

# Лекция 2

## Оптимизация ветвлений и циклов (branch prediction & loop optimization)

**Курносков Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Высокопроизводительные вычислительные системы»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

# Вычислительный конвейер (Instruction pipeline)

## Выполнение инструкции на процессоре без конвейера

Этапы выполнения инструкции  
(пусть каждый этап длится 1 такт):

- **IF** – выбор инструкции
- **ID** – декодирование инструкции
- **IE** – выполнение инструкции
- **WB** – запись результата

```
i1: movl    %ebx, %eax  
i2: cmpl    $0x10, %ecx  
i3: movl    %edx, %ebx  
i4: xorl    %r8d, %r8d
```

16 тактов

Процессор (ядро)

Такт	IF	ID	IE	WB
1	i1			
2		i1		
3			i1	
4				i1
5	i2			
6		i2		
7			i2	
8				i2
9	i3			
10		i3		
11			i3	
12				i3
13	i4			
14		i4		
15			i4	
16				i4

# Вычислительный конвейер (Instruction pipeline)

## Выполнение инструкции на процессоре без конвейера

Этапы обработки инструкции  
можно сделать независимыми  
и сократить ожидания

```
i1: movl %ebx, %eax  
i2: cmpl $0x10, %ecx  
i3: movl %edx, %ebx  
i4: xorl %r8d, %r8d
```

16 тактов

Процессор (ядро)

Такт	IF	ID	IE	WB
1	i1			
2		i1		
3			i1	
4				i1
5	i2			
6		i2		
7			i2	
8				i2
9	i3			
10		i3		
11			i3	
12				i3
13	i4			
14		i4		
15			i4	
16				i4

# Вычислительный конвейер (Instruction pipeline)

## Выполнение инструкции на процессоре без конвейера

Этапы обработки инструкции  
можно сделать независимыми  
и сократить ожидания

```
i1: movl %ebx, %eax  
i2: cmpl $0x10, %ecx  
i3: movl %edx, %ebx  
i4: xorl %r8d, %r8d
```

16 тактов

Процессор (ядро)

Такт	IF	ID	IE	WB
1	i1			
2		i1		
3			i1	
4				i1
5	i2			
6		i2		
7			i2	
8				i2
9	i3			
10		i3		
11			i3	
12				i3
13	i4			
14		i4		
15			i4	
16				i4

# Вычислительный конвейер (Instruction pipeline)

Выполнение инструкции на процессоре с конвейером

Достигнутое ускорение (Speedup)

$$S_4 = \frac{16}{7} = 2.3$$

```
i1: movl    %ebx, %eax  
i2: cmpl    $0x10, %ecx  
i3: movl    %edx, %ebx  
i4: xorl    %r8d, %r8d
```

} 7 тактов



Процессор (ядро)



Такт	IF	ID	IE	WB
1	i1			
2	i2	i1		
3	i3	i2	i1	
4	i4	i3	i2	i1
5		i4	i3	i2
6			i4	i3
7				i4
8				
9				
10				
11				
12				
13				
14				
15				
16				

# Вычислительный конвейер (Instruction pipeline)

## Выполнение инструкции на процессоре с конвейером

### Анализ ускорения (Speedup)

- $n$  – количество этапов (stages) конвейера
- $L$  – количество инструкций в программе
- $t$  – время выполнения одного этапа

$$S_n = \frac{T_{serial}}{T_{pipeline}} = n$$

$$T_{serial} = L \cdot n \cdot t$$

$$T_{pipeline} = (n - 1)t + Lt$$

$$S = \frac{Lnt}{(n - 1 + L)t} = \frac{Ln}{n - 1 + L}$$

$$\lim_{L \rightarrow \infty} S_n = \frac{Ln}{n - 1 + L} = n$$

Такт	IF	ID	IE	WB
1	i1			
2	i2	i1		
3	i3	i2	i1	
4	i4	i3	i2	i1
5		i4	i3	i2
6			i4	i3
7				i4
8				
9				
10				
11				
12				
13				
14				
15				
16				

# Показатели производительности конвейера

---

- **CPI** (Cycles per instruction, Clocks per instruction) – среднее количество тактов процессора, необходимых для выполнения одной инструкции
- **CPI** зависит от структуры программы (статистики инструкций)
  - Процессор без конвейера (слайд 2):  $CPI = 4$
  - Процессор с конвейером (слайд 4):  $CPI = 1$
  - Если процессор суперскалярный:  $CPI < 1$
- Суперскалярный процессор, выполняющий 2 инструкции за такт:  $CPI = 0.5$

**Цель разработчиков суперскалярных процессоров (ядер)**  
 **$CPI \rightarrow \min$**

- **IPC** (Instructions Per Clock) – это среднее количество инструкций, выполняемых процессором за один такт

$$IPC = 1 / CPI$$

# Показатели производительности конвейера

---

- Программа состоит из  $L$  инструкций
- Тактовая частота процессора  $F$  (Hz = Cycles / Second),  
длительность такта  $C = 1 / F$  (Seconds / Cycle)
- Время выполнения программы:

$$T = L \cdot \text{CPI} \cdot C$$

- Оценка среднего значения CPI

$$\text{CPI} = T / L$$



# От чего зависит время выполнения программы?

---

$$T = L \cdot \text{CPI} \cdot C$$

	$L$ (Instruction Count)	CPI (Cycles Per Instruction)	$1 / C$ (Clock Rate)
Program	+	+	-
Compiler	+	+	-
Instruction Set Architecture (ISA)	+	+	-
CPU microarchitecture	-	+	+
Technology	-	-	+

# Linux Perf

---

```
// prog.c
int *p = malloc(sizeof(*p) * n);
sum = 0;
for (i = 0; i < n; i++)
    sum += p[i];
```

```
$ perf stat -e instructions,cycles ./prog
Performance counter stats for './loopunrolling':



11,071,120,048      instructions      #    1.20  insns per cycle
 9,258,782,431      cycles

2.894491012 seconds time elapsed
```

IPC = 1 / CPI = 1.2

CPI = 0.8

# Intel VTune Amplifier

General Exploration - Hardware Issues  

Intel VTune Amplifier XE 2011

Analysis Target Analysis Type Collection Log Summary Bottom-up

Grouping: Function

Function	Hardware Event Count		CPI Rate	Module	Retire Stalls	Execution Stalls	LLC Miss	Instruction Starvation	Branch Mispredict	Contested Accesses	Data Sharing	LLC Load Misses
	CPU_CLK_UNHALTED...	INST_RETIRED...										
reduce	20,689,803,144	12,333,075,888	1.678	stf.exe	0.698	0.239	0.000	0.011	0.001	0.000	0.000	
strlen	30,768	15,384	2.000	stf.exe	0.000	0.000	0.000	0.000	0.975	0.000	0.000	
strcpy_s	15,384	76,920	0.200	stf.exe	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
LocaleUpdate::_LocaleU	15,384	0		stf.exe	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
setmbcp	0	15,384	0.000	stf.exe	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
unlockexit	0	15,384	0.000	stf.exe	0.000	0.000	0.000	0.000	0.000	0.000	0.000	
setSBUpLow												
init												

Selected 1 row(s):

Line	Source	CPU_CLK_UNHALTED...	RESOURCE_STALLS.ANY	RESOURCE_STALLS.ROB_FULL	RESOURCE_STALLS.STORE
12					
13	unsigned int change_endianness(char *big)				
14	{				
15	char temp;				
16	temp = big[0];	649,326,840	1,040,400,000		1,004,600,000
17	big[0] = big[3];	1,799,982			
18	big[3] = temp;	638,593,614	1,038,800,000		913,400,000
19	temp = big[1];	466,662	200,000		
20	big[1] = big[2];	751,592,484	1,044,800,000	1,000,000	1,032,600,000
21	big[2] = temp;	599,994			
22	return *(unsigned int *)big;				
23	}				
24					
25	void reduce()				
26	{				
27	int i, n;				
28	unsigned int sum = 0;				
29	init();				
30					
31	for (n = 0; n < ITER_NUM; n++){	466,662	1,000,000		1,400,000
32	for (i = 0; i < 1024; i++){	17,898,954,342	12,198,800,000	9,211,000,000	3,178,200,000
33	sum += change_endianness((char *) (buf+i));	778,725,546	74,200,000	2,200,000	99,600,000
34	printf ("Sum is %d\n", sum);				
35	}				
	Selected 1 row(s):	17,898,954,342			

# Oracle Solaris Studio

The screenshot displays the Oracle Solaris Studio Performance Analyzer interface. The title bar reads "test.1.er - Oracle Solaris Studio Performance Analyzer". The menu bar includes "File", "Views", "Tools", and "Help". A toolbar with various icons is located below the menu bar. A "Find:" search box with a dropdown arrow and a "Match Case" checkbox is on the right.

The left sidebar contains a "Views" panel with the following options: "Welcome", "Overview", "Functions", "Timeline", "Call Tree", "Source" (selected), and "Callers-Callees". Below this is a "No Active Filters" section with a filter icon and a text box that says "To add a filter, select a row from a view (such as Functions) and then click".

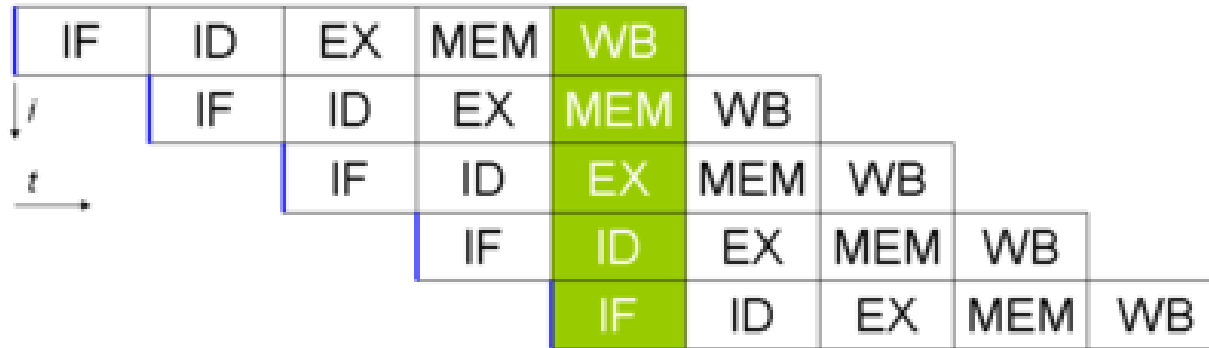
The main window displays a table of performance data for a C source file named "main.c". The table has three columns: "Incl. Total CPU (sec.)", "Incl. CPI", and "Source File: main.c". The data is as follows:

Incl. Total CPU (sec.)	Incl. CPI	Source File: main.c
0.	0.	3.
0.	0.	4. void foo(int m, int n, double *x, double *y, double *A) {
		<Function: foo>
		5. int i, j;
		6.
0.	0.	7. for ( j = 0; j < n; j++ )
0.240	0.273	8. for ( i = 0; i < m; i++ )
1.241	0.914	9. y[j] += A[i+m*j] * x[i];
		10.
0.	0.	11. for ( i = 0; i < m; i++ )
0.140	3.000	12. for ( j = 0; j < n; j++ )
3.753	1.806	13. y[j] += A[i+m*j] * x[i];
0.	0.	14. }
		15.

The status bar at the bottom shows "Local Host:", "Remote Host:", "Working Directory: ...ver1", "Compare: off", "Filters: off", and a "Warning" icon.

# 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

# Конфликты конвейера (Hazards)

---

- **Конфликт конвейера (Hazard)** – ситуация, когда выполнение следующей инструкции не может быть начато/продолжено
- **Виды конфликтов:**
  - Структурные конфликты (Structural Hazards)
  - Конфликты данных (Data Hazards)
  - Конфликты управления (Control Hazards)

**Конфликты являются причиной замедления  
работы конвейера**

# Структурные конфликты (Structural Hazards)

- Некоторые исполняющие модули процессора могут разделяться несколькими этапами (stages) конвейера (для сокращения его стоимости)
- Пример 1:** один модуль доступа к памяти разделяется этапами IF и MEM

Инструкция	1	2	3	4	5	6	7	8	
I1 (Load)	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				IF	ID	EX	MEM	WB	

- Инструкция **I1** на такте 4 обратилась к памяти (MEM)
- В тоже время инструкция **I4** должна быть выбрана из памяти (IF)
- Так как порт доступа к памяти один => **структурный конфликт**

# Структурные конфликты (Structural Hazards)

---

- Для разрешения структурного конфликта процессор вставляет перед инструкцией **I4** (IF) задержку на несколько тактов (pipeline stall/bubbling)

Инструкция	1	2	3	4	5	6	7	8	9
I1 (Load)	IF	ID	EX	MEM	WB				
I2		IF	ID	EX	MEM	WB			
I3			IF	ID	EX	MEM	WB		
I4				STALL	IF	ID	EX	MEM	WB



# 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  
 $R2 = R4 + R7$   
 $R2 = R1 + R3$

# Конфликты данных (Data Hazards)

Instructions (DST, OP1, OP2)		1	2	3	4	5	6	7	8	9
ADD	R1, R2, R3	IF	ID	EX	MEM	WB				
SUB	R4, R5, R1		IF	ID <sub>sub</sub>	EX	MEM	WB	RAW-dep. ✗		
AND	R6, R1, R7			IF	ID <sub>and</sub>	EX	MEM	WB	RAW-dep. ✗	
OR	R8, R1, R9				IF	ID <sub>or</sub>	EX	MEM	Запись R1 на первой половине такта 5, чтение – на второй	
XOR	R10, R1, R11					IF	ID <sub>xor</sub>	EX	MEM	WB

Состояния конвейера на тактах 1..9

Запись в R1 будет закончена только на такте 5

# Конфликты данных (Data Hazards)

---

## Устранение конфликтов

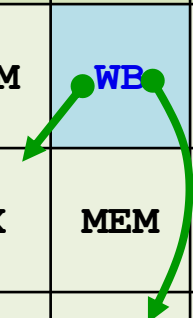
1. Запись в регистровый файл на первой половине такта 5, а чтение из него – на второй (пример с OR на пред. слайде, такт 5)
2. Техника ***forwarding*** (bypassing, short-circuiting)
3. Внеочередное исполнение команд (Out-of-order execution: register renaming, ...)

# Конфликты данных (Data Hazards)

## Forwarding

- Техника **forwarding** (bypassing, short-circuiting) – в конвейере реализуется возможность передачи значений от инструкции к инструкции минуя регистровый файл
- Выход некоторых функциональных устройств аппаратурно связывается со входом других (ALU → EX, ALU → MEM, MEM → MEM, ...)

Instruction (DST, OP1, OP2)		1	2	3	4	5	6	7	8	9
ADD	R1, R2, R3	IF	ID	EX	MEM	WB				
SUB	R4, R5, R1		IF	ID <sub>sub</sub>	EX	MEM	WB			
AND	R6, R1, R7			IF	ID <sub>and</sub>	EX	MEM	WB		



# Внеочередное выполнение команд (OOE)

---

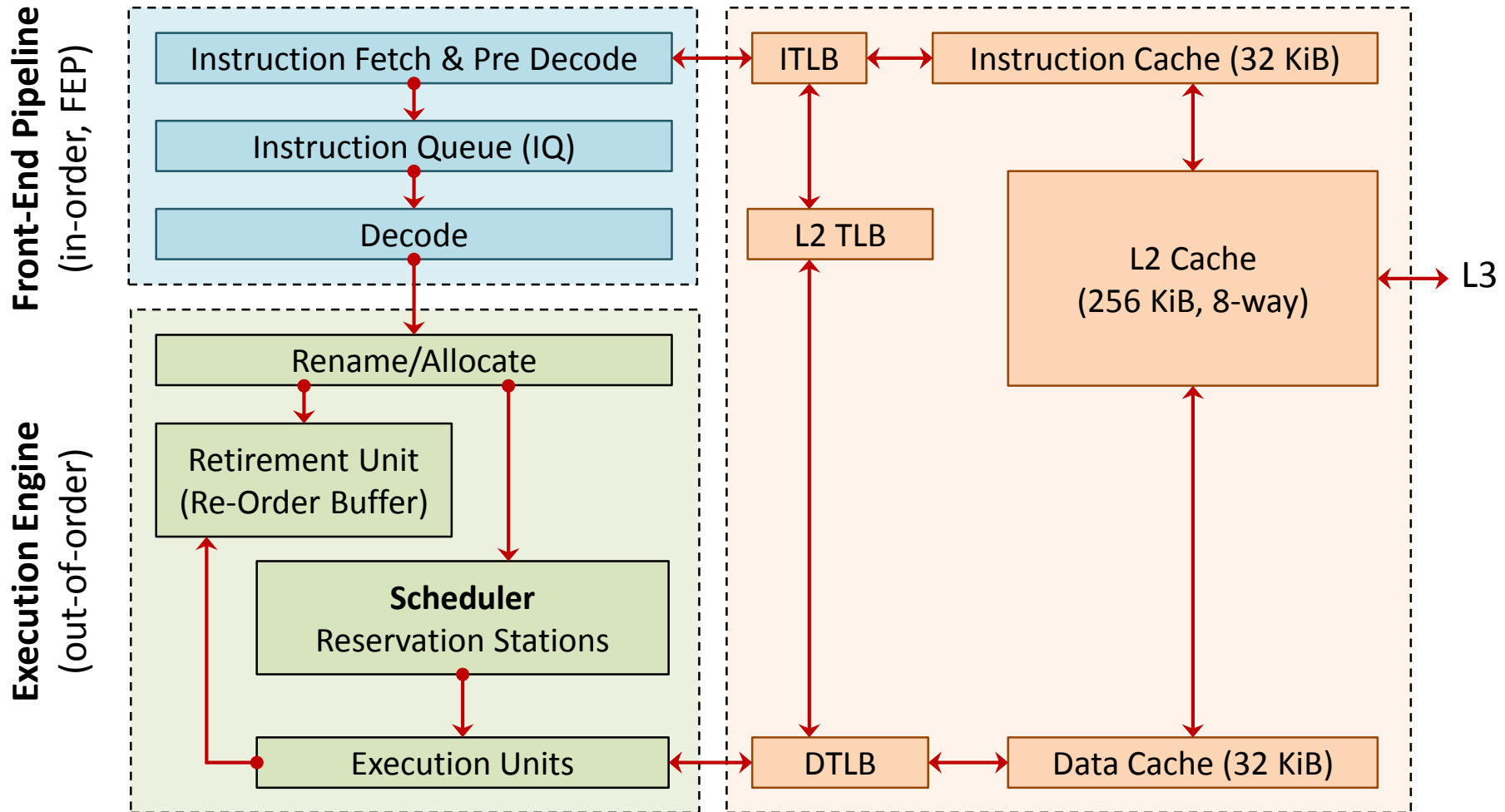
- **Внеочередное выполнение команд** (Out-of-order execution, dynamic scheduling, OOE) – это подход к сокращению простоев конвейера (stall) за счет внеочередного выполнения инструкций по готовности их данных (ограниченная форма dataflow-вычислений)
- Позволяет скрыть задержки при доступе к памяти
- **CDC 6000**: 1964, метод scoreboard – ограниченные возможности по разрешению структурных конфликтов, конфликтов WAR, WAW
- **IBM 360/91**: 1967, алгоритм Роберта Томасуло (R. Tomasulo, 1967) для полной поддержки внеочередного выполнения команд
- **IBM POWER1**: 1990, первый OOE-микропроцессор
- В 1990-х OOE широко применяется в процессорах:
  - IBM PowerPC 601 (1993)
  - Intel Pentium Pro (1995)
  - AMD K5 (1996)
  - DEC Alpha (1998)

# Внеочередное выполнение команд (ООЕ)

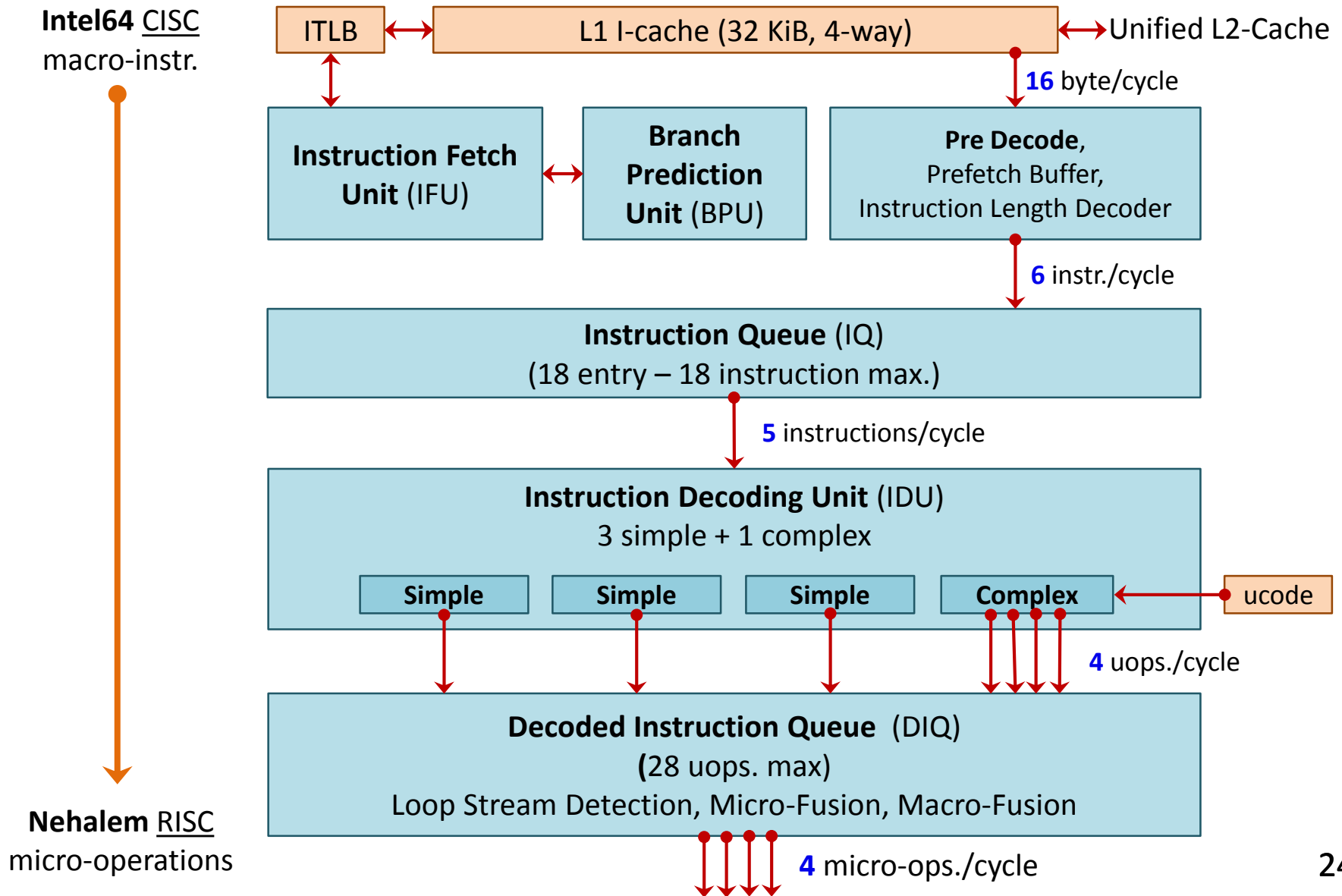
---

1. Инструкция выбирается из памяти (одна или несколько)
2. Инструкция направляется (dispatch) в *очередь инструкций* (instruction queue, instruction buffer, reservation station)
3. Находясь в очереди инструкция ожидает пока её операнды станут доступными. После чего инструкция может покинуть очередь раньше более старых команд
4. Инструкция направляется на подходящее исполняющее устройство
5. Результаты выполнения инструкции помещаются в очередь
6. Инструкция записывает данные в регистровый файл, только после того как более старые инструкции сохранили свои результаты (retire stage)

# Intel Nehalem Core Pipeline

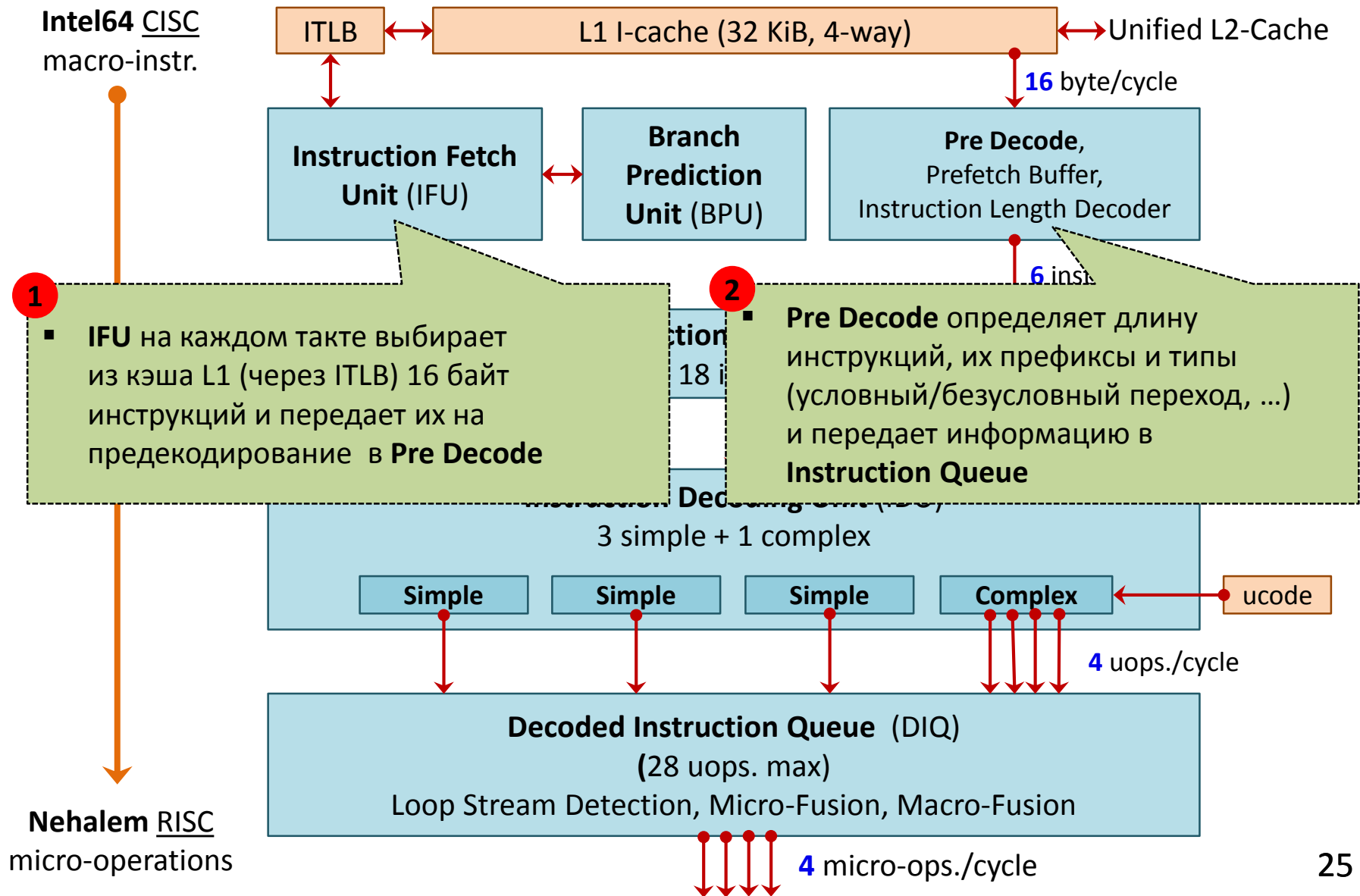


# Intel Nehalem Frontend Pipeline (in-order)



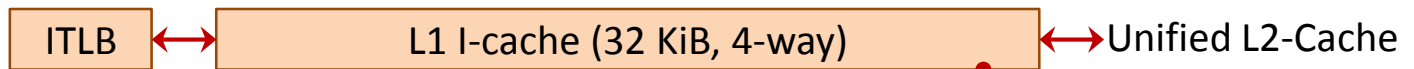


# Intel Nehalem Frontend Pipeline (in-order)



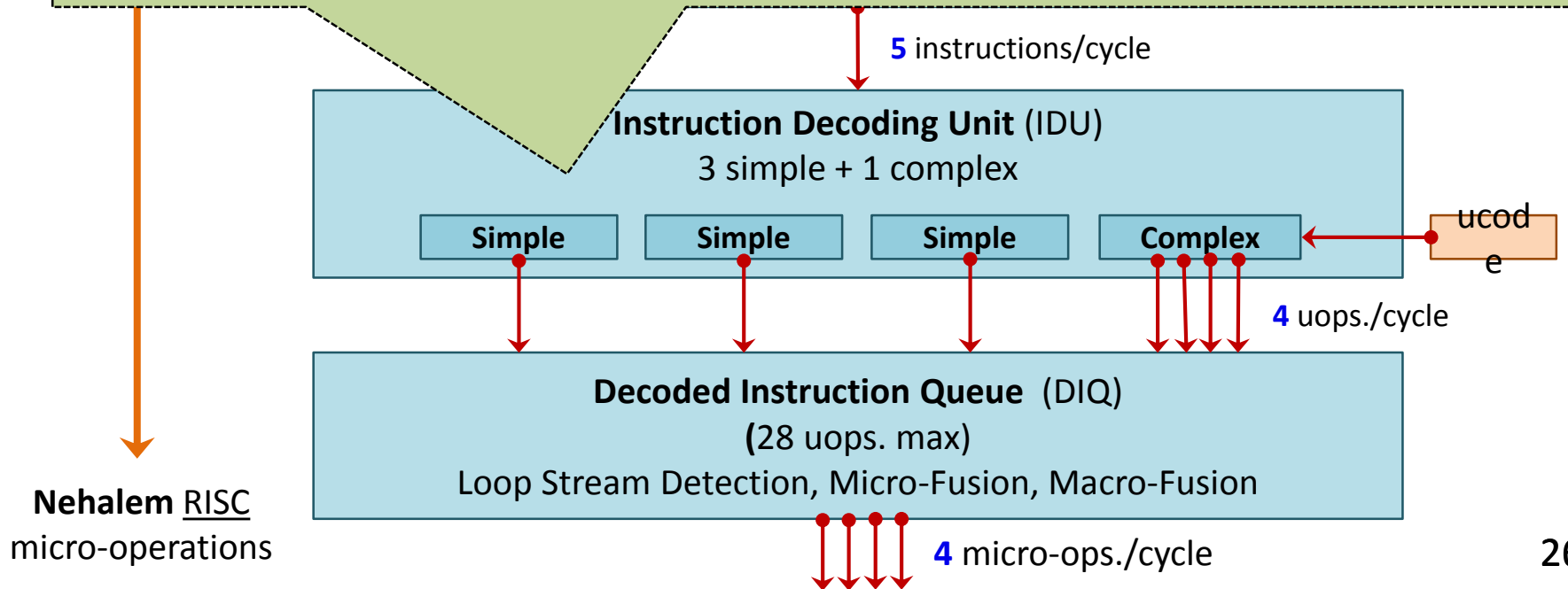
# Intel Nehalem Frontend Pipeline (in-order)

Intel64 CISC  
macro-instr.

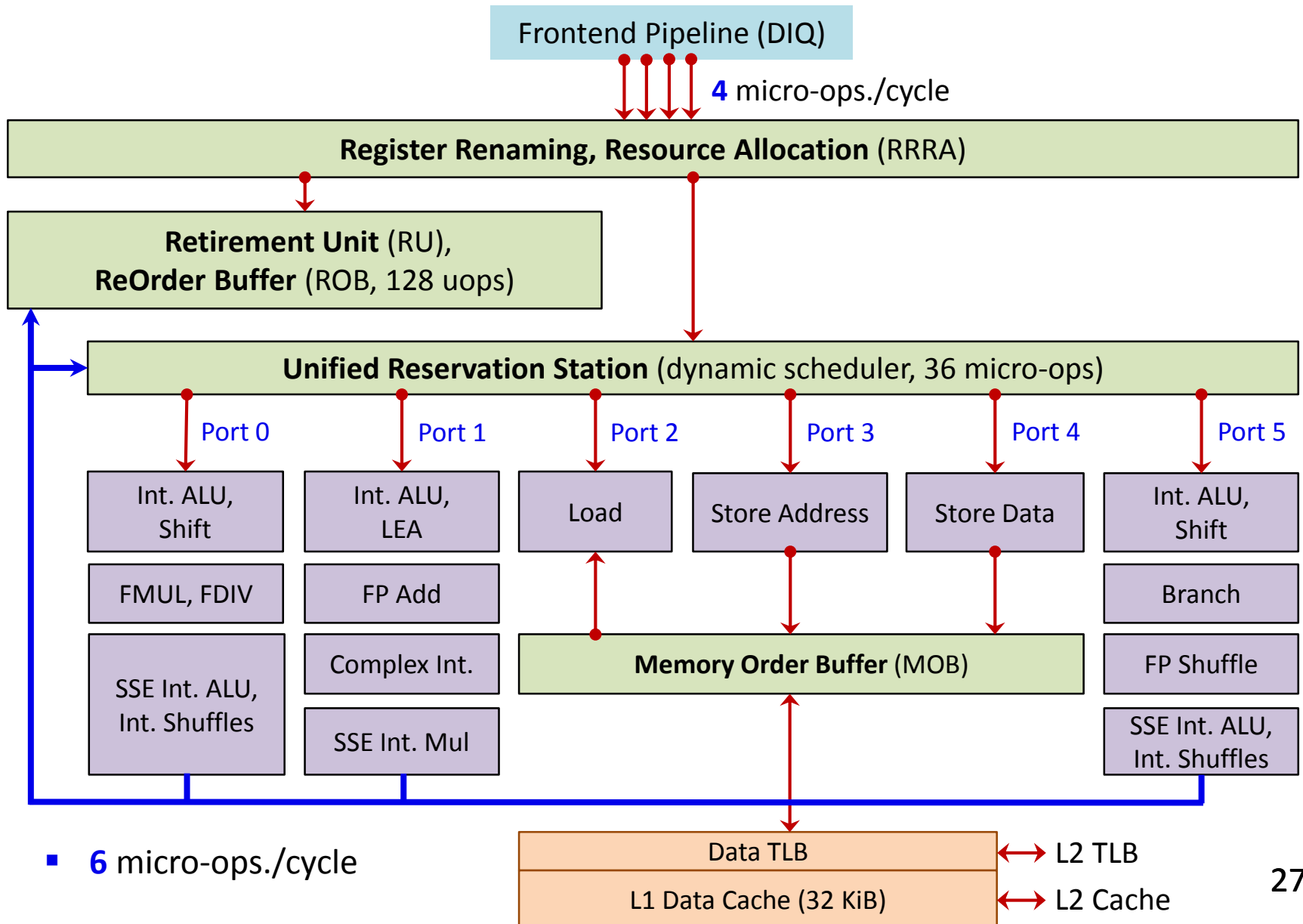


3

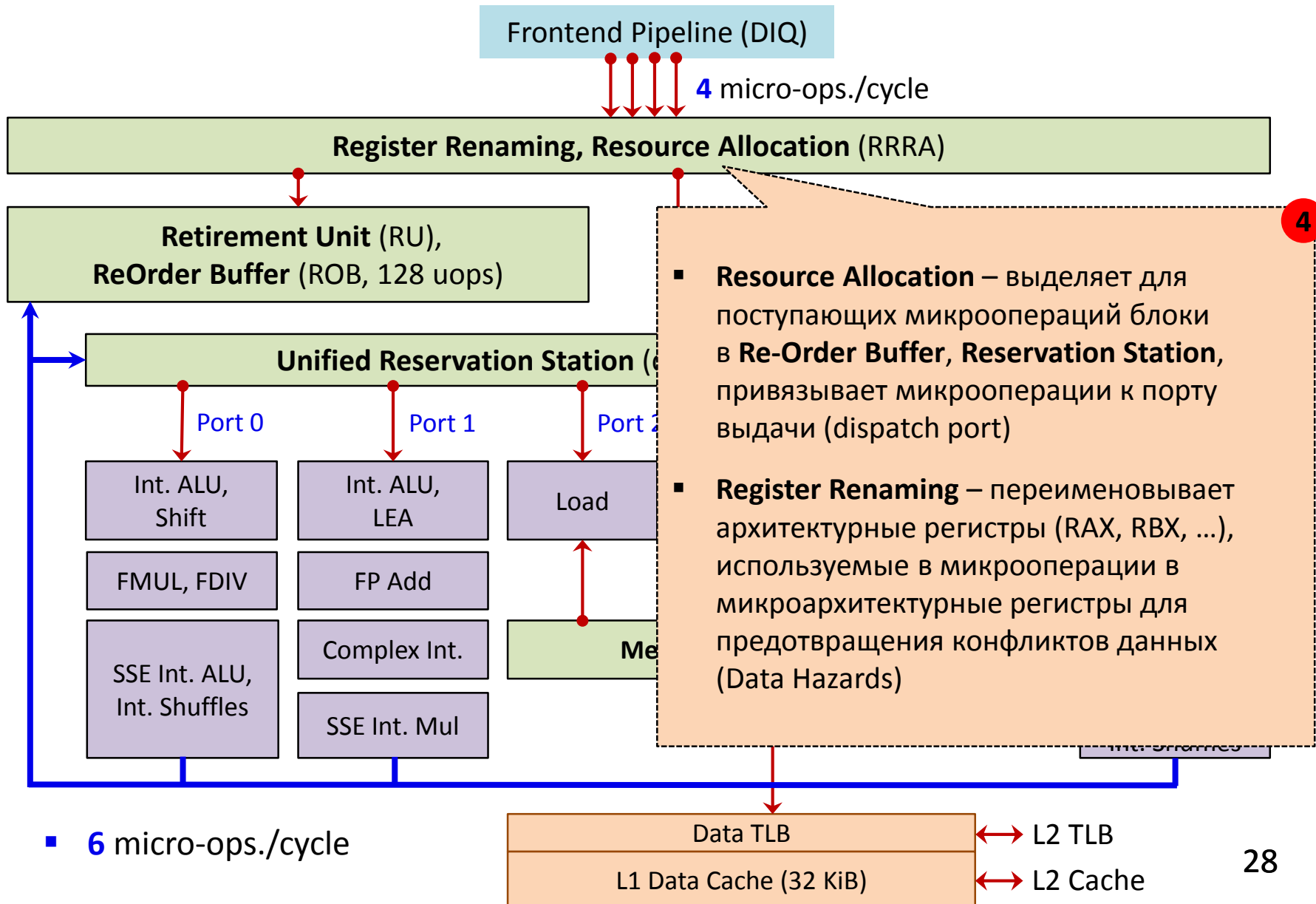
- **IDU** преобразует Intel64-инструкции в микрооперации (uops). Сложные инструкции преобразуются в несколько микроопераций.
- **IDU** передает микрооперации в очередь **DIQ**, где выполняется поиск циклов (LSD, для предотвращения их повторного декодирования), слияние микроопераций (для увеличения пропускной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



# Intel Nehalem Execution Engine



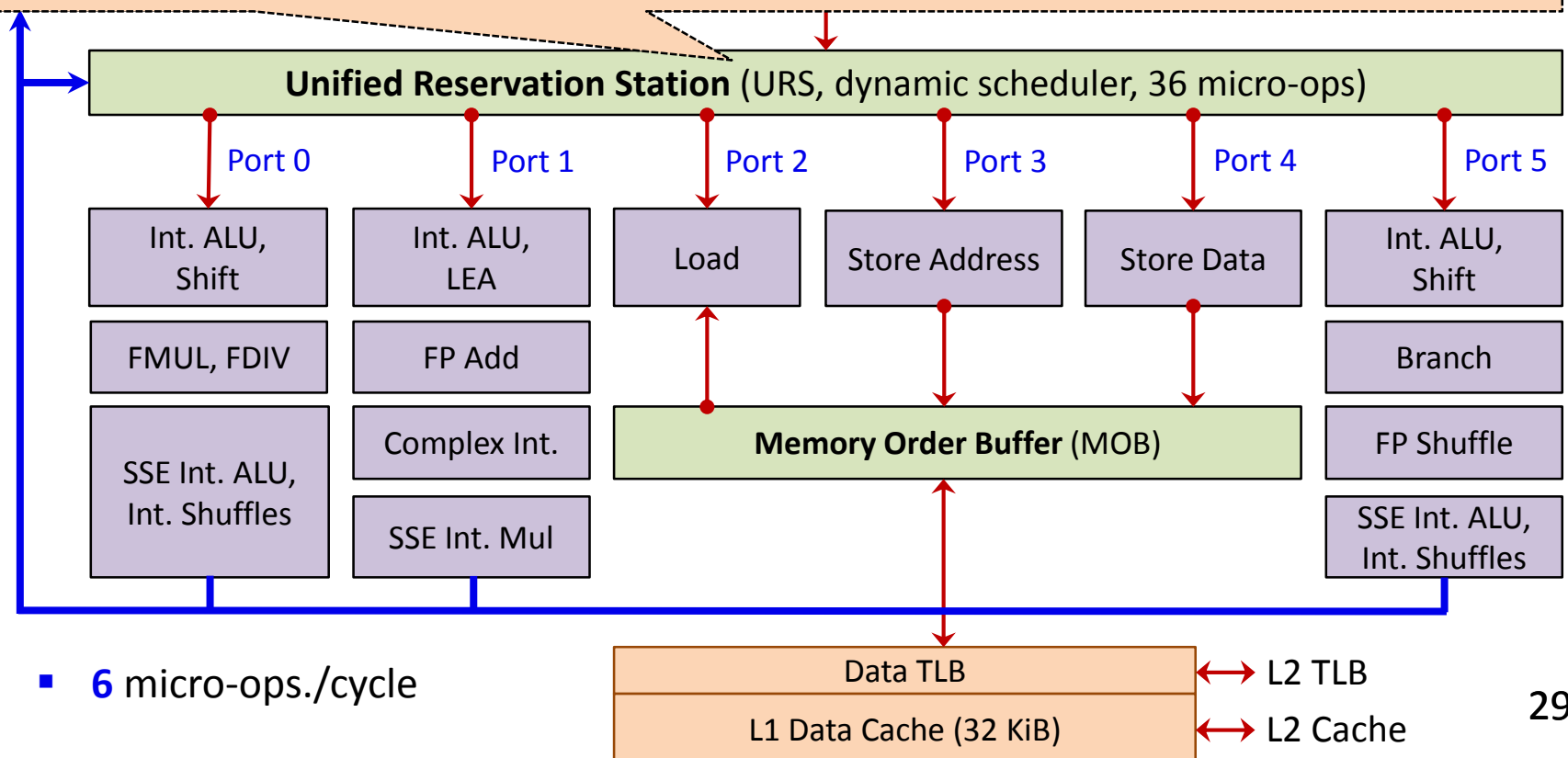
# Intel Nehalem Execution Engine



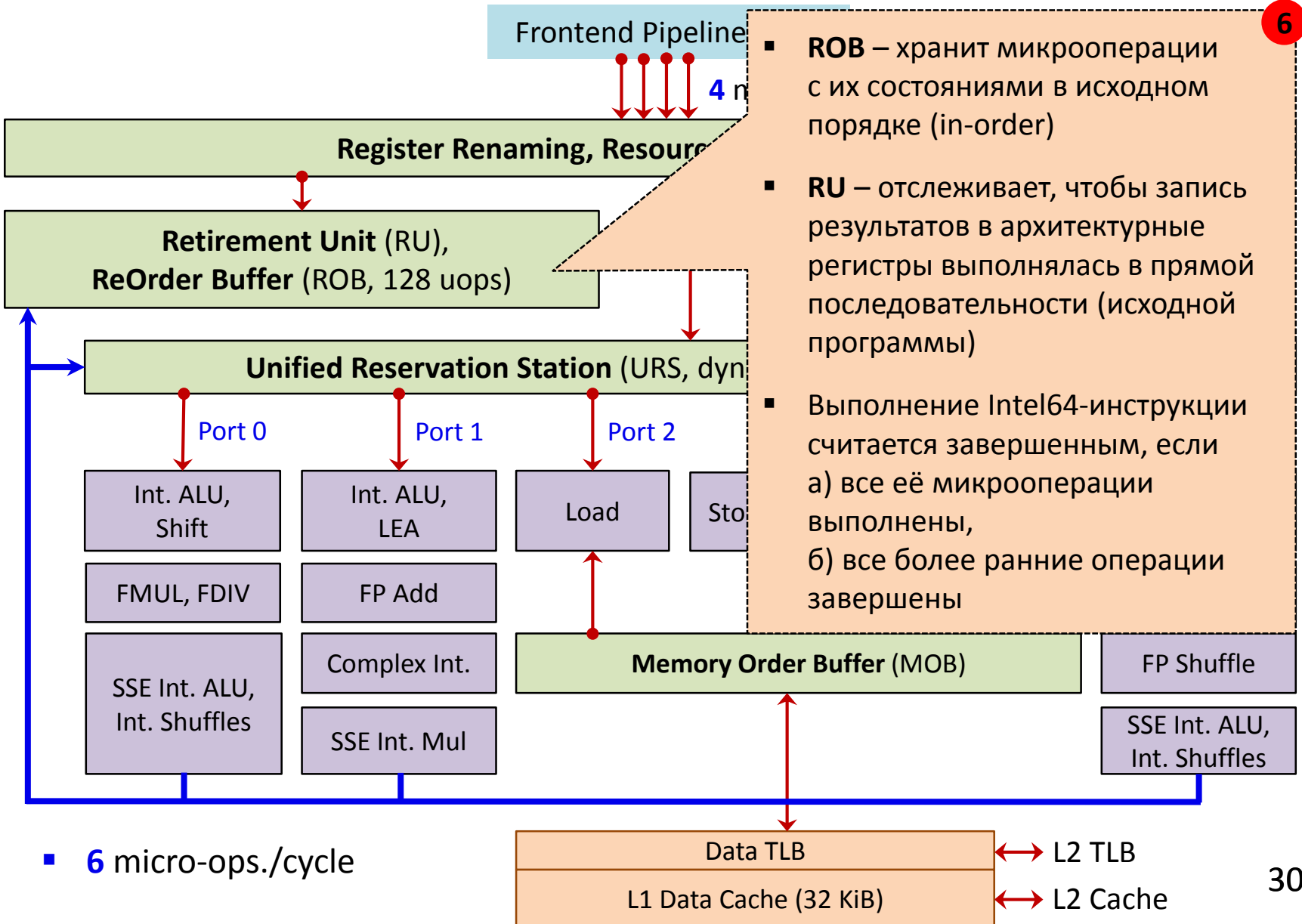
# Intel Nehalem Execution Engine

5

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



# Intel Nehalem Execution Engine



# Intel Architecture Code Analyzer

---

- **Intel Architecture Code Analyzer** – статический анализатор кода, позволяющий анализировать зависимость по данным, латентность и пропускную способность инструкций участков (kernels) вашей программы

```
#include "iacaMarks.h"
IACA_START
for (int i = 0; i < n; i++) {
    /* Code */
}
IACA_END
```

```
$ gcc -o prog ./prog.c
$ iaca -64 -arch SNB -analysis LATENCY ./prog
```

# Intel Architecture Code Analyzer

## Latency Analysis Report

Latency: 7 Cycles

Inst Num	Resource Delay In Cycles							FE	
	0 - DV	1	2 - D	3 - D	4	5			
0								CP	vpadd xmm0, xmm0, xmm1
1								CP	vpadd xmm0, xmm0, xmm2
2								CP	vpadd xmm0, xmm0, xmm3
3								CP	vpadd xmm0, xmm0, xmm4
4								CP	vpadd xmm0, xmm0, xmm5
5								CP	vpadd xmm0, xmm0, xmm6
6								CP	vpadd xmm0, xmm0, xmm7

## Resource Conflict on Critical Paths:

Port	0 - DV	1	2 - D	3 - D	4	5
Cycles	0 0	0	0 0	0 0	0	0

## List Of Delays On Critical Paths

Intel Architecture Code Analyzer User's Guide



# Конфликты управления (Control hazards)

```
1:  movl  %ebx, %eax
2:  cmpl  $0x10, %eax
3:  jne   not_equal
4:  movl  %eax, %ecx
5:  jmp   end
```

not\_equal:

```
6:  movl  $-0x1, %ecx
```

end: ...

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

Step 4: результат `cmpl` еще не известен; по какому адресу выбирать следующую инструкцию – 4 или 6?

В процессоре присутствует  
**модуль предсказания переходов (Branch Prediction Unit)**

# Предсказание переходов (Branch prediction)

---

- **Модуль предсказания условных переходов** (Branch Prediction Unit, BPU) – модуль процессора, определяющий по адресу инструкции ветвления *будет ли выполнен переход и по какому адресу*
- Предсказывает условные переходы, вызовы/возвраты из функций
- Вероятность правильного предсказания переходов в современных процессорах превышает 0.9
- После предсказания процессор начинает спекулятивно выполнять инструкции (Speculative execution)
- **Альтернативный подход (без BPU)** – спекулятивно выполнять обе ветви ветвления, пока не будет вычислено управляющее выражение (условие)

# Branch Prediction Unit (BPU)

---

- **Статическое предсказание (Static prediction)** – фиксированное правило работы предсказателя (например, условный переход не выполняются никогда)
- Статические методы предсказания используются когда невозможно задействовать динамические
- **Ранние SPARC, MIPS:** условные переходы никогда не выполняются – сразу начинается выборка следующей инструкции
- **Современные CPU:** обратный переход (переход на более младшие адреса, backwards branch), является циклом и выполняется, а любой прямой переход (на более старшие адреса, forward-pointing branch), не выполняется

# Статическое предсказание переходов

---


## Intel 64 and IA-32 Architectures Optimization Reference Manual

- Процессоры Pentium 4, Pentium M, Intel Core Solo и Intel Core Duo имеют схожие алгоритмы статического предсказания переходов:
  - безусловные переходы выполняются (to be taken)
  - косвенные переходы (indirect jump, jmp %eax) не выполняется
  - условные переходы предсказываются динамическим алгоритмом (даже при первом выполнении)

# Динамическое предсказание переходов

- **Динамическое предсказание (Dynamic prediction)** – такие методы осуществляют накопление и анализ истории ветвлений
- Информация о предыдущих ветвлениях хранится в буфере предсказания переходов (Branch Target Buffer – BTB)
- **BTB** – это ассоциативный массив (хеш-таблица), сопоставляющий адресу инструкции ветвления историю переходов и адрес перехода

```
0xFF01 11:  movl %ebx, %eax
0xFF02      cmpl $0x10, %eax
0xFF03      jne  12
0xFF04      movl %eax, %ecx
0xFF05      jmp  13
0xFF06 12:  movl $-0x1, %ecx
0xFF07 13:
0xFF08      cmpl $0xFF, %ebx
0xFF09      jne  11
```




Instr. Address (low bits)	History	Target Address
0xFF03	1	0xFF06
0xFF09	0	

# Динамическое предсказание переходов

- На этапе IF по адресу инструкции (Instruction pointer – IP) происходит обращение в BTB:
  - Если запись для IP есть, значит загруженная инструкция – это ветвление и в BTB имеется адрес перехода (Target address)
  - Далее используя историю ветвлений принимается решение осуществлять переход или нет

```
0xFF01 11:  movl %ebx, %eax
0xFF02      cmpl $0x10, %eax
0xFF03      jne  12
0xFF04      movl %eax, %ecx
0xFF05      jmp  13
0xFF06 12:  movl $-0x1, %ecx
0xFF07 13:
0xFF08      cmpl $0xFF, %ebx
0xFF09      jne  11
```



Instr. Address (low bits)	History	Target Address
0xFF03	1	0xFF06
0xFF09	0	

# 1-Bit dynamic predictor

Адрес инструкции перехода (IP):



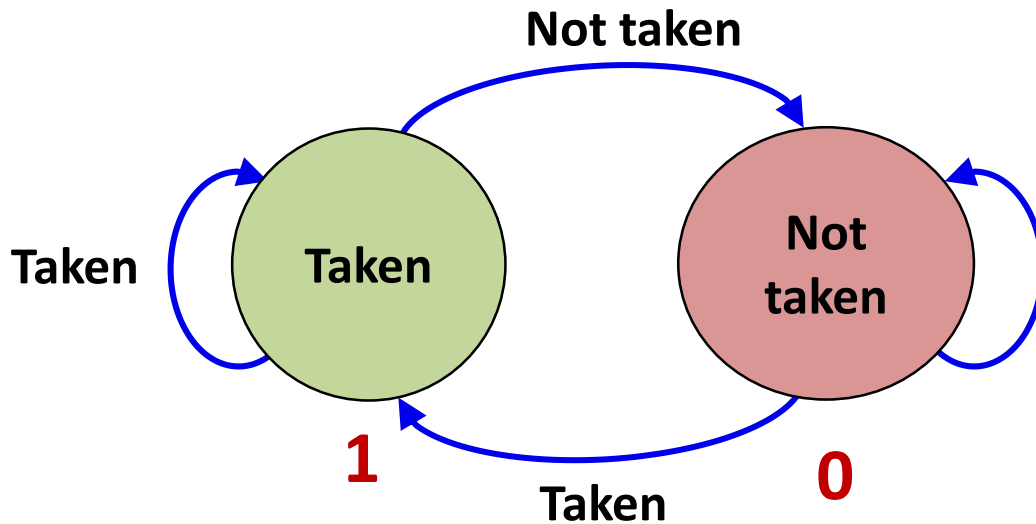
Branch Target Buffer (BTB)

Record	Branch History	Target
0	1	0xAF06
1	0	0x1134
2	1	0x01FC
3	0	0xFF06
...		
$2^k - 1$	1	0xBEAF

Branch History (1 bit)

- 0 – ветвление не состоялось, не осуществлять переход
- 1 – ветвление состоялось, осуществлять переход

# 1-Bit dynamic predictor



```
movl $0, %ecx
jmp .L2

.L1:
    /* code */
    addl $1, %ecx
.L2:  cmpl $8, %ecx
      jle .L1
```

$i = 0$ , predicted 0, **MISPREDICTION** (state  $\rightarrow$  1)

$i = 1$ , predicted 1, **OK**, (state  $\rightarrow$  1)

$i = 2$ , predicted 1, **OK**, (state  $\rightarrow$  1)

$i = 3$ , predicted 1, **OK**, (state  $\rightarrow$  1)

...

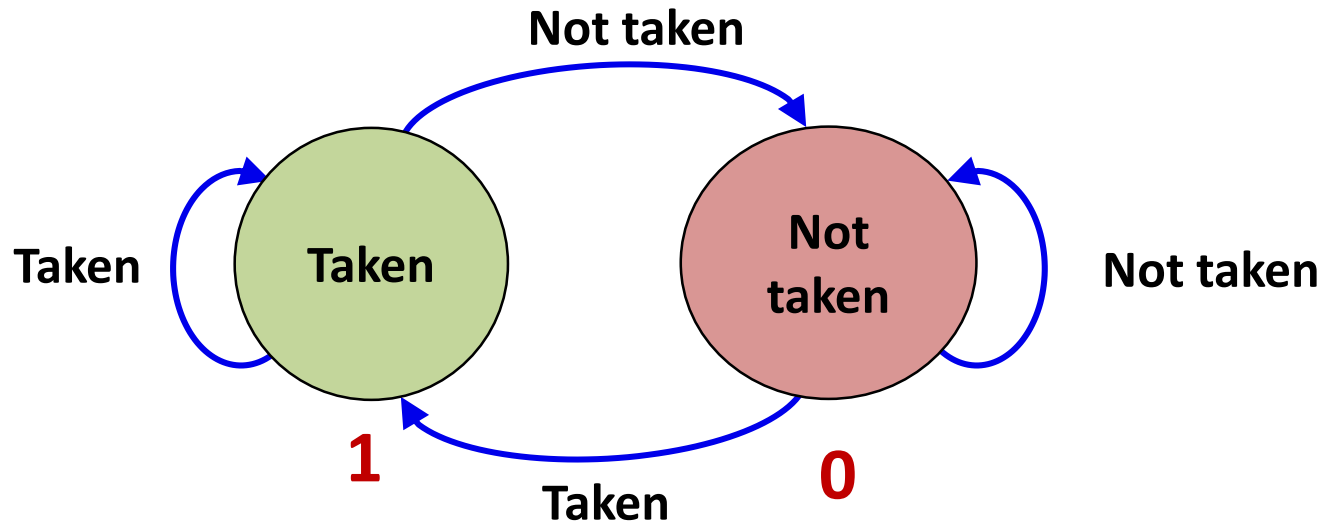
$i = 9$ , predicted 1, **MISPREDICTION** (state  $\rightarrow$  0)

**80% ветвлений  
предсказано  
корректно**



# 1-Bit dynamic predictor

---



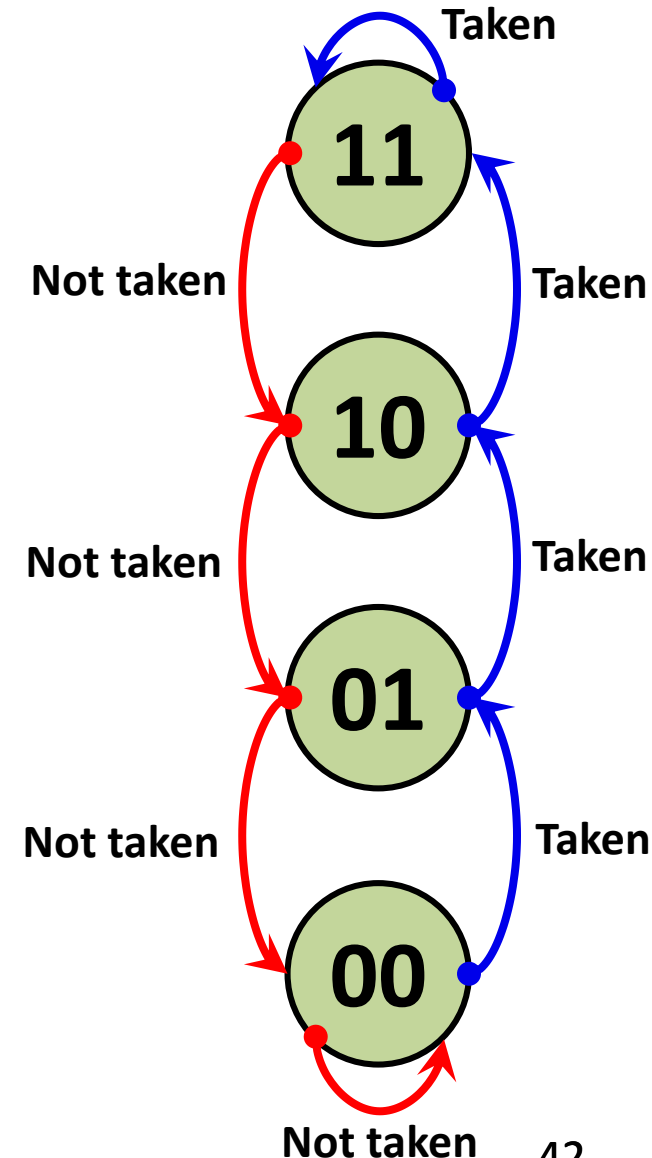
```
for (i = 0; i < 6; i++) {  
    if ((i & 1) == 0)  
        /* Code 1 */  
    else  
        /* Code 2 */  
}
```

Точность 0%

I	Predicted	Real
0	0 (NOT TAKEN)	1 (TAKEN)
1	1	0
2	0	1
3	1	0
4	0	1
5	1	0

# Saturating 2-bit counter (Bimodal predictor)

- При выполнении перехода (Taken) состояние увеличивается на 1
- При невыполнении перехода (Not taken) состояние уменьшается на 1
- **Предсказание:**  
**{00, 01}:**  
переход не выполняется  
  
**{10, 11}:**  
переход выполняется
- Использовался в Intel Pentium
- Можно обобщить на случай  $n$ -битного предсказателя:  
если значение счетчика  $\geq$  половины своего максимального значения, то ветвление выполняется



# Реализации ВТВ

---

- **Intel Pentium:** saturating 2-bit counter,
- **Intel Pentium {MMX, Pro, II, III}:**  
two-level adaptive branch predictor (4-bit history)
- **Pentium 4:** Agree predictor (16-bit global history)
- **Intel Atom:** two-level adaptive branch predictor
- **Intel Nehalem:** two-level branch predictor, misprediction penalty is at least 17 clock cycles

Agner Fog. **The microarchitecture of Intel, AMD and VIA CPUs**  
(an optimization guide for assembly programmers and compiler makers) //  
<http://www.agner.org/optimize/microarchitecture.pdf>

# Ветвления в программах

---

```
if (x == 0)
else if (x == 1)
else
```

```
switch (x) {
    case 0:
        break;
    case 1:
        break;
    default:
}
}
```

```
for (i = 0; i < 10; i++) {
}
```

```
while (data > 0) {
    data--;
}
```

```
do {
    data--;
} while (data > 0);
```

# Ветвления в программах (gcc 5.1.1)

---

```
// prog.c
if (a > 100)
    printf("Case 1\n");
else
    printf("Case 2\n");
```

```
$ gcc -O0 -o prog ./prog.c --save-temps
```



```
// prog.s
    cmpl    $100, -4(%rbp)
    jle     .L2
    movl    $.LC0, %edi
    call    puts
    jmp     .L3
.L2:      movl    $.LC1, %edi
    call    puts
.L3:
```

# Ветвления в программах (gcc 5.1.1)

---

```
// prog.c
if (a > 100)
    printf("Case 1\n");
else
    printf("Case 2\n");
```

```
$ gcc -O2 -o prog ./prog.c --save-temps
```



```
// prog.s
        cmp1    $100, %eax
        jg      .L6
        movl    $.LC1, %edi
        call    puts
.L3:
        // ...
        ret
.L6:
        movl    $.LC0, %edi
        call    puts
        jmp     .L3
```

# Циклы в программах (gcc 5.1.1)

---

```
// prog.c
for (int i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

```
// prog.s, -O0
        movl    $0, -4(%rbp)
        jmp     .L2

.L3:
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        movl    $.LC0, %edi
        movl    $0, %eax
        call    printf
        addl    $1, -4(%rbp)

.L2:
        cmpl    $9, -4(%rbp)
        jle     .L3
```

```
// prog.s, -O2
.L2:
        movl    %ebx, %esi
        xorl    %eax, %eax
        movl    $.LC0, %edi
        addl    $1, %ebx
        call    printf
        cmpl    $10, %ebx
        jne     .L2
```

# Оптимизация инвариантных ветвлений

---

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

**Сколько будет выполнено условных переходов?**



# Оптимизация инвариантных ветвлений

---

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

## Условные переходы:

- |                |                |
|----------------|----------------|
| 1. 0 < 10      | 13. 6 < 10     |
| 2. value > 10  | 14. value > 10 |
| 3. 1 < 10      | 15. 7 < 10     |
| 4. value > 10  | 16. value > 10 |
| 5. 2 < 10      | 17. 8 < 10     |
| 6. value > 10  | 18. value > 10 |
| 7. 3 < 10      | 19. 9 < 10     |
| 8. value > 10  | 20. value > 10 |
| 9. 4 < 10      | 21. 10 < 10    |
| 10. value > 10 |                |
| 11. 5 < 10     |                |
| 12. value > 10 |                |

**21 условный  
переход**

# Оптимизация инвариантных ветвлений

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

Инвариантное ветвление –  
не зависит от состояние программы  
внутри цикла

## Условные переходы:

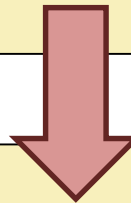
- |                |                |
|----------------|----------------|
| 1. 0 < 10      | 13. 6 < 10     |
| 2. value > 10  | 14. value > 10 |
| 3. 1 < 10      | 15. 7 < 10     |
| 4. value > 10  | 16. value > 10 |
| 5. 2 < 10      | 17. 8 < 10     |
| 6. value > 10  | 18. value > 10 |
| 7. 3 < 10      | 19. 9 < 10     |
| 8. value > 10  | 20. value > 10 |
| 9. 4 < 10      | 21. 10 < 10    |
| 10. value > 10 |                |
| 11. 5 < 10     |                |
| 12. value > 10 |                |

**21 условный  
переход**

# Оптимизация инвариантных ветвлений

```
for (i = 0; i < 10; i++) {  
    if (value > 10)  
        data++;  
    else  
        data--;  
}
```

**21 условный  
переход**



```
if (value > 10) {  
    for (i = 0; i < 10; i++)  
        data++;  
} else {  
    for (i = 0; i < 10; i++)  
        data--;  
}
```

**12 условных переходов**

Меньше обращений  
к модулю предсказания  
переходов (BPU)

**value > 10**  
**0 < 10**  
**1 < 10**  
...  
**10 < 10**

# Оптимизация инвариантных ветвлений

---

```
void MyFunc(int size, int blend, float *src,
            float *dest, float *src_1, ...)
{
    int j;

    for (j = 0; j < size; j++) {
        if (blend == 255)
            dest[j] = src_1[j];
        else if ( blend == 0 )
            dest[j] = src_2[j];
        else
            dest[j] = (src_1[j] * blend + src_2[j]
                       * (255 - blend)) / 256;
    }
}
```

# Оптимизация инвариантных ветвлений

```
void MyFunc(int size, int blend, float *src,  
            float *dest, float *src_1, ... )  
{  
    int j;  
  
    if (blend == 255)  
        for (j = 0; j < size; j++)  
            dest[j] = src_1[j];  
    else if (blend == 0)  
        for (j = 0; j < size; j++)  
            dest[j] = src_2[j];  
    else  
        for (j = 0; j < size; j++)  
            dest[j] = (src_1[j] * blend + src_2[j]  
                      * (255 - blend)) / 256;  
}
```

- Инвариантное ветвление вынесли за цикл
- Сократили число переходов

# Типовые ошибки

```
void fun()  
{  
    if (t1 == 0 && t2 == 0 && t3 == 0) {  
        /* Code 1 */  
    } else {  
        /* Code 2 */  
    }  
}
```

```
    cmpl    $0, -4(%rbp)  
    jne     .L2  
    cmpl    $0, -8(%rbp)  
    jne     .L2  
    cmpl    $0, -12(%rbp)  
    jne     .L2  
    /* Code 1 */  
    jmp     .L3  
.L2:  
    /* Code 2 */  
.L3:
```

**3 ветвления**

# Типовые ошибки

```
void fun()  
{  
    if ((t1 | t2 | t3) == 0) {  
        /* Code 1 */  
    } else {  
        /* Code 2 */  
    }  
}
```

```
    movl    -8(%rbp), %eax  
    movl    -4(%rbp), %edx  
    orl     %edx, %eax  
    orl     -12(%rbp), %eax  
    testl   %eax, %eax  
    jne     .L2  
    movl    $133, -16(%rbp)  
    jmp     .L3  
.L2:  
    movl    $333, -16(%rbp)  
.L3:
```

1 переход

# GCC branch annotation

---

```
#define likely(x)    __builtin_expect (!!(x), 1)
#define unlikely(x)  __builtin_expect (!!(x), 0)
```

```
const char *home;

home = getenv("HOME");
if (likely(home))
    printf("Your home is %s\n", home);
else
    fprintf (stderr, "HOME not set!\n");
```



# GCC branch annotation

---

```
#define likely(x)      __builtin_expect (!!(x), 1)
#define unlikely(x)  __builtin_expect (!!(x), 0)
```

```
unsigned int __skb_checksum_complete(struct sk_buff *skb)
{
    unsigned int sum;
    sum = (u16)csum_fold(skb_checksum(skb, 0, skb->len,
                                     skb->csum));

    if (likely(!sum)) {
        if (unlikely(skb->ip_summed == CHECKSUM_HW))
            netdev_rx_csum_fault(skb->dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY;
    }
    return sum;
}
```

# Hardware Performance Counters

---

- Linux perf
- Linux MSR Tools (rdmsr, wrmsr)
- Intel VTune
- PAPI

```
$ perf list
```

```
List of pre-defined events (to be used in -e):
```

cpu-cycles OR cycles	[Hardware event]
instructions	[Hardware event]
cache-references	[Hardware event]
cache-misses	[Hardware event]
branch-instructions OR branches	[Hardware event]
branch-misses	[Hardware event]
bus-cycles	[Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend	[Hardware event]
stalled-cycles-backend OR idle-cycles-backend	[Hardware event]
ref-cycles	[Hardware event]
...	

# Linux Perf (Sampling)

---

```
$ perf stat -e branch-misses ./prog
```

```
Performance counter stats for './prog':
```

709.737801	task-clock	#	0.995 CPUs utilized
81	context-switches	#	0.114 K/sec
5	CPU-migrations	#	0.007 K/sec
2,379	page-faults	#	0.003 M/sec
2,106,950,459	cycles	#	2.969 GHz
398,589,229	branches	#	561.601 M/sec
1,280,498	branch-misses	#	0.32% of all

<http://perf.wiki.kernel.org>

# Loop peeling (Loop splitting)

---

```
int p = 10;
for (int i = 0; i < 10; ++i) {
    y[i] = x[i] + x[p];
    p = i;
}
```

Только на первой  
итерации  $p = i$ ,  
затем  $p = i - 1$



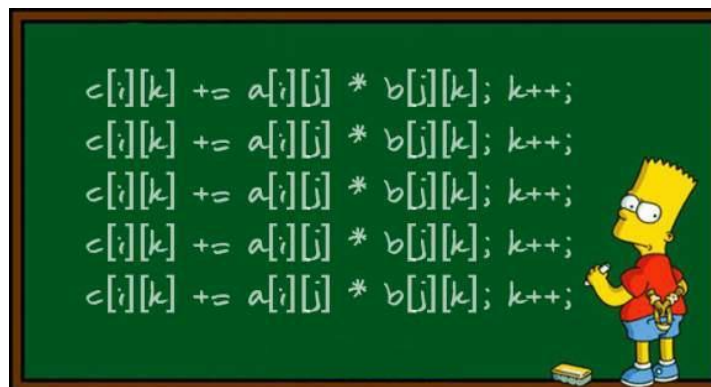
```
y[0] = x[0] + x[10];
for (int i = 1; i < 10; ++i) {
    y[i] = x[i] + x[i - 1];
}
```

- **Loop peeling** – это оптимизация цикла путем выноса из его тела проблемных/избыточных итераций

# Разворачивание циклов (Loop unrolling)

---

- **Loop unrolling (раскрутка, unwinding)** – это преобразование цикла, при котором код его тела многократно тиражируется
- Количество итераций цикл сокращается (за одну итерацию выполняем сразу несколько)
- Раскрутка цикла позволяет:
  - эффективнее загрузить суперскалярный процессор независимыми операциями
  - сократить количество ветвлений при выполнении цикла



# Разворачивание циклов (Loop unrolling)

---

```
enum { n = 40 * 1024 * 1024 };

int main()
{
    int *p;
    int i, sum;

    p = (int *)malloc(sizeof(*p) * n);
    for (i = 0; i < n; i++)
        p[i] = rand();

    for (sum = 0, i = 0; i < n; i++) {
        sum += p[i];
    }
    return 0;
}
```

# Разворачивание циклов (Loop unrolling)

```
enum { n = 40 * 1024 * 1024 };

int main()
{
    int *p;
    int i, sum;

    p = (int *)malloc(sizeof(*p) * n);
    for (i = 0; i < n; i++)
        p[i] = rand();

    for (sum = 0, i = 0; i < n; i += 4) {
        sum += p[i];
        sum += p[i + 1];
        sum += p[i + 2];
        sum += p[i + 3];
    }
    return 0;
}
```

Количество ветвлений ( $i < n$ )  
сократилось в 4 раза

# Разворачивание циклов (Loop unrolling)

```
enum { n = 40 * 1024 * 1024 };

int main()
{
    int *p;
    int i, sum;

    p = (int *)malloc(sizeof(*p) * n);
    for (i = 0; i < n; i++)
        p[i] = rand();

    for (sum = 0, i = 0; i < n; i += 4) {
        sum += p[i];
        sum += p[i + 1];
        sum += p[i + 2];
        sum += p[i + 3];
    }
    return 0;
}
```

Зависимость по данным  
между инструкциями  
(RAW - Read After Write)



# Разворачивание циклов (Loop unrolling)

```
enum { n = 40 * 1024 * 1024 };

int main()
{
    int *p;
    int i, sum, t1, t2, t3, t4;

    /* Initialization code ... */

    t1 = t2 = t3 = t4 = 0;
    for (sum = 0, i = 0; i < n; i += 4) {
        t1 += p[i];
        t2 += p[i + 1];
        t3 += p[i + 2];
        t4 += p[i + 3];
    }
    sum = t1 + t2 + t3 + t4;
    return 0;
}
```

**Speedup 98 %**

- Intel Core i5 2520M
- Linux x86\_64 (Fedora 19)
- GCC 4.8.1, opt. flags: -O0

# Разворачивание циклов (Loop unrolling)

---

- На какую глубину раскручивать цикл?
- Можно подбирать глубину учитывая количество функциональных устройств суперскалярного процессора (ALU, FPU, ...)
- Если количество итераций цикл не кратно глубине раскрутки, то цикл разбиваем на два – первый выполняет максимально возможное число итераций, кратное глубине раскрутки, а второй остаток итераций

# Разворачивание циклов (Loop unrolling)

---

```
void fun()  
{  
    for (i = 0; i < 1024; i++) {  
        if (i & 1)  
            fun_a(i);  
        else  
            fun_b(i);  
    }  
}
```

# Разворачивание циклов (Loop unrolling)

---

```
void fun()  
{  
    for (i = 0; i < 1024; i += 2) {  
        fun_a(i);  
        fun_b(i + 1);  
    }  
}
```

# Литература

---

- Герберт Р., Бик А., Смит К., Тиан К. **Оптимизация ПО. Сборник рецептов.** - СПб: Питер, 2010. - 352 с.
- Randal E. Bryant, David R. O'Hallaron. **Computer Systems: A Programmer's Perspective.** - Addison-Wesley, 2010
- Agner Fog. **The microarchitecture of Intel, AMD and VIA CPUs** (an optimization guide for assembly programmers and compiler makers) // <http://www.agner.org/optimize/microarchitecture.pdf>
- Michael E. Thomadakis. **The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms** // <http://sc.tamu.edu/systems/eos/nehalem.pdf>
- **Intel® 64 and IA-32 Architectures Optimization Reference Manual** // <http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf>
- Intel 64 and IA-32 Architectures Optimization Reference Manual
- Software Optimization Guide for AMD64 Processors