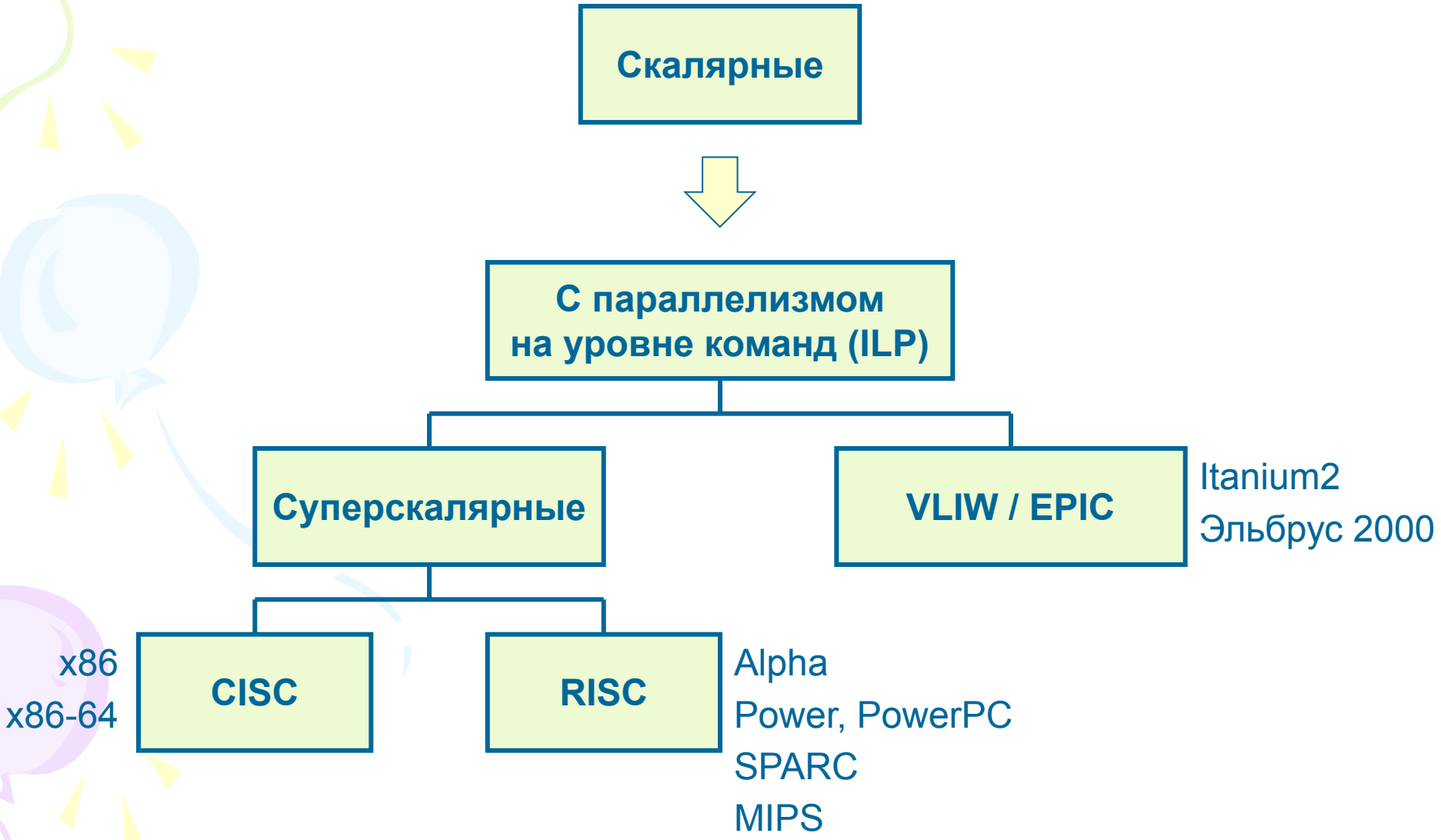
Архитектура VLIW / EPIC

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 планирования потока команд.
  - Невозможность учесть динамику исполнения программы

Сравнение суперскалярных и VLIW/EPIC-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/EPIC-процессоров

Какие задачи управления приходится решать, чтобы процессор работал быстро:

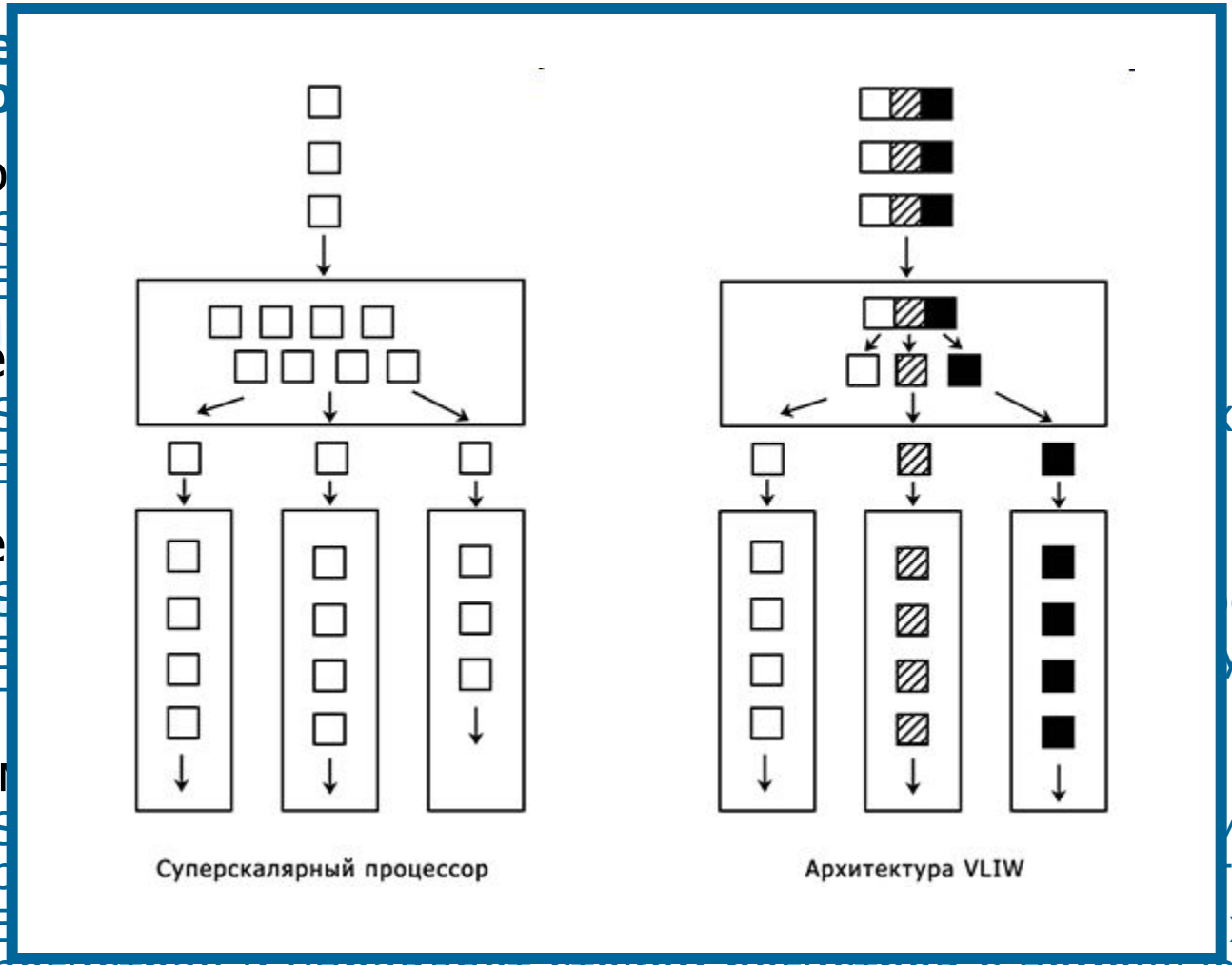
- Параллельное исполнение команд
  - SS: Независимые команды ищет процессор
  - EPIC: Независимые команды ищет компилятор
- Спекулятивное исполнение команд
  - SS: Процессор автоматически предсказывает переход
  - EPIC: Компилятор подсказывает процессору как поступить
- Спекулятивная загрузка данных
  - 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

суперскалярных и VLIW/EPIC-процессоров

Какие … чтобы про…

- Пар…
  - …
  - …
- Спе…
  - …код
  - …
- Спе…
  - …сть
  - …у
  - …
- Разн…
  - …ные
  - …тров
  - …х

регистрах и управляет стеком регистров с помощью специальных команд



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/EPIC-процессоров

Суперскалярные	VLIW-EPIC
Простой компилятор, процессор планирует поток команд ⇓ **Меньше** команд за такт: • 3, 4, 5 (в среднем < 50%)	**Сложный компилятор планирует поток команд** ⇓ **Больше** команд за такт: • 6, 8, до 23 (в среднем > 50%)
Сложный исполнительный конвейер ⇓ **Меньше** места на кристалле для ресурсов процессора • Исполнительные устройства • Регистры, кэш-память	**Простой исполнительный конвейер** ⇓ **Больше** места на кристалле для ресурсов процессора • Исполнительные устройства • Регистры, кэш-память

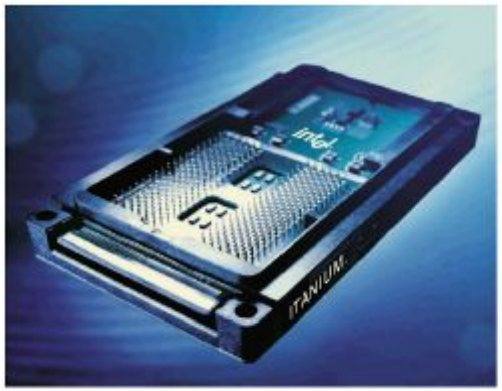
Сравнение конвейеров

Этапы обработки команды

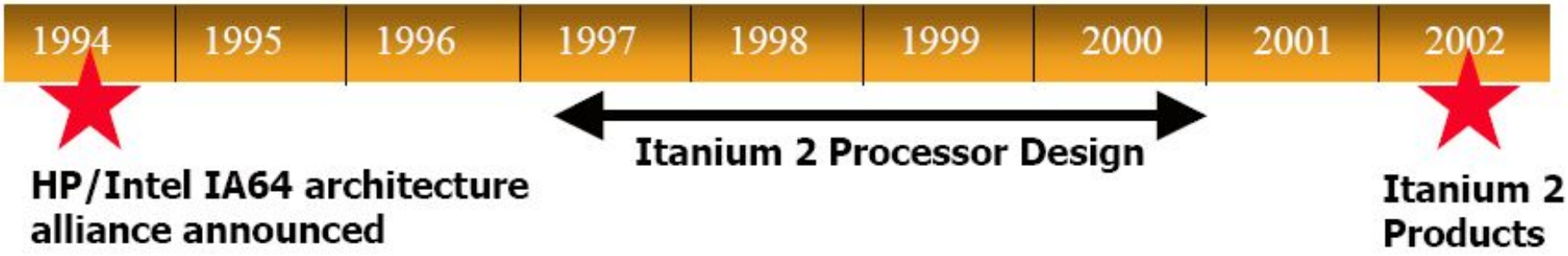
- Предсказание ветвлений
- Выборка
- Декодирование в RISC
- Переименование регистров
- Переупорядочение и распараллеливание
- Исполнение
- Завершение

Архитектура VLIW / EPIC

- История
  - M-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



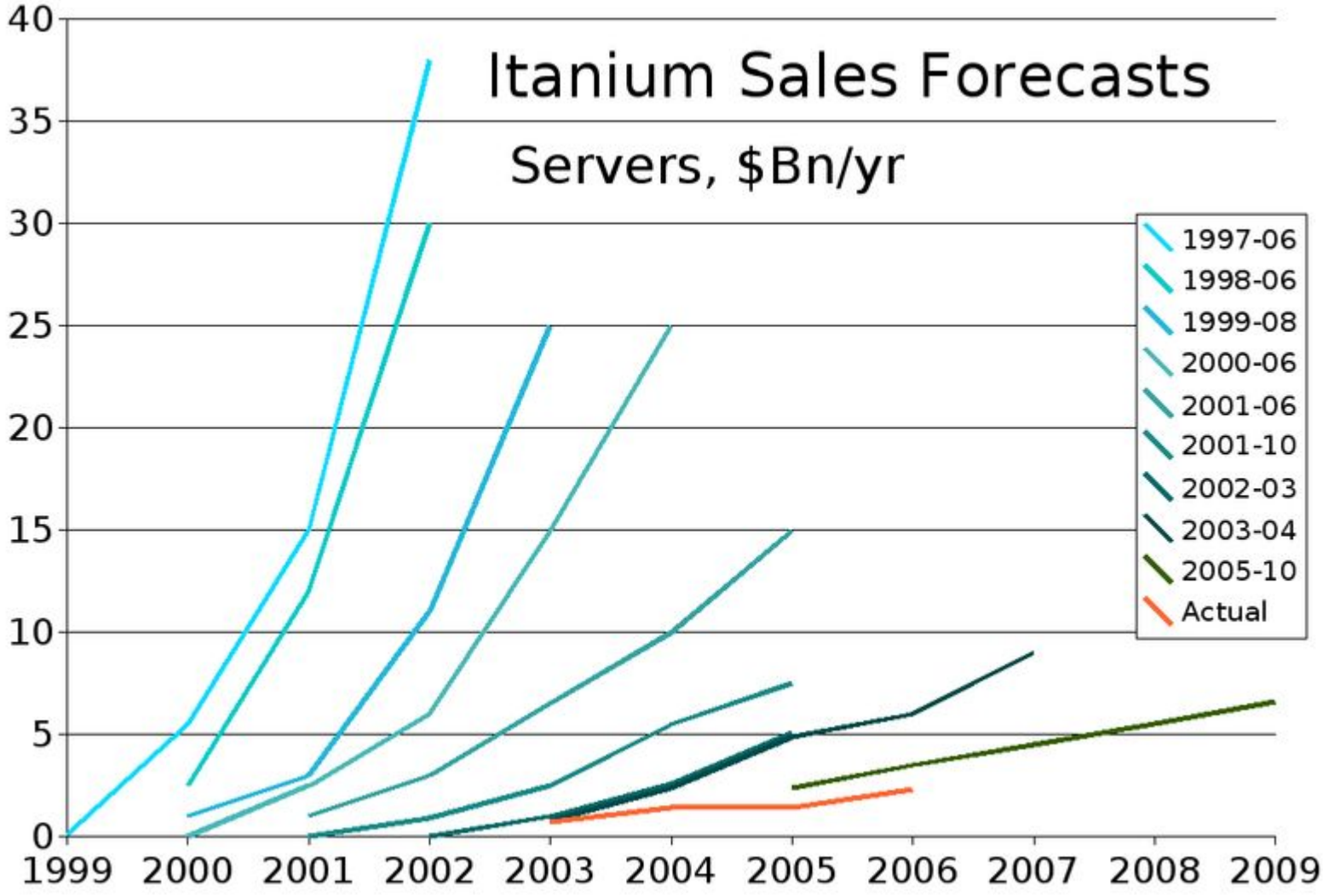
Архитектура Itanium



Семейство процессоров Itanium

2001	2002	2003	2006	2010
				
Itanium (Merced) 800 MHz 4 MB L3 cache 180 nm	Itanium2 (McKinley) 1 GHz 3 MB L3 cache 180 nm	Itanium2 (Madison) 1.5 GHz 6 MB L3 cache 130 nm	Itanium2 (Montecito) 1.66 GHz 2×12 MB L3 cache 2 cores HyperThreading 90 nm	Itanium (Tukwila) 1.73 GHz 24 MB L3 cache 4 cores HyperThreading 65 nm

Itanium: планы и реальность



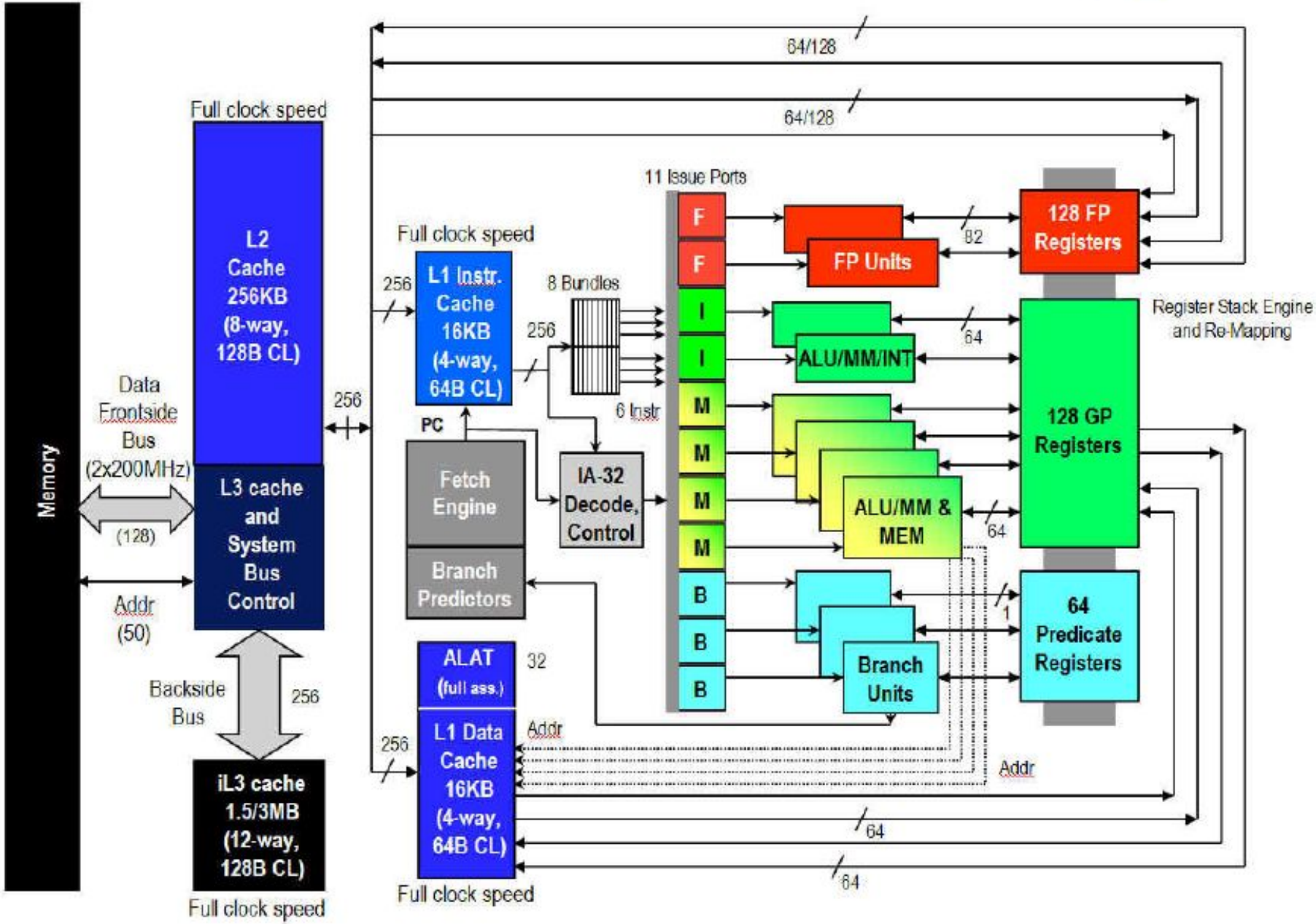
Архитектура Itanium (IA-64)

- **Явный ILP (параллелизм на уровне команд)**
  - Компилятор объединяет команды процессора в связки, которые могут быть выполнены параллельно,
  - Процессор обеспечивает большое число ресурсов для реализации ILP.
- **Способы увеличения ILP**
  - Явная спекуляция по данным и управлению (уменьшает задержки по памяти),
  - Предикатное исполнение команд (устраняет ветвления),
  - Аппаратная поддержка программной конвейеризации циклов,
  - Предсказание ветвлений.
- **Специальные способы увеличения производительности программ**
  - Специальная поддержка модульности программ (регистровый стек, вращающиеся регистры),
  - Высокопроизводительная вещественная арифметика,
  - Специальные векторные инструкции.

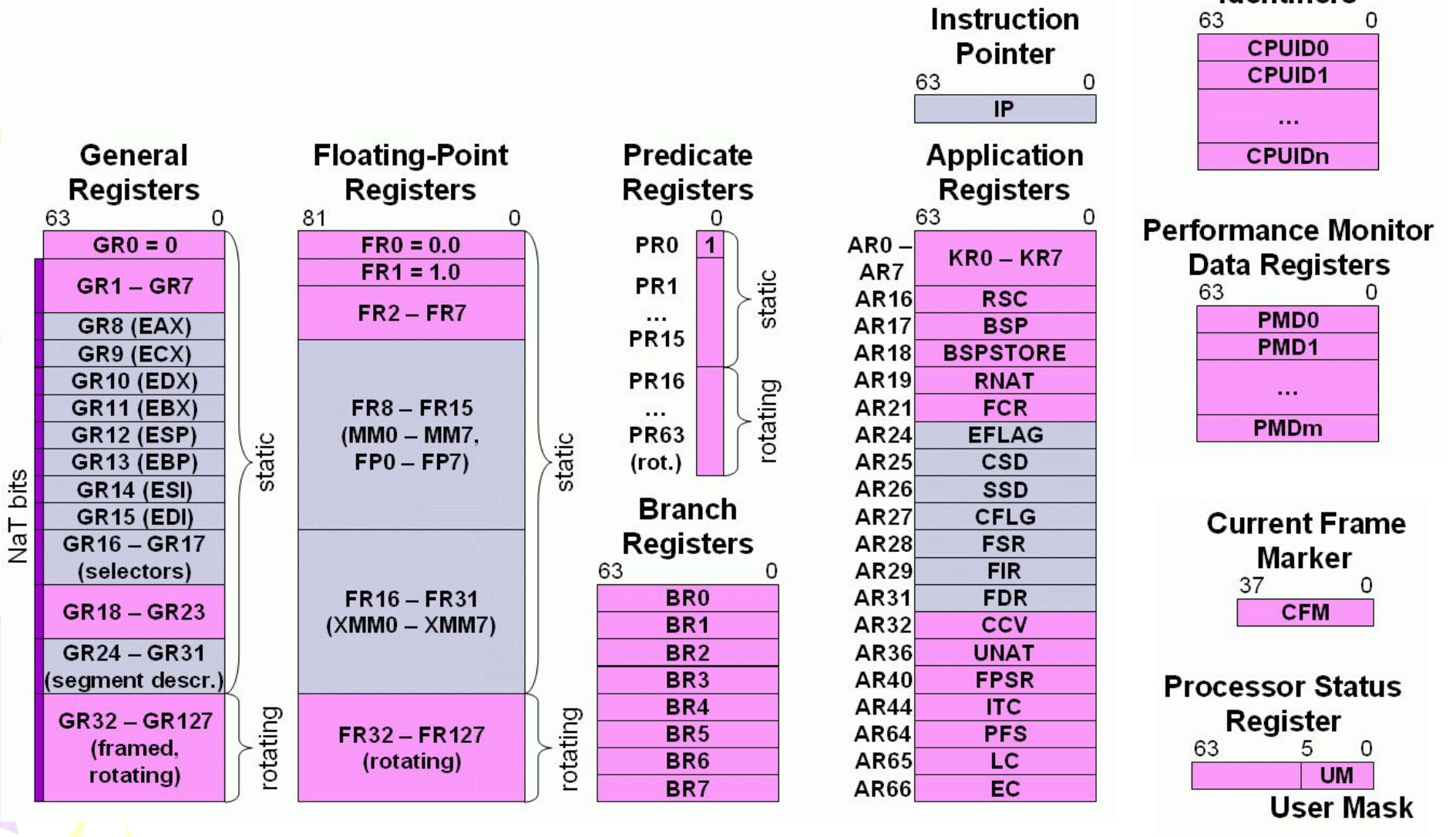
Особенности процессоров архитектуры Itanium (IA-64)

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

Itanium® 2 Processor Block Diagram



Регистры IA-64

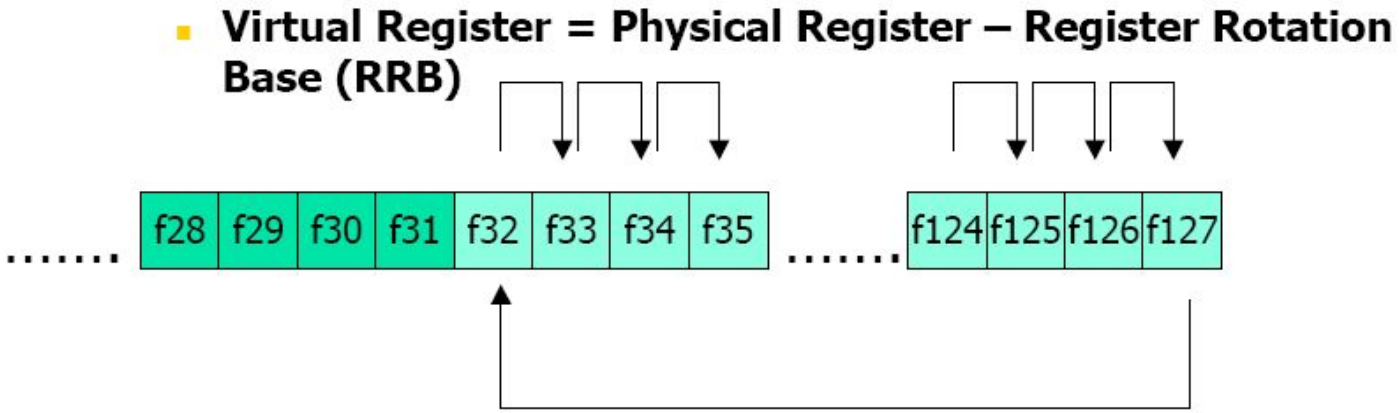


Регистры IA-64

- 128 целочисленных регистра
  - 64 бита + 1 бит NAT
  - r0 = 0
  - целочисленные скалярные и векторные данные (1,2,4,8 байт)
- 128 вещественных регистра
  - 82 бита (17 + 64 + 1)
  - f0 = 0.0, f1 = 1.0
  - вещественные скалярные и векторные данные (82, 2x32 бита)
- 64 предикатных регистра
  - 1 бит
  - p0 = 1
  - указания, выполнять ли команду
- 8 регистров ветвлений
  - 64 бита
  - адреса перехода
- 128 прикладных регистра
- Instruction Pointer

Вращение регистров

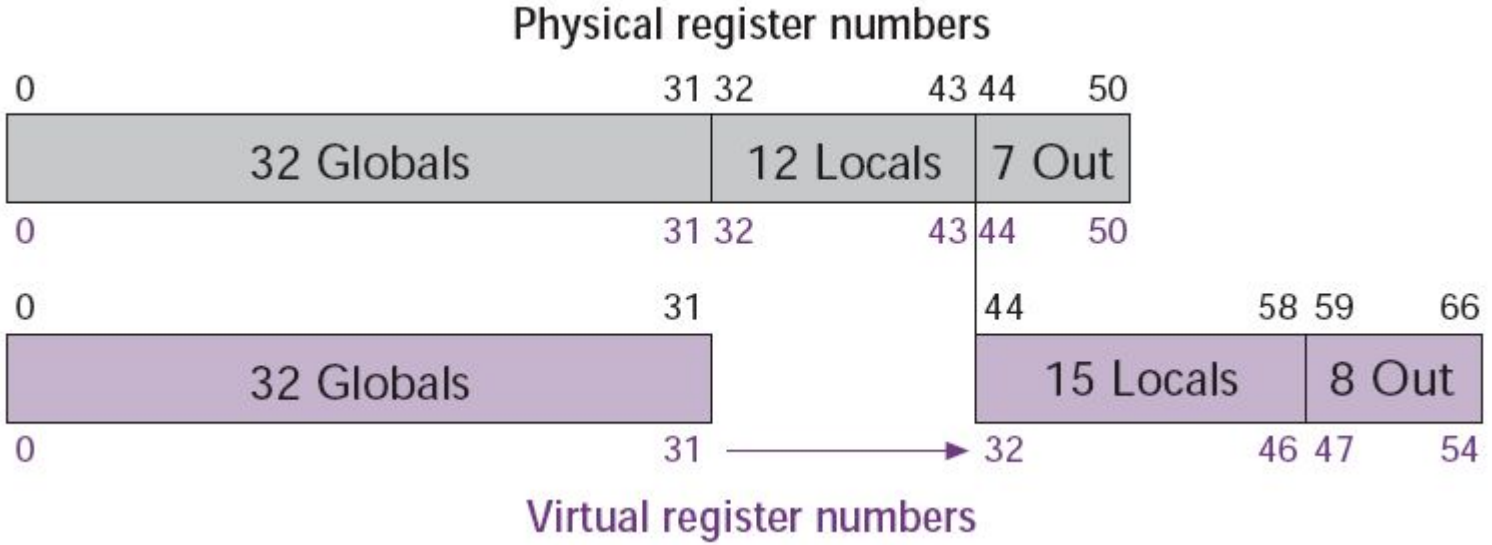
- Верхние 75% регистров вращающиеся:
  - целочисленные: r32 – r127
  - вещественные: f32 – f127
  - предикатные: p16 – p63
- При выполнении специальной команды перехода (в цикле) вращающиеся регистры сдвигаются вправо на один:



- Используется при программной конвейеризации циклов.

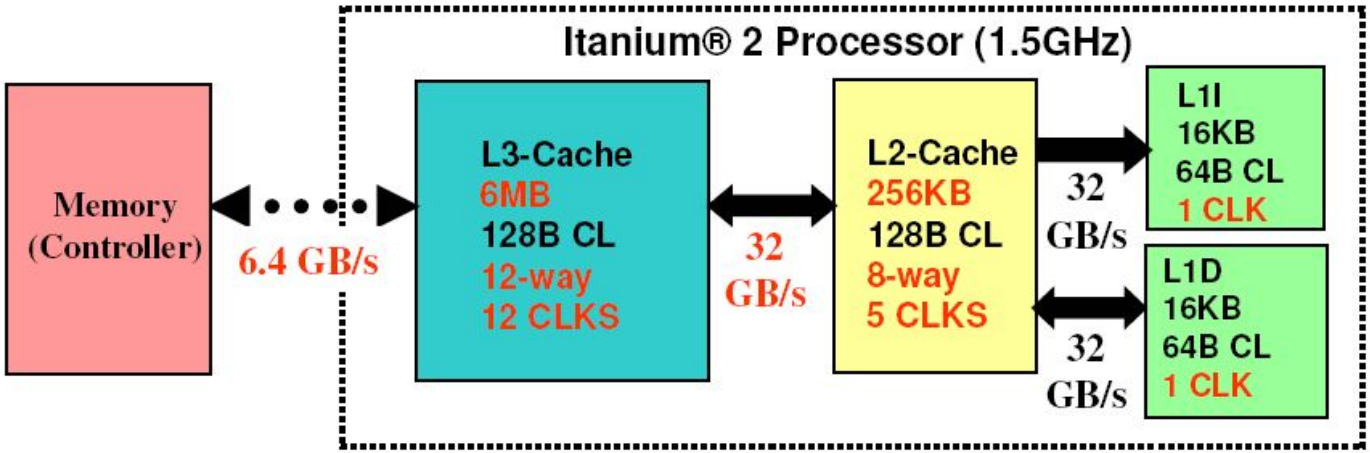
Стек регистров

- 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.

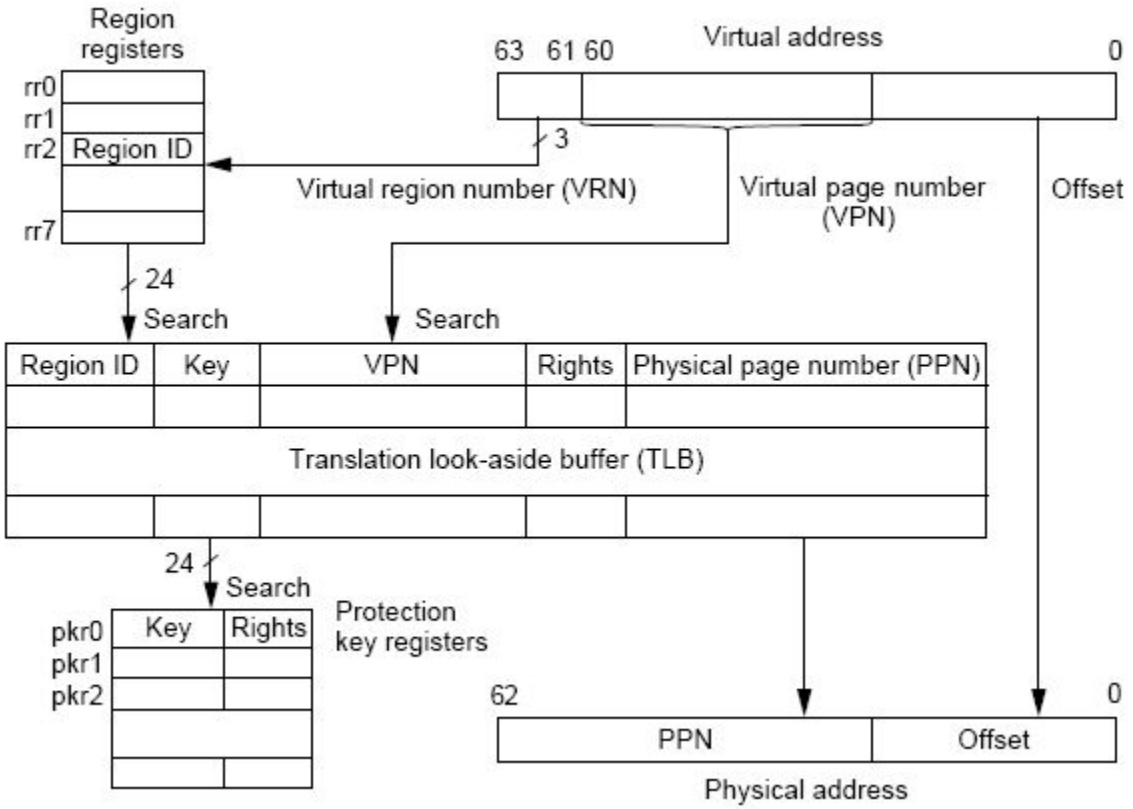
Itanium2



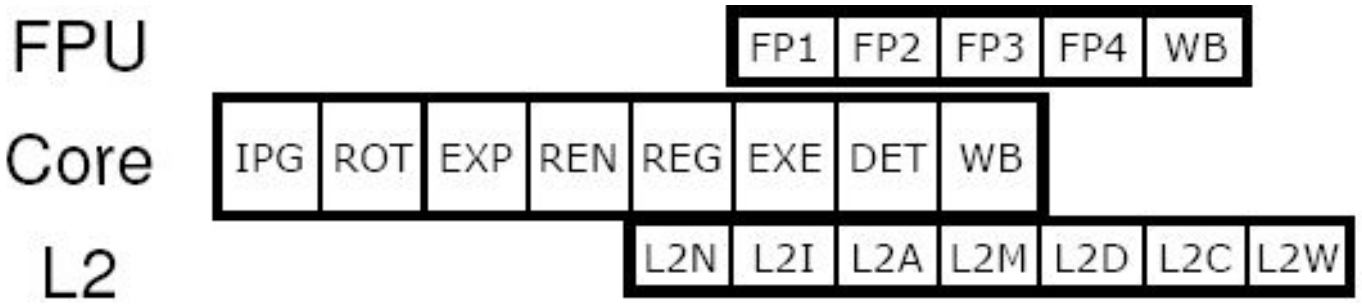
	L1I	L1D	L2	L3
Size	16K	16K	**256K**	**12M on die**
Line Size	64B	64B	128B	128B
Ways	4	4	**8**	**12**
Replacement	LRU	NRU	NRU	NRU
Latency (load to use)	**I-Fetch:1**	**INT:1**	**INT: 5 FP: 6**	**INT: 12 FP: 13**
Write Policy	-	WT (RA)	WB (WA + RA)	WB (WA)
Bandwidth	**R: 32 GBs**	**R: 16 GBs W: 16 GBs**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

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:



Конвейер Itanium2

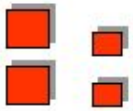
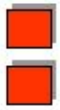
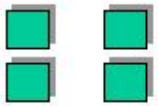
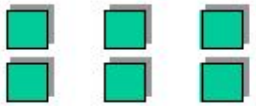
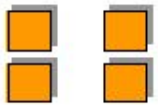
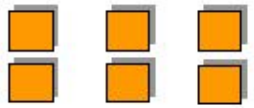
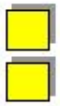
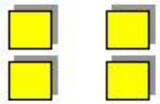
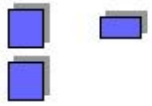
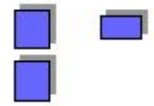


IPG	Вычисление IP, чтение кэша L1I (**6** инст.) и TLB.	**EXE**	Выполнение (**6**), обращение к кэшу L1D и TLB + обращение к тэгам L2 кэша (**4**)
ROT	Расцепление и буферизация инструкций.	**DET**	Обнаружение исключений, выполнение переходов
EXP	Разворачивание инструкции, назначение порта	**WB**	Завершение,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

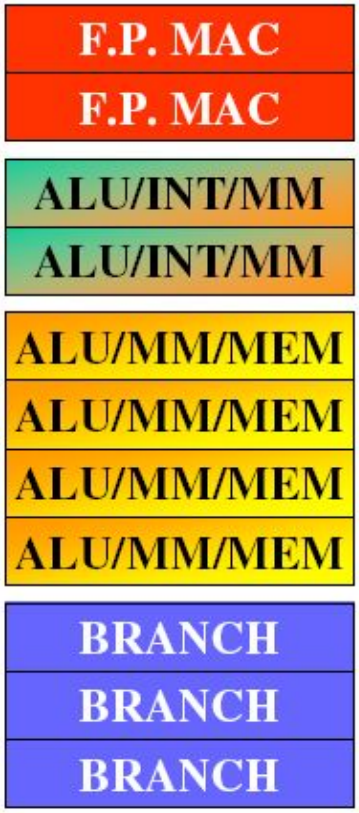
Короткий 8-стадийный конвейер

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

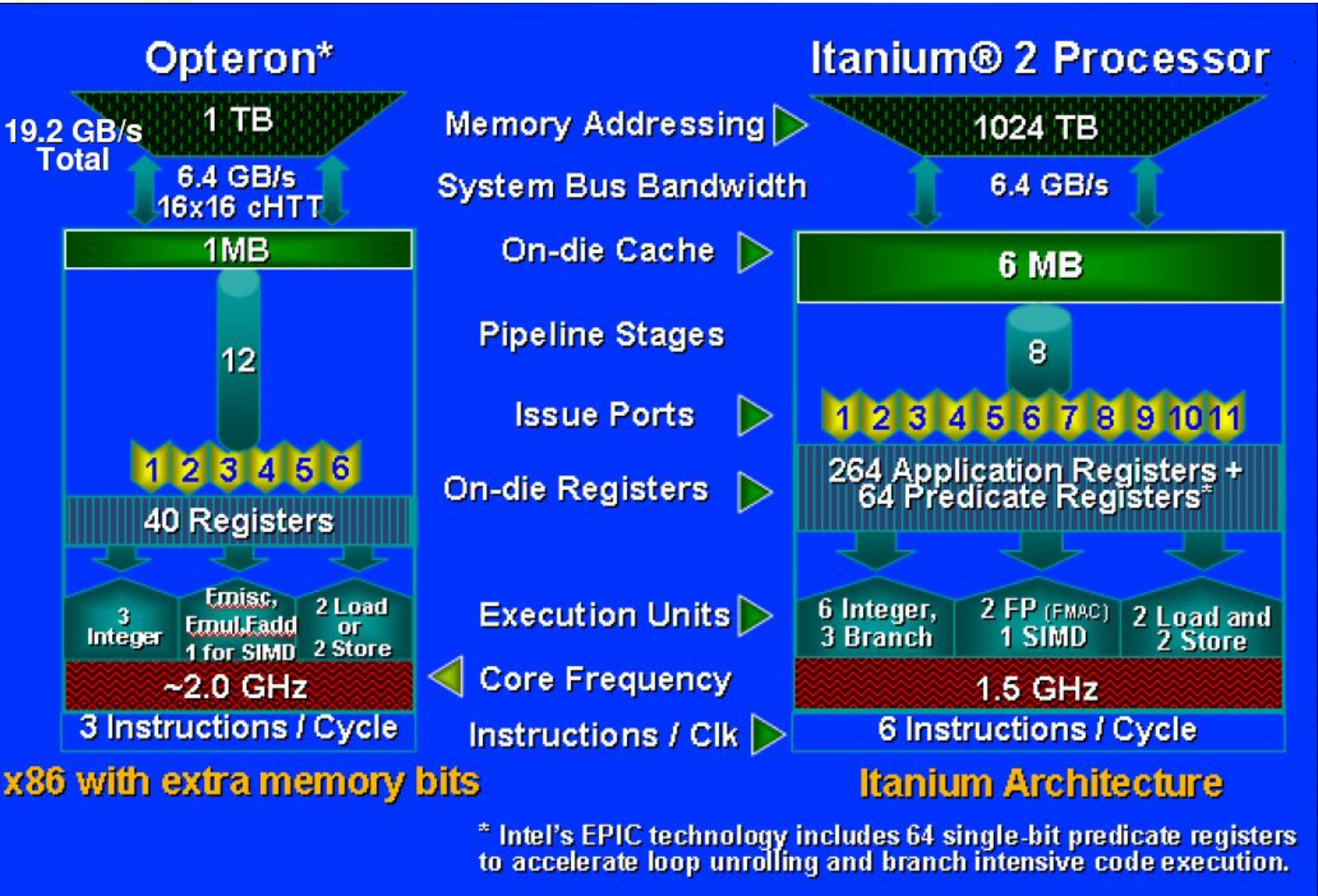
Исполнительные устройства

	Itanium®	Itanium® 2
F.P.		
Integer		
Multimedia		
Load/Store		
Branch		

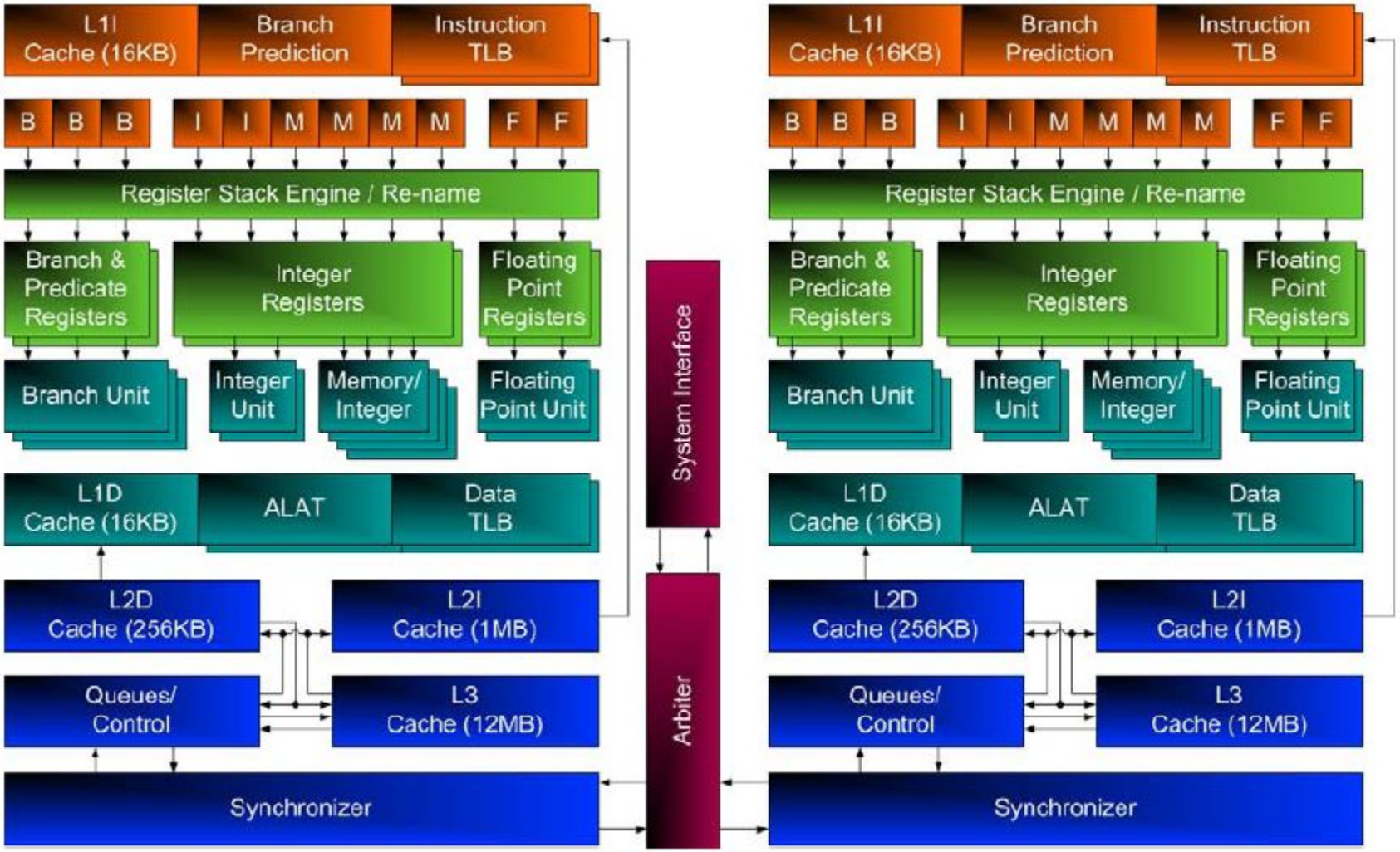
Issue Ports/Units



Сравнение Itanium2 и Opteron



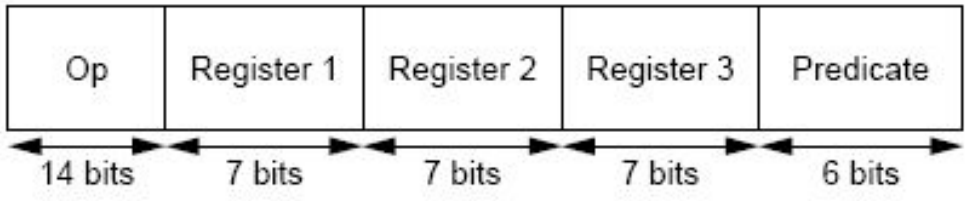
Itanium2 Montecito (2006)



Montecito: 2 ядра по 2 потока (HyperThreading)

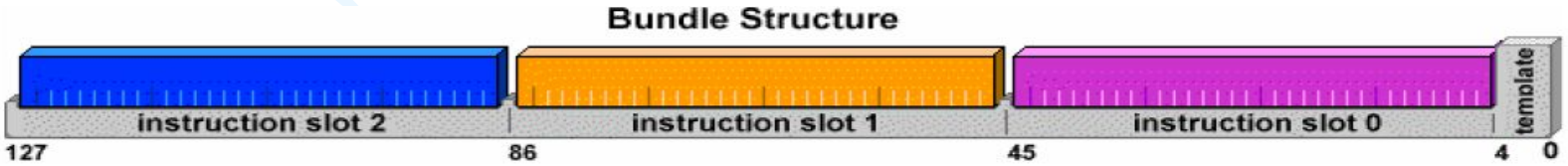
Команды IA-64

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

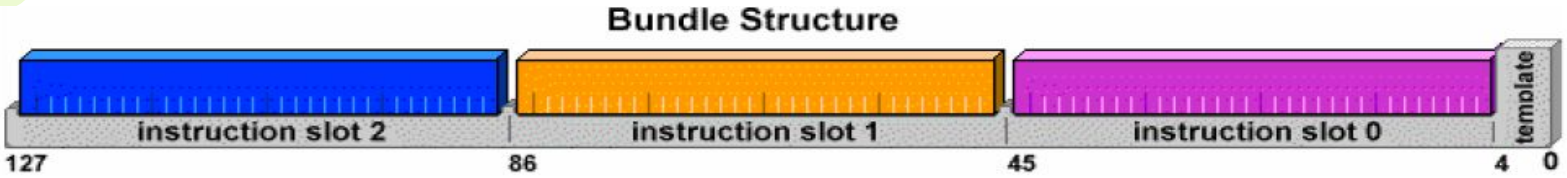


  - Пример команды: (p3) add r1 = r3, r4

- Команды IA-64 объединяются в связки по три:



Команды IA-64



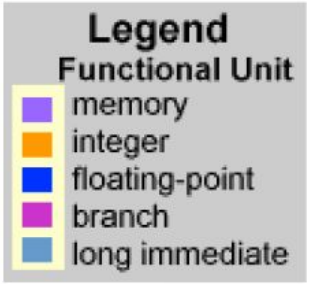
- Связка содержит 3 команды, поле шаблона и стоп-биты.
- Шаблон указывает типы команд в связке. Он определяет, какие исполнительные устройства будут задействованы при исполнении.
- Типы команд: Устройство:
  - M – memory / move M
  - I – complex integer / multimedia I
  - A – simple integer / logic / 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	MIIs	MMFs
MIsI	MIB	MIsIs	MIBs
MLX*	MBB	MLXs*	MBBs
MMI	BBB	MMIs	BBBs
MsMI	MMB	MsMIs	MMBs
MFI	MFB	MFIs	MFBs

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

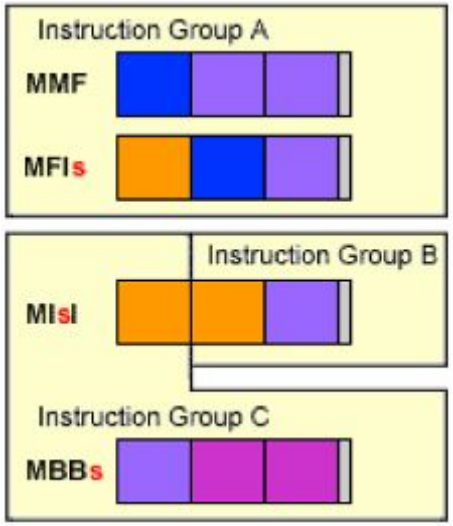


- Процессор загружает максимум по 2 связки за такт.
- Только некоторые сочетания шаблонов в связках могут полностью загрузить исполнительные устройства:

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

* 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 (pmpy, …)
- Команды ветвлений (br, …)
- Команды управления циклом (br.cloop, …)
- Вещественные (fma, …)
- SIMD вещественные (fpma, …)
- Команды чтения / записи данных в памяти (ld, …)
- Команды присваивания (mov, …)
- Команды управления кэшированием (lfetch, …)

Особенности целочисленной арифметики в Itanium2

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

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

(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert < 2^{-8.886}$	table lookup
(2)	$q_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(3)	$e_0 = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(4)	$q_1 = (q_0 + e_0 \cdot q_0)_{rn}$	82-bit register format precision
(5)	$e_1 = (e_0 \cdot e_0 + 2^{-34})_{rn}$	82-bit register format precision
(6)	$q_2 = (q_1 + e_1 \cdot q_1)_{rn}$	82-bit register format precision
(7)	$q = \mathrm{trunc}\ (q_2)$	floating point to signed integer conversion (RZ mode)

Особенности вещественной арифметики в Itanium2

- Максимальная 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 ☐ INT (getf): 5 тактов
  - INT ☐ 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)

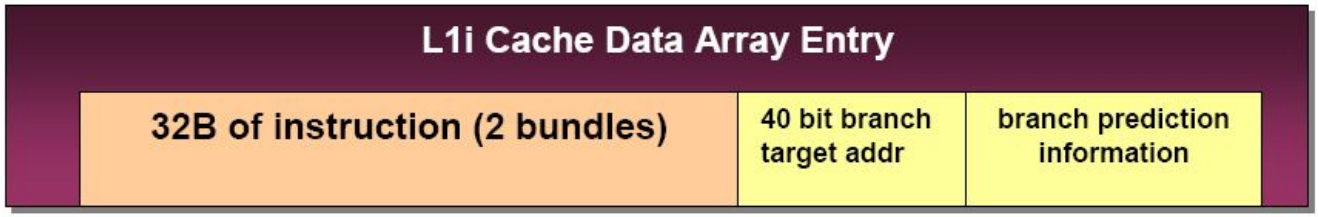
(1)	$y_0 = 1 / b \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.886}$	table lookup
(2)	$d = (1 - b \cdot y_0)_{rn}$	82-bit register format precision
(3)	$e = (d + d \cdot d)_{rn}$	82-bit register format precision
(4)	$y_1 = (y_0 + e \cdot y_0)_{rn}$	82-bit register format precision
(5)	$q_1 = (a \cdot y_1)_{rn}$	17-bit exponent, 24-bit mantissa
(6)	$r = (a - b \cdot q_1)_{rn}$	82-bit register format precision
(7)	$q = (q_1 + r \cdot y_1)_{rnd}$	single precision

- Вычисление корня (32-bit float)

(1)	$y_0 = \left(1 / \sqrt{a}\right) \cdot (1+\varepsilon_0),\ \ \lvert\varepsilon_0\rvert<2^{-8.831}$	table lookup
(2)	$H_0 = (0.5 \cdot y_0)_{rn}$	82-bit register format precision
(3)	$S_0 = (a \cdot y_0)_{rn}$	82-bit register format precision
(4)	$d = (0.5 - S_0 \cdot H_0)_{rn}$	82-bit register format precision
(5)	$d' = (d + 0.5 * d)_{rn}$	82-bit register format precision
(6)	$e = (d + d * d')_{rn}$	82-bit register format precision
(7)	$S_1 = (S_0 + e * S_0)_{rn}$	17-bit exponent, 24-bit mantissa
(8)	$H_1 = (H_0 + e * H_0)_{rn}$	82-bit register format precision
(9)	$d_1 = (a - S_1 \cdot S_1)_{rn}$	82-bit register format precision
(10)	$S = (S_1 + d_1 \cdot H_1)_{rnd}$	single precision

Предсказание ветвлений в Itanium2

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



  - Таблица на 12K 4-битных историй
- Pattern History Table
  - Таблица на 16K 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

```
.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 ;;
}
```

Синтаксис инструкций:

(qp) ops[.comp$_1$] $r_1 = r_2, r_3$

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
    jump EQ
    y=3
    jump END
EQ: y=4
END:
```

Predicated Code

```
    cmp.eq p1,p2=a,b
p1  y=4
p2  y=3
```

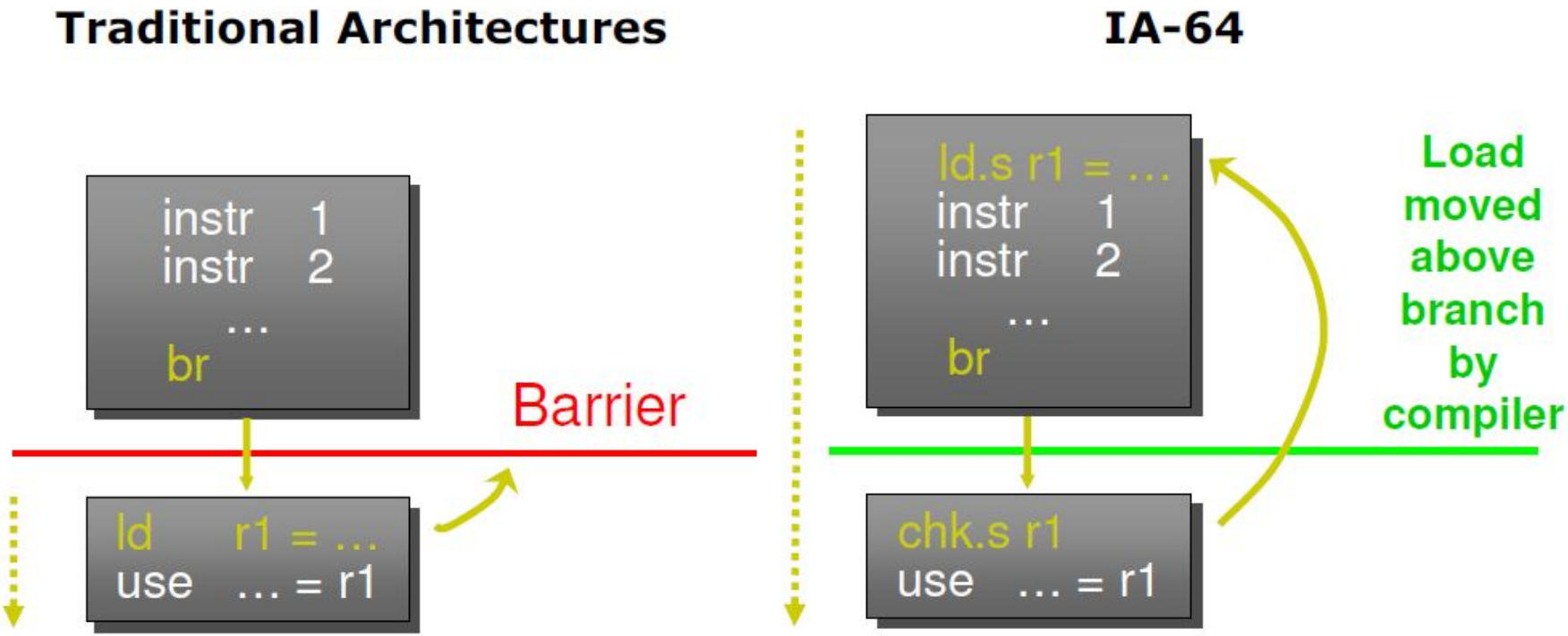
Аппаратные счетчики циклов

- Архитектурная поддержка циклов
  - По специальной команде перехода счетчики автоматически уменьшаются и делается проверка на выход из цикла
  - Можно не задействовать регистры общего назначения.
- Пример:

```
        mov ar.lc = 10 ;;
    Label:
        … тело цикла …
        br.cloop.sptk Label
```

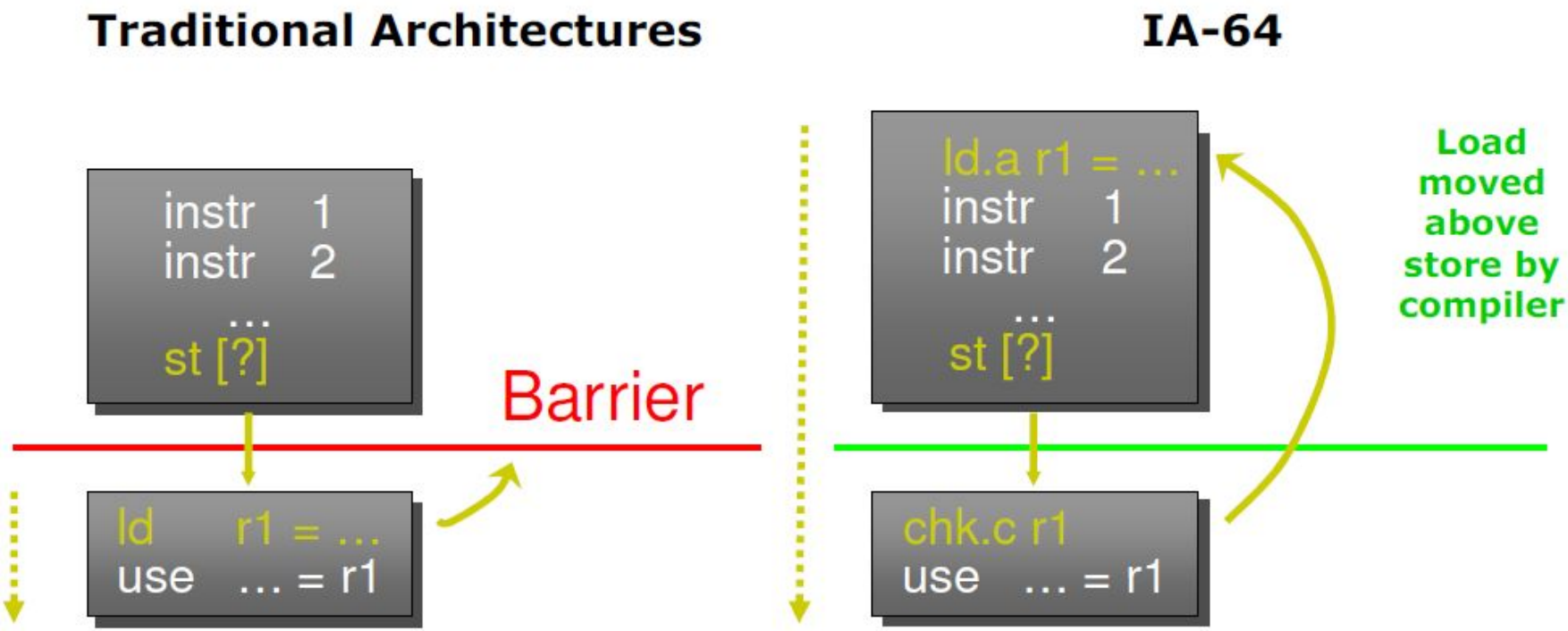
Спекуляция по управлению

- Команды загрузки могут выполняться до того, как обнаружится, что это действительно нужно



Спекуляция по данным

- Команды загрузки могут выполняться до того, как обнаружится, что это действительно можно



Программная конвейеризация цикла

- 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
```

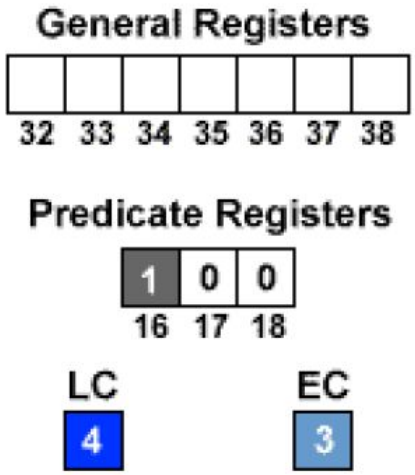
Pseudo Assembly Code

```
// 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.

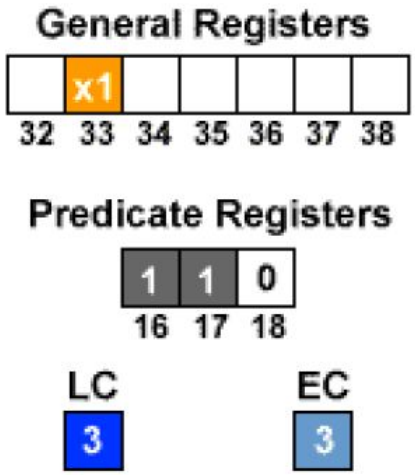
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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.

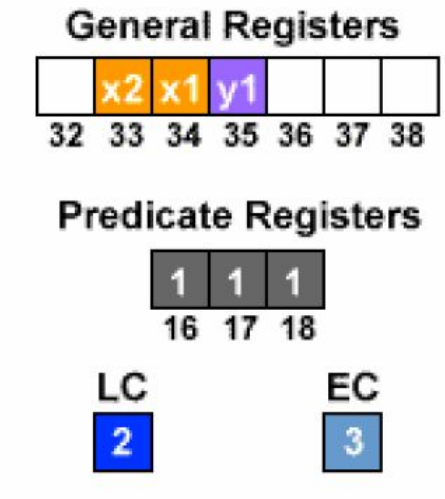
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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.

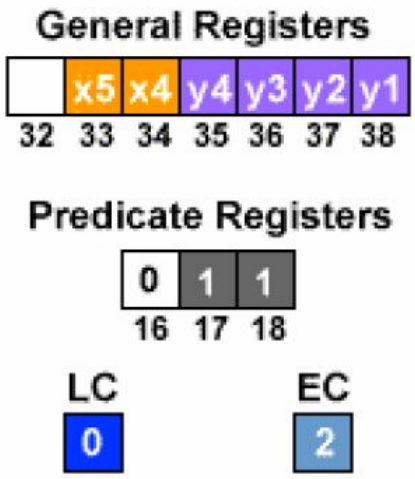
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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.

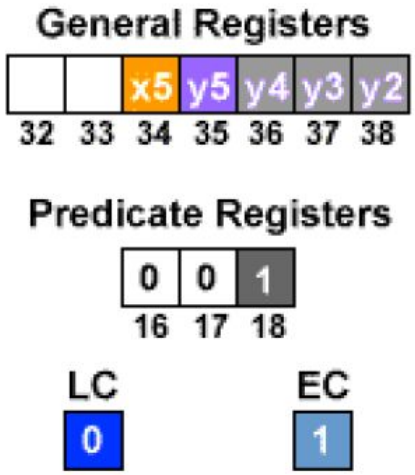
```
loop:
  (p16)   ld1 r32    = [r12],1
  (p17)   add r34    = 1,r33
  (p18)   st1 [r13] = r35,1
          br.ctop loop
```



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

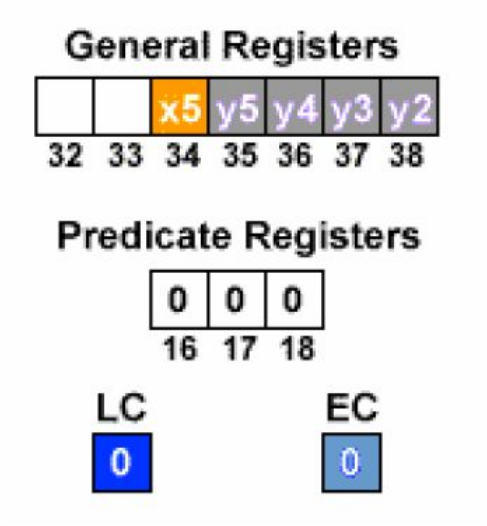
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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

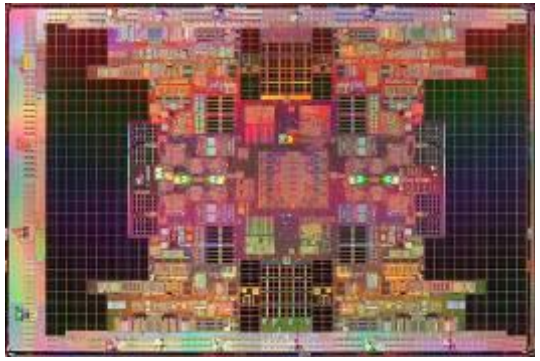


Процессоры Itanium 9300 (Tukwila)

Особенности нового Itanium-а

- Частота: до 1.73 GHz
- Режим Turbo boost: до 1.86 GHz
- 4 ядра
- Hyperthreading – 2 потока на ядро
- Интегрированный контроллер памяти
- Шина QPI – Quick Path Interconnect

Первый в мире процессор, содержащий более 2 млрд. транзисторов



Процессоры Transmeta

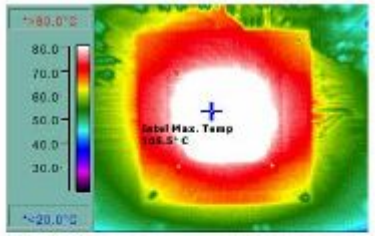
Процессоры Transmeta

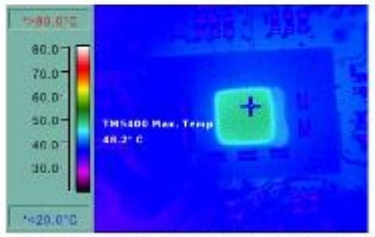
Особенности архитектуры

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

Процессоры

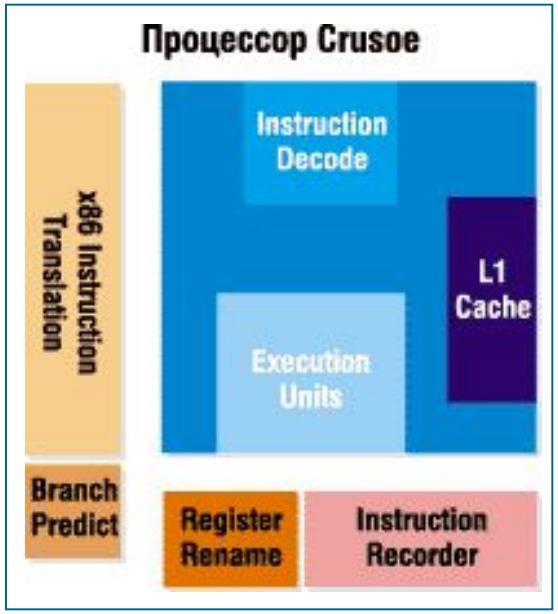
- Crusoe (2000) 1.0 GHz
- Efficion (2003) 1.7 GHz





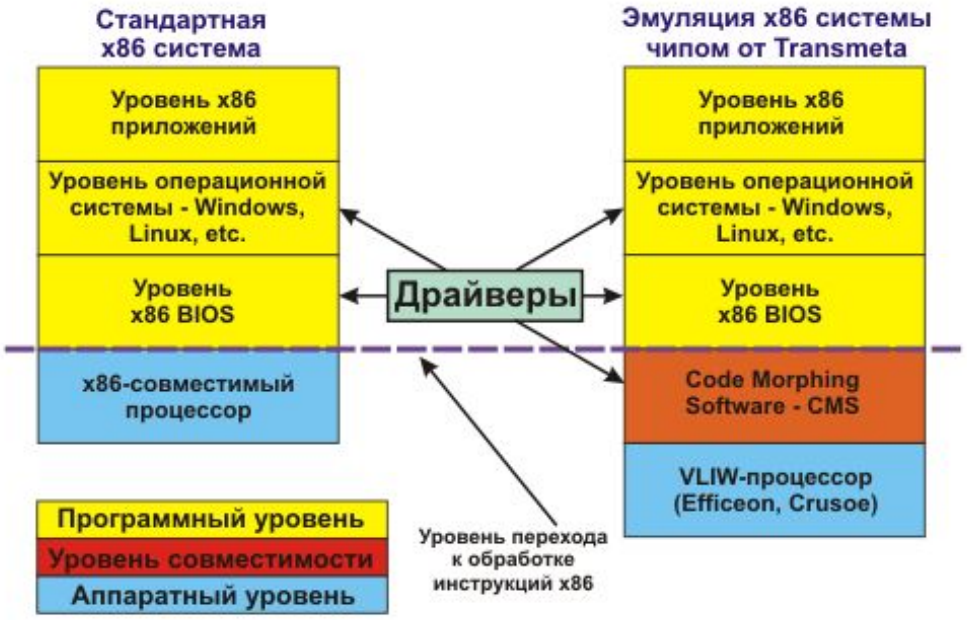
Динамическая двоичная компиляция

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



Динамическая двоичная компиляция

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

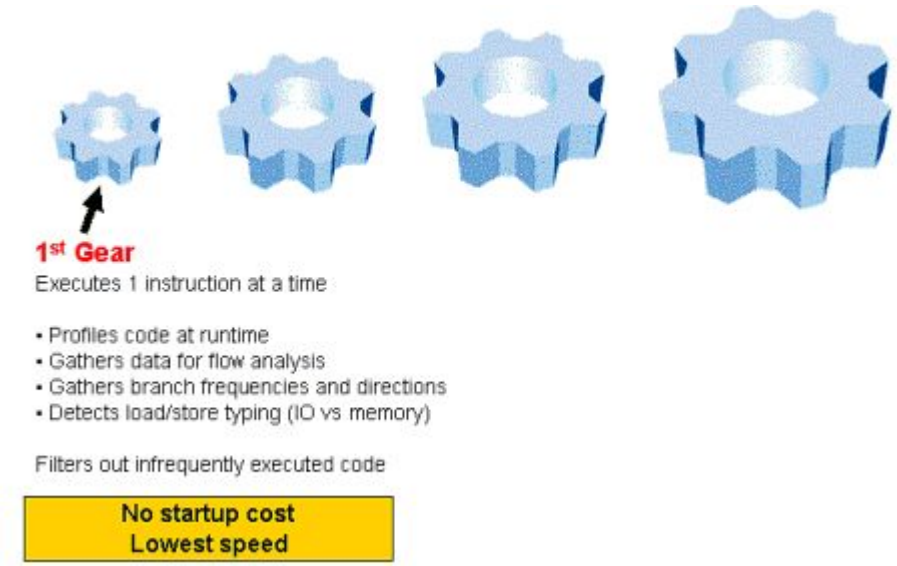


Простое изменение входной системы команд

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

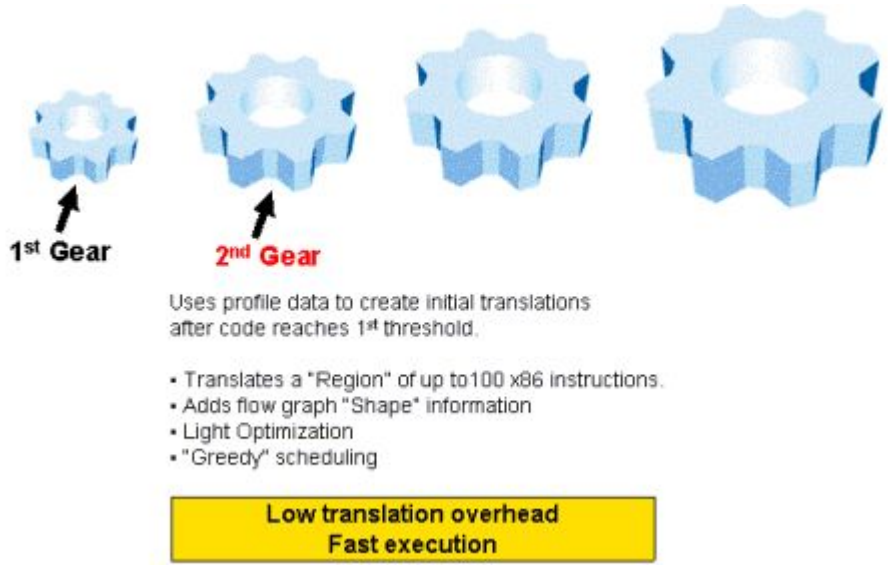
Динамическая двоичная компиляция

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



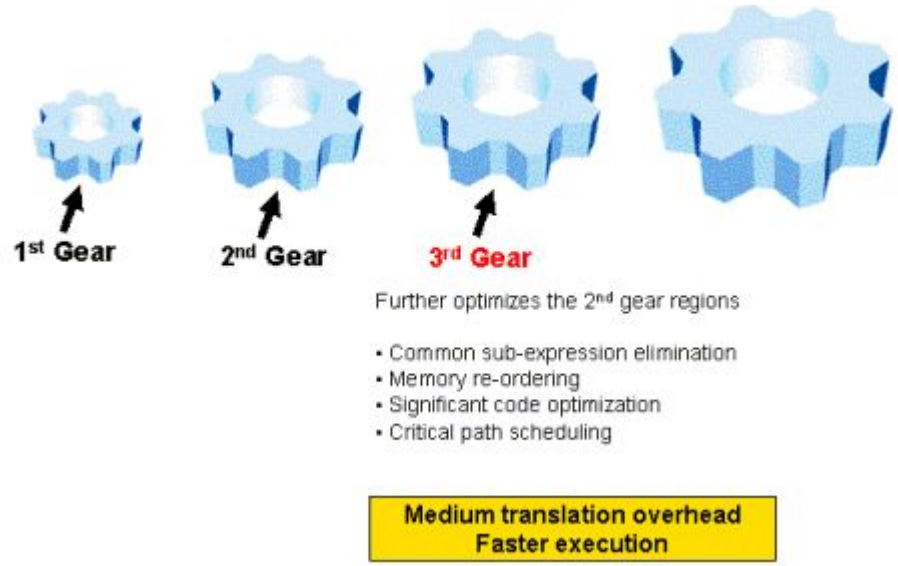
Динамическая двоичная компиляция

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



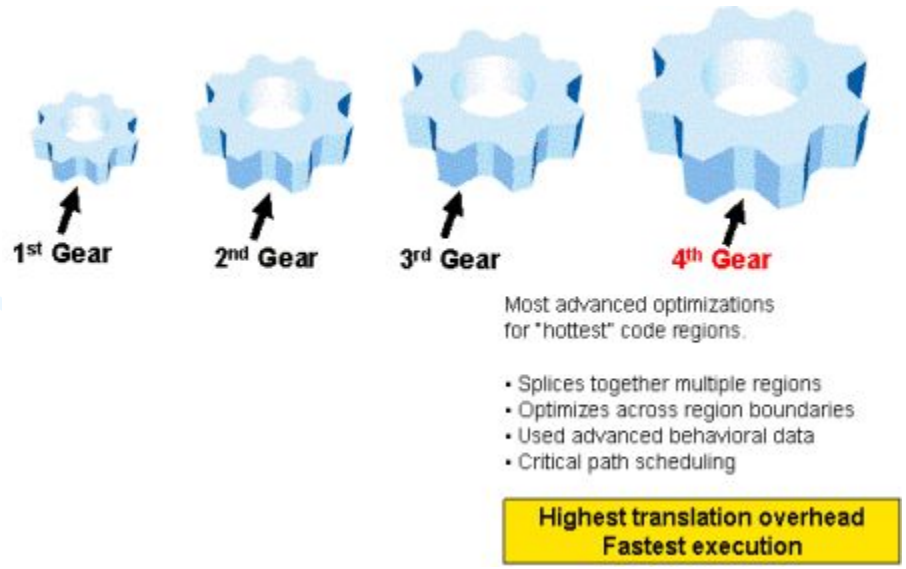
Динамическая двоичная компиляция

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



Динамическая двоичная компиляция

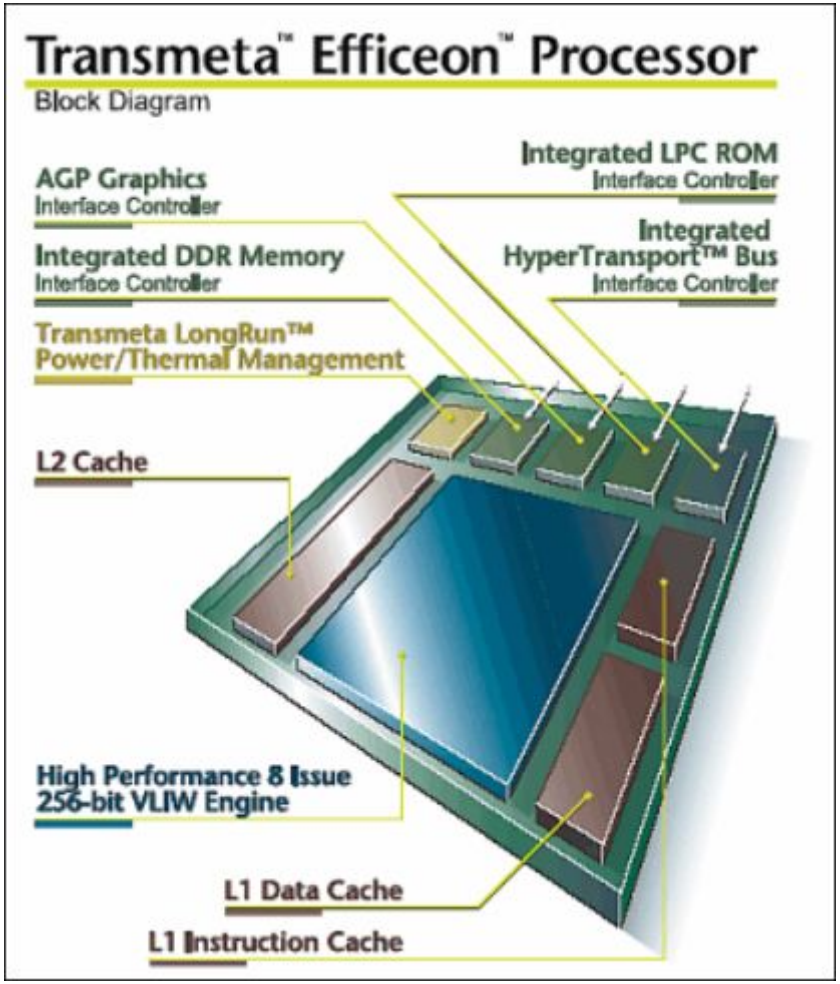
- Технология Code Morphing
  - Преобразование команд x86 в команды VLIW
  - Хранение транслированного кода в специальной области памяти (32 MB)
  - Динамическая оптимизация VLIW-кода



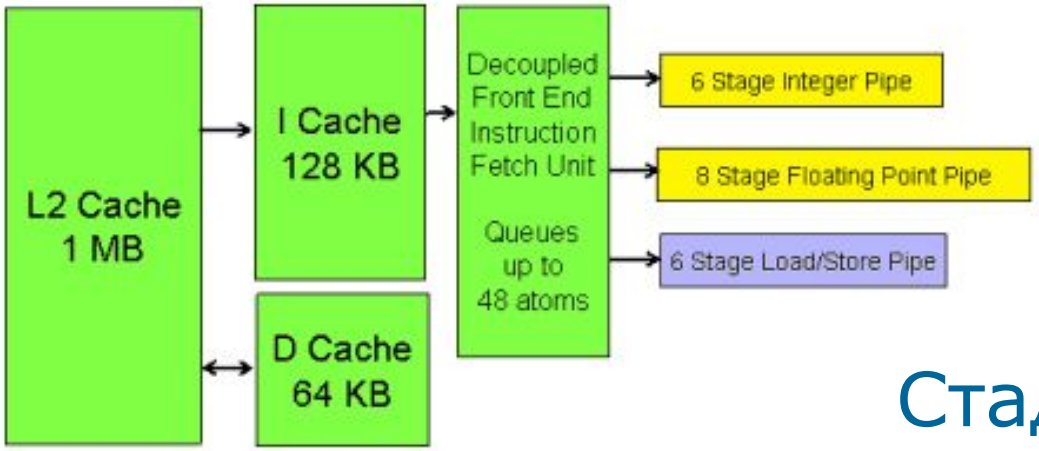
Процессор Transmeta Efficion

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

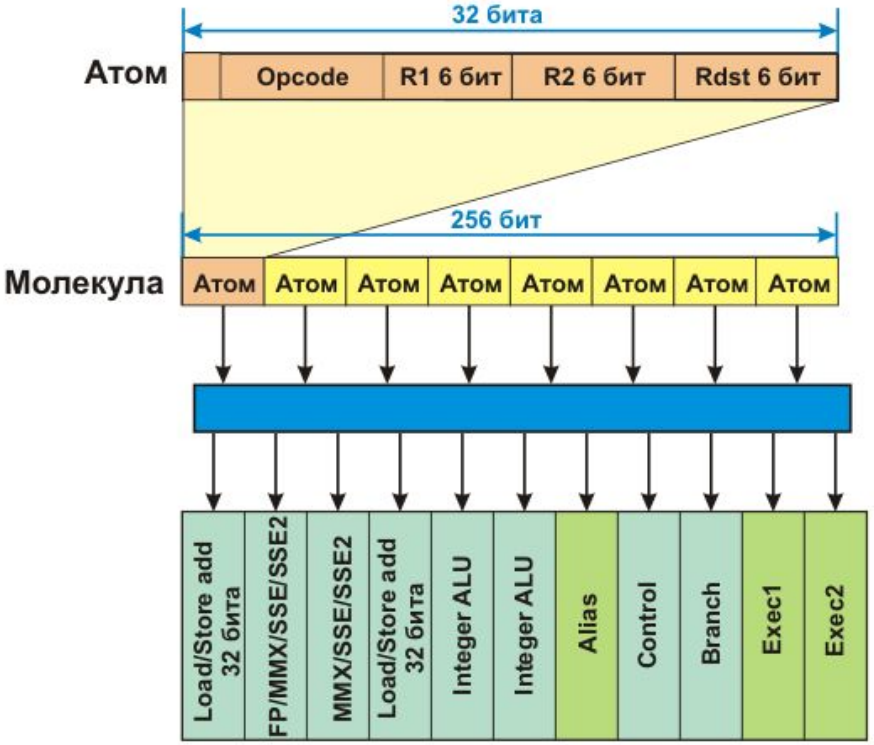
IS: Instruction Issue
DR: Instruction Decode
RM: Register Read for ALU operands
EM: Execute ALU operation
CM: ALU Condition flag completion
WB: Write Back results to integer register file

8-Stage Floating Point Pipe

IS: 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

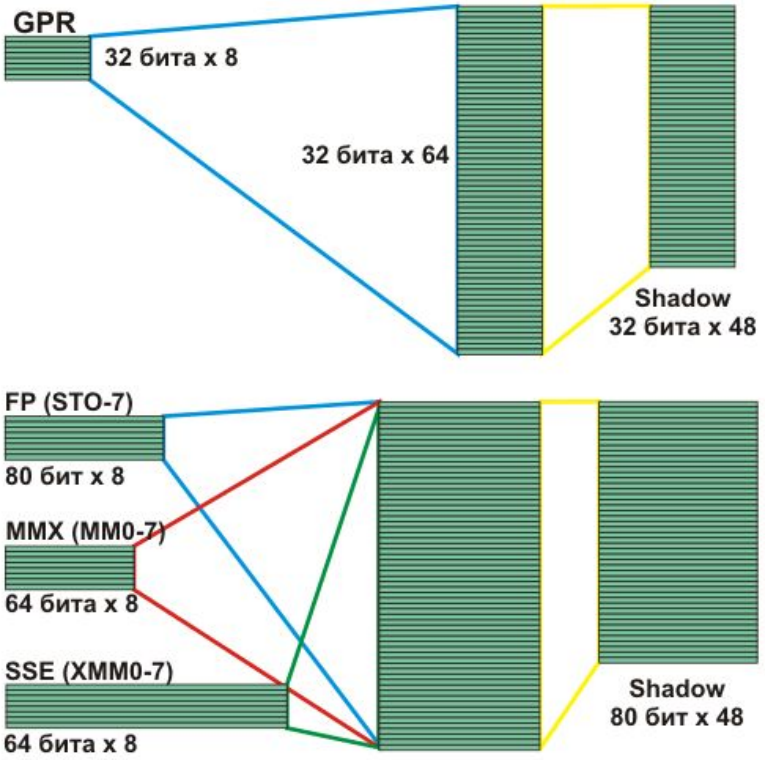
Процессор Transmeta Efficion

Структура команды



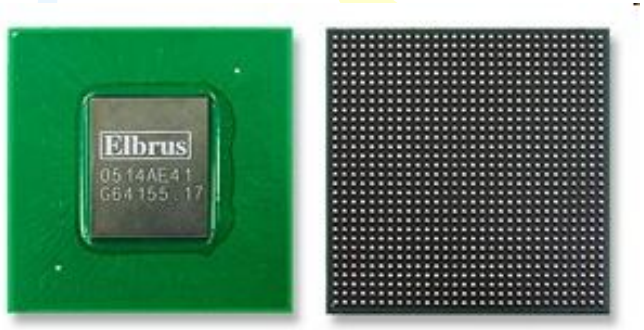
исполнительные устройства

Отображение регистров



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 E2K

- Архитектура VLIW переменной длины
- Двоичная трансляция кода: x86 ☐ VLIW
- Аппаратная поддержка типов данных

Реализации

- МВК Эльбрус 3 (1986-1994)
- Эльбрус 3М (2005) 300 MHz

Процессор Эльбрус

Характеристики

- Командное слово переменной длины (2 – 16 слогов)
- До 23 операций за такт
- Конвейер
  - Целочисленный:8 тактов
  - Чтение/запись: 9 тактов
- Двоичная трансляция команд
- Аппаратная поддержка типов данных
- Разрядность данных
  - Целые: 32, 64
  - Вещественные: 32, 64, 80
- Кэш-память
  - данных L1: 64 KB
  - кода L1: 64 KB
  - L2: 256 KB

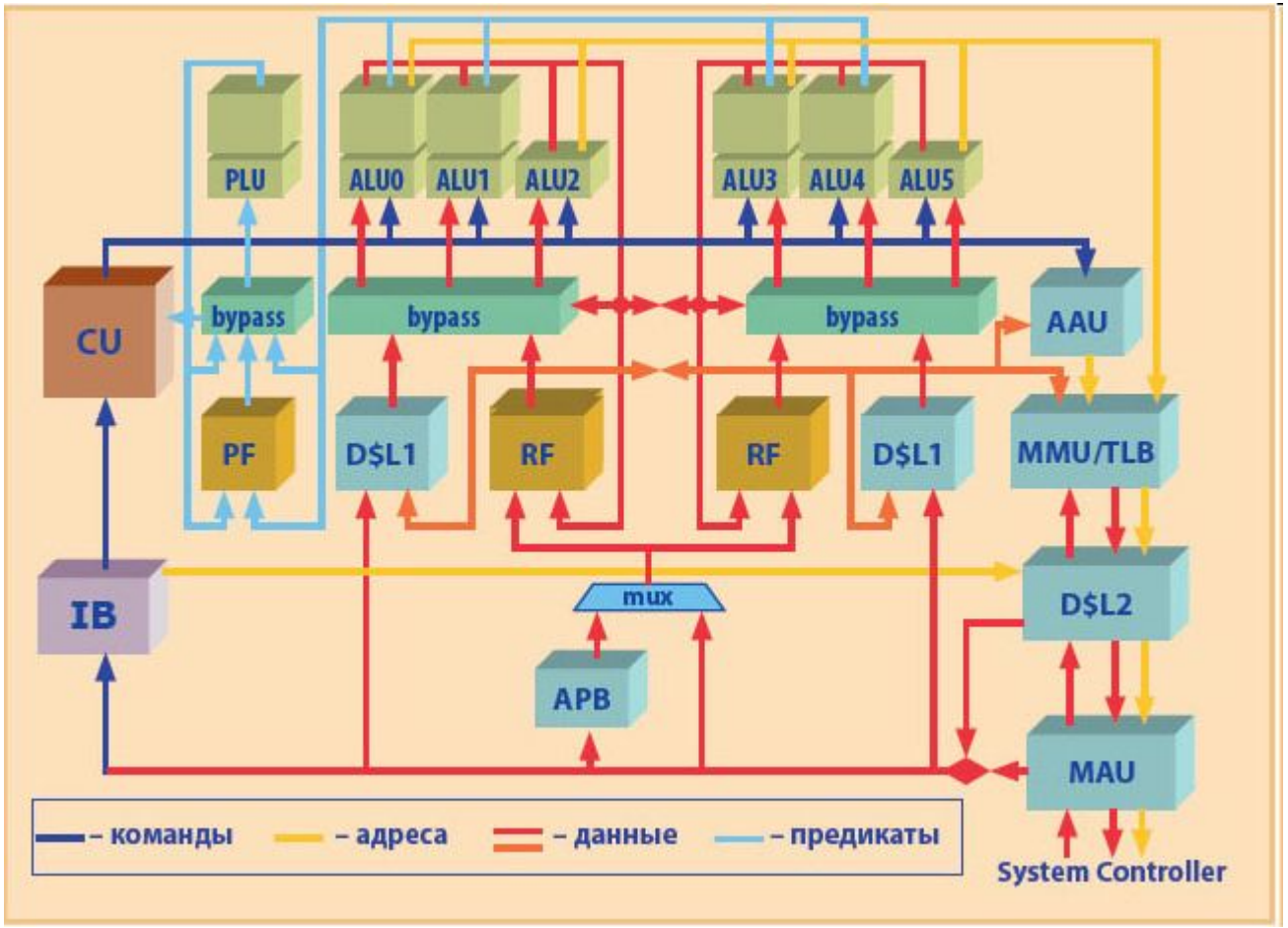
Процессор Эльбрус

Заголовок	Слог 1	Слог 2	…	Слог N

- Формат команды:
  - 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)

Процессор Эльбрус

- ALU0...ALU5 – арифметико-логические устройства;
- APU – устройство предварительной подкачки массивов;
- APB – буфер предварительной подкачки массивов;
- Bypass – обходные каналы;
- CU – устройство управления;
- PF – предикатный файл;
- IB – буфер команд;
- D$L1 – кэш данных 1-го уровня;
- D$L2 – кэш данных 2-го уровня;
- MAU – устройств организации доступа в оперативную память;
- MMU – устройство организации виртуальной памяти.



Процессор Эльбрус

- Динамическая трансляция кода