Архитектура современных микропроцессоров и мультипроцессоров

Лекция 10

Вопросы по предыдущей лекции

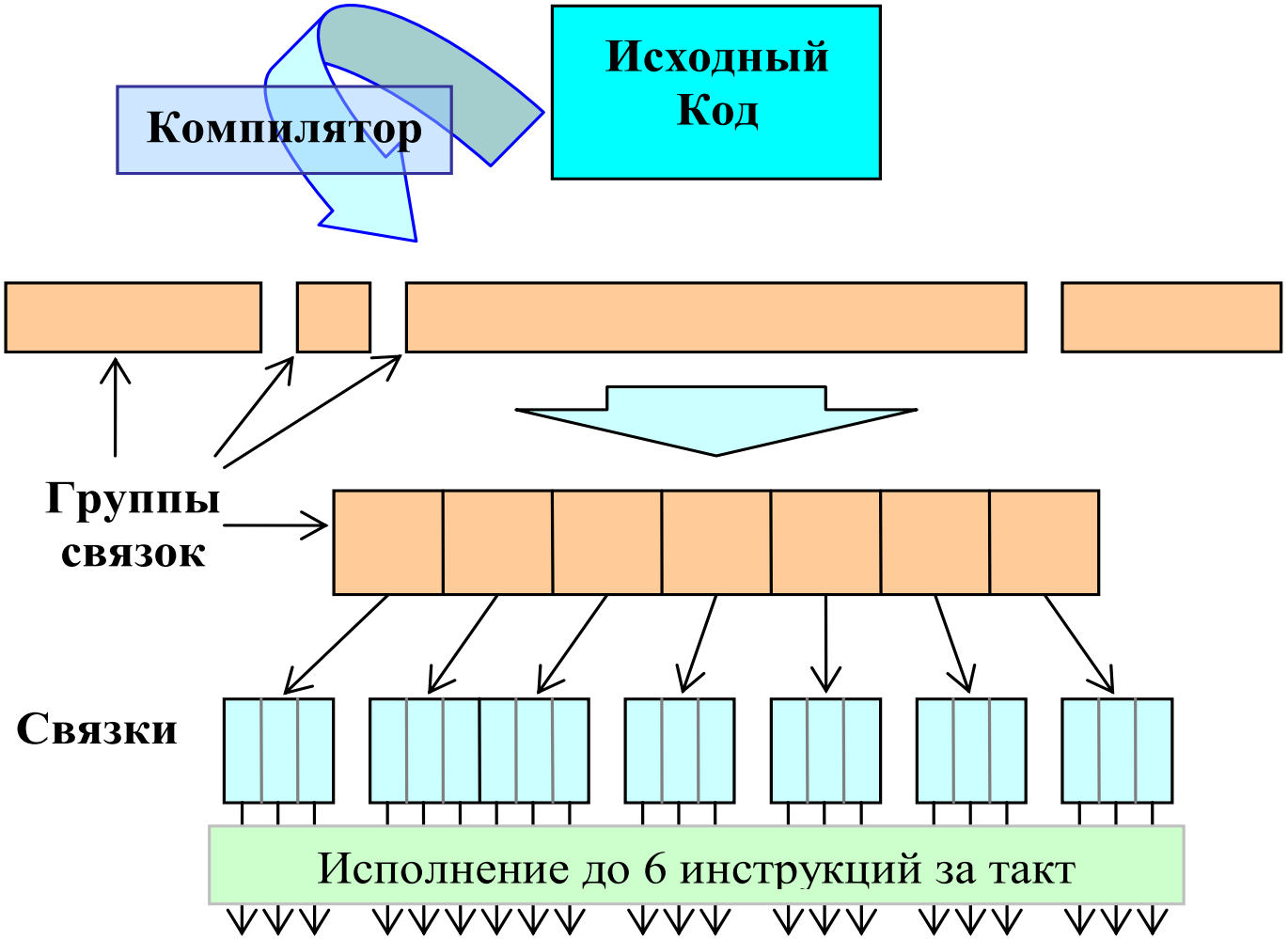
- Зачем нужна схема «тик-так»?
- Зачем понадобились tri-gate транзисторы взамен обычных?
- Какая подсистема занимает больше всего места на кристалле?
- Так ли нужна аппаратная поддержка (Processor Trace) для профилирования и отладки по сравнению с программными средствами?

EPIC: Explicitly Parallel Instruction Computing (IA 64)

Особенности IA 64 архитектуры

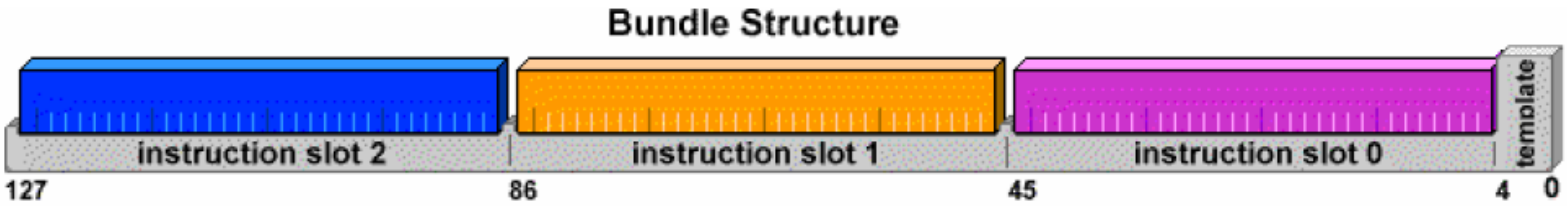
- Спекуляция кода.
- Спекуляция данных.
- Предсказания.
- Регистровый стек.
- Ветвления.
- Вращение регистров.
- Архитектура вычислений с плавающей точкой.

Группы и связки



Команды IA64

- Команды IA64 объединяются в связки по три **независимых** инструкции:

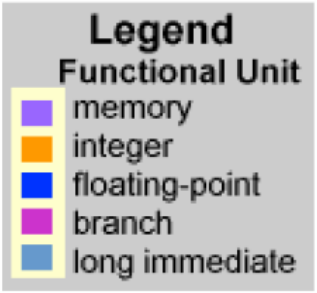


- Связка содержит 3 команды и поле шаблона.
- Шаблон указывает типы команд в связке. Он определяет, какие исполнительные устройства будут задействованы при исполнении.
- Типы команд:
  - **M** – memory / move
  - **I** – complex integer / multimedia
  - **A** – simple integer / logic / multimedia
  - **F** – floating point (normal / SIMD)
  - **B** – branch

Команды IA64

- 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



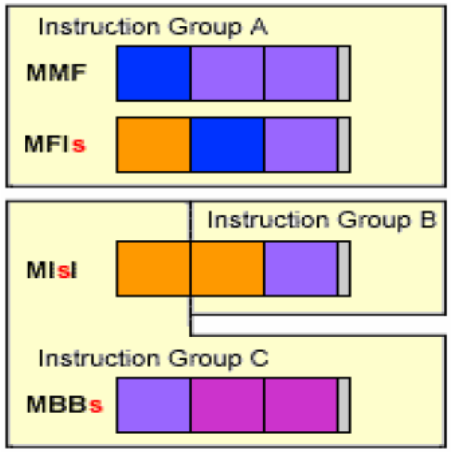
* L+X is an extended type that is dispatched to the I-unit.

- Процессор загружает 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



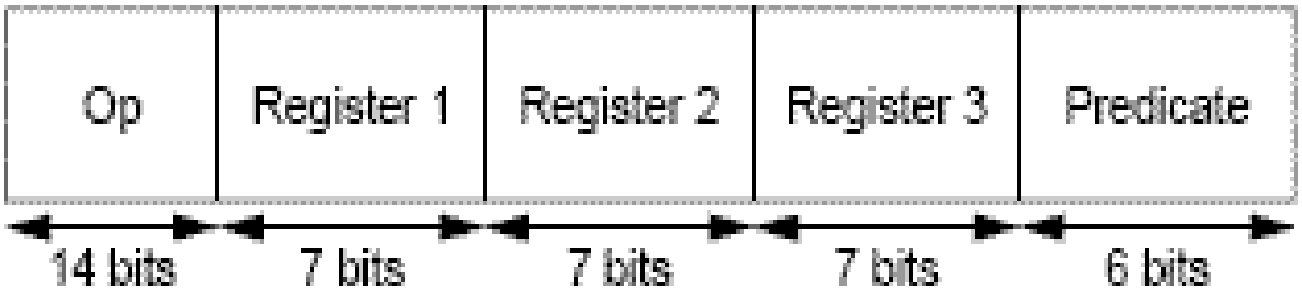
Система команд

Система команд IA 64 обладает основными чертами RISC-архитектуры.

- **Регулярная длина.**
- **Команды вида (reg3 = reg1 *op* reg2).**
- **Операции проводятся только над регистрами, для чтения/записи из/в память существует специальная группа команд.**
- **В архитектуре IA 64 нет команд с высокой латентностью, например: трансцендентных, деления, целочисленного умножения и т.п.**

Структура команды

- **Команды (41 бит) упаковываются в связки (bundle) по 128 бит.**
- **Старшие 5 бит связки – шаблон. Шаблон указывает к каким ИУ направляется каждая из команд и границы разделения групп команд, которые не имеют зависимостей.**



Типы команд IA64

- Логические (and, …)
- Арифметические (add, …)
- Команды сравнения
- Команды сдвига
- Мультимедиа (целочисленные SIMD)
- Команды ветвлений (перехода)
- Команды ветвлений, управляющие циклом
- Вещественные (fma, …)
- SIMD вещественные (fpma, …)
- Команды чтения / записи данных в памяти
- Команды присваивания
- Команды управления кэшированием

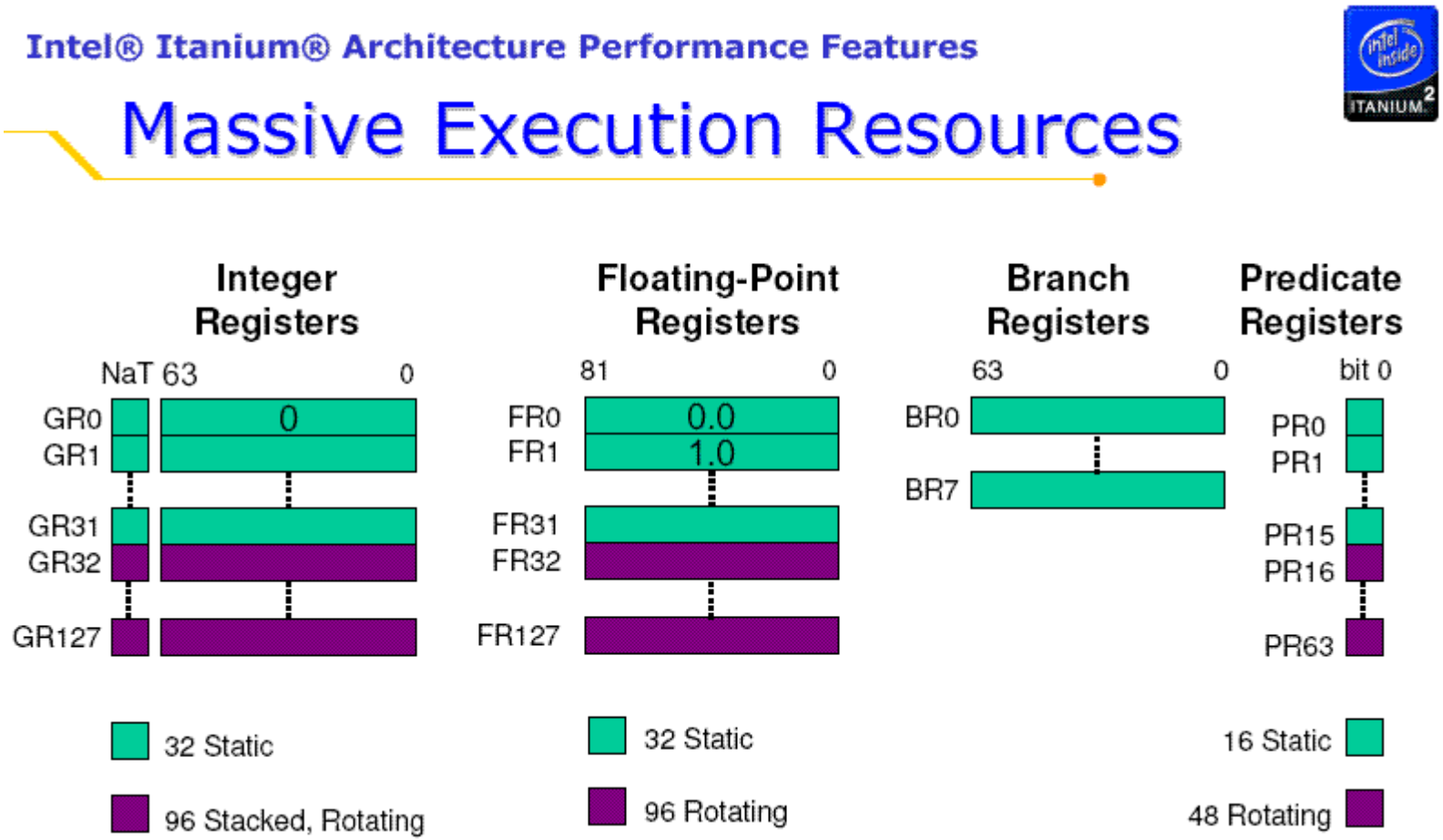
Особенности вещественной арифметики в Itanium2

- Максимальная производительность
  - 2 за такт: двойная точность
  - 4 за такт: одинарная точность (SIMD)
- Основная операция
  - fma: f = a * b + c (4 такта)
  - Используется и для целочисленного умножения
- 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

Регистры

- **128** 64-битных регистров общего назначения GR0-GR127.
- **128** 82-битных вещественных регистров FR0-FR127.
- **64** 1-битных предикатных регистра PR0-PR63.
- **8** 64-битных регистров ветвлений BR0-BR7.
- специальные архитектурные регистры, среди которых есть регистры архитектурной поддержки циклов и вызовов функций.

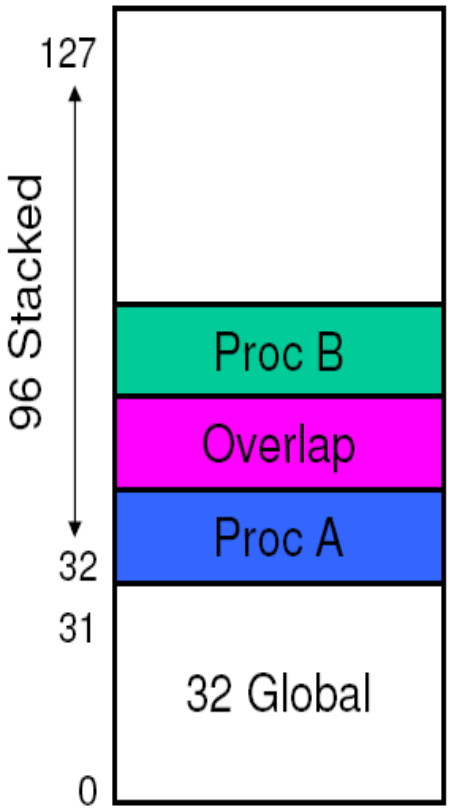
Регистры


Intel® Itanium® Architecture Performance Features
Massive Execution Resources
Integer Registers
NaT 63
0
GR0
GR1
0
GR31
GR32
GR127
32 Static
96 Stacked, Rotating
Floating-Point Registers
81
0
FR0
FR1
0.0
1.0
FR31
FR32
FR127
32 Static
96 Rotating
Branch Registers
63
0
BR0
BR7
Predicate Registers
bit 0
PR0
PR1
PR15
PR16
PR63
16 Static
48 Rotating

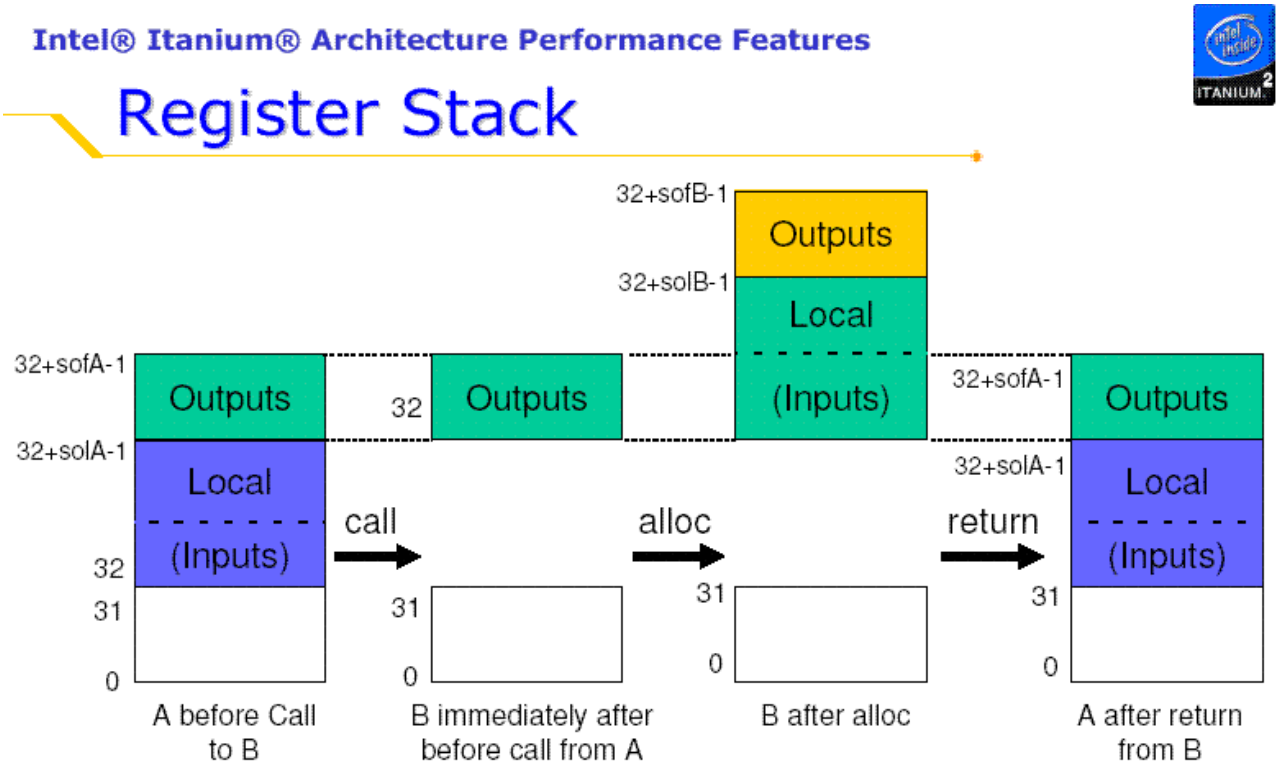
Регистровый стек

- **механизм, помогающий избежать ненужных сохранений регистров при вызове функций.**
- **регистры GR32- GR128 - стековые.**
- **они динамически переименовываются при входе в функцию и выделении регистрового фрейма стека из внутреннего регистрового файла.**

- **при переходах между подпрограммами можно избегать загрузки значений локальных переменных и параметров из памяти в регистры**



Регистровый стек



Регистровый стек

- автоматически сохраняет/восстанавливает регистры без вмешательства программы
- обеспечивает иллюзию бесконечности числа физических регистров
- достигается путем отображения стека физических регистров в память
- RSE может быть настроен на использование свободной части канала памяти для сохранения/восстановления регистров в фоновом режиме

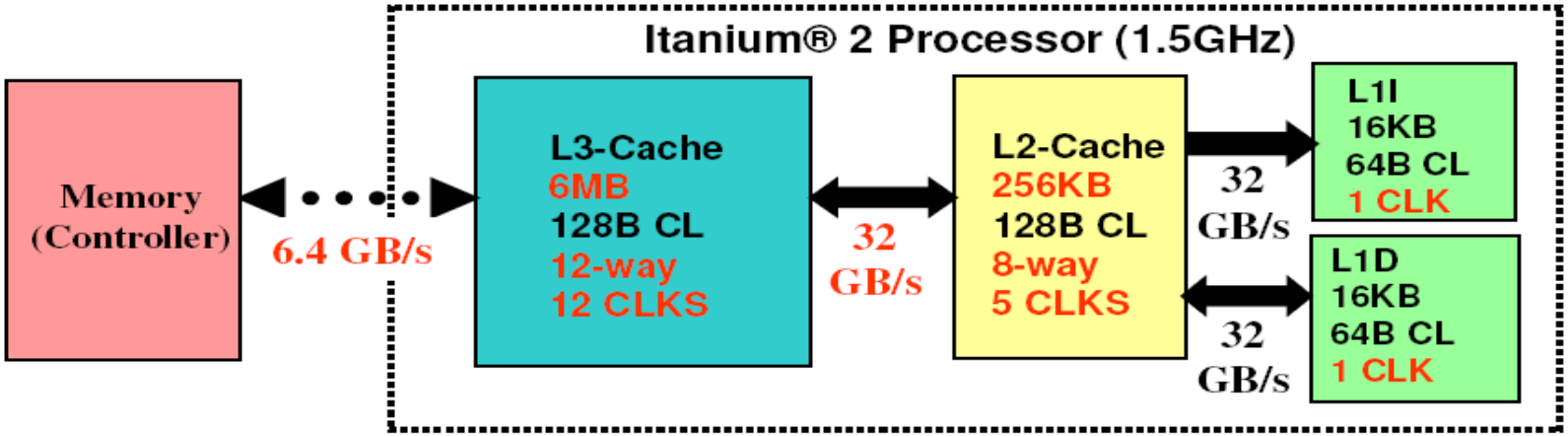
Память

- **IA-64 определяет единственное однородное, адресное пространство размером 2^{64} байт.**

– **единственное означает, что код и данные находятся в одной памяти.**

– **однородное означает, что память не разделена на участки с предопределенной функциональностью.**

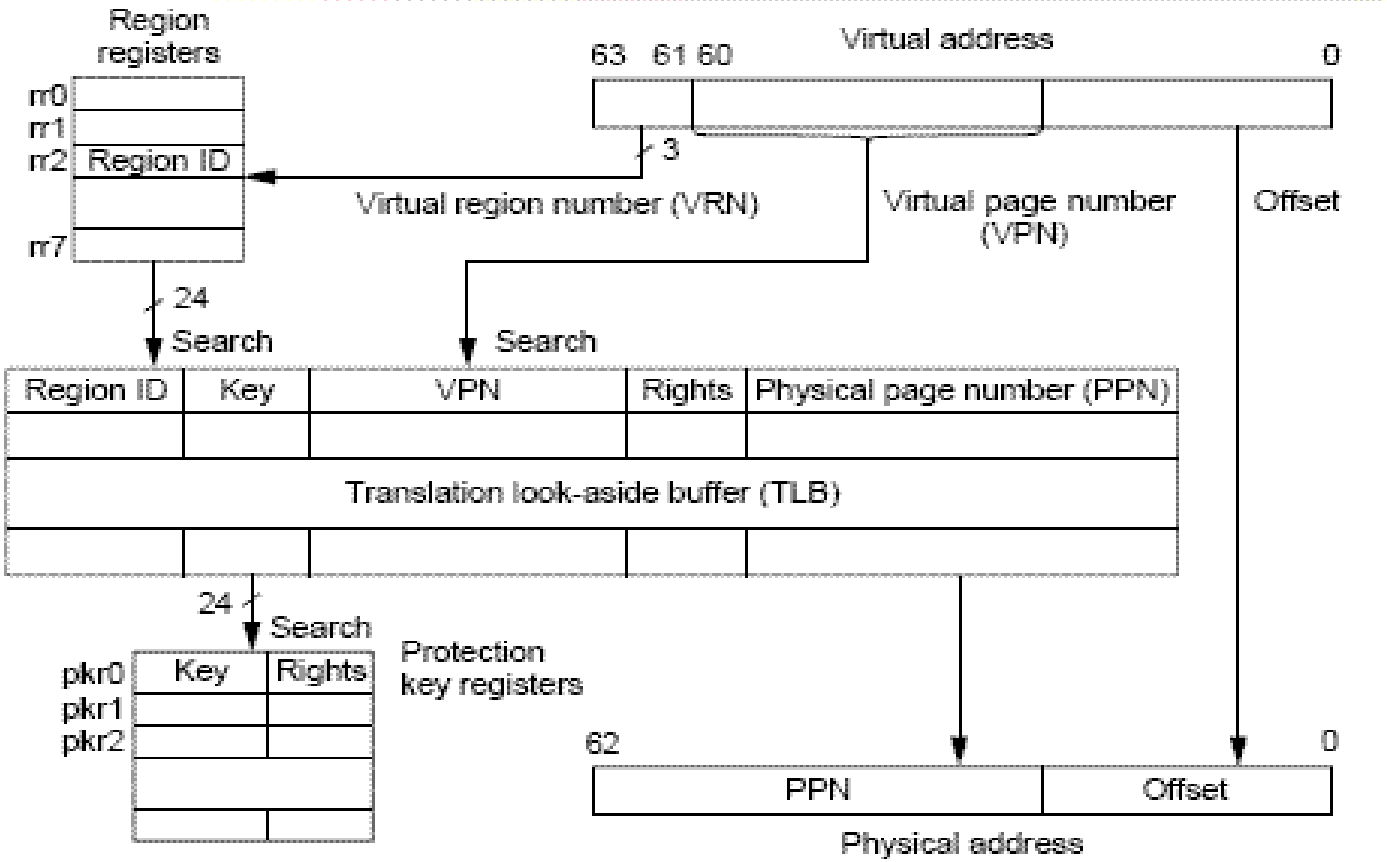
Иерархия кэш-памяти Itanium2



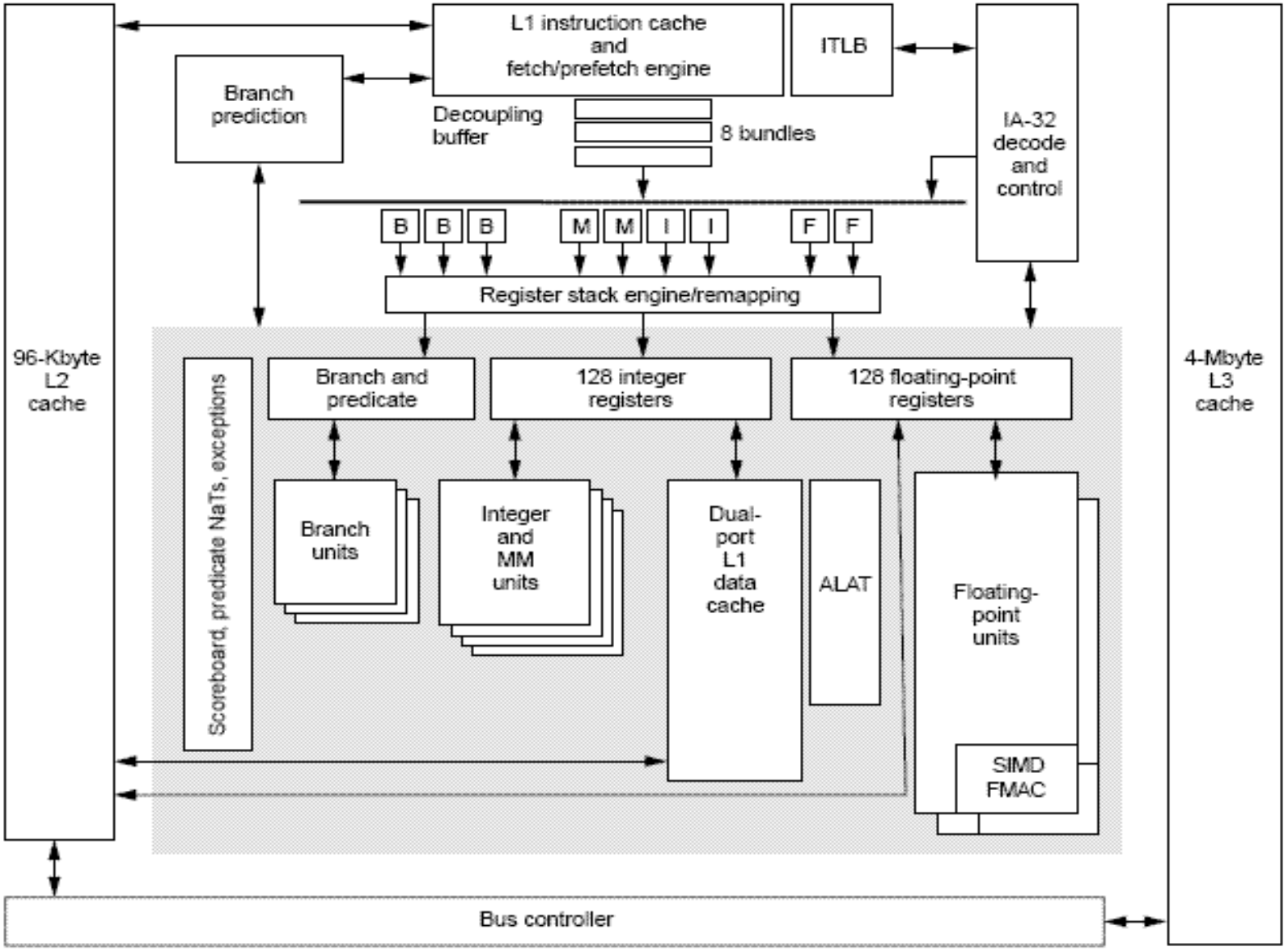
	L1I	L1D	L2	L3
Size	16K	16K	**256K**	**12M on die**
Line Size	64B	64B	128B	128B
Ways	4	4	**8**	**12**
Replacement	LRU	NRU	NRU	NRU
Latency (load to use)	**I-Fetch:1**	**INT:1**	**INT: 5 FP: 6**	**INT: 12 FP: 13**
Write Policy	-	WT (RA)	WB (WA + RA)	WB (WA)
Bandwidth	**R: 32 GBs**	**R: 16 GBs W: 16 GBs**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

Виртуальная память

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

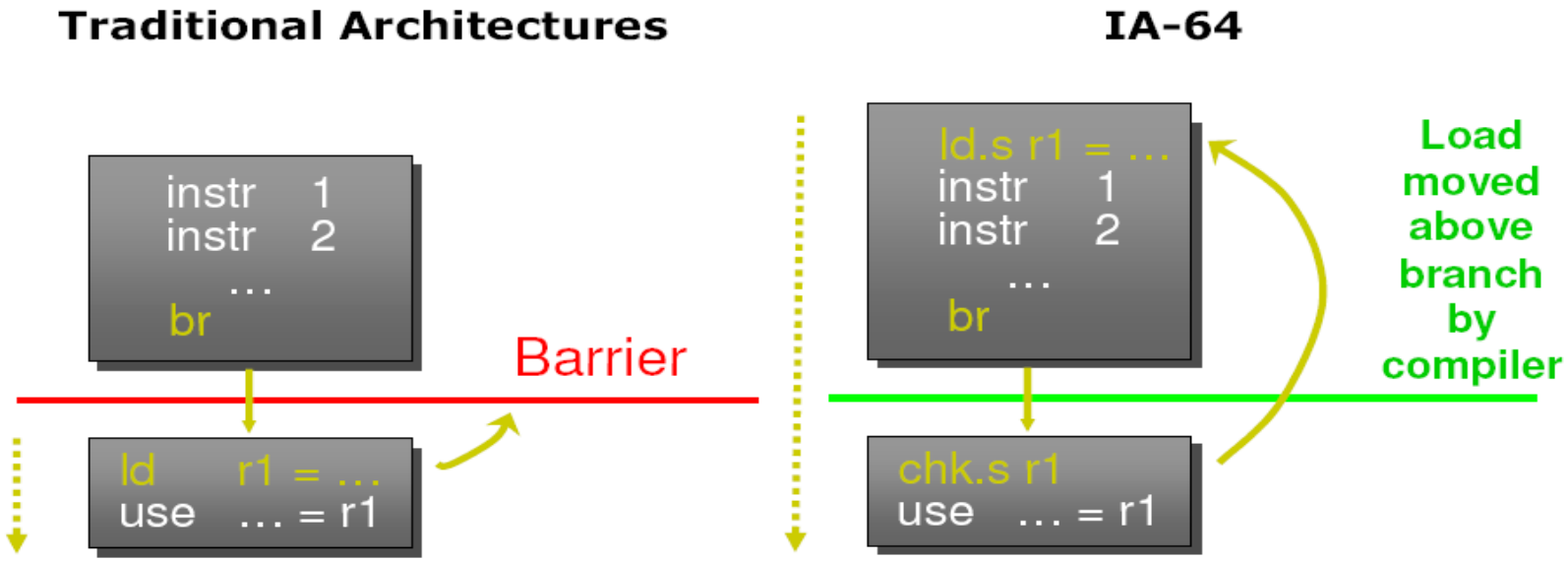


Структура процессора Itanium



Спекулятивное исполнение по коду

Команда может быть выполнена задолго до её оригинального месторасположения (используется для разрешения проблемы с задержкой памяти).



Ветвление и предикаты

Различные ветви программы могут выполняться параллельно под предикатом (это позволяет перевести зависимость по управлению в зависимость по данным).

Unpredicated code

```
      cmp   a, b
      je    EQ
      y = 3
      jump  END
EQ: y = 4
END:
```

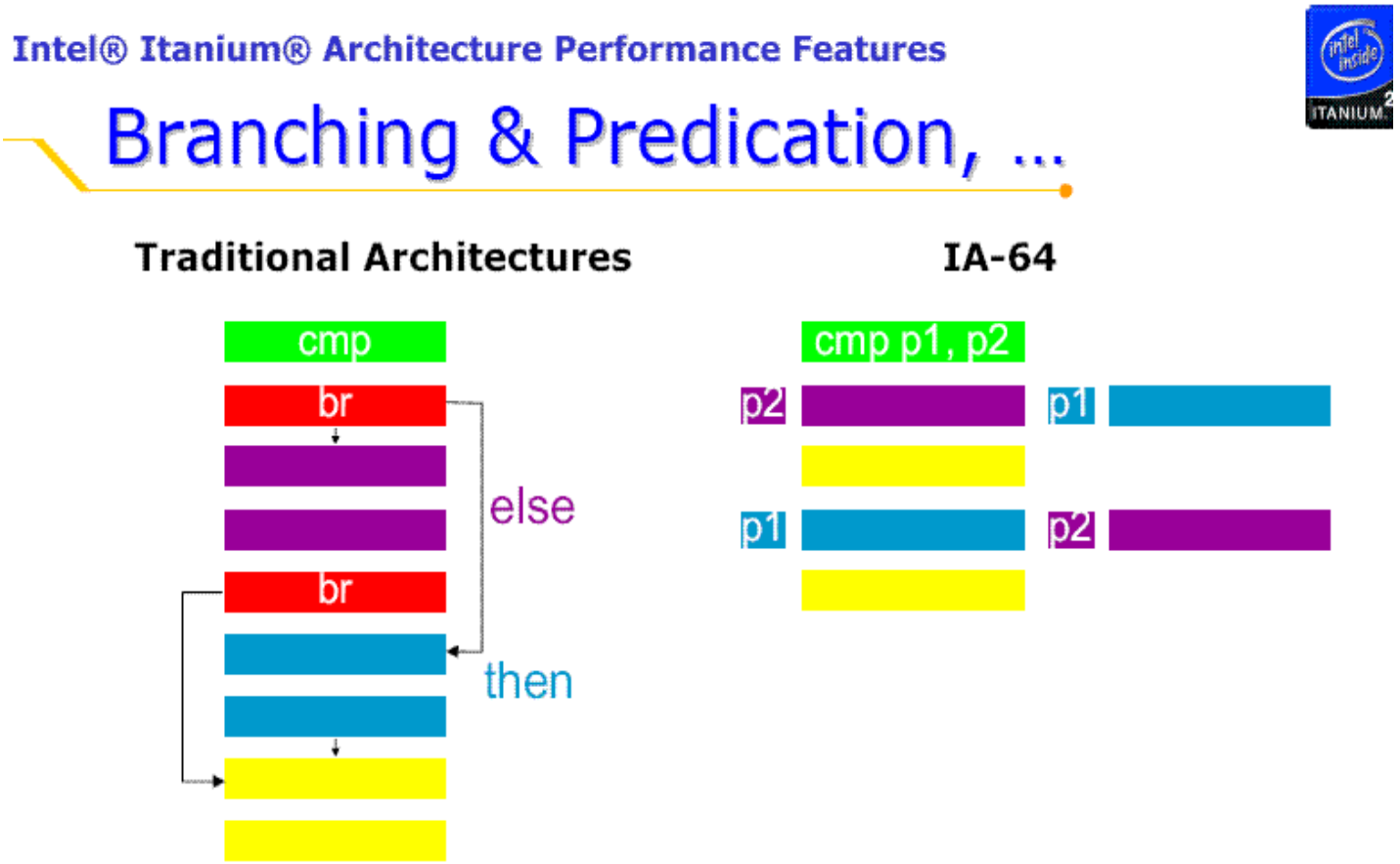
Predicated code

```
      cmp.eq   p1, p2=a, b
P1    y = 3
P2    y = 4
```

Предсказание

Intel® Itanium® Architecture Performance Features

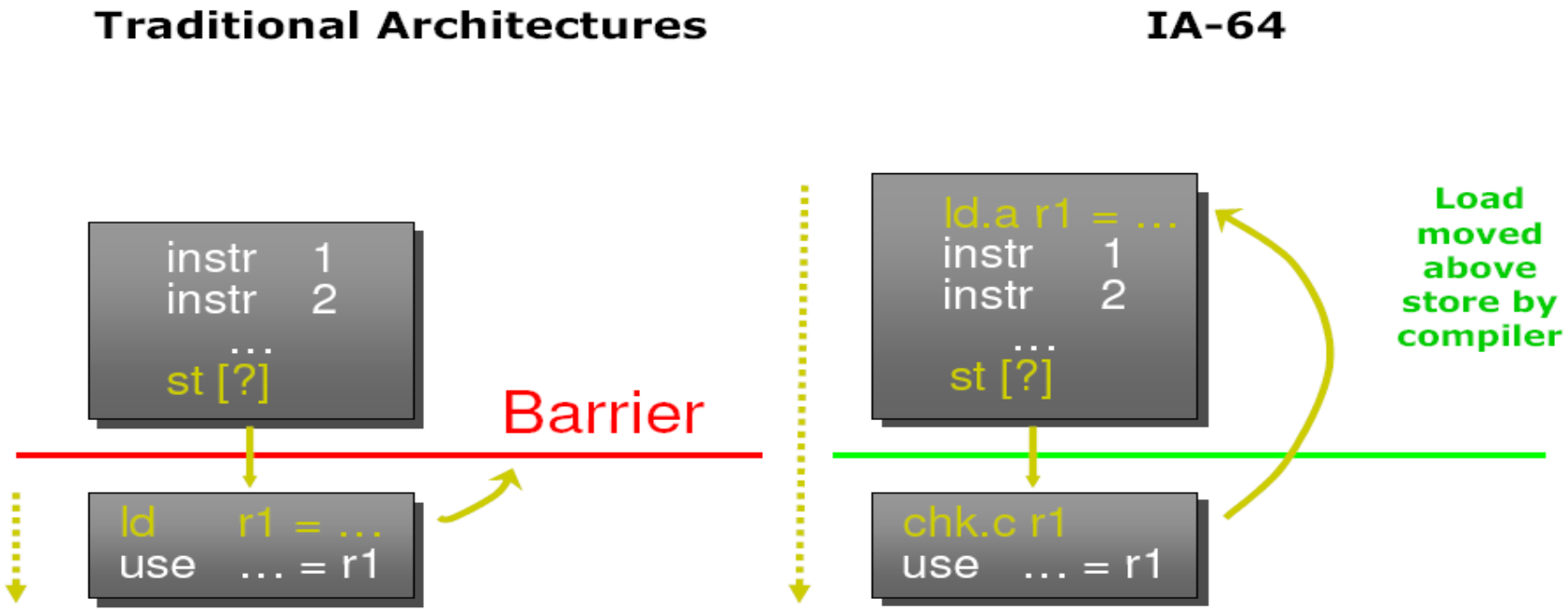
Branching & Predication, …



Control flow to Data flow

Спекулятивное исполнение по данным

Заключается в смене порядка обращений к памяти (используется для разрешения зависимостей по данным).



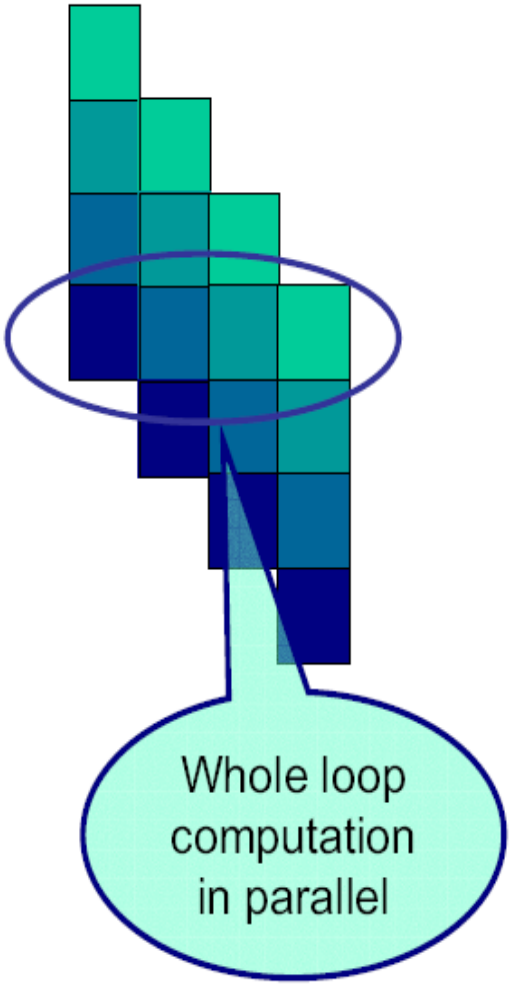
Программная конвейеризация циклов

Программная конвейеризация циклов обеспечивается за счет использования:

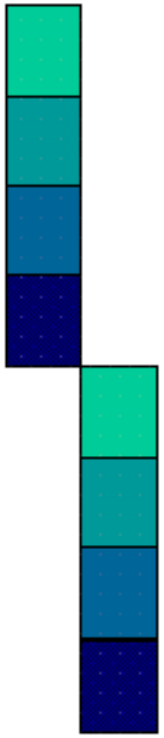
- **специальных инструкций ветвления**

-- **вращения регистров**

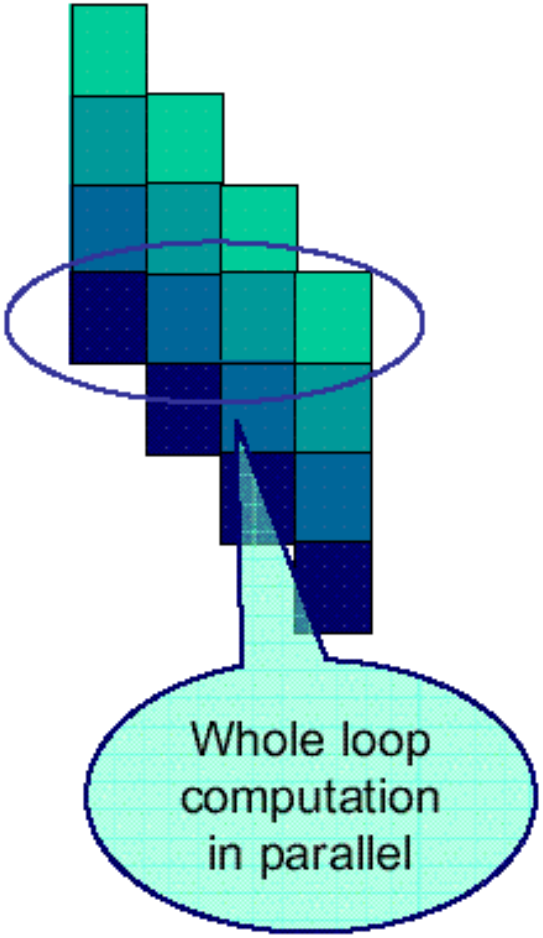
- **архитектурных регистров**
  loop counter (LC)
  epilogue counter (EC)



Software Pipelining Support, …



- High performance loops without code size overhead
- No prologue/epilogue
  - Register rotation (rrb)
  - Predication
  - Loop control registers (LC, EC)
  - Loop branches (br.ctop, br.wtop)
- Especially valuable for integer loops with small trip counts

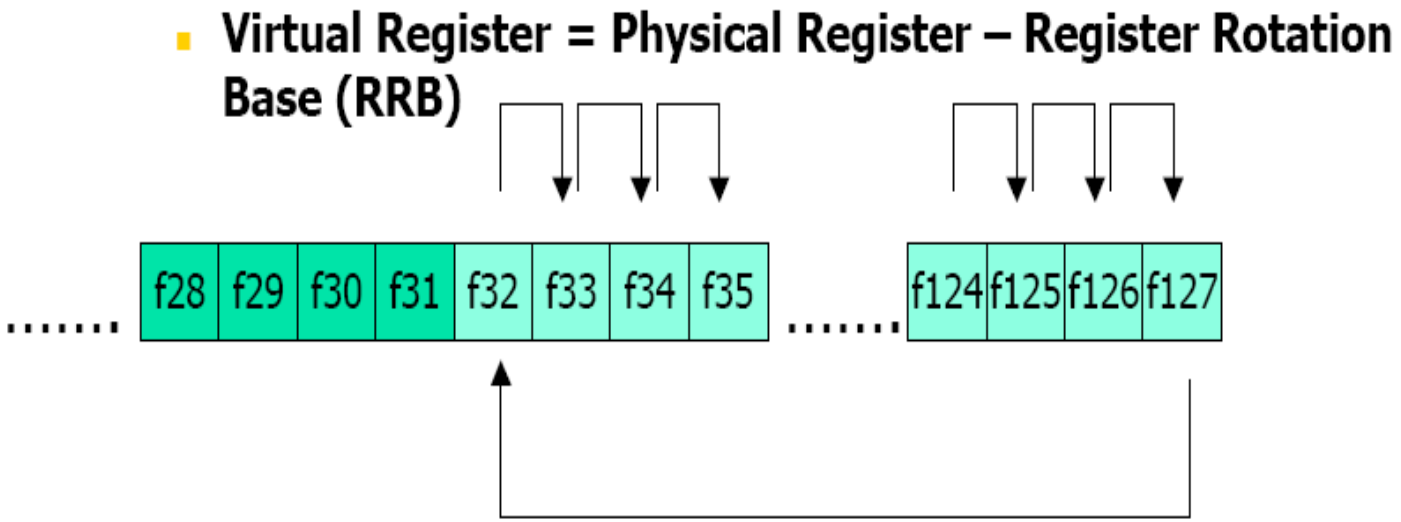


Программная конвейеризация

- **Программная конвейеризация циклов** – это оптимизированный способ выполнения цикла. Если итерации в цикле могут выполняться независимо, то за счет имеющихся ФУ они могут исполняться параллельно. Как только первая команда первой итерации выполнилась, начинает выполняться вторая команда первой итерации и первая команда второй итерации. Конвейер наполняется (**стадия пролога**), и затем с конвейера за один такт сходит по одной итерации цикла (**стадии ядра и эпилога**). На стадии эпилога конвейер освобождается.
- Предикатные регистры (начиная с p16) контролируют, какие из команд должны быть выполнены, а какие – нет (на стадиях пролога и эпилога). Два счетчика: **LC (loop counter)** и **EC (epilogue counter)** определяют количество итераций и количество команд в одной итерации.

Вращение регистров

- **Зависимость между итерациями устраняется за счет вращения (переименования) регистров.**
- **Верхние 75% регистров вращающиеся**:
- **При выполнении специальной команды перехода (в цикле) вращающиеся регистры сдвигаются вправо на один:**



Пример

- **mov pr.rot = 0** // очистка предикатных регистров вращения
- **cmp.eq p16, p0 = r0, r0** // установить p16=1
- **mov ar.lc = 4** // Счетчик цикла LC=n-1
- **mov ar.ec = 3** // Счетчик эпилога EC=3

Loop:

- **(p16) ld1 r32 = [r12], 1;;** // #1 загрузить X
- **(p17) add r34 = 1, r33;;** // #2 Y = X+1
- **(p18) st1 [r13] = r35, 1;;** // #3 сохранить Y
- **br.ctop.sptk.few Loop**

Пример

Аналогичный обычный цикл

Loop:

- **ld1 r32=[r12], 1 // загрузить X**
- **add r33=1, r32 // Y = X+1**
- **st1 [r13]=r33, 1 // сохранить Y**
- **br Loop**

Обычное выполнение

	r32	r33	i
			5
1	X1		5
2	X1	Y1	5
3	X1	Y1	4
4	X2	Y1	4
5	X2	Y2	4
6	X2	Y2	3
7	X3	Y2	3
8	X3	Y3	3
9	X3	Y3	2
10	X4	Y3	2
11	X4	Y4	2
12	X4	Y4	1
13	X5	Y4	1
14	X5	Y5	1
15	X5	Y5	0

Конвейеризация цикла

	r32	r33	r34	r35	r36	r37	r38	p16	p17	p18	LC	EC
								1	0	0	4	3
1		X1						1	1	0	3	3
2		X2	X1	Y1				1	1	1	2	3
3		X3	X2	Y2	Y1			1	1	1	1	3
4		X4	X3	Y3	Y2	Y1		1	1	1	0	3
5		X5	X4	Y4	Y3	Y2	Y1	0	1	1	0	2
6			X5	Y5	Y4	Y3	Y2	0	0	1	0	1
7				X5	Y5	Y4	Y3	0	0	0	0	0

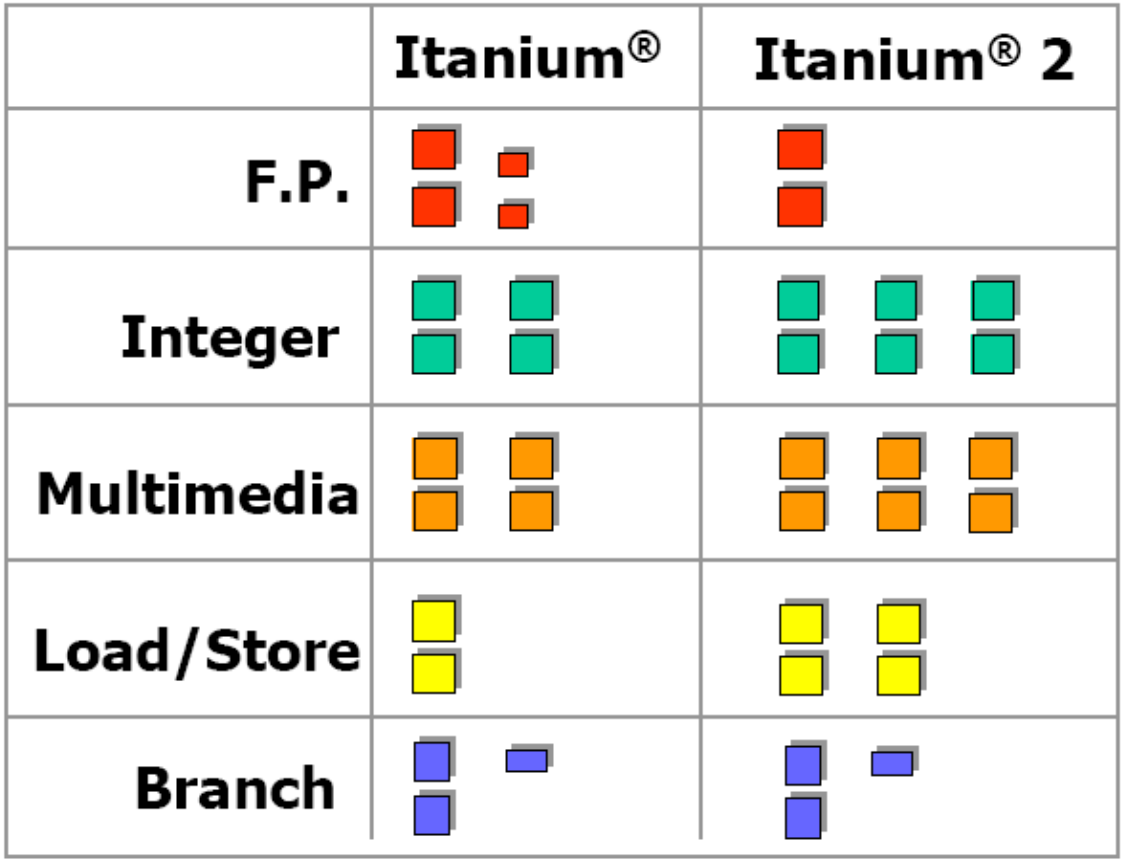
Загрузка

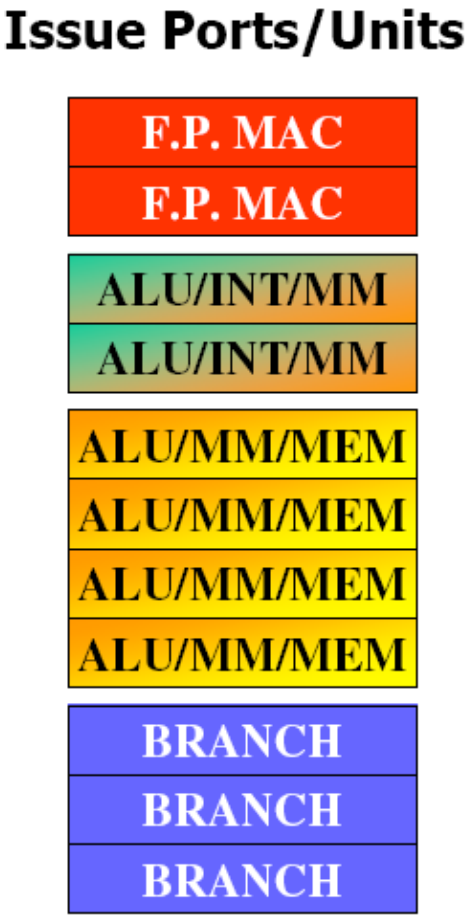
Вычисление

Сохранение

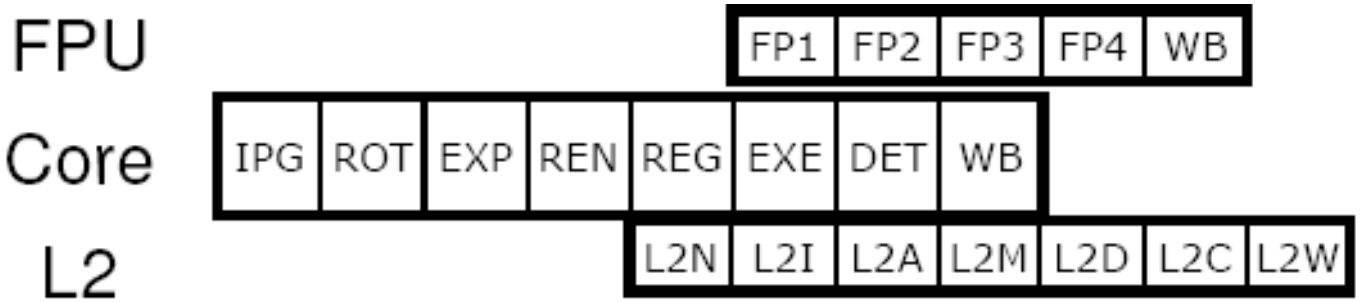
Не используется

Исполнительные устройства





Конвейер Itanium2



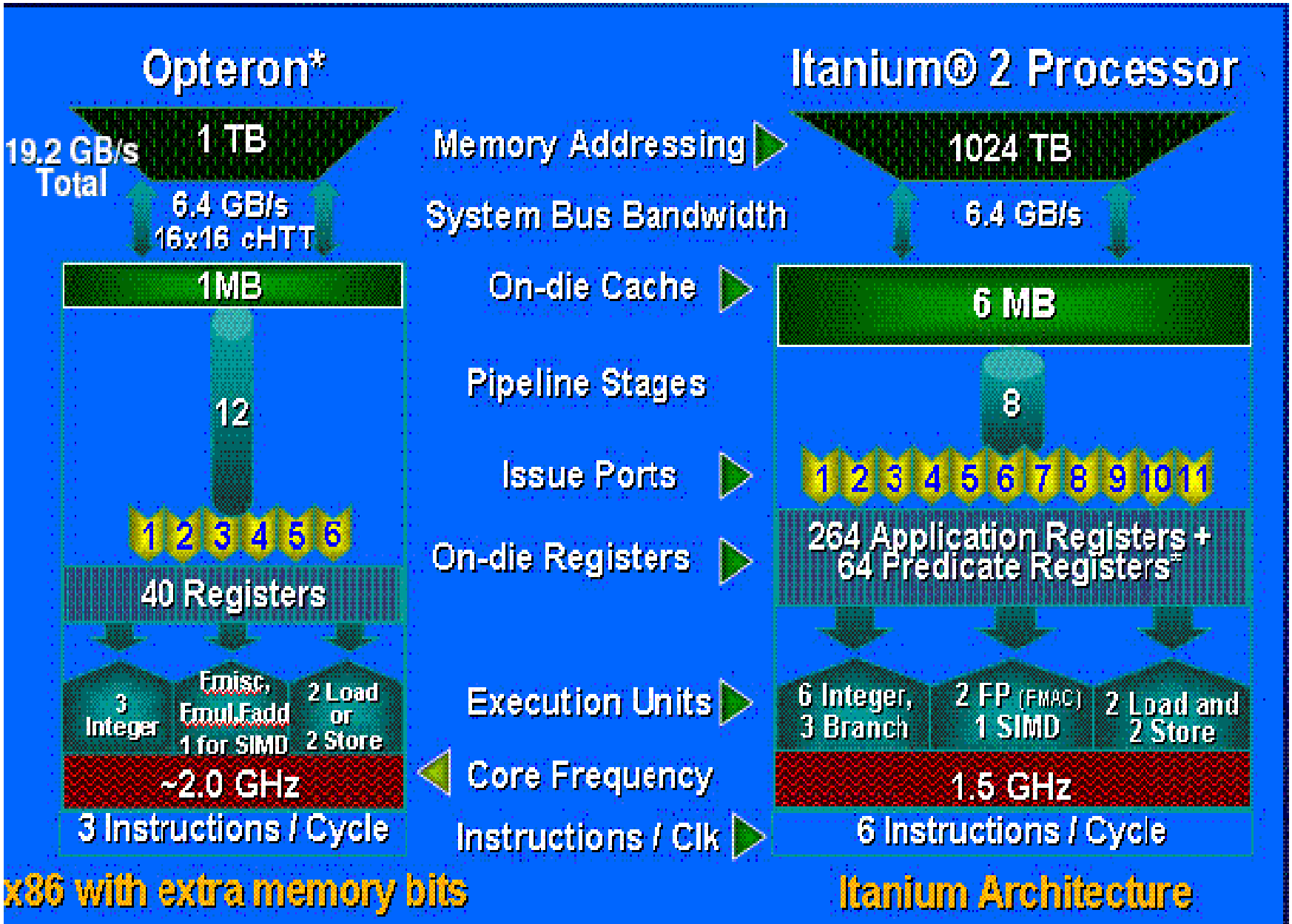
IPG	Вычисление IP, чтение кэша L1I (**6** инст.) и TLB.	**EXE**	Выполнение (6), обращение к кэшу L1D и TLB + обращение к тэгам L2 кэша (**4**)
ROT	Расцепление и буферизация инструкций.	**DET**	Обнаружение исключений, выполнение переходов
EXP	Разворачивание инструкции, назначение порта	**WB**	Завершение, 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
- Требует малых задержек по памяти

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)
  - Используется и для целочисленного умножения
- 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


Opteron*
Itanium® 2 Processor
19.2 GB/s Total
1 TB
Memory Addressing
1024 TB
6.4 GB/s 16x16 cHTT
System Bus Bandwidth
6.4 GB/s
1MB
On-die Cache
6 MB
12
Pipeline Stages
8
1 2 3 4 5 6
Issue Ports
1 2 3 4 5 6 7 8 9 10 11
40 Registers
On-die Registers
264 Application Registers + 64 Predicate Registers*
3 Integer
Fmisc, Fmul,Fadd 1 for SIMD
2 Load or 2 Store
Execution Units
6 Integer, 3 Branch
2 FP (FMAC) 1 SIMD
2 Load and 2 Store
~2.0 GHz
Core Frequency
1.5 GHz
3 Instructions / Cycle
Instructions / Clk
6 Instructions / Cycle
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.

Сравнение двух архитектур

свойства	суперскаляр	VLIW/EPIC
Выявление ILP	Аппаратура	Компилятор
Формат команд	RISC команда	Связка из RISC команд
Конвейер	Синхронный	Асинхронный
Предсказание переходов	Динамическое, статическое	Статическое,
Исполнение команд	Простое	Простое и предикатное
Переименование регистров	Есть	Нет
Число одновре-менно исполняе-мых команд	Несколько команд	Несколько связок команд

Сравнение двух архитектур

свойства	суперскаляр	VLIW/EPIC
Число одновременно исполняемых команд	Несколько команд	Несколько связок команд
Спекуляция по коду	Есть, исполнение вне порядка	Есть, исполнение вне порядка и предикатное
Спекуляция по данным	нет	Есть, исполняется программно

Software Pipelining Example

C Code Example

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

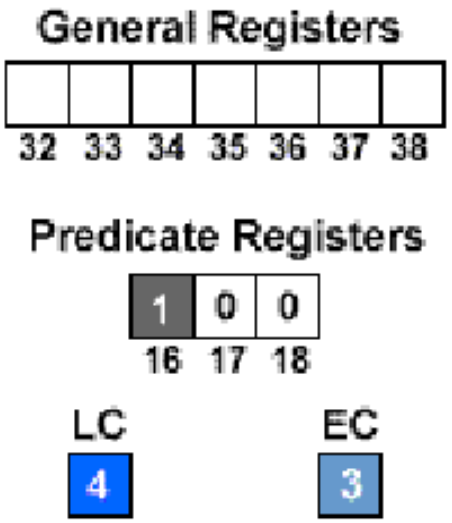
Pseudo Assembly Code

```
// 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.

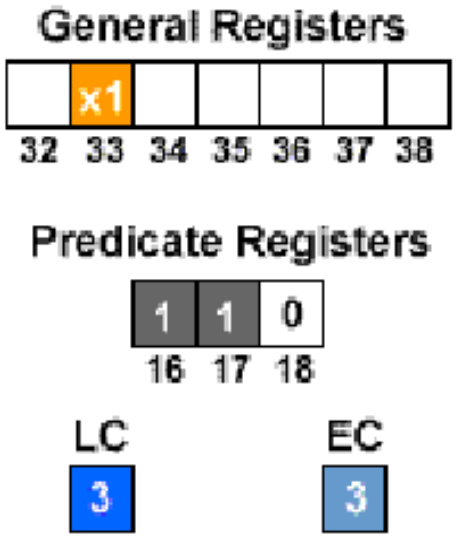
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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.

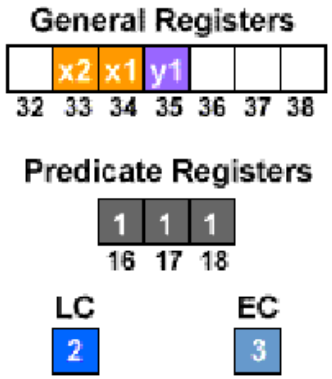
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13]  = r35,1
           br.ctop loop
```



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.

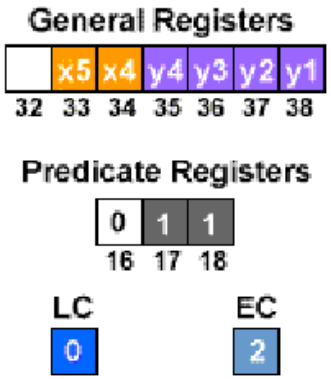
```
loop:
  (p16)   ld1 r32    = [r12],1
  (p17)   add r34    = 1,r33
  (p18)   st1 [r13] = r35,1
          br.ctop loop
```



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.

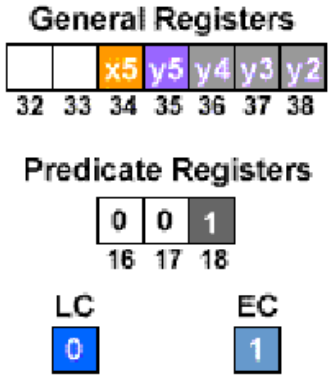
```
loop:
  (p16)   ld1 r32    = [r12],1
  (p17)   add r34    = 1,r33
  (p18)   st1 [r13] = r35,1
          br.ctop loop
```



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

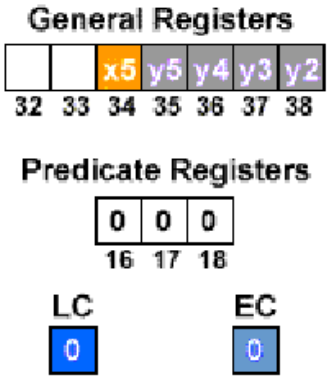
```
loop:
   (p16)   ld1 r32    = [r12],1
   (p17)   add r34    = 1,r33
   (p18)   st1 [r13] = r35,1
           br.ctop loop
```



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
```



Эльбрус

Процессоры Эльбрус

- Передовые процессоры российского производства с архитектурой VLIW (производство с 2014 г.)
- 65 нм техпроцесс, частота 800 МГц, 4 ядра и 8 МБ кэш L2
- Набор команд «Эльбрус», использует технологию двоичной компиляции для совместимости с x86

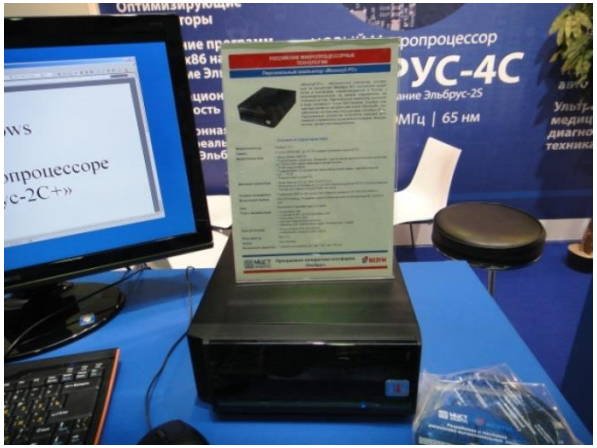
Поддержка исполнения x86 приложений и ОС

Обеспечивает исполнение ОС: MS DOS, Windows (95, NT, 2000, XP), нескольких вариантов Linux, FreeBSD, QNX

Особенности архитектуры Эльбрус

- Регистровый файл: 256 РОН (84 бит)
- Предикатных регистров: 32
- Поддержка конвейеризации циклов
- Программируемый аппаратный предзагрузчик данных в кэш
- 1 VLIW-команда задаёт до 23-х операций
- Специализированные модули для формирования многопроцессорных систем

Стандартные комплектации



Сравнительное тестирование производительности

	Intel Core i7-2600	Эльбрус-2С+	Эльбрус-4С
Количество ядер	4	2	4
Тактовая частота, МГц	3400	500	700
Объём кеш-памяти, МБ	8	2	8
Количество процессоров в системе	1	1	4
Объём оперативной памяти системы, ГБ	16	4	64

	Intel Core i7-2600	Эльбрус-2С+	Эльбрус-4С
Архивация по алгоритму 7zip (сжатие), Мбайт/с	3,95	0,543	0,665
Архивация по алгоритму 7zip (распаковка), Мбайт/с	33,437	6,296	8,679
Цифровая фильтрация сигнала, с	1,384	3,469	2,474
Шифрование по алгоритму ГОСТ, с	2,102	1,601	1,112

Q? & A!