой
"terminate called without an active exception"

Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread_detached()
{
    std::thread mythread(handler); // Запустили подсоединяемый поток (joinable)
    mythread.detach(); // Отсоединили поток
    std::this_thread::sleep_for(std::chrono::seconds(2));
    // Вызывается деструктор объекта mythread
}

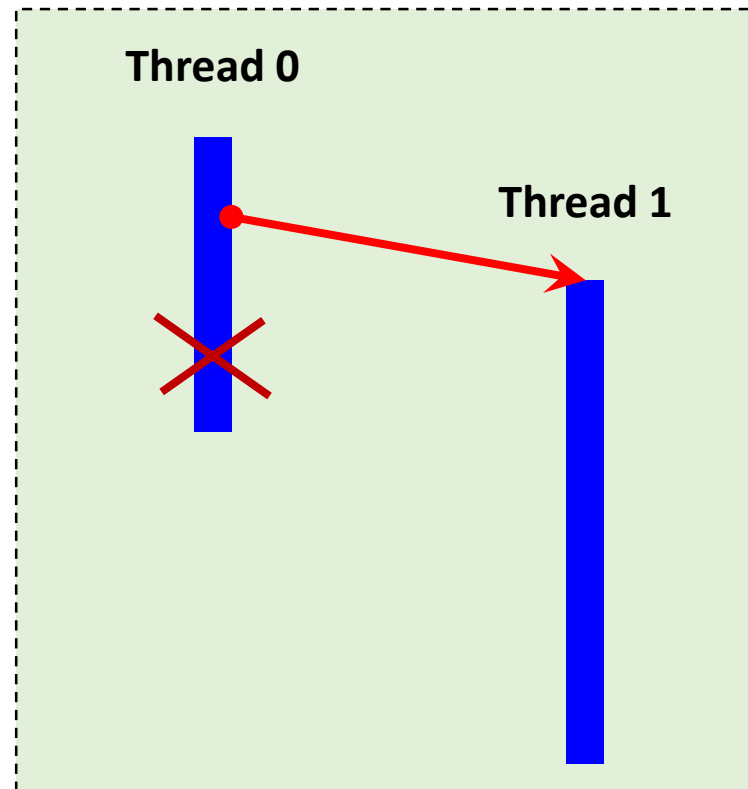
int main()
{
    spawn_thread_detached();

    return 0; // Вызывается std::exit(0), завершаются все потоки процесса
}
```

Выполнение отсоединенного потока mythread
будет прервано при завершении главного
потока (при вызове std::exit())

Подсоединяемые (joinable) и отсоединённые (detached) потоки

- Как сделать так, чтобы отсоединенный поток (detached) продолжил свое выполнение после завершения главного потока?



Подсоединяемые (joinable) и отсоединённые (detached) потоки

```
void handler()
{
    for (int i = 0; i < 10; ++i) {
        std::cout << "Do something" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

void spawn_thread_detached()
{
    std::thread mythread(handler); // Подсоединяемый поток (joinable)
    mythread.detach(); // Отсоединили поток
    std::this_thread::sleep_for(std::chrono::seconds(2));
    // Вызывается деструктор объекта mythread
}

int main()
{
    spawn_thread_detached();

    pthread_exit(NULL); // Завершается только главный поток (библиотека POSIX threads)
}
```

Висячие ссылки

```
class handler {
    int& state_;    // Объект класса handler хранит ссылку на данные
public:
    handler(int& state): state_(state) {}
    void operator()() const {
        for (int i = 0; i < 5; ++i) {
            std::cout << "State = " << state_ << "\n";
            std::this_thread::sleep_for(std::chrono::seconds(1));
        }
    }
};

void run() {
    int i = 33;
    handler h(i);    // Инициализируем объект h ссылкой на переменную i (размещена в стеке потока 0)
    std::thread t(h);
    t.detach();
} // Объект i разрушается, поток t может все еще использовать i (некорректные значения)

int main() {
    run();
    std::this_thread::sleep_for(std::chrono::seconds(2));
    return 0;
}
```

Ожидание в случае исключения

```
void handler() {  
    for (int i = 0; i < 3; i++)  
        std::this_thread::sleep_for(std::chrono::seconds(1));  
}  
  
void do_something() {  
    throw "Exception";  
}  
  
void spawn_thread() {  
    std::thread t(handler);  
    do_something();  
    t.join();  
}  
  
int main()  
{  
    try { spawn_thread(); }  
    catch (...) { std::cout << "Exception\n"; }  
    return 0;  
}
```

- Функция `do_something()` генерирует исключение
- Происходит раскрутка стека
- При выходе из функции `spawn_thread()` вызывается деструктор объекта `t`
- Деструктор `thread::~~thread()` вызывает `std::terminate()`
- **Метод `thread::join()` не вызван!**

Ожидание в случае исключения

```
// ...
```

```
void do_something()
{
    throw "Exception";
}
```

```
void spawn_thread()
{
    std::thread t(handler);
    try { do_something(); }
    catch (...) {
        t.join();
        throw;
    }
    t.join();
}
```

```
int main()
{
    try { spawn_thread(); }
    catch (...) { std::cout << "Exception\n"; }
    return 0;
}
```

- В случае исключения, ожидаем завершения потока и повторно генерируем исключение
- Можно использовать идиому RAII

Запуск многих потоков

```
#include <algorithm>
#include <iostream>
#include <thread>
#include <vector>

void handler(unsigned int id)
{
    std::cout << id << ": Hello, Multithreaded World!\n";
}

int main()
{
    unsigned int nthreads = std::thread::hardware_concurrency();
    std::cout << "Logical processors: " << nthreads << "\n";

    std::vector<std::thread> threads;
    for (size_t i = 0; i < nthreads; ++i) {
        threads.push_back(std::thread(handler, i));
    }
    std::for_each(threads.begin(), threads.end(),
                  std::mem_fn(&std::thread::join));
    return 0;
}
```

Запуск многих потоков

```
$ ./thread_vector
Logical processors: 4
0: Hello, Multithreaded World!
2: Hello, Multithreaded World!
1: Hello, Multithreaded World!
3: Hello, Multithreaded World!

$ ./thread_vector
Logical processors: 4
01: Hello, Multithreaded World!
3: Hello, Multithreaded World!
: Hello, Multithreaded World!
2: Hello, Multithreaded World!
```

- `std::cout` не гарантирует потокобезопасного поведения

Банковский счет (Account)

```
class Account {  
public:  
    Account(int balance): balance(balance) { }  
  
    int getBalance() const { return balance; }  
  
    void deposit(int amount) { balance += amount; }  
  
    bool withdraw(int amount)  
    {  
        if (balance >= amount) {  
            balance -= amount;  
            return true;  
        }  
        return false;  
    }  
  
private:  
    int balance;  
};
```

Клиент банка – снимает со счета определенную сумму

```
void client(int clientid, Account &account, int amount)
{
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
    bool result = account.withdraw(amount);    // Снимаем со счета сумму amount
    if (result)
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
    else
        std::printf("Client %d withdraw %d FAILED\n", clientid, amount);
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
}
```

```
int main(int argc, char *argv[])
{
    Account account(100);
    std::thread t1(client, 1, std::ref(account), 90);
    std::thread t2(client, 2, std::ref(account), 90);
    t1.join(); t2.join();
    std::cout << "Account balance " << account.getBalance()
               << "\n";
    return 0; }
```

Ожидаемый результат

```
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

Результаты запусков (Fedora 20, Intel Core i5-3320M – 2 cores + HT)

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 10
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 100
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

```
$ ./bank_account
Client 1 balance: 100
Client 2 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 withdraw 90 OK
Client 2 balance: -80
Account balance -80
```

```
$ ./bank_account
Client 1 balance: 100
Client 2 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: -80
Client 2 withdraw 90 OK
Client 2 balance: -80
Account balance -80
```

```
$ ./bank_account  
Client 1 balance: 100  
Client 1 withdraw 90 OK  
Client 1 balance: 10  
Client 2 balance: 10  
Client 2 withdraw 90 FAILED
```

```
$ ./bank_account  
Client 1 balance: 100  
Client 2 balance: 100  
Client 1 withdraw 90 OK  
Client 1 balance: 10  
Client 2 withdraw 90 OK
```

В чем причина?

```
Client 1 balance: 10  
Client 2 balance: 100  
Client 2 withdraw 90 FAILED  
Client 2 balance: 10  
Account balance 10
```

```
Client 1 withdraw 90 OK  
Client 1 balance: -80  
Client 2 withdraw 90 OK  
Client 2 balance: -80  
Account balance -80
```

Отрицательный баланс

```
bool withdraw(int amount)
{
    if (balance >= amount) {
        // Приостановим поток на 1 мс, второй успеет “проскочить” условие
        std::this_thread::sleep_for(std::chrono::milliseconds(1));

        balance -= amount;
        return true;
    }
    return false;
}
```

Отрицательный баланс

```
bool withdraw(int amount)
{
    if (balance >= amount) {
        balance -= amount;
        return true;
    }
    return false;
}
```

Два потока осуществляют конкурентный доступ к полю balance – одновременно читают его и записывают

```
// balance -= amount
Load balance -> %reg0
Load amount -> %reg1
Sub %reg0 %reg1 -> %reg0
Store reg0 -> balance
```


Состояние гонки (Race condition, data race)

- **Состояние гонки (race condition, data race)** – это состояние программы, в которой несколько потоков одновременно конкурируют за доступ к общей структуре данных (для чтения/записи)
- Порядок выполнения потоков заранее не известен – носит случайный характер
- Планировщик динамически распределяет процессорное время учитывая текущую загрузженность процессорных ядер, нагрузку (потоки, процессы) создают пользователи, поведение которых носит случайных характер
- Состояние гонки данных (race condition, data race) трудно обнаруживается в программах и воспроизводится в тестах (Гейзенбаг – heisenbug)

Обнаружение состояния гонки (Race condition, data race)

Динамические анализаторы кода

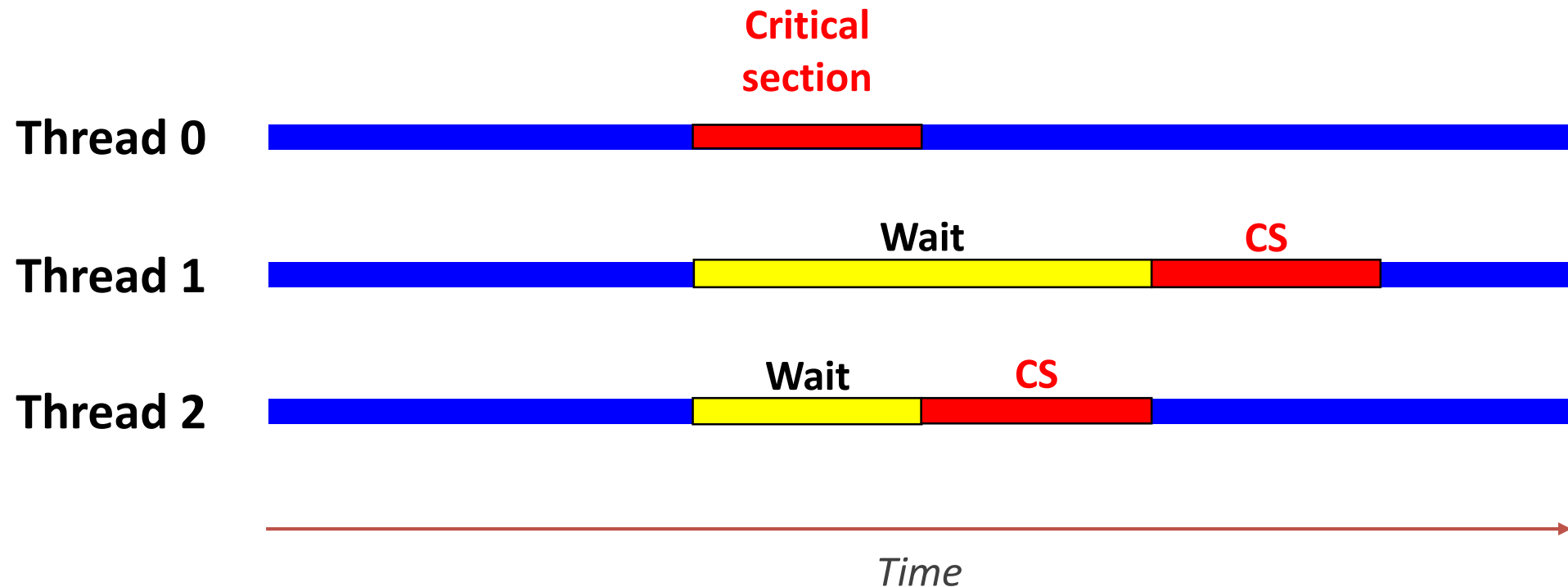
- Valgrind Helgrind, DRD
- ThreadSanitizer – a data race detector for C/C++ and Go (gcc 4.8, clang)
- Intel Thread Checker
- Oracle Studio Thread Analyzer
- Java ThreadSanitizer
- Java Chord

Статические анализаторы кода

- PVS-Studio (Viva64)

Понятие критической секции (Critical section)

- **Критическая секция** (Critical section) — это участок исполняемого кода, который в любой момент времени выполняется только одним потоком



Банковский счет (версия 1 – исходная)

```
class Account {
public:
    Account(int balance): balance(balance) { }
    int getBalance() const { return balance; }
    void deposit(int amount) { balance += amount; }

    bool withdraw(int amount)
    {
        if (balance >= amount) {
            balance -= amount;    // Data race!
            return true;
        }
        return false;
    }

private:
    int balance;
};
```

Банковский счет (версия 2 – mutex)

```
class Account {
public:
    Account(int balance): balance(balance) { }
    int getBalance() const {
        m.lock();
        int val = balance; // Критическая секция
        m.unlock();
        return val;
    }
    bool withdraw(int amount) {
        m.lock();
        if (balance >= amount) {
            balance -= amount; // Критическая секция
            m.unlock();
            return true;
        }
        m.unlock();
        return false;
    }
private:
    int balance;
    mutable std::mutex m;
};
```

Банковский счет (версия 2 – mutex + lock_guard)

```
class Account {  
    // ...  
    int getBalance() const {  
        std::lock_guard<std::mutex> lock(m);  
        return balance;  
    }  
  
    bool withdraw(int amount) {  
        std::lock_guard<std::mutex> lock(m);  
        if (balance >= amount) {  
            balance -= amount;  
            return true;  
        }  
        return false;  
    }  
  
private:  
    int balance;  
    mutable std::mutex m;  
};
```

Результаты

```
$ ./bank_account
Client 1 balance: 100
Client 1 withdraw 90 OK
Client 1 balance: 10
Client 2 balance: 100
Client 2 withdraw 90 FAILED
Client 2 balance: 10
Account balance 10
```

- В функции `client()` потоки одновременно обращаются к методу `Account::getBalance()`, затем вызывают метод `Account::withdraw()`
- Код функции `client()` должен выполняться как неделимая операция – между вызовами `Account::getBalance()` и `Account::withdraw()` другие потоки не должны изменять состояния счета

Банковский счет (версия 3 – recursive_mutex)

```
class Account {  
    // ...  
    int getBalance() const {  
        std::lock_guard<std::recursive_mutex> lock(m);  
        return balance;  
    }  
  
    bool withdraw(int amount) {  
        std::lock_guard<std::recursive_mutex> lock(m);  
        if (balance >= amount) {  
            balance -= amount;  
            return true;  
        }  
        return false;  
    }  
  
    std::recursive_mutex& getMutex() { return m; }  
  
private:  
    int balance;  
    mutable std::recursive_mutex m;  
};
```


Банковский счет (версия 3 – recursive_mutex)

```
void client(int clientid, Account &account, int amount)
{
    std::lock_guard<std::recursive_mutex> lock(account.getMutex());

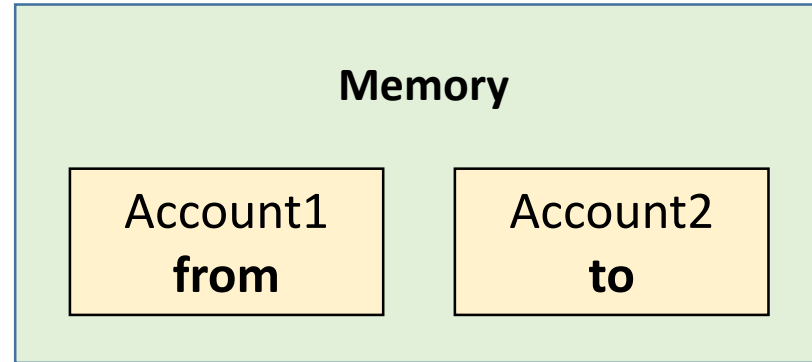
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
    bool result = account.withdraw(amount);
    if (result) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d FAILED\n", clientid, amount);
    }
    std::printf("Client %d balance: %d\n", clientid, account.getBalance());
}
```

- Функция client – критическая секция

Результаты

```
$ ./bank_account  
Client 1 balance: 100  
Client 1 withdraw 90 OK  
Client 1 balance: 10  
Client 2 balance: 10  
Client 2 withdraw 90 FAILED  
Client 2 balance: 10  
Account balance 10
```

Перевод денег между счетами



Thread 0

```
void transfer(  
    int clientid,  
    Account& from,  
    Account& to,  
    int amount  
)
```

Thread 1

```
void transfer(  
    int clientid,  
    Account& from,  
    Account& to,  
    int amount  
)
```

Перевод денег между счетами

```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    std::unique_lock<std::mutex> lock_from(from.getLock());
    // 1. Снимаем
    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);

        std::unique_lock<std::mutex> lock_to(to.getLock());
        to.deposit(amount);    // 2. Зачисляем
        lock_to.unlock();
        std::printf("Client %d deposit %d OK\n", clientid, amount);

        lock_from.unlock();
    } else {
        lock_from.unlock();
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}
```

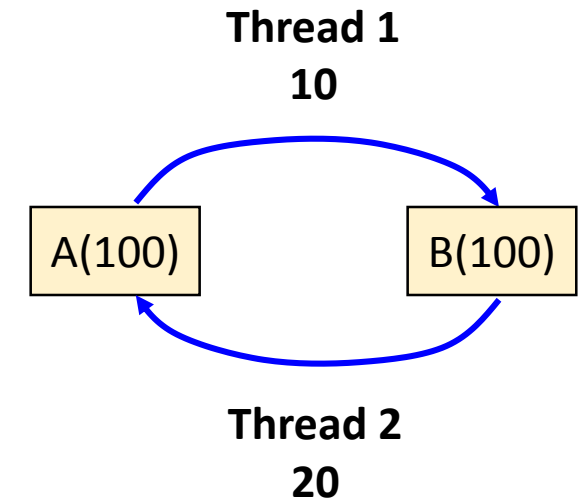
Перевод денег между счетами

```
class Account {  
    // ...  
    int getBalance() const { return balance; }  
    void deposit(int amount) { balance += amount; }  
    bool withdraw(int amount) {  
        if (balance >= amount) {  
            balance -= amount;  
            return true;  
        }  
        return false;  
    }  
  
    std::unique_lock<std::mutex> getLock()  
    {  
        std::unique_lock<std::mutex> lock(m);  
        return lock;    // Вызывается перемещающий конструктор unique_lock  
    }  
  
private:  
    int balance;  
    std::mutex m;  
};
```

Перевод денег между счетами

```
int main(int argc, char *argv[])
{
    Account a(100);
    Account b(100);
    std::thread t1(transfer, 1, std::ref(a), std::ref(b), 10);
    std::thread t2(transfer, 2, std::ref(b), std::ref(a), 20);
    t1.join();
    t2.join();

    // Assert: a=110, b=90
    std::cout << "A balance: " << a.getBalance() << "\n";
    std::cout << "B balance: " << b.getBalance() << "\n";
    return EXIT_SUCCESS;
}
```



```
./bank_account
Client 2 withdraw 20 OK
Client 1 withdraw 10 OK
... Program hangs
```

Взаимная блокировка (Deadlock)

- **Взаимная блокировка (Deadlock)** – несколько потоков находятся в состоянии бесконечного ожидания ресурсов, занятых самими этими потоками

Шаг	Thread 1	Thread 2
1	Захватил ресурс A (mutex)	Захватил ресурс B (mutex)
2	Пытается захватить ресурс B – <i>ожидает</i> его освобождение потоком 1	Пытается захватить ресурс A – <i>ожидает</i> его освобождение потоком 0

```
a.lock()  
b.lock()  // ∞ ожидание  
// Do something  
b.unlock()  
a.unlock()
```

```
b.lock()  
a.lock()  // ∞ ожидание  
// Do something  
a.unlock()  
b.unlock()
```

Взаимная блокировка (Deadlock)



Перевод денег (версия 2)

```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
        to.deposit(amount);
        std::printf("Client %d deposit %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}
```

```
void client(int clientid, Account& from, Account& to, int amount)
{
    if (&from > &to) {
        std::unique_lock<std::mutex> lock_from(from.getLock());
        std::unique_lock<std::mutex> lock_to(to.getLock());
    } else {
        std::unique_lock<std::mutex> lock_to(to.getLock());
        std::unique_lock<std::mutex> lock_from(from.getLock());
    }
    transfer(clientid, from, to, amount);
}
```

```
./bank_account
Client 1 withdraw 10 OK
Client 1 deposit 10 OK
Client 2 withdraw 20 OK
Client 2 deposit 20 OK
A balance: 110
B balance: 90
```

Перевод денег (версия 3): активное ожидание + sleep

```
class Account {  
public:  
    // ...  
    std::unique_lock<std::mutex> getDeferLock()  
    {  
        // Строим lock, но не захватываем мьютекс  
        std::unique_lock<std::mutex> lock(m, std::defer_lock);  
        return lock;  
    }  
    // ...  
};
```

Перевод денег (версия 3): активное ожидание + sleep

```
void client(int clientid, Account& from, Account& to, int amount)
{
    bool success = false;
    std::unique_lock<std::mutex> lock_from(from.getDeferLock());
    std::unique_lock<std::mutex> lock_to(to.getDeferLock());
    while (true) {
        if (lock_from.try_lock()) {
            if (lock_to.try_lock()) {
                transfer(clientid, from, to, amount);
                lock_to.unlock();
                success = true;
            }
            lock_from.unlock();
        }
        if (success)
            break;
        std::this_thread::sleep_for(std::chrono::milliseconds(rand() % 100));
    }
}
```

Перевод денег (версия 4): std::lock

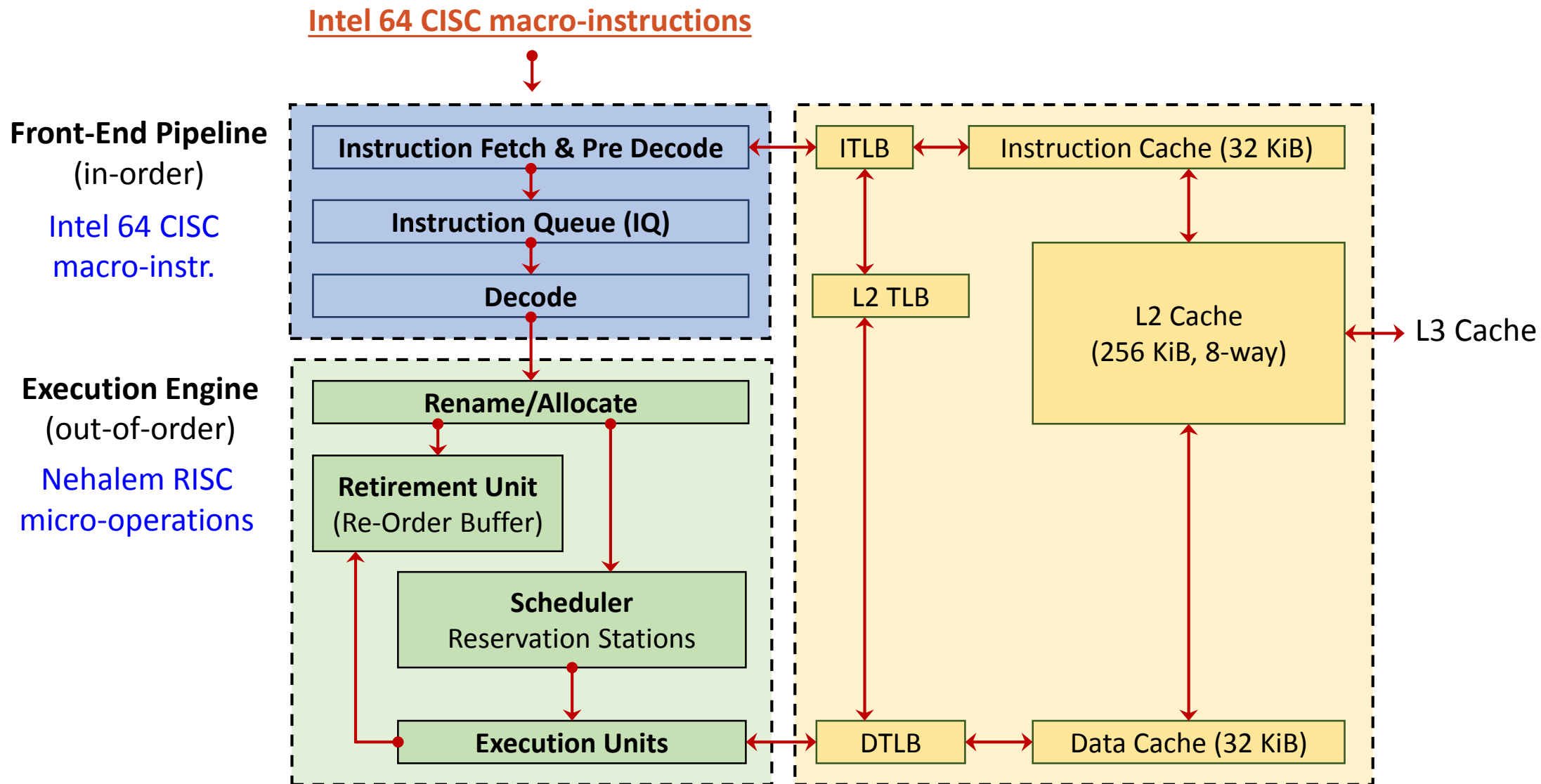
```
void transfer(int clientid, Account& from, Account& to, int amount)
{
    std::unique_lock<std::mutex> lock_from(from.getDeferLock());
    std::unique_lock<std::mutex> lock_to(to.getDeferLock());
    std::lock(lock_from, lock_to);

    if (from.withdraw(amount)) {
        std::printf("Client %d withdraw %d OK\n", clientid, amount);
        to.deposit(amount);
        std::printf("Client %d deposit %d OK\n", clientid, amount);
    } else {
        std::printf("Client %d withdraw %d ERROR\n", clientid, amount);
    }
}
```

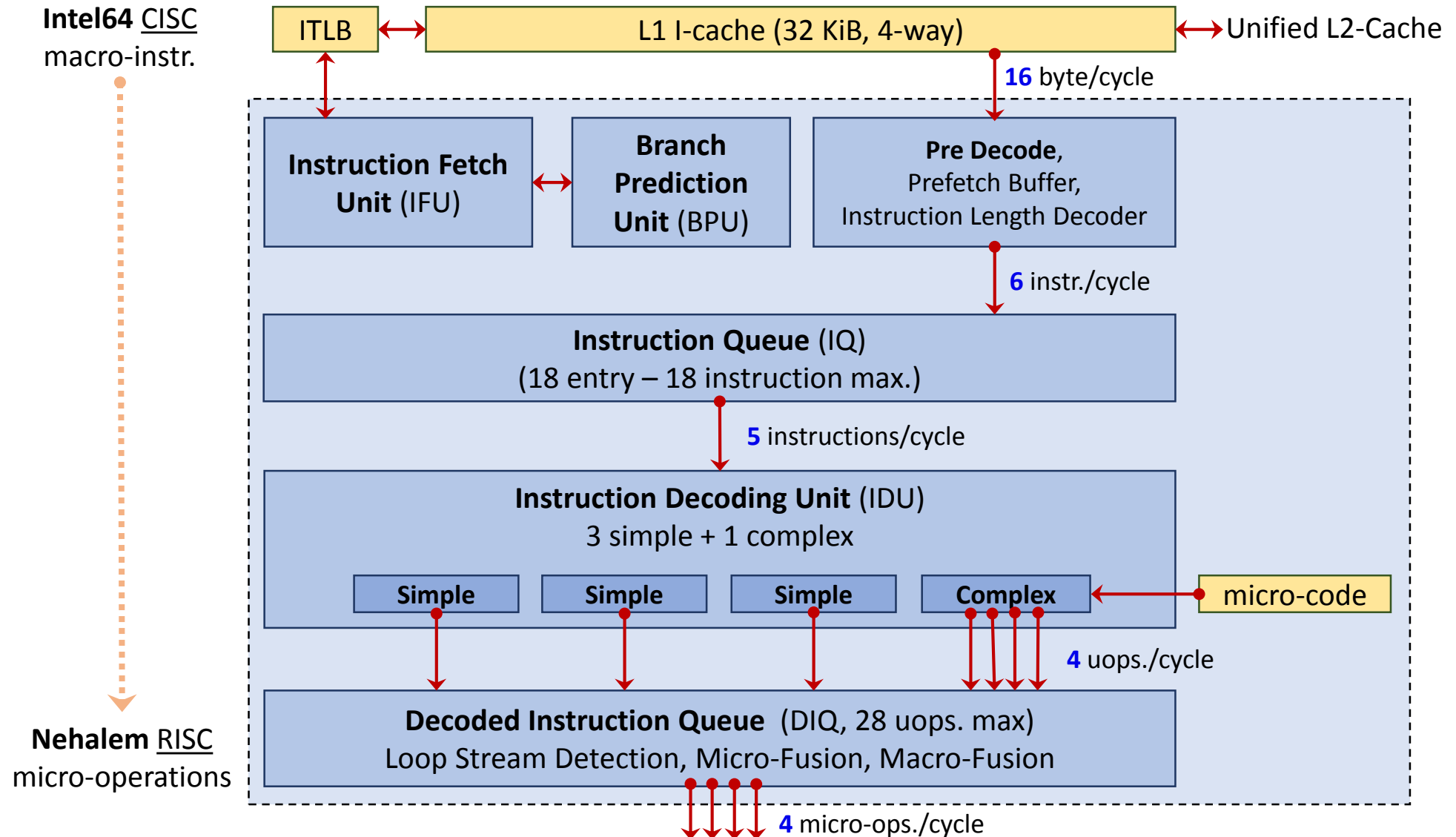
- **C++ and the Perils of Double-Checked Locking //**
http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf
- Herb Sutter. **You Don't Know const and mutable //** C++ and Beyond, 2012
<http://isocpp.org/blog/2012/12/you-dont-know-const-and-mutable-herb-sutter>

Спасибо за внимание!

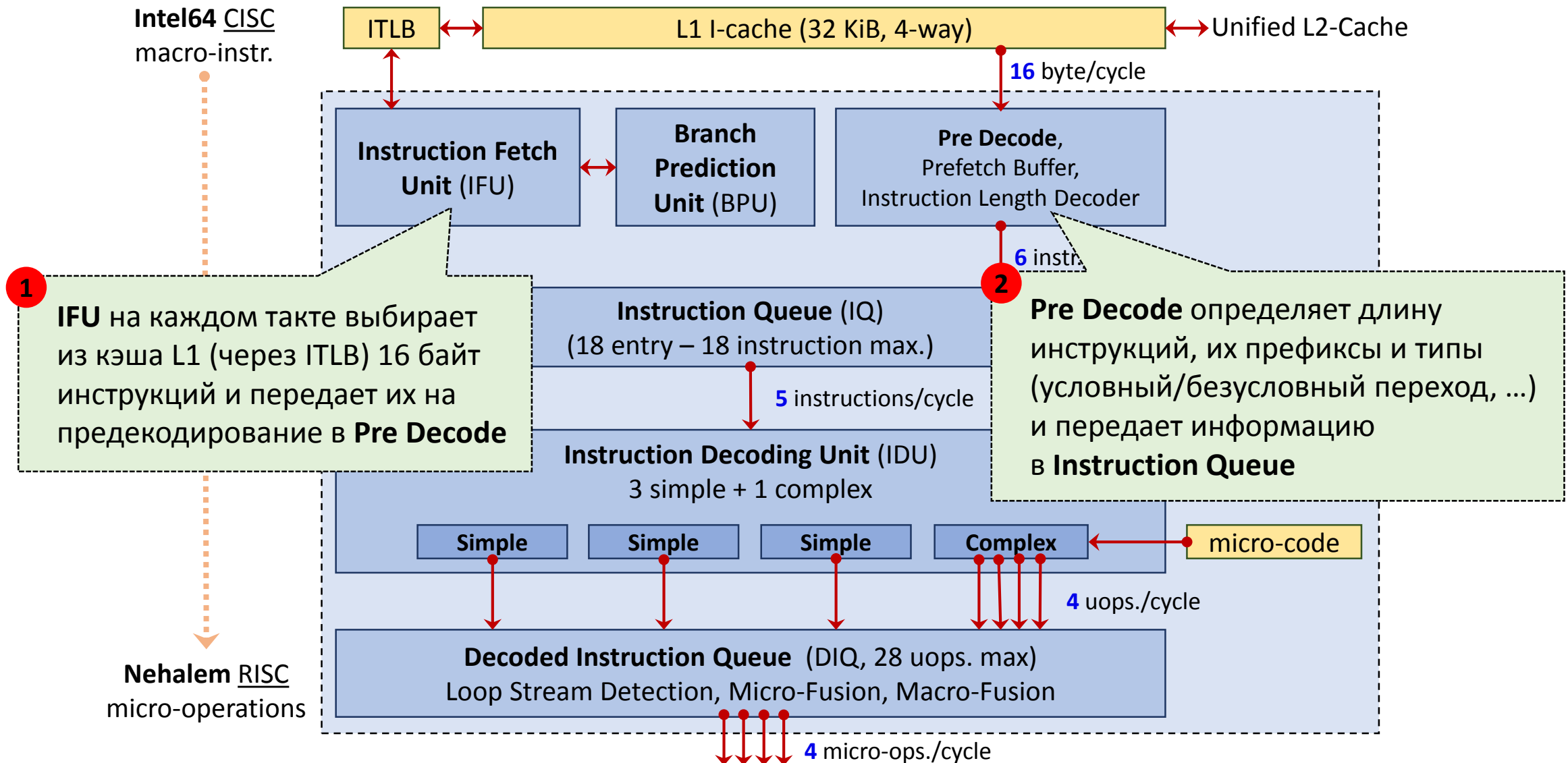
Intel Nehalem Core Pipeline



Intel Nehalem Frontend Pipeline (in-order)



Intel Nehalem Frontend Pipeline (in-order)

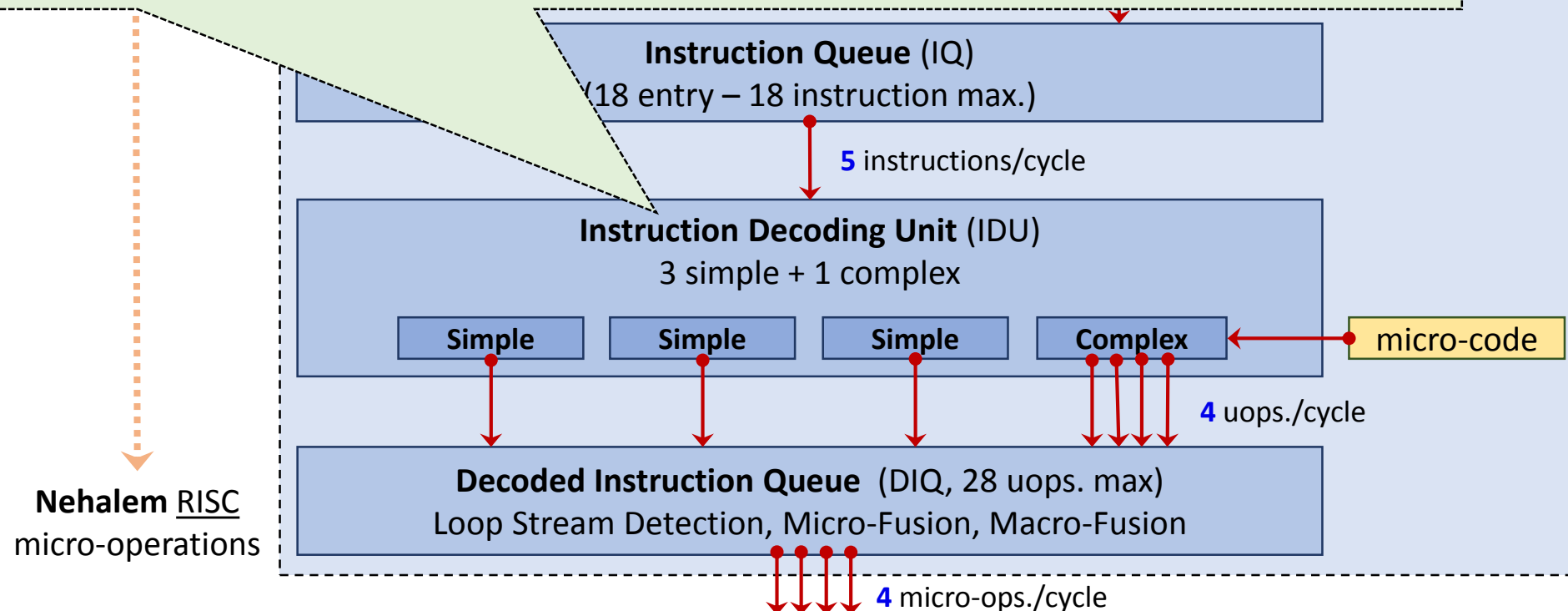


Intel Nehalem Frontend Pipeline (in-order)

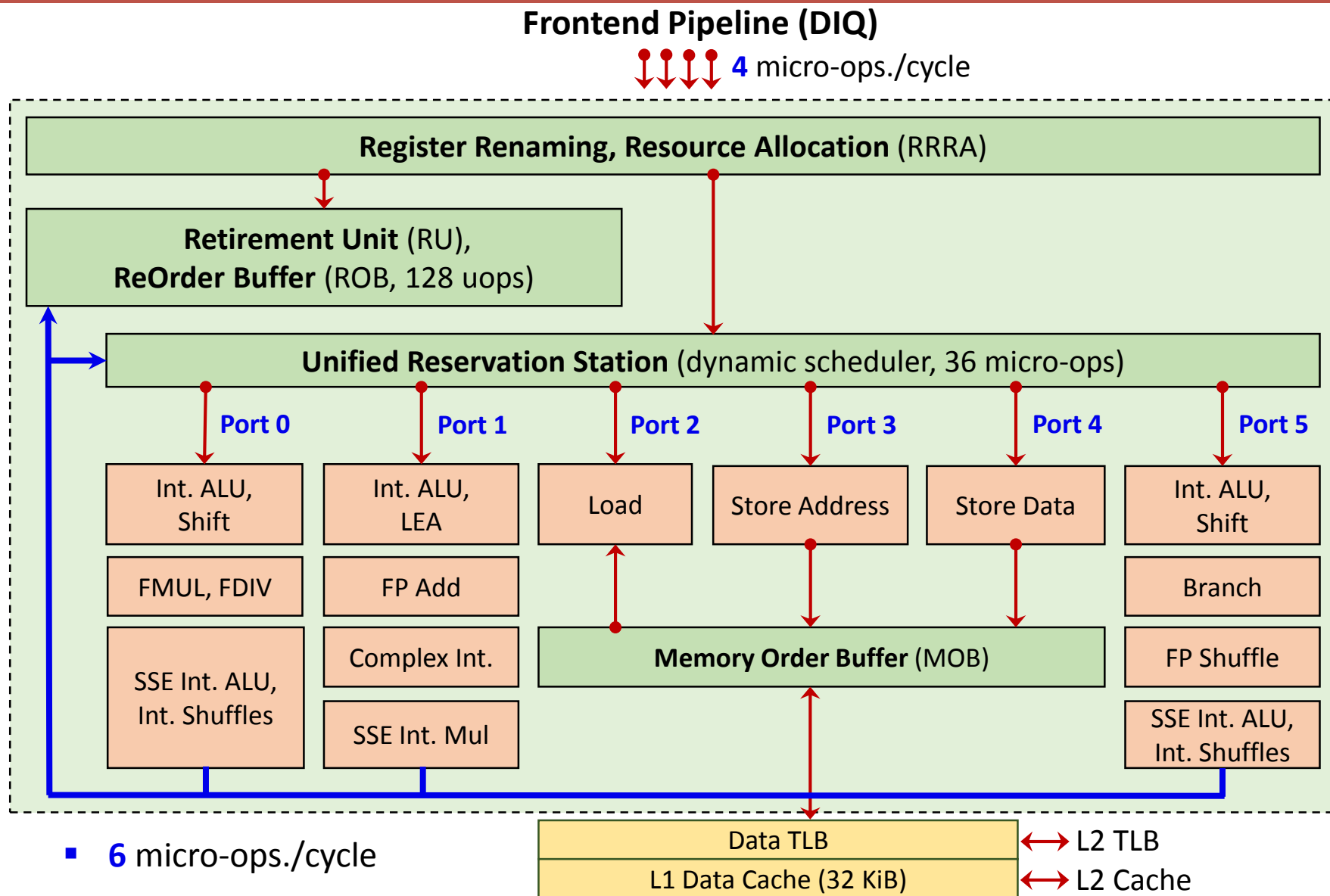
3

- **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

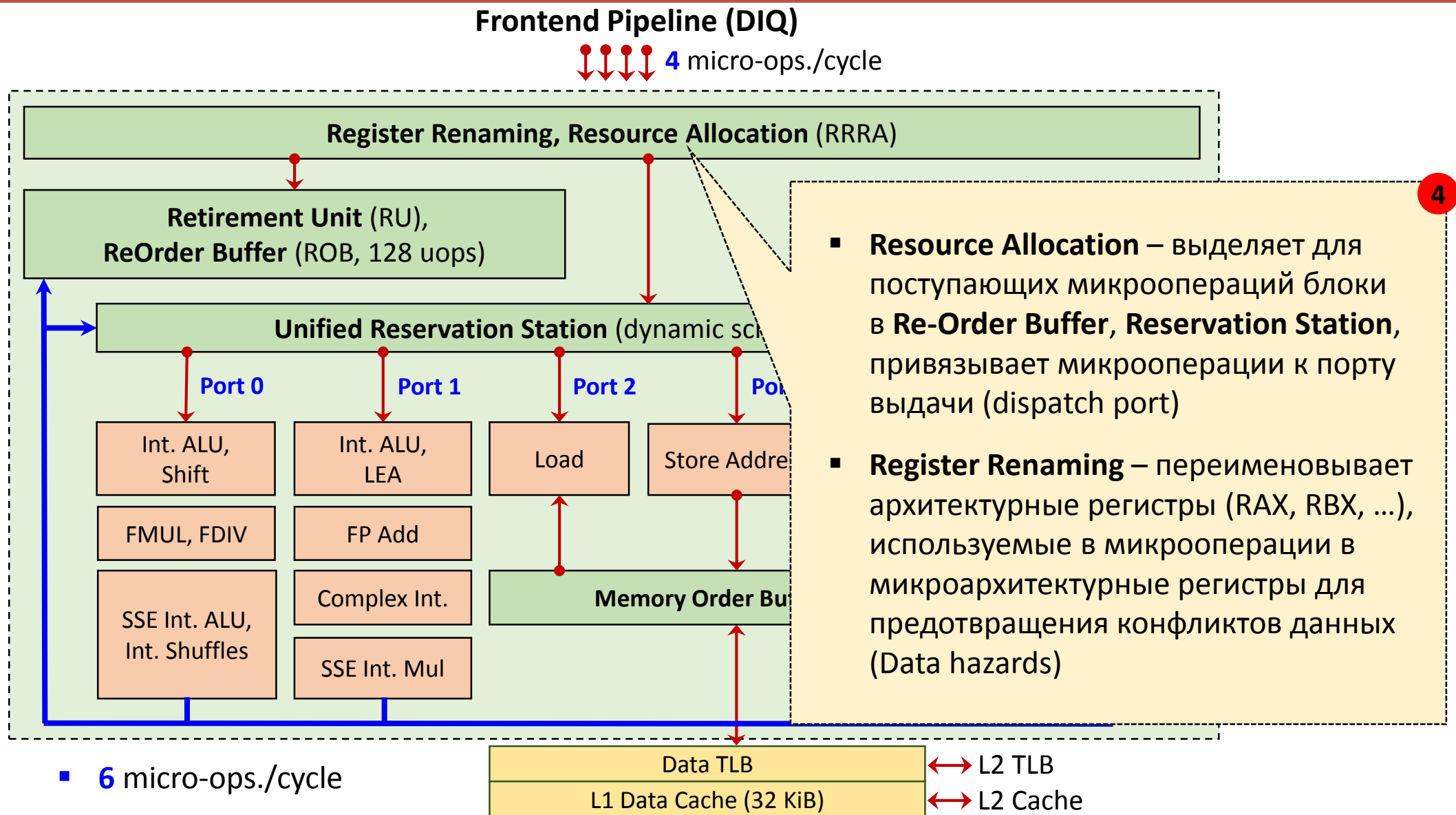
and L2-Cache



Intel Nehalem Execution Engine



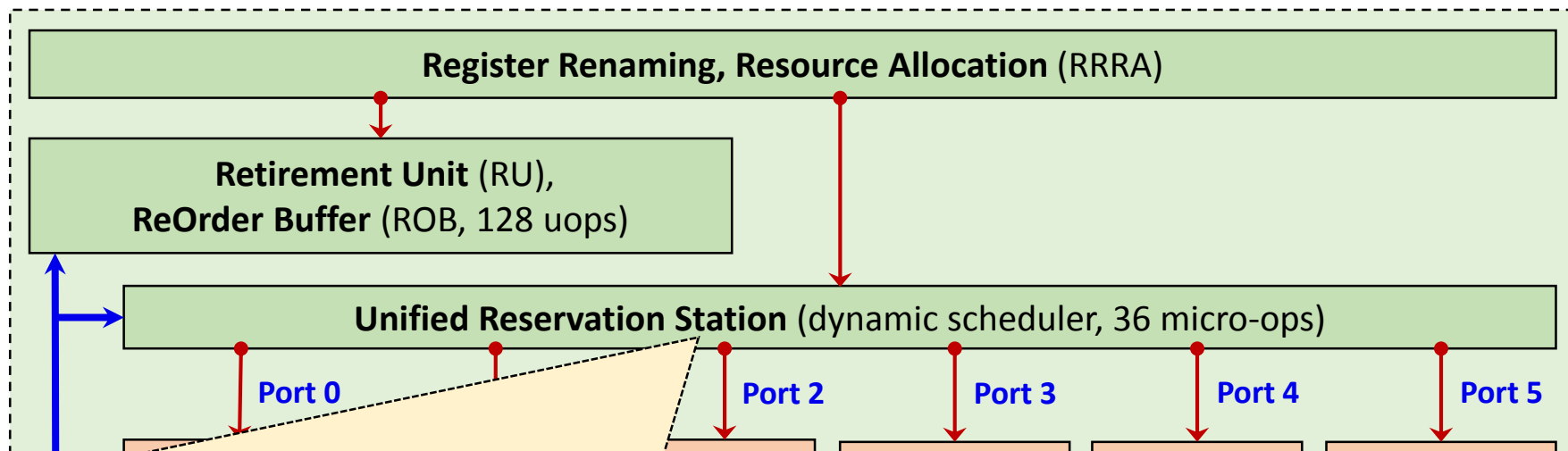
Intel Nehalem Execution Engine



Intel Nehalem Execution Engine

Frontend Pipeline (DIQ)

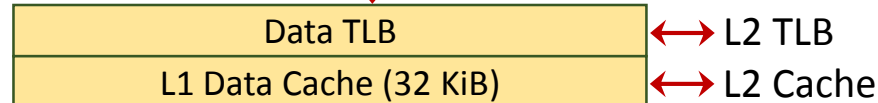
4 micro-ops./cycle



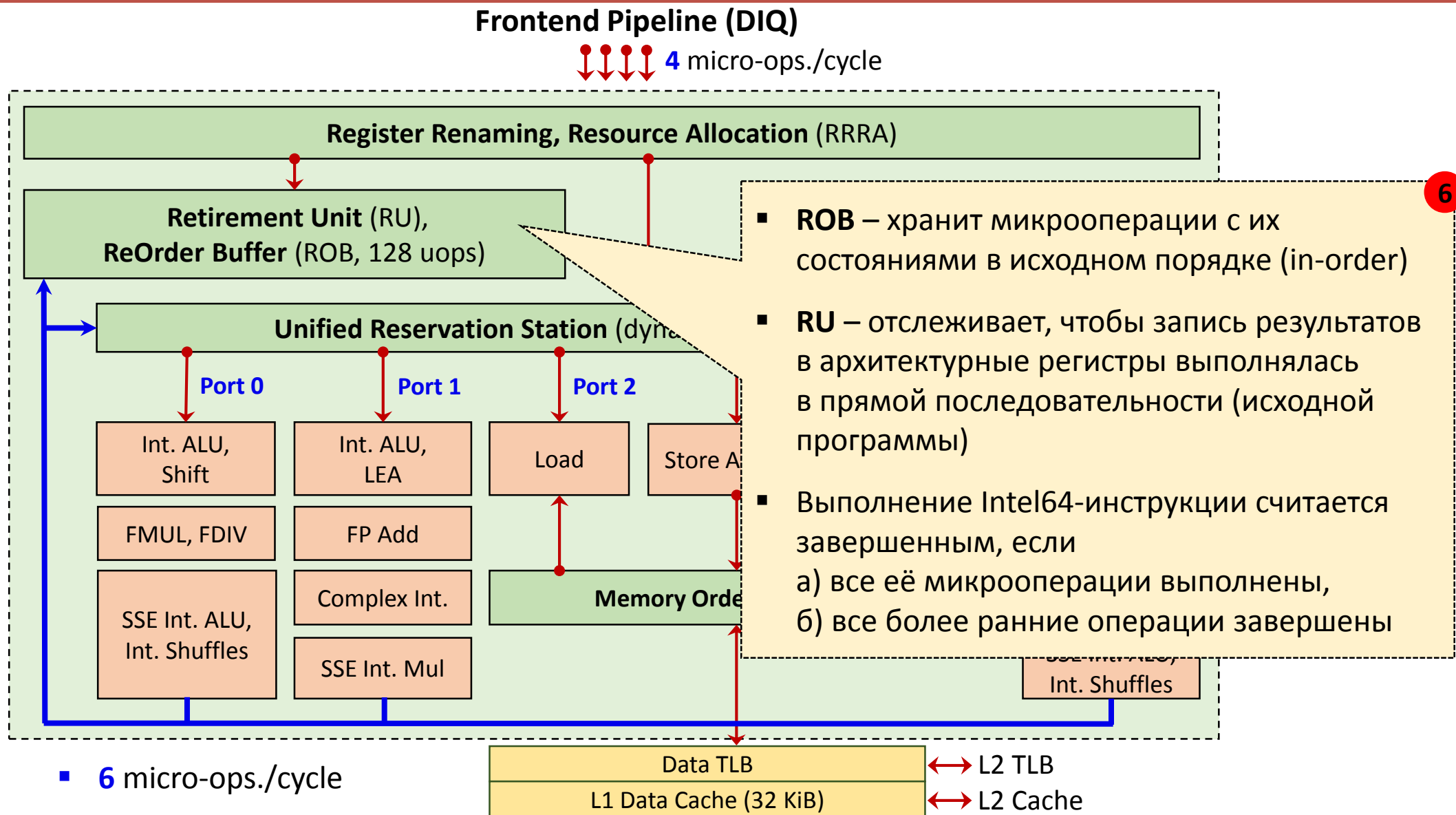
5

- **URS** – пул из 36 микроопераций + динамический планировщик
- Если операнды микрооперации готовы, она направляется на одно из 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)

6 micro-ops./cycle



Intel Nehalem Execution Engine

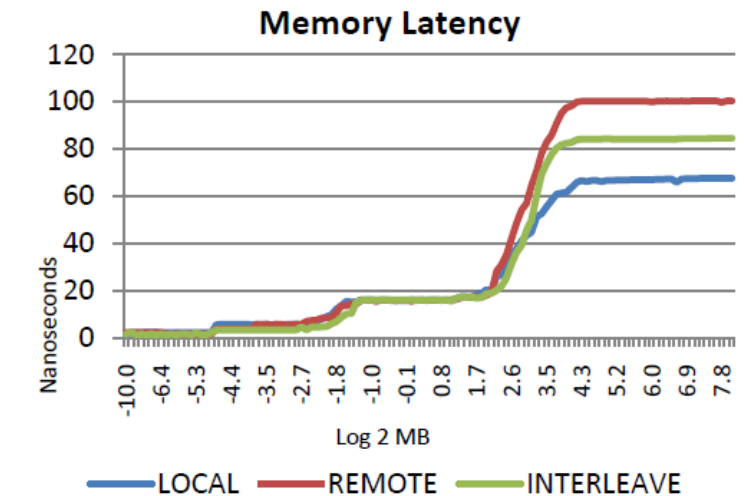
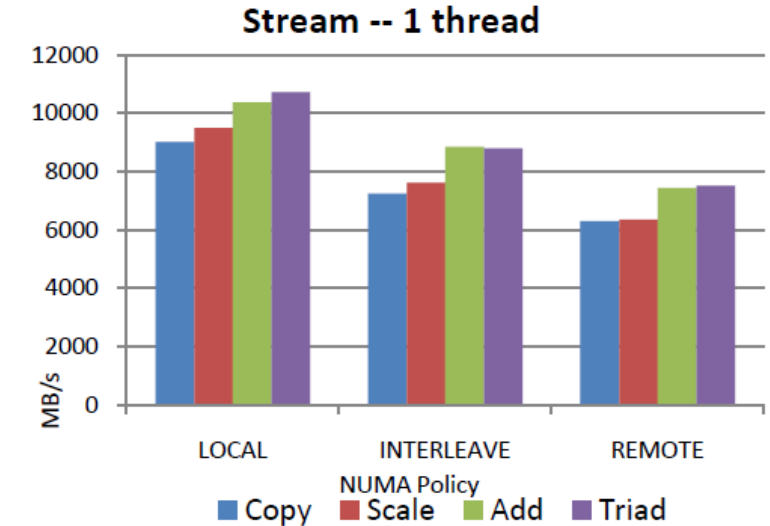


Политики управления памятью NUMA-системы

- 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

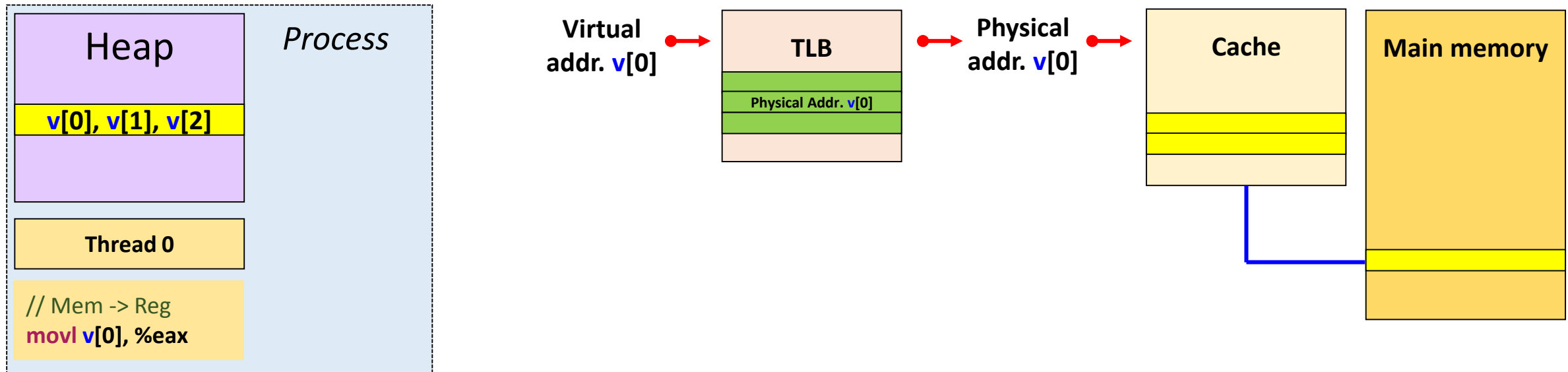
Memory latency and bandwidth accessing local, remote memory for a PowerEdge R610 server (Dual Intel Xeon X5550 Nehalem, 6 x 4GB 1333 MHz RDIMMS)

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



Многопроцессные vs. многопоточные программы

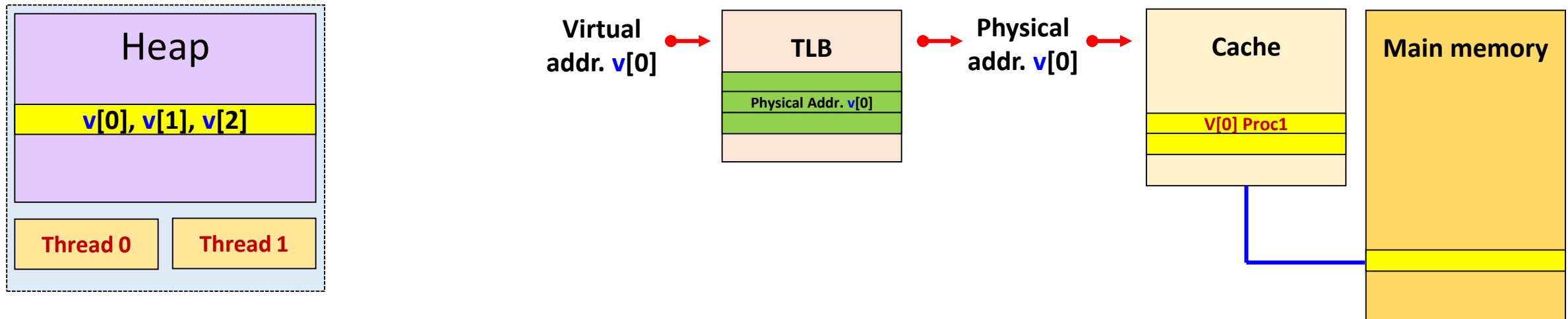
Однопоточный процесс



- Главный поток процесса обращается к элементу массива `v[0]` в его памяти (heap)
- Виртуальный адрес `v[0]` преобразуется в физический адрес: находится соответствие (запись) в кеш-памяти TLB
- По физическому адресу данные загружаются из кеш-памяти или оперативной памяти в регистр

Многопроцессные vs. многопоточные программы

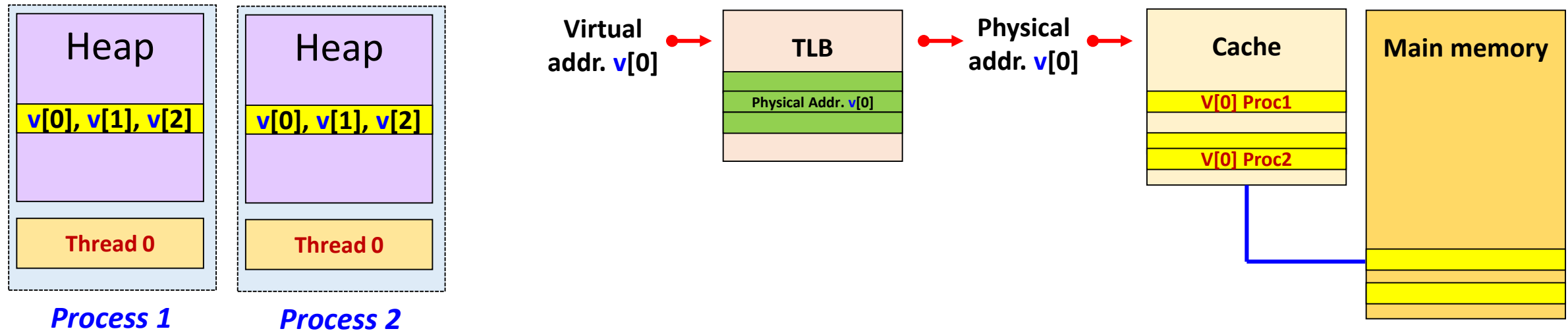
Многопоточная программа (Multithreaded application)



- Запустили процесс с двумя потоками, оба потока загружают значение $v[0]$ в регистр
- Виртуальные адреса элемента $v[0]$ в обоих потоках совпадают, так как куча (heap) у них общая
- Физические адреса $v[0]$ в каждом потоке также совпадают, страницы памяти выделяются процессу и совместно используются его потоками
- Два потока займут в буфере TLB и кеш-памяти 1 запись

Многопроцессные vs. многопоточные программы

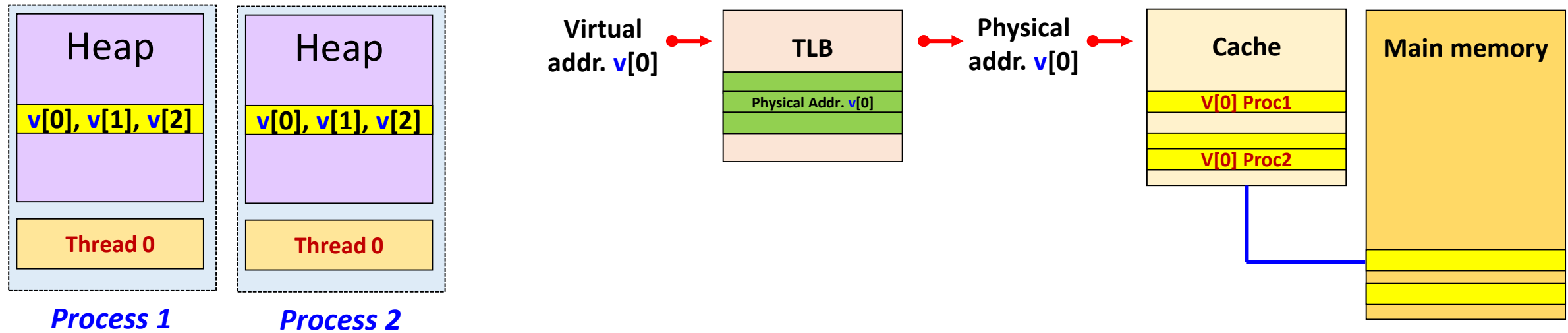
Многопроцессная программа (Multi-process application)



- Запустили 2 процесса одной и той же программы, оба процесса читают $v[0]$ в регистр
- Виртуальный адрес $v[0]$ в обоих процессах совпадает
- Физические адреса $v[0]$ в процессах разные, т.к. процессам выделены разные страницы физической памяти
- Данные каждого процесса будут занимать место в буфере TLB и кеш-памяти

Многопроцессные vs. многопоточные программы

Многопроцессная программа (Multi-process application)



- Многопроцессная версия программы оказывает большую нагрузку на TLB и кеш-память процессора (при условии, что количество процессов > числа логических процессоров)
- В многопоточной программе отказ одного потока может привести к отказу всего приложения