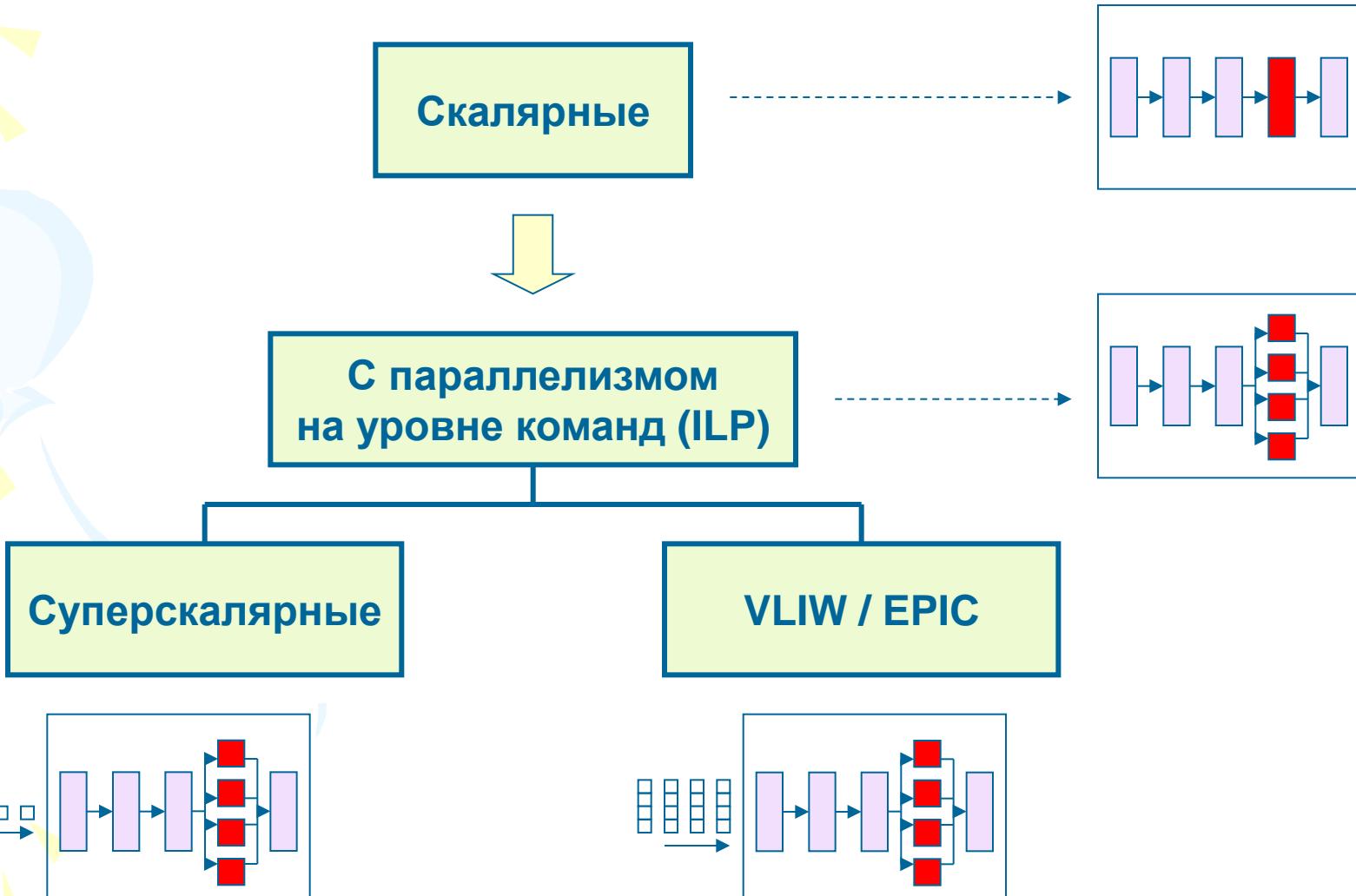




# Архитектура VLIW / EPIC

# Классификация архитектур



# Параллелизм на уровне команд (Instruction Level Parallelism)

## ILP-процессоры

- Имеют несколько исполнительных устройств
- Могут исполнять несколько команд одновременно

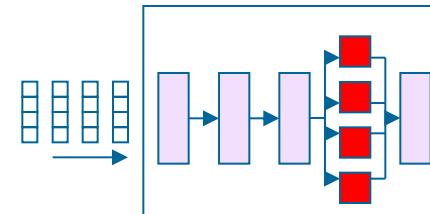
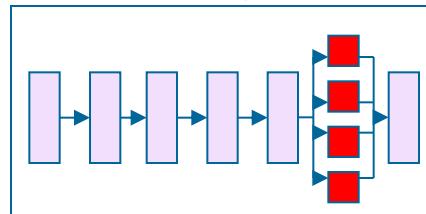
### Суперскалярные процессоры

- Процессор сам распределяет ресурсы

### VLIW / EPIC-процессоры

Very Long Instruction Word /  
Explicitly Parallel Instruction Computing

- Компилятор распределяет ресурсы процессора



# Архитектура VLIW / EPIC

**VLIW** – Very Long Instruction Word

**EPIC** – Explicitly Parallel Instruction Computing

- На входе процессора последовательность больших команд, состоящих из нескольких простых операций, которые могут исполняться параллельно.
- Преимущества перед суперскалярами:
  - Меньше места на процессоре тратится на управление, больше остается на ресурсы: регистры, исполнительные устройства, кэш-память.
  - Более тщательное планирование дает лучшее заполнение исполнительных устройств – больше команд за такт.
- Недостатки:
  - Долгое время компиляции.
  - Сложно учесть динамику исполнения программы.

# Сравнение суперскалярных и VLIW/EPIIC-процессоров

**Какие задачи управления приходится решать, чтобы процессор работал быстро:**

- Параллельное исполнение команд
  - Нужно найти независимые команды
- Спекулятивное исполнение команд
  - Нужно заранее угадать, выполнится ли переход
- Спекулятивная загрузка данных
  - Нужно проверить корректность преждевременной загрузки данных
- Размещение данных на регистрах
  - Нужно оптимально использовать регистры процессора

# Сравнение суперскалярных и VLIW/EPIIC-процессоров

**Какие задачи управления приходится решать, чтобы процессор работал быстро:**

- Параллельное исполнение команд
  - **SS**: Независимые команды ищет процессор
  - **VLIW**: Независимые команды ищет компилятор
- Спекулятивное исполнение команд
  - **SS**: Процессор динамически предсказывает переход
  - **VLIW**: Компилятор подсказывает процессору как поступить
- Спекулятивная загрузка данных
  - **SS**: Процессор динамически проверяет корректность
  - **VLIW**: Компилятор использует специальную команду проверки
- Размещение данных на регистрах
  - **SS**: Процессор автоматически отображает программные регистры на аппаратные и управляет стеком регистров
  - **VLIW**: Компилятор размещает данные на аппаратных регистрах и управляет стеком регистров с помощью специальных команд

# Сравнение суперскалярных и VLIW/EPIIC-процессоров

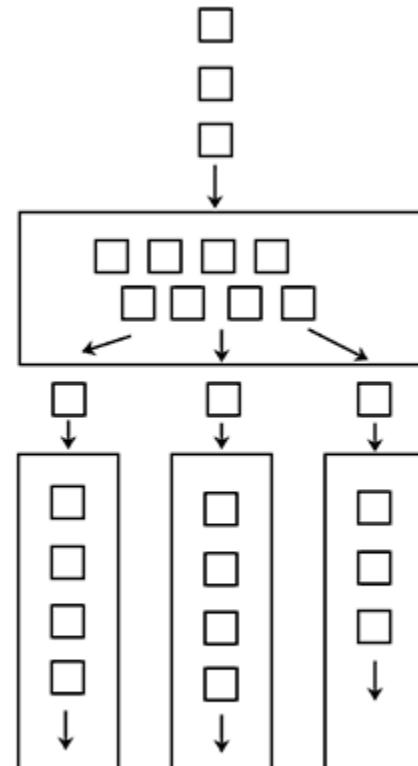
Какие  
про

- Пара

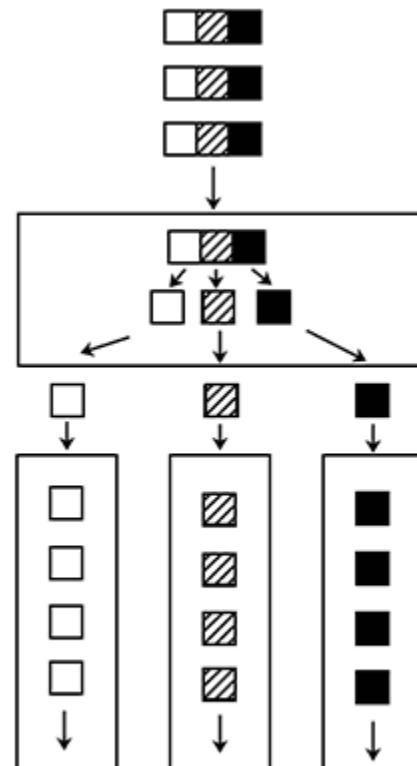
- Спе

- Спе

- Разм



Суперскалярный процессор



Архитектура VLIW

управляет стеком регистров с помощью специальных команд

ы

верки

регистры

истрах и

# Сравнение суперскалярных и VLIW/EPIC-процессоров

**Какие задачи управления приходится решать, чтобы процессор работал быстро:**

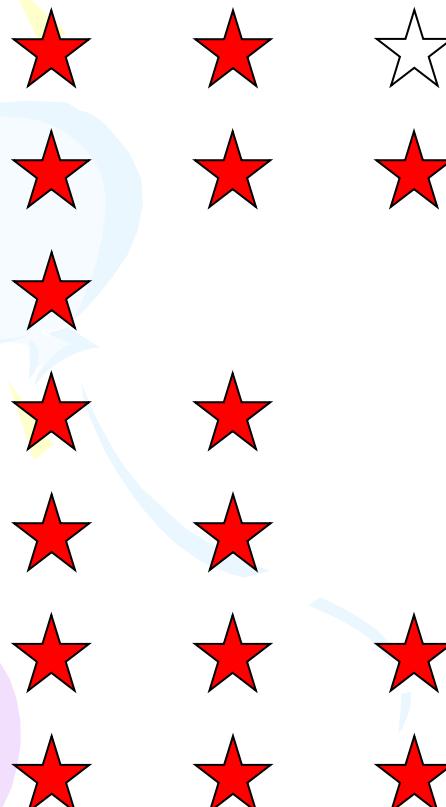
- Параллельное исполнение команд
  - SS: Независимые команды ищет процессор
  - EPIC: Независимые команды ищет компилятор
- Спекулятивное исполнение команд
  - SS: Процессор автоматически предсказывает переход
  - EPIC: Компилятор подсказывает процессору
- Спекулятивная загрузка данных
  - SS: Процессор автоматически проверяет корректность
  - EPIC: Компилятор использует специальную команду проверки
- Размещение данных на регистрах
  - SS: Процессор автоматически отображает программные регистры на аппаратные и управляет стеком регистров
  - EPIC: Компилятор размещает данные на аппаратных регистрах и управляет стеком регистров с помощью специальных команд

# Сравнение суперскалярных и VLIW/EPIIC-процессоров

Суперскалярные	VLIW/EPIIC
<p><b>Поток команд планируется процессором динамически</b></p> <p><b>Меньше</b> команд за такт:</p> <ul style="list-style-type: none"><li>• 3, 4, 5 (в среднем &lt; 50%)</li></ul>	<p><b>Поток команд планируется компилятором статически</b></p> <p><b>Больше</b> команд за такт:</p> <ul style="list-style-type: none"><li>• 6, 8, ... (в среднем &gt; 50%)</li></ul>
<p><b>Сложный исполнительный конвейер</b></p> <p><b>Меньше</b> места на кристалле для ресурсов процессора (регистры, кэш-память, исполнительные устройства)</p>	<p><b>Простой исполнительный конвейер</b></p> <p><b>Больше</b> места на кристалле для ресурсов процессора (регистры, кэш-память, исполнительные устройства)</p>
<p><b>«Простой» компилятор</b></p>	<p><b>«Сложный» компилятор</b></p>

# Сравнение конвейеров

CISC    RISC    VLIW



Этапы обработки команды

Предсказание ветвлений

Выборка

Декодирование в RISC

Переименование регистров

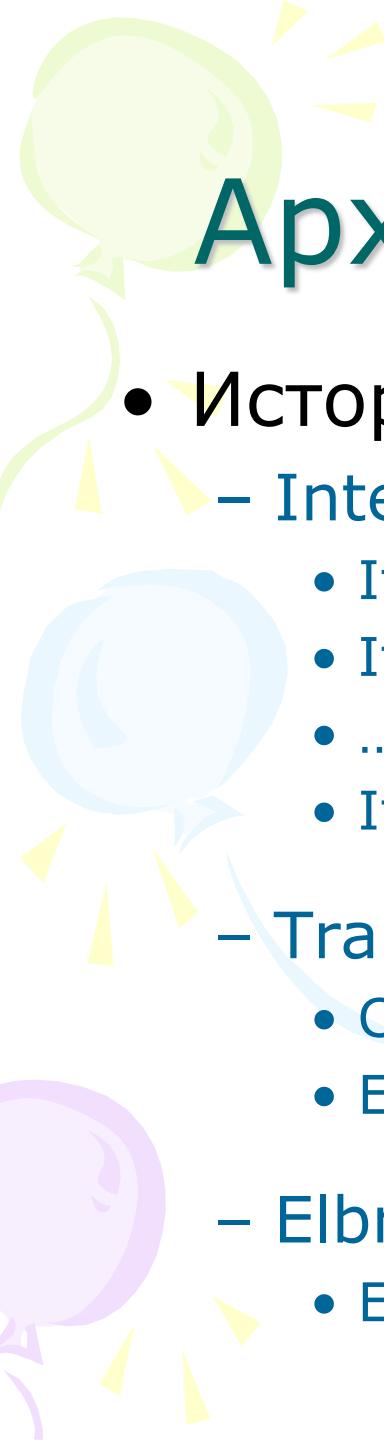
Переупорядочение и распараллеливание

Исполнение

Завершение

# Архитектура VLIW / EPIC

- История
  - М-10 (1972)
  - Cydrome (1984-1988)
    - Cydra-5
      - 256 bit VLIW (7 ops.), reg. rotation., sw. pipeline
  - МВК Эльбрус 3 (1986-1994)
  - NXP Semiconductors
    - TriMedia (1987, 1997, ...)
      - VLIW / DSP, 5-8 ops., 256x128 bit regs, 45 FUs
  - Texas Instruments
    - C6000
      - VLIW / DSP



# Архитектура VLIW / EPIC

- История
  - Intel Itanium
    - Itanium (2001)
    - Itanium2 (2002)
    - ...
    - Itanium 95xx (2012)
  - Transmeta
    - Crusoe (2000)
    - Efficion (2003)
  - Elbrus 2000
    - Elbrus 3M (2005)



# Архитектура Itanium



HP/Intel IA64 architecture  
alliance announced

Itanium 2  
Products

# Семейство процессоров Itanium

2001



Itanium  
(Merced)  
800 MHz  
4 MB L3 cache  
180 nm

2002



Itanium2  
(McKinley)  
1 GHz  
3 MB L3 cache  
180 nm

2003



Itanium2  
(Madison)  
1.5 GHz  
6 MB L3 cache  
130 nm

2006



Itanium2  
(Montecito)  
1.66 GHz  
2×12 MB L3 cache  
2 cores  
HyperThreading  
90 nm

2010



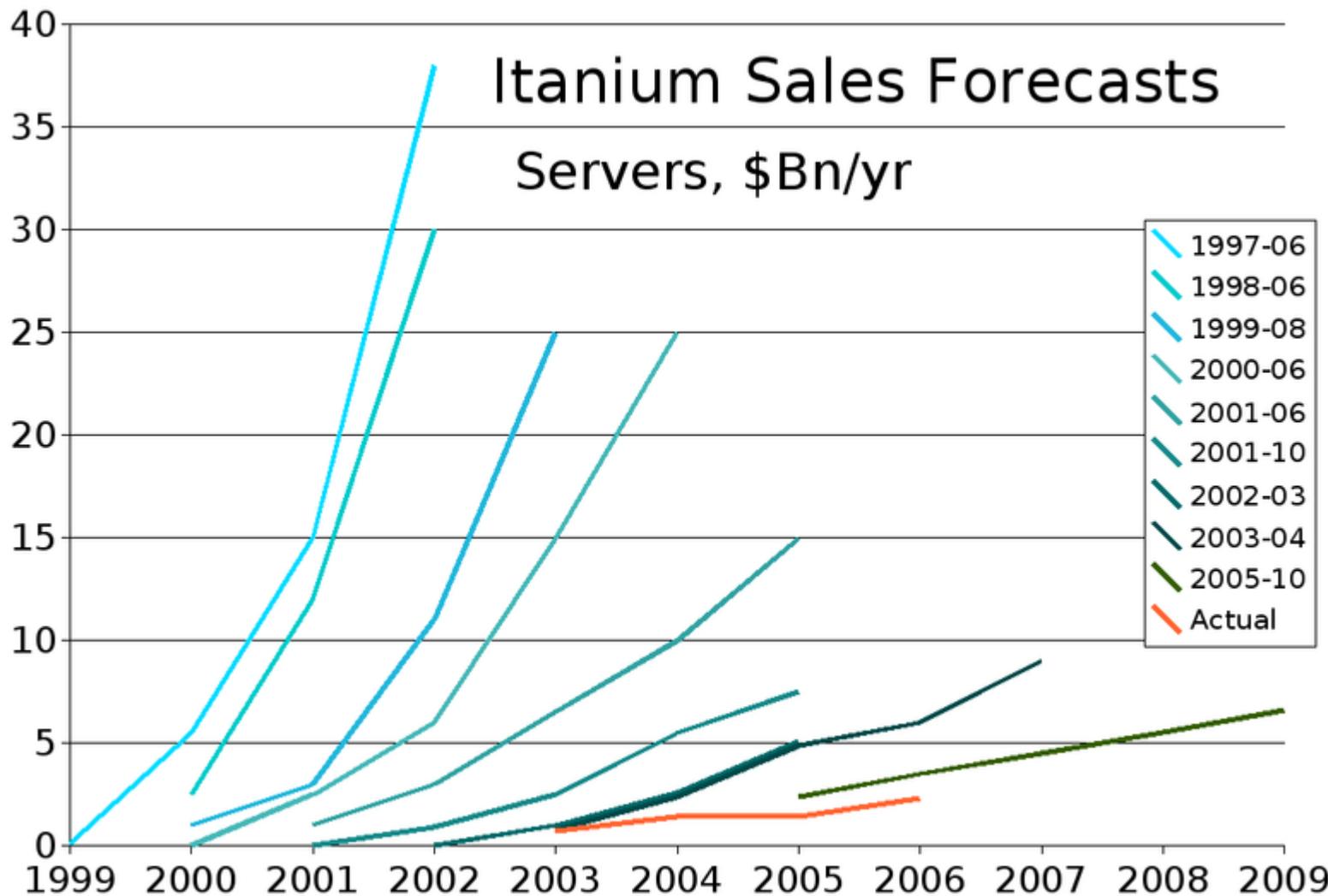
Itanium 9300  
(Tukwila)  
1.73 GHz  
24 MB L3 cache  
4 cores  
HyperThreading  
65 nm

2012



Itanium 9500  
(Poulson)  
2.533 GHz  
32 MB L3 cache  
8 cores  
HyperThreading  
32 nm

# Itanium: планы и реальность



# Архитектура Itanium (IA-64)

- **Архитектура VLIW/EPIC**

- Компилятор полностью управляет ресурсами процессора и планирует поток команд, формирует группы независимых команд.
- Процессор обеспечивает большое число ресурсов для реализации ILP

- **Учет динамики исполнения программы**

- Предварительная загрузка данных (уменьшает задержки по памяти),
- Предикатное исполнение команд (устраняет ветвления),
- Динамическое предсказание ветвлений.

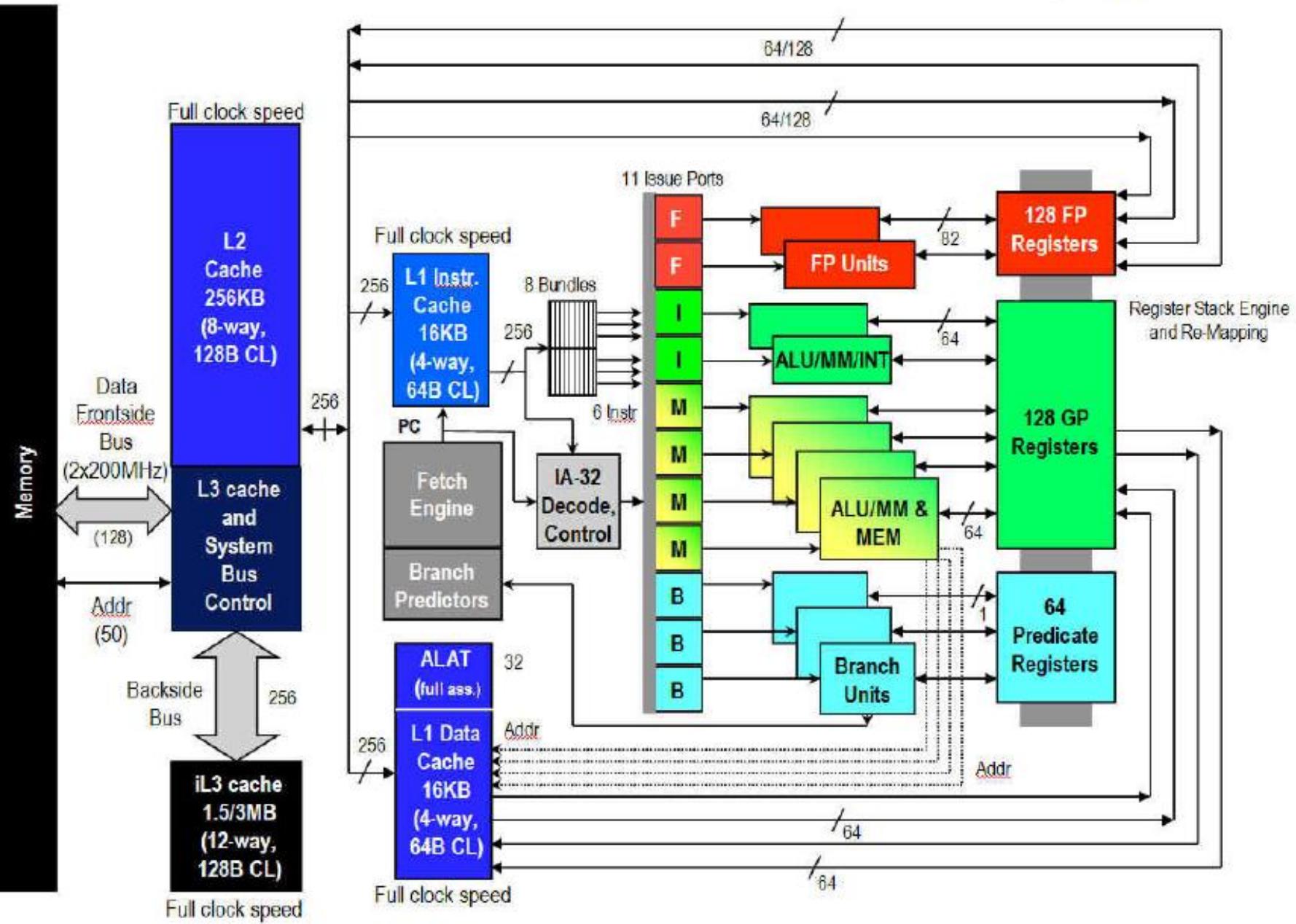
- **Специальные способы увеличения производительности программ**

- Аппаратная поддержка программной конвейеризации циклов (вращающиеся регистры, предикатные регистры, специальные счетчики, специальные команды),
- Специальная поддержка модульности программ (регистровый стек),
- Высокопроизводительная вещественная арифметика,
- Специальные векторные инструкции.

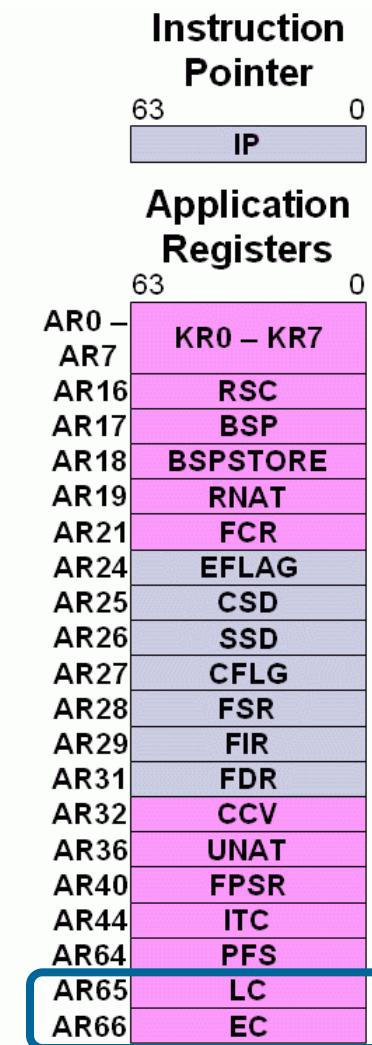
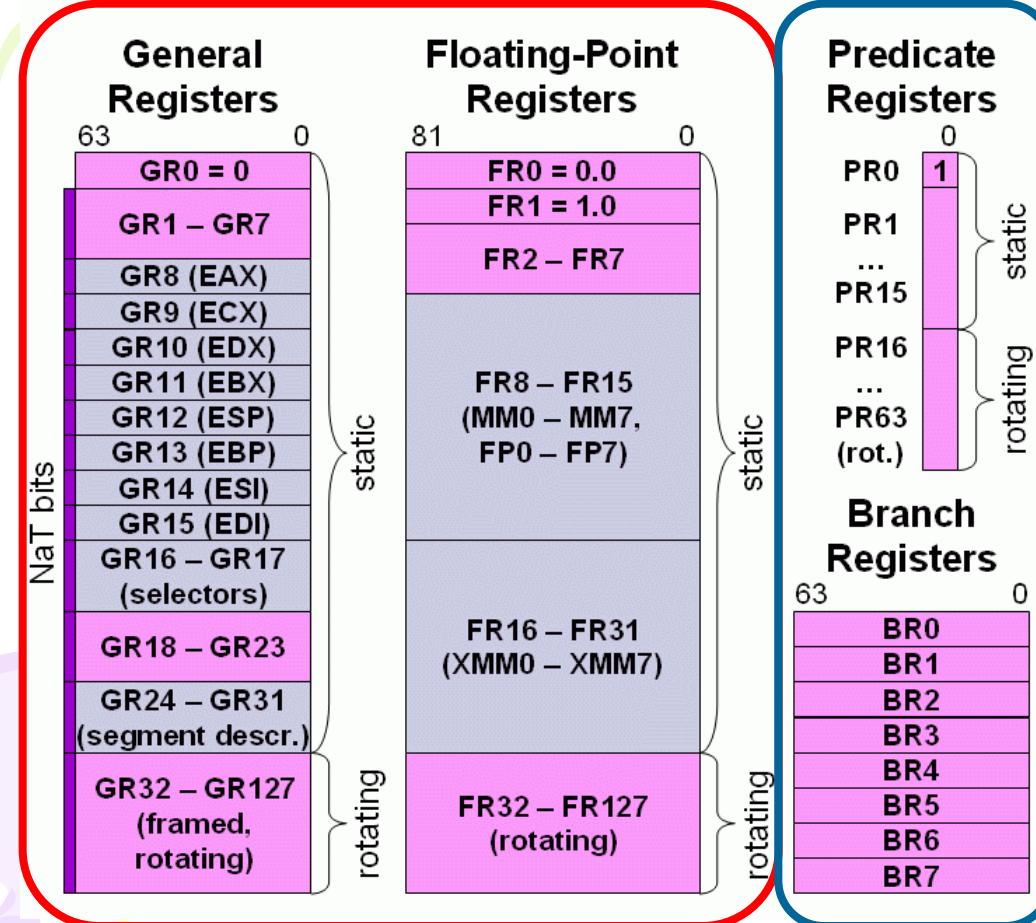
# Особенности процессоров Itanium

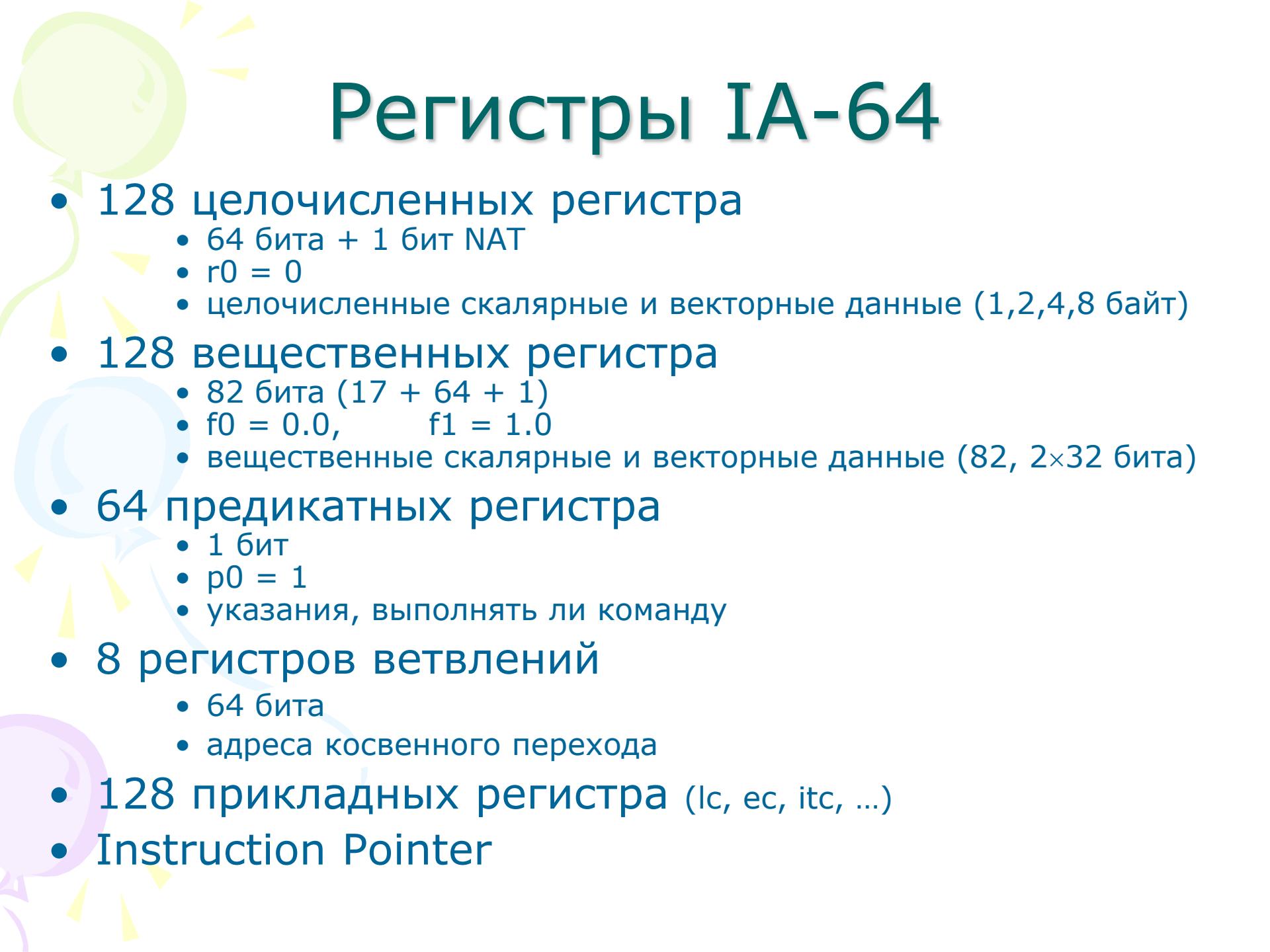
- Простой широкий конвейер
  - Много команд за такт (до 6)
- Большие вычислительные ресурсы
  - Много исполнительных устройств (11)
  - Большой объем (до 12 МВ) кэш-памяти
  - Большое число регистров (264)

# Itanium® 2 Processor Block Diagram



# Регистры IA-64



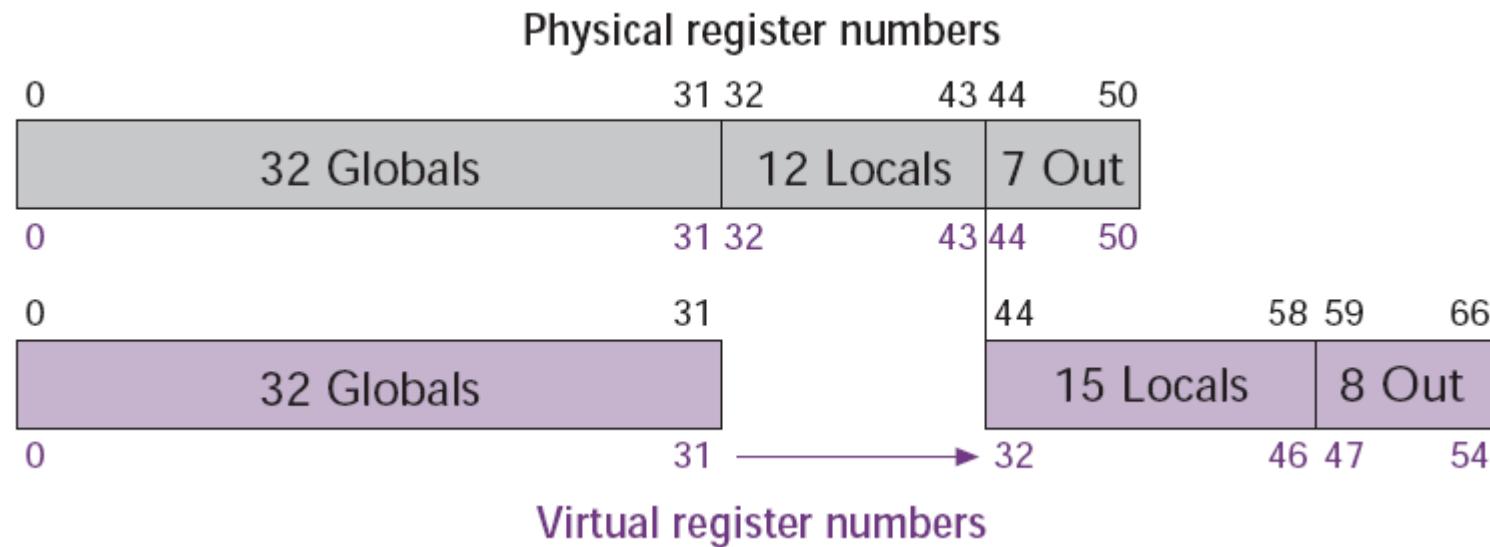


# Регистры IA-64

- 128 целочисленных регистра
  - 64 бита + 1 бит NAT
  - $r_0 = 0$
  - целочисленные скалярные и векторные данные (1, 2, 4, 8 байт)
- 128 вещественных регистра
  - 82 бита ( $17 + 64 + 1$ )
  - $f_0 = 0.0, f_1 = 1.0$
  - вещественные скалярные и векторные данные ( $82, 2 \times 32$  бита)
- 64 предикатных регистра
  - 1 бит
  - $p_0 = 1$
  - указания, выполнять ли команду
- 8 регистров ветвлений
  - 64 бита
  - адреса косвенного перехода
- 128 прикладных регистра ( $lc, ec, itc, \dots$ )
- Instruction Pointer

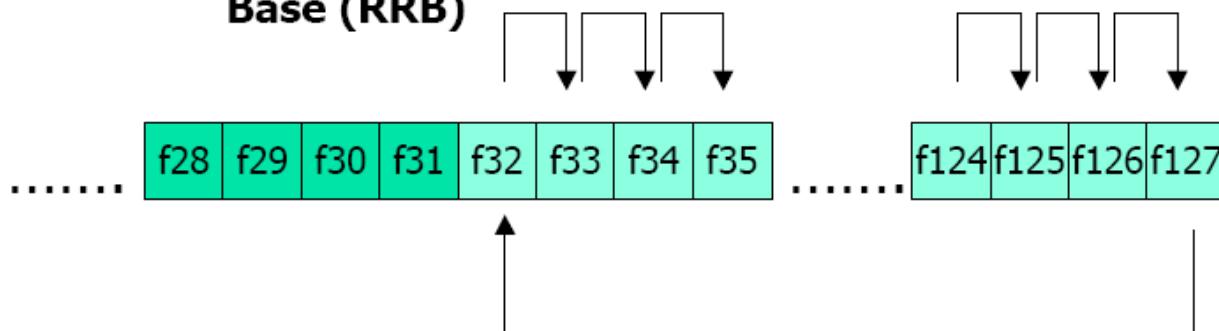
# Стек регистров

- При вызове подпрограмм и возврате происходит сдвиг регистрового окна – целочисленные регистры работают как стек.

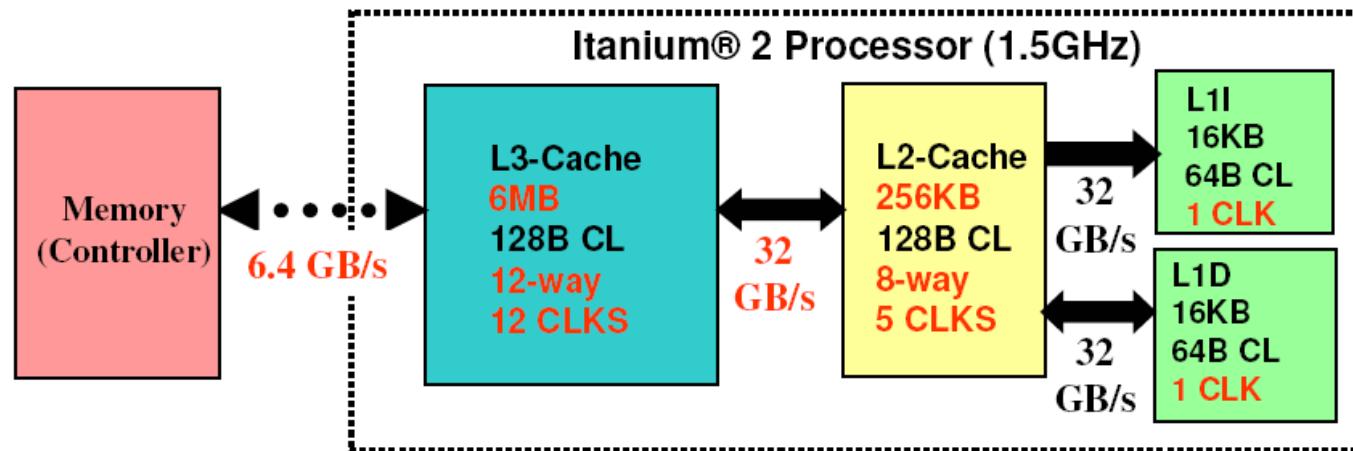


- Для автоматического сохранения/восстановления регистров в памяти при «переполнении/переизбытке» стека работает аппаратура RSE (Register Stack Engine). Она приостанавливает выполнение команд, ждущих соответствующие регистры.

# Вращение регистров

- Верхние 75% регистров вращающиеся:
  - целочисленные: r32 – r127 (локальные)
  - вещественные: f32 – f127 (локальные)
  - предикатные: p16 – p63
- При выполнении специальной команды перехода (br.ctop / br.cexit / br.wtop / br.wexit) вращающиеся регистры сдвигаются вправо на один:
  - **Virtual Register = Physical Register – Register Rotation Base (RRB)**
- Используется при программной конвейеризации циклов.

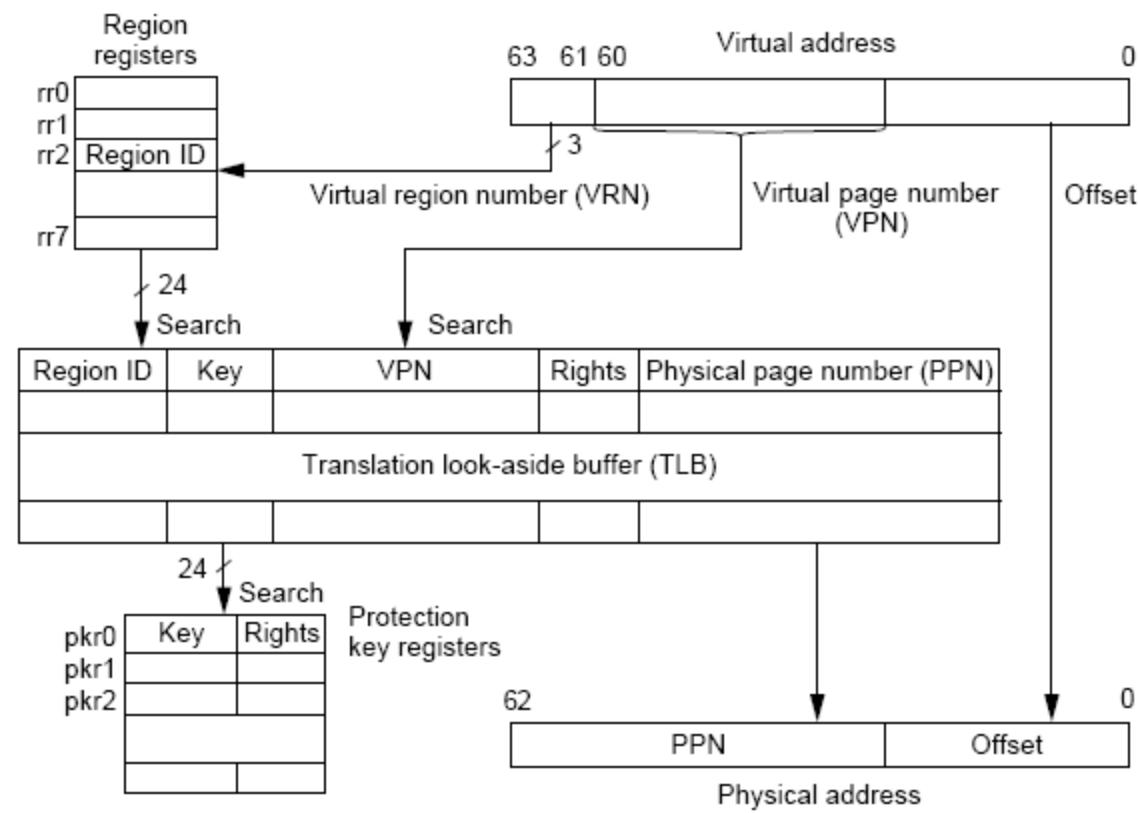
# Иерархия кэш-памяти Itanium2



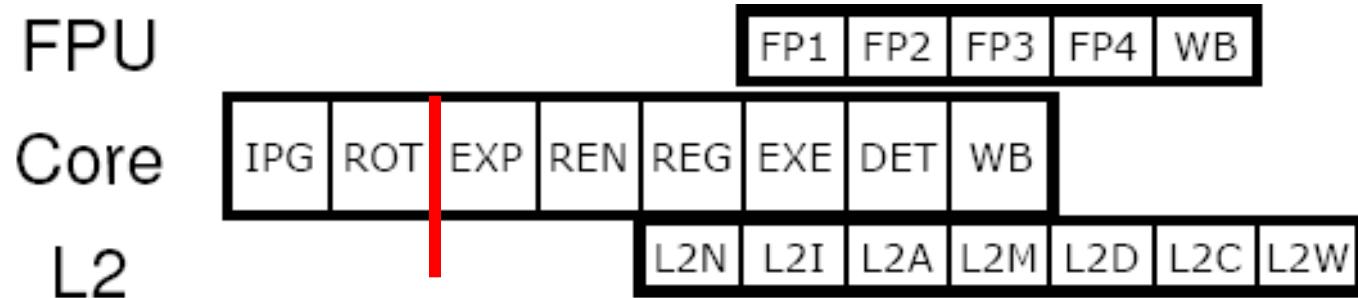
	L1I	L1D	L2	L3
Size	16K	16K	<b>256K</b>	<b>12M on die</b>
Line Size	64B	64B	128B	128B
Ways	4	4	<b>8</b>	<b>12</b>
Replacement	LRU	NRU	NRU	NRU
Latency (load to use)	<b>I-Fetch:1</b>	<b>INT:1</b>	<b>INT: 5 FP: 6</b>	<b>INT: 12 FP: 13</b>
Write Policy	-	WT (RA)	WB (WA + RA)	WB (WA)
Bandwidth	<b>R: 32 GBs</b>	<b>R: 16 GBs W: 16 GBs</b>	<b>R: 32 GBs W: 32 GBs</b>	<b>R: 32 GBs W: 32 GBs</b>

# Виртуальная память в IA-64

- 64-битное виртуальное адресное пространство
- Размер страницы: 4 KB – 4 GB
- 32 entry L1d TLB (4KB), 128 entry Data TLB (4KB-4GB)
- Схема преобразования виртуального адреса в физический:



# Стадии конвейера Itanium2



<b>IPG</b>	Вычисление IP, чтение кэша L1I <b>(6 инст.)</b> и TLB.	<b>EXE</b>	Выполнение <b>(6)</b> , обращение к кэшу L1D и TLB + обращение к тэгам L2 кэша <b>(4)</b>
<b>ROT</b>	Расцепление и буферизация инструкций.	<b>DET</b>	Обнаружение исключений, выполнение переходов
<b>EXP</b>	Разворачивание инструкции, назначение порта	<b>WB</b>	Завершение, запись регистрового файла
<b>REN</b>	Переименование регистров <b>(6 инстр.)</b>	<b>FP1-WB</b>	Конвейер FP FMAC + запись результата в регистр
<b>REG</b>	Чтение регистрационных файлов <b>(6)</b>	<b>L2N-L2I</b>	L2 Queue Nominate / Issue <b>(4)</b>
		<b>L2A-W</b>	L2 Access, Rotate, Correct, Write <b>(4)</b>

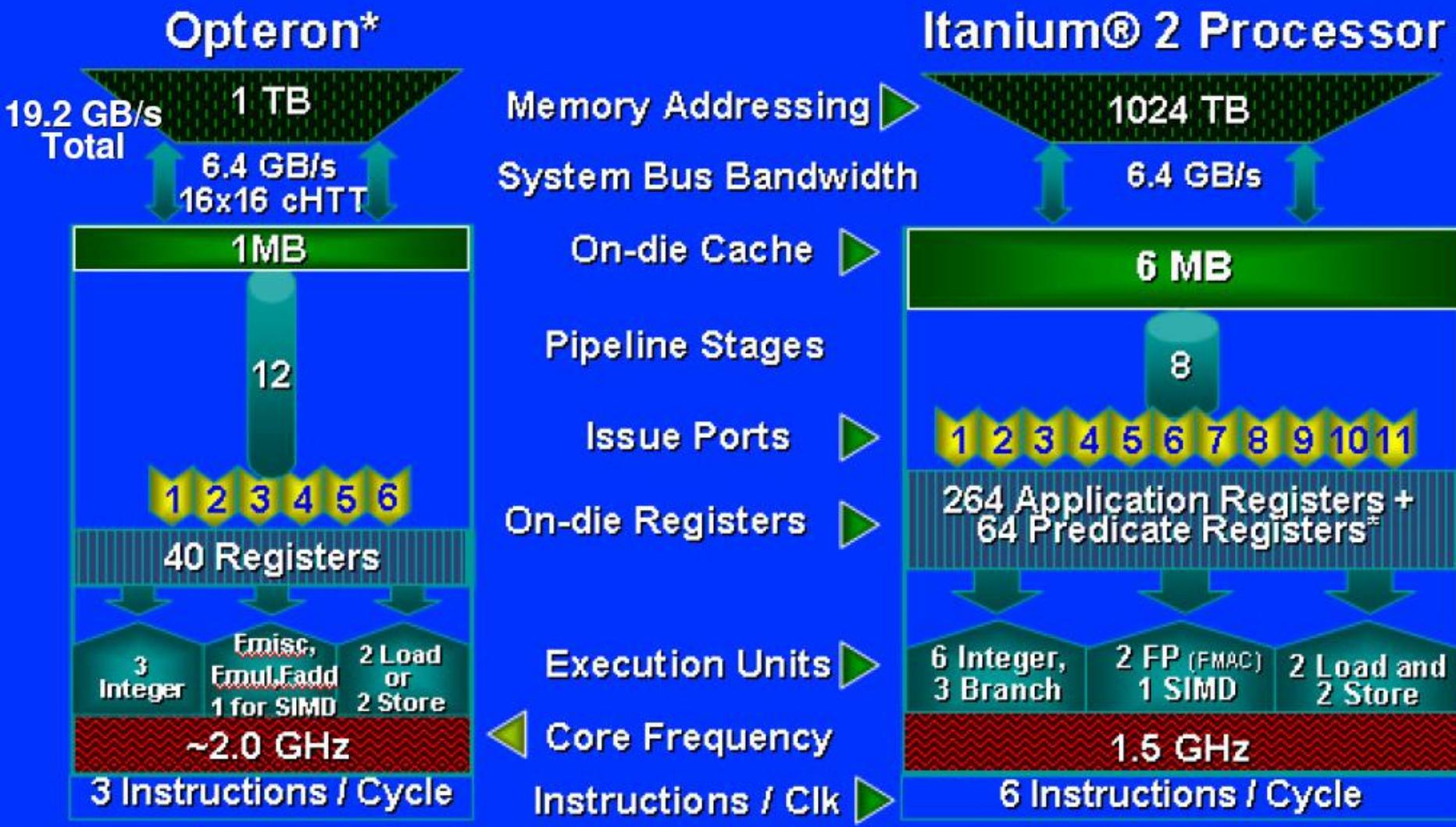
## Короткий 8-стадийный конвейер

- Полностью детерминированный путь команд
- Упорядоченная выборка команд, неупорядоченное завершение
- Рассчитан на малые задержки при чтении данных!

# Исполнительные устройства

		Число операций за такт		Issue Ports/Units
		Itanium®	Itanium® 2	
F.P.				
Integer				
Multimedia				
Load/Store				
Branch				

# Сравнение Itanium2 и Opteron



x86 with extra memory bits

Itanium Architecture

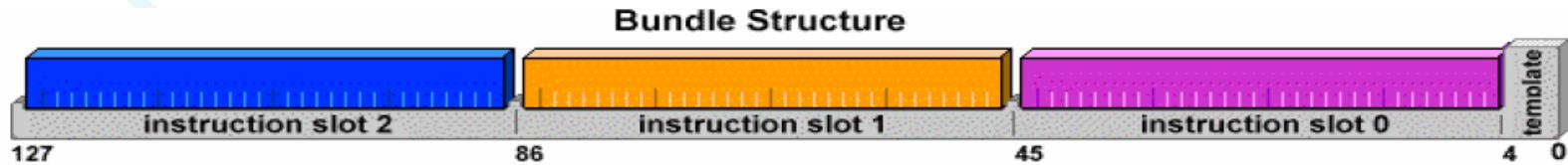
\* Intel's EPIC technology includes 64 single-bit predicate registers to accelerate loop unrolling and branch intensive code execution.

# Команды IA-64

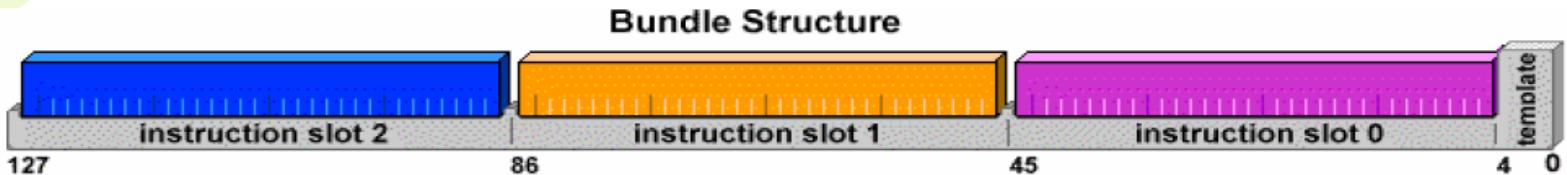
- Команды IA-64 имеют RISC-подобный фиксированный формат:



- Пример команды: (p3) add r1 = r3, r4
- Команды IA-64 объединяются в связки по три:



# Команды IA-64

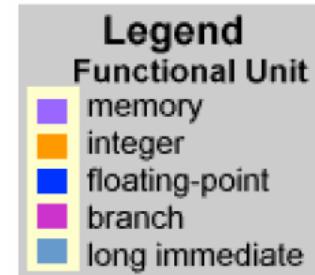


- Связка содержит 3 команды, поле шаблона и стоп-биты.
- Шаблон указывает типы команд в связке. Он определяет, какие исполнительные устройства будут задействованы при исполнении.
- Типы команд:
  - M – memory / move M
  - I – complex integer / multimedia I
  - A – simple integer / logic / multimedia I или M
  - F – floating point (normal / SIMD) F
  - B – branch B
  - L+X – extended I / B
- Стоп-биты определяют, после каких команд должен быть переход на следующий такт.

# Команды IA-64

- Всего возможно  
24 различных  
шаблона:

MII	MMF	MII <sub>s</sub>	MMF <sub>s</sub>
MISI	MIB	MISI <sub>s</sub>	MIB <sub>s</sub>
MLX*	MBB	MLX <sub>s</sub> *	MBB <sub>s</sub>
MMI	BBB	MMI <sub>s</sub>	BBB <sub>s</sub>
MSMI	MMB	MSMI <sub>s</sub>	MMB <sub>s</sub>
MFI	MFB	MFI <sub>s</sub>	MFB <sub>s</sub>



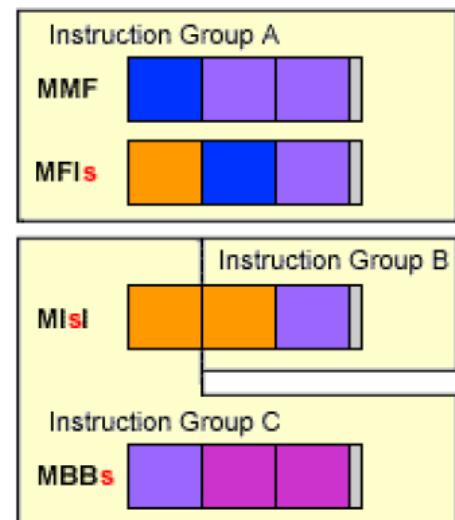
\* L+X is an extended type that is dispatched to the I-unit.

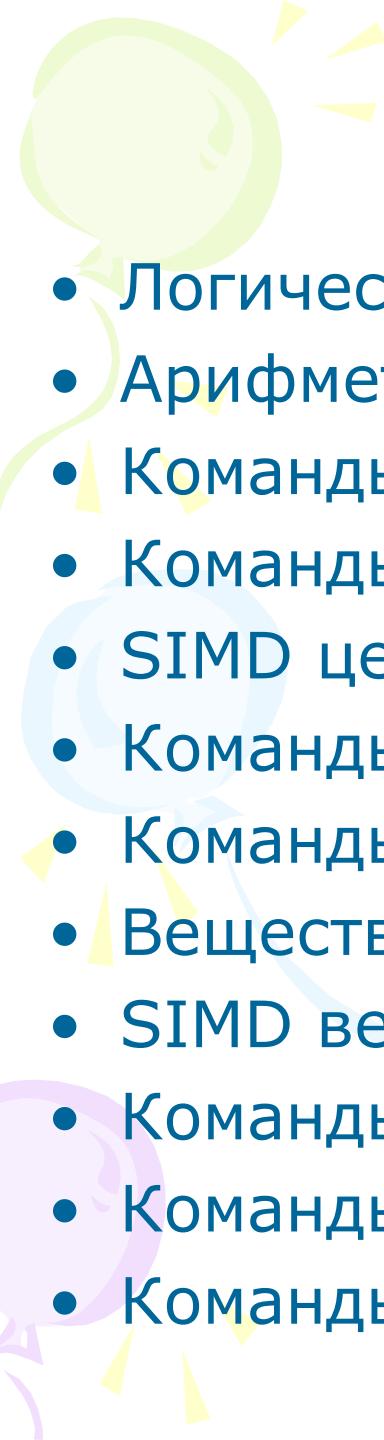
- Процессор загружает максимум по 2 связки за такт.
- Только некоторые сочетания шаблонов в связках могут полностью загрузить исполнительные устройства:

	MII	MLI	MMI	MFI	MMF	MIB	MBB	BBB	MBB	MFM
MII	blue			blue	blue	blue	red	red	blue	red
MLI	blue	blue		red		red	blue	red		red
MMI	blue	blue	blue		blue	blue	blue	red	blue	blue
MFI	blue	red	blue	red	blue	red	red	blue	blue	red
MMF					blue	blue	blue	blue	blue	blue
MIB*	red	red	blue	red	blue	red	red		blue	red
MBB										
BBB										
MMB*	blue	blue	blue	blue	blue	blue		blue	blue	blue
MFB*	red	red	blue	red	blue	red		blue	blue	red

\* hint in first bundle

■ Possible Itanium 2 full issue  
■ Possible Itanium processor and Itanium 2 full issue





# Команды IA-64

- Логические (and, ...)
- Арифметические (add, ...)
- Команды сравнения (cmp, ...)
- Команды сдвига (shl, ...)
- SIMD целочисленные (rptru, ...)
- Команды ветвлений (br, ...)
- Команды управления циклом (br.coop, ...)
- Вещественные (fma, ...)
- SIMD вещественные (frma, ...)
- Команды чтения / записи данных в памяти (ld, ...)
- Команды присваивания (mov, ...)
- Команды управления кэшированием (lfetch, ...)

# Особенности целочисленной арифметики в Itanium2

- Высокая производительность: до 6 операций за такт
- Операция **fma** ( $y=a*b+c$ ) выполняется на регистрах FR
- Реализованы некоторые операции над некоторыми векторами (1В, 2В, 4В)
- Целочисленное деление реализуется программно
  - Пример деления 32-битных целых чисел:

$$(1) \quad y_0 = 1 / b \cdot (1 + \epsilon_0), \quad |\epsilon_0| < 2^{-8.886}$$

$$(2) \quad q_0 = (a \cdot y_0)_m$$

$$(3) \quad e_0 = (1 - b \cdot y_0)_m$$

$$(4) \quad q_1 = (q_0 + e_0 \cdot q_0)_m$$

$$(5) \quad e_1 = (e_0 \cdot e_0 + 2^{-34})_m$$

$$(6) \quad q_2 = (q_1 + e_1 \cdot q_1)_m$$

$$(7) \quad q = \text{trunc}(q_2)$$

table lookup

82-bit register format precision

floating point to signed integer conversion (RZ mode)

$$q = \left\lfloor \frac{a}{b} \right\rfloor$$

# Особенности вещественной арифметики в Itanium2

- Высокая производительность
  - 2 за такт: двойная точность
  - 4 за такт: одинарная точность (SIMD)
- Основная операция
  - fma:  $f = a * b + c$  (4 такта)
- Быстрое преобразование значений между целыми и вещественными регистрами
  - FP  $\rightarrow$  INT (getf): 5 тактов
  - INT  $\rightarrow$  FP (setf): 6 тактов
- Операции деления (вещественного и целочисленного) и взятия квадратного корня реализованы программно

# Особенности вещественной арифметики в Itanium2

- Вещественное деление (32-bit float)

$$(1) \quad y_0 = 1 / b \cdot (1 + \varepsilon_0), \quad |\varepsilon_0| < 2^{-8.886}$$

$$(2) \quad d = (1 - b \cdot y_0)_{rn}$$

$$(3) \quad e = (d + d \cdot d)_{rn}$$

$$(4) \quad y_1 = (y_0 + e \cdot y_0)_{rn}$$

$$(5) \quad q_1 = (a \cdot y_1)_{rn}$$

$$(6) \quad r = (a - b \cdot q_1)_{rn}$$

$$(7) \quad q = (q_1 + r \cdot y_1)_{rnd}$$

table lookup

82-bit register format precision

82-bit register format precision

82-bit register format precision

17-bit exponent, 24-bit mantissa

82-bit register format precision

single precision

- Вычисление корня (32-bit float)

$$(1) \quad y_0 = (1 / \sqrt{a}) \cdot (1 + \varepsilon_0), \quad |\varepsilon_0| < 2^{-8.831}$$

$$(2) \quad H_0 = (0.5 \cdot y_0)_{rn}$$

$$(3) \quad S_0 = (a \cdot y_0)_{rn}$$

$$(4) \quad d = (0.5 - S_0 \cdot H_0)_{rn}$$

$$(5) \quad d' = (d + 0.5 * d)_{rn}$$

$$(6) \quad e = (d + d * d')_{rn}$$

$$(7) \quad S_1 = (S_0 + e * S_0)_{rn}$$

$$(8) \quad H_1 = (H_0 + e * H_0)_{rn}$$

$$(9) \quad d_1 = (a - S_1 \cdot S_1)_{rn}$$

$$(10) \quad S = (S_1 + d_1 \cdot H_1)_{rnd}$$

table lookup

82-bit register format precision

82-bit register format precision

82-bit register format precision

82-bit register format precision

17-bit exponent, 24-bit mantissa

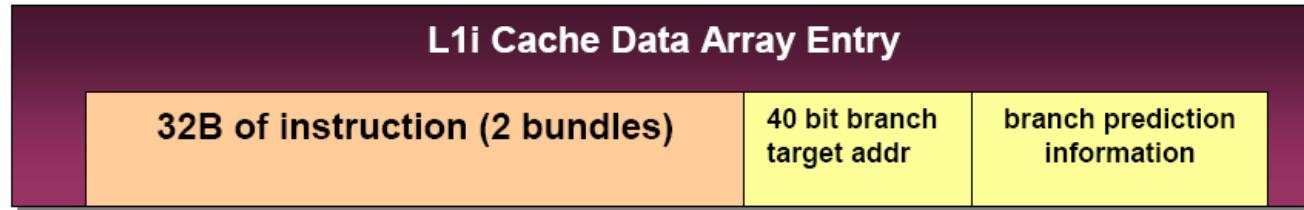
82-bit register format precision

82-bit register format precision

single precision

# Предсказание ветвлений в Itanium2

- ВНТ – таблица истории ветвлений
  - Адрес перехода и информация о предсказании в кэше L1i



- Таблица на 12К 4-битных историй
- Pattern History Table
  - Таблица на 16К 2-битных счетчиков
- RSB – Буфер стека возврата
  - 8 элементов
- Предсказание косвенных переходов
  - Использует 8 регистров ветвлений, подсказки компилятора
- Механизм предсказания выхода из циклов
  - Использует специальные счетчики

# Предвыборка инструкций в Itanium2

- Автоматическая предвыборка следующей кэш-строки в кэш команд L1, если она содержится в кэше L2.
- Подсказка компилятора в команде перехода:
  - br.few <address>
  - br.many <address>
- Подсказка компилятора:
  - brp.few <address>
  - brp.many <address>
  - Move address to Branch Register

# Фрагмент кода на ассемблере для IA-64

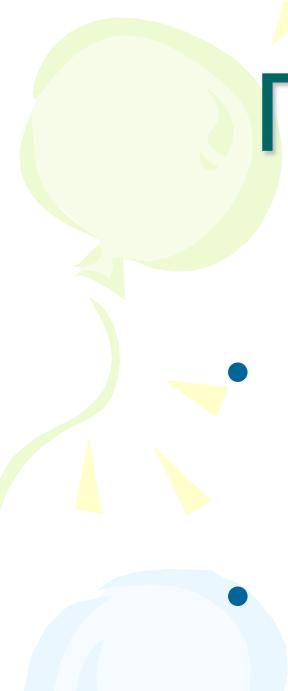
Синтаксис инструкций:  
**(qr) ops[.comp<sub>1</sub>]**       $r_1 = r_2, r_3$

```
.b7_12:
{
    .mmf
    (p16)ld4      r38=[r29],8
    (p17)shladd   r47=r27,3,r45
    (p21)fma.d    f36=f38,f1,f42
}
{
    .mmi
    (p17)ldfd     f32=[r39]
    (p16)ld4.s    r64=[r24],8
    nop.i         0 ;;
}
{
    .mii
    (p16)ld4      r46=[r19],8
    (p16)sxt4     r63=r38
    (p16)shladd   r39=r64,3,r45
}
{
    .mmb
    (p17)ldfd     f38=[r47]
    (p17)lfetch.nt1 [r32]
    nop.b         0 ;;
}
{
    .mfi
    (p16)shladd   r38=r63,3,r45
    (p20)fma.d    f42=f44,f1,f35
    (p16)sxt4     r27=r46
}
{
    .mib
    (p16)lfetch.nt1 [r17],8
    (p16)add       r32=1,r33
    br.ctop.sptk   .b7_12 ;;
}
```



# Средства повышения производительности в IA-64

- Предикатное исполнение команд
- Аппаратные счетчики циклов
- Спекуляция по данным и управлению
- Регистровый стек, RSE
- Аппаратная поддержка программной конвейеризации циклов



# Предикатное исполнение команд

- Позволяет зависимости по управлению (т.е. условные переходы) преобразовать в зависимости по данным.
- Пример: **if (a==b) y=4; else y=3;**

## Unpredicated Code

cmp a, b

jmp EQ

y=3

jmp END

EQ: y=4

END:

## Predicated Code

cmp.eq p1, p2=a, b

p1 y=4

p2 y=3

# Аппаратные счетчики циклов

- Архитектурная поддержка циклов
  - Специальные регистры-счетчики: ar.lc, ar.ec
  - По специальной команде перехода:
    - счетчики автоматически уменьшаются,
    - выполняется проверка на выход из цикла
  - Можно не задействовать регистры общего назначения.
- Пример:

```
    mov ar.lc = 10 ;;
```

Label:

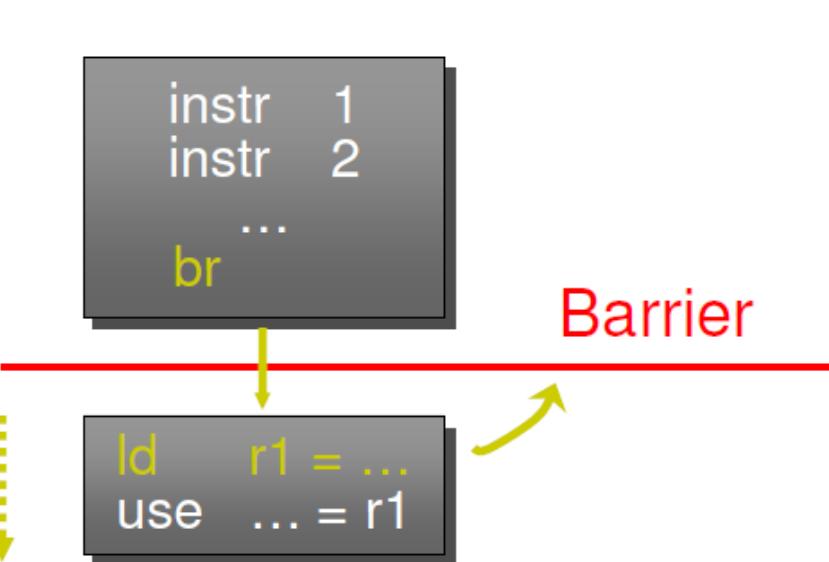
... тело цикла ...

```
    br.cloop.sptk Label
```

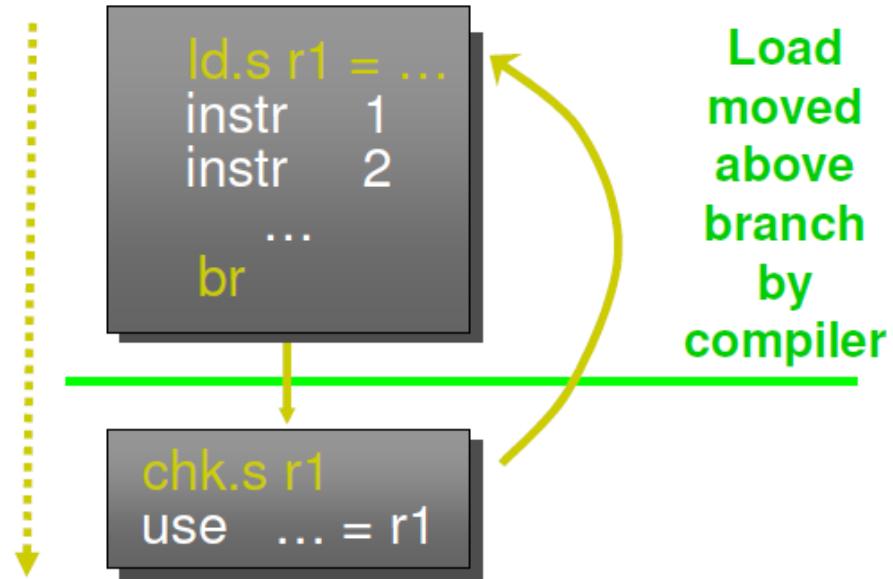
# Спекуляция по управлению

- Команды загрузки могут выполняться заранее – до того, как обнаружится, что это действительно нужно
  - До перехода на нужную ветку программы

Traditional Architectures



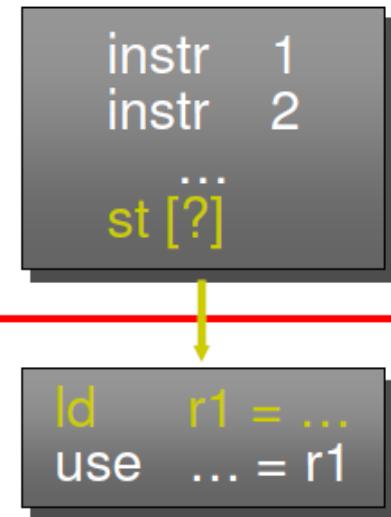
IA-64



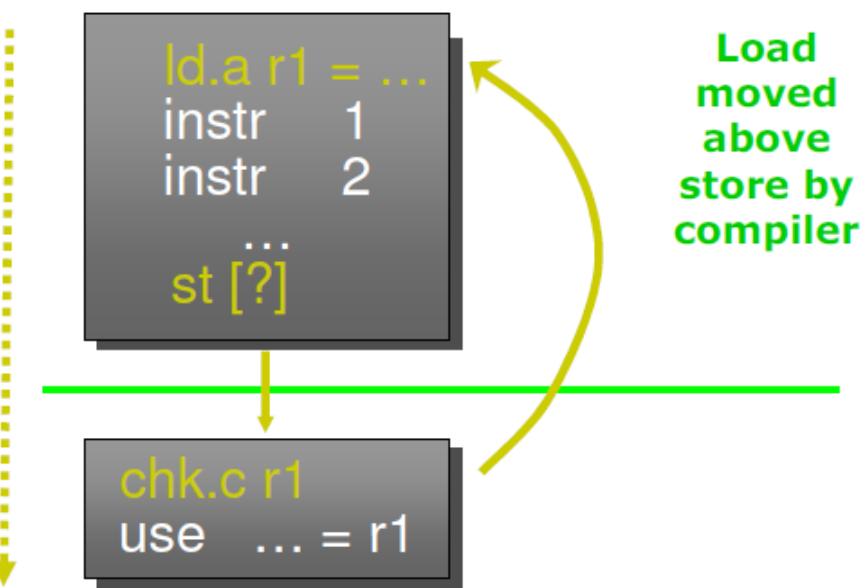
# Спекуляция по данным

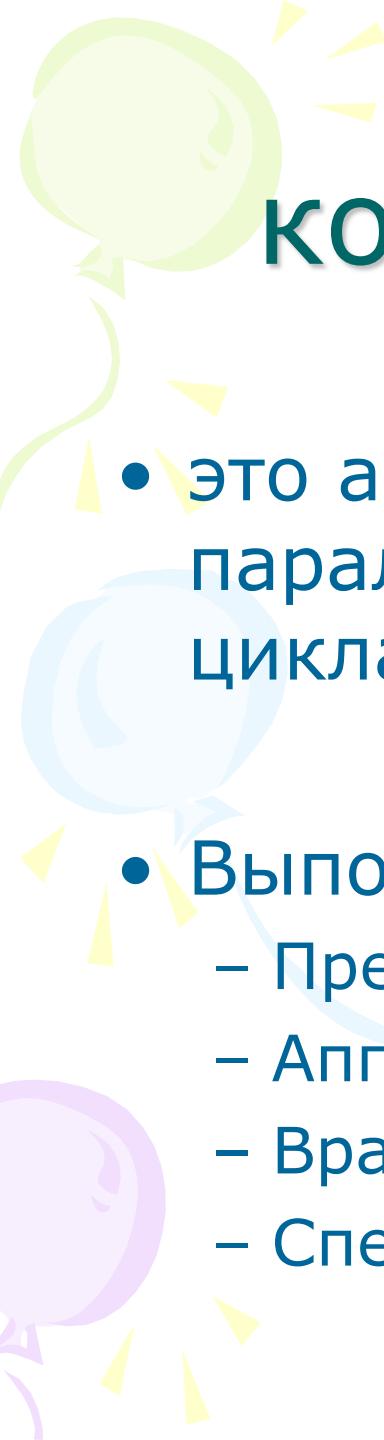
- Команды загрузки могут выполняться заранее – до того, как обнаружится, что это действительно можно.
  - До проверки, что данные не пересекаются

Traditional Architectures



IA-64





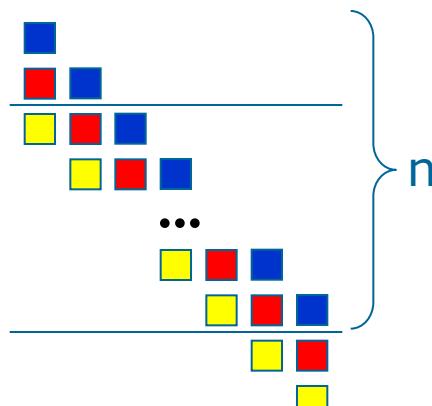
# Программная конвейеризация цикла

- это архитектурная поддержка параллельного исполнения команд цикла.
- Выполняется с помощью:
  - Предикатных регистров
  - Аппаратных счетчиков цикла
  - Вращающихся регистров
  - Специальных команд перехода

# Software Pipelining Example

## C Code Example

```
int n=5, i;
for(i=0;i<n;i++)
    y[i]=x[i]+1
```



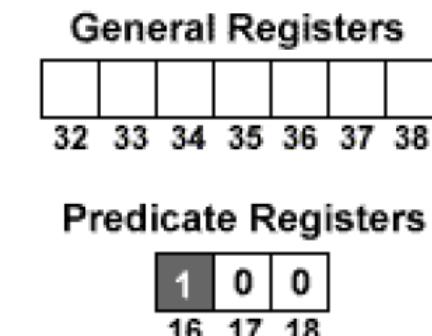
## Pseudo Assembly Code

```
// Initialization
    mov pr.rot      = 0
// Clear all rotating predicate registers
    cmp.eq p16,p0 = r0,r0 // set p16=1
    mov ar.lc       = 4 // set LC to n-1
    mov ar.ec       = 3
// Set epilog counter to 3
    ...
// loop
loop:
    (p16) ld1 r32 = [r12],1 // #1: load x
    (p17) add r34 = 1,r33 // #2: y=x+1
    (p18) st1 [r13] = r35,1 // #3: store y
// Branch back
    br.ctop.sptk.few loop
```

# Software Pipelining Example, ...

- This simulation assumes 5 iterations and one-cycle latencies.

```
loop:  
  (p16)  ld1 r32    = [r12],1  
  (p17)  add r34    = 1,r33  
  (p18)  st1 [r13] = r35,1  
        br.ctop loop
```

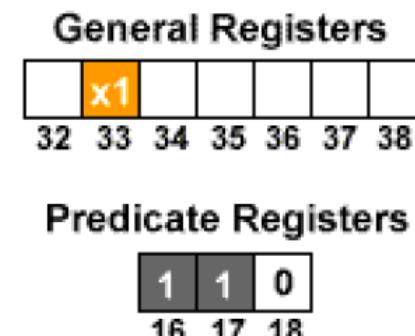


# Software Pipelining Example, ...

- This is the first iteration in the prolog stage. Only the load instruction executes.
  - The add and store instructions are executed as NOPs.
  - After the branch instruction, the data rotates from register GR 32 to GR 33.
  - $p17=p16=1$  and LC decrements to 3.

**loop:**

```
(p16)  ld1 r32    = [r12],1
(p17)  add r34    = 1,r33
(p18)  st1 [r13] = r35,1
       br.ctop loop
```



# Software Pipelining Example, ...

- This is the second and last iteration in the prolog stage. The add instruction also executes.
  - After the branch instruction the data rotates from registers GR 32-34 to GR 33-35.
  - $p18=p17=p16=1$  and LC decrements to 2.

**loop:**

```
(p16) ld1 r32    = [r12],1  
(p17) add r34    = 1,r33  
(p18) st1 [r13] = r35,1  
      br.ctop loop
```

General Registers

	x2	x1	y1				
32	33	34	35	36	37	38	

Predicate Registers

1	1	1
16	17	18

LC

2

EC

3

# Software Pipelining Example, ...

- This is the kernel stage. All three instructions execute.
  - After each branch instruction the data rotates and LC decrements.
  - At the end of this stage LC=0 and EC decrements to 2.

**loop:**

```
(p16)  ld1 r32    = [r12],1  
(p17)  add r34    = 1,r33  
(p18)  st1 [r13] = r35,1  
       br.ctop loop
```

General Registers

	x5	x4	y4	y3	y2	y1
32	33	34	35	36	37	38

Predicate Registers

0	1	1
16	17	18

LC

0

EC

2

# Software Pipelining Example, ...

- This is the first iteration of the epilogue stage.  
Only the add and store instructions execute.
  - At the end of this iteration  $p16=p17=0$   $p18=1$  and EC decrements to 1

**loop:**

```
(p16)    ld1 r32    = [r12],1  
(p17)    add r34    = 1,r33  
(p18)    st1 [r13] = r35,1  
          br.ctop loop
```

General Registers

		x5	y5	y4	y3	y2
32	33	34	35	36	37	38

Predicate Registers

0	0	1
16	17	18

LC

0

EC

1

# Software Pipelining Example, ...

- This is the second and last iteration of the epilogue stage. Only the store instruction executes.
  - At the end of this iteration EC=0, p16=p18=p17=0

**loop:**

(p16)	ldl r32	= [r12],1
(p17)	add r34	= 1,r33
<b>(p18)</b>	stl [r13]	= r35,1
	br.ctop loop	

General Registers

32	33	34	x5	y5	y4	y3	y2
35	36	37					

Predicate Registers

0	0	0
16	17	18

LC

0

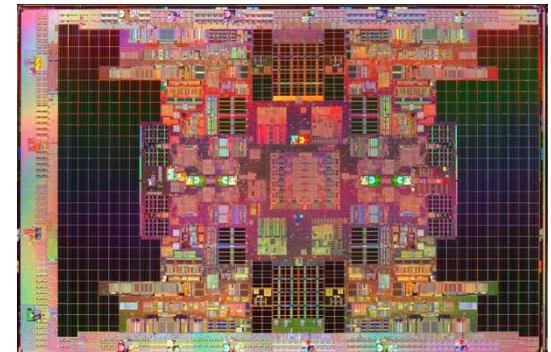
EC

0

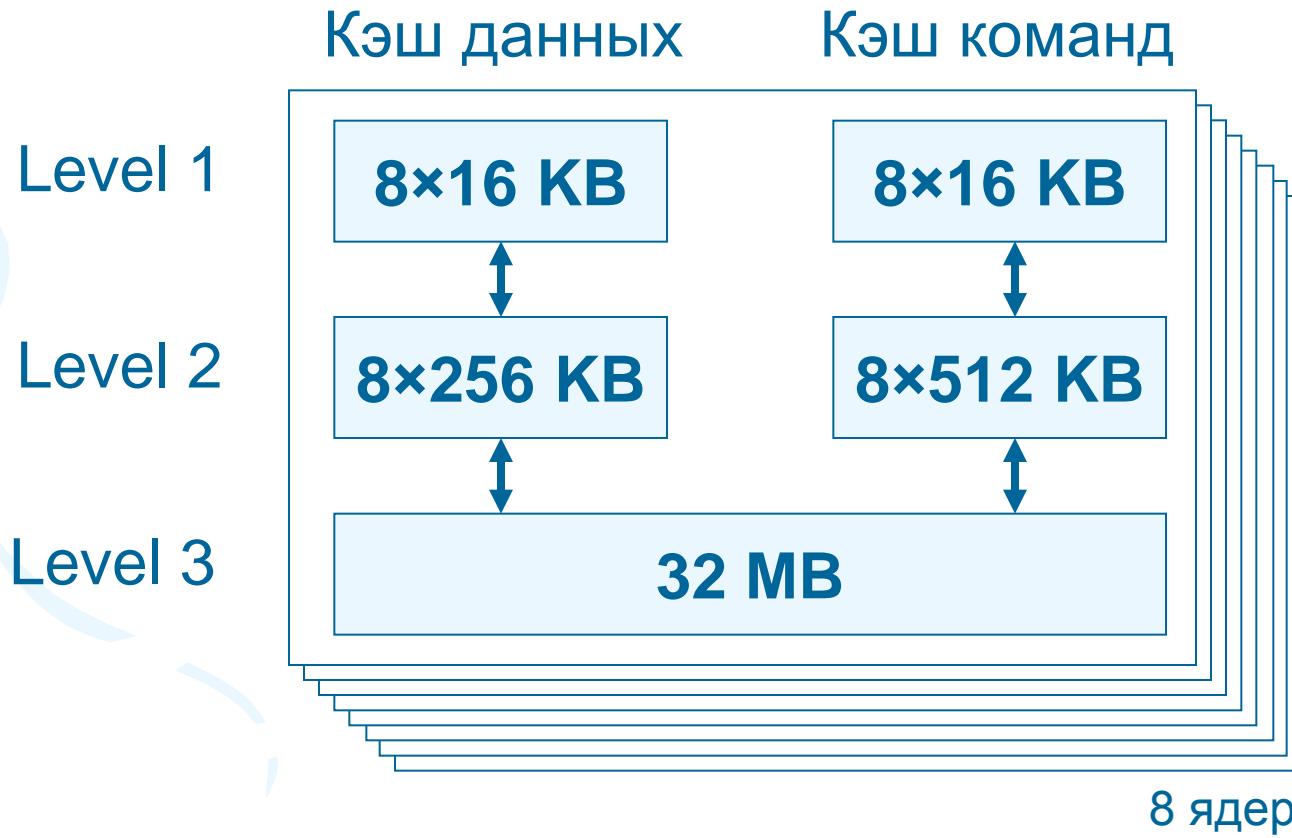
# Процессор Itanium 9500 (Poulson)

## Особенности последнего Itanium-а

- Частота: 2.53 GHz (+Turbo boost)
- 8 ядер
- Hyper-threading – 2 потока на ядро
- Интегрированные контроллеры памяти
- Шина QPI – Quick Path Interconnect



# Кэш-память процессора Itanium 9300 (Tukwila)



Intel Intel Xeon E7-8870 (2011): 30 MB + 10\*256 KB  
AMD Opteron 6180 SE (2011): 12 MB + 12\*512 KB  
IBM POWER7 (2010): 32 MB + 8\*256 KB

# Сравнение современных микропроцессоров

Процессор	Год	Число ядер	Частота	Пиковая производительность	
				одинарная точность	двойная точность
Intel Itanium 9560 (Poulson)	2012	8	2.53 GHz	$8 \times 20.26 = 162.11$ GFLOPS	$8 \times 10.13 = 81.05$ GFLOPS
Fujitsu SPARC64 VIIIfx	2009	8	2.0 GHz	$8 \times 16 = 128$ GFLOPS	$8 \times 16 = 128$ GFLOPS
IBM Power7+	2012	8	4.4 GHz	$2 \times 35.2 = 281.6$ GFLOPS	$8 \times 17.6 = 140.8$ GFLOPS
Intel Xeon X5690 (Westmere-EP)	2011	6	3.46 GHz (3.73 GHz)	$6 \times 27.68 = 166.08$ GFLOPS	$6 \times 13.84 = 83.04$ GFLOPS
Intel Xeon E7-8870 (Westmere-EX)	2011	10	2.4 GHz (2.8 GHz)	$10 \times 19.2 = 192$ GFLOPS	$10 \times 9.6 = 96$ GFLOPS
AMD Phenom II X6 1100T (Thuban)	2010	6	3.3 GHz	$6 \times 26.4 = 158.4$ GFLOPS	$6 \times 13.2 = 79.2$ GFLOPS
AMD Opteron 6282 SE (Interlagos)	2012	16	2.6 GHz	$16 \times 20.8 = 332.8$ GFLOPS	$16 \times 10.4 = 166.4$ GFLOPS
IBM PowerXCell 8i	2008	1 PPE 8 SPE	3.2 GHz	$8 \times 25.6 + 25.6 = 230.4$ GFLOPS	$8 \times 12.8 + 6.4 = 108.8$ GFLOPS

# Процессоры Transmeta



# Процессоры Transmeta

## Особенности архитектуры

- Архитектура VLIW
- Динамическая трансляция кода: x86 → VLIW
- Интегрированный северный мост
- Ориентация на низкое энергопотребление

## Процессоры

- Crusoe (2000) 1.0 GHz
- Effusion (2003) 1.7 GHz

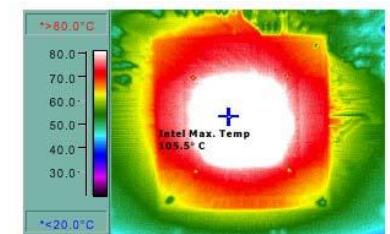


Figure 3: A Pentium III processor plays a DVD at 105° C (221° F).

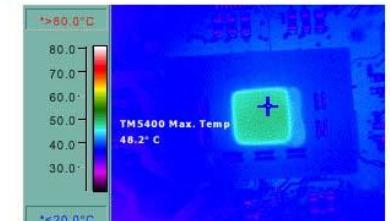
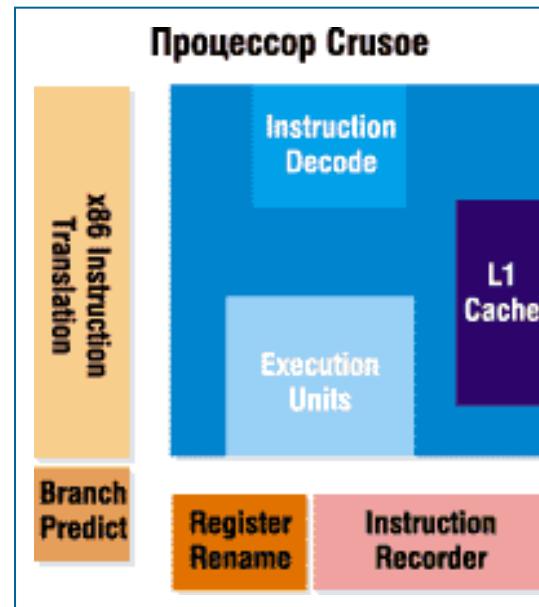


Figure 4: A Crusoe processor model TM5400 plays a DVD at 48° C (118° F).

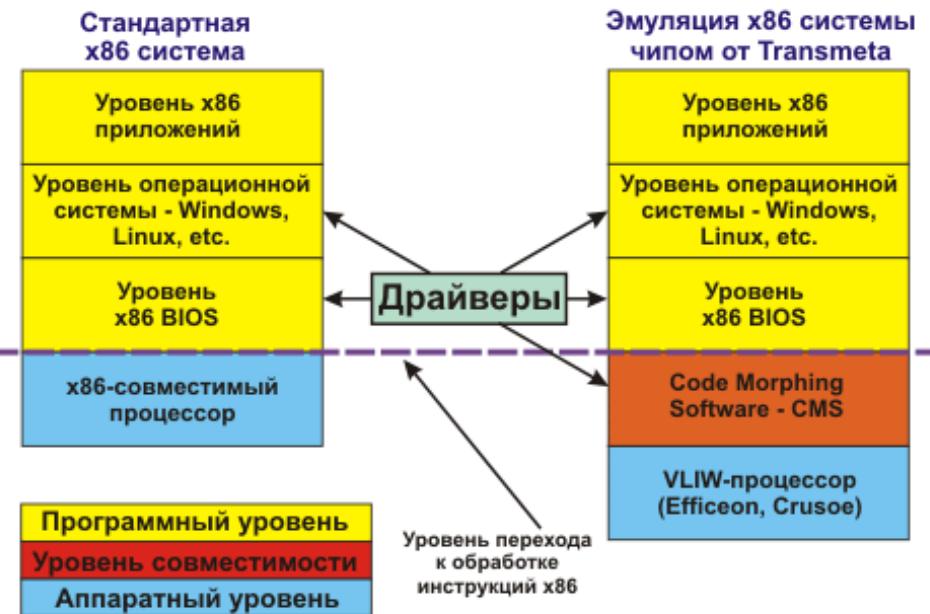
# Динамическая двоичная компиляция

- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



# Динамическая двоичная компиляция

- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



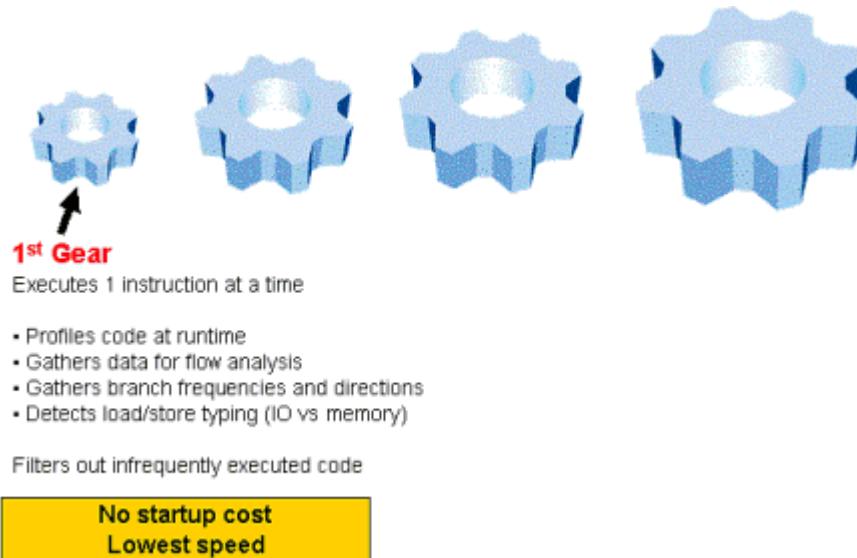
Простое изменение  
входной

системы команд

- исправление ошибок
- оптимизация процесса трансляции
- расширение системы команд
- поддержка различных программных архитектур

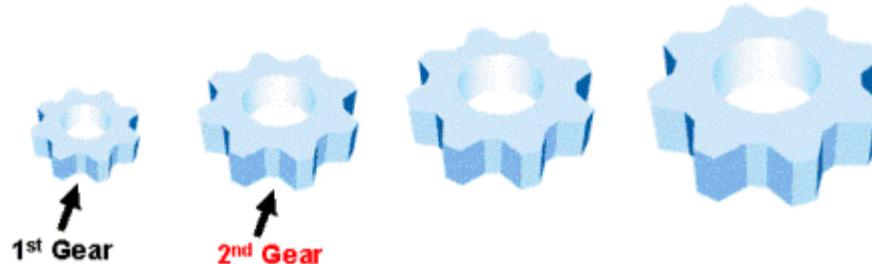
# Динамическая двоичная компиляция

- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



# Динамическая двоичная компиляция

- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



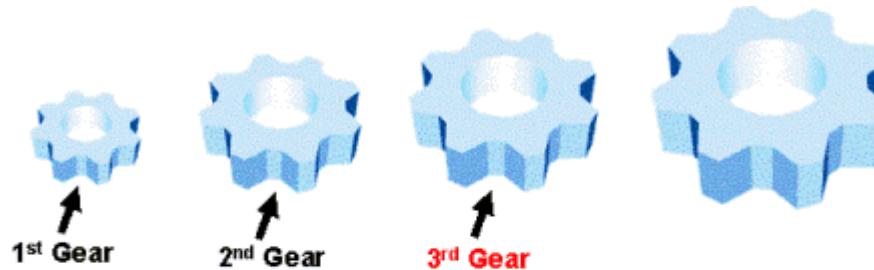
Uses profile data to create initial translations after code reaches 1<sup>st</sup> threshold.

- Translates a "Region" of up to 100 x86 instructions.
- Adds flow graph "Shape" information
- Light Optimization
- "Greedy" scheduling

Low translation overhead  
Fast execution

# Динамическая двоичная компиляция

- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



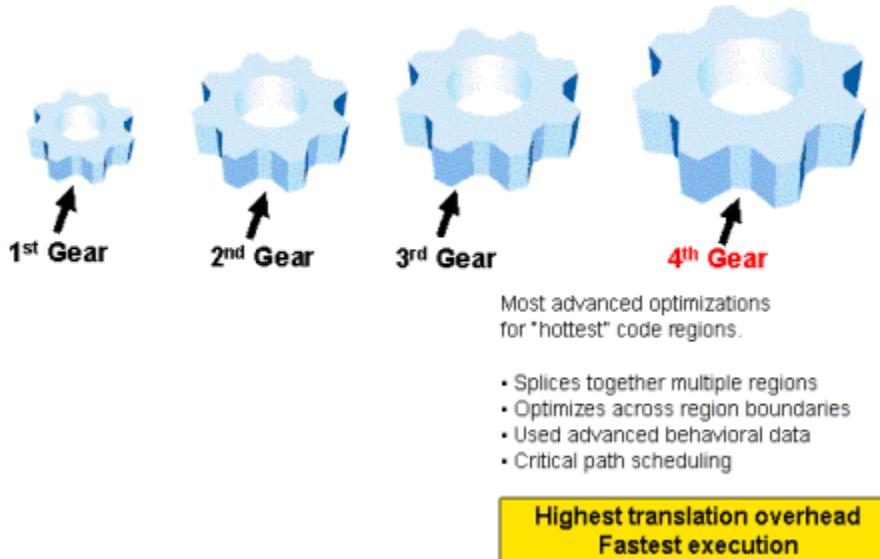
Further optimizes the 2<sup>nd</sup> gear regions

- Common sub-expression elimination
- Memory re-ordering
- Significant code optimization
- Critical path scheduling

Medium translation overhead  
Faster execution

# Динамическая двоичная компиляция

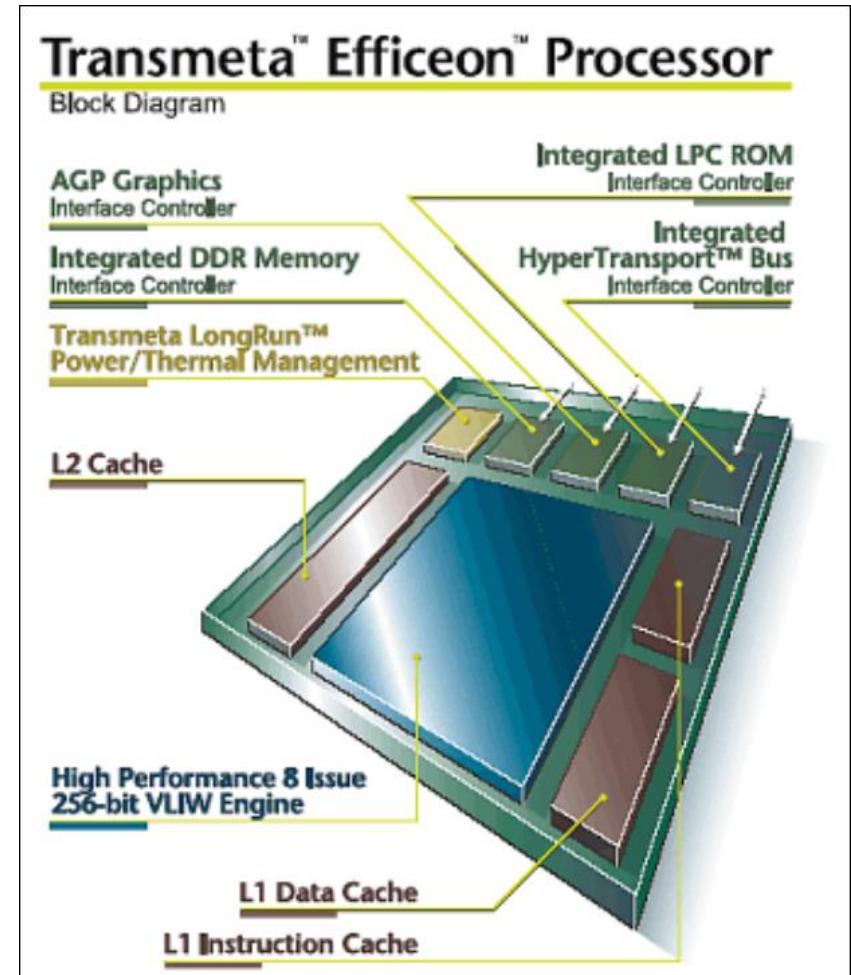
- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 МВ)
  - Динамическая оптимизация VLIW-кода



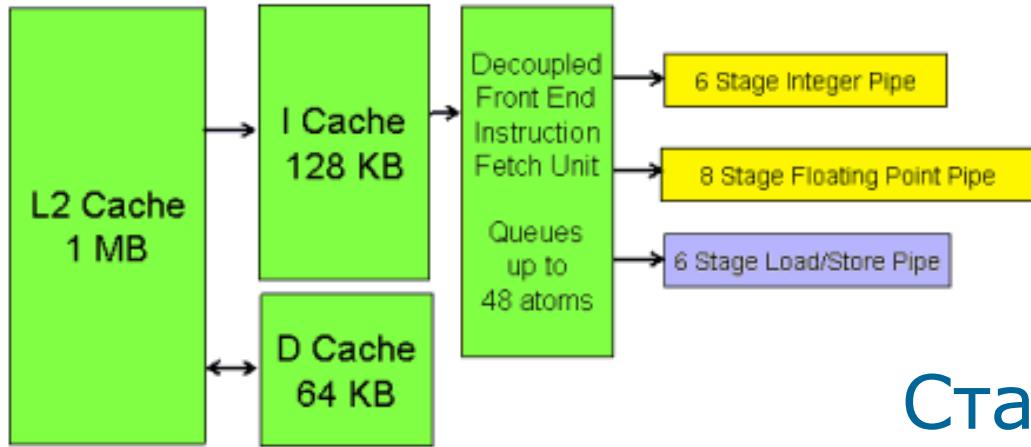
# Процессор Transmeta Efficeon

## Особенности

- Ширина командного слова 256 бит (8 команд)
- Кэш L1: 64 data / 128 KB code
- Кэш L2: 1 MB
- Интегрированный северный мост
  - Контроллер памяти DDR
  - Шина AGP
  - Шина HyperTransport
- Технология энергосбережения LongRun



# Процессор Transmeta Efficion



## Стадии конвейера

### 6-Stage Integer Pipe

IS	DR	RM	EM	CM	WB
----	----	----	----	----	----

IS: Instruction Issue

DR: Instruction Decode

RM: Register Read for ALU operands

EM: Execute ALU operation

CM: ALU Condition flag completion

WB: Write Back results to integer register file

### 8-Stage Floating Point Pipe

IS	DR	DT	XA	XB	XC	XD	WB
----	----	----	----	----	----	----	----

IS: Instruction Issue

DR: Decode-1

DT: Decode-2

XA: Floating Point compute stage-1

XB: Floating Point compute stage-2

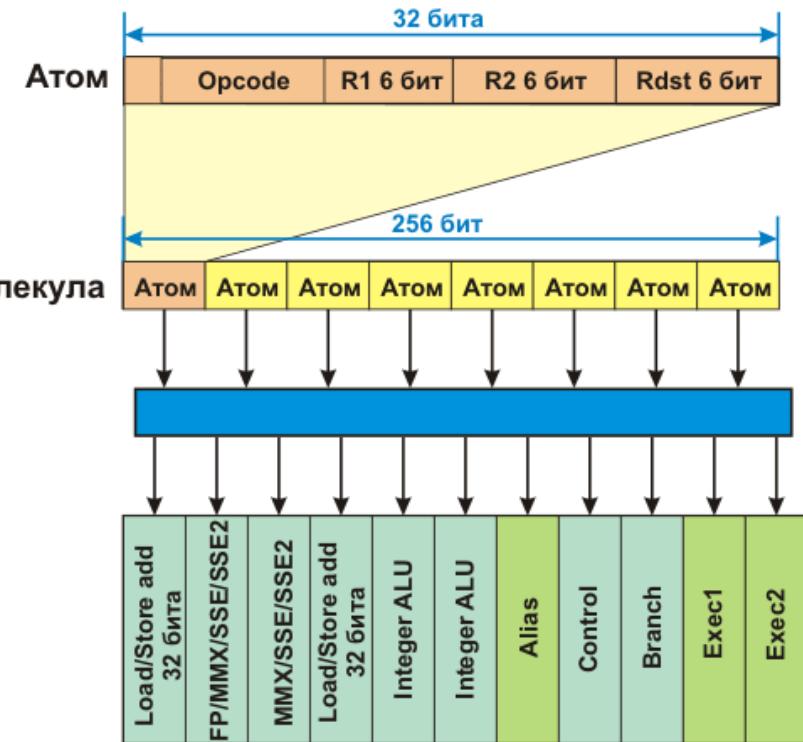
XC: Floating Point compute stage-3

XD: Floating Point compute stage-4

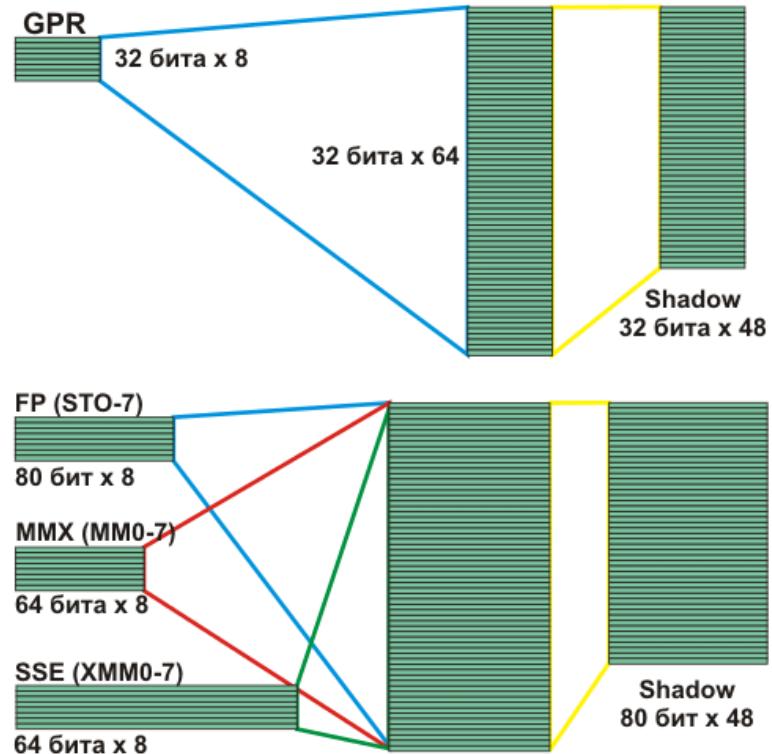
WB: Write Back to floating point register file

# Процессор Transmeta Efficion

## Структура команды



## Отображение регистров



Исполнительные устройства



Бабаян  
Борис  
Арташесович  
чл.корр. РАН  
*Intel Fellow*

# Архитектура Эльбрус 2000



# Эльбрус 2000

ELBRUS – ExpLicit Basic Resources Utilization Scheduling  
(явное планирование использования основных ресурсов)

## Особенности архитектуры Е2К

- Архитектура VLIW переменной длины
- Динамическая трансляция кода: x86 → VLIW
- Аппаратная поддержка типов данных

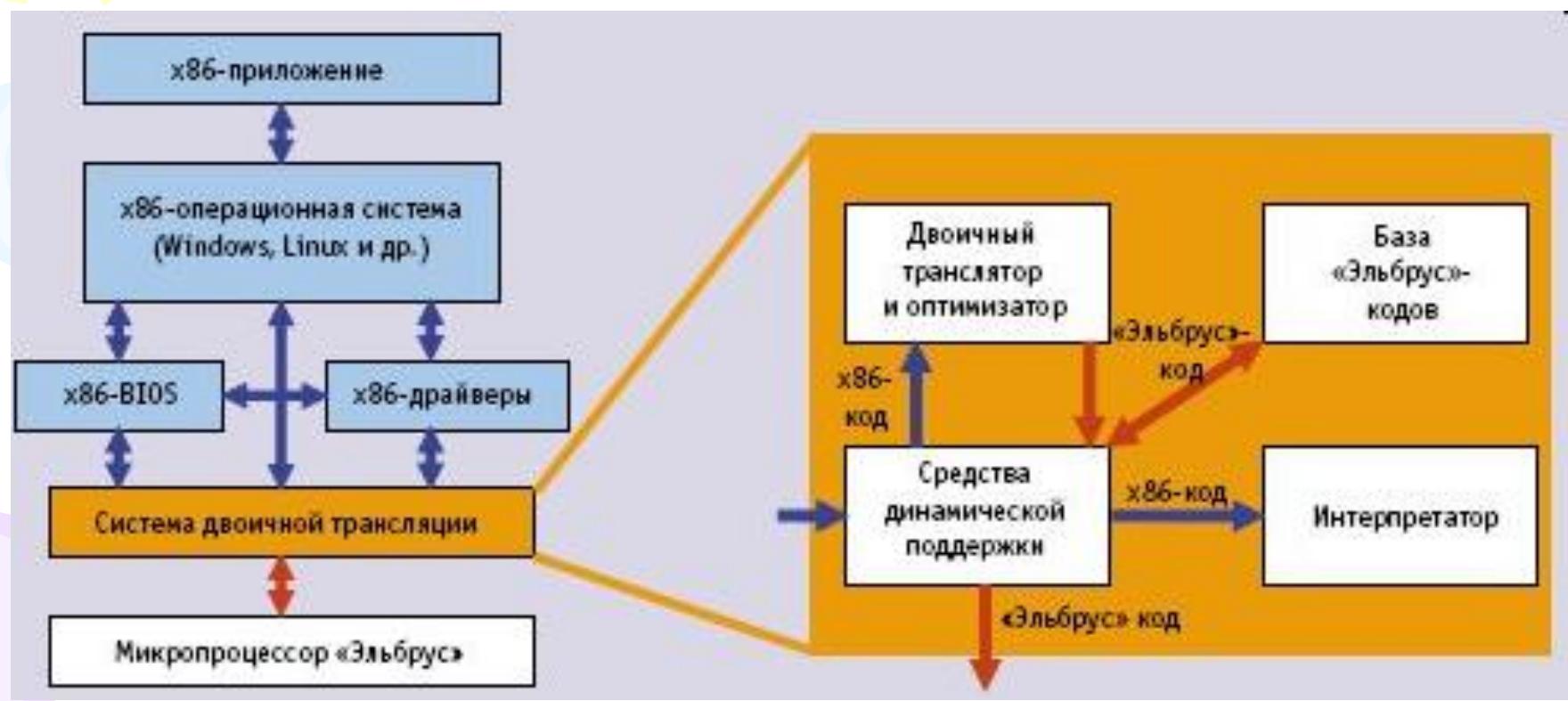
## Реализации

- МВК Эльбрус 3 (1986-1994)
- Эльбрус 3М (2005) 300 MHz
- Эльбрус S (2010) 500 MHz
- Эльбрус 2С+ (2011) 500 MHz



# Эльбрус 2000

- Динамическая трансляция кода



# Эльбрус 2000

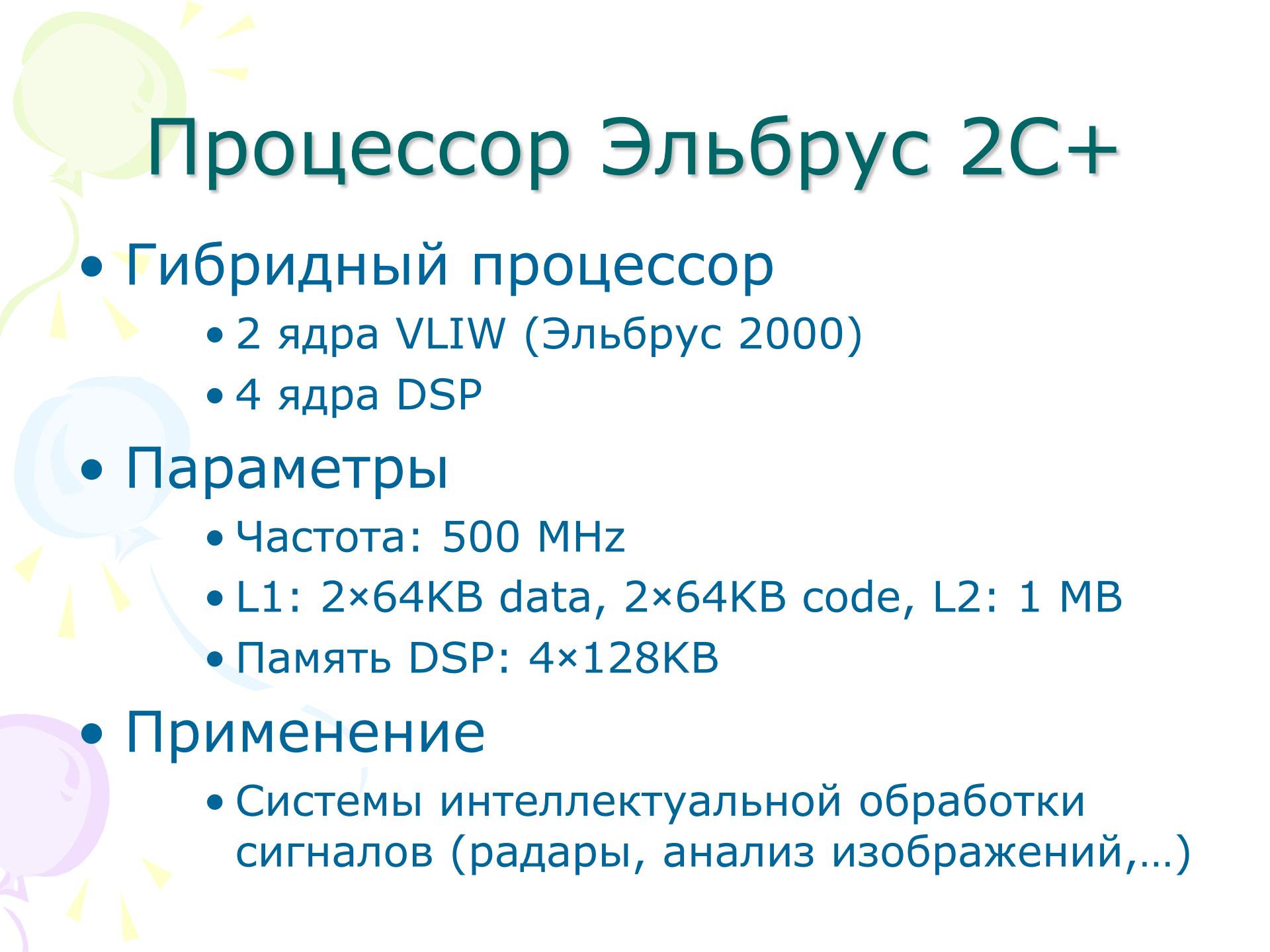
- Формат команды:
  - Переменное число слогов: 2 – 16
  - Типы слогов (максимальное число в команде)
    - Заголовок (1)
    - Операции АЛУ (6)
    - Управление подготовкой перехода (3)
    - Дополнительные операции АЛУ при зацеплении (2)
    - Загрузка из буфера предварительной выборки массивов в регистр (4)
    - Литеральные константы для ФУ (4)
    - Логические операции с предикатами (3)
    - Предикаты и маски для управления ФУ (3)
  - До 6 предикатов в команде
- Регистры
  - Общего назначения: 256 (64 бита): целочисл. и веществ.
    - Механизм переключения окон
  - 32 предикатных регистра (1 бит)

Заголовок	Слог 1	Слог 2	...	Слог N
-----------	--------	--------	-----	--------

# Процессор Эльбрус 3М

Характеристики:

- Частота: 300 MHz
- Исполнение до 23 операций за такт!!!
- Конвейер
  - Целочисленный: 8 тактов
  - Чтение/запись: 9 тактов
- Разрядность данных
  - Целые: 32, 64 bit
  - Вещественные: 32, 64, 80 bit
- Кэш-память
  - L1: d:64 + i:64 KB
  - L2: 256 KB
- Пиковая производительность:
  - 23.7 GIPS
  - 2.4 GFLOPS



# Процессор Эльбрус 2С+

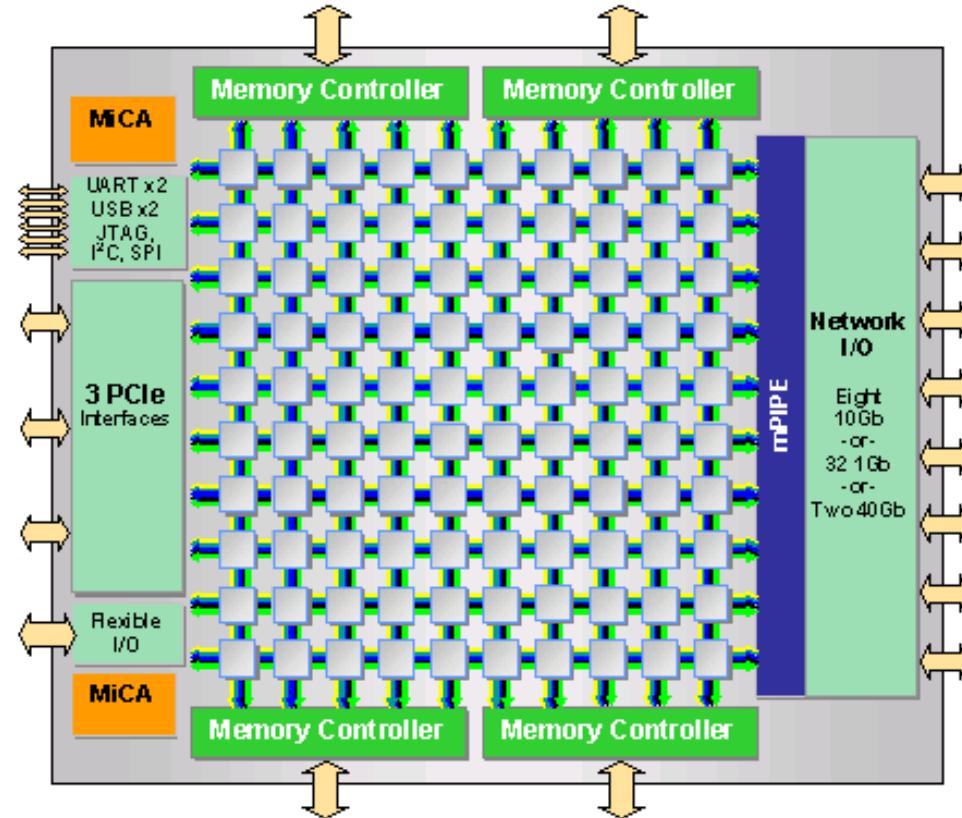
- Гибридный процессор
  - 2 ядра VLIW (Эльбрус 2000)
  - 4 ядра DSP
- Параметры
  - Частота: 500 MHz
  - L1: 2×64KB data, 2×64KB code, L2: 1 MB
  - Память DSP: 4×128KB
- Применение
  - Системы интеллектуальной обработки сигналов (радары, анализ изображений,...)

# Процессоры Tilera



# Процессоры Tile-Gx

- 16-100 ядер
  - 64-битная архитектура
  - 64-bit VLIW: 3 инстр./такт
  - 3-стадийный конвейер
  - L1: 32+32 KB
  - L2: 256 KB
- 1.0 – 1.5 GHz
- Сеть: 2D решетка
- Внешние интерфейсы
  - DDR3
  - PCI-E
  - Network
  - ...



Many Core :)

