"Uma heurística é um algoritmo que não funciona (exceto na prática, às vezes, talvez)" (Erickson J., Algorithms; 2019).

LRU, Caches multinível e Coerência de Cache

Paulo Ricardo Lisboa de Almeida

DSBD

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

Relembrando da aula passada...

Exemplo.

Considerando uma cache associativa de 2 vias com:

- Blocos de 4 bytes.
- Cache com capacidade para armazenar 8 blocos no total.
- Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Exemplo

Onde o byte no endereço $0000\ 1001_2$ pode ser mapeado?

Podemos mapear para qualquer bloco do conjunto!

Cache

Conjunto	Val?	Tag	Bloco	Val?	Tag	Bloco
00						
01						
10						
11						

Memória Principal

Bloco	Endereço	Palavra
0000 00	0000 0000	Byte0
	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
0000 01	0000 0100	Byte4
	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
0000 10	0000 1000	Byte8
	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
0000 11	0000 1100	Byte12
	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Problema

E se esses blocos já estiverem ocupados?

Precisamos substituir um deles.

Qual?

Cache

Conjunto	Val?	Tag	Bloco	Val?	Tag	Bloco
00						
01						
10	1	0001	Byte24Byte25 Byte26Byte27	1	0100	Byte72Byte73Byte7 4Byte75
11						

Memória Principal

Bloco	Endereço	Palavra
	0000 0000	Byte0
0000 00	0000 0001	Byte1
	0000 0010	Byte2
	0000 0011	Byte3
	0000 0100	Byte4
0000 01	0000 0101	Byte5
	0000 0110	Byte6
	0000 0111	Byte7
	0000 1000	Byte8
0000 10	0000 1001	Byte9
	0000 1010	Byte10
	0000 1011	Byte11
	0000 1100	Byte12
0000 11	0000 1101	Byte13
	0000 1110	Byte14
	0000 1111	Byte15
	0001 0000	Byte16
	…	…

Qual bloco substituir?

Qual bloco substituir?

Poderíamos selecionar aleatoriamente.

Funciona, mas pode não ser uma boa ideia.

Podemos remover um bloco que está sendo usado o tempo todo na cache.

Poderíamos pensar em lógicas sofisticadas para isso.

Bloco mais distante dos seus vizinhos, menos acessado, mais distante da instrução sendo executada, uma junção de todas essas métricas, ... ?

Pode nos levar a decisões melhores.

Problemas?

Qual bloco substituir?

Qual bloco substituir?

Poderíamos selecionar aleatoriamente.

Funciona, mas pode não ser uma boa ideia.

Podemos remover um bloco que está sendo usado o tempo todo na cache.

Poderíamos pensar em lógicas sofisticadas para isso.

Bloco mais distante dos seus vizinhos, menos acessado, mais distante da instrução sendo executada, uma junção de todas essas métricas, ... ?

Pode nos levar a decisões melhores.

Pode custar muito tempo e hardware para tomar essa decisão!

Qual bloco substituir?

Necessária uma solução com um bom custo x benefício.

Que seja pelo menos melhor que uma seleção desinformada, e que custe pouco tempo e hardware.

LRU - Least recently used

LRU - Least recently used (O usado menos recentemente).
Remover o bloco que teve o acesso mais antigo.
Esquema comumente encontrado nas CPUs.

Em uma cache associativa de 2 vias é relativamente simples de se implementar.

Exemplo:

Podemos manter um “bit de uso” em cada bloco do conjunto da cache.
Toda vez que um bloco do conjunto é acessado:
Seu bit de uso é setado.
O bit de uso do outro bloco é resetado.

Exemplo

Bits de uso em uma cache associativa de 2 vias com capacidade para 8 blocos.

Se, por exemplo, precisarmos substituir um bloco do conjunto 1, uma escolha razoável é o segundo bloco do conjunto, pois foi acessado menos recentemente.

Conjunto	Val?	Uso	Tag	Bloco	Val?	Uso	Tag	Bloco
00		0				1		
01		1				0		
10		1				0		
11		1				0		

LRU - Least recently used

Obviamente, seja qual a estratégia implementarmos, não podemos garantir a melhor escolha.

Não podemos prever o futuro.

Mas técnicas mais informadas tendem a diminuir a chance de tomar uma decisão ruim.

LRU - Least recently used

Em uma cache associativa de 2 vias podemos manter um bit para cada bloco do conjunto.

E como fazer para uma cache associativa de 4 vias?

E de 8 vias? 12 vias? ...

LRU - Least recently used

Para um número suficientemente grande de vias, as coisas se complicam.

Necessário manter muitos bits, e técnicas de atualização mais complicadas para saber quem é o bloco acessado menos recentemente no conjunto.

Por isso, mesmo caches simples com apenas 4 vias podem implementar alguma aproximação do LRU.

LRU - Least recently used

Aproximação simples para o LRU para uma cache associativa de n vias, onde n›2.

Manter um bit de uso para cada bloco.

Quando um bloco é acessado:

- O seu bit de uso é setado.
- Os bits de uso de todos os demais blocos é resetado.

Agora sabemos **qual o bloco usado mais recentemente**.

- Mas **não sabemos** exatamente qual o bloco usado menos recentemente.
- Sabemos apenas que os demais não são "a pior escolha".
- Nesse caso podemos escolher aleatoriamente entre esses blocos.

Técnicas mais sofisticadas (+ caras e + complexas).

- Manter uma estrutura de árvore para decidir qual o bloco mais antigo.
- Utiliza mais bits, mas leva a uma aproximação melhor do LRU.

LRU para associatividades “grandes”

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Número de misses para cada 1000 instruções para cada técnica de substituição de bloco considerando diferentes tamanhos de cache (Henessy, Patterson; 2014).

LRU para associatividades “grandes”

Quando o nível de associatividade e o tamanho da cache são grandes o suficiente, o desempenho do LRU se aproxima de uma escolha aleatória.

Nesses casos, muitas vezes não vale a pena o custo de complexidade de se implementar um LRU.

Uma escolha aleatória se torna um melhor custo x benefício.

	Associativity								
	Two-way			Four-way			Eight-way		
Size	LRU	Random	FIFO	LRU	Random	FIFO	LRU	Random	FIFO
16 KiB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KiB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KiB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Caches Multinível

Processadores atuais utilizam múltiplos níveis de cache.

Exemplos:

	ARM Cortex-X1	AMD Ryzen 5 5800X	Intel Xeon Platinum 9282
Exemplo de Uso	Seu celular – Google Pixel 6.	Seu computador pessoal.	Servidores e clusters de alto desempenho (2019).
Cache L1	32 KiB dados + 32 KiB instr. por core.	32 KiB dados + 32 KiB instr. por core (512 KiB)	32 KiB dados + 32 KiB instr. por core (3,5 MiB)

Caches Multinível

Processadores atuais utilizam múltiplos níveis de cache.

Exemplos:

	ARM Cortex-X1	AMD Ryzen 5 5800X	Intel Xeon Platinum 9282
Exemplo de Uso	Seu celular – Google Pixel 6.	Seu computador pessoal.	Servidores e clusters de alto desempenho (2019).
Cache L1	32 KiB dados + 32 KiB instr. por core.	32 KiB dados + 32 KiB instr. por core (512 KiB)	32 KiB dados + 32 KiB instr. por core (3,5 MiB)
Cache L2	1 MiB por core	4 MiB	1 MiB por core (56 MiB)

Caches Multinível

Processadores atuais utilizam múltiplos níveis de cache.

Exemplos:

	ARM Cortex-X1	AMD Ryzen 5 5800X	Intel Xeon Platinum 9282
Exemplo de Uso	Seu celular – Google Pixel 6.	Seu computador pessoal.	Servidores e clusters de alto desempenho (2019).
Cache L1	32 KiB dados + 32 KiB instr. por core.	32 KiB dados + 32 KiB instr. por core (512 KiB)	32 KiB dados + 32 KiB instr. por core (3,5 MiB)
Cache L2	1 MiB por core	4 MiB	1 MiB por core (56 MiB)
Cache L3	8 MiB	32 MiB	77 MiB

Caches Multinível

Níveis mais altos mais próximos da CPU.

- Foco na redução do tempo de acesso e custo do miss.
- Geralmente de acesso exclusivo para cada núcleo.
- Comumente segmentadas entre cache de instrução e cache de dados.
  - **Arquitetura Harvard.**
- Caches menores.

Caches Multinível

Níveis mais altos mais próximos da CPU.

- Foco na redução do tempo de acesso e custo do miss.
- Geralmente de acesso exclusivo para cada núcleo.
- Comumente segmentadas entre cache de instrução e cache de dados.
  - **Arquitetura Harvard.**
- Caches menores.

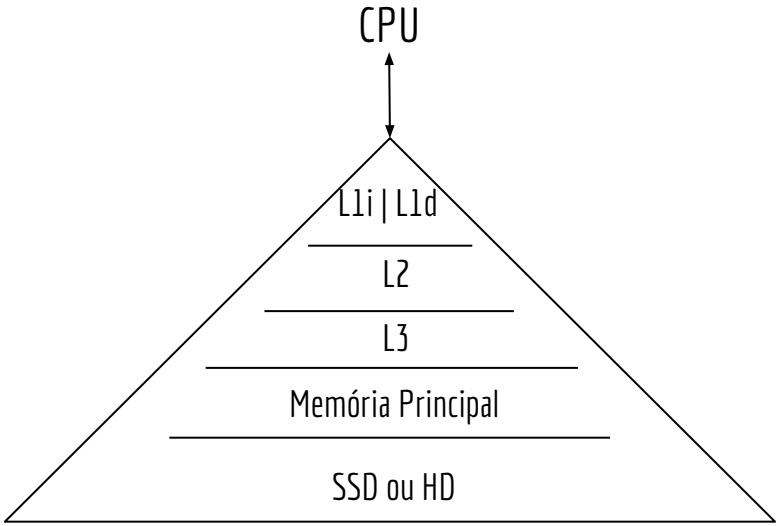
Níveis mais baixos focam na redução da probabilidade de miss.

- Caches maiores.
- Comumente compartilhadas entre os núcleos.

Caches Multinível

Exemplo:

Em caso de miss na L1, faz a carga a partir da L2.
Em caso de miss na L2, faz a carga a partir da L3
...
Sem pular níveis.



Caches Multinível

Níveis mais altos mais próximos da CPU.

Focam na redução do tempo de acesso e redução do miss penalty.

Como?

Caches Multinível

Níveis mais altos mais próximos da CPU.

Focam na redução do tempo de acesso e redução do miss penalty.

- Reduzir o custo do miss.
  - Reduzir o tamanho do bloco ou barramentos grandes para carga paralela.
- Reduzir o tempo de acesso.
  - Reduzir associatividade.
  - Caches não bloqueantes.
    - Ex.: enquanto a cache de dados gera um miss, a cache de instruções pode continuar operando.
- Cache fisicamente mais próxima do processador.
  - Caches individuais por núcleo ou compartilhadas entre poucos núcleos.

Caches Multinível

Níveis mais baixos mais distantes da CPU.

Focam na redução da probabilidade de miss.

Como?

Caches Multinível

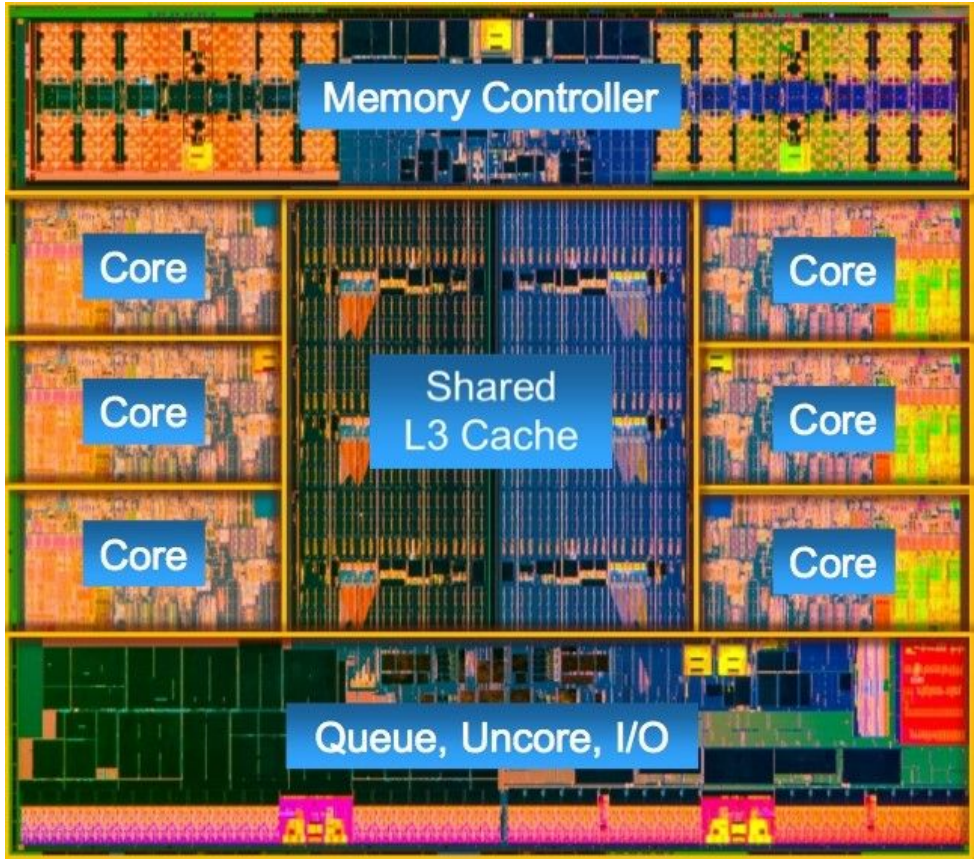
Níveis mais baixos mais distantes da CPU.

Focam na redução da probabilidade de miss.

- Maior associatividade (localidade temporal).
- Tamanhos de bloco maiores (localidade espacial).
- Caches maiores.
- Cache compartilhadas entre múltiplos núcleos.
  - Evitar dados duplicados entre núcleos para otimizar o uso do espaço na cache.
- Contar com o compilador e com o programador para organizar as instruções corretamente.

Exemplo

Core i7-4960X

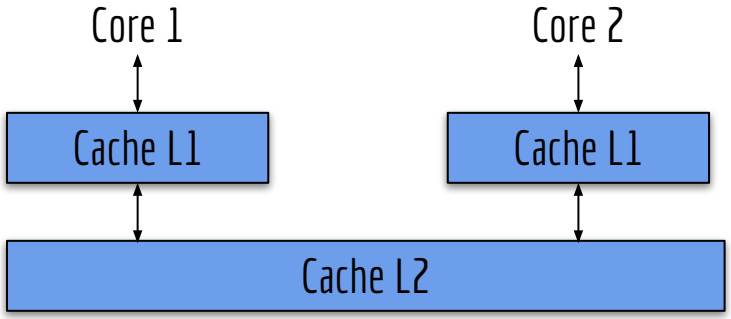


Problemas de Coerência de Cache

CPUs modernas geram problemas modernos.

Considere uma CPU de dois “núcleos”.

- Uma cache L1 exclusiva para cada núcleo.
- Uma cache L2 compartilhada entre os núcleos.
- Caches com write-back.
- Como essas CPUs podem ver versões diferentes de um mesmo dado?



Problemas de Coerência de Cache

A CPU (Core) 1 solicita um dado da L1.
- L1 não têm o dado (miss) e solicita da L2.
- O dado é copiado da L2 para a L1 da CPU 1.
- A CPU 1 modifica esse dado.
- O dado é escrito em sua L1, mas não na L2.
  - Write-back: o dado só vai ser atualizado no outro nível quando o bloco for substituído.

A CPU (Core) 2 solicita o mesmo dado da sua L1.
- L1 não tem o dado.
- O dado é copiado da L2 para a L1 da CPU 2.
  - O dado da L2 está desatualizado!

E agora como resolver?

Problemas de Coerência de Cache

Podemos tornar a cache **write-through**.

Funciona, mas já vimos que **não é eficiente**.

Protocolo de Snooping

Quando uma CPU escreve em um bloco da sua cache, envia um sinal para todas as demais CPUs via broadcast para "sujar a cache".

Uma CPU Intel por exemplo pode usar sua interconexão QPI ou UPI interna para enviar a mensagem.

Parecido com o protocolo do PCI Express estudado.

As demais CPU's e níveis de cache desligam o bit de validade do bloco caso elas possuam esse mesmo bloco.

Se duas CPU's tentam escrever ao mesmo tempo, uma delas "ganha a corrida" e envia o broadcast antes que a outra.

Protocolo de Snooping

Agora, quando as outras CPUs precisarem do dado, o bit de validade está desligado.

Efetivamente força que a CPU faça uma nova cópia desse dado.

Agora a cópia é mais complexa.

Pode vir da cache do vizinho, ou podemos forçar que a CPU que possui a cópia mais recente escreva no nível de baixo para que a CPU vizinha possa enxergar esse dado.

Depende de como implementamos o hardware do processador.

Protocolo de Snooping

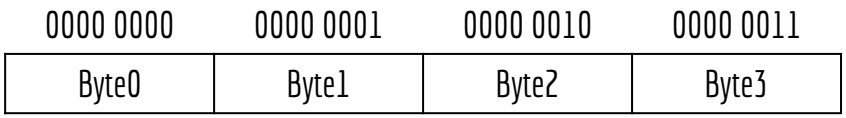
Não vamos entrar em detalhes sobre como é implementado o protocolo de Snooping.

Mas é importante que você saiba pelo menos o básico sobre o que está acontecendo.

Caso contrário você não será capaz de criar programas eficientes com mais de um processo/thread.

Faça você mesmo

Considere que temos blocos de 4 palavras, e o bloco a seguir:



Endereço na memória principal.

Se fizermos um programa que executa **em paralelo** em **duas CPUs** com a seguinte lógica.

```
CPU 1

Carregue 0000 0000 para reg 1
Carregue 0000 0001 para reg2
reg 3 = reg1 + reg2
Salve reg3 em 0000 0000
```

```
CPU 2

Carregue 0000 0010 para reg1
Carregue 0000 0011 para reg2
reg 3 = reg1 + reg2
Salve reg3 em 0000 0010
```

Esses programas compartilham algum dado em algum momento? Esse programa “paralelo” executa 2x mais rápido do que se fizéssemos tudo sequencialmente em uma única CPU?

Faça você mesmo

Os programas **não** compartilham dados.

Enquanto a CPU 1 está trabalhando com os Bytes 0 e 1, a CPU 2 trabalha com os Bytes 2 e 3.

No entanto, note que todos os dados estão no **mesmo bloco**.

Quando invalidamos algo na cache, invalidamos o bloco inteiro, e não só um pedaço do bloco.

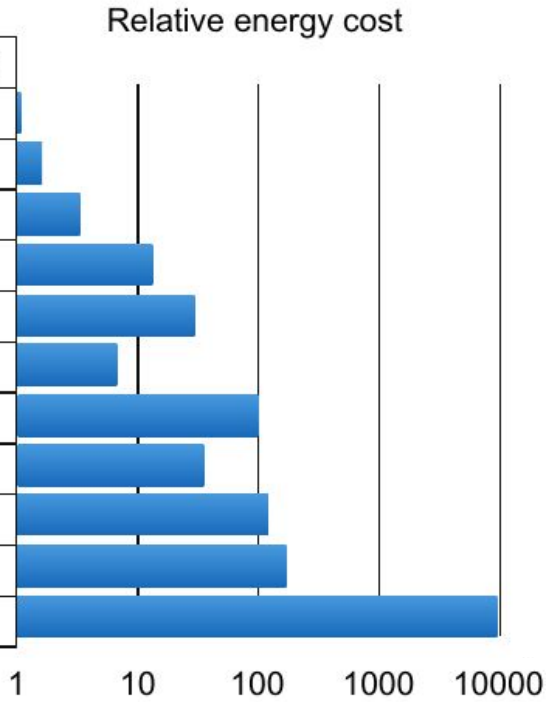
Esse problema é chamado de **falso compartilhamento**.

Esse programa provavelmente vai executar mais lento do que se fizéssemos uma versão que utiliza uma única CPU.

Na versão atual, **uma CPU atrapalha a outra**.

Custo Energético

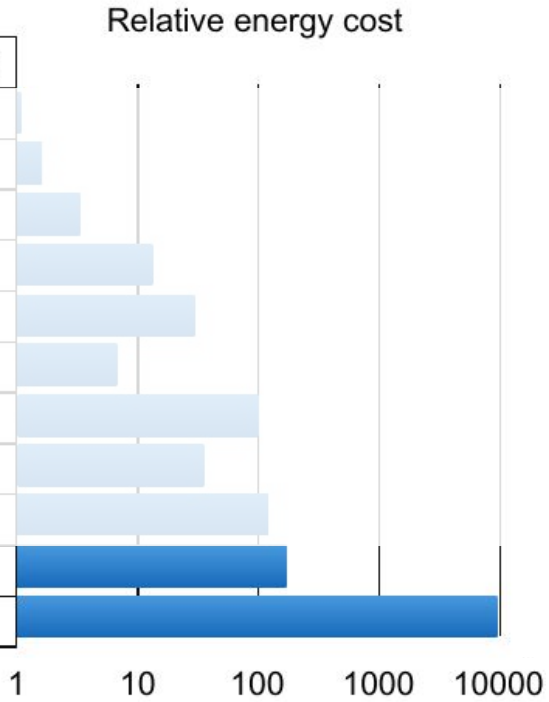
Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



J. Henessy; D. Patterson. 2019

Custo Energético

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FB Add	0.4
32b FB Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FB Mult	1.1
32b FB Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



J. Henessy; D. Patterson. 2019

Exercício

1. Para reduzir a probabilidade de falso compartilhamento, o que podemos fazer com o tamanho de blocos na cache?
2. Considere blocos de 4 bytes, e uma cache associativa de duas vias, com 4 conjuntos no total. Assumindo a cache inicialmente vazia, qual o estado final da cache se requisitarmos os bytes nos seguintes endereços (nesta ordem): $0000\ 0000_2$, $0011\ 0000_2$, $0011\ 0010_2$, $0000\ 0010_2$, $1000\ 0000_2$, $0000\ 1000_2$, $0000\ 0000_2$, $0000\ 0110_2$, e $0000\ 1000_2$. Utilize o LRU para substituir blocos quando necessário.

Exercício - Respostas

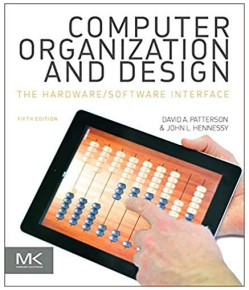
1. Para reduzir a probabilidade de falso compartilhamento, o que podemos fazer com o tamanho de blocos na cache?

Blocos menores reduzem a chance de compartilhamento de variáveis.

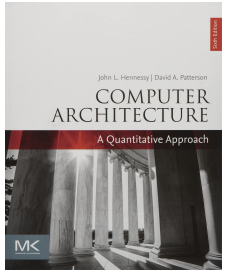
Quando você criar um programa, deve também levar em consideração o tamanho dos blocos para evitar essa situação.

Referências

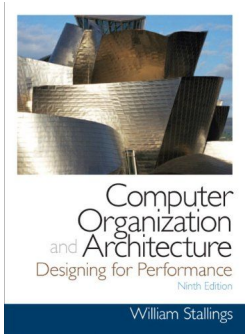
Patterson, Hennessy . Arquitetura e Organização de Computadores: A interface hardware/software. 2014.



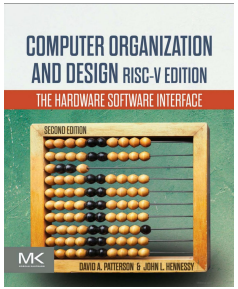
Hennessy, Patterson. Arquitetura de Computadores: uma abordagem quantitativa. 2019.



Stallings, W. Organização de Arquitetura de Computadores. 10a Ed. 2016.



Patterson, Hennessy. Computer Organization and Design RISC-V Edition: The Hardware Software Interface. 2020.



Licença