

**Pipeline:** é uma técnica que transforma o processador em uma "linha de montagem"

⚠ O ciclo é único, o tempo de clock deve ser longo e suficiente para a instrução mais lenta ser executada.

- O tempo para executar uma única tarefa completa (latência) não muda com o pipelining, o principal ganho está na vazão (throughput), um número maior de tarefas pode ser completado em um longo período, com o potencial de ser  $n$  vezes mais rápido, onde  $n$  é o número de estágios do pipeline.

- **Estágios do pipeline:**

- ↳ **IF (instruction fetch):** busca da instrução na memória
- ↳ **ID (instruction decode):** leitura dos registradores e decodificação da instrução.

- ↳ **EX (Execution):** execução da operação ou cálculo de endereço.

- ↳ **MEM (Memory Access):** acesso à memória de dados.

- ↳ **WB (Write Back):** escrita dos resultados em um registrador

- **Ajustando o tempo de clock:** com o pipelining, o tempo de clock é ajustado para o estágio mais demorado. O ganho de vazão é dado por:  $\frac{\text{Tempo de ciclo sem pipeline}}{\text{Tempo de ciclo para a unidade mais lenta}} = n$ .

Temos assim temos  $n$  vezes mais rápido com pipeline.

Sendo assim temos  $n$  vezes mais rápido com pipeline.

**Hazards:** são problemas ou riscos que surgem ao implementar o pipeline, impedindo que a próxima instrução continue sua execução no ciclo de clock seguinte.

- **Hazards estruturais:** ocorrem quando o hardware não consegue suportar duas instruções no pipeline porque elas competem por um mesmo componente. A implementação do MIPS abordada evita

a maioria dos hazards estruturais duplicando unidades funcionais que poderiam conflitar

- **Hazards de dados:** acontece quando uma instrução depende do resultado de uma instrução anterior que ainda não está pronto.

- ↳ **Forwarding (bypassing):** é uma técnica para mitigar alguns hazards de dados. Permite "emprestar" o resultado de uma unidade funcional assim que ele está pronto, mesmo antes de ser escrito no banco de registradores ou na memória principal. Prioriza-se o resultado mais recente disponível

- ↳ **Pipeline Stall:** Quando um forwarding não consegue resolver um hazard de dados, por exemplo um load seguido de uma instrução que usa um dado carregado, o processador precisa adiar a execução da próxima instrução. Isso é feito injetando uma instrução balsa (nop - no operation) no pipeline, que não realiza nenhuma operação, apenas gasta tempo.

- **Hazard de controle:** ocorre quando o pipeline não consegue determinar qual será a próxima instrução a ser executada, tipicamente em casos de desvios (branches). A CPU só sabe se o desvio será tomado após o estágio EX. Uma solução simples é assumir que o desvio nunca será tomado e continuar executando as próximas instruções, se a previsão estiver errada, as instruções carregadas no pipeline devem ser descartadas e os fluxos direcionados para o endereço correto. Pode-se usar também previsões dinâmicas de desvios, como um buffer de previsão, que armazenam se a instrução foi tomada de última vez, podendo atingir taxa de acerto de 90%.

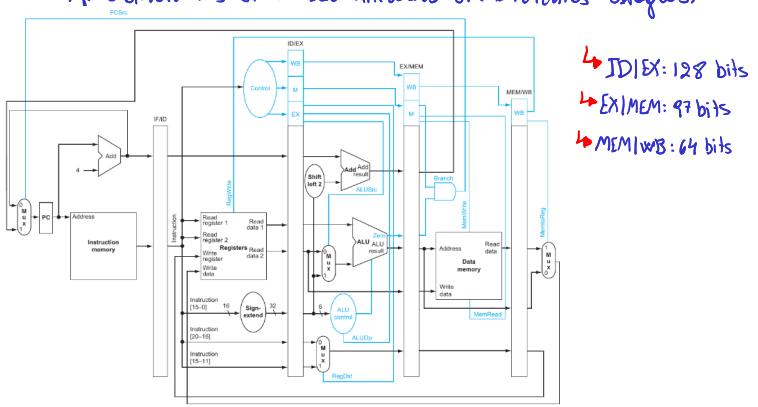
## Comunicação de dados com Pipeline

↳ **Registradores de pipeline:** O principal problema na implementação do pipeline é que a informação processada em um estágio precisa ser transferida para o próximo estágio no ciclo de clock seguinte. Para resolver isso, são introduzidos os registradores de pipeline, que armazenam todas as informações relevantes para o estágio subsequente.

↳ Eles são nomeados pela frontaria que separam, como IF/ID, ID/EX, EX/MEM, MEM/WB.

↳ **Correção de bug na escrita:** um problema inicial do design do pipeline é que o endereço dos registradores de destino para a escrita poderia se perder, resultando na escrita do resultado no estágio errado. A solução é garantir que o endereço do registrador de destino seja salvo e propagado através dos registradores de pipeline até o estágio WB, para que a instrução correta escreva no lugar certo.

↳ **Sinais de controle:** Assim como os dados, os sinais de controle também precisam ser salvos e propagados através dos registradores de pipeline, pois diferentes sinais são utilizados em diferentes estágios.



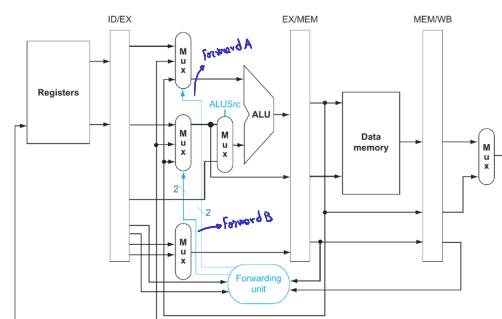
↳ **ID/EX: 128 bits**  
↳ **EX/MEM: 97 bits**  
↳ **MEM/WB: 64 bits**

## Construindo Forwardings (bypasses)

↳ Para mitigar hazards de dados, se utiliza uma técnica de Forwarding que empresta o resultado de uma unidade funcional assim que ele está pronto, mesmo antes de ser escrito no banco de registradores ou na memória principal. Prioriza-se o resultado mais recente disponível

↳ **Problema do reg. \$zero:** um cuidado especial é necessário para forcer o forwarding no reg. \$zero, pois ele deve conter o valor zero.

↳ **Unidade de Forwarding:** tem como objetivo mitigar os hazards de dados que surgem no pipeline



Sinal MUX	Fonte	Descrição
ForwardA = 00	ID/EX	Primeiro operando da ALU deve vir do banco de registradores (sem forward)
ForwardA = 10	EX/MEM	Primeiro operando da ALU deve vir de EX/MEM (forward)
ForwardA = 01	MEM/WB	Primeiro operando da ALU deve vir de MEM/WB (forward)
ForwardB = 00	ID/EX	Segundo operando da ALU deve vir do banco de registradores (sem forward)
ForwardB = 10	EX/MEM	Segundo operando da ALU deve vir de EX/MEM (forward)
ForwardB = 01	MEM/WB	Segundo operando da ALU deve vir de MEM/WB (forward)

↳ **Stalls causados por hazards:** não é possível solucionar qualquer hazard de dados através de forwardings, a única combinação que causaria stalls é operação de lw seguida de alguma instrução que usa o conteúdo do registrador carregado.

↳ **Código da unidade de forwarding:**

```

ForwardA = "00"
ForwardB = "00"
stall = False

if ID_EX.MemRead and (
    (ID_EX.RegisterRt == IF_ID.RegisterRs) or
    (ID_EX.RegisterRt == IF_ID.RegisterRt)
):
    stall = True

if not stall:
    if EX_MEM.ReqWrite and (EX_MEM.RegisterRd != 0) and (EX_MEM.RegisterRd == ID_EX.RegisterRs):
        ForwardA = "10"
    if EX_MEM.ReqWrite and (EX_MEM.RegisterRd != 0) and (EX_MEM.RegisterRd == ID_EX.RegisterRt):
        ForwardB = "10"
    if MEM_WB.ReqWrite and (MEM_WB.RegisterRd != 0) and (MEM_WB.RegisterRd == ID_EX.RegisterRs) and not (
        EX_MEM.ReqWrite and (EX_MEM.RegisterRd != 0) and (EX_MEM.RegisterRd == ID_EX.RegisterRs)
    ):
        ForwardA = "01"
    if MEM_WB.ReqWrite and (MEM_WB.RegisterRd != 0) and (MEM_WB.RegisterRd == ID_EX.RegisterRt) and not (
        EX_MEM.ReqWrite and (EX_MEM.RegisterRd != 0) and (EX_MEM.RegisterRd == ID_EX.RegisterRt)
    ):
        ForwardB = "01"

```

### Soluções para Hazards de Controle

↳ **Assumir que o desvio nunca é tomado:** a CPU carrega os próximos指令es, se essa "previsão" estiver errada, as instruções que já foram inseridas na pipeline após o desvio devem ser descartadas e o fluxo de instrução é redirecionado para o endereço correto. Para descartar as instruções, a unidade de controle pode injetar NOP no estágio IF e zerar os sinais de controle para as instruções que estão no estágio ID e EX, três ciclos são inutilizados.

↳ **Reducindo atrasos:** é possível antecipar o cálculo de desvio para um estágio anterior da pipeline para o estágio ID. Isso envolve mover os componentes responsáveis pelo cálculo de endereço de salto e pela comparação dos registradores para o estágio ID. Com essa modifica-

ção, o custo de uma previsão incorreta é reduzido para uma stall. Podemos usar um XNOR, reg1 XNOR reg2 retorna 1 se reg1 == reg2. No entanto, essa otimização pode gerar hazards de dados, pois os operandos de branch podem ser resultados de instruções que estão nos estágios EX, MEM ou WB.

↳ **Previsão dinâmica de desvios:** são sistemas que tentam "aprender" se os desvios serão tomados ou não.

↳ **Buffer de previsão de desvios:** é uma pequena memória que armazena uma tabela com parte do endereço de instrução e um bit indicando se o desvio foi tomado na última vez que essa instrução foi executada. Isso é útil para desvios em loops. O buffer pode ser implementado no estágio IF da pipeline, esquemas mais sofisticados utilizam mais de um bit para previsão.

Nesse caso, utilizamos 3 bits (do 3º ao 5º bits menos significativos) para endereçar

Endereço (binário)	Endereço	Desviar?
0000 0000 0000 0000	000	0
0000 0000 0000 0100	001	0
0000 0000 0000 1000	010	1
0000 0000 0000 1100	011	0
0000 0000 0001 0000	100	1
0000 0000 0001 0100	101	1
0000 0000 0001 1000	110	0
0000 0000 0001 1100	111	1
...		
0000 0000 0010 0000		
0000 0000 0010 0100		
0000 0000 0010 1100		

Temos instruções competindo pelo mesmo lugar no buffer.

↳ **Branch Delayed Slot:** coloca uma instrução logo abaixo de um branch que é sempre executada, independente do desvio.

**Exceções e Interrupções:** são eventos imprevistos que, assim como desvios e saltos, podem alterar o fluxo normal de um programa.

↳ **No MIPS:**

↳ **Exceção:** é um evento interno inesperado dentro do processador.

↳ **Interrupção:** é um evento inesperado gerado externamente (Ex: input).

**Tratando uma exceção:** no caso de um overflow aritmético, os passos básicos para um tratamento incluem:

1. Salvar o endereço da instrução que causou a exceção em um registrador especial (EPC - Exception PC)
2. Salvar um código informando a causa da exceção, reg. cause no MIPS
3. Transferir a execução para o sistema operacional, que verifica a cause para definir a ação, tratar ou encerrar.

↳ **Interrupções autorizadas:** são uma forma alternativa de tratamento onde o controle é transferido para um endereço específico, dependendo da fonte de exceção, eliminando a necessidade de cause.

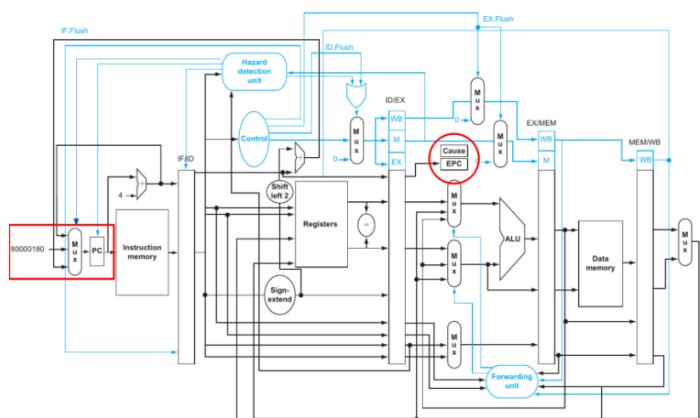
↳ **Vantagens:** tratamento mais rápido, pois a SO não precisa definir o que fazer.

↳ **Desvantagens:** requer mais hardware (tableta de desvios) e oferece menor flexibilidade.

↳ **X86:** usa utilze interrupções autorizadas.

↳ **MIPS:** usa o reg. cause, com exceção a cenários de falta de páginas, veremos em S0P.

**Tratando exceções no pipeline:** Uma exceção é tratada como um hazard de controle. O endereço da instrução que causou é salvo no reg. EPC, estando salvando PC+4. Um código que informa a causa da exceção é salvo no reg. cause. As instruções que já foram corrigidas e estão na pipeline são descartadas. A unidade de tratamento de hazard redireciona para um endereço de tratamento de exceções. No exemplo 0x80000180.



### Comunicando-se com dispositivos

↳ **Dispositivos de E/S:** todo dispositivo de E/S é composto por uma parte mecânica e uma parte eletrônica. A parte eletrônica possui regis. que indicam o que eles estão fazendo ou precisam fazer.

↳ **Espaço de E/S:** controladores especializados atribuem um número de porta de E/S única a cada req. de cada dispositivo de E/S conectado. Instruções podem ser desenvolvidas para ler (in reg, porta) ou escrever (out reg, porta), permitindo comunicação com os dispositivos.

↳ **E/S mapeada na memória:** uma outra opção é atribuir um endereço via memória para cada req. E/S. Quando o processador lê ou escreve nesse endereço, o controlador detecta a operação e redireciona os dados para o reg. do dispositivo, sem que leitura/escrita ocorra na memória principal.

Com isso, as instruções de leitura (escrito de memória) pedem (lw e sw no MIPS) podem ser usadas para interagir com dispositivos de E/S. Isso requer um controlador que intercepta o sinal da CPU, e redireciona para o local correto.

↳ **E/S:** no MIPS utiliza E/S mapeada, x86 usam ambos as técnicas.

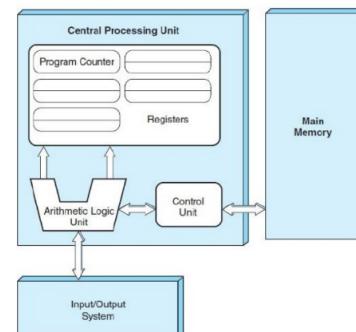
↳ **Enviando sinais ao processador (interrupções):** dispositivos de E/S podem enviar sinais ao processador, caracterizando E/S baseada em interrupções. Quando o processador recebe um sinal externo, ele gera uma interrupção (exceção gerada por um dispositivo externo). O processador pode ler informações sobre a interrupção do barramento externo (ex. qual hardware solicitou). Ao contrário de exceções internas que exigem interrupção imediata, o processador pode terminar de executar as instruções que já estão no pipeline antes de redirecionar para o SO. Isso evita stalls pois a interrupção não impede a conclusão das instruções em andamento.

↳ **Mundo Real:** os principais controladores que fazem as traduções e enviam as interrupções dos dispositivos de E/S para a CPU são:

↳ **Ponte norte:** fica mais próximo da CPU, conecta dispositivos que exigem maior largura de banda (ex. memória, PCIe x16...) e faz o caminho para a ponte de sul.

↳ **Ponte sul:** conecta os dispositivos mais leves, ex. mouse, teclado

↳ **Processadores atuais:** a maioria dos processadores x86-64 incorporam as pontes dentro da CPU, reduzindo atrasos no no circuito e criando um controlador otimizado mais próximo dos núcleos de processamento. Os problemas são que tudo se comunica diretamente com a CPU, gastando área do chip para isso, e pode exigir pinos extras



↳ **Arquitetura de Harvard:** define memórias separadas para dados e instruções, o processador MIPS estudado em aula possui duas memórias separadas para instruções e dados. Nós vimos com arquitetura Harvard pura seguindo estritamente este conceito, sendo comum em microcontroladores (computadores completos em um chip).

Máquinas Harvard modificadas relaxam a separação física, e a maioria dos PCs modernos (x86-64) se enquadra aqui. Eles têm caches separados para dados e instruções nos níveis de memória próximos ao processador, mas usam uma máquina de Von Neumann para acessar a memória principal mais distante através de um único barramento.

↳ **Arquiteturas paralelos:** atualmente, são amplamente utilizadas, em um computador pessoal a supercomputadores. Existem diversos níveis de paralelismo, como pipelining, processadores superscalares, multicore, multiprocessadores, grids e clusters. O foco aqui é em multicore.

↳ **Processadores Multicore:** possuem múltiplos núcleos de processamento em um único chip, e todos os processadores acionam a mesma memória principal por um mesmo barramento (Máquina UMA - Uniform Memory Access). Muitos

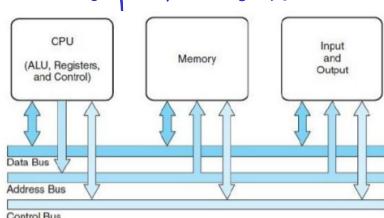
## Arquitetura e Abstrações

↳ **Arquitetura de Von Neumann:** foi originalmente criado por John W. Mauchly e J. Presper Eckert para o sucessor de ENIAC, o EDVAC, durante a segunda guerra mundial. John Von Neumann publicou e popularizou as ideias, levando a arquitetura a ser criada por ele. É famosa pelo conceito de "programa armazenado", antes os programas eram 'hardwired', exigindo mudar o circuito para mudar o programa. Os programas executam o ciclo de Von Neumann, que consiste em:

1. A CPU busca a próxima instrução na memória
2. Instrução é decodificada.
3. Operadores são carregados

4. A ALU executa a operação e o resultado é armazenado.

A arquitetura de Von Neumann é composta por uma CPU (com unidade de controle, ALU, registradores e contador de program), uma memória principal que armazena programas e dados, e um sistema de E/S. Possui um único caminho (lógico ou físico) para a memória principal, o que gera um gargalo conhecido como "Von Neumann bottleneck". Isso pode causar hazards estruturais em pipelines de CPU, exigindo a inserção de stalls. A maioria dos CPUs modernas são variantes da arquitetura de Von Neumann.



não são classificadas como Von Neumann por executarem múltiplas instruções simultâneas (não sequencial). No entanto outros consideram que cada "core" é uma máquina de Von Neumann que coopera para executar tarefas.

↳ **Mix and Match (misturando arquiteturas):** uma CPU x86-64 atual pode ser vista de diferentes maneiras, dependendo do nível de memória ou de entendimento. Pode ser uma arquitetura Harvard (nível de cache) e uma arquitetura de Von Neumann (níveis de memória mais distantes). Pode ser vista como uma Arquitetura de Von Neumann ou como uma arquitetura paralela separada.

↳ **RISC vs CISC:** são estilos de conjuntos de instruções.

↳ **CISC (Complex Instruction Set Computer):** geralmente contém um grande número de instruções complexas de tamanhos variados, que podem executar "múltiplos coisas" em uma única operação (ex: buscar, operar e armazenar). O processador de um PC é frequentemente classificado como CISC.

↳ **RISC (Reduced Instruction Set Computer):** Possui um conjunto de instruções reduzido e simplificado. As instruções são menos "poderosas", mas o objetivo é simplificar o processo e acelerar a execução. As instruções tem tamanhos fixos e poucos formatos. O processador MIPS é RISC, assim como processadores de celular e microcontroladores.

⚠ Processador de um PC é CISC mas quebra as instruções em menores, envolvendo RISC.

## ↳ Tabela comparativa:

RISC	CISC
Instruções de tamanho fixo	Instruções de tamanho variado
Muitos Registradores	Poucos Registradores
Instruções de 3 operandos	Instruções com 1 ou 2 operandos
Parâmetros passados via registrador	Parâmetros passados via pilha
Controle hardwired	Controle microprogramado
Pipeline Profundo e simplificado	Pipeline raso e mais complexo
Poucas instruções simples	Muitas instruções sofisticadas e complexas
Somente loads e stores acessam a memória	Muitas instruções podem acessar a memória

## Parallelismo: conceitos básicos

↳ **Parallelismo no MIPS:** aplica o parallelismo de instruções via pipeline, executando múltiplas instruções em estágios sobrepostos. No entanto, apenas uma é emitida e finalizada por ciclo. Isso reduz o tempo por instrução, mas cada uma opera em apenas um dado.

↳ **Taxonomia de Flynn:** categorização de arquiteturas de computadores com base nos fluxos de instrução e dados.

- **SISD (Single Instruction Single Data):** Uma instrução por vez, operando em um único ciclo dado. Ex: MIPS.

- **MIMD (Multiple Instruction Multiple Data):** Múltiplas instruções operando em múltiplos dados. Multiprocessadores são considerados MIMD, pois possuem vários processadores, cada um executando uma instrução diferente em um dado diferente.

- **MISD (Multiple Instruction Single Data):** Múltiplas instruções operando em um único dado. Atualmente não existem computadores puramente MISD.

- **SIMD (Single Instruction Multiple Data):** Uma única instrução operando em múltiplos dados simultaneamente. Isso é comum em processadores atuais e útil para operações em vetores ou matrizes.

↳ **Multiprocessadores:** é composto por múltiplos processadores, hoje conhecidos como microprocessadores multicore (ex: x86-64). Cada processador pode executar uma tarefa independente, permitindo o parallelismo a nível tarefa/processo. Para aproveitar os múltiplos processadores, é fundamental criar programas paralelos (usando fork, Pthreads).

↳ **Implementação de instruções SIMD:** Para adicionar capacidades SIMD no MIPS, uma abordagem seria criar instruções que operam com dados de 4 em 4 (ou mais). O desafio é que a instrução se tornaria muito longa para o formato 32 bits. A solução é adicionar regis. grandes (ex: 128 bits), como os regis. xmm0 a xmm7 do x86. Com esses regis. uma única instrução (ex: lws xmm0,0(\$t0)) pode corregar múltiplos dados, e outra instrução SIMD (ex: addis xmm0,xmm0,10) pode realizar a mesma operação em paralelo em segmentos desse registrador. Para isso, a CPU MIPS precisa de regis. SIMD separados e múltiplos ALUs para realizar os cálculos em paralelo. Essa estratégia é usada nos CPUs x86 atuais com conjuntos de instruções como SSE (Streaming SIMD Extensions). As operações SSE frequentemente exigem que o vetor comece em um endereço de memória alinhado (múltiplo de 16).

## Hierarquia de Memória e Cache

↳ **Necessidade e desafios:** Apesar de desejarmos ter uma memória infinitamente rápida, limitações físicas e organizacionais impedem essa realidade. Processadores modernos são significativamente mais rápidos do que a memória principal (DRAM) pode suprir.

↳ **Hierarquia de memória:** para mitigar essa diferença de velocidade, os computadores utilizam uma hierarquia de memórias com múltiplos níveis. Os níveis mais próximos da CPU (como os caches) são menores, mais rápidos e mais caros. Os níveis mais distantes da CPU (como a memória principal e o armazenamento secundário) são maiores, mais lentos e mais baratos, priorizando capacidade e menor custo. Em um PC x86-64, a hierarquia tipicamente inclui: **Caches (SRAM - static Random Access Memory)**, **Memória Principal DRAM (Dynamic Random Access Memory)** e **Memória Secundária (SSD ou HD)**. Na maioria das implementações, a CPU armazena dados apenas no nível mais alto (L1 cache), que então requisita do nível inferior (L2), e assim sucessivamente, sem pulos níveis.

↳ **Transparência e Importância da localidade:** Para o programador a hierarquia de memória é geralmente transparente; o programador assume que está na memória principal, no entanto para alcançar o máximo desempenho, é fundamental entender como os diferentes níveis de memória operam. A eficiácia da cache baseia-se em dois princípios de localidade:

- **Localidade Temporal:** Se um item de memória é acessado, é muito provável que seja acessado novamente em um futuro próximo.

- **Localidade Espacial:** Se um item de memória é acessado, é muito provável que seus vizinhos também sejam acessados.

O objetivo da cache é manter os dados mais recentemente utilizados e seus vizinhos para diminuir o acesso à memória principal, que é mais lenta.

↳ **Conceitos Fundamentais da Cache:**

- **Blocos:** tanto a memória principal quanto a cache são divi-

didos em blocos de mesmo tamanho.

- **Hit (acerto):** ocorre quando o dado solicitado pelo processador está presente na cache.

- **Miss (frio):** ocorre quando o dado solicitado não está na cache, exigindo que ele seja carregado no nível de memória inferior.

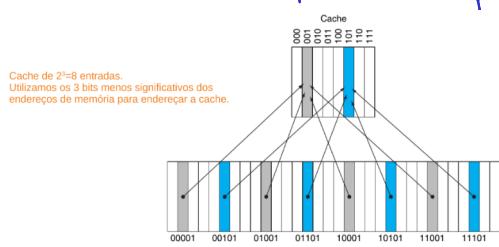
- **Hit Rate:** taxa de acerto,  $hitRate = \frac{hits}{total\ Acessos}$

- **Miss Rate:** taxa de erros,  $missRate = 1 - hitRate$ .

- **Hit Time:** é o tempo necessário para acessar um bloco na cache e verificar se houve um acerto.

- **Miss Penalty:** é o tempo necessário para carregar/substituir um bloco no nível inferior de memória para a cache.

↳ **Cache diretamente mapeado:** Nesse tipo de cache, cada endereço de memória é mapeado para exatamente uma posição na cache. Para cada cache com  $2^n$  entradas, os n bits menos significativos do endereço são usados para determinar a posição da cache.



Cada entrada de cache contém não apenas o bloco (dados) mas também uma tag (os bits mais altos do endereço original) que não foram usados para mapeamento e um bit de validade (para indicar se a entrada contém dados válidos ou lixo, especialmente

te após a inicialização do computador). Um hit é confirmado se o bit de validade forativo e o tag armazenado na cache corresponder ao tag da endereço solicitado. Em caso de miss, o bloco é solicitado do nível de memória inferior, correspondendo para cache (substituindo o conteúdo existente, se houver), o campo Tag é atualizado e o bit de validade é setado antes de o dado ser enviado ao processador.

	valido	Tag	Palavra
000	1	00101	Dado x
001	0	lixo	lixo
010	0	lixo	lixo
011	1	11111	Dado y
100	1	00000	Dado z
101	1	00101	Dado w
110	0	lixo	lixo
111	1	10101	Dado k

### Blocos de Cache e Associatividade

Cache com blocos de uma palavra: Por enquanto a nossa cache se beneficia da localidade temporal, essa configuração não a localidade espacial.

Mapamento por blocos: para tirar proveito da localidade espacial, a memória é dividida em blocos de  $n$  bytes. O endereço de bloco é obtido dividindo-se o endereço de memória pelo tamanho do bloco ( $\frac{\text{Endereço}}{\text{Tamanho do bloco}}$ ). Quando o processador solicita um dado, os bits mais baixos de bloco são usados para procurar na cache e os bits mais altos dos blocos são comparados com o tag. Em caso de Hit, todo o bloco está na cache e os bits que foram descartados para se obter o endereço do bloco, podem ser usados para obter o endereço dentro do bloco.

Ex: se usarmos blocos de 8 bytes e cada endereço suporta 1 byte e a CPU solicita 0000 1011, está no bloco 00001, deslocando 011 dentro desse bloco.

Em caso de miss, todo o bloco é corregido da memória para a cache. As vantagens é que aumentamos a localidade

espacial ao corregor dados e seus vizinhos. As desvantagens é que aumenta a competição na cache (um bloco grande pode possuir muitos dados úteis) e a penalidade de falta, pois mais dados precisam ser corregidos da memória principal. É necessário um equilíbrio no tamanho do bloco para otimizar a localidade espacial e temporal.

Treatamento de escritas: aborda como as operações de escrita são gerenciadas na cache, considerando a hierarquia de memória:

• **Write-Through**: As escritas são sempre propagadas para os níveis mais baixos da memória. O problema é que isso pode tornar a cache inefficiente, pois a CPU precisa esperar a conclusão da escrita nos níveis mais lentos.

• **Write Back**: O dado é atualizado apenas na cache e só é propagada para os níveis mais baixos de memória quando o bloco na cache é substituído. Isso é a política usada na maioria dos CPUs atuais. No entanto, é mais complexa de gerenciar, especialmente em máquinas com múltiplos CPUs, pois pode levar a inconsistência de dados entre os caches (se a CPU 12 uma versão desatualizada da memória principal).

Associatividade da cache:

• **Cache diretamente mapeado**: Cada endereço de memória é mapeado para uma única posição na cache. Se esse posição já estiver ocupada, o conteúdo é substituído, mesmo que outras posições estejam livres.

• **Cache totalmente associativo**: um bloco da memória pode ser corregido para qualquer posição livre na cache. A vantagem é que reduz o número de misses devido à maior flexibilidade na escolha dos blocos a serem

substituídos. As desvantagens são que aumenta o custo em hardware, tempo e complexidade, pois é necessário comparar o tag com o endereço solicitado em todos os posicionamentos da cache em paralelo.

• **Cache Associativo por conjunto**: representa um meio-termo entre os dois modelos anteriores. A cache é dividida em conjuntos, e cada conjunto separado em  $n$  blocos (cache associativo de  $n$  vias). Os blocos de memória são mapeados para conjuntos específicos, mas dentro de um conjunto, um bloco pode estar em qualquer posição. A solicitar um endereço, os bits de índice são usados para encontrar o conjunto, e os tags de todos os blocos desse conjunto são comparados em paralelo. Caches com menor associatividade geralmente significam menos hardware e podem ser um pouco mais rápidas, mas aumentam as misses.

Endereçamento com Associatividade: considere que a memória é endereçada com  $j$  bits, com blocos de tamanho  $K$  (número de endereços por bloco), os últimos  $\lg K$  bits são utilizados para se descrever o endereçamento de bloco (offset). Tendo  $n$  conjuntos na memória,  $\lg n$  bits após os bits de deslocamento (offset) são utilizados para se descrever o conjunto da cache. Os demais bits fazem parte do campo Tag:



Ex: considere uma máquina onde a memória é endereçada usando 64 bits. Considera blocos de 16 bits, e uma cache

associativa de 4 vias com 8 conjuntos. Temos então

64 bits → Tag = 64 - lg 8 - lg 16 = 57 | conjunto lg 8 = 3 | offset lg 16 = 4

**LRU**: Least recently used: no caso de um miss em uma cache associativa, se todos os blocos do conjunto estiverem ocupados, é necessário substituir um bloco existente. A política LRU remove o bloco que foi acessado a mais tempo. É um esquema comum em CPUs modernas devido ao seu bom custo-benefício, sendo melhor do que uma seleção aleatória. Em caches associativas de 2 vias, a implementação do LRU é relativamente simples, podendo ser feita com um bit de uso para cada bloco: quando um bloco é acessado, seu bit é ativado e o outro bloco é desativado. Para caches com maior número de vias (ex: 4, 8 ou 16 vias), a implementação se torna mais complexa e cara devido à necessidade de mais bits e lógicas de atualizações mais elaboradas. Por isso, caches com muitas vias frequentemente usam aproximações do LRU ou até mesmo esquemas aleatórios, especialmente quando a associatividade é grande e o desempenho do LRU se aproxima da escolha aleatória.

**Caches Multiníveis**: processadores modernos utilizam múltiplos níveis de cache para otimizar o desempenho. Os níveis mais altos (L1, L2, etc.), mais próximos da CPU, são menores e fornem uma redução do tempo de acesso e de penalidade de miss. Geralmente são caches de acesso exclusivo para cada núcleo e frequentemente segmentados

em cache de instruções e cache de dados (característico de arquitetura de Harvard neste nível). Os níveis mais baixos, mais distantes da CPU, são maiores e fornecem uma redução da probabilidade de miss.

**Problemas de coerência de cache:** Em CPUs com múltiplos núcleos e caches write-back exclusivos para cada núcleo (L1:L1) e uma cache compartilhada (L2:L2), pode ocorrer que diferentes CPUs vejam versões diferentes da mesma dados. Isso acontece porque, com write-back, os dados são atualizados nos níveis inferiores da memória (como a L2) somente quando o bloco no cache de nível superior (como a L1) é substituído ou explicitamente escrito de volta.

↳ **Protocolo de Snooping:** é uma solução popular para este problema. Quando uma CPU escreve em um bloco de sua cache, ela envia um sinal de broadcast para os demais CPUs para "surpar a cache". Isso faz com que as outras CPUs invalidem (desliguem o bit de validade) o bloco correspondente em suas próprias caches, forçando-as a buscar uma cópia mais recente dos dados quando necessário.

### Construção de Memórias

↳ **Memória de Acesso Aleatório (RAM - Random Access Memory):** permite acesso direto a qualquer palavra na memória, ao contrário de mídias como fitas, onde o acesso é sequencial. O termo "RAM" é frequentemente usado para se referir a memória principal dos PCs, embora a maioria das memórias utilizadas sejam de tipo RAM.

↳ **Static RAM (SRAM):** utiliza portas lógicas (flip-flop) para arm-

azenar bits. Consegue manter os dados enquanto o circuito está alimentado. É comumente utilizado na construção de memória cache e registradores devido à sua velocidade. É geralmente mais rápido que a DRAM.

↳ **Dynamic RAM:** armazena carga em capacitores, onde  $\text{C} = \frac{\text{Q}}{\text{V}}$  é um capacitor carregado e  $\text{C} = 0$  descarregado. Capacitores perdem a carga com o tempo, necessitando de recargas (refresh) periódicas (ca. a cada 64ms). A leitura de um capacitor remove sua carga, o que significa que o dado é perdido se não for renovado após uma leitura. É denominada "dinâmico" por sua tendência em perder a carga. É comumente usada na memória principal dos PCs. Possui um circuito mais simples e menor, sendo assim mais densa e barata que a SRAM.

↳ **Memória somente leitura (ROM - Read Only Memory):** permite somente a leitura de dados, não a escrita. É uma memória não volátil, o que significa que não precisa ser constantemente alimentada para manter os dados. Os dados são "hardwired" (gravados fisicamente) na memória resultando em um alto custo de fabricação inicial.

↳ **Tipos de ROM Programável:**

- **Programmable ROM (PROM):** pode ser gravada uma única vez, geralmente por equipamentos que queimam circuitos internos.
- **Erasable Programmable ROM (EPROM):** Pode ser lido ou escrito mas antes da escrita precisa ser completamente apagada, utilizando radiação ultravioleta.
- **Electrically Erasable Programmable ROM (EEPROM):** permite ler e escrever qualquer palavra individualmente, com a escrita sendo mais lenta que a leitura.

↳ **Memórias Flash:** o microchip é construído para que uma seção de memória seja rapidamente apagada em um único "flash". O arranjo NAND Flash é um dos mais comuns, com comportamento similar a portas NAND. Utiliza transistores com floating gate. Para apagar, a linha inteira de transistores é apagada. O floating gate se desgasta com o tempo, o que confere às memórias flash uma durabilidade limitada. São comumente utilizadas em pendrives (flash-drivers) e dispositivo de estado sólido (SSDs).

### Microcontroladores

↳ **Diferença entre computadores:** PCs são máquinas complexas com múltiplos processadores, grandes volumes de memória RAM e armazenamento, capazes de executar centenas de tarefas simultaneamente com sistemas operacionais gigantescos e interfaces gráficas. No entanto, a maioria dos aparelhos eletrônicos do dia a dia precisam de algum tipo de processamento, mas não justifica o uso de processadores de desktop devido a questões de energia, custo e projeto.

↳ **Dispositivos:** muitos dispositivos cotidianos, como geladeiras, smart TVs, carros etc. requerem processamento. Esses dispositivos têm as seguintes características:

- Não permitem alteração de programa pelo usuário;
- Executam um único programa;
- Não possuem SOs.
- São otimizadas para a tarefa para a qual foram desenvolvidas.
- Possuem baixo custo e baixo consumo de energia.
- Têm recursos limitados.

↳ **Sistemas Embocados:** são dispositivos com processamento dedicado a executar uma tarefa específica são comumente chamados de sistemas embocados.

• **ASIC (Application-Specific Integrated Circuit):** circuitos específicos para um problema, onde o programa está "hardwired" nas portas lógicas. São caros para projetar e produzir, mas são viáveis para produção em larga escala e tendem a ser mais rápidos.

• **FPGA (Field Programmable Gate Array):** matrizes de elementos de processamento que permitem implementar funções lógicas diretamente no hardware, sendo hardware programável.

↳ **Microcontroladores:** são um sistema computacional simples e completo em um único chip. Em um chip comumente incluem: Processadores, memória de trabalho (RAM principal), memória de armazenamento (ROM/Flash) e controladores e pinos para dispositivos de entrada e saída (Ex: PWMs para motores, conversores analógicos/digitais). São baratos e de baixo consumo de energia.

• **PIC 16F698A:** possui palavra de 14 bits para a instrução e 8 bits para dados. Memória: 2048 palavras para instrução, 224 palavras para dados não permanentes (memória principal), e 198 palavras de dados permanentes. Isso equivale a  $\frac{2048 \times 14}{8} = 3584$  bytes = 3.5 kB. Utiliza Arquitetura Harvard e instruções RISC.

↳ **Aplicações de tempo Real:** Microcontroladores (ASICs/FPGAs) são comuns em sistemas de tempo real, onde há um tempo máximo pré-definido para obter uma resposta. Em contraste, PCs com SOs multitarefas convencionais tornam difícil calcular o tempo de pior caso de uma aplicação, devido à competição

por recursos da CPU, invalidações de cache e rotinas do sistema operacional.

↳ **Ciclos por instrução (CPI)**: O CPI mede o médio de ciclos de clock necessários para completar uma instrução. Na CPU MIPS estudada (sem stalls), o CPI seria 1, o que é o melhor caso possível para uma CPU sem unidades funcionais redundantes. CPUs x86 modernas são superescaláveis, possuem unidades funcionais replicadas internamente (múltiplos ALUs, por exemplo), o que permite que seu CPI seja menor que 1.