Universidade Federal do Rio Grande do Sul
Instituto de Informática

Organização de Computadores

Aula 23

Arquiteturas VLIW

Arquiteturas VLIW

1. Introdução

2. Escalonamento

3. Escalonamento acíclico

4. Escalonamento cíclico

5. Exemplo

6. Processador Itanium

7. Processador Crusoe

1. Introdução

- **VLIW é Very Long Instruction Word**
- **Máquinas que exploram paralelismo no nível das instruções**

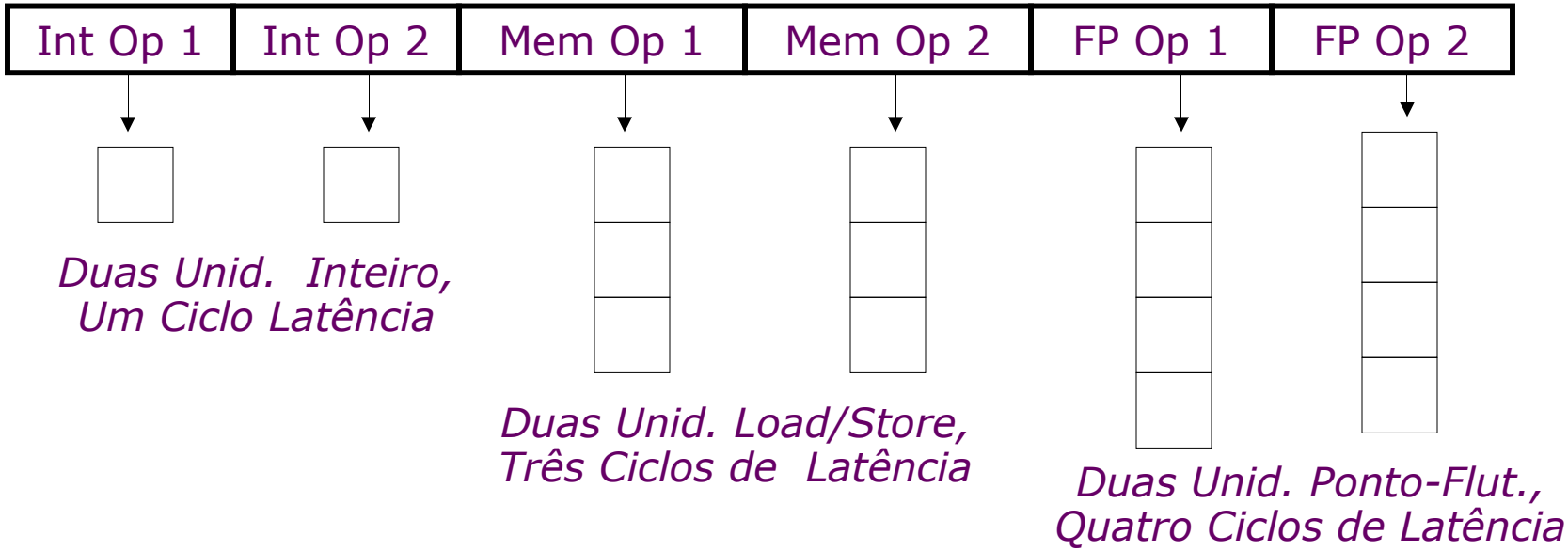
- **Várias operações são executadas em paralelo em diferentes unidades funcionais, tais como em máquinas superescalares**

- **A diferença está no controle do despacho e da terminação das operações**
- **Superescalares: as dependências são resolvidas em tempo de execução por um hardware dedicado**
- **VLIW: as dependências são resolvidas em tempo de compilação pelo compilador**

Introdução

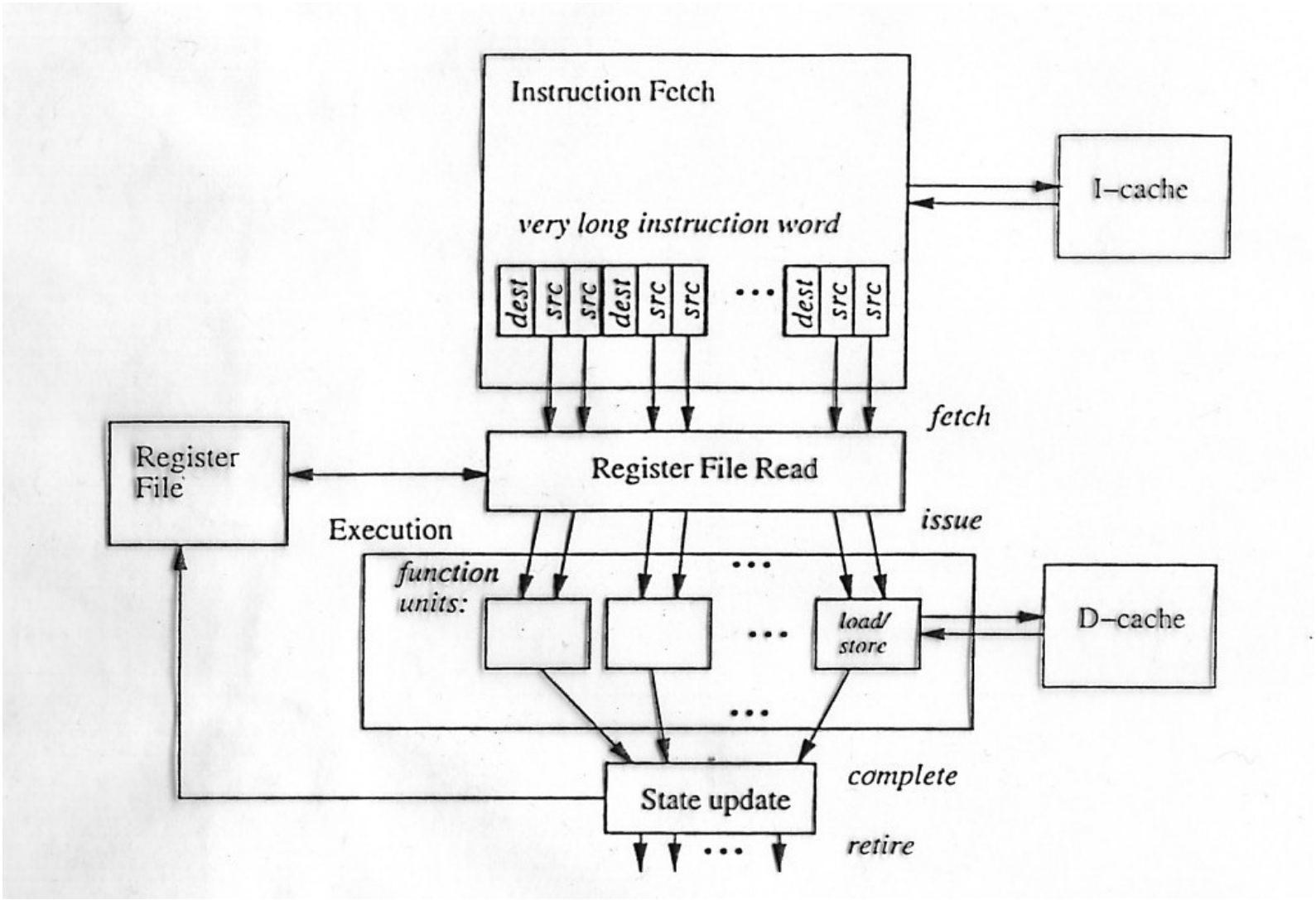
- **Em uma máquina VLIW, várias operações (instruções em uma máquina normal) são codificadas em uma mesma instrução**
- **A palavra de instrução é bastante longa, podendo conter várias operações (que operam sobre vários operandos) independentes**
- **A posição de cada operação dentro da palavra VLIW determina a unidade funcional que será usada**
- **O hardware de despacho é simples**

VLIW: Very Long Instruction Word



- **Múltiplas operações empacotadas em uma instrução**
- **Cada campo de operação é para uma função fixa**
- **Latências constantes de operações são especificadas**
- **A arquitetura deve garantir:**
  - **Paralelismo dentro das instruções**
  - **Nenhum dado seja usado antes estar pronto => sem interlocks de dados**

Organização VLIW



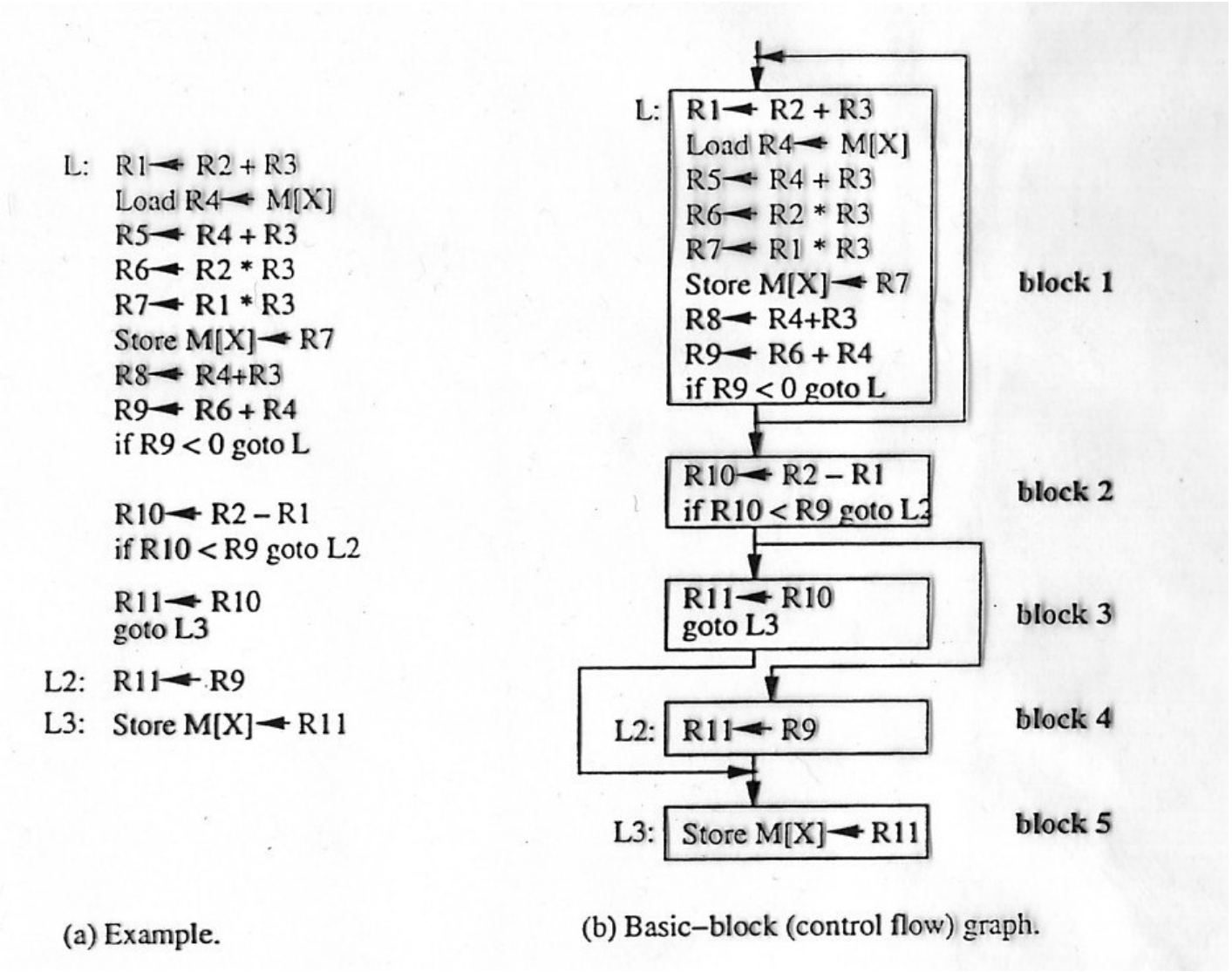
Programas e Fluxos

- **A compilação de um programa é (normalmente) feita em três fases**
  - ***parsing* da linguagem de entrada para uma descrição (estrutura) intermediária**
  - **otimização da estrutura intermediária**
  - **geração do código para a arquitetura-alvo**

- **A estrutura de dados intermediária representa:**
  - **fluxo de controle – as diferentes seqüências de execução das instruções que formam o programa**
  - **fluxo de dados – as dependências de dados que existem entre as várias operações**

- **Blocos básicos:**
  - **grupos de instruções que são SEMPRE executadas em seqüência**

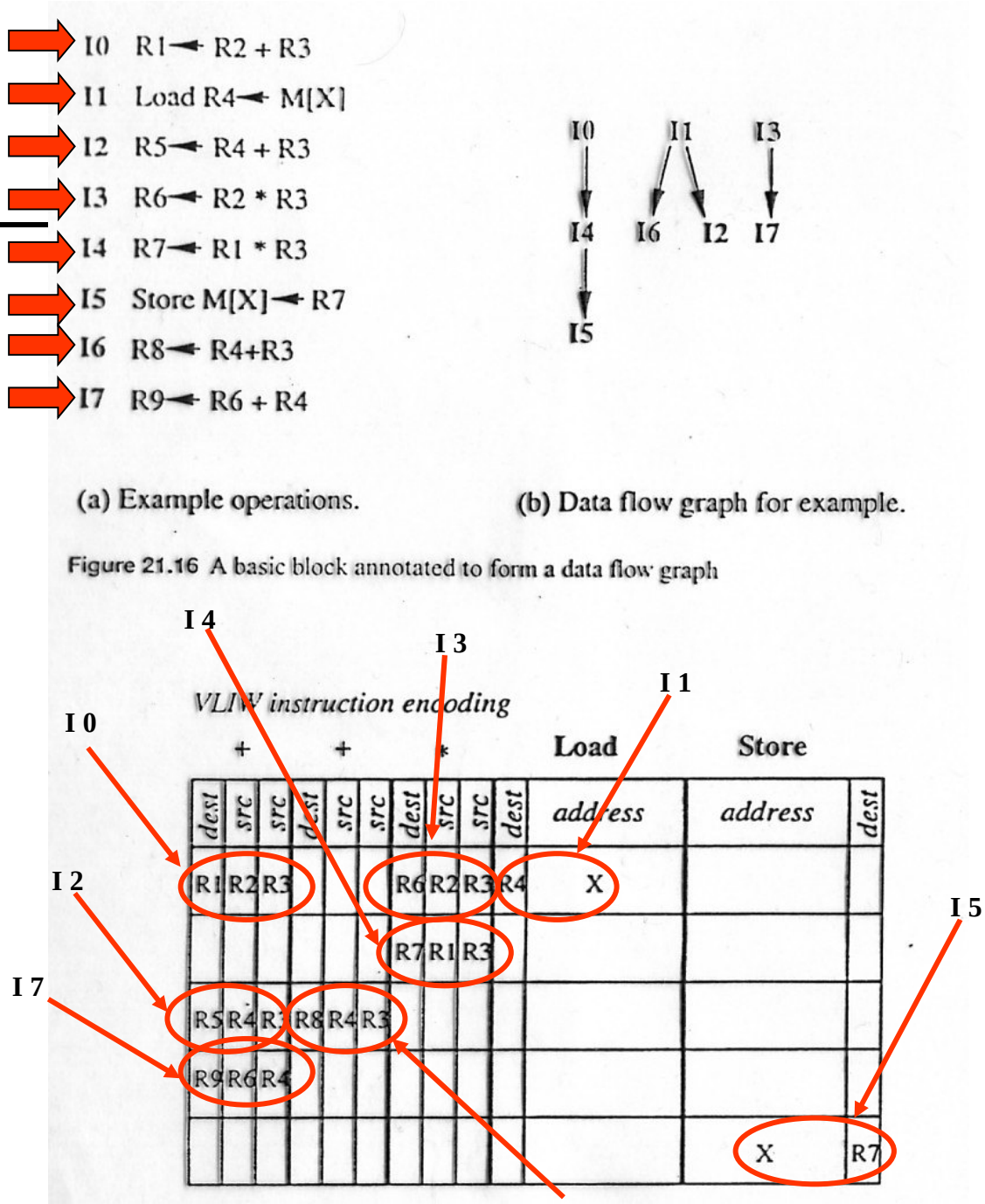
Blocos Básicos



2. Escalonamento

- **O escalonamento das operações consiste em determinar as operações que serão executadas em paralelo**
- **Em uma máquina VLIW o compilador é responsável por esta tarefa**
- **Operações que são executadas em paralelo são atribuídas à mesma palavra de instrução**
- **Em um mesmo bloco básico (seqüencial) estas instruções podem ser escalonadas com base no fluxo de dados**
  - **escalonamento ASAP (*as soon as possible*)**
- **A transição da fronteira entre blocos exige técnicas mais elaboradas**
  - **escalonamento acíclico: programas com desvios, sem ciclos**
  - **escalonamento cíclico: programas com ciclos e laços**

Escalonamento dentro de um bloco básico



Latências:

Adder = 1 ciclo

Multiplier = 3 ciclos

Load = 2 ciclos

Store = 1 ciclo

3. Escalonamento Acíclico

- **Exemplo de técnica: escalonamento baseado em predicados**
- **Operações que se seguem a um *if* são associadas com um predicado**
  - **valor do predicado é definido em função do resultado do teste do *if***
- **A palavra VLIW deve ser acrescida de um registrador de predicado para cada instrução**
- **A máquina deve ser acrescida de uma unidade operativa para implementar operador de definição de predicados**
- **Uma operação ...**
  - **é completada se o predicado é verdadeiro**
  - **não é completada se o predicado é falso**
- **Esta técnica é conhecida como *if-conversion***

Predicação

- **Conceito:**
  - **Execução condicional de instruções**
- **Exemplo: if (a>b) c = c + 1**

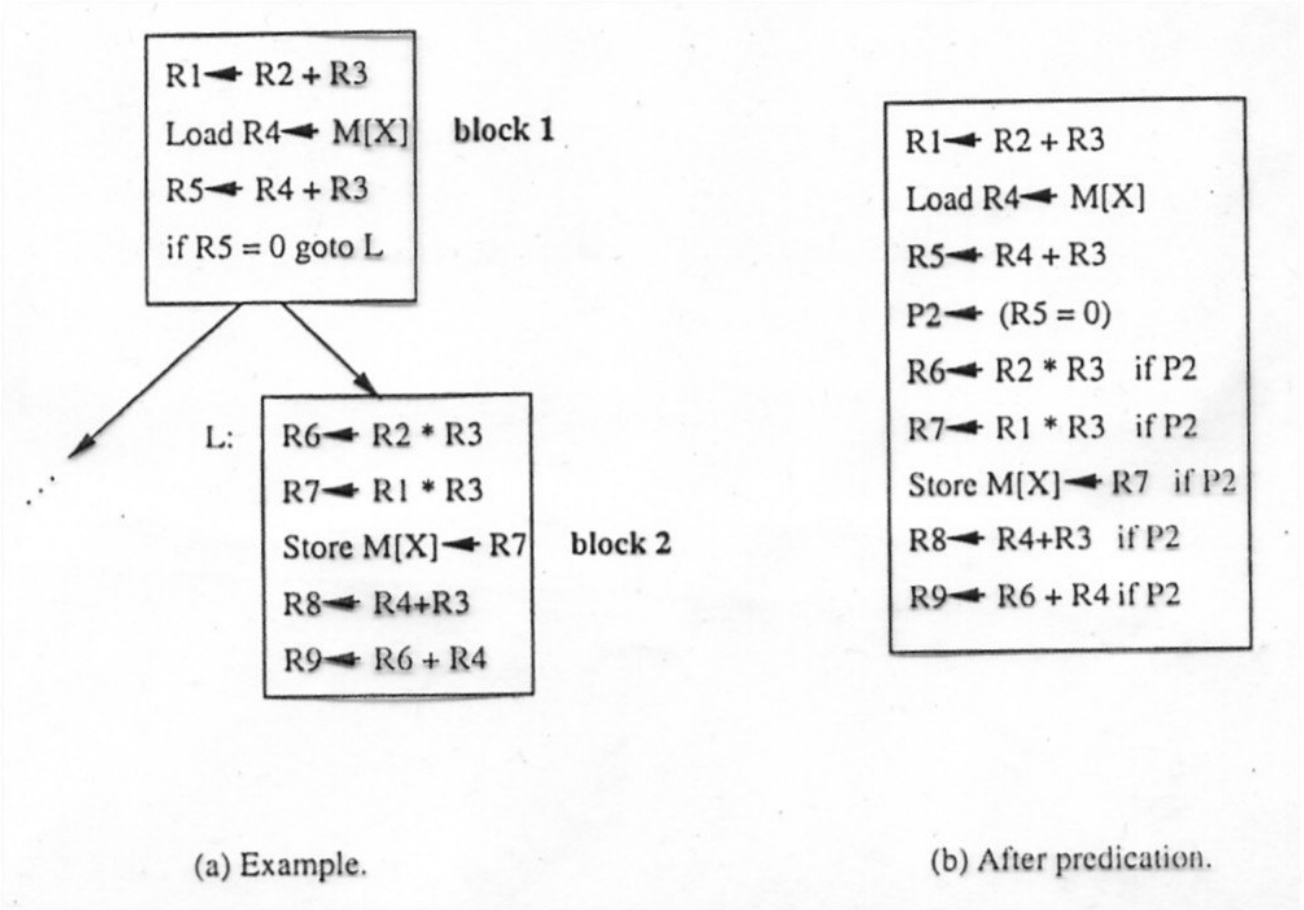
  else d = d * e + f
- **Çódigo transformado pelo compilador:**

  pT,pF = compare(a>b)

  if (pT) c = c + 1

  if (pF) d = d * e + f
- **Conseqüências:**
  - **Dependências de controle são transformadas em depend. dados**
  - **As duas operações podem ocorrer em paralelo**
  - **pT e pF (predicados) podem ser implementados em hardware**

Predicação


R1 ← R2 + R3
Load R4 ← M[X]
R5 ← R4 + R3
if R5 = 0 goto L
block 1
L:
R6 ← R2 * R3
R7 ← R1 * R3
Store M[X] ← R7
R8 ← R4+R3
R9 ← R6 + R4
block 2
(a) Example.
R1 ← R2 + R3
Load R4 ← M[X]
R5 ← R4 + R3
P2 ← (R5 = 0)
R6 ← R2 * R3 if P2
R7 ← R1 * R3 if P2
Store M[X] ← R7 if P2
R8 ← R4+R3 if P2
R9 ← R6 + R4 if P2
(b) After predication.

4. Escalonamento Cíclico

- **O código do ciclo inicial é reescrito seqüencialmente até se conseguir um padrão de instruções que se repetem ao longo do tempo**
  - ***loop unrolling***
- **Este padrão repetido ao longo do tempo é chamado de *kernel* do laço**
- **O kernel do laço será realizado por uma (ou mais) instruções VLIW**
- **Cada instrução VLIW pode conter operações de diferentes ciclos antes da duplicação (isto é representado com índices)**

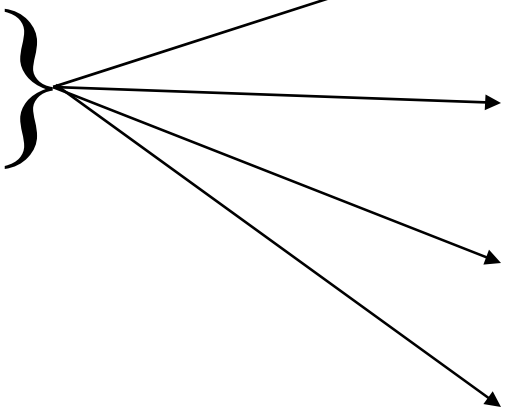
Escalonamento Cíclico: exemplo

```
R1 ⇐ 2.14
R3 ⇐ X
R5 ⇐ Y

L:  R2 ⇐ load M [ R3++ ]
    R4 ⇐ R2*R1
    store M [ R5++ ] ⇐ R4

    if R3 < X+100 goto L
```

Laço inicial



```
R1 ⇐ 2.14
R3 ⇐ X
R5 ⇐ Y

L:  R2 ⇐ load M [ R3++ ]
    R4 ⇐ R2*R1
    store M [ R5++ ] ⇐ R4

    R2 ⇐ load M [ R3++ ]
    R4 ⇐ R2*R1
    store M [ R5++ ] ⇐ R4

    R2 ⇐ load M [ R3++ ]
    R4 ⇐ R2*R1
    store M [ R5++ ] ⇐ R4

    R2 ⇐ load M [ R3++ ]
    R4 ⇐ R2*R1
    store M [ R5++ ] ⇐ R4

    if R3 < X+100 goto L
```

Laço reescrito 4 vezes

Escalonamento Cíclico: exemplo

```
        R1 ⇐ 2.14
        R3 ⇐ X
        R5 ⇐ Y

L:      R2 ⇐ load M[R3++]
        R4 ⇐ R2*R1
        store M[R5++] ⇐ R4

        R2 ⇐ load M[R3++]
        R4 ⇐ R2*R1
        store M[R5++] ⇐ R4

        R2 ⇐ load M[R3++]
        R4 ⇐ R2*R1
        store M[R5++] ⇐ R4

        R2 ⇐ load M[R3++]
        R4 ⇐ R2*R1
        store M[R5++] ⇐ R4

        if R3<X+100 goto L
```

Laço reescrito 4 vezes seqüencialmente

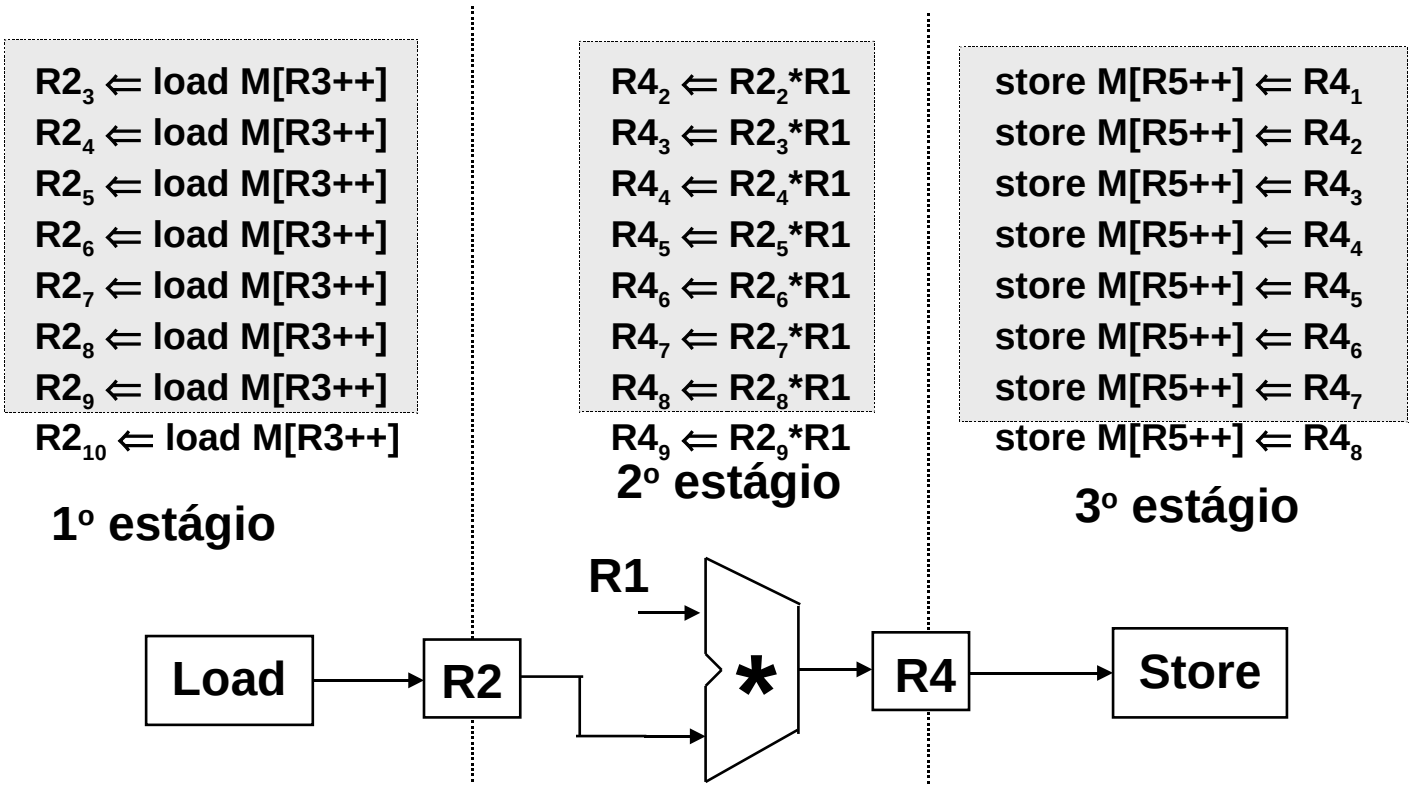
```
R1 ⇐ 2.14
R3 ⇐ X
R5 ⇐ Y

R2_1 ⇐ load M[R3++]
R2_2 ⇐ load M[R3++]   R4_1 ⇐ R2_1*R1
R2_3 ⇐ load M[R3++]   R4_2 ⇐ R2_2*R1    store M[R5++] ⇐ R4_1
R2_4 ⇐ load M[R3++]   R4_3 ⇐ R2_3*R1    store M[R5++] ⇐ R4_2
R2_5 ⇐ load M[R3++]   R4_4 ⇐ R2_4*R1    store M[R5++] ⇐ R4_3
R2_6 ⇐ load M[R3++]   R4_5 ⇐ R2_5*R1    store M[R5++] ⇐ R4_4
R2_7 ⇐ load M[R3++]   R4_6 ⇐ R2_6*R1    store M[R5++] ⇐ R4_5
R2_8 ⇐ load M[R3++]   R4_7 ⇐ R2_7*R1    store M[R5++] ⇐ R4_6
R2_9 ⇐ load M[R3++]   R4_8 ⇐ R2_8*R1    store M[R5++] ⇐ R4_7
R2_10 ⇐ load M[R3++]  R4_9 ⇐ R2_9*R1    store M[R5++] ⇐ R4_8
                      R4_10 ⇐ R2_10*R1   store M[R5++] ⇐ R4_9
                                         store M[R5++] ⇐ R4_10
if R3<X+100 goto L
```

Ao longo do tempo estas três operações podem sempre ser executadas em paralelo

Escalonamento Cíclico: exemplo

Ao longo do tempo estas três operações podem sempre ser executadas em paralelo



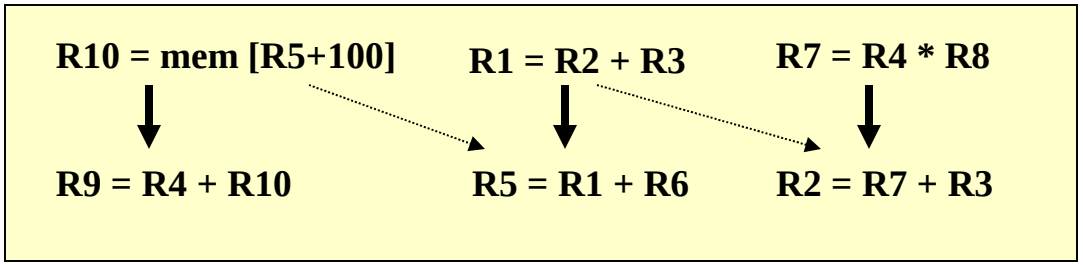
Na prática o compilador usou as unidades operativas para configurar (via software) um pipeline de 3 estágios dedicado à execução deste laço

5. Exercício

I1: R1 = R2 + R3
I2: R10 = mem[R5+100]
I3: R5 = R1 + R6
I4: R7 = R4 * R8
I5: R2 = R7 + R3
I6: R9 = R4 + R10

Somador tem latência de 1 ciclo
Multiplicador e memória têm latência de 2 ciclos

Dependências de dados → **verdadeiras**
⇢ **falsas**



+	+	*	load / store
R1 = R2 + R3	nop	R7 = R4 * R8	R10 = mem […]
R5 = R1 + R6	nop	nop	nop
R9 = R4 + R10	R2 = R7 + R3	nop	nop

VLIW gasta mais memória de programa que superescalar

6. Processador Itanium

- **IA64 é uma arquitetura VLIW de 64 bits da Intel**
- ***Itanium* é o primeiro processador projetado com a tecnologia IA64**
- **O compilador agrupa e escalona as instruções com base na filosofia EPIC (*Explicitly Parallel Instruction Computing*)**

Visão conceitual do Itanium

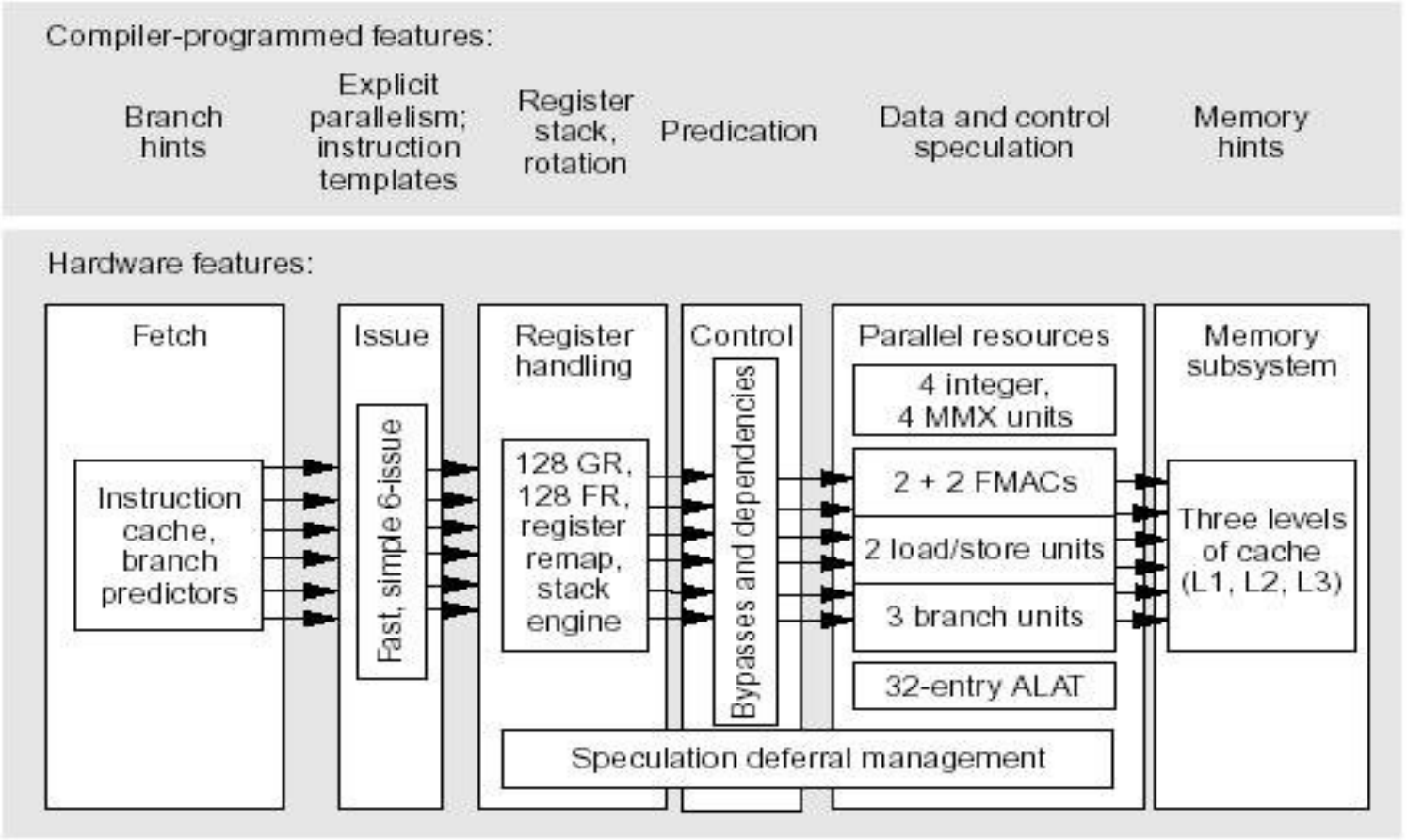
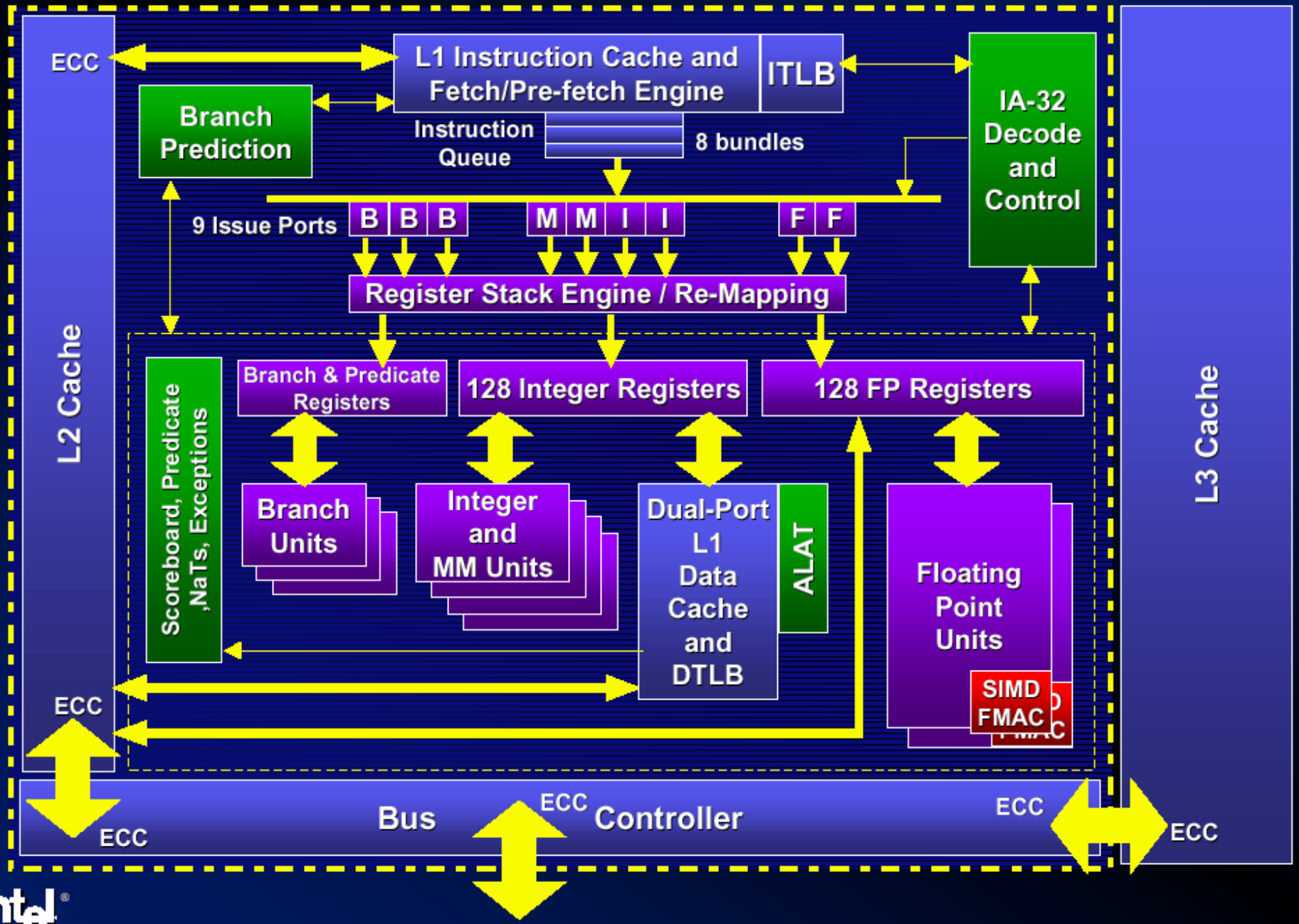


Figure 1. Conceptual view of EPIC hardware. GR: general register file; FR: floating-point register file

Intel® Itanium™ Processor Block Diagram



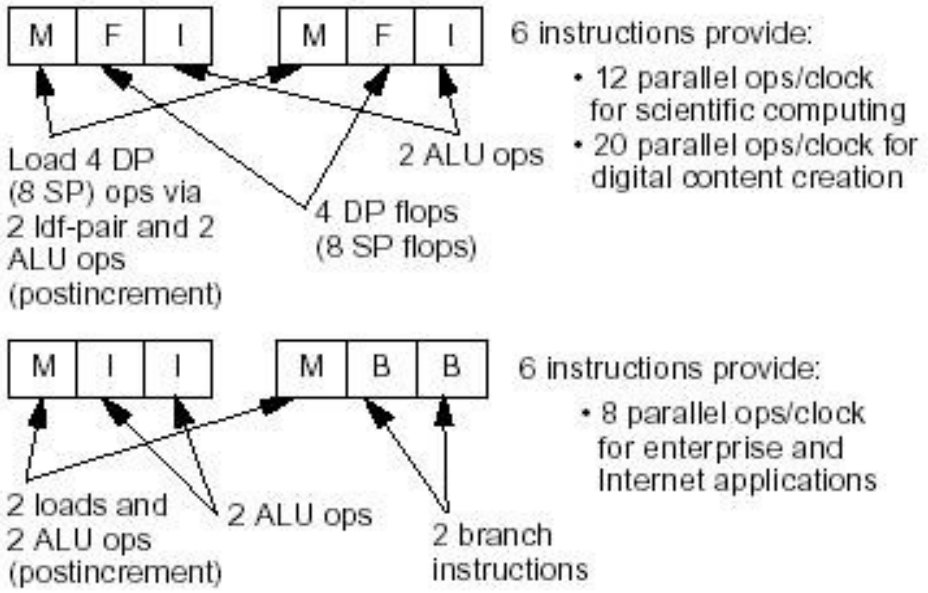
Itanium: Empacotamento de Instruções

- **Instruções são empacotadas em maços (*bundles*)**
- **O compilador identifica as instruções com independência de dados e as empacota em uma instrução VLIW**
  - **Como conseqüência, o processador não necessita verificar a dependência de dados, apenas carregar as instruções nas unidades apropriadas**
- **O compilador resolve também os problemas de falsas dependências → WAR e WAW**
- **Três instruções são combinadas em uma instrução de 128 bits**
- ***Template* auxilia na decodificação e roteamento das instruções para as unidades de execução**

Itanium: Leitura das Instruções

- **A memória *cache* de instruções possui 16 Kbytes**
  - **4-way set-associative**
- **Permite a leitura simultânea de 32 bytes**
  - **2 instruções VLIW (*bundles*)**
  - **6 instruções lógico/aritméticas ou de acesso à memória**
- **As instruções lidas são armazenadas em uma "fila de instruções" com capacidade para 8 *bundles***
  - **Instruções continuam a ser pré-carregadas na fila mesmo quando o processador necessita ser paralisado, devido a um erro na previsão de desvio, por exemplo**

Itanium: Configurações para o despacho de instruções para as unidades funcionais



• Instruções são codificadas de modo a resultar em mais de uma operação a ser executada concorrentemente, devido à disponibilidade de unidades funcionais no processador

Itanium: Problemas com Instruções de Desvio

- **Problemas relativos à execução de instruções de desvio são tratados através de 4 estratégias:**

  1. separação das operações de desvio em:
  - ***Prepare-to-branch*, que calcula o endereço de desvio;**
  - ***Compare*, que calcula a condição do desvio; e**
  - ***Actual branch*, que especifica quando o controle é transferido para as instruções destino do desvio.**

  2. suporte à eliminação de instruções de desvio

  3. suporte à movimentação de instruções entre diferentes blocos básicos

  4. predicação dinâmica de desvios

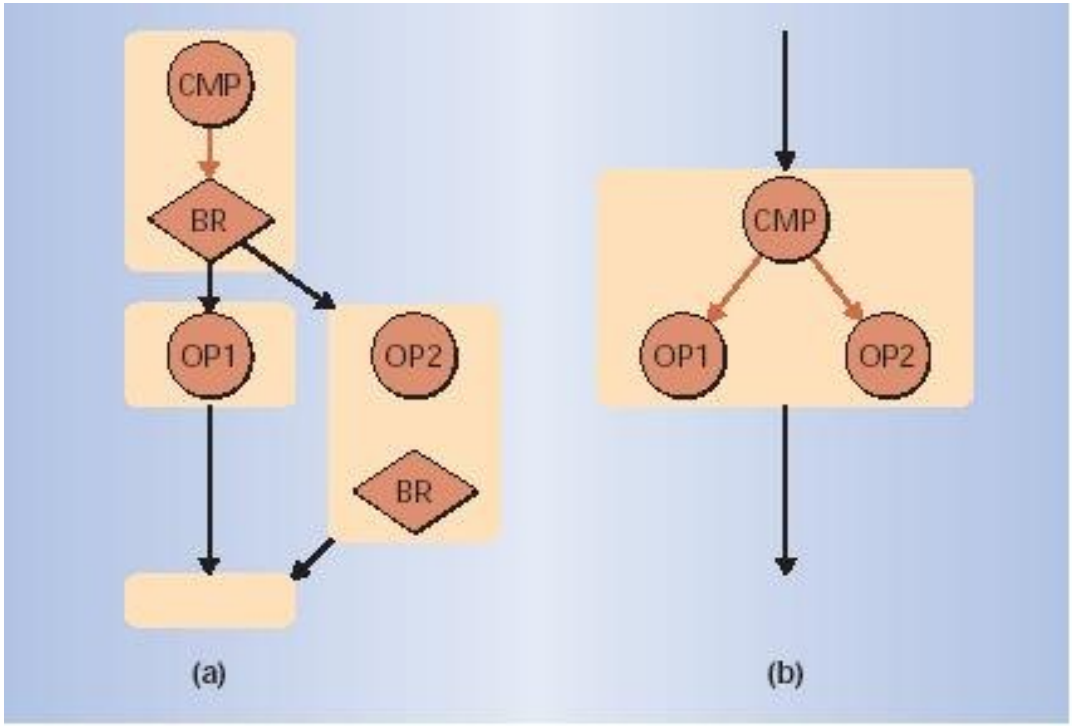
- **1, 2 e 3 referem-se ao escalonamento estático e são realizadas pelo compilador**

- **4 refere-se ao escalonamento dinâmico e é implementada em HW**

Itanium: Eliminação de Desvios

- **Instruções de desvio podem ser eliminadas através da *predicação* de desvios (ou *if-conversion)***
- **Essa técnica baseia-se na separação das instruções de desvio em 3 sub-instruções**
- **As sub-instruções de *Prepare-to-branch e Compare* são escalonadas antecipadamente à posição original do desvio no código**
- **Um bit de *predicado* é associado a cada bloco básico seguinte à instrução de desvio**
  - **as instruções que estão no caminho *verdadeiro* da condição do desvio (calculada por *compare*) são marcadas como "verdadeiras" e as demais, como "falsas"**
  - **as instruções com predicado "falso" não são executadas pelo processador**

Itanium: Exemplo - Eliminação de Desvios



(a) programa original, com instrução de branch BR

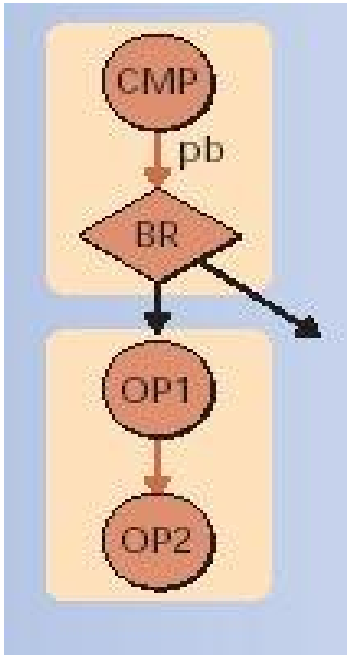
(b) programa transformado, com OP1 e OP2 sendo instruções com predicação

Itanium: Movimentação de Instruções com "Predicação"

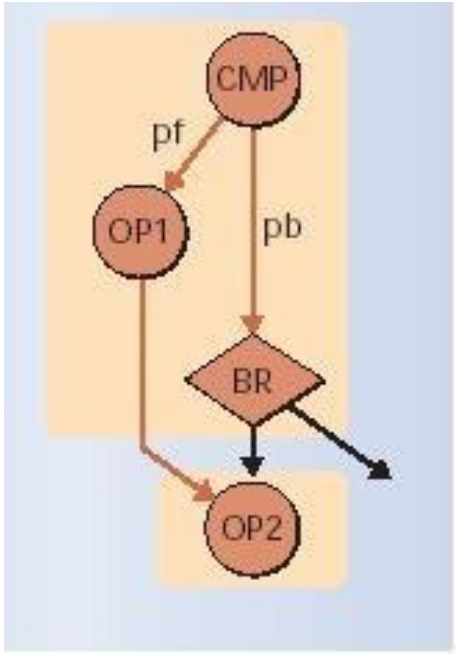
- **A movimentação de instruções entre diferentes blocos básicos é suportada pela predicação de instruções**

- **Através dessa técnica as instruções podem ser movimentadas sem a necessidade de se realizar execução especulativa**
  - **a movimentação de instruções permite ao compilador explorar melhor o paralelismo oferecido pelo processador**
  - **processadores EPIC não executam instruções de desvio especulativamente**

Itanium: Exemplo – Movimentação de Instruções



Estrutura original



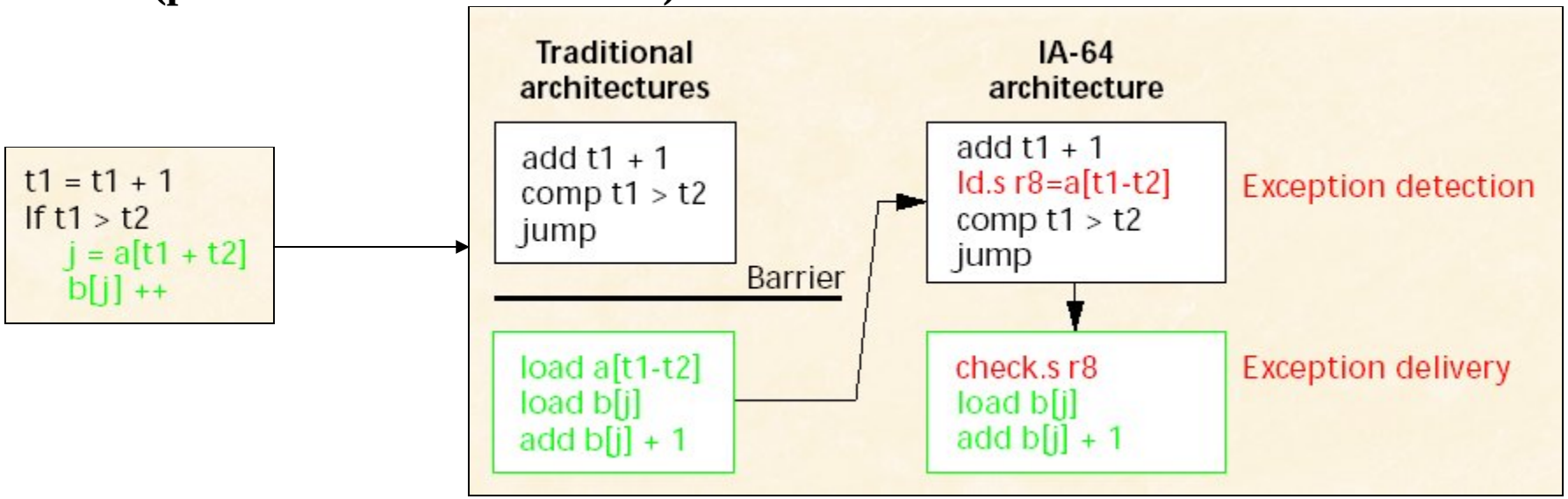
- "OP1" movimentada para outro Bloco Básico
- "OP1" somente executa se o *branch* for verdadeiro, pois seu predicado é pf = $\overline{pb}$

Itanium: Predição Dinâmica de Desvios

- **A *cache* de instruções permite a leitura de instruções e a verificação das previsões de desvio em apenas 1 ciclo de relógio**
- **Instruções de desvio são lidas na primeira metade de um ciclo e na segunda metade é verificada a sua informação acerca da previsão de desvios**
- **Se uma previsão é assumida como "verdadeira", as próximas instruções buscadas na memória correspondem ao endereço destino do desvio**
- **O *Itanium* possui uma *branch history cache* de 12 Kbytes**
  - **a previsão é realizada com base nas 4 últimas execuções da instrução de desvio**

Itanium: Especulação de Controle

- **Objetivo: executar instruções *load* antecipadamente para minimizar o impacto da latência de memória e aumentar o ILP**
- **Mover a execução de uma instrução de *load* para posição anterior ao controle do desvio (necessidade de pré-carregar dados para cache)**
- **É necessário tratar possível exceção devida a endereço inválido no *load* (por *cache miss* ou *TLB miss*)**



Itanium: Leitura Especulativa de operandos na memória

- **Instruções para leitura (*load*) e escrita na memória (*store*) são especulativamente executadas, baseados na premissa que os dados não serão alterados**

IA-32

Instruction

Instruction

Store [y]=r2

Load r1=[x]

Para aumentar o desempenho, [X] pode ser carregado antecipadamente em relação a sua posição original no código. No entanto, [y] pode sobrescrever [x] → [y] é um ponteiro para [x]

IA-64

Load.s r1=[x] especula

Instruction

Instruction

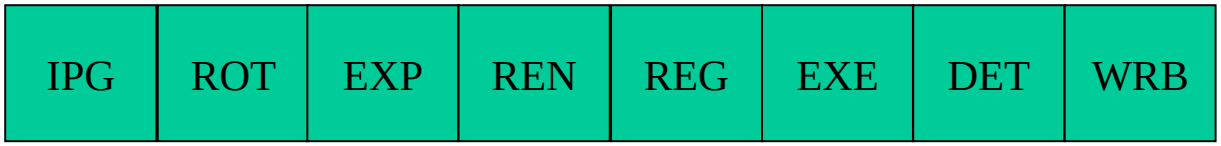
Store [y]=r1

Load.c r1=[x] verifica

[x] é carregado especulativamente e sua posição de memória é verificada (para ver se não ocorreram alterações) quando o valor é efetivamente utilizado

Itanium: *Pipeline*

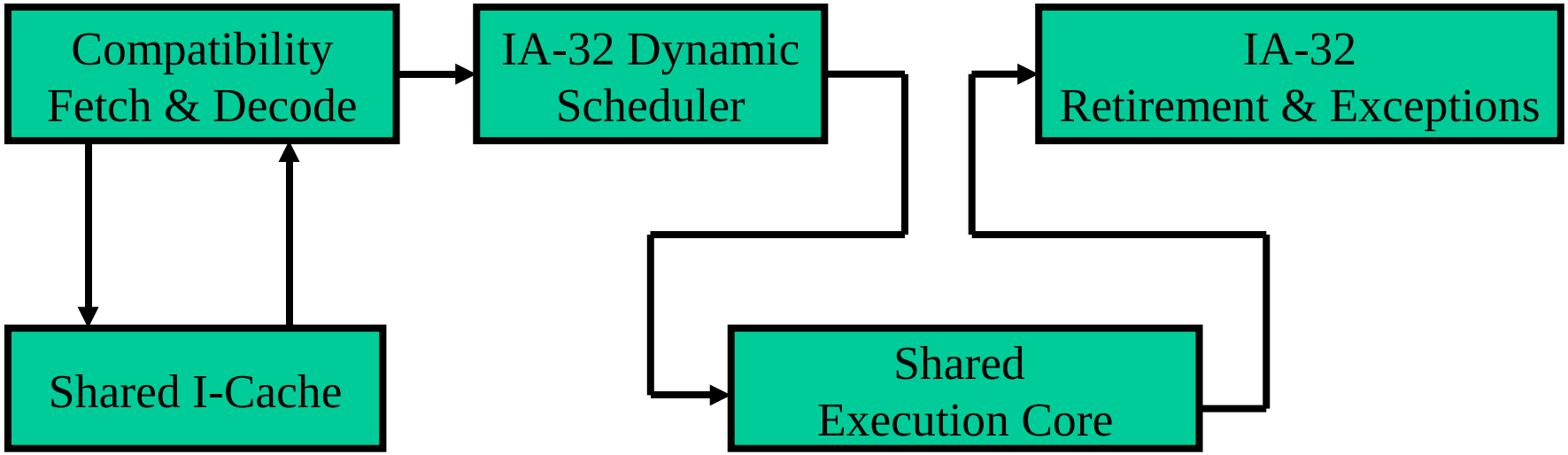
- **O *Itanium* possui um *pipeline* de 8 estágios**



IPG: cálculo do ponteiro de instruções
ROT: leitura das instruções
EXP: despacho de instruções
REN: renomeação de registradores
REG: leitura dos operandos
EXE: execução das operações
DET: detecção de exceções, correção de *branches* erroneamente previstos
WRB: escrita dos resultados, atualização dos registradores

Itanium: Compatibilidade com código IA32

- **execução direta de código IA32**



7. Processador Crusoe

- **Processador simples**
  - **poucas unidades funcionais**
  - **parte de controle simplificada**
    - **economia de potência**

- **Compatível com o conjunto de instruções "x86"**

- **Realiza tradução dinâmica de código, no nível do conjunto de instruções**
  - **aplicações são escritas com código "assembly" para o conjunto de instruções do "x86"**

- **Implementa um *compromisso* entre HW e SW na execução de algoritmos**
  - **SW → decodificação de instruções "x86" e escalonamento destas em instruções VLIW**
  - **HW → processador VLIW**

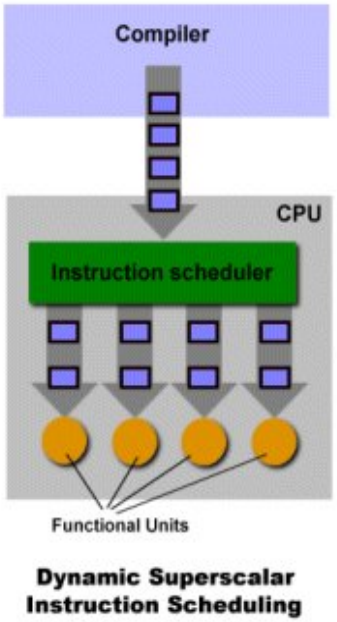
Crusoe: Características Arquiteturais

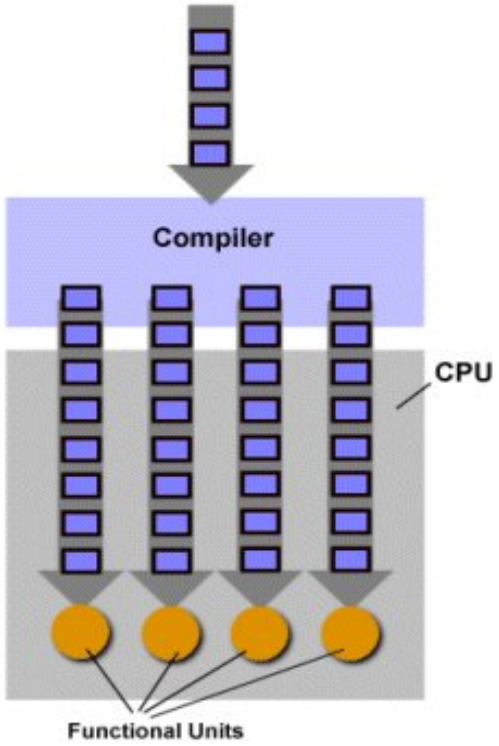
- **Processador VLIW com:**
  - **2 unidades de inteiros**
  - **1 unidade de ponto flutuante**
  - **1 unidade de memória**
  - **1 unidade para desvios**
  - **banco de registradores com 64 registradores para inteiros**
  - **banco de registradores com 32 registradores para ponto flutuante**

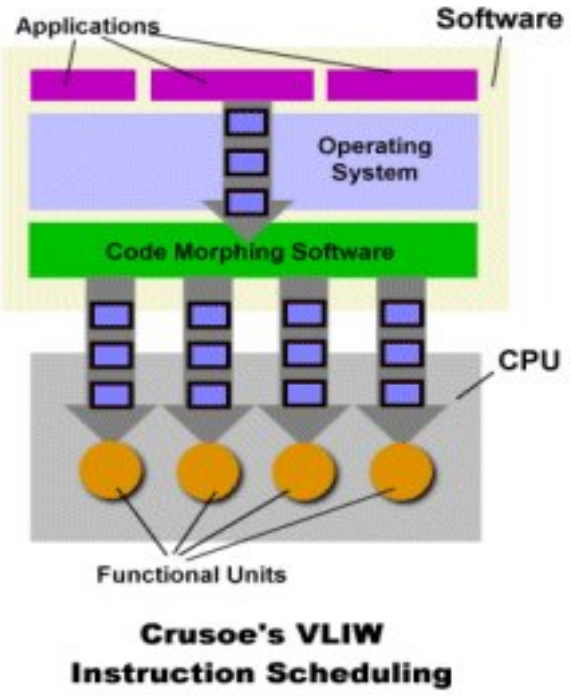
- **1 instrução VLIW é chamada de *molécula***
  - **pode conter até 4 instruções RISC, chamadas de *átomos***
  - **tamanho varia de 64 até 128 bits**
  - **todos os átomos executam em paralelo**
  - **a posição de cada átomo determina a unidade na qual este é executado**
    - **simplifica as unidades de decodificação e despacho de instruções**

Crusoe: Comparação entre Arquiteturas

Superescalar VLIW Crusoe





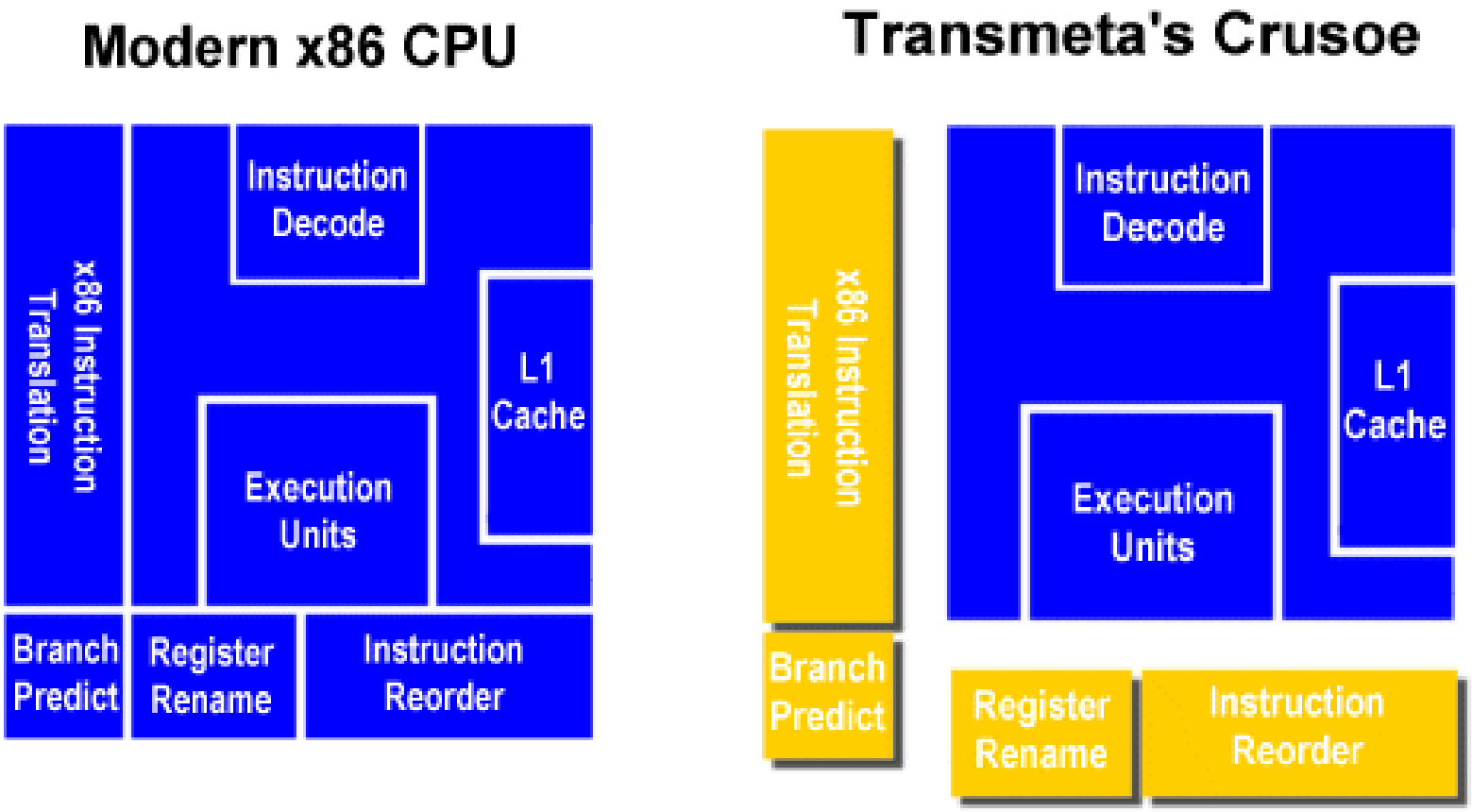


Crusoe: Moléculas e Mapeamento – princípios

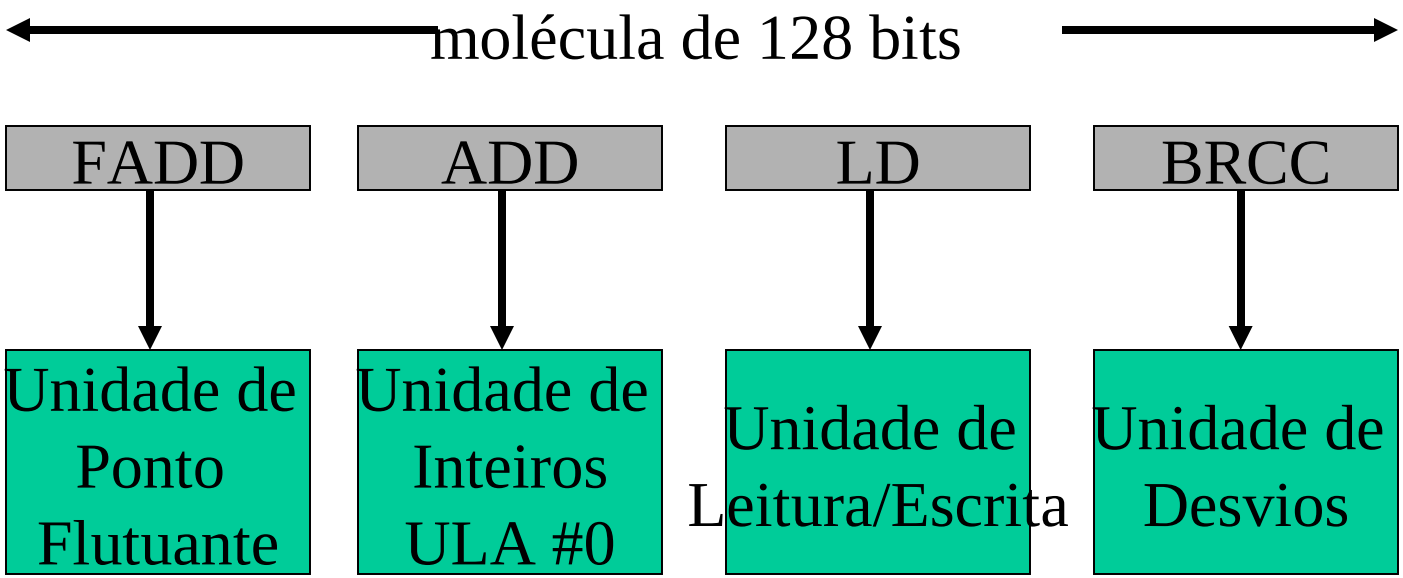
- ***Moléculas* são executadas sequencialmente**
  - **não há execução fora-de-ordem**
  - **hardware da parte de controle simplificado**

- **Moléculas são criadas a partir da tecnologia chamada "Code Morphing"**
  - **durante a execução do programa**
  - **tradução de código "x86"**
  - **tenta paralelizar ao máximo as instruções**
    - **moléculas com 4 instruções**

Crusoe *vs.* x86



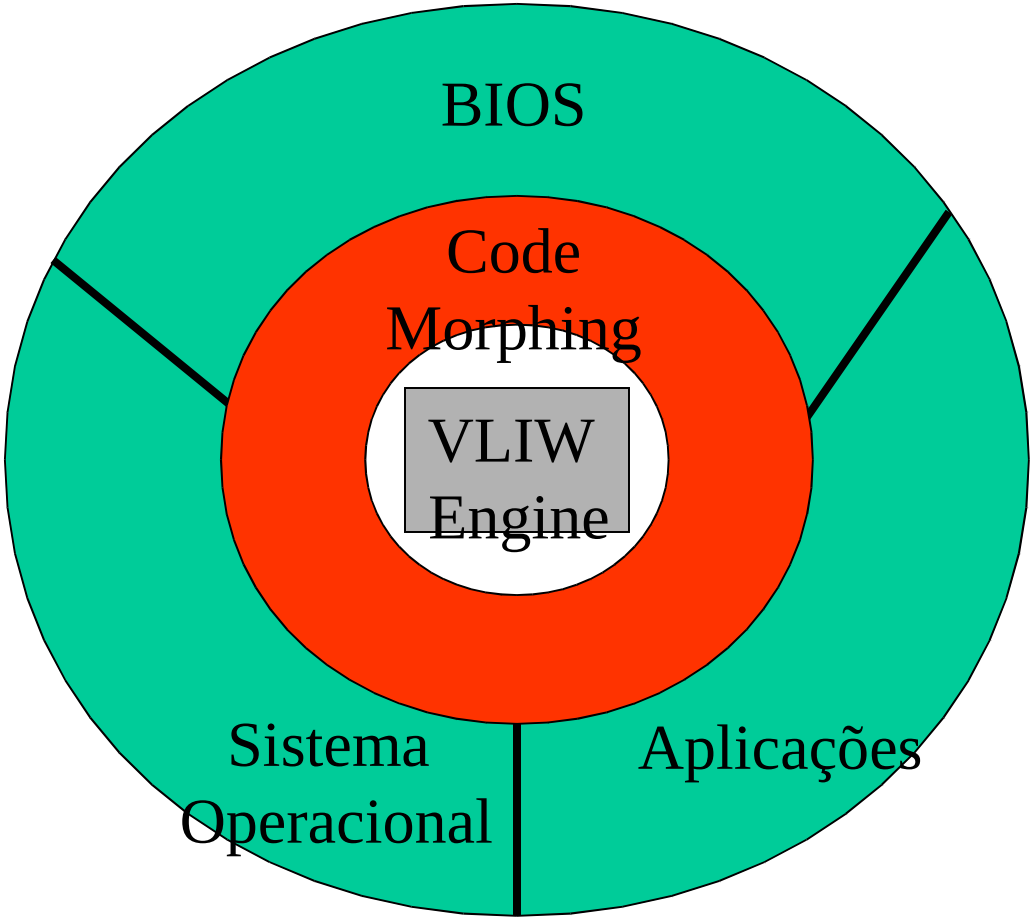
Crusoe: Moléculas e Mapeamento – exemplo



Crusoe: *Code Morphing*

- **Sistema de tradução dinâmica**
  - **x85 ISA → Crusoe - VLIW ISA**
- **Armazenado em uma ROM dentro do processador**
- **É o primeiro programa a ser executado quando o processador inicializa**
- **Apenas o próprio código do *Code Morphing* é escrito através do conjunto de instruções do Crusoe**

Crusoe: Relações entre o código da aplicação e o "code morphing"

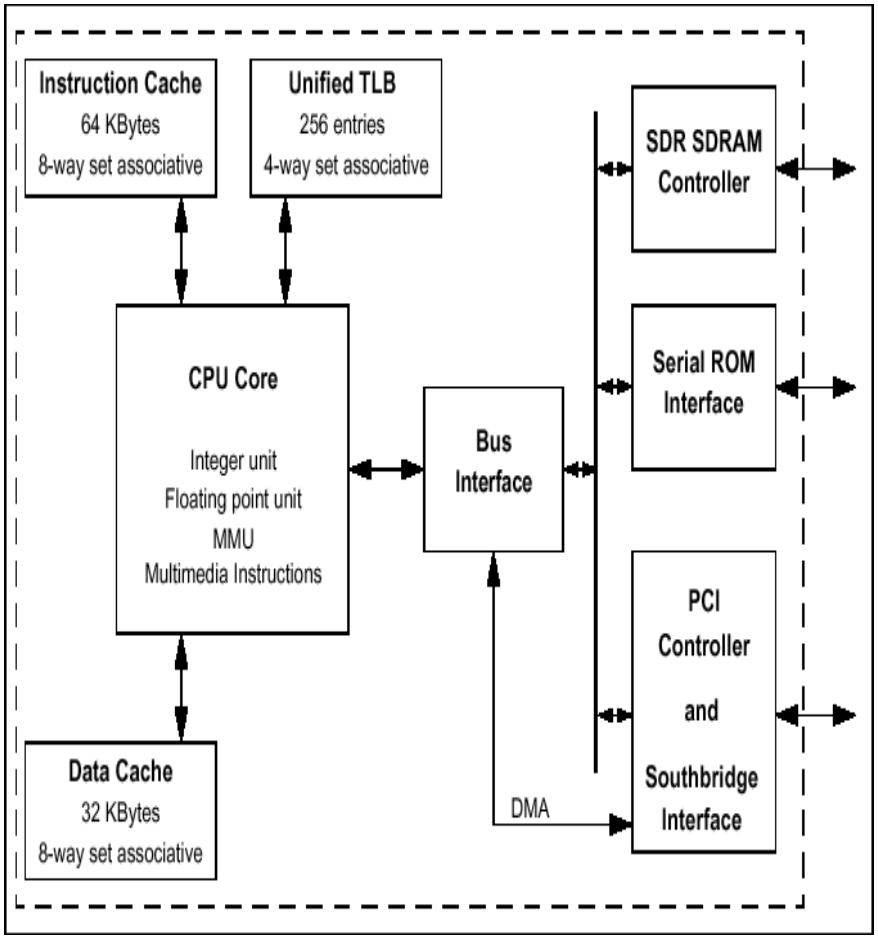


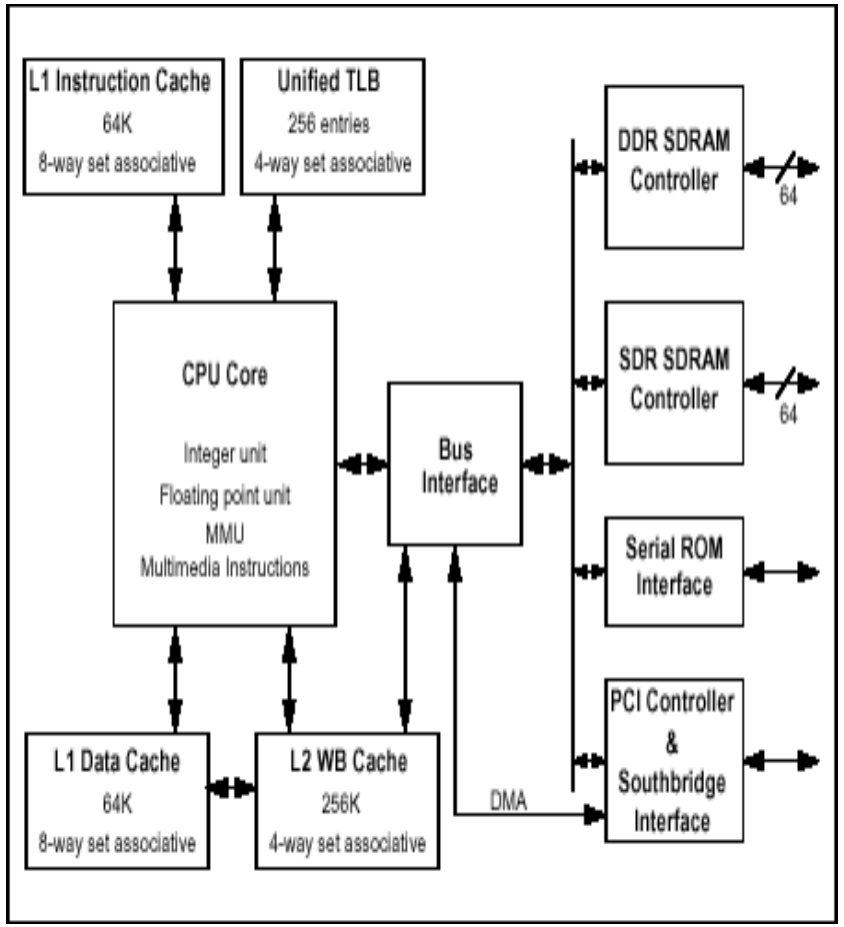
Crusoe: Características do “Code Morphing”

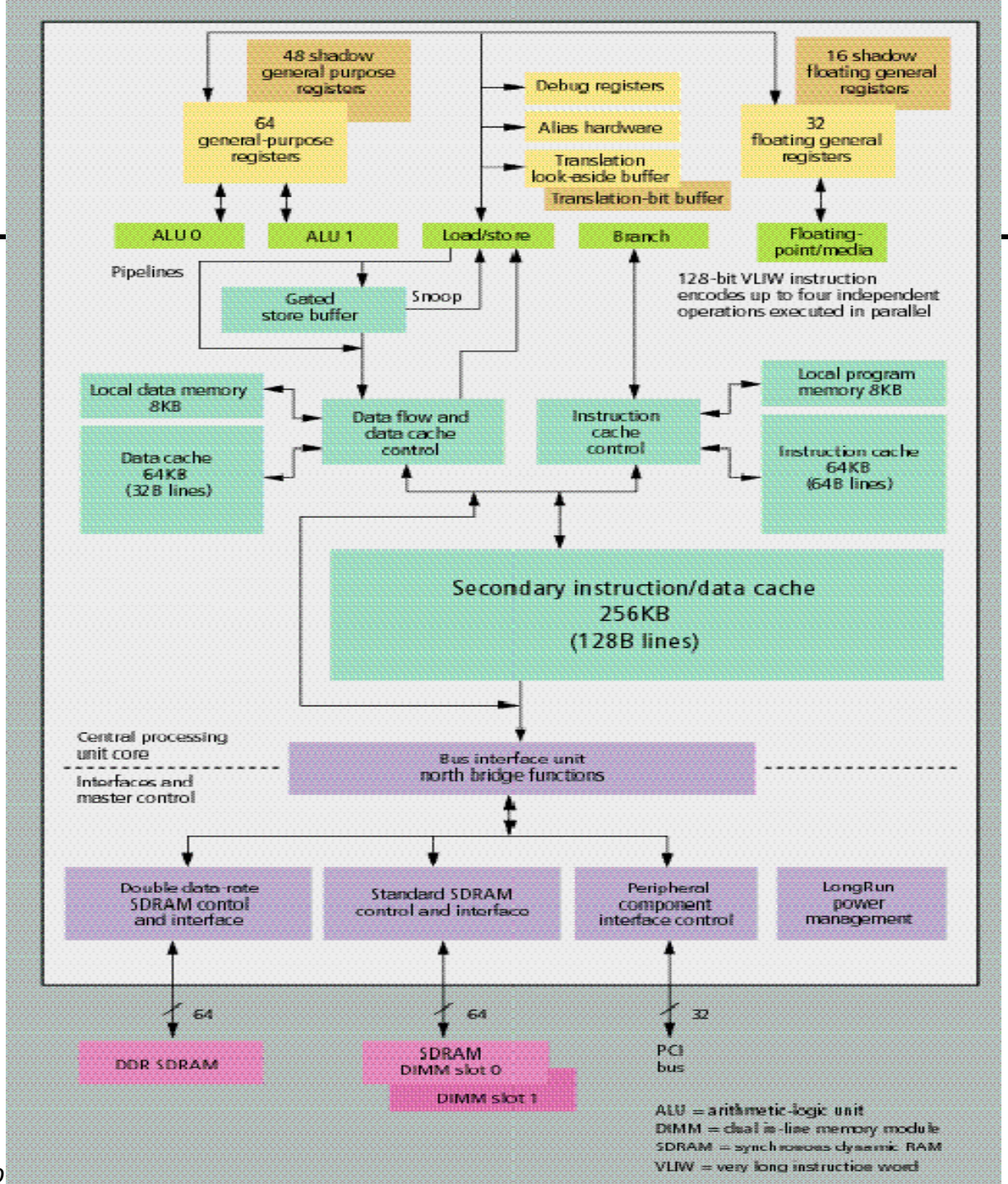
- **Pode traduzir um grupo inteiro de operações “x86” simultaneamente**
  - **processadores “nativos” traduzem (para microcódigo) uma instrução por vez**

- **As instruções traduzidas são armazenadas em uma memória *cache* específica → *translation cache***
  - **traduções são realizadas apenas “uma” vez durante a execução do código**
  - **o tempo de tradução e o consumo de potência são amortizados através das sucessivas execuções**
    - **O que permite emprego de algoritmos para traduções e escalonamento mais “sofisticados”**
  - **somente possível quando não ocorrem falhas na *translation cache***
  - **o tamanho da *translation cache* é determinado pelo sistema operacional**

TM5400 & TM3120






48 shadow general purpose registers
64 general-purpose registers
Debug registers
Alias hardware
Translation look-aside buffer
Translation-bit buffer
16 shadow floating general registers
32 floating general registers
ALU 0
ALU 1
Load/store
Branch
Floating-point/media
Pipelines
128-bit VLIW instruction encodes up to four independent operations executed in parallel
Gated store buffer
Snoop
Local data memory 8KB
Data cache 64KB (32B lines)
Data flow and data cache control
Instruction cache control
Local program memory 8KB
Instruction cache 64KB (64B lines)
Secondary instruction/data cache 256KB (128B lines)
Central processing unit core
Interfaces and master control
Bus interface unit north bridge functions
Double data-rate SDRAM contol and interface
Standard SDRAM control and interfaco
Peripheral component interface control
LongRun power management
64
64
32
DDR SDRAM
SDRAM DIMM slot 0
DIMM slot 1
PCI bus
ALU = arithmetic-logic unit
DIMM = dual in-line memory module
SDRAM = synchronous dynamic RAM
VLIW = very long instruction word

FIM