Revisão – riscos de controle

- Desvio incorre uma bolha se:
  - ⋆ circuito de decisão seja movido para DECOD e
  - ⋆ cálculo do endereço de destino seja também movido para DECOD
- Previsão estática de desvios – prevê não-tomado ou prevê tomado
- Desvios atrasados – nem sempre é possível executar trabalho útil
- Previsão dinâmica de desvios – prevê com comportamento dinâmico
  - ⋆ tabela de histórico de desvios – se desvia ou não
  - ⋆ tabela de destinos – para onde desvia (qual instrução no destino)
- Interrupções e excessões complicam muito projeto de pipelines
  - ⋆ precisão é útil para SO, mas cara em termos de hardware

Desempenho da Segmentação

$$\text{ganho} = \frac{\text{CPI sem pipeline}}{\text{CPI com pipeline}} \times \frac{\text{ciclo sem pipeline}}{\text{ciclo com pipeline}}$$

Desempenho ótimo: ganho $\approx$ número de estágios

Causa das Perdas:

- dependências estruturais — escalonamento de instruções resolve riscos
- dependências de dados — escalonamento e adiantamento resolvem riscos
- dependências de controle — previsão resolve $\approx$ 85-95% dos casos
- Lei de Amdahl: ganho é limitado pelo pior componente

Superpipelining

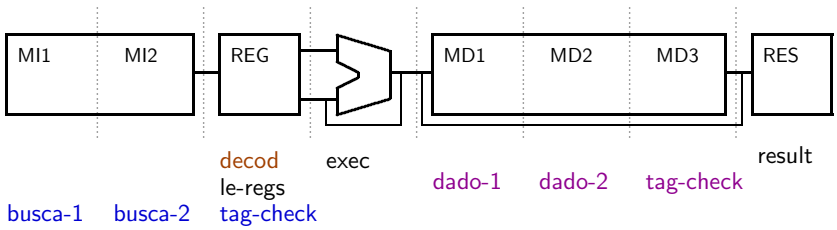
tempo de CPU = núm instr x CPI x período do relógio

Se aumentar velocidade do relógio (e reduzir período),
pode haver ganho de desempenho
→ implementar estágios com menor latência e em maior número
→ reduz período mas aumenta CPI

Com um pouco de sorte (uh?) ↗frequência compensa CPI↗

Exemplos:
MIPS 4000 usa dois estágios na busca e três em MEM/cache
Pentium III tem pipeline com 10 estágios e
Pentium IV tem pipeline com 20-22 estágios,
com dois ciclos só para transmitir bits através da CPU

Superpipelining – MIPS R4000



busca dura **2** ciclos – verifica acerto na L1-I enquanto decodifica
desvios tomados causam três bolhas

mem dura **3** ciclos por causa da verificação de acerto na cache
→ não pode escrever valor até estar certo do acerto na L1-D
só pode usar valor de `load` depois de dois ciclos
usa valor do `load` *antes de verificar etiqueta?*

Superpipelining

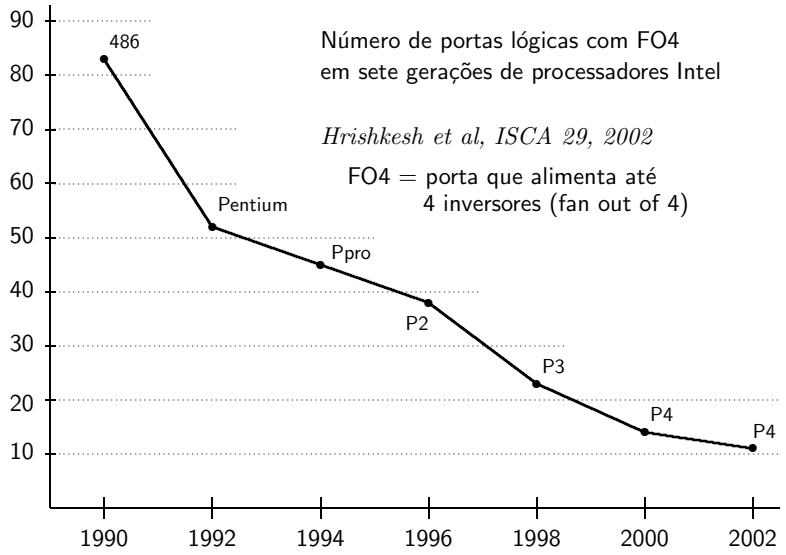
Ganhos:

freqüência do relógio mais alta — moda até há pouco
permite ligar CPU rápida à memória lenta — interf c/ mem segmentada

Perdas:

CPI mais alto
penalidade maior nos desvios
penalidade maior nas faltas nas caches (mais ciclos)
penalidade maior no uso do valor de `load`
penalidade maior nas excessões
maior complexidade:
circuitos de adiantamento, bloqueios...

Superpipelining



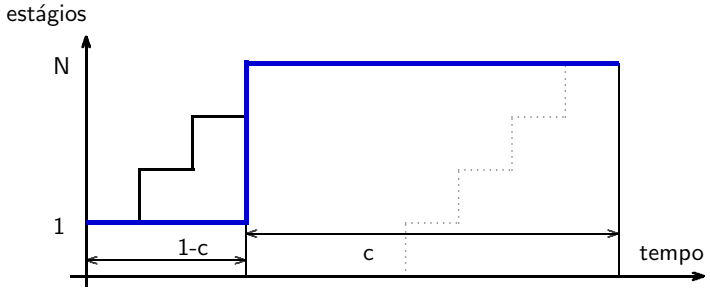
Número de portas lógicas com FO4
em sete gerações de processadores Intel

Hrishkesh et al, ISCA 29, 2002

FO4 = porta que alimenta até
4 inversores (fan out of 4)

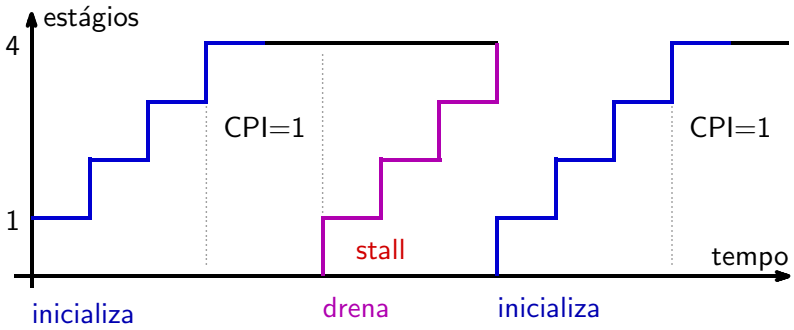
Lei de Amdahl

$$\lim_{g \to \infty} \frac{1}{(1-c) + c/g} = \frac{1}{(1-c)}$$



- quando c é um pouquinho menor que 100%, a queda no desempenho é enorme!
- a fração $1 - c$ deve ser minimizada

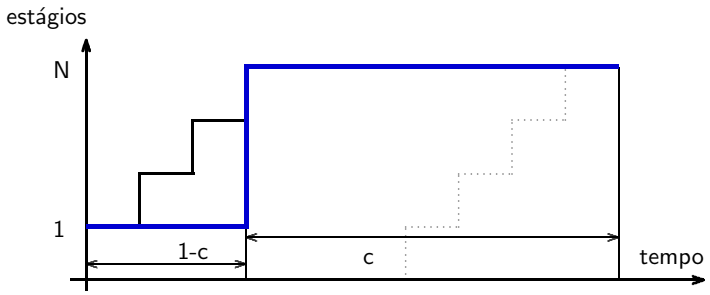
Desempenho da Segmentação I



Desempenho da Segmentação II

Se executar mais de uma instrução por ciclo,
a fração $1 - c$ pode ser reduzida pelo fator de escalaridade S

$$\text{Ganho} = \frac{1}{\frac{(1-c)}{S} + \frac{c}{g}}$$



Segmentação $\rightarrow$ CPI ≤ 1

Gargalo de Flynn

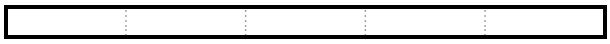
* emissão de uma instrução por ciclo limita CPI=IPC=1
* riscos + overhead $\rightarrow$ CPI≥ 1 (IPC≤ 1)
* ganhos cada vez menores com super-pipelining (# estágios $\gg$5)

solução: emitir mais de uma instrução por ciclo

	1	2	3	4	5	6	7	8	9
inst0	B	D	E	M	R				
inst1	B	D	E	M	R				
inst2		B	D	E	M	R			
inst3		B	D	E	M	R			

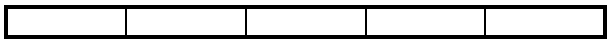
Escalar, Superescalar

escalar, multiciclo



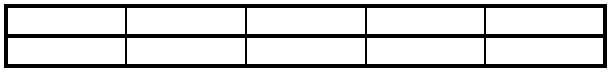
IPC $\approx 1/5$

escalar, segmentado



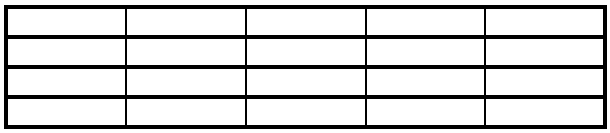
IPC ≤ 1.0

superescalar



IPC ≤ 2.0

superescalar largo



IPC ≤ 4.0

IPC $= 1/\text{CPI}$

Paralelismo no nível de instrução (PNI)

PNI é uma propriedade do *software* (e não do *hardware*)
P: Quanto paralelismo existe entre as instruções de um programa?
R: Depende MUITO do programa

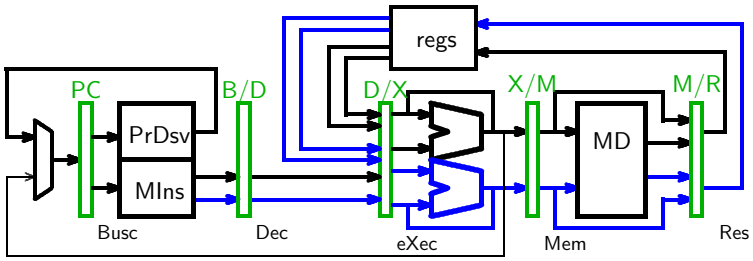
Inúmeras maneiras de explorar PNI:

- *pipelining*
- superescalar
- execução fora de ordem (ci312 em 201?-1)
- escalonamento pelo compilador (ci312 em 201?-1)

Superescalar: implementação base

- superescalar com escalonamento estático, em-ordem
  - ▷ executa programas seqüenciais sem modificação
  - ▷ descobre sozinho o que pode ser executado em paralelo
  - ▷ exemplos: Sun UltraSPARC, Alpha 21164, Pentium

Processador com 5 estágios e emissão dupla



- o que é necessário para
  - ⋆ buscar duas instruções por ciclo?
  - ⋆ decodificar duas instruções por ciclo?
  - ⋆ executar duas instruções de ALU no mesmo ciclo?
  - ⋆ acessar a cache de dados duas vezes no mesmo ciclo?
  - ⋆ escrever dois registradores no mesmo ciclo?
- e se forem 4 ou 8 instruções num ciclo?

Busca com largura N

- O que é necessário na busca de N instruções em um ciclo?
- se as instruções são seqüenciais **e**
  - ∗ no mesmo bloco (**k**) da cache → nada
  - ∗ em blocos diferentes → cache intercalada + rede de combinação

	0	1	2	3	4	5	6	7			
bloco k	x	y	z	w					→	xyzw	
bloco i			x	y							rede combina
bloco j					z	w			→	xyzw	os 2 blocos

- se as instruções não são seqüenciais saltos e desvios
  - ∗ dois acessos em série: acesso1 → prevê_destino → acesso2
- desvios no meio de um bloco: fácil se desvios não-tomados (NT)
  - ∗ acesso serial + previsão em paralelo
  - ∗ se previsão é tomado (T) → descarta instrs após desvio

Decodificação com largura N

- O que é necessário para decodificar N instruções em um ciclo?
- decodificar as instruções
  - fácil se instrs tem tamanho fixo (múltiplos decodificadores ||s)
  - difícil, porém possível, se tamanho variável x86
- ler operandos dos registradores
  - 2N portas de leitura no bloco de registradores
  - na verdade, menos que 2N porque muitos valores são adiantados
- como fica a lógica de controle dos atrasos (*stalls*)?

Decodificação com largura N

- lógica de controle de atrasos com segmentação simples:
  - `rs1(D)==rd(DE) || rs1(D)==rd(EM) || rs1(D)==rd(MR)`
  - mesmo para `rs2`
  - com adiant completo: `rs1(D)==rd(DE) && opc(DE)==load`
- dobrando a largura de emissão, quadruplica lógica de atrasos
  - ⋆ não são só 2 instruções em Decod, mas 2 instr em todos estágios
  - ⋆ `rs1(D1)==rd(DE1) && opc(DE1)==load`
  - ⋆ `rs1(D1)==rd(DE2) && opc(DE2)==load`
  - ⋆ repetir para `rs1(D2)`, `rs2(D1)`, `rs2(D2)`
  - ⋆ testar dependência da segunda instr na primeira: `rs1(D2)==rd(D1)`
- num processador de largura N, circuito de atrasos cresce com N^2
  mesmo vale para adiantamento, só que pior...
- **|lógica de controle de atrasos|** $\propto N^2$

Execução com largura N

- O que é necessário para executar N instruções em um ciclo?
- múltiplas unidades funcionais. N de cada tipo?
  - ⋆ N ULAs? Pode ser, ULAs são pequenas
  - ⋆ N divisores de ponto flutuante? Não, circuito é enorme e divPF é infreqüente
- tipicamente, usa combinação proporcional ao uso
  - ⋆ RS/6000: 1 ULA/endereços/desvios + 1 ULA-PF
  - ⋆ Pentium: 1 ULA complexa + 1 ULA simples
  - ⋆ PentiumII: 1 ULA-PF + 1 ULA + 1 load + 1 store + 1 desvios
  - ⋆ Alpha 21164: 1 ULA-PF + 1 desvios + 2 ULA + 1 load/store
- **circuito de adiantamento** $\propto N^2$
  - lógica de controle é pequena porque variáveis tem 5 bits (regs)
  - circuitos de dados são gigantes (32 ou 64 bits por caminho)
  - layout da fiação é infernal, MUXes são enormes e lentos
  - menos horrível se agrupar unidades funcionais em clusters

Interface de memória com largura N

- O que é necessário para acessar memória 2 vezes no mesmo ciclo?
- cache de dados com múltiplos bancos (detalhes mais tarde)
  - ▷ necessita lógica de detecção de conflitos (2 refs ao mesmo banco)
  - ▷ necessita lógica de detecção de riscos RAW na memória
- aproximadamente 20% das instruções são `loads` e 15% `stores`
  - ▷ para largura N, são necessárias
    0,2N portas de leitura e
    0,15N portas de escrita na memória

Gravação de resultados com largura N

- O que é necessário para escrever 2 registradores no mesmo ciclo?
- apenas mais uma porta de escrita no bloco de registradores
  - ▷ preparação (dependências) já ocorreu nos estágios anteriores
- MAS o tratamento de excessões é ainda mais complicado
  deve-se usar buffer de re-ordenação à lá Smith&Plezkun (ci312)

Processador Super-escalar

Segmento de busca puxa duas instruções da memória

Estágio de decodificação escolhe tuplas de instruções
e as despacha de acordo com disponibilidade de unidades funcionais

Inter-travamento entre estágios e unidades funcionais
resolve dependências de dados e de controle

Algoritmo do Placar ou Algoritmo de Tomasulo: detalhes em ci312
resolvem "problema dos gladiadores"
→ que gladiador luta contra qual, e quando

Unidade de reordenamento enfilera resultados de acordo com
dependências entre resultados e operandos

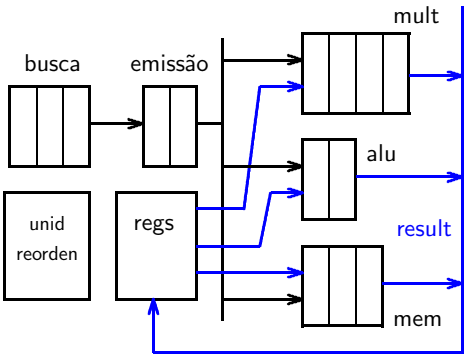
Execução Especulativa

Se emite ≥ 4 instruções/ciclo, quase sempre há um desvio entre elas;
$\rightarrow$ dependências de controle são resolvidas com
execução especulativa

Instruções nos 2 lados do desvio são executadas especulativamente;
$\rightarrow$ quando decide, anula efeito das instruções do caminho errado

Registradores "invisíveis" mantém valores da execução especulativa;
quando resolve desvio copia de regs invisíveis para regs visíveis, e
atualiza estado do processador

Segmentação não-linear

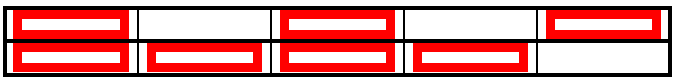


Processadores de alto desempenho são organizados com um
front end que busca, decodifica e emite as instruções,
unidades funcionais (segmentadas) que operam em paralelo, e a
unidade de reordenamento armazena resultados das instruções

Ganhos pequenos com superescalaridade?

Utilização das unidades funcionais é baixa
por causa das dependências de dados e de controle.

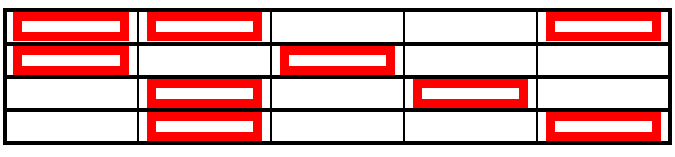
superescalar



$IPC < 2$

$U < 1$

superescalar largo



$IPC \ll 4$

$U \ll 1$

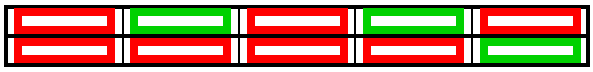
Lei de Amdahl: aumentar largura (ou profundidade) não adianta!!!

Threads, Simultaneous Multithreading

Solução: Mais de um thread na CPU para aumentar utilização

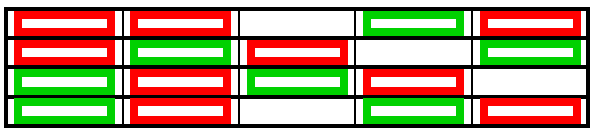
. *thread* = linha de execução

superescalar



$IPC \ll 2$
$IPC \ll 2$
$U \approx 1$

superescalar largo



$IPC \ll 4$
$IPC \ll 4$
$U \approx 1$

Quando um thread bloqueia (por dependência ou acesso à memória), executa instruções de outro thread.
Truque: instruções marcadas com nome do thread `threadId`

Arquitetura x86

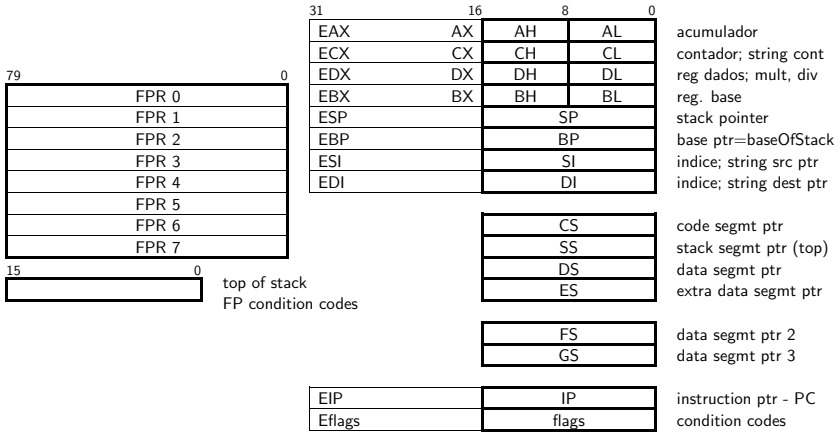
The x86 isn't all that complex—it just doesn't make a lot of sense
Mike Johnson, projetista da família x86 da AMD

80x86 foi projetado na era da memória cara e a escolha de projeto foi usar uma codificação compacta – instruções populares são curtas

Conjunto de instruções complexo CISC = Complex Instruction Set Computer

- instruções de tamanho variável – 1 a 17 bytes
- instruções complexas
- operandos em memória
- poucos registradores
- uso de registradores idiossincrático
- memória virtual com segmentação e paginação

x86 – conjunto de registradores



Esta figura ignora as extensões para 64 bits...

x86 – modos de endereçamento I

Instruções com dois operandos;
um dos operandos pode estar em memória

```
add R1, R2     # R1 <- R1 + R2
```

oper-1/resultado	operando-2
registrador	registrador
registrador	imediato
registrador	memória
memória	registrador
memória	imediato

x86 – modos de endereçamento II

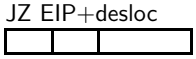
modo	**endereço efetivo**
indireto a registrador	em registrador
base-deslocamento	reg + deslocamento
base+índice∗escalar	reg + (índice∗$\mathbf{2}^{\text{escalar}}$)
base-desloc+índice∗escalar	reg + (índice∗$\mathbf{2}^{\text{escalar}}$+desloc)
escalar $\in\{0,1,2,3\}$, deslocamento de 8 ou 32-bits	

x86 – tipos de instruções

controle	
JMP	salto incondicional; ender = EIP+desloc$_{[8,16]}$
JNZ, JZ	salta se condição; ender = EIP+desloc$_{[8]}$
CALL	ender = EIP+desloc$_{[16]}$; end. retorno na pilha
RET	desemplilha e salta
LOOP	decrementa ECX e desvia se ECX≠0
movimentação de dados	
MOV	entre regs e regs & memória
PUSH,POP	empilha e desempilha
lógica e aritmética	
ADD,SUB	formato reg-memória
INC,DEC	increm., decrem. operando; formato reg-memória
CMP	compara operandos; formato reg-memória
RCR	gira p/ dir. com 'carry' no bit-maisSignif.
CBW	converte byte em palavra$_{[16]}$ no EAX
cadeias de caracteres	
MOVS	move de *ESI para *EDI (em loop)
LODS	move elmto$_{[8,16,32]}$ em cadeia p/ EAX

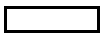
x86 – codificação das instruções I

JZ EIP+desloc



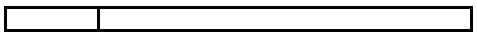
je cond. desloc.

PUSH ESI



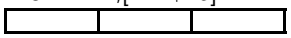
push-reg

CALL desloc



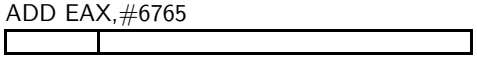
call deslocamento

MOV EBX,[EDI+45]



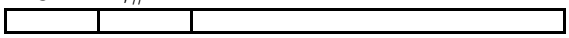
mov-d,w pos-byte desloc.

ADD EAX,#6765



add-reg,w imediato

TEST EDX,#42

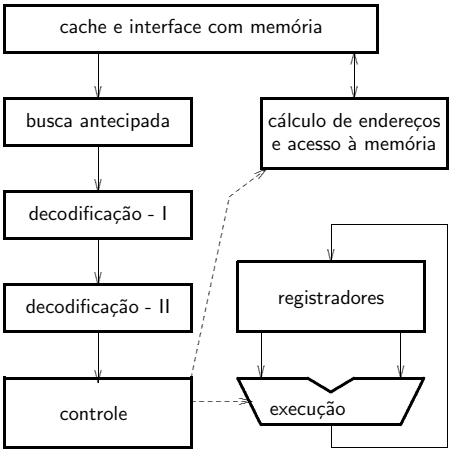


test-w pos-byte imediato

maior instrução com 17bytes:

prefixo$_8$
+opcode$_{16}$
+pós-byte$_8$
+imediato$_{32}$
+deslocamento$_{32}$

80486 – diagrama de blocos



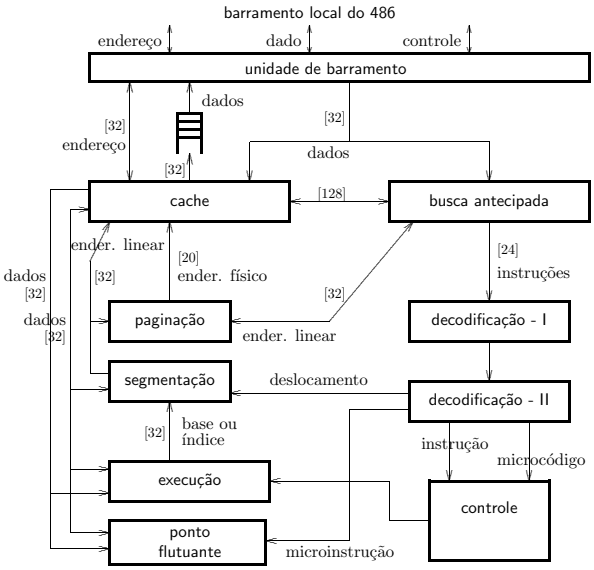
80486 foi a primeira implementação segmentada do x86;

instruções simples completam em um ciclo;

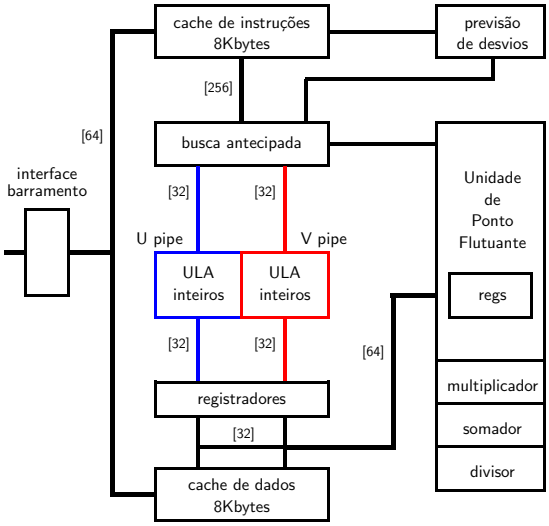
instruções complexas são interpretadas em microcódigo

Projeto de 1989

80486 – organização



Pentium – diagrama de blocos



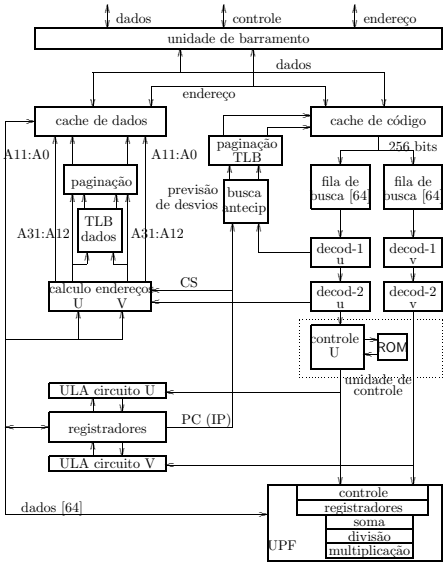
Implementação super-escalar

Pentium emite até 2 instr por ciclo; 2 simples, ou 1 complexa (U) e 1 simples (V).

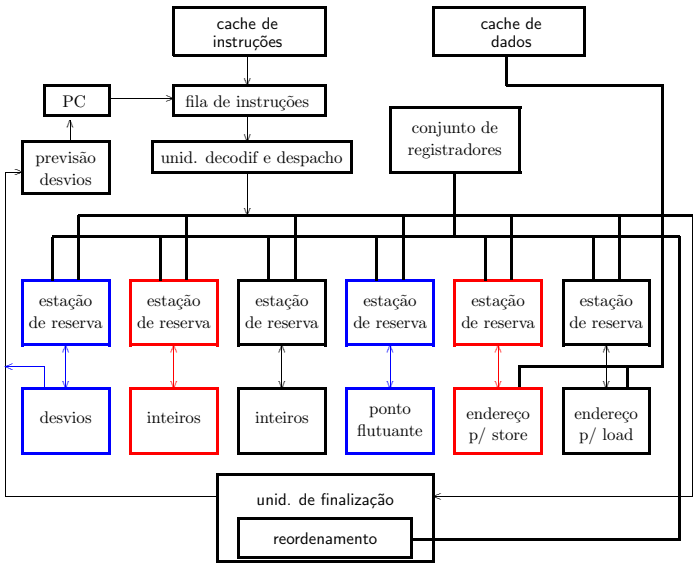
Instruções complexas são interpretadas em microcódigo.

Projeto de 1993.

Pentium – organização



PIII e PIV – diagrama de blocos



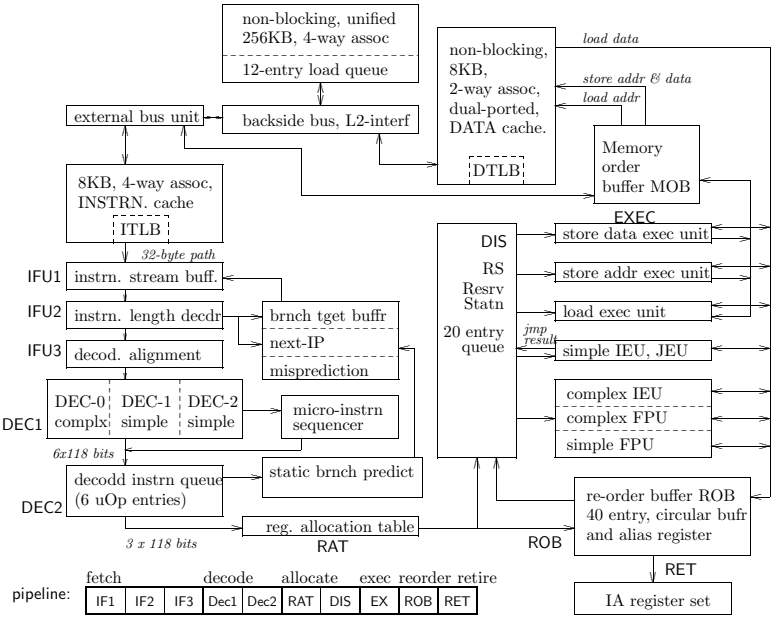
PIII e PIV – organização

Processador super-escalar com emissão múltipla tenta ≈ 4 instr/ciclo
Instruções do x86 são traduzidas para "instruções internas" (RISC)
e então executadas;
Instruções complexas são interpretadas com microcódigo

Escalonamento dinâmico de instruções com ≈Algoritmo de Tomasulo
instruções são emitidas, e completam, fora-de-ordem

segmentos do PIII
3 ciclos na busca
2 na decodificação
2 na alocação de registradores e emissão para unidades funcionais
$\geq$1 ciclos executando
2 ciclos para retirar e completar

no P4, são 22 estágios, 2 só para mover bits através do CI



Resumo – processadores superescalares

- recursos replicados para aumentar paralelismo no nível de instrução
- algoritmo para garantir dependências de dados e controle
  algoritmo do placar *(scoreboard)* ou de Tomasulo
- execução especulativa: executa os dois lados dos desvios
  anula instruções "do lado errado", que não escrevem em regs!!
- não percam os próximos e emocionantes episódios em ci312