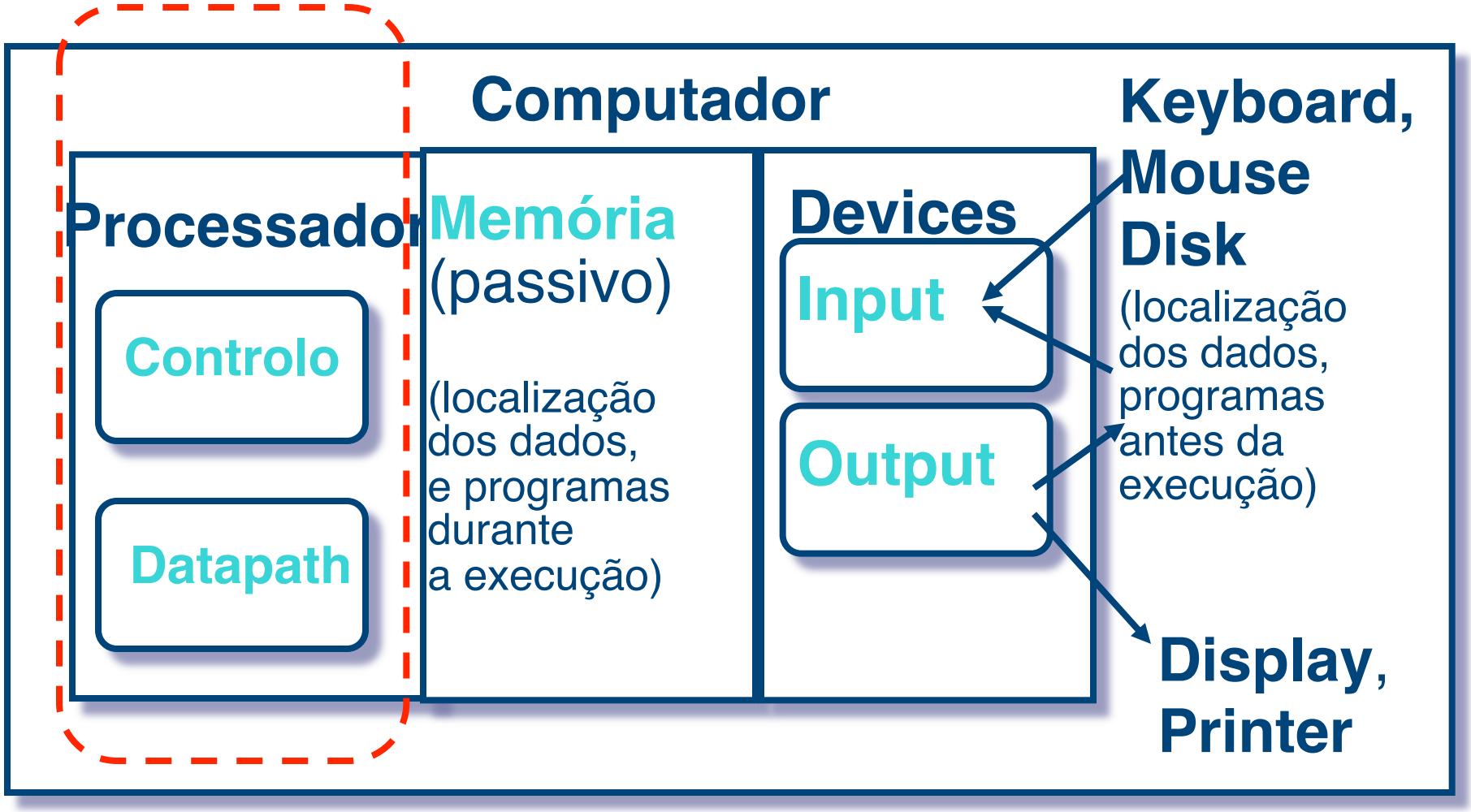
Introdução à Arquitetura de Computadores
- Etapas do Datapath -

Sistemas de Microprocessadores 2021/22

Os 5 componentes fundamentais



A CPU

- Processador ou Unidade Central de Processamento (CPU): a parte ativa do computador que faz o trabalho (manipulação de dados e tomada de decisões)
- Datapath: parte do processador que contém o hardware necessário ao desempenho de operações (the brawn)
- Control: parte do processador (também em hardware) que diz ao datapath o que é preciso ser feito (the brain)

Etapas do Datapath : Overview

- Problema: A utilização de um único bloco de hardware que “execute a instrução” do início ao fim conduziria a um design complexo e a um desempenho ineficiente.

- Ideia Chave: dividir o processo de “executar uma instrução” num conjunto de etapas, e depois ligar todas estas etapas para criar o datapath completo
  - Etapas menores especializadas são mais simples de desenhar em hardware (dividir o problema em sub-problemas)
  - Podemos otimizar uma determinada etapa sem interferir com as outras (modularidade)

Etapas do Datapath (1/6)

- O “Instruction Set” do MIPS é composto por instruções muito variadas: Quais serão as etapas que elas têm em comum?

- Etapa 1: Instruction Fetch
  - A word de 32-bits na qual a instrução é codificada tem que ser sempre lida da memória (instruction fetch)
  - Para além disso o PC (programa counter) tem que ser sempre incrementado para apontar para a instrução seguinte (PC = PC + 4)

Etapas do Datapath (2/6)

- Etapa 2: Instruction Decode
  - Depois do fetch, é necessário fazer a descodificação da instrução e obter os dados associados a cada campo
  - Primeiro, ler o opcode para determinar o tipo de instrução e o tamanho dos campos
  - Segundo, ler os dados de todos os registos indicados de forma a definir os operandos
    - Para o `add`, lê-se dois registos
    - Para o `addi`, lê-se um único registo
    - Para o `jal`, não é necessário ler-se registos

Etapas do Datapath (3/6)

- Etapa 3: ALU (Unidade Aritmética e Lógica)
  - Na maior parte das instruções o trabalho efetivo é feito neste nível: aritmética (+, -, *, /), deslocamento, lógica (&, |), comparações (`slt`)
  - E quanto aos loads e stores?
    - `lw  $t0, 40($t1)`
    - Repare que é necessário calcular o endereço final através da adição de 40 (imediato) ao contéudo do registo `$t1`
    - A adição para o cálculo do endereço é feita nesta etapa

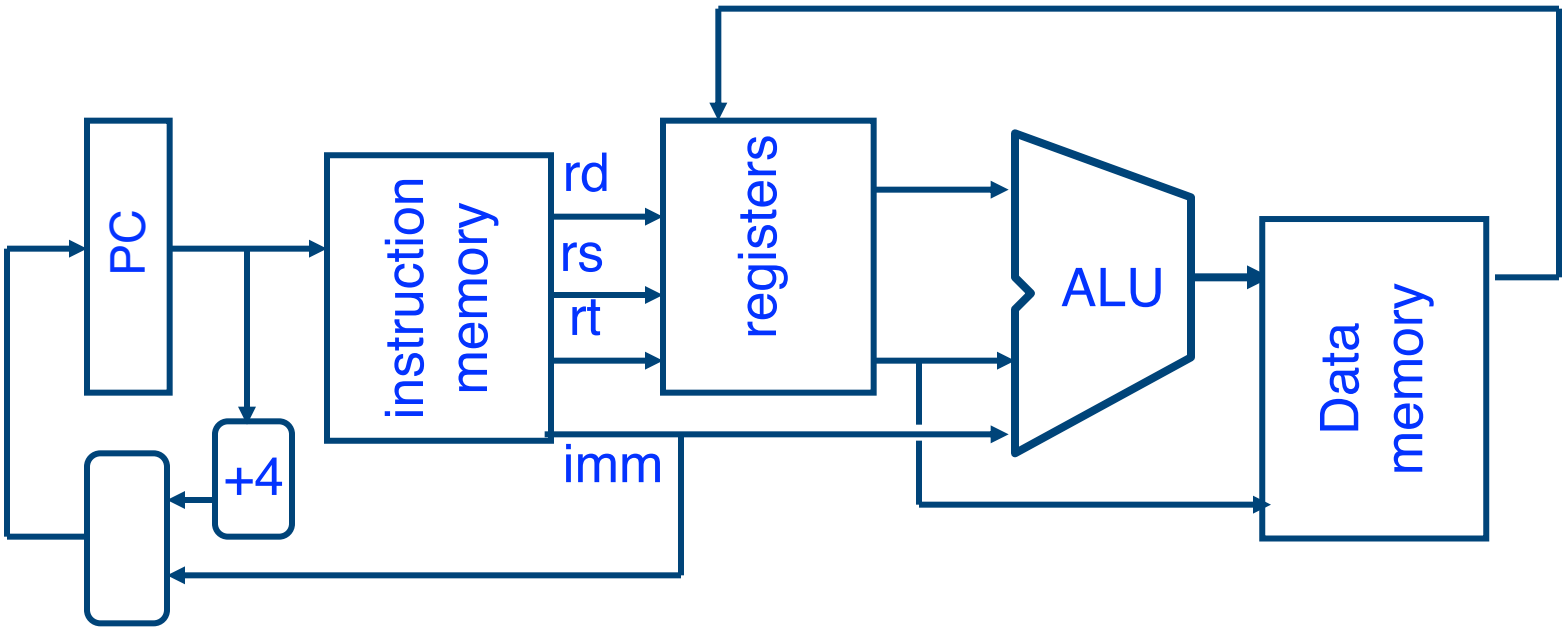
Etapas do Datapath (4/6)

- Etapa 4: Memory Access
  - São somente as instruções load e store que fazem trocas de informação com a memória (leitura e escrita); todas as outras instruções ficam inativas (*idle*) durante esta etapa.
  - Esta é uma etapa incontornável para a implementação dos loads e stores. Assim, e apesar das outras instruções não terem este passo, o datapath tem que conter esta etapa.

Etapas do Datapath (5/6)

- Etapa 5: Register Write
  - A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
  - Exemplos: operações aritméticas e lógicas, deslocamentos, loads, `slt`
  - E quanto aos stores, jumps e branches?
    - Estas instruções não escrevem nenhum resultado num registo destino
    - São instruções que permanecem inativas durante esta etapa.

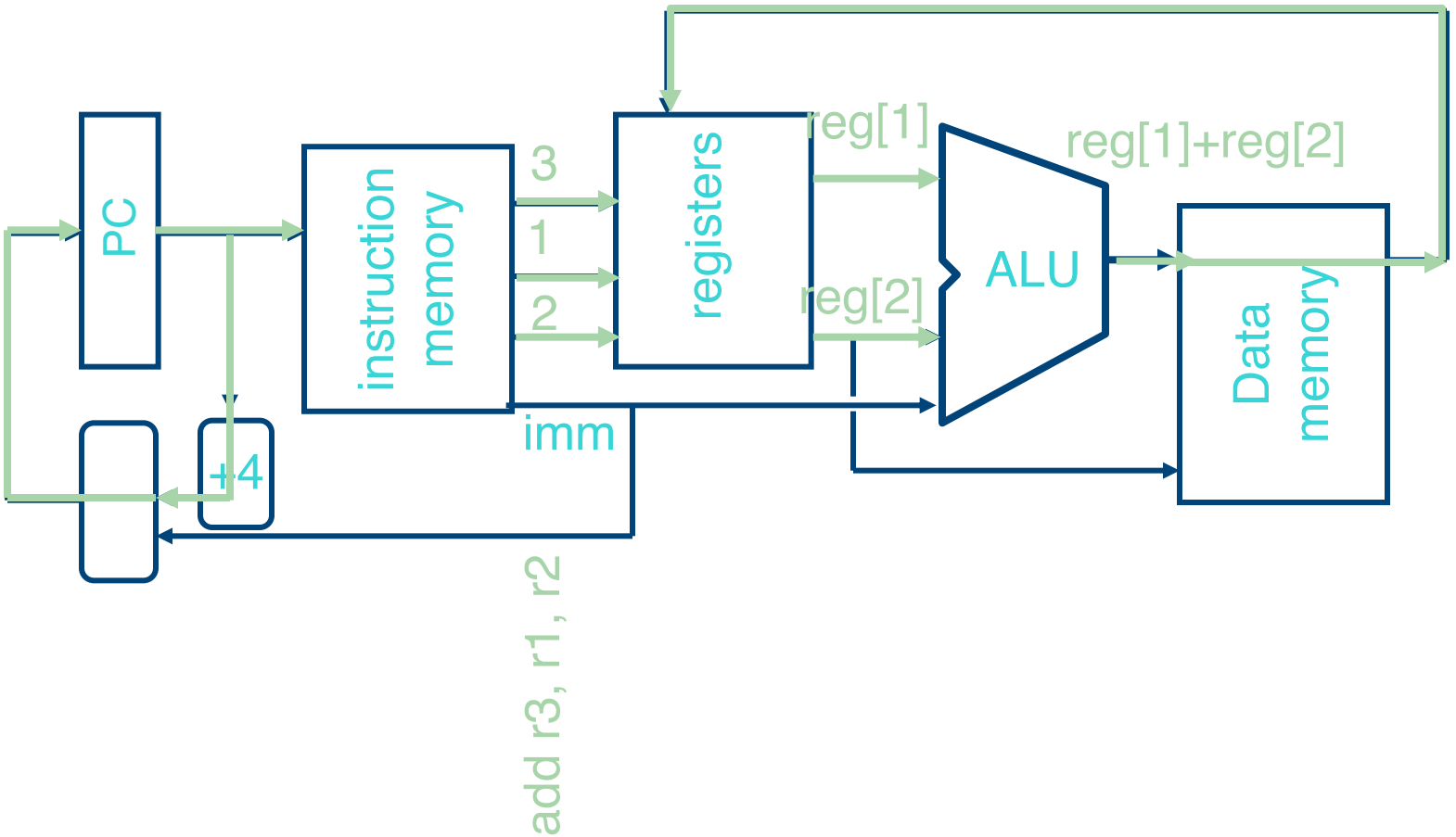
Etapas do Datapath (6/6)



Datapath Walkthroughs (1/3)

- `add  $r3,$r1,$r2  # r3 = r1+r2`
  - Etapa 1: instruction fetch, incrementa PC
  - Etapa 2: descodificação para determinar que é um `add`. Leitura dos registos `$r1` e `$r2`
  - Etapa 3: soma dos dois valores provenientes da etapa 2
  - Etapa 4: idle (não há qualquer leitura/escrita de memória)
  - Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

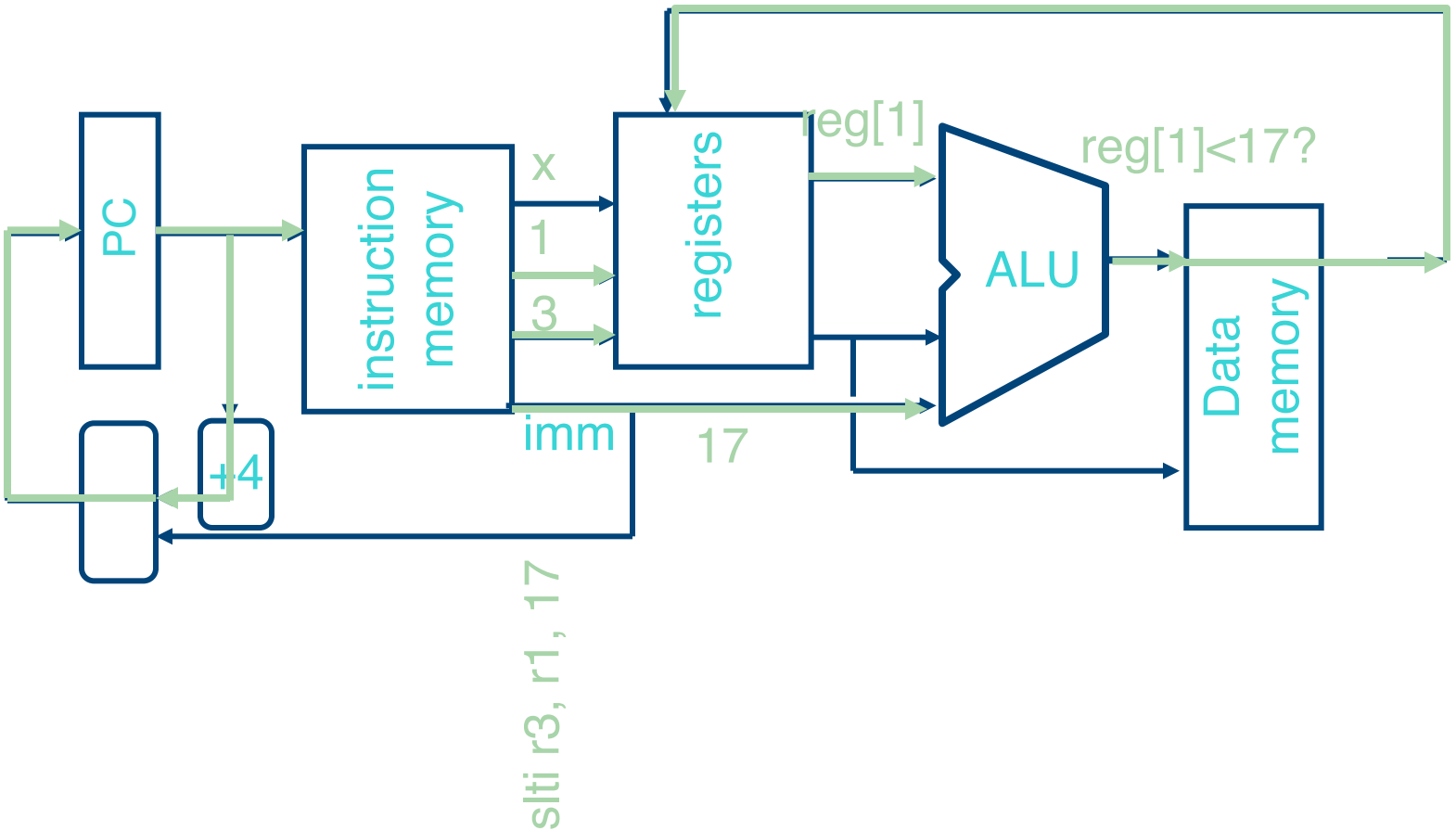
Exemplo: instrução `add`



Datapath Walkthroughs (2/3)

- `slti  $r3,$r1,17`
  - Etapa 1: fetch da instrução, incrementa PC
  - Etapa 2: descodificação para descobrir que é um `slti`. Leitura do registo `$r1`
  - Etapa 3: comparação do valor proveniente da Etapa 2 com o inteiro 17
  - Etapa 4: idle
  - Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

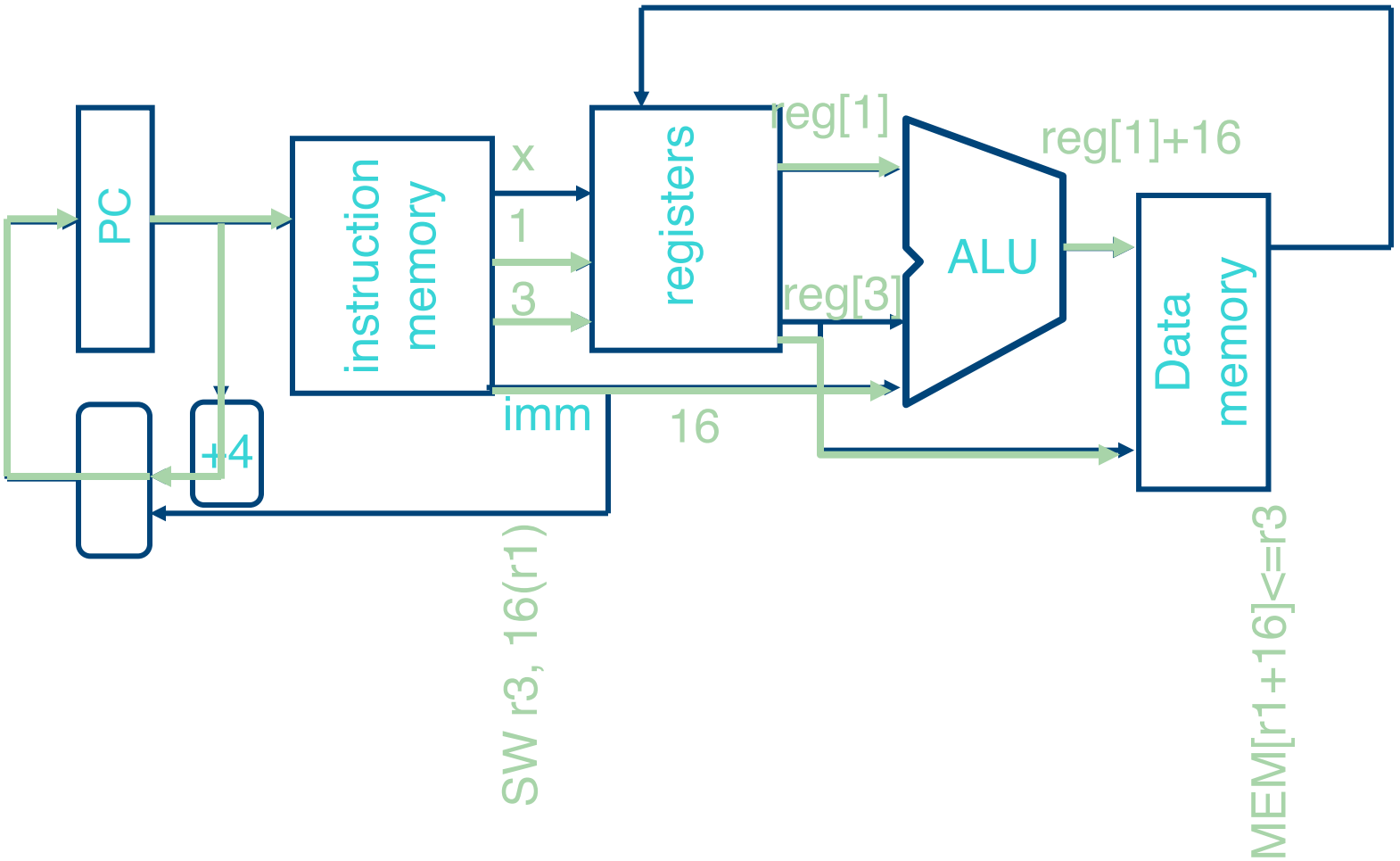
Exemplo: Instrução `slti`



Datapath Walkthroughs (3/3)

- `sw  $r3, 16($r1)`
  - Etapa 1: fetch da instrução, incrementa PC
  - Etapa 2: descodificação para saber que é um sw. Leitura dos registos $r1 e $r3
  - Etapa 3: soma de 16 ao valor do registo $r1
  - Etapa 4: escrita do valor no registo $r3 (proveniente da Etapa 2) na posição de memória com o endereço calculado na Etapa 3
  - Etapa 5: idle (não há nada a escrever nos registos)

Exemplo: Instrução `sw`



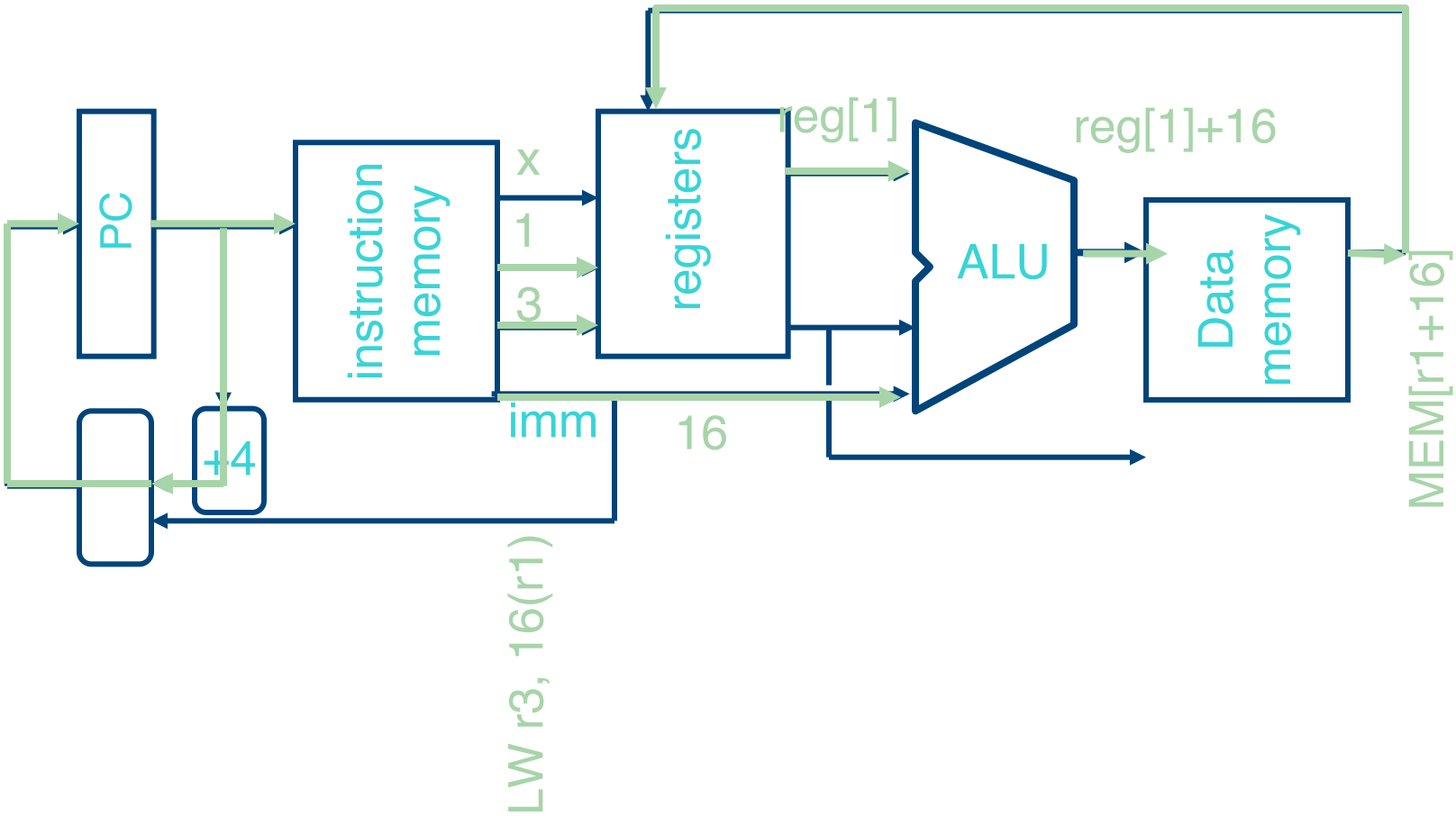
Por quê 5 etapas? (1/2)

- Poderíamos ter um número diferente de etapas?
  - Sim, há outras arquiteturas que têm um número diferente
- Então porque é que o MIPS tem 5 etapas quando a maior parte das instruções estão inativas em pelo menos um estágio? Quatro não seria suficiente?
  - As cinco etapas são a união de todas as operações necessárias à implementação do Instruction Set.
  - Há uma instrução que está ativa nas cinco etapas: o load

Por quê 5 etapas? (2/2)

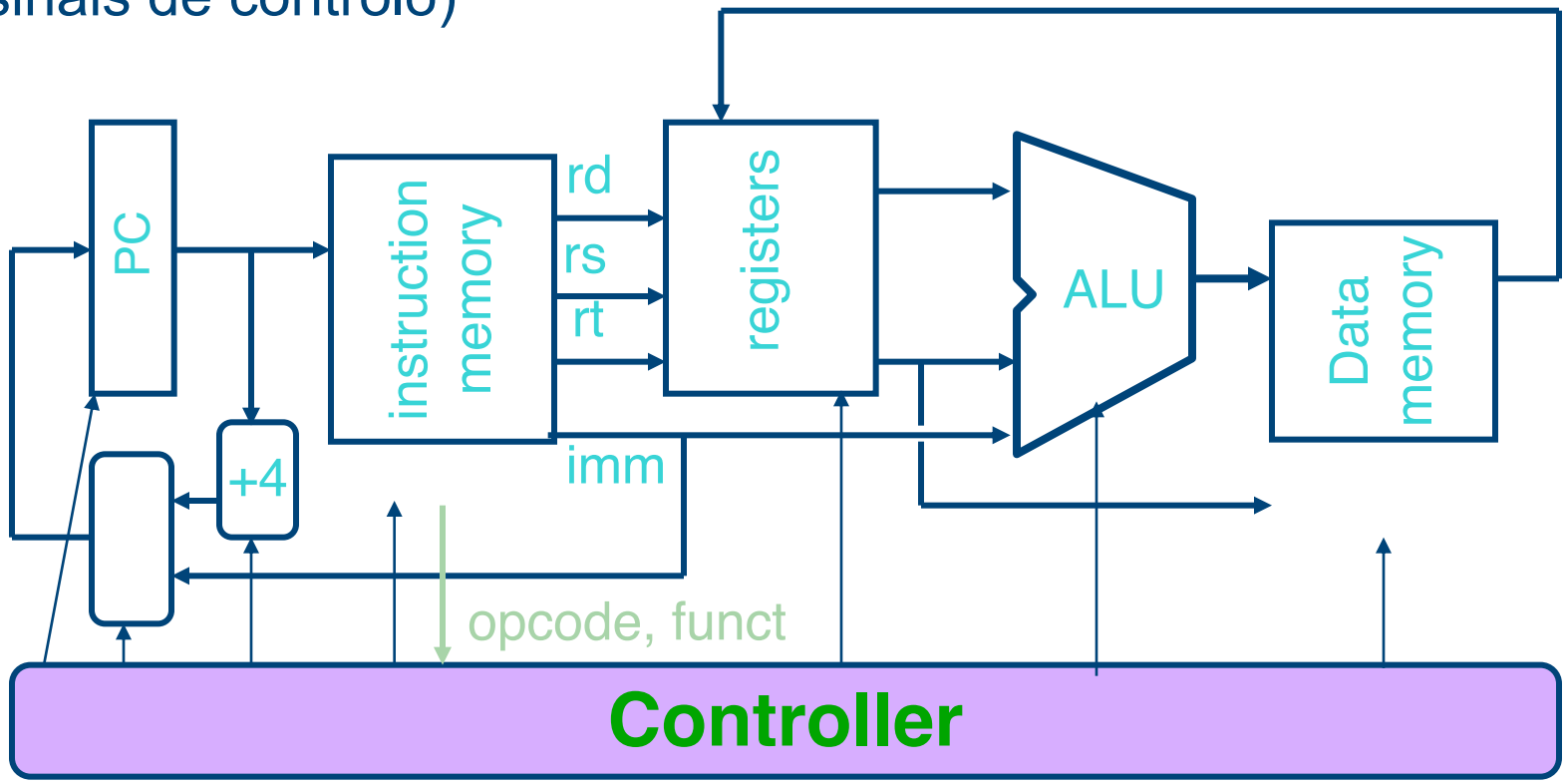
- `lw  $r3, 16($r1)`
  - Etapa 1: fetch da instrução, incrementa PC
  - Etapa 2: descodificação para determinar que é um `lw`. Leitura do registo `$r1`
  - Etapa 3: soma 16 ao valor do registo `$r1`
  - Etapa 4: leitura da posição de memória com o endereço calculado no estágio 3
  - Etapa 5: escrita do valor lido no registo `$r3`

Example: `lw` Instruction



Sumário - Datapath

- O datapath é definido pelas transferências de dados necessárias à execução da instrução
- O *controlador* faz acontecer as transferências de dados corretas (sinais de controlo)



Qual é o hardware necessário? (1/2)

- PC: um registo que guarda o endereço de memória onde se encontra a próxima instrução

- Registos de utilização geral
  - Usados nas etapas 2 (leitura) e 5 (escrita)
  - MIPS tem 32 registos destes

- Memória
  - Usada nas etapas 1 (fetch) e 4 (R/W)
  - Veremos à frente que o sistema de cache tenta tornar estas duas etapas tão rápidas como as restantes.

Qual é o hardware necessário? (2/2)

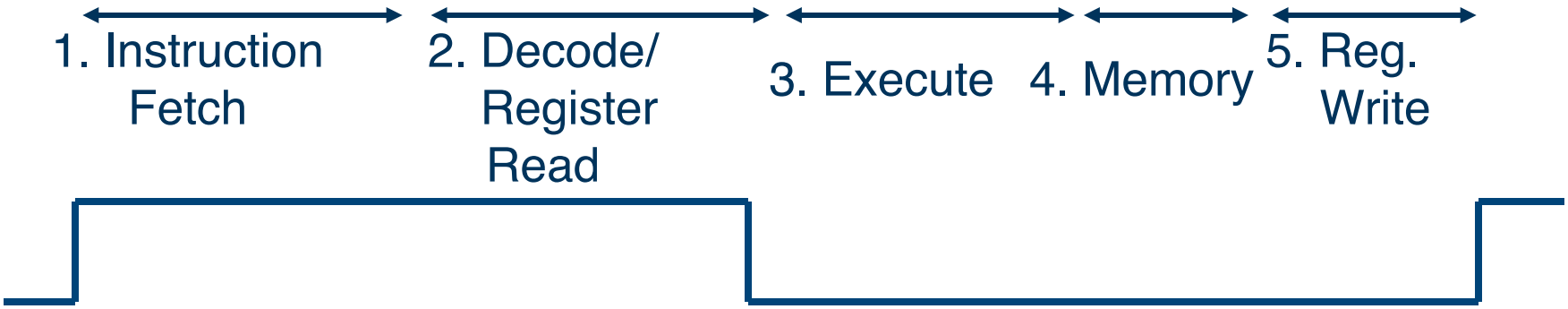
- ALU
  - Usada na etapa 3
  - Algo que implementa todas as funções necessárias: aritméticas, lógicas, etc.

- Registos auxiliares
  - Nas implementações em que cada etapa é executada num ciclo de relógio, é muitas vezes necessário utilizar registos auxiliares para guardar resultados intermédios entre etapas, bem como sinais de controlo que viajam de uma etapa para a outra.

CPU clocking (1/2)

Como é que controlamos o fluxo de informação que atravessa o datapath?

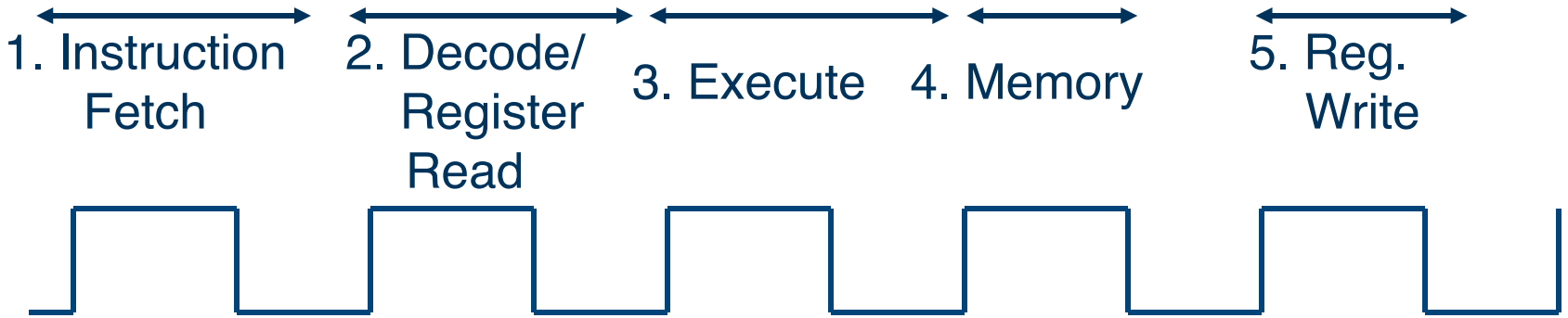
- Single Cycle CPU: todas as etapas de uma instrução são completadas em um único *longo* ciclo de relógio.



CPU clocking (2/2)

Como é que controlamos o fluxo de informação que atravessa o datapath?

- Multiple-cycle CPU: cada etapa corresponde a um ciclo de relógio.
  - O período do relógio é igual à duração da etapa mais longa



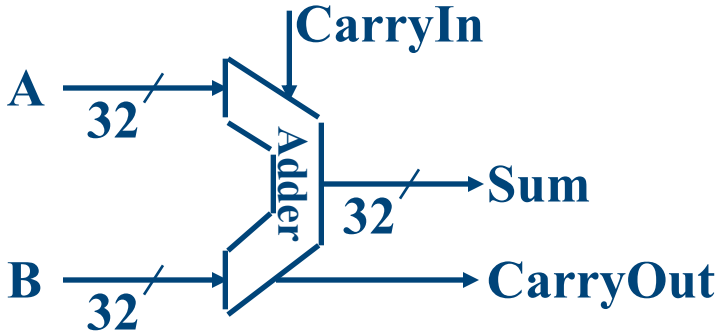
- O multi-cycle tem vantagens em relação ao single cycle:
  - Podemos saltar etapas em que uma determinada instrução está inativa
  - Podemos implementar mecanismos de sobreposição/pipelining.

Como desenhar um processador: passo-a-passo

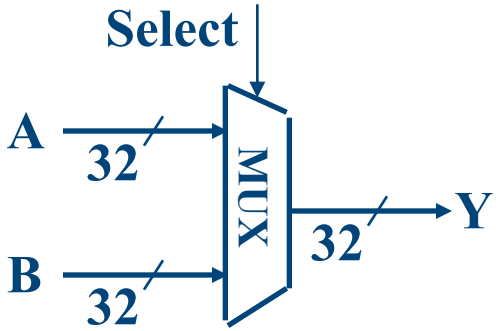
1. Analisar o “Instruction Set” a ser implementado (ISA) para obter os requisitos do datapath
   - Cada instrução define um conjunto de transferências entre registos que deve ser suportada pelo datapath.
2. Selecionar os componentes de hardware (somadores, mux, etc) que se utilizam e definir um método de clocking:
   - Single Cycle CPU ou Multi-Cycle CPU
3. Fazer a montagem do datapath de forma a ir ao encontro dos requisitos.
4. Analisar a implementação de cada instrução para determinar os pontos de controlo que afetam a transferência entre registos.
5. Construir a lógica de controlo

Building Blocks - Lógica Combinatória

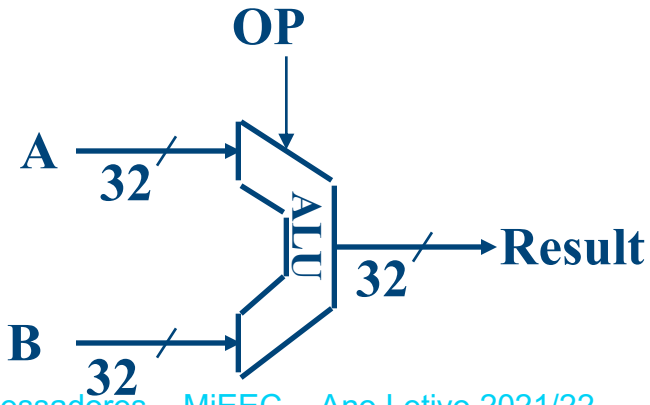
- Somador



- MUX

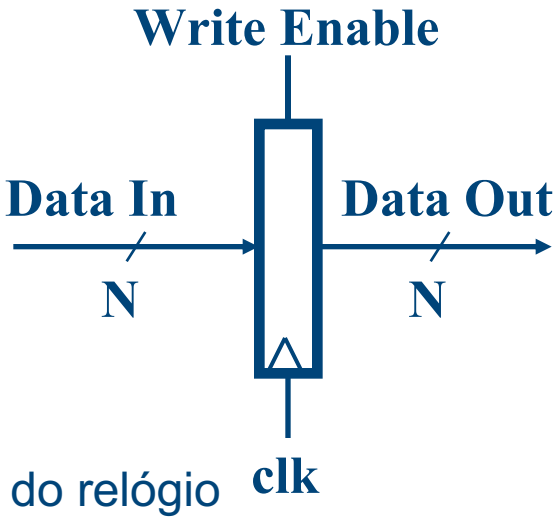


- ALU



Building Blocks - Armazenamento em registos

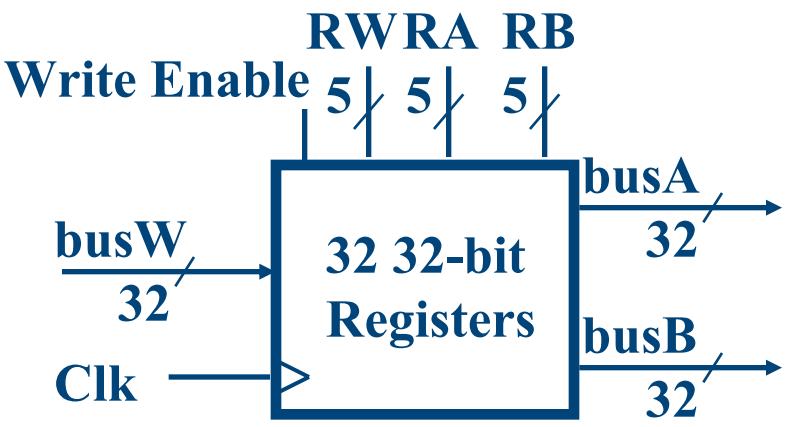
- Semelhante a um Flip-Flop D exceto
  - Entrada e saída de N-bits
  - Write Enable



- Write Enable:
  - Não asserido (0):
    Data Out não se modifica
  - Asserido (1):
    Data Out fica igual a Data In na vertente positiva do relógio

Armazenamento: Register File

- Consiste em 32 registos:
  - 2 buses de saída de 32-bit (busA and busB)
  - 1 bus de entrada de 32-bit: busW
- O Registo é selecionado por:
  - RA (número) seleciona o registo para busA
  - RB (número) seleciona o registo para busB
  - RW (número) seleciona o registo a ser escrito via busW quando Write Enable é 1



- Repare que é possível fazer leitura e escrita simultaneamente
- Clock input (clk)
  - O clk input só é importante para operações de escrita
  - Na leitura o "register file" comporta-se como lógica combinacional:
    - RA ou RB válido $\Rightarrow$ busA ou busB válido depois de "access time."

Notas Finais

- O desenho da lógica de controlo é sempre a parte mais complexa na implementação em hardware de uma arquitetura
- Repare que consegue antever como tudo isto pode ser feito usando os conhecimentos que adquiriu em Laboratório de Sistemas Digitais.
- O livro discute como fazer a implementação de um single-cycle CPU (Cap. 5.3) e de um multi-cycle CPU (Cap. 5.5).
- Disciplinas avançadas que discutem o desenho de CPUs
  - Arquitetura de Computadores (MiEEC)
  - Projeto de Sistemas Digitais (MiEEC)
- Se tivermos tempo ainda voltaremos a esta questão ... mas para já vamos assumir uma implementação multi-cycle e discutir como aumentar o desempenho tirando partido do paralelismo entre instruções.

Para saber mais ...

- P&H - Capítulos 5.1 e 5.2

- P&H - Capítulos 5.3, 5.4 (implementação de um single cycle CPU) e 5.5 (implementação de um multi-cycle CPU). Esta matéria não foi dada em detalhe nas aulas, mas deverá interessar aos mais curiosos.

10 mandamentos

- 1 - Não sobreporás a pilha à sua heap!
- 2 - Static, não te mexerás durante todo o processo.
- 3 - Faz free na heap, não sejas preguiçoso!
- 4 - Não esquecerás de deixar a pilha tal como a encontraste.
- 5 - Armazenarás sempre globais na static!
- 6 - Tratarás a memória por dois 0s no final. Ela agradecerá.
- 7 - Honrarás o Code.
- 8 - Fragmentarás o menos possível a memória.
- 9 - Não farás free duas vezes ao mesmo ptr.
- 10 - Ama a memória como a mais nenhuma outra coisa; não uses ídolos que não a memória. Protege-a com a vida!

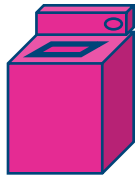
Introdução à Arquitetura de Computadores
- Pipelining para melhoria de Desempenho -

Sistemas de Microprocessadores 2021/22

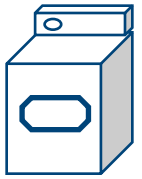
Vamos lavar a roupa ...

- A Ana, Bernardo, Carlos e Diana têm um saco de roupa suja para lavar, secar, dobrar e arrumar na gaveta.

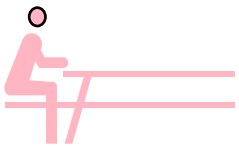
- A máquina de lavar demora 30 minutos



- O secador de roupa demora 30 minutos

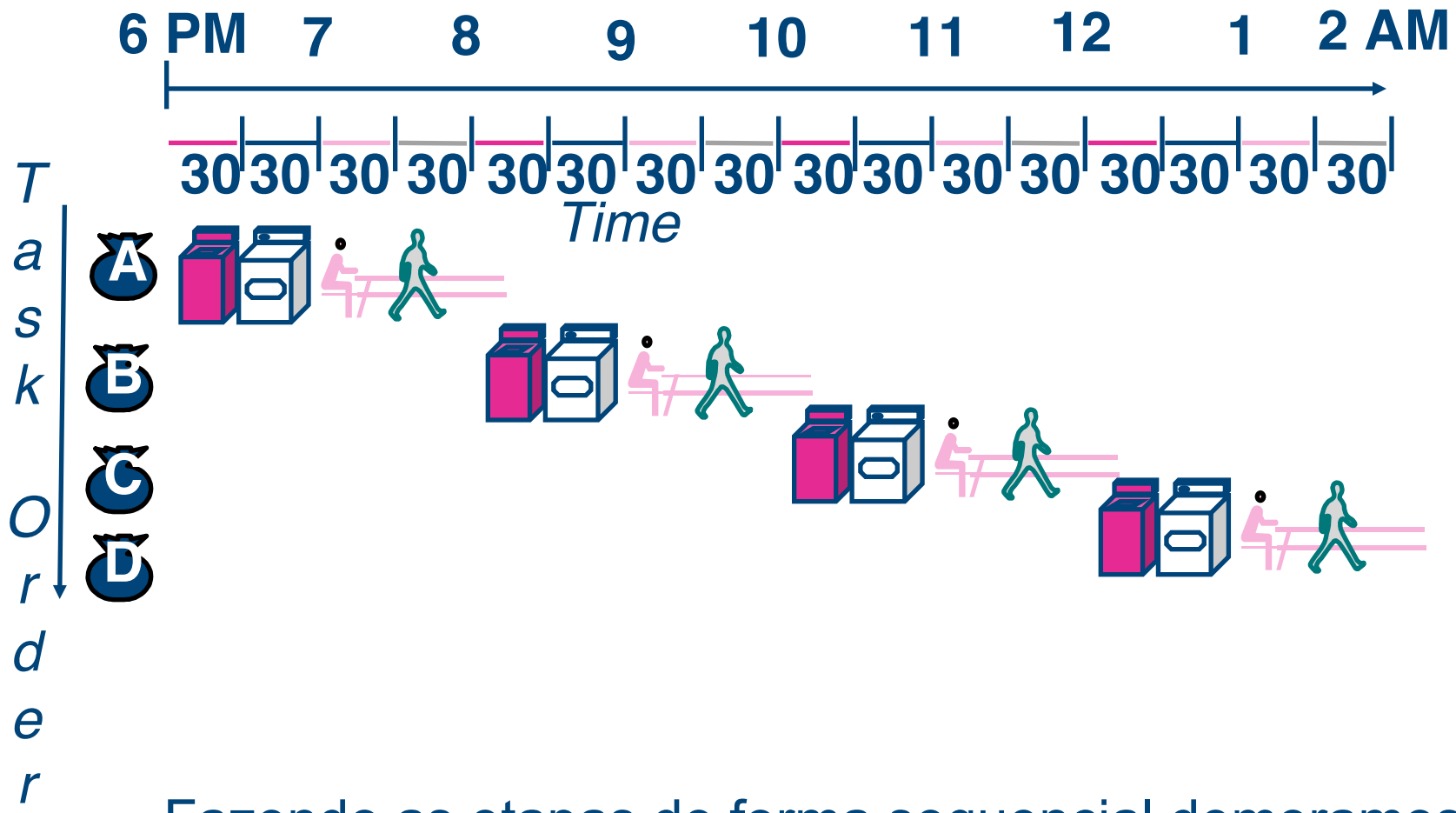


- A “dobragem” demora 30 minutos



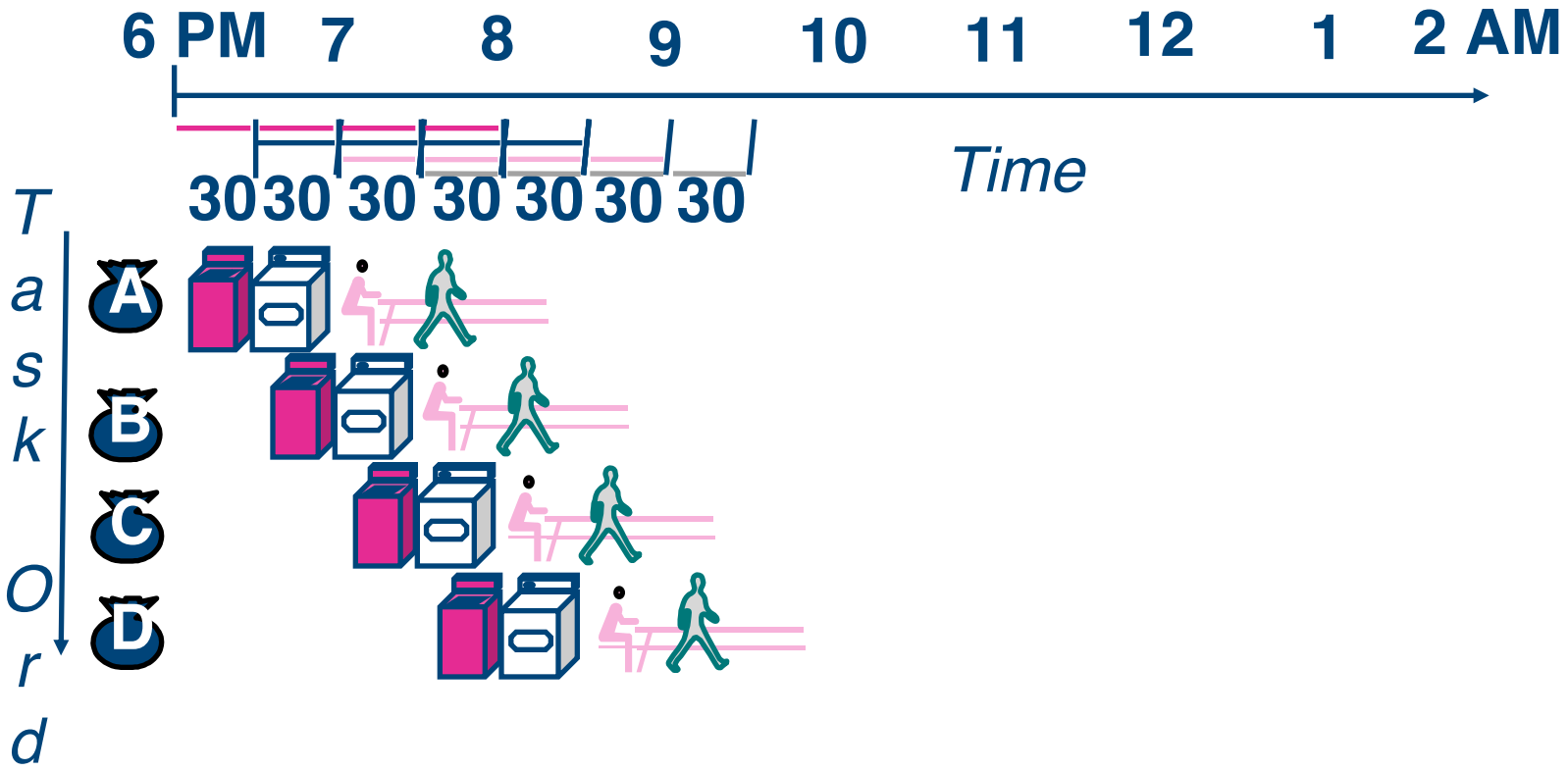
- A “arrumação” na gaveta demora 30 minutos

Operação Sequencial



- Fazendo as etapas de forma sequencial demoramos um total de 8 horas para 4 cargas de roupa

Operação em Pipeline

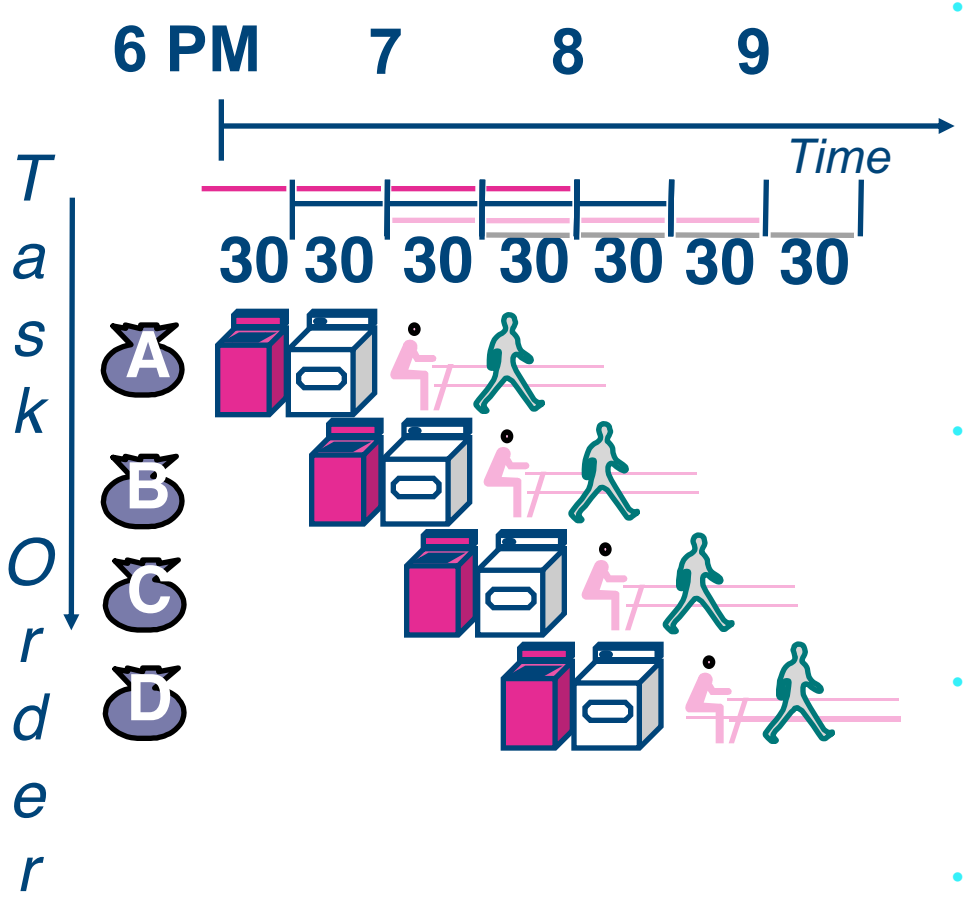


- Executando em pipeline demoramos 3.5 horas para 4 cargas de roupa sem precisarmos de recursos adicionais (ex: outra máquina de lavar ou secar)!

Definições

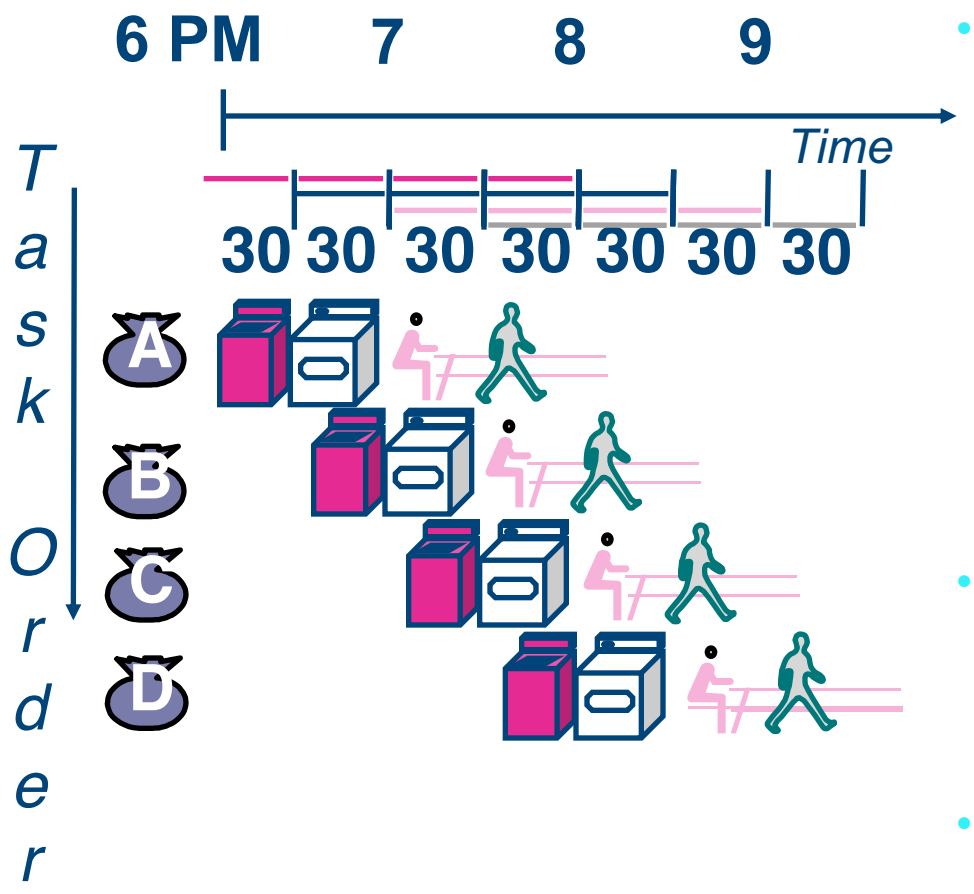
- Latência: tempo necessário à execução de uma determinada tarefa
  - Exemplo: o tempo para ler um setor do disco é o tempo de acesso a disco ou latência do disco
- Throughput: Quantidade de trabalho que conseguimos fazer durante um determinado período de tempo.
- Speedup: fator multiplicativo de aceleração

Lições sobre execução em Pipelining (1/2)



- O Pipelining não melhora a latência inerente a cada tarefa; aquilo que faz é melhorar o throughput na execução de um número de tarefas (workload), que podem ser total ou parcialmente paralelizáveis.
- A ideia base é executar múltiplas tarefas simultaneamente usando diferentes recursos físicos.
- Potential speedup = Número de estágios/etapas no pipe
- O tempo necessário para "encher" e "limpar" o pipeline reduz o speedup:
  - 2.3X (8/3.5) versus. 4X (8/4)

Lições sobre execução em Pipelining (2/2)



- Imagine que novas máquinas reduzem os tempos de lavagem e secagem para 20 minutos. Será que isto vai melhorar o desempenho global?
- Não! O pipeline é limitado pela duração da etapa mais lenta.
- Desequilíbrios na duração dos estágios da linha implicam uma redução de speedup.

Estágios de Pipeline no MIPS

1) IFtch: Instruction Fetch, Incrementa PC

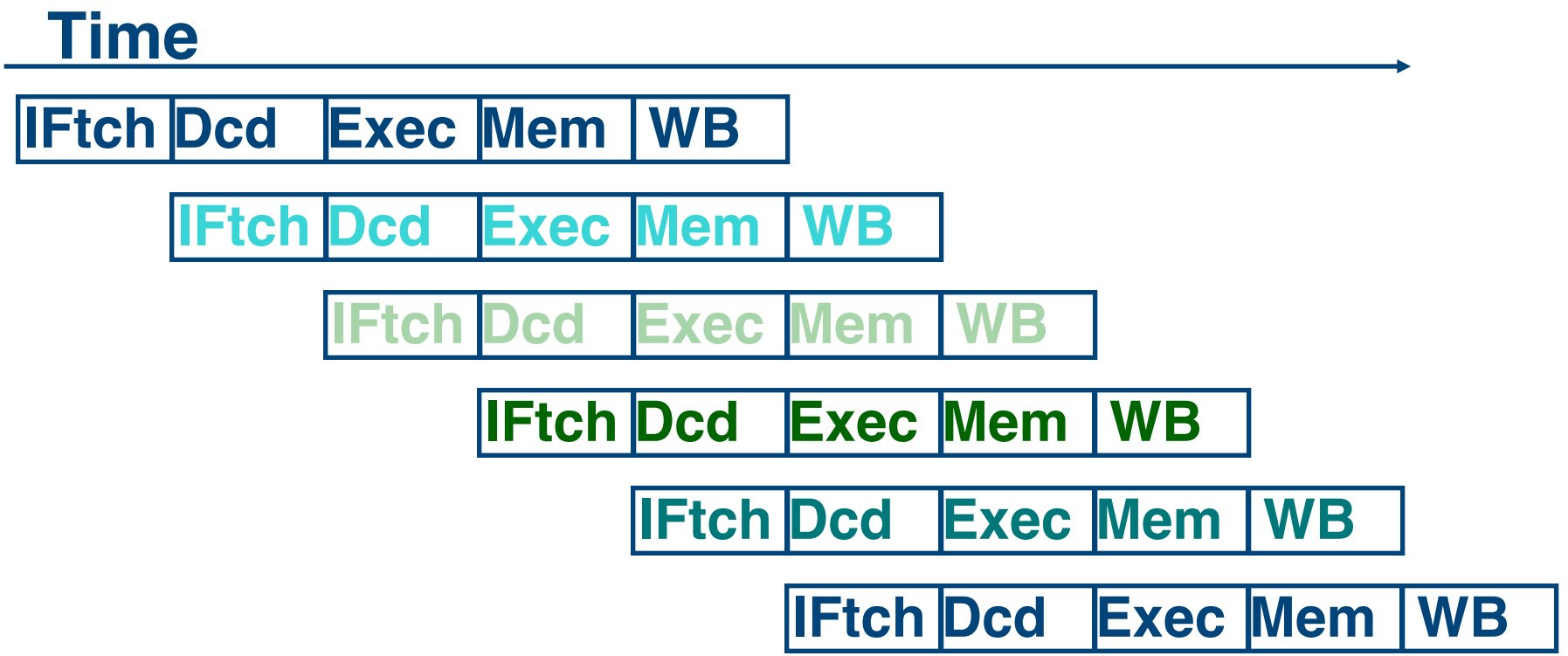
2) Dcd: Instruction Decode, Lê Registos

3) Exec:
   Mem-ref: Calcula endereços
   Arith-log: Executa a operação

4) Mem:
   Load: Leitura de dados da memória
   Store: Escrita de dados para a memória

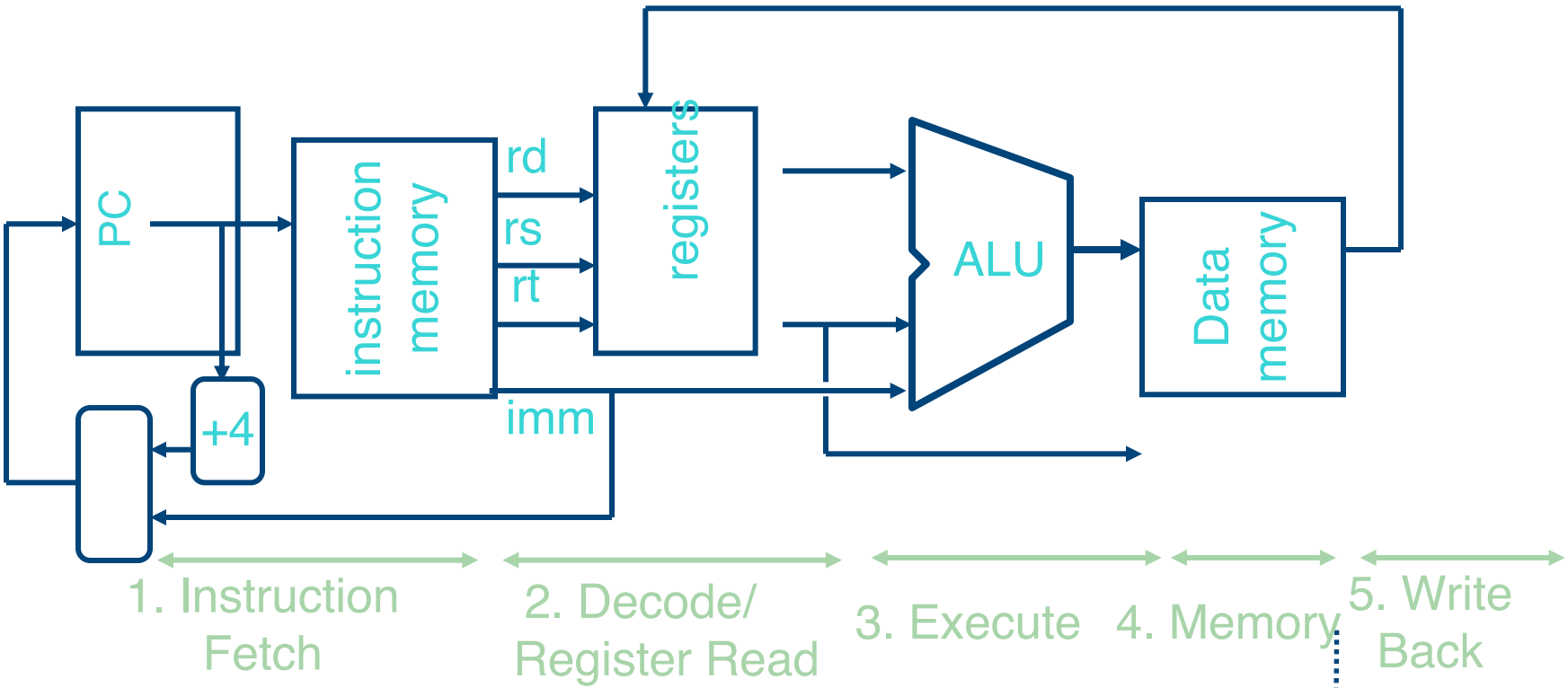
5) WB: Write Data Back to Register

Representação da Execução em Pipeline

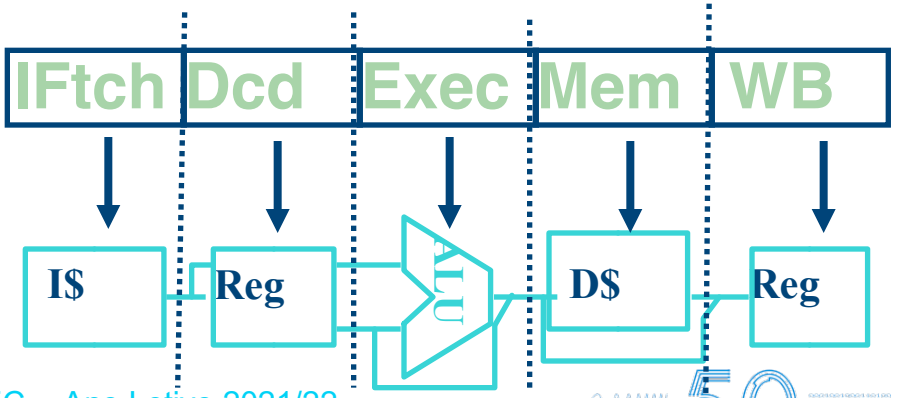


- Cada instrução tem que passar pelo mesmo número de etapas, designadas como "estágios" do pipeline. Já vimos que algumas das instruções ficam inativas em alguns dos estágios.

Revisão: Datapath para o MIPS

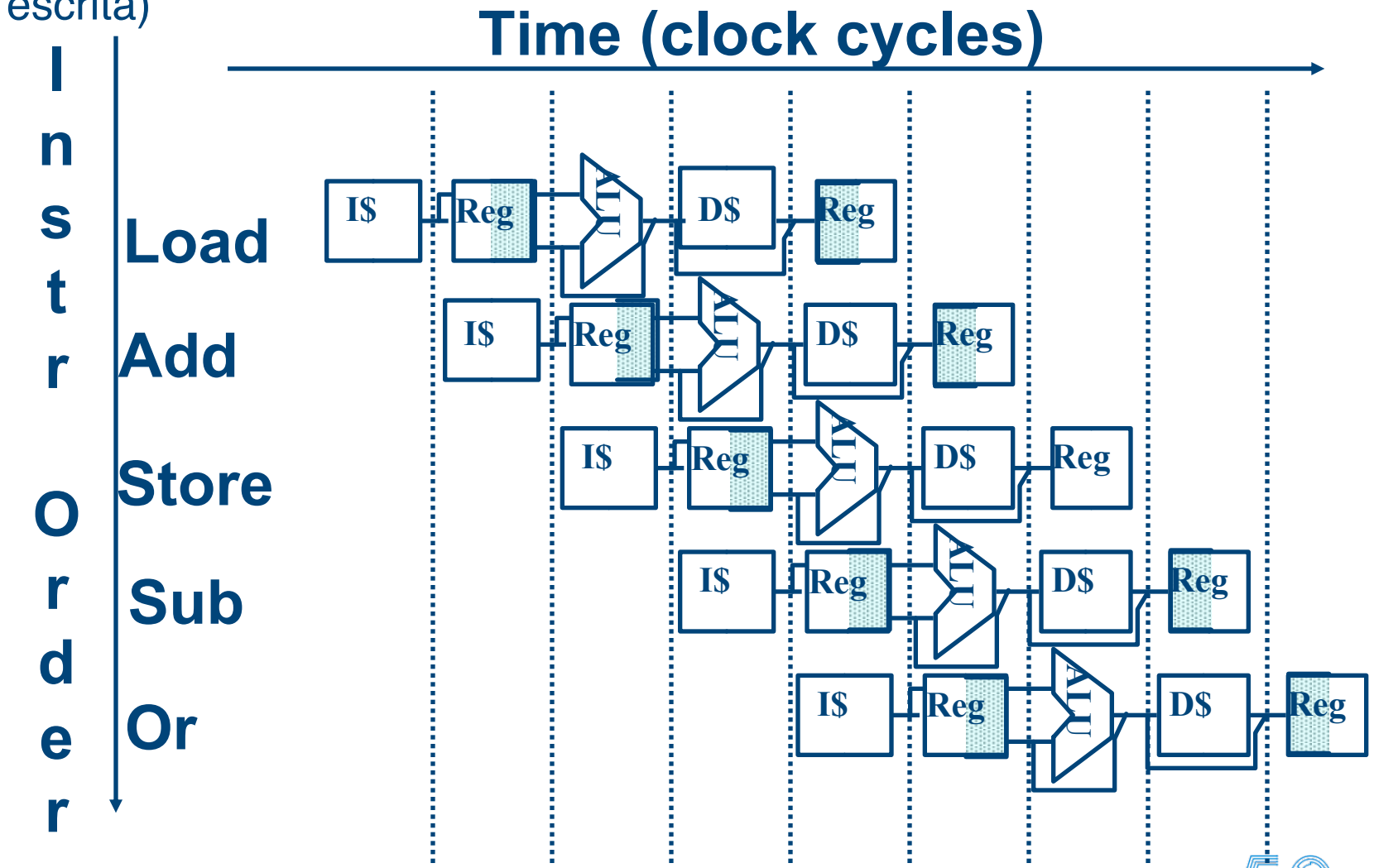


- Use as figuras do datapath para representar o pipeline

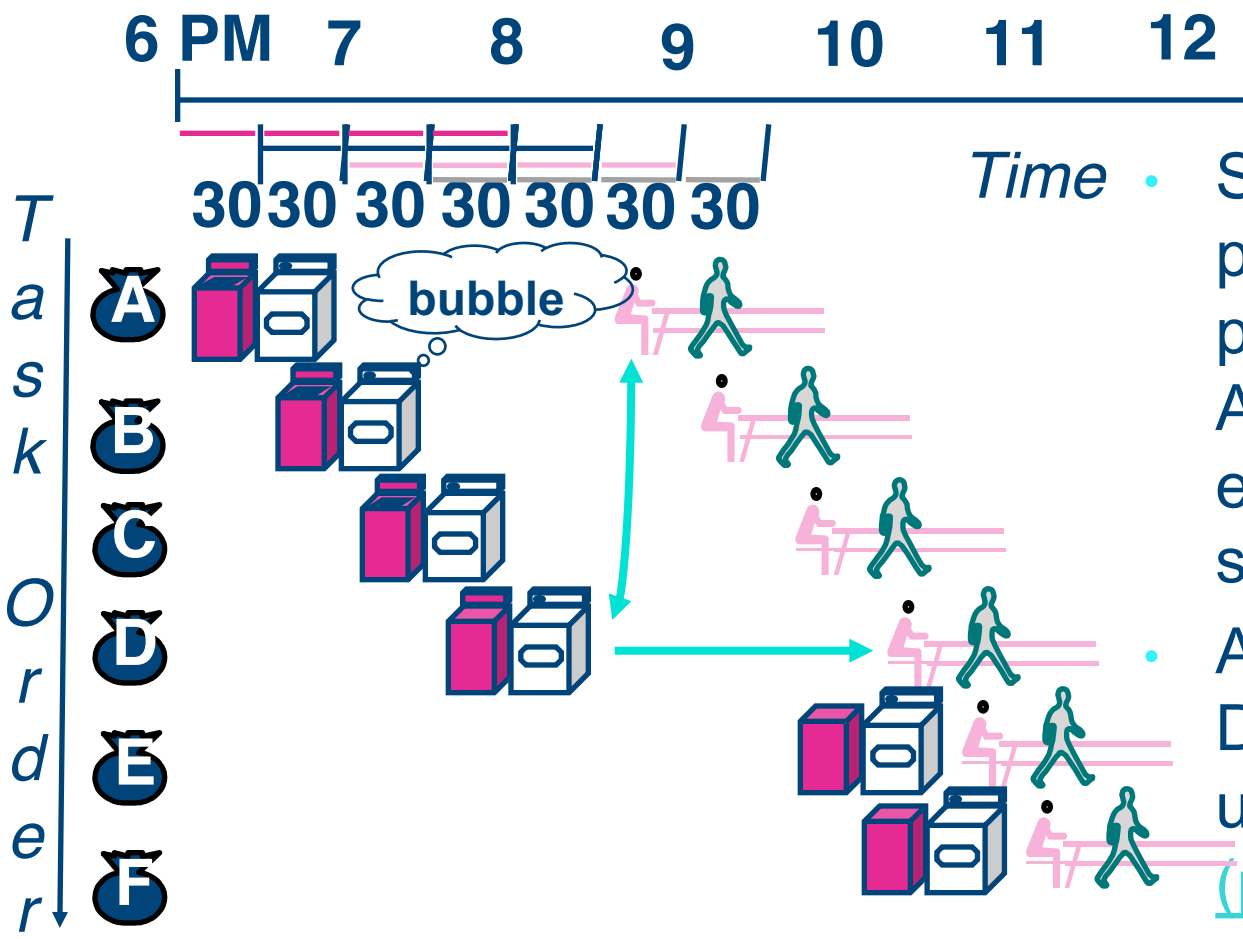


Representação Gráfica do Pipeline

(Nos Registos, sombra do lado direito significa leitura, e no lado esquerdo escrita)



Conflitos no Pipeline (Pipeline Hazards)

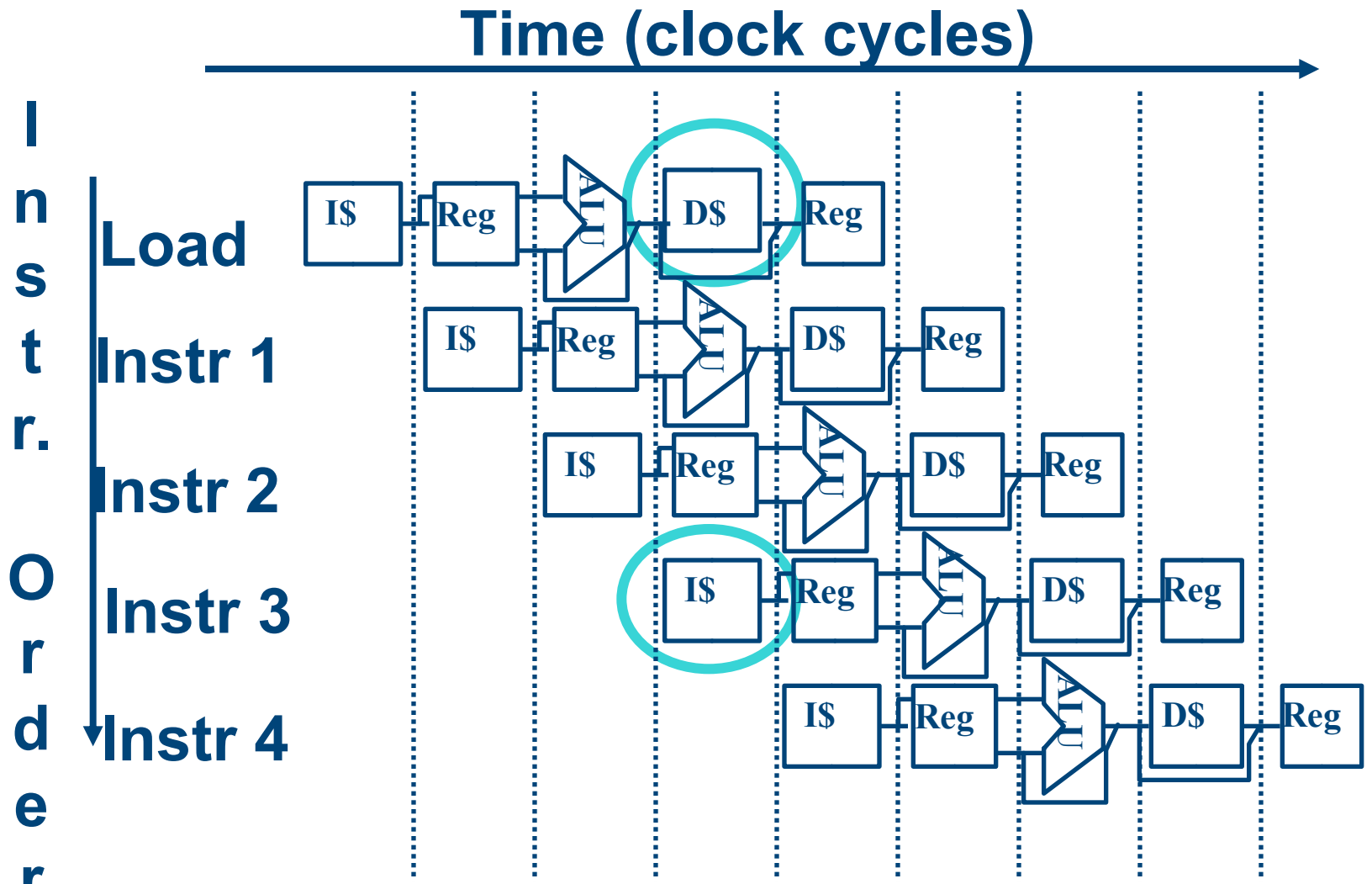


- Suponha que o par de uma peúga no saco A seguiu por engano no saco D.
- A depende de D; isto causa um stall (paragem) no estágio de "dobragem"

Problemas no Pipeline

- Limitações da técnica de Pipelining: podem ocorrer conflitos que bloqueiam a instrução seguinte, evitando que ela seja executada no ciclo de relógio previsto
  - Conflitos Estruturais (*structural hazards*): O HW físico não permite suportar determinadas combinações de instruções (e.g. uma única pessoa não pode dobrar e arrumar a roupa simultaneamente)
  - Conflitos de Controlo (*control hazards*): Quando aparecem saltos potenciais no fluxo de execução (instruções de branch) existe incerteza quanto às instruções que se seguem. Isto causa paragens e poderá levar a uma limpeza do pipeline e retrocesso na execução ("flush").
  - Conflitos de Dados (*data hazards*): Instruções que dependem do resultado de outras instruções que ainda estão no pipeline (o caso do par de peúgas)
- Qualquer um destes conflitos conduz a situações de paragem ("stalls"), criando "bolhas" no pipeline.

Conflito Estrutural #1: acesso a memória (1/2)

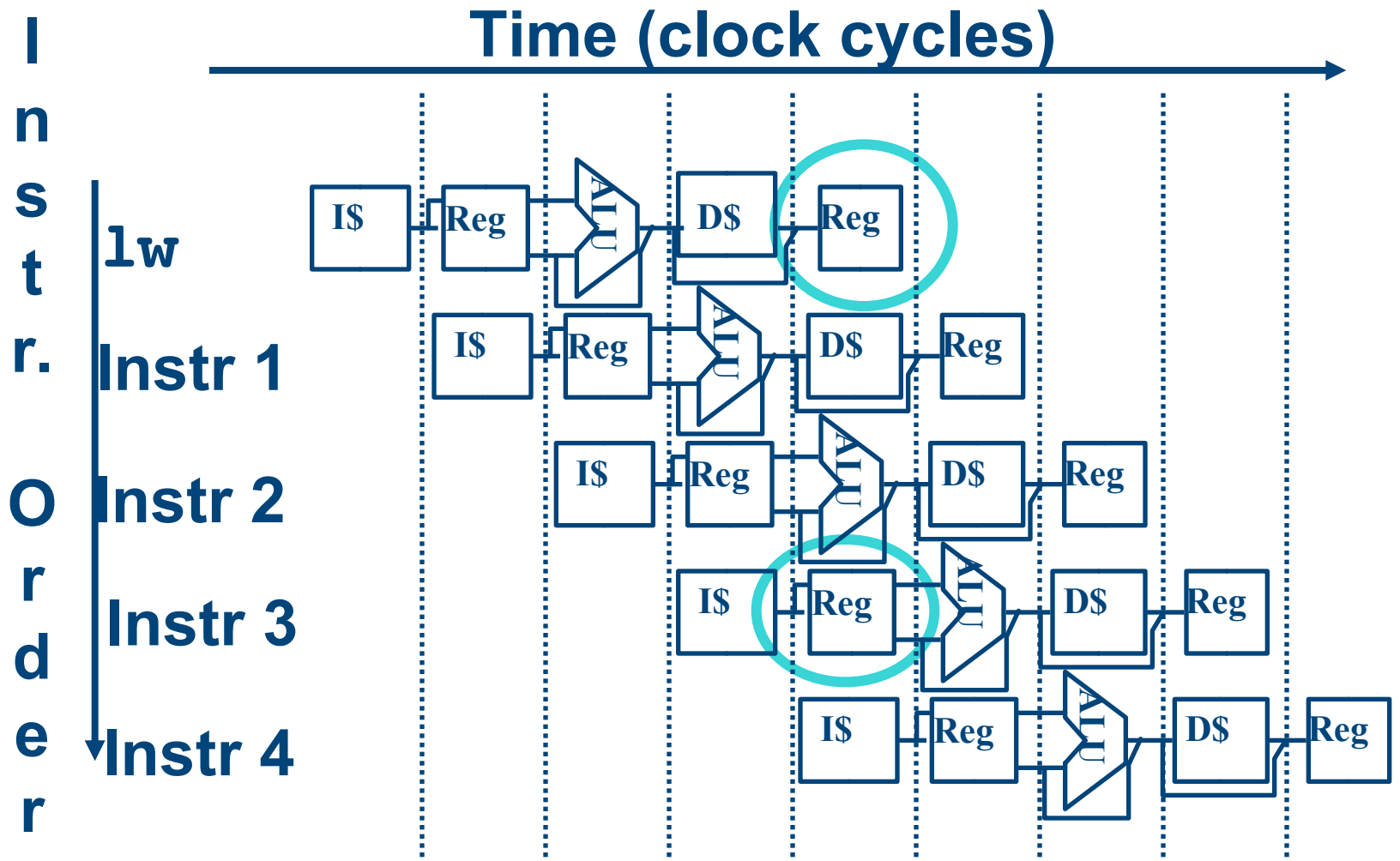


Duas leituras de memória no mesmo clock cycle

Conflito Estrutural #1: acesso a memória (2/2)

- Solução:
  - Replicar as memórias: ineficiente e não exequível (veremos isto melhor quando falarmos da hierarquia de memória)
  - Simular duas memórias usando dois níveis de Cache Level 1 (uma cache é uma pequena cópia temporária da memória, com a informação que foi usada recentemente)
  - Neste caso teremos uma Instruction Cache e uma Data Cache, sendo o HW de controlo mais complexo no caso de haver dois "*cache misses*" simultâneos.

Conflito Estrutural #2: registos (1/2)



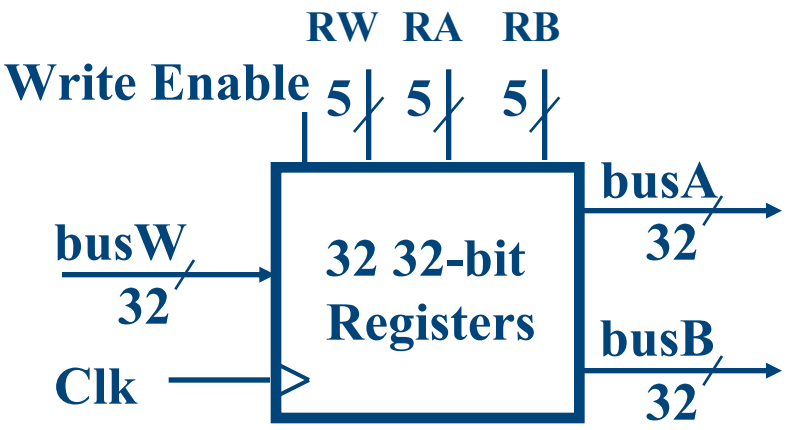
Podemos ler e escrever simultaneamente em registos?

Conflito Estrutural #2: registos (2/2)

- Existem duas soluções diferentes para este problema:
  - 1) O acesso ao *ficheiro* de registos é muito rápido: demora menos de metade do tempo da etapa ALU. Assim,
    - Podemos escrever no *reg. file* durante a primeira metade do ciclo de relógio
    - Ler os registos na segunda metade do ciclo
    - *Será que faria sentido fazer primeiro a leitura e depois a escrita?*
  - 2) Implementar o RegFile em HW definindo portos independentes para leitura e escrita (já vimos isto).
- Resultado: É possível escrever e ler os registos no mesmo ciclo de relógio

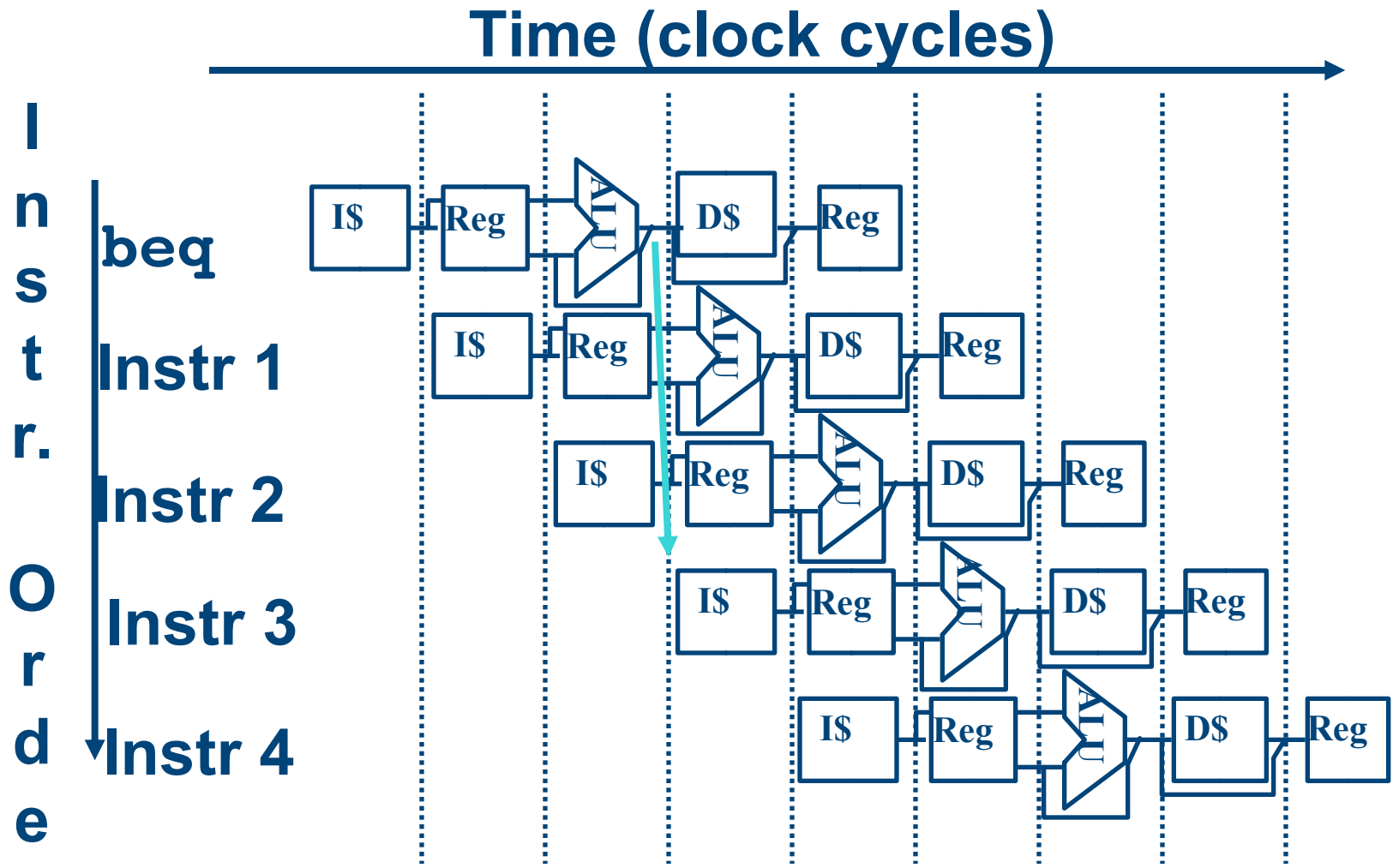
Revisão: Register File

- Consiste em 32 registos:
  - 2 buses de saída de 32-bit (busA and busB)
  - 1 bus de entrada de 32-bit: busW
- O registo é selecionado por:
  - RA (número) seleciona o registo para busA
  - RB (número) seleciona o registo para busB
  - RW (número) seleciona o registo a ser escrito via busW quando Write Enable é 1



- Repare que é possível fazer leitura e escrita simultaneamente
- Clock input (clk)
  - O clk input só é importante para operações de escrita
  - Na leitura o "register file" comporta-se como lógica combinacional:
    - RA ou RB válido $\Rightarrow$ busA ou busB válido depois de "access time".

Conflitos de Controlo: branching (1/7)



Quando é feita a comparação que decide o branch?

Conflitos de Controlo: branching (2/7)

- Até aqui assumimos a decisão de salto é tomada quando é feita comparação no estágio ALU.
  - Assim, existem sempre duas instruções depois do branch que entram no pipeline. Se houver salto essas instruções não são para executar, perdendo-se dois ciclos.

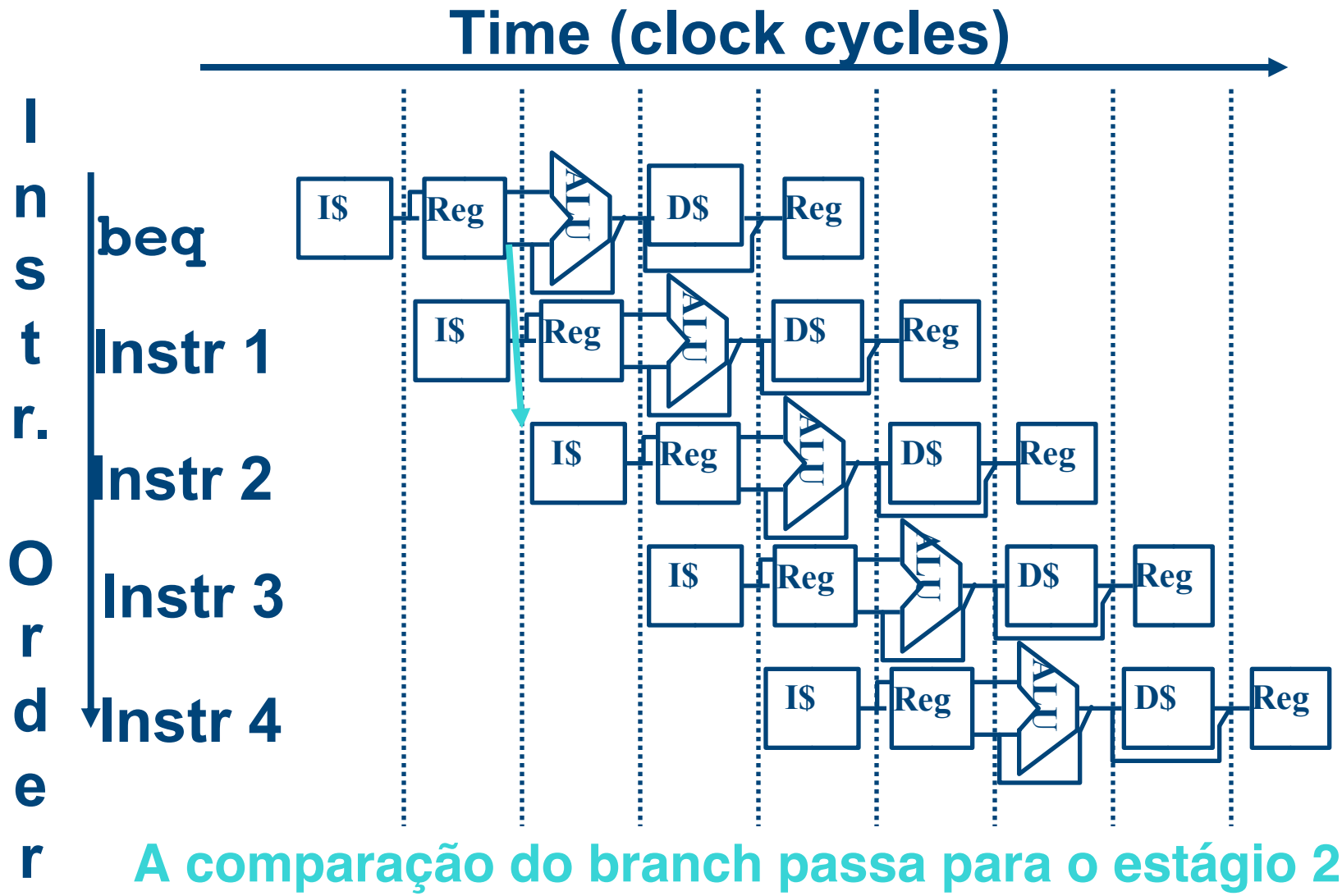
- Idealmente um branch deve funcionar da seguinte forma:
  - Se o salto não ocorrer, a execução deve continuar de forma normal sem perda de tempo
  - Se o salto ocorrer, as instruções a seguir ao branch não devem ser executados, passando a execução para o ponto indicado pelo “label”

Conflitos de Controlo: branching (3/7)

- Solução 1 : Paragem no pipeline
  - Inserir instruções “no-op” a seguir ao branch, ou não fazer fetch de instruções até a decisão de salto ser tomada (stall durante 2 ciclos de relógio).
  - Desvantagem: as instruções de branch passam a demorar 3 ciclos de relógio em vez de um único ciclo

- Otimização #1:Implementar um comparador para “branches no estágio 2
  - Assim que uma instrução é descodificada, verifica-se se o opcode corresponde a um branch. Neste caso a decisão é imediatamente tomada e o PC é ajustado de forma adequada.
  - Vantagem: Como o branch é completado no estágio 2, só a instrução a seguir é que entra no pipeline, bastando um único “nop”
  - Nota: A instrução de “branch” está inativa nos estágios 3, 4 e 5.

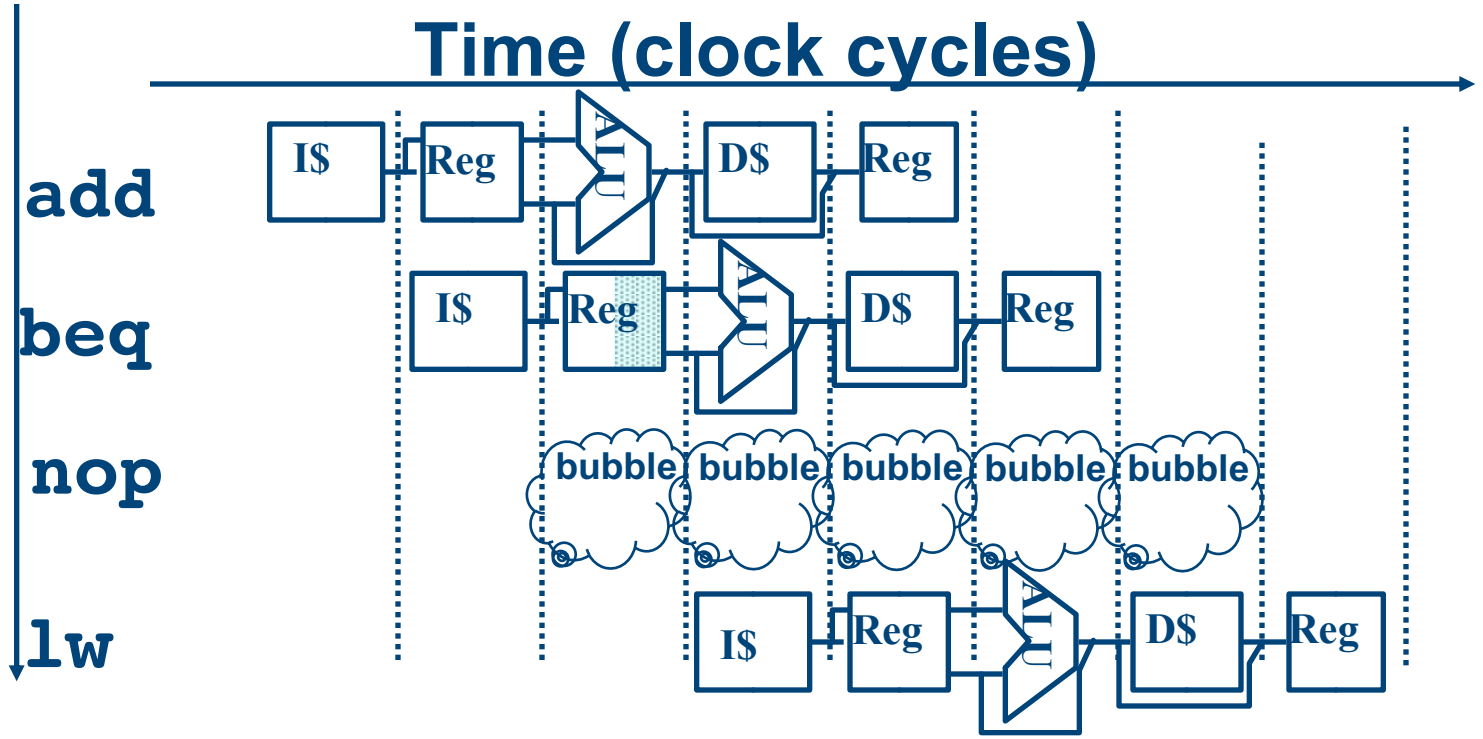
Conflitos de Controlo: branching (4/7)



A comparação do branch passa para o estágio 2.

Conflitos de Controlo: branching (5/7)

- O utilizador/programador insere uma instrução “no-op”



- Impacto: 2 ciclos de relógio / instrução de branch ⇒ ainda é lento

Conflitos de Controlo: branching (6/7)

- Otimização #2: Redefinir o comportamento do branch
  - Definição até agora: se o salto acontecer, nenhuma das instruções a seguir ao "branch" deve ser acidentalmente executada.
  - Nova definição: independentemente do salto acontecer, ou não, a instrução a seguir ao branch deve ser sempre executada (chama-se a isto branch-delay slot)

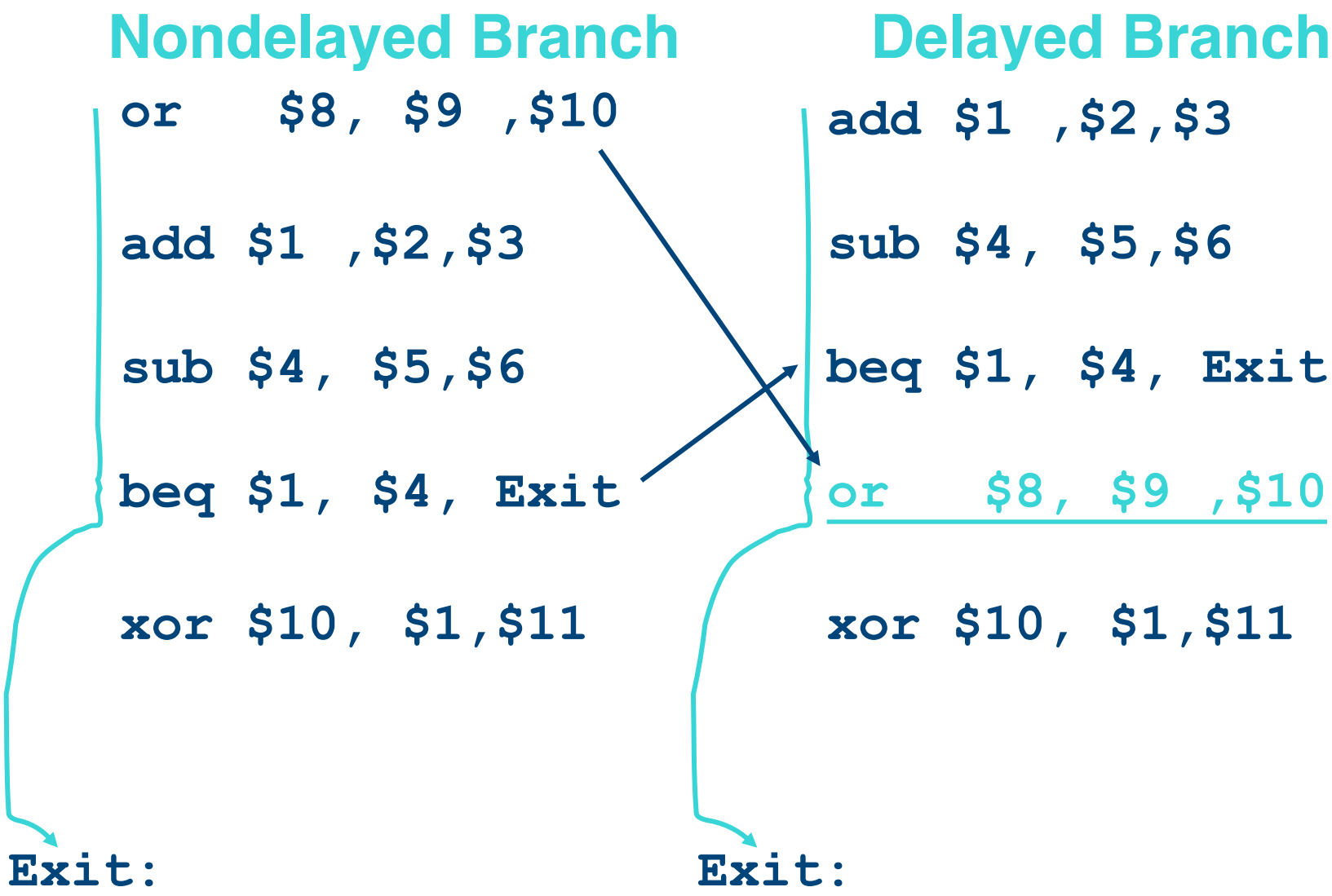
- O termo "delayed branch" significa que a instrução a seguir ao branch é sempre executada

- Esta otimização é utilizada no MIPS

Conflitos de Controlo: branching (7/7)

- Como funciona o Branch-Delay Slot?
  - Worst-Case Scenario: colocamos uma instrução “no-op” no branch-delay slot
  - Solução mais otimizada: podemos colocar no branch-delay slot, uma instrução originalmente antes do “branch”, que pode ser colocada depois sem afetar o correto fluxo de execução.
    - A reordenação das instruções é muitas vezes utilizada para acelerar os programas
    - O compilador tem que ser muito “esperto” para fazer esta reordenação de forma automática
    - Em cerca de 50% dos casos é possível encontrar uma instrução para preencher o “delay slot”, evitando-se completamente o conflito de controlo
    - Repare que os jumps têm o mesmo problema dos branches …

Exemple: Nondelayed vs. Delayed Branch

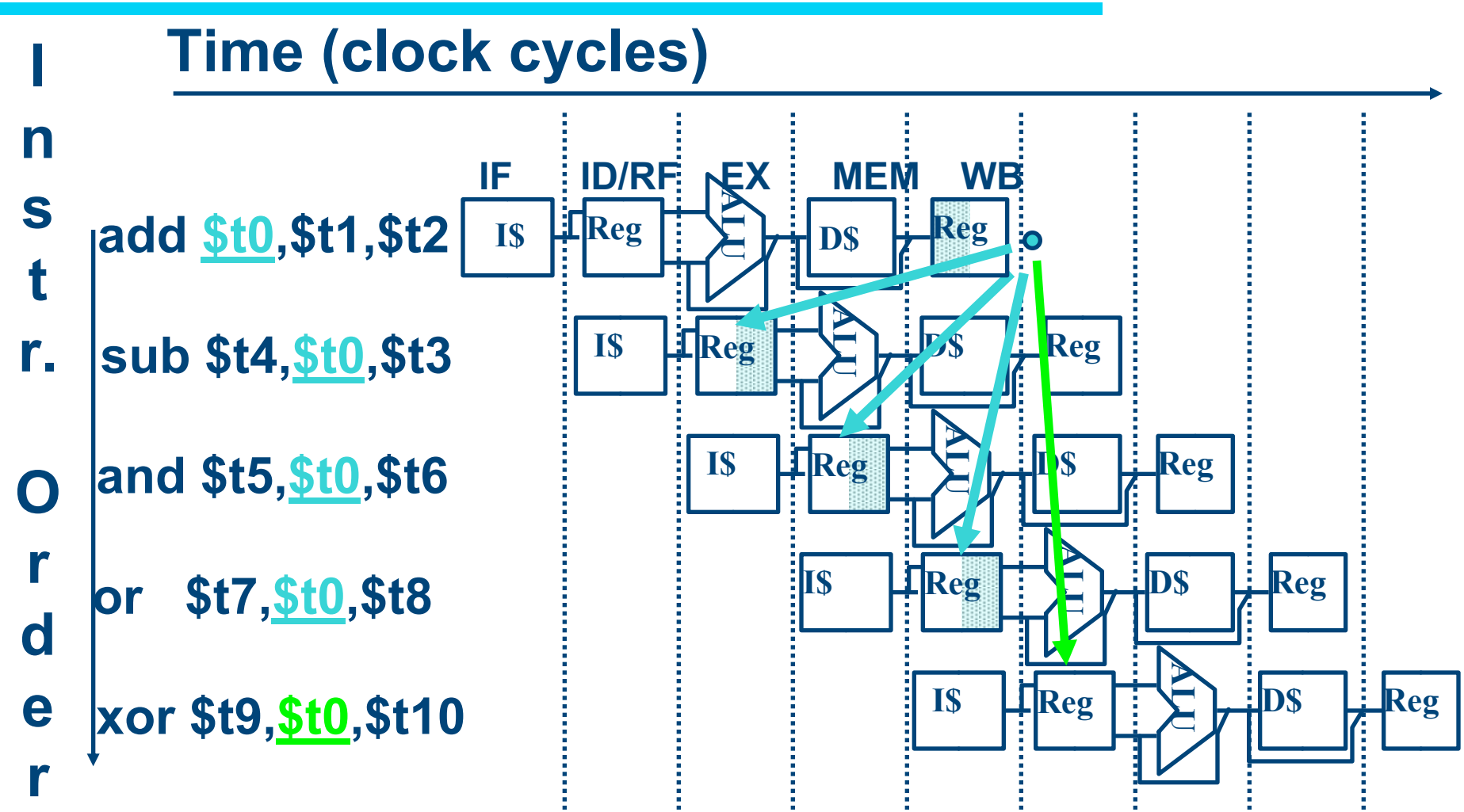


Conflitos de Dados (1/2)

- Considere a seguinte sequência de instruções

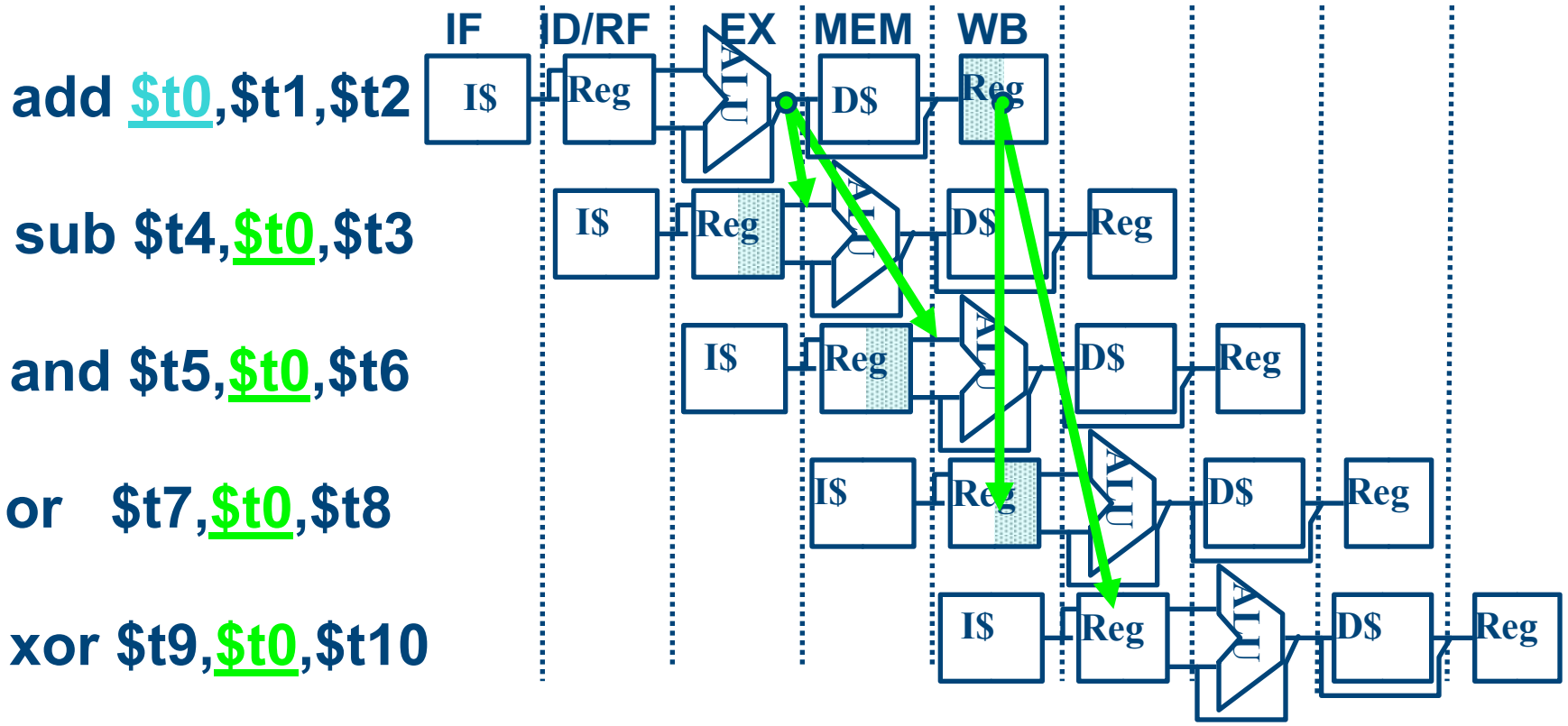
```
add $t0, $t1, $t2
sub $t4, $t0 ,$t3
and $t5, $t0 ,$t6
or  $t7, $t0 ,$t8
xor $t9, $t0 ,$t10
```

Conflitos de Dados (2/2)



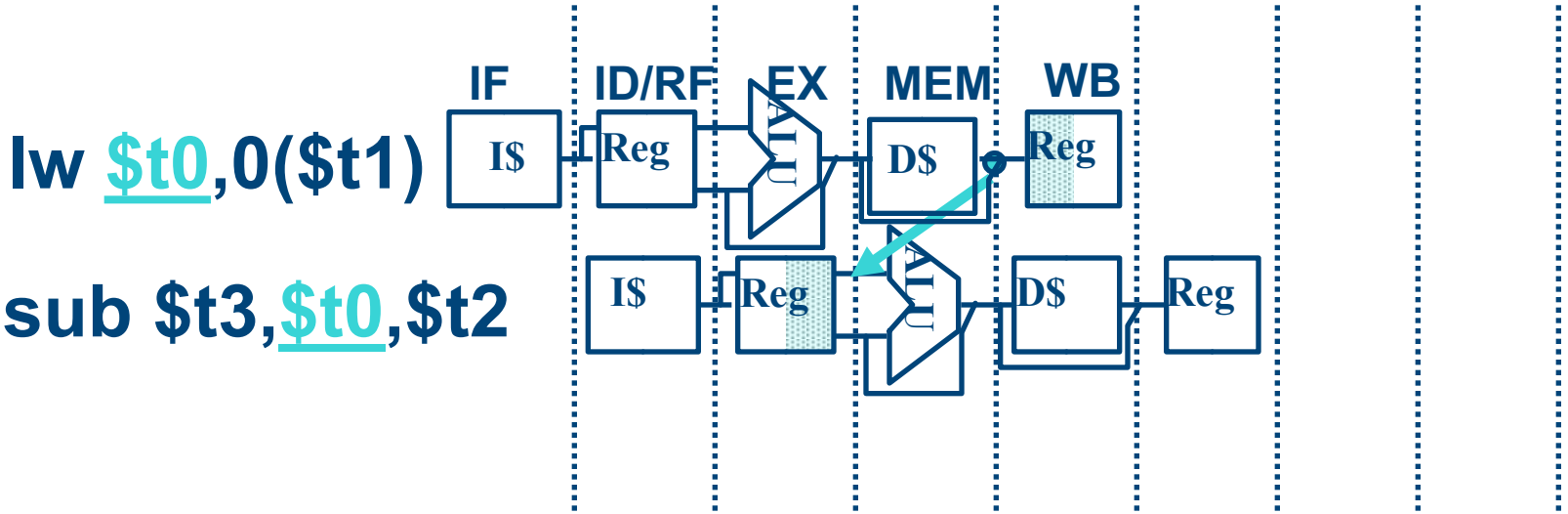
Fluxos de informação no sentido contrário ao tempo geram conflitos de dados

Solução para Conflitos de Dados: forwarding



- Repare que o valor a ser escrito em $t0 está disponível à saida da ALU
- Podemos fazer FORWARD de um estágio para outro de forma a evitar conflitos
- Repare que o conflito no “or” é evitado pelo HW do RegFile (escrita antes da leitura)

Conflitos de Dados: loads (1/4)



- Neste caso o valor para o “sub” não é conhecido antes de ser necessário
- A técnica de “forwarding” não resolve a situação
- É necessário colocar um “stall” depois do load, e depois fazer forwarding (mais hardware específico para realizar esta operação)

Conflitos de Dados: loads (2/4)

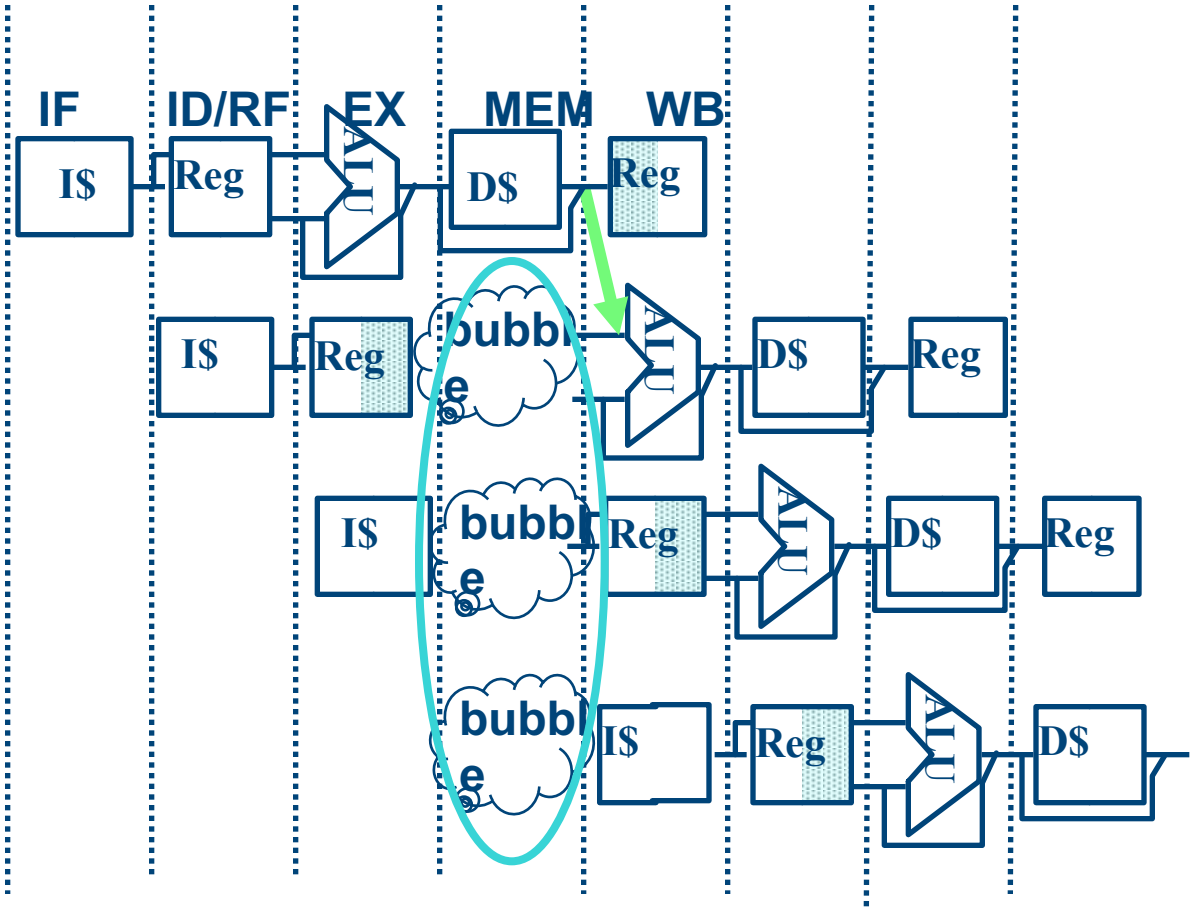
- O próprio HW faz “stall” do pipeline: chama-se a isto “interlock”

lw $t0, 0($t1)

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Conflitos de Dados : loads (3/4)

- A slot depois do load é chamada "load delay slot"
- Se a instrução utilizar o resultado do load, então o hardware faz um interlock para fazer parar o pipeline durante um ciclo de relógio (stall).
- Repare que o HW consegue saber se deve ou não colocar o "stall". Já identificou o load, e a instrução também já foi descodificada sendo os operandos conhecidos.
- O compilador pode fazer um reordenamento de forma a que a instrução na "load delay slot" não dependa do load. Neste caso evita-se a bolha no pipeline.
- Deixar o HW fazer o "interlock" é equivalente a colocar uma instrução "no-op" a seguir ao load (só que esta última solução implica mais espaço para código)

Conflitos de Dados: loads (4/4)

- Stall é equivalente a nop

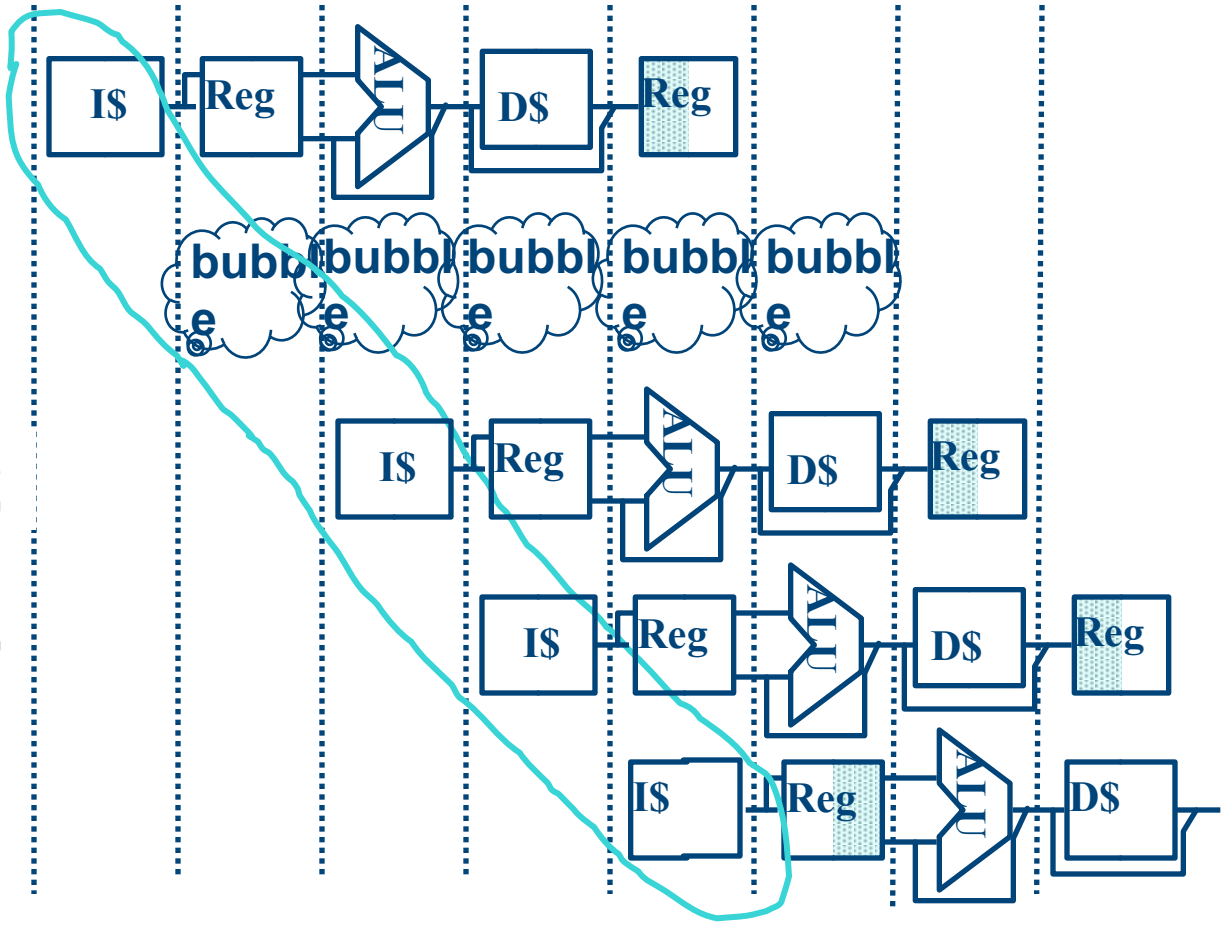
lw $t0, 0($t1)

nop

sub $t3,$t0,$t2

and $t5,$t0,$t4

or $t7,$t0,$t6



Curiosidade histórica

- A primeira versão do MIPS caracterizava-se por não existir nenhum mecanismo de “interlock” por hardware. A resolução de conflitos tinha que ser feita ao nível do compilador

  Microprocessor without
  Interlocked
  Pipeline
  Stages

- ... e não a interpretação do acrónimo “Millions of Instructions Per Second” que depois muita gente fez.

Sumário: Pipelining (1/2)

- Pipelining em circunstâncias ideais
  - Cada estágio executa uma parte da instrução num ciclo de relógio
  - Assim, o processador termina a execução de uma instrução por cada ciclo de relógio.
  - Em média a execução torna-se muito mais rápida.
- Porque é que isto funciona?
  - Em geral, a semelhança e uniformidade das instruções permitem-nos usar os mesmos estágios para executar cada uma delas (filosofia dos processadores RISC).
  - A divisão em estágios/etapas é equilibrada de forma a que cada um deles tenha aproximadamente a mesma duração: minimizar o desperdício de tempo.
- O pipelining é uma GRANDE IDEIA, sendo muito utilizada.

Sumário: Pipelining (2/2)

- Quais são os problemas e limitações inerentes a fazer pipelining?
  - Conflitos Estruturais: Tratam-se de conflitos devidos à falta de recursos físicos. Imagine que só temos uma cache que é partilhada por dados e instruções? ⇒ A solução passa por ampliar os recursos de HW disponíveis
  - Conflitos de Controlo: Nas instruções de salto (branches e jumps) não sabemos qual é a instrução que se segue. ⇒ Solução possível: Delayed branch, ou seja reordenar as instruções para colocar uma instrução anterior ao branch na "delay slot" (se isto não for possível o compilador coloca um no-op)
  - Conflitos de Dados: Fluxo de informação no sentido contrário ao tempo / estágios do pipeline.
    - Forwarding evita muitos destes conflitos
    - Load delay slot / interlock é necessário porque forwarding não resolve

QUIZ

A. Graças à execução em pipeline, eu sou capaz de reduzir o tempo de lavagem da minha camisa.

B. Pipelines mais longos são sempre vantajosos! (havendo menos trabalho por estágio é possível acelerar o relógio).

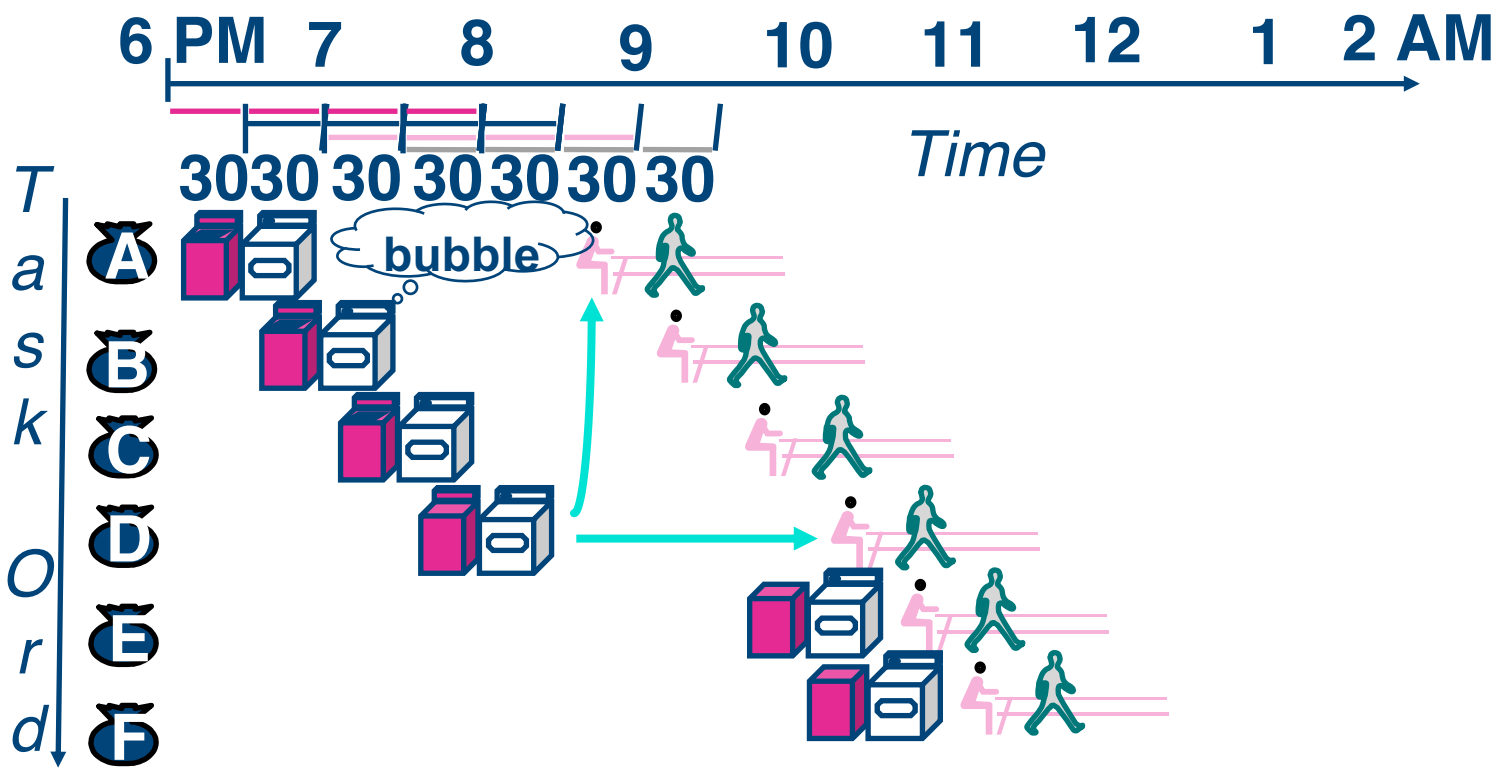
C. Podemos utilizar os compiladores para nos ajudar a evitar os conflitos de dados através de um re-ordenamento das instruções.

	ABC
0:	FFF
1:	FFT
2:	FTF
3:	FTT
4:	TFF
5:	TFT
6:	TTF
7:	TTT

Mas a história não termina aqui ...

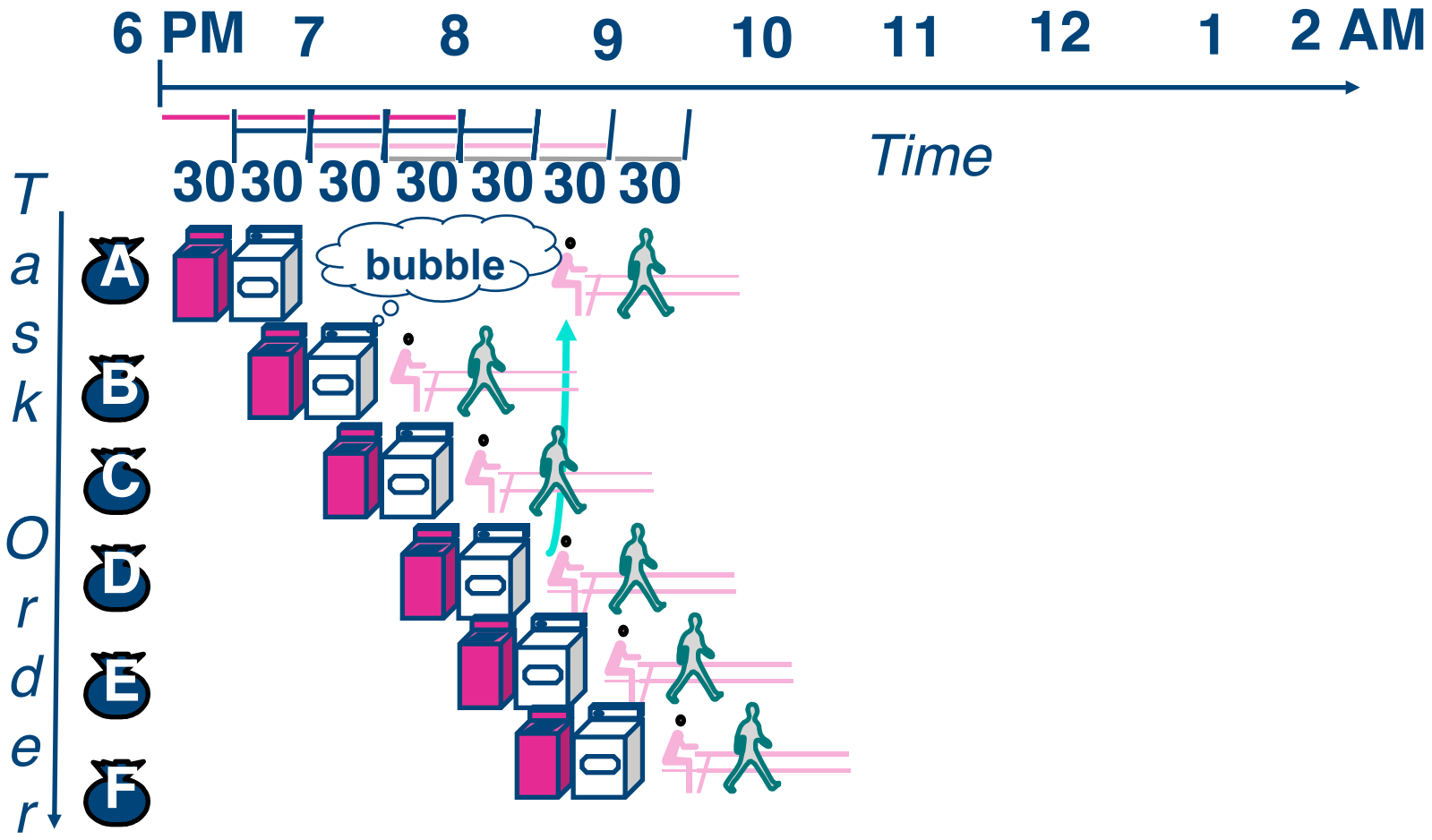
- Desempenhos mais agressivos com processadores super-escalares:
  - Exemplo: placas gráficas com vários pipelines em paralelo
- Execução fora de ordem
- Todos estes mecanismos exigem replicação de recursos de HW

Pipeline Hazard: O problema de juntar as peúgas



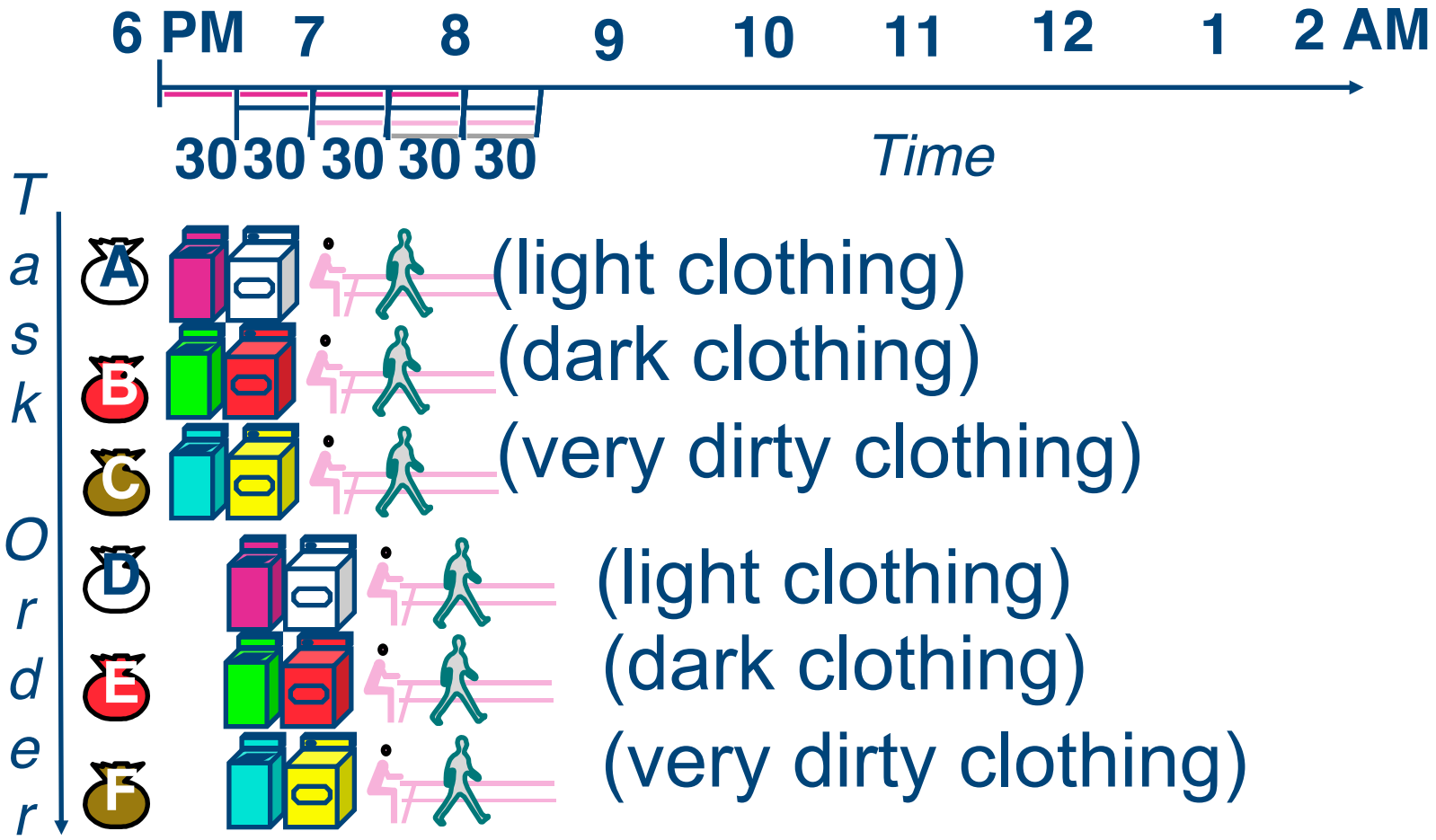
- A depende de D; causando um stall na dobragem;
- Note que isto é diferente dos conflitos que vimos até agora ... nunca tivemos uma instrução a depender do resultado de outra instrução que vem a seguir
- Chama-se a isto execução fora de ordem

Execução Fora de Ordem: Não Espere!

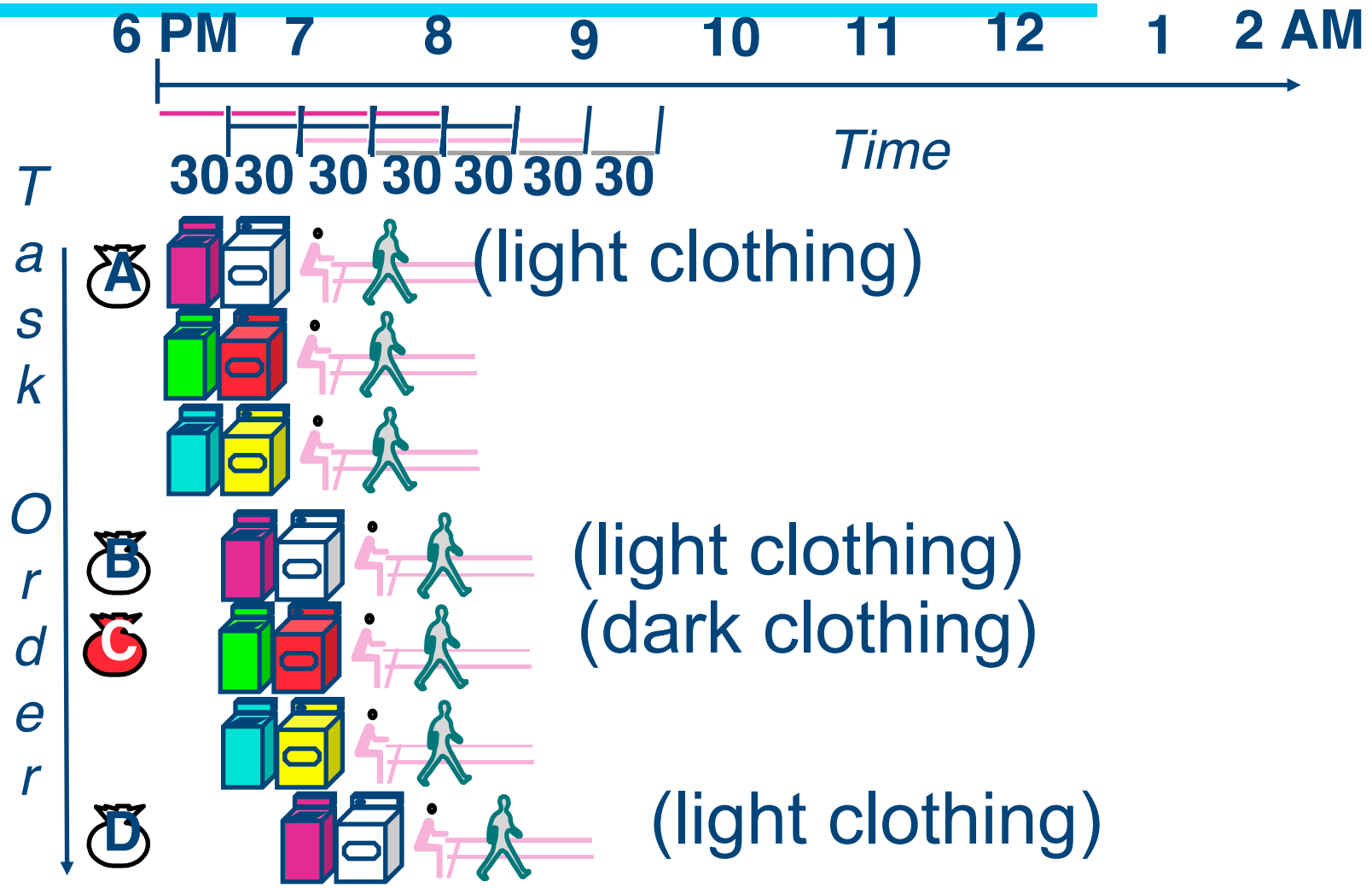


- A depende de D; continuamos com o resto; são precisos mais recursos

Execução Superscalar : Estágios Paralelos



Execução Superscalar: desperdício de recursos



Task mix underutilizes extra resources

QUIZ (1/2)

- Assuma 1 instr/clock, delayed branch, 5 estágios de pipeline, forwarding, interlock nos conflitos de dados envolvendo o load. O loop tem 10^3 iterações (pipeline cheio).

2. (data hazard so stall)

```
Loop:  1. lw     $t0, 0($s1)
       3. addu   $t0, $t0, $s2
       4. sw     $t0, 0($s1)
       5. addiu  $s1, $s1, -4
       6. bne    $s1, $zero, Loop
       7. nop
```

(delayed branch so exec. nop)

- Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 5 6 7 8 9 10

QUIZ (2/2)

- Assuma 1 instr/clock, delayed branch, 5 estágios de pipeline, forwarding, interlock nos conflitos de dados envolvendo o load. O loop tem 10^3 iterações (pipeline cheio). Reescreva o código para otimizar o tempo de execução

```
Loop:       lw      $t0, 0($s1)
            addu    $t0, $t0, $s2
            sw      $t0, 0($s1)
            addiu   $s1, $s1, -4
            bne     $s1, $zero, Loop
            nop
```

- Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 5 6 7 8 9 10

QUIZ (2/2)

- Reescreva o código para otimizar o tempo de execução

(no hazard since extra cycle)

```
Loop:  1.  lw      $t0, 0($s1)
       2.  addiu   $s1, $s1, -4
       3.  addu    $t0, $t0, $s2
       4.  bne     $s1, $zero, Loop
       5.  sw      $t0, +4($s1)
```

(modified sw to put past addiu)

- Qual é a duração em ciclos de relógio para a execução de uma iteração do ciclo?

1 2 3 4 5 6 7 8 9 10

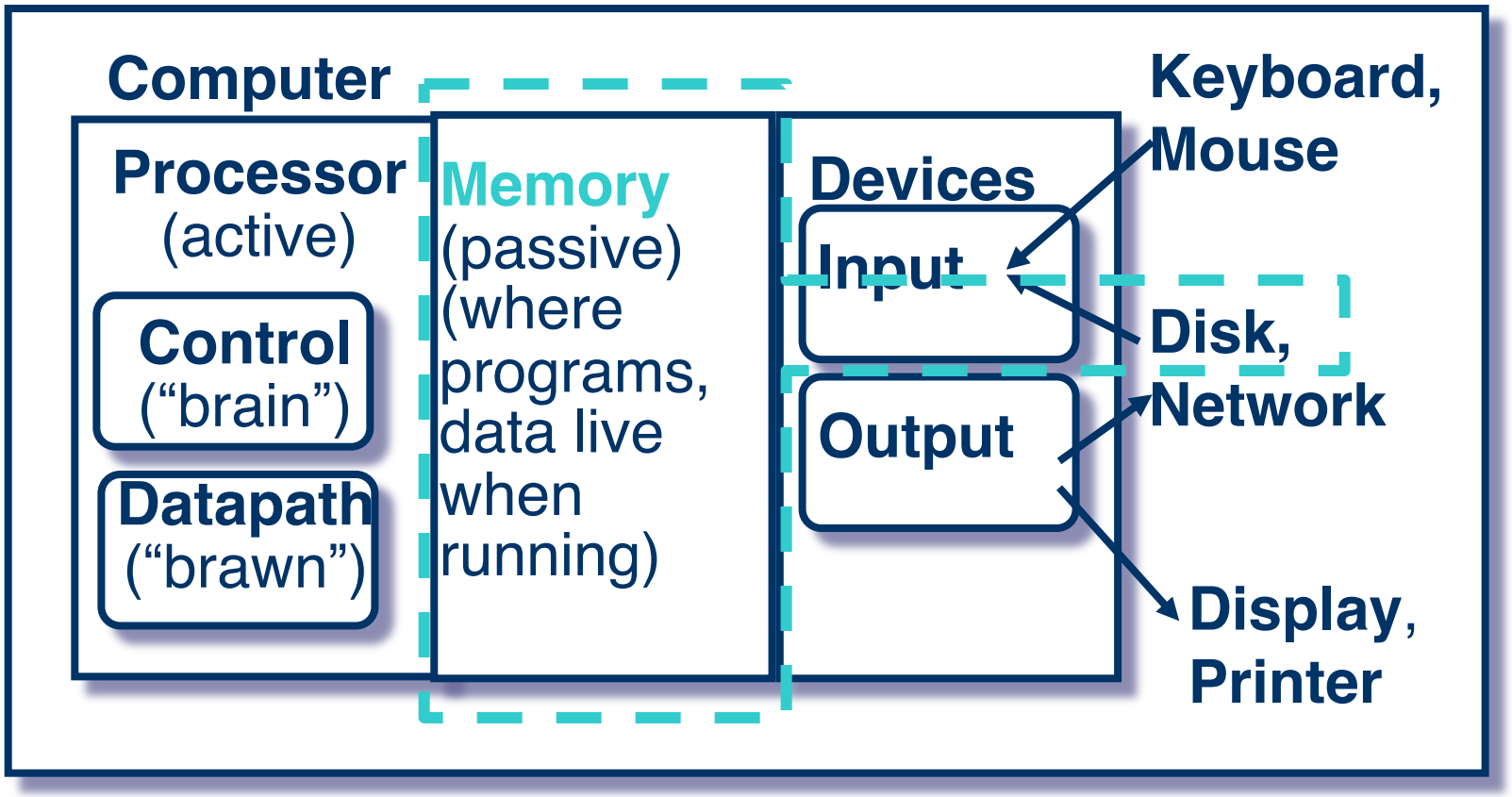
Para saber mais ...

- P&H - Capítulos 6.1 a 6.6
- É essencial que estudem pelo livro!

Introdução à Arquitetura de Computadores
- Hierarquia de Memória -

Sistemas de Microprocessadores 2021/22

The Big Picture



Memory Hierarchy

O armazenamento em computadores:

Processador

- Guarda dados num ficheiro de registos (~100 Bytes)
- Registo são acedidos numa escala de nanossegundos

Memória (chamamos de Memória Principal)

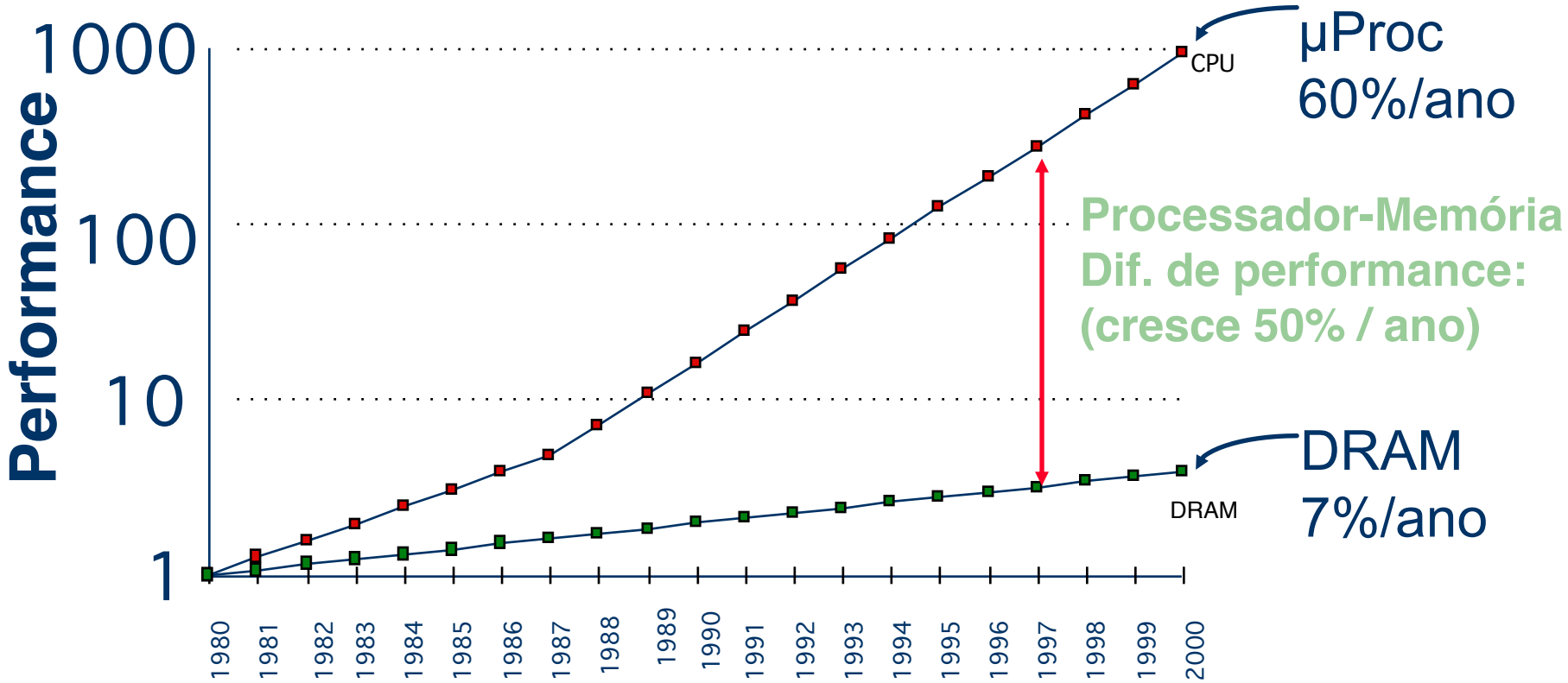
- Mais capacidade do que registos (~Gbytes)
- Tempo de acesso ~50 a 100 nanossegundos
- Centenas de clock cycles por acesso a memória?!

Disco

- Enorme capacidade (virtualmente sem limite)
- Muito lento: escala de milisegundos

Motivação: por que se usam caches?

1989 first Intel CPU with cache on chip
1998 Pentium III has two levels of cache on chip



Cache de Memória

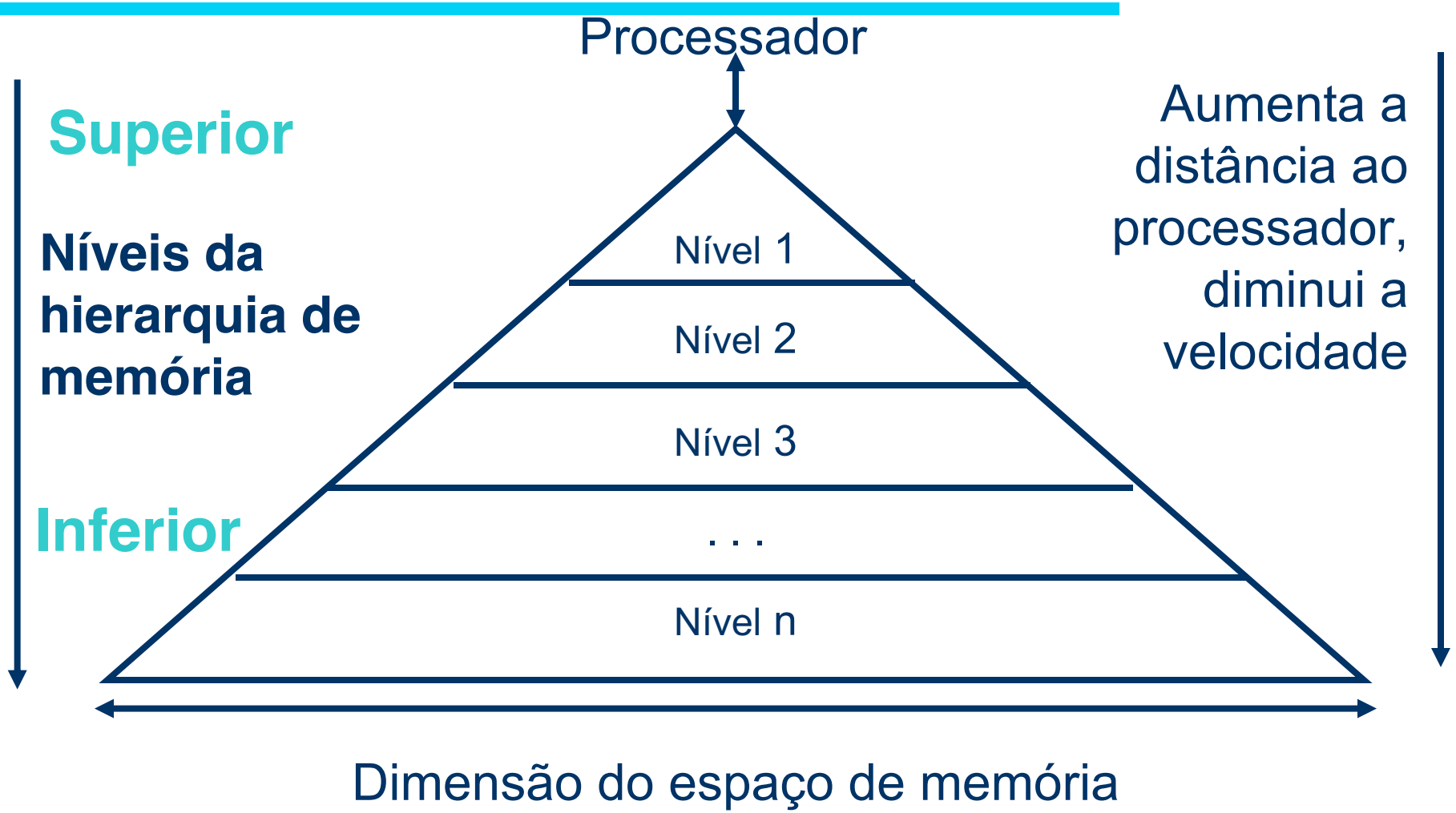
Diferença entre as velocidades do processador e da memória induz num outro nível adicional: a memória cache

Implementada com a mesma tecnologia de circuito integrado do processador (CPU), usualmente integrada no mesmo chip: mais rápida mas muito mais dispendiosa do que a memória DRAM.

A cache é uma cópia de um subconjunto da memória principal.

Muitos processadores têm caches separadas para instruções e dados (recordar a discussão sobre conflitos estruturais).

Hierarquia de Memória



À medida que a memória passa aos níveis inferiores, a latência sobe e o preço por bit diminui.

Hierarquia de Memória

Se o nível se aproxima do processador, é

- mais pequeno
- mais rápido
- subconjunto dos níveis inferiores (contém menos dados, mas os mais recentes)

O nível mais baixo da hierarquia (usualmente o disco) contém todos os dados disponíveis (*ou vai além do disco?*)

A hierarquia de memória apresenta o processador com a ilusão de uma memória muito grande e muito rápida.

Analogia da memória: biblioteca (1/2)

Queremos escrever um documento numa mesa de uma Biblioteca

A Biblioteca é equivalente ao disco:

- Essencialmente capacidade ilimitada
- Muito lento o processo de conseguir um livro

A mesa é a memória principal

- Menor capacidade: significa que é necessário devolver o livro quando a mesa encher
- É mais fácil e mais rápido encontrar um livro em cima da mesa, se já o tivermos ido buscar

Analogia da memória: biblioteca (1/2)

Livros abertos na mesa são cache

- Menor capacidade: só se consegue ter um número muito limitado de livros na mesa; novamente, quando a mesa enche, é preciso fechar um livro
- Muito, muito mais rápido para obter dados

Ilusão criada: toda a Biblioteca pode ser usada na mesa

- Manter tantos livros recentemente usados quanto possível, porque é muito provável que se voltem a usar
- Manter o máximo de livros na mesa possível, porque é mais rápido do que ir à Biblioteca

Base da hierarquia de Memória

A cache contém cópias dos dados em memória que estão a ser usados.

A memória contém cópias dos dados em discos que estão a ser usados.

As caches exploram os princípios da localidade temporal e espacial.

- Localidade temporal: se estamos a usar agora, as probabilidades de o vir a usar daqui a pouco tempo, são elevadas
- Localidade espacial: se estamos a usar uma célula de memória agora, as probabilidades de usarmos no futuro próximo uma célula vizinha é elevada

Desenho das caches

Como organizamos as caches?

Para onde mapeamos cada endereço de memória?

(recordar que a cache é um subconjunto da memória, e por isso há vários endereços de memória que podem ser mapeados para a mesma localização na cache)

Como sabemos quais os elementos que estão na cache?

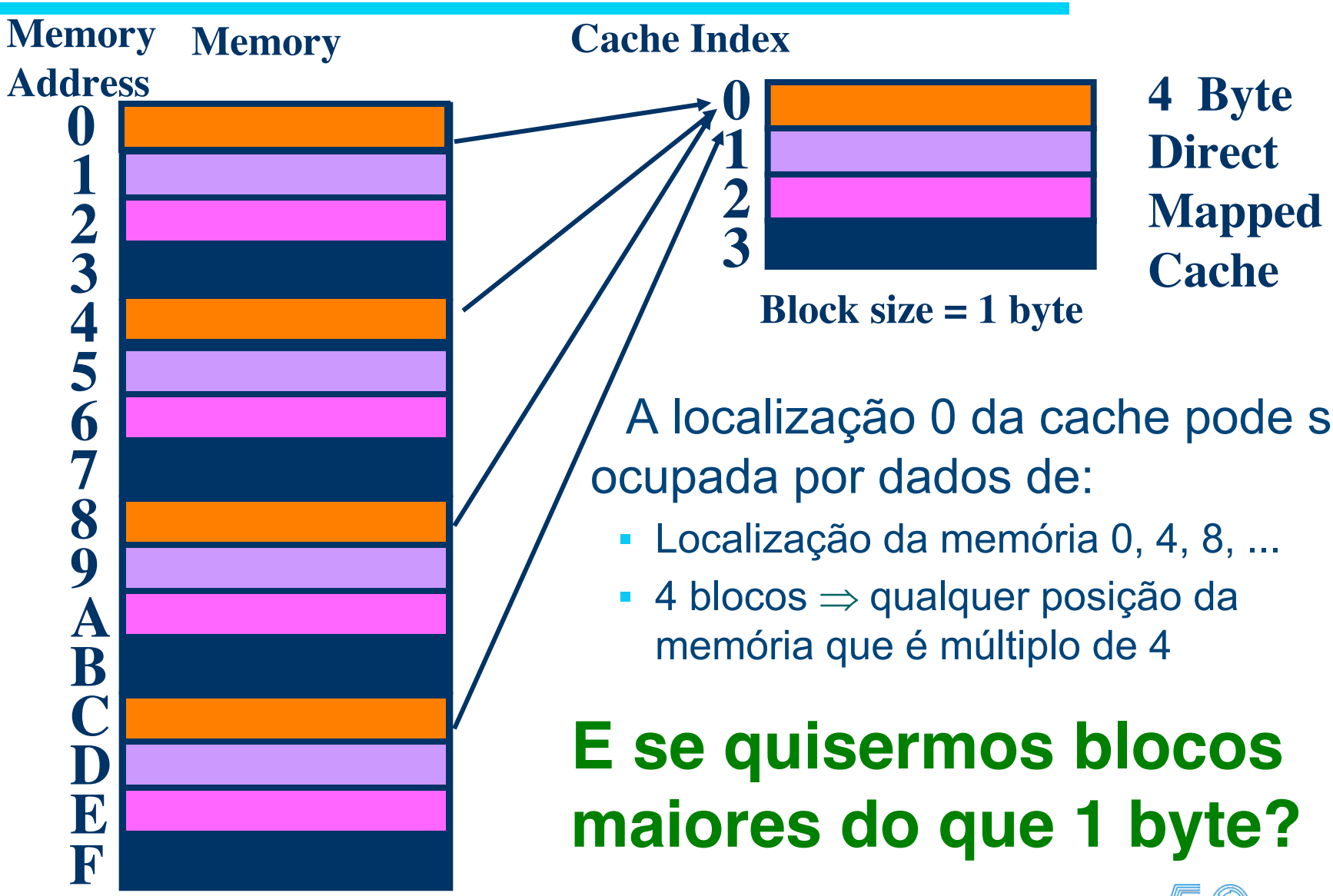
Como podemos localizá-los rapidamente?

Direct-Mapped Cache (1/4)

Numa direct-mapped cache (cache de mapeamento direto), cada endereço de memória está associado com um bloco possível dentro da cache.

- Para se pesquisar se uma dada célula de memória está na cache, só é necessário olhar para uma única localização
- Bloco é a unidade de transferência de informação entre a cache e a memória

Direct-Mapped Cache (2/4)

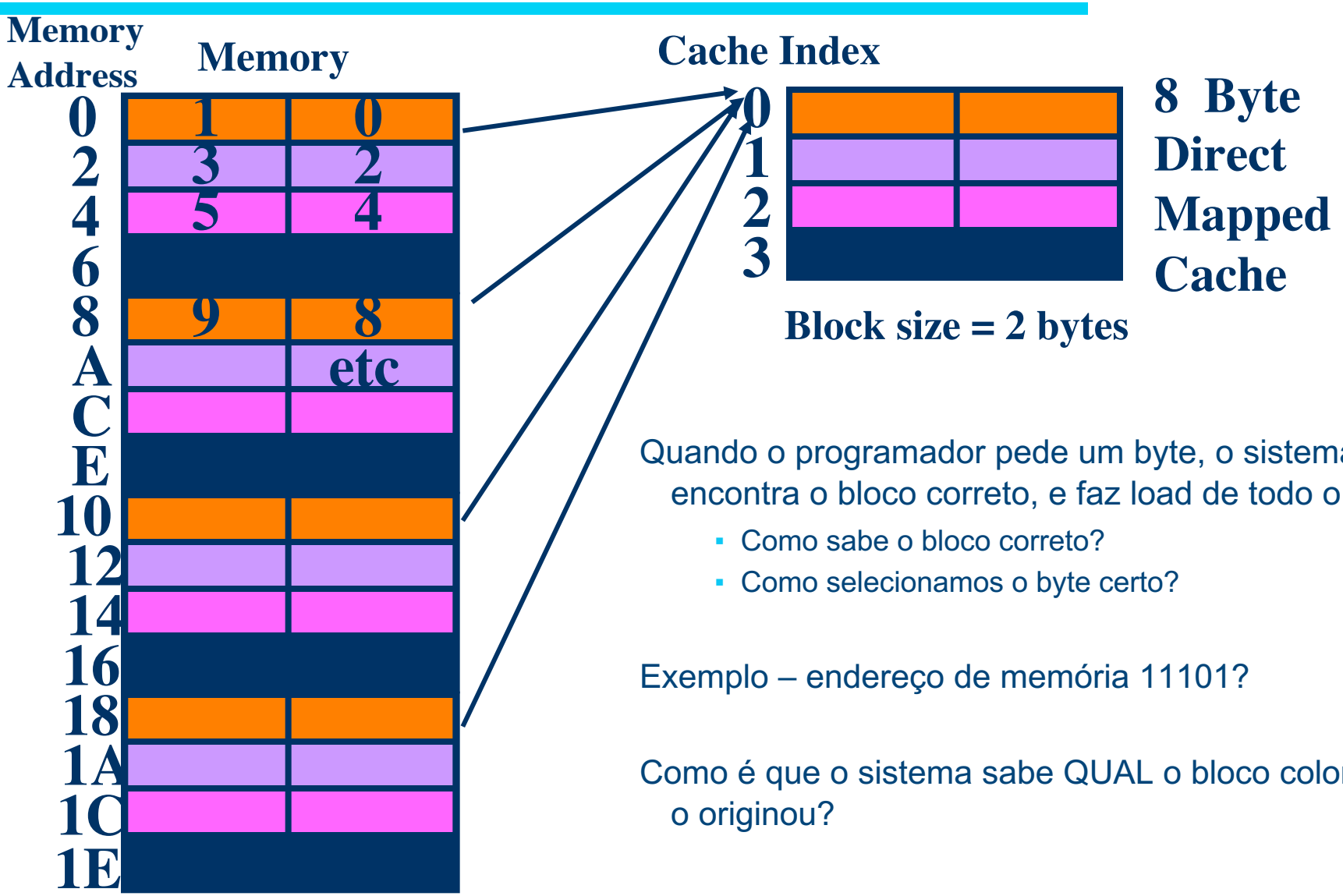


A localização 0 da cache pode ser ocupada por dados de:

- Localização da memória 0, 4, 8, ...
- 4 blocos ⇒ qualquer posição da memória que é múltiplo de 4

E se quisermos blocos maiores do que 1 byte?

Direct-Mapped Cache (3/4)



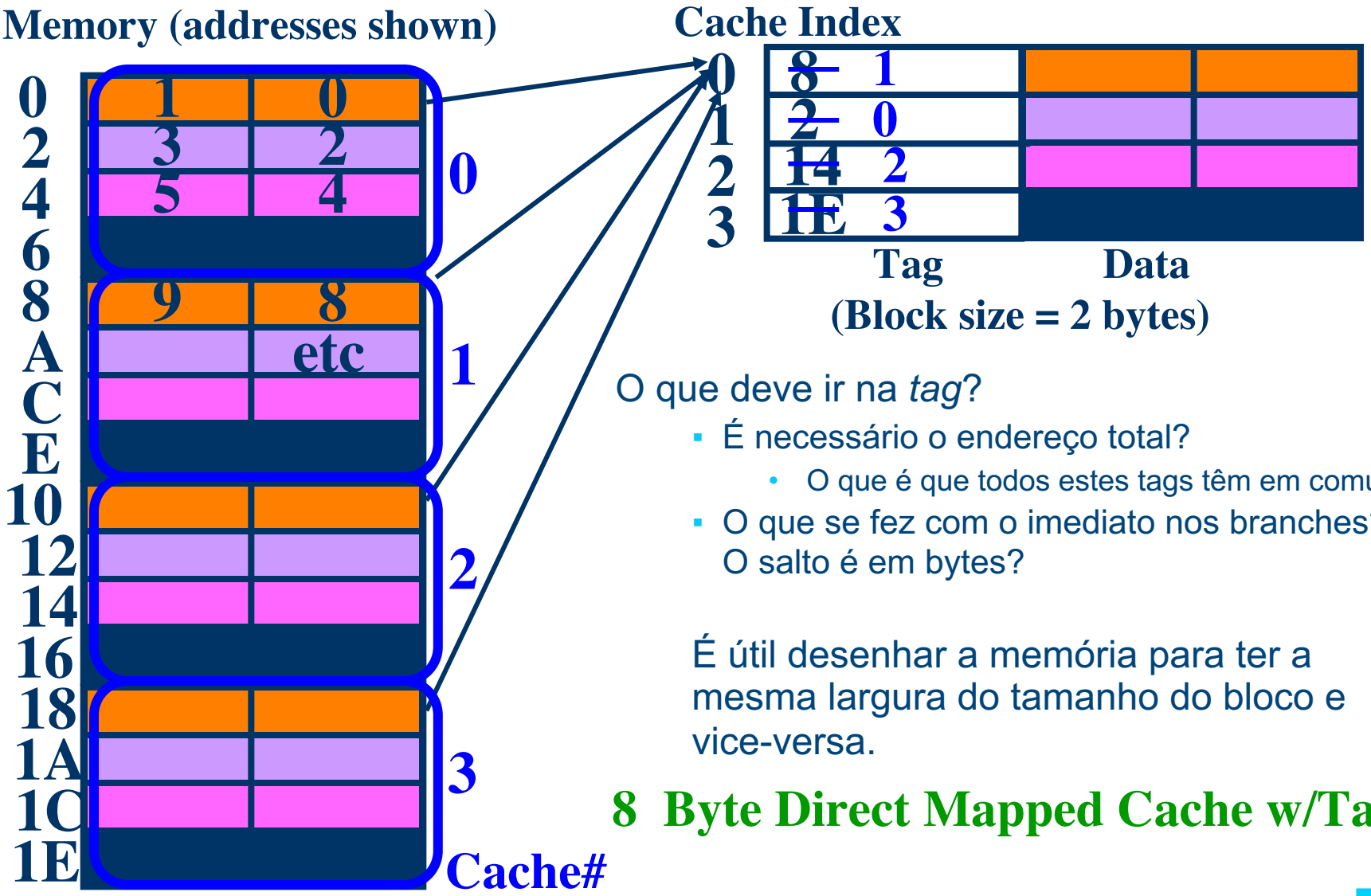
Quando o programador pede um byte, o sistema encontra o bloco correto, e faz load de todo o bloco.

- Como sabe o bloco correto?
- Como selecionamos o byte certo?

Exemplo – endereço de memória 11101?

Como é que o sistema sabe QUAL o bloco colorido que o originou?

Direct-Mapped Cache (4/4)



O que deve ir na *tag*?

- É necessário o endereço total?
  - O que é que todos estes tags têm em comum?
- O que se fez com o imediato nos branches? – O salto é em bytes?

É útil desenhar a memória para ter a mesma largura do tamanho do bloco e vice-versa.

8 Byte Direct Mapped Cache w/Tag!

Problemas com Direct-Mapped cache

Uma vez que há múltiplos endereços de memória a mapear para o mesmo índice da cache, como sabemos qual deles está efetivamente na cache?

E se a dimensão do bloco for superior a 1 byte?

Resposta: dividir os endereços da memória em três campos:

tag
para testar se o endereço do bloco é o correto

index
para selecionar o bloco

offset
byte desejado dentro do bloco

Terminologia das caches Direct-Mapped

Todos os campos são lidos como inteiros sem sinal (unsigned)

Index: especifica o índice na cache (qual a "linha"/bloco da cache em que devemos procurar o byte pretendido)

Offset: depois de encontrado o bloco correto, especifica qual o byte pretendido dentro do bloco

Tag: os bits sobrantes depois do index e offset serem determinados; são usados para distinguir entre todos os endereços de memória que mapeiam para a mesma localização

Exemplo de cache Direct-Mapped (1/3)

Suponha que há uma cache direct-mapped com 16 KB de dados com blocos de 4 words.

Determine a dimensão dos campos tag, index e offset se estivermos a usar uma arquitetura de 32-bits

Offset

- necessita de especificar corretamente o byte dentro do bloco
- o bloco contém 4 words

  = 16 bytes

  = 2^4 bytes
- necessita de 4 bits para especificar o byte corretamente

Exemplo de cache Direct-Mapped (2/3)

Index: (~índice dentro do “array of blocos”)

- necessita de especificar o bloco correto na cache
- a cache contém 16 KB = 2^{14} bytes
- os blocos contêm 2^4 bytes (4 words)
- # blocks/cache

$$= \frac{\text{bytes/cache}}{\text{bytes/block}}$$

$$= \frac{2^{14}\ \text{bytes/cache}}{2^4\ \text{bytes/block}}$$

$$= 2^{10}\ \text{blocks/cache}$$

- necessita de 10 bits para especificar esta totalidade de blocos

Exemplo de cache Direct-Mapped (3/3)

Tag: usa os restantes bits como tag

- dimensão da tag = largura do endereço – offset - index
  = 32 - 4 - 10 bits
  = 18 bits
- então as tag são os 18 bits mais à esquerda do endereço de memória

Por que não usar todos os 32 bits do endereço como tag?

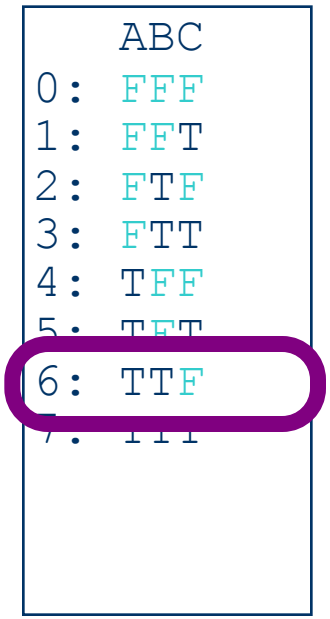
- Todos os bytes dentro do bloco necessitam de ter o mesmo endereço (4 bits)
- O index deve ser o mesmo para todos os endereços dentro do mesmo bloco, portanto é redundante na verificação da tag. Então pode ser deixado de fora para poupar memória (10 bits neste caso)

QUIZ

A. The number of bits in the tag only depends of the cache size. It does not depend of the block size.

B. If you know your computer's cache size, you can often make your code run faster.

C. Memory hierarchies take advantage of spatial locality by keeping the most recent data items closer to the processor.



Conclusão

Idealmente toda a memória de disco deveria funcionar à mesma velocidade do processador: infelizmente isto não é possível

Por isso criamos uma hierarquia de memória:

- cada nível sucessivamente mais baixo contém os dados "mais usados" do nível imediatamente acima
- explora as localidades temporal e espacial
- o princípio é "faz o caso comum mais rápido, preocupa-te menos com as exceções" – princípio base da construção do MIPS

Localidade da referência (endereço) é uma grande ideia!

Para saber mais ...

P&H - Capítulo 7.1 e 7.2

Slides sobre “Code Optimization”

Introdução à Arquitetura de Computadores
- Hierarquia de Memória II -

Sistemas de Microprocessadores 2021/22

Terminologia de cache

Quando há uma leitura à memória, podem acontecer três situações:

1. cache hit: o bloco da cache é válido e contém o endereço apropriado, por isso lê a word pretendida

2. cache miss: não há nada na cache no bloco apropriado, por isso faz "fetch" da memória

3. cache miss, block replacement: os dados na cache no bloco apropriado estão errados, por isso descarta-o e vai à memória buscar os dados pretendidos (a cache é sempre uma cópia)

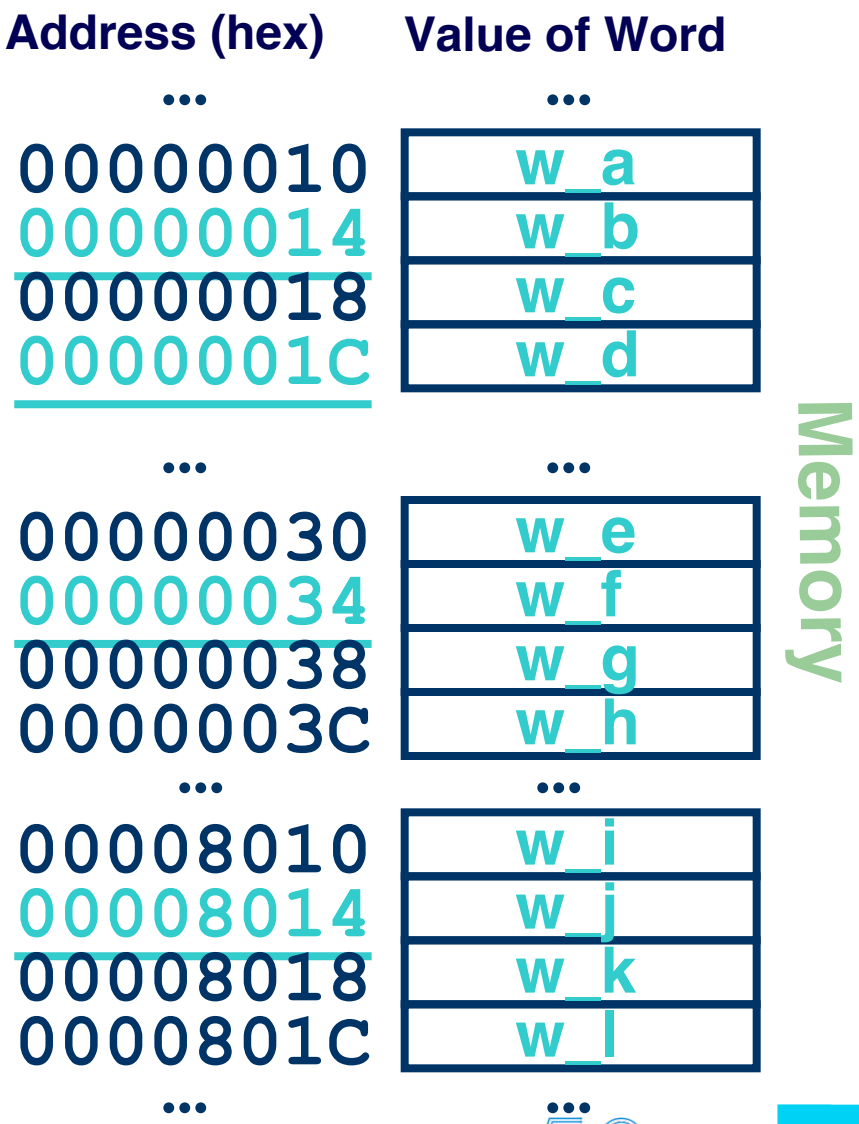
Aceder a dados numa cache direct mapped

Ex.: 16KB de dados, direct-mapped, blocos de 4 words

Lê 4 endereços

1. 0x00000014
2. 0x0000001C
3. 0x00000034
4. 0x00008014

Valores da memória à direita:



Aceder a dados numa cache direct mapped

4 endereços:

- `0x00000014, 0x0000001C, 0x00000034, 0x00008014`

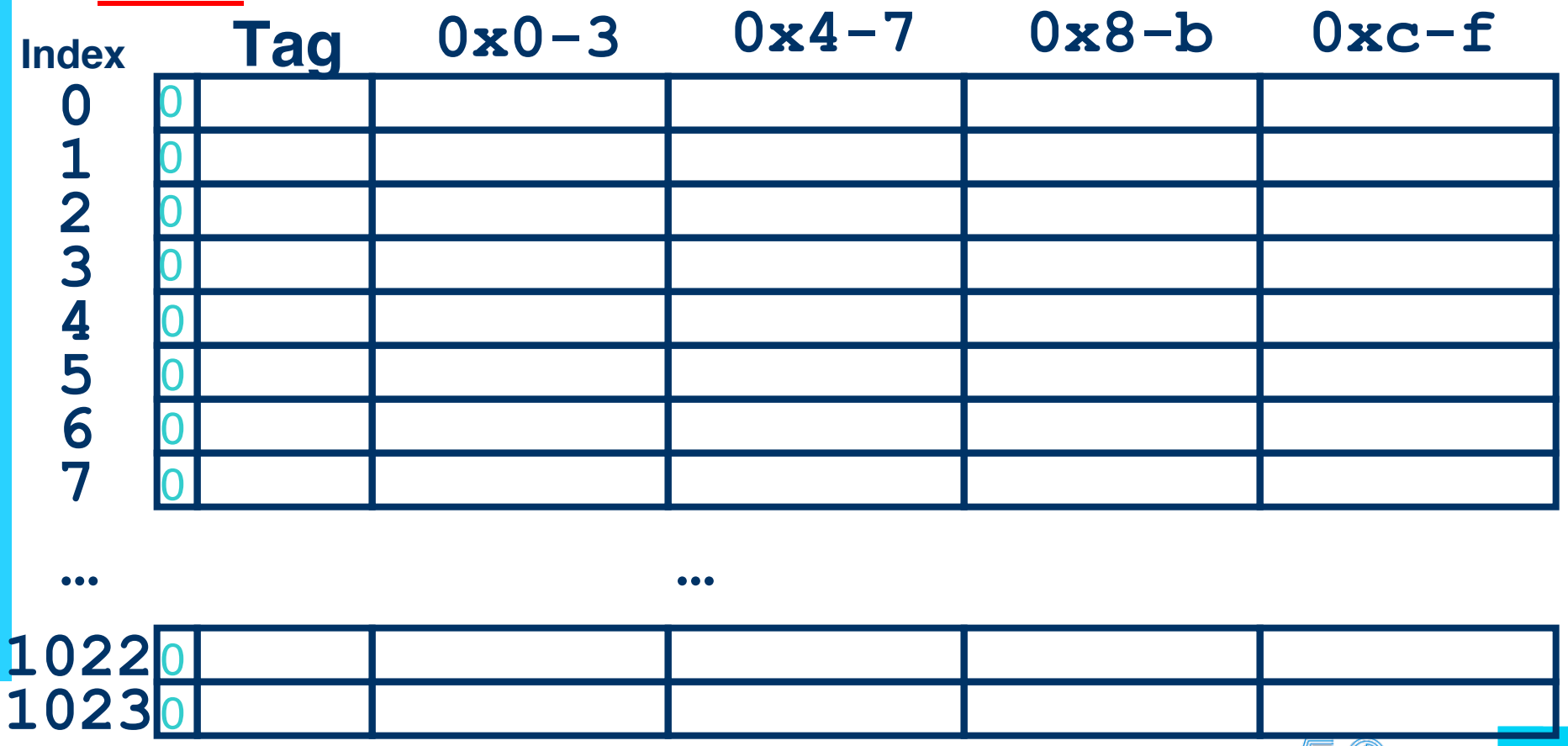
4 endereços divididos (por conveniência) nos campos Tag, Index e Offset

```
00000000000000000 0000000001 0100
00000000000000000 0000000001 1100
00000000000000000 0000000011 0100
00000000000000010 0000000001 0100
```

Tag Index Offset

16 KB Direct Mapped Cache, 16B blocks

Valid bit: determines whether anything is stored in that row
(when computer initially turned on, all entries invalid)



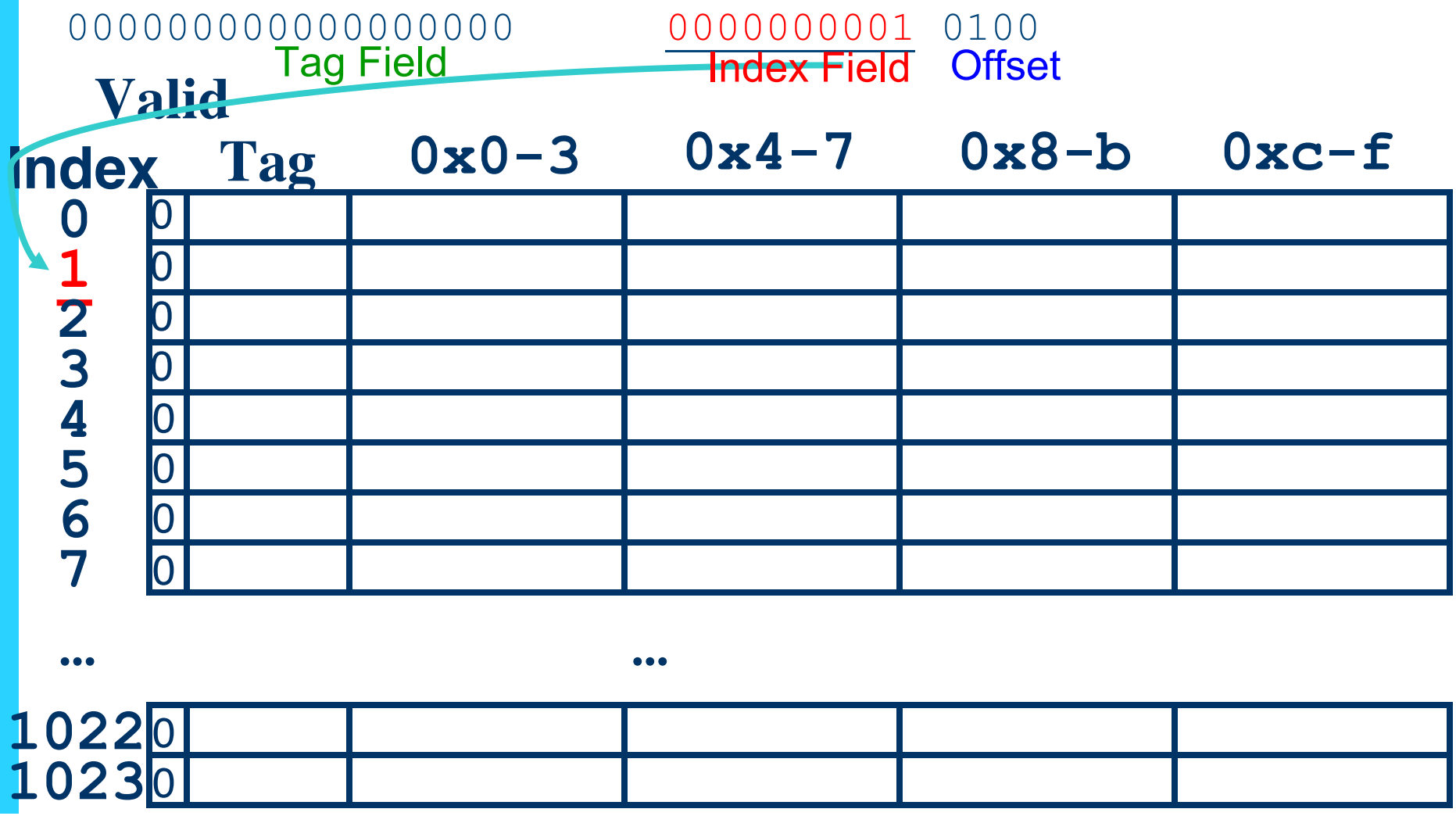
1. Read 0x00000014

00000000000000000000 0000000001 0100

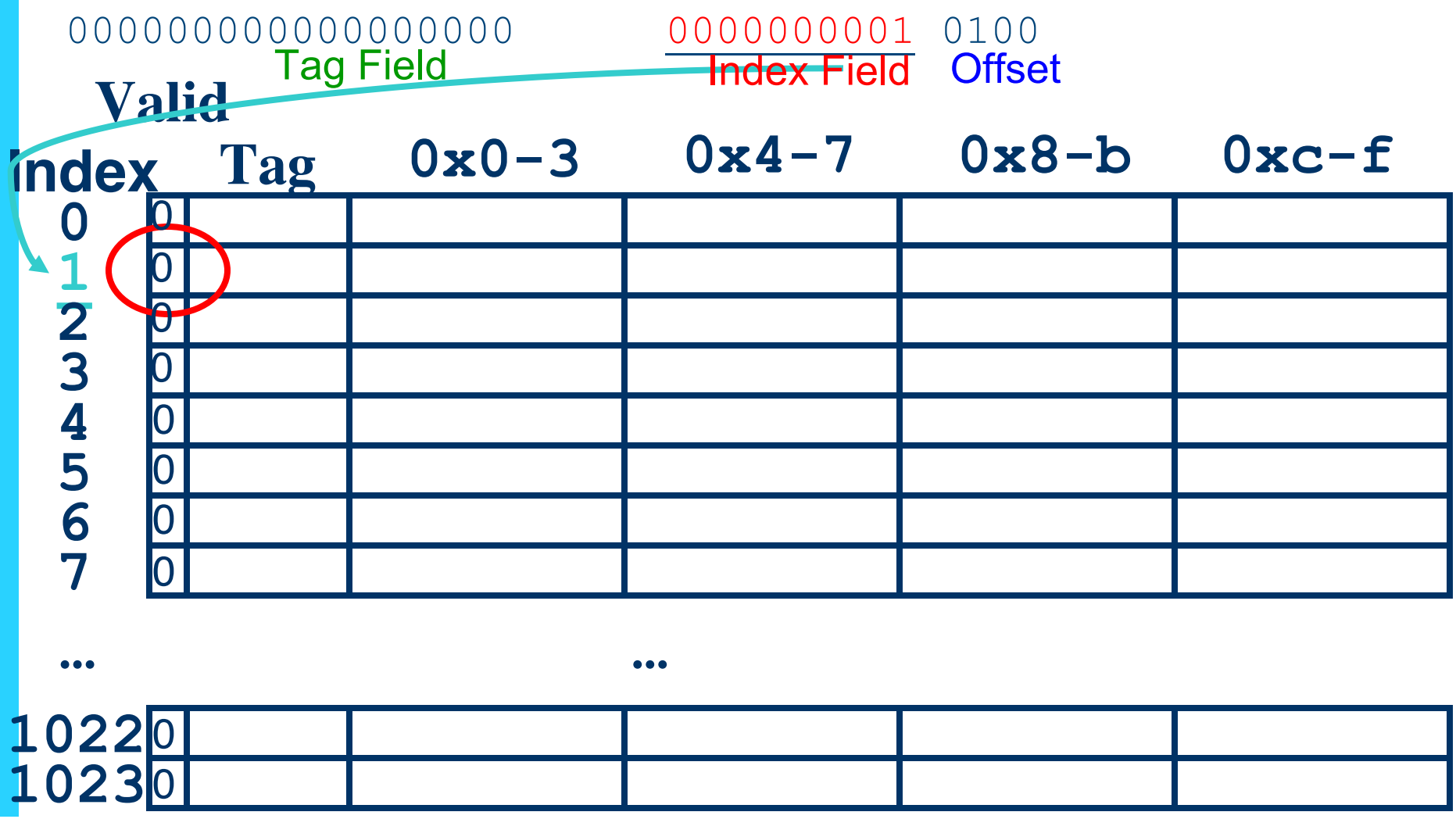
Tag Field Index Field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	0					
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

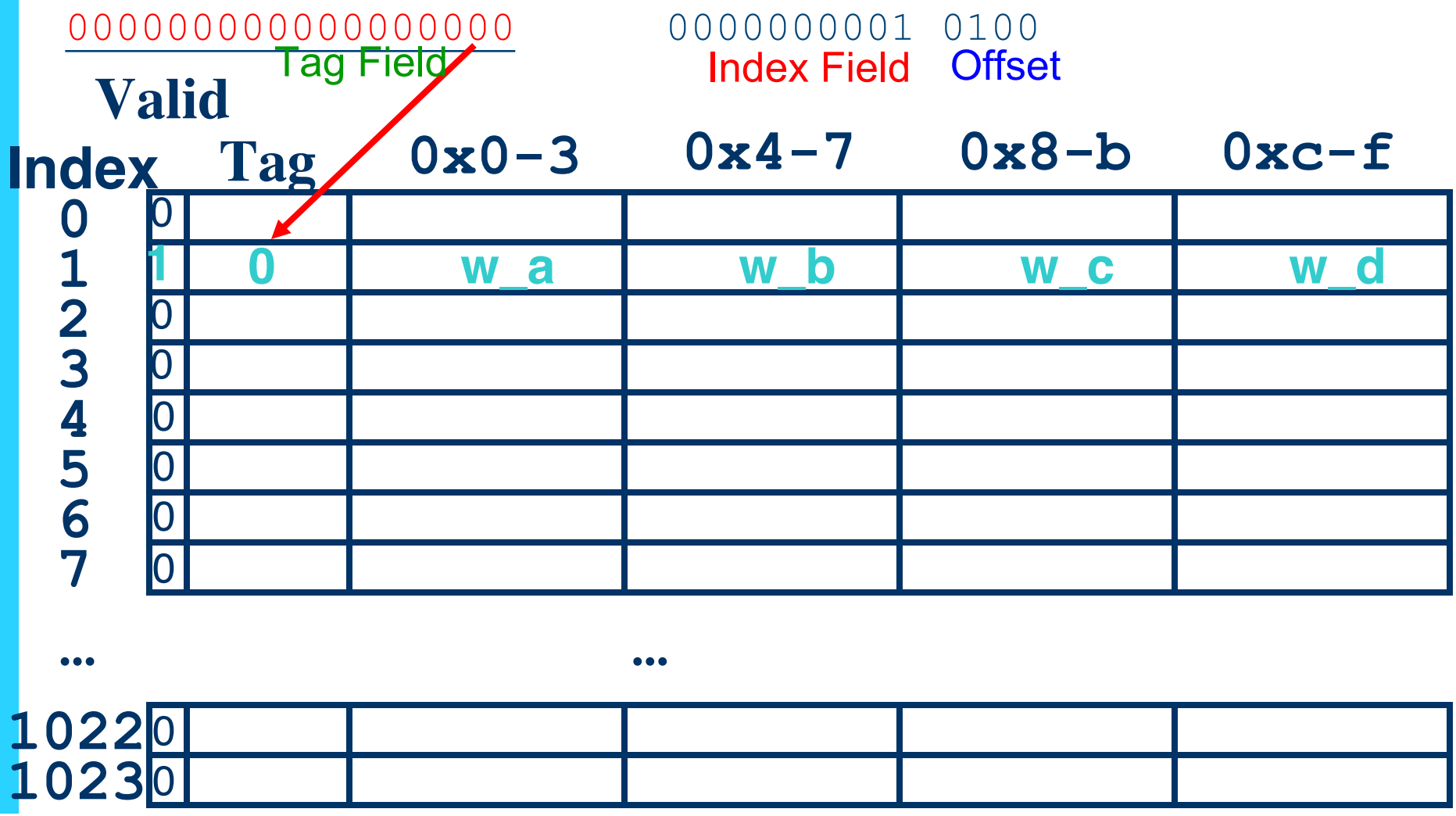
So we read block 1 (0000000001)



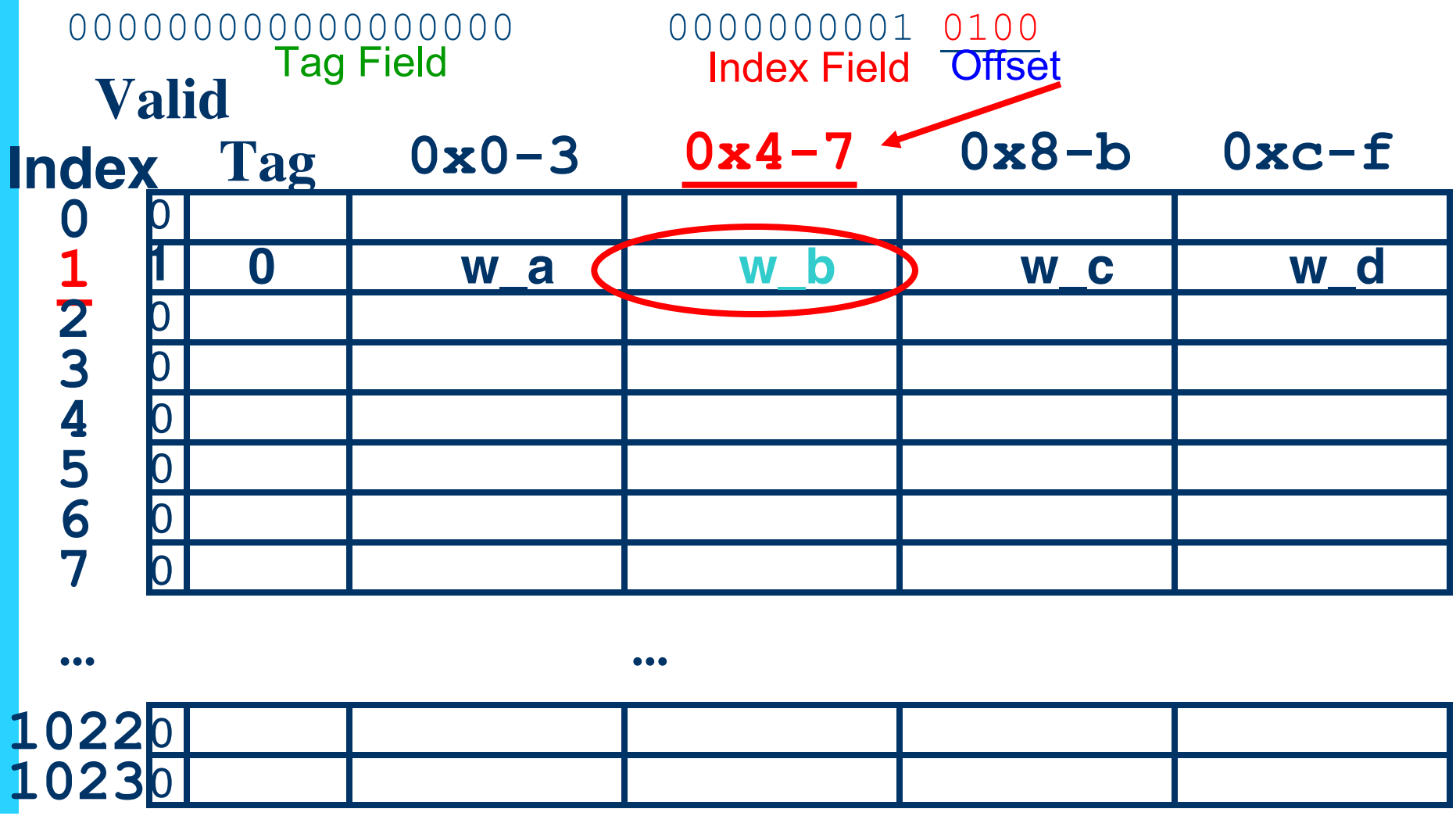
No valid data - Cache Miss



So load that data into cache, setting tag, valid



Read from cache at offset, return word b



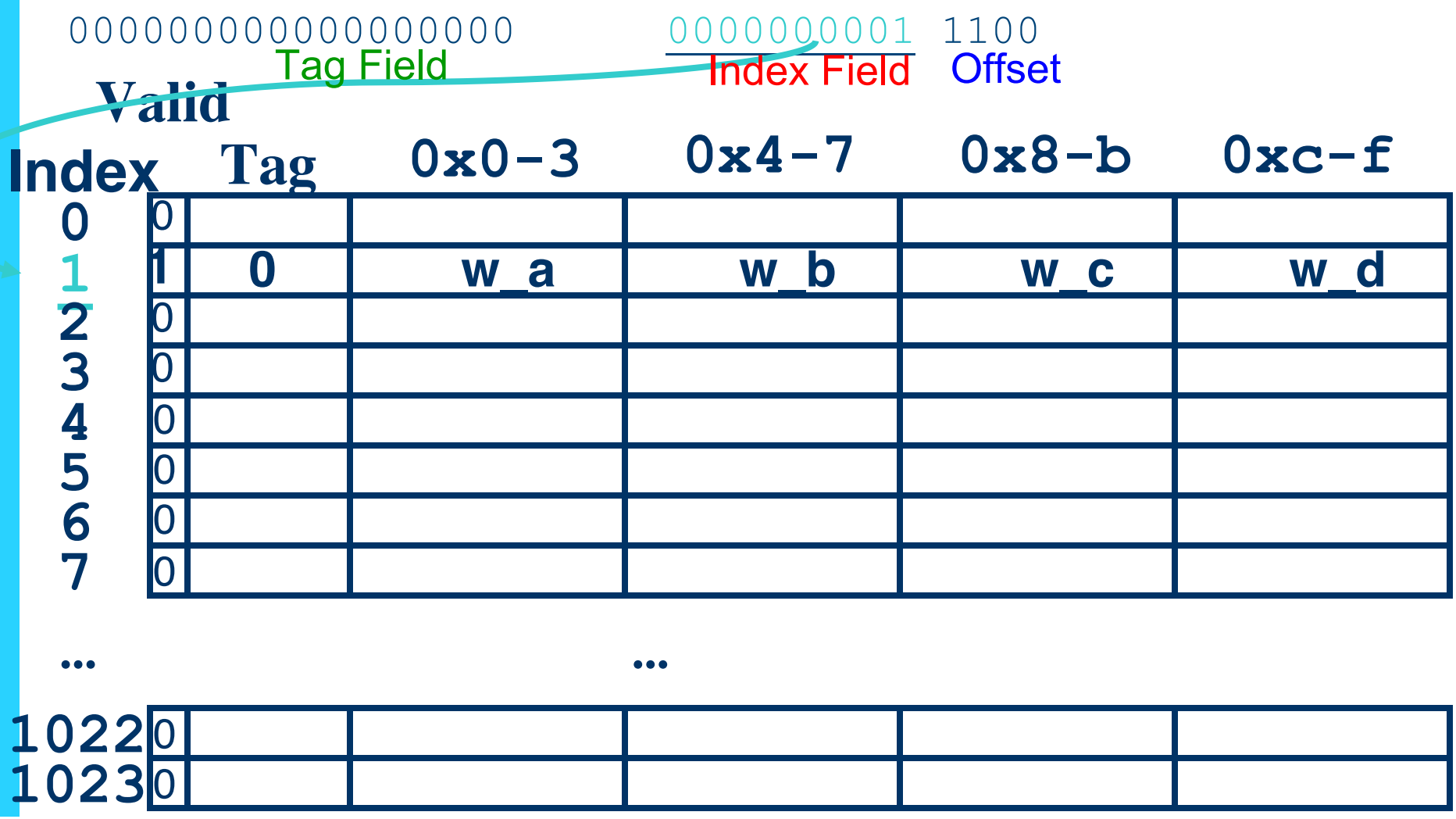
2. Read 0x0000001C = 0…00 0..001 1100

00000000000000000000 0000000001 1100

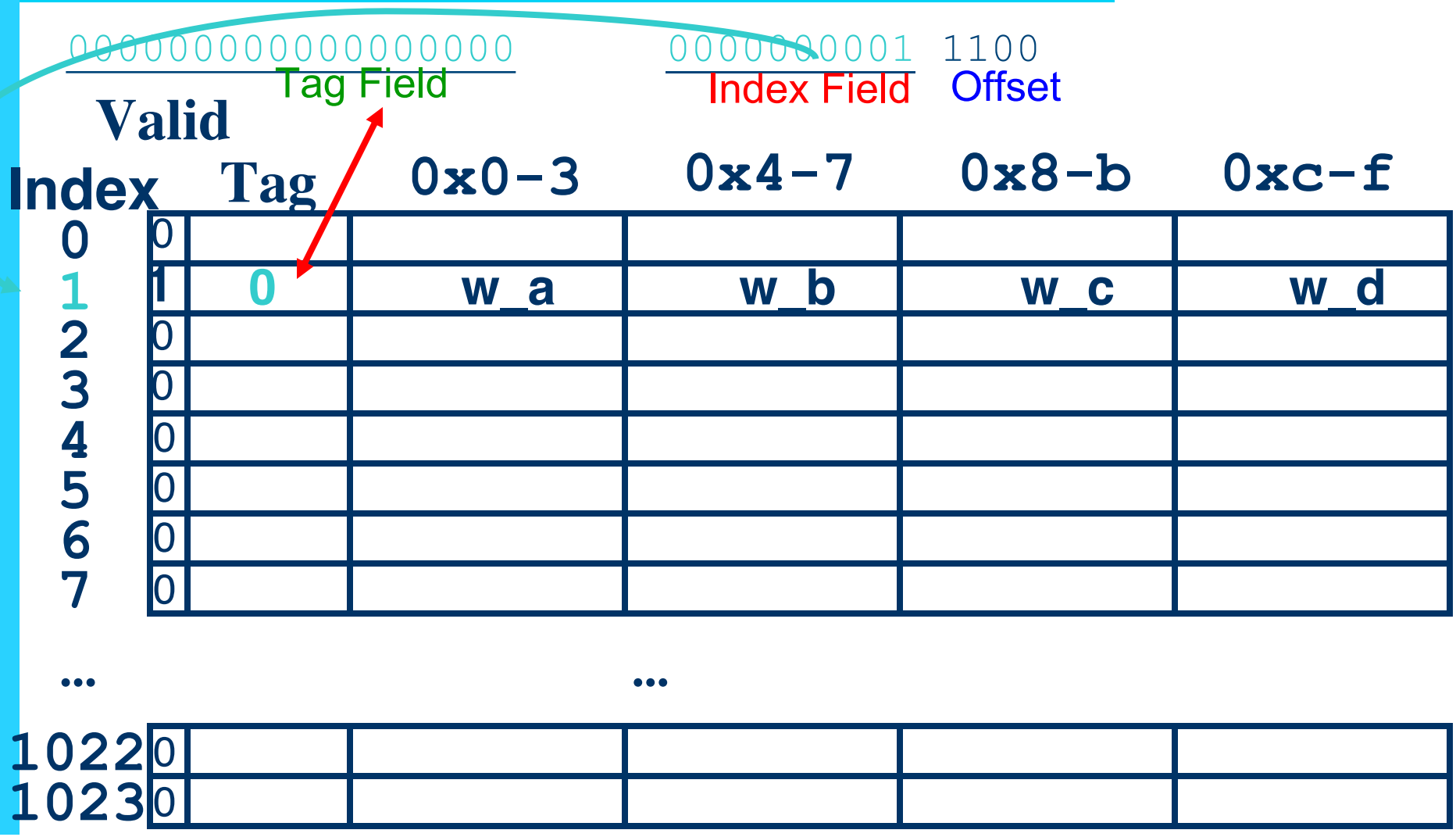
Tag Field Index Field Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	w_a	w_b	w_c	w_d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…			…			
1022	0					
1023	0					

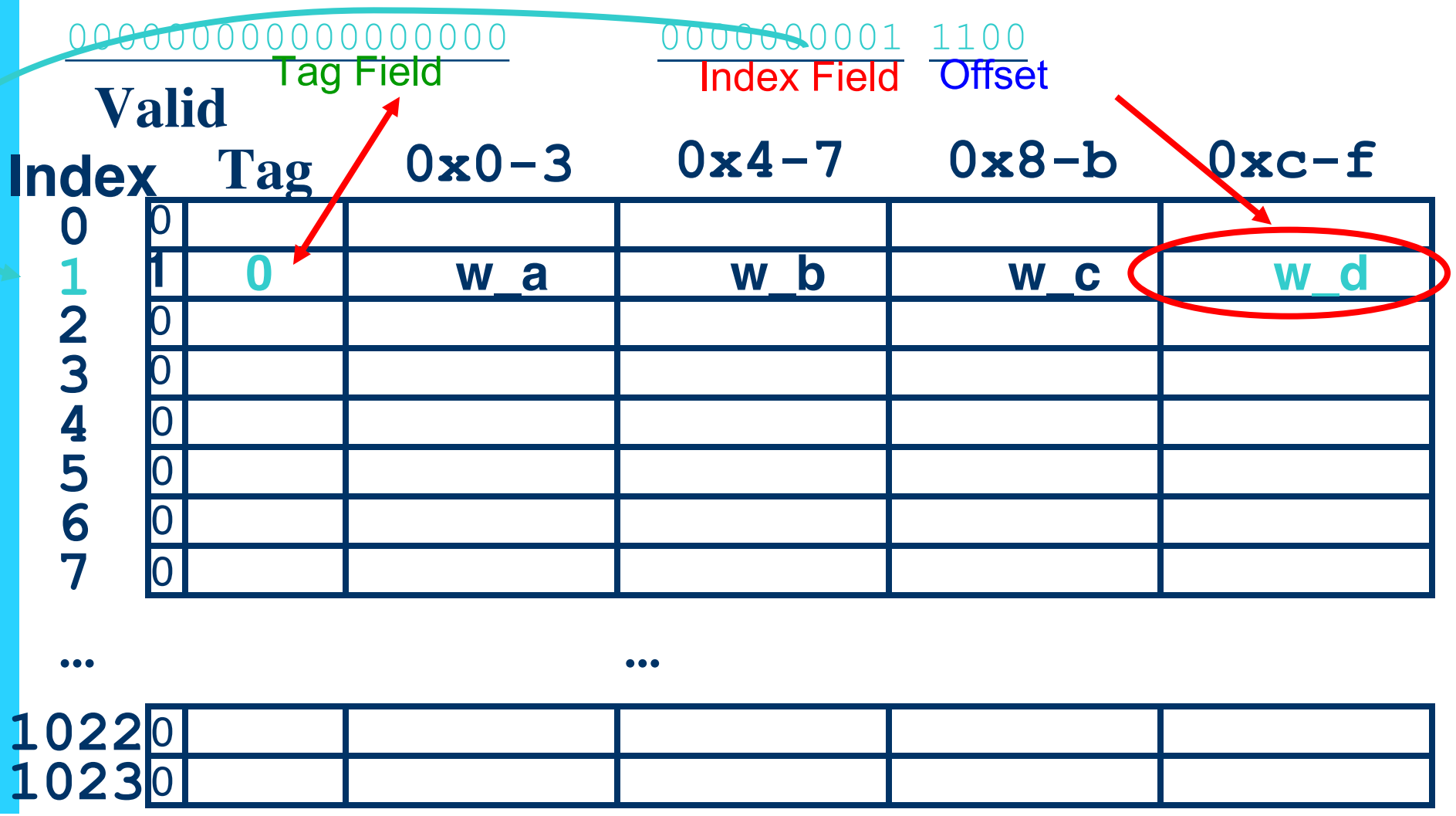
Index is Valid



Index valid, Tag Matches - Cache Hit



Index Valid, Tag Matches, return d



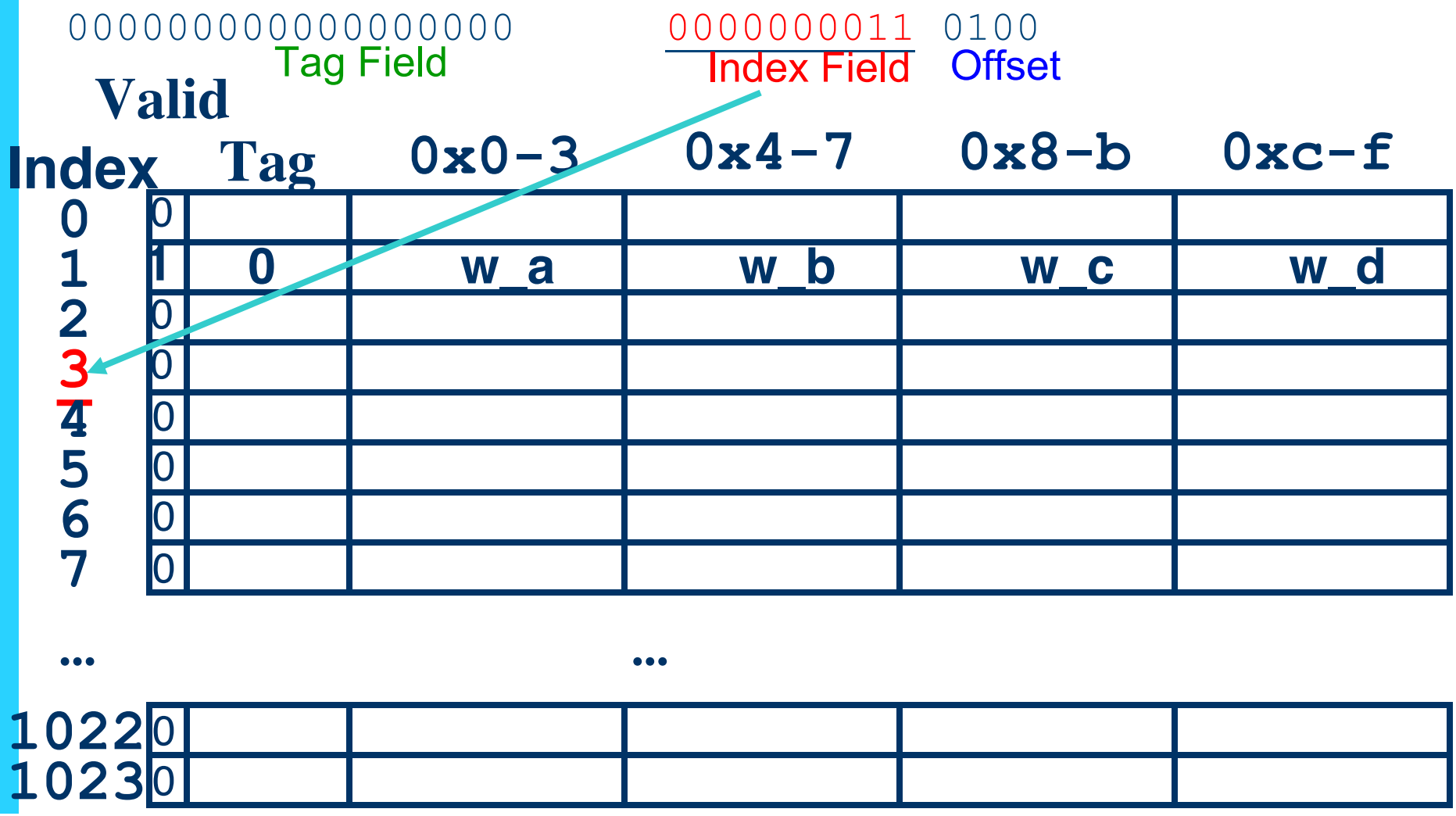
3. Read 0x00000034 = 0…00 0..011 0100

0000000000000000000 0000000011 0100

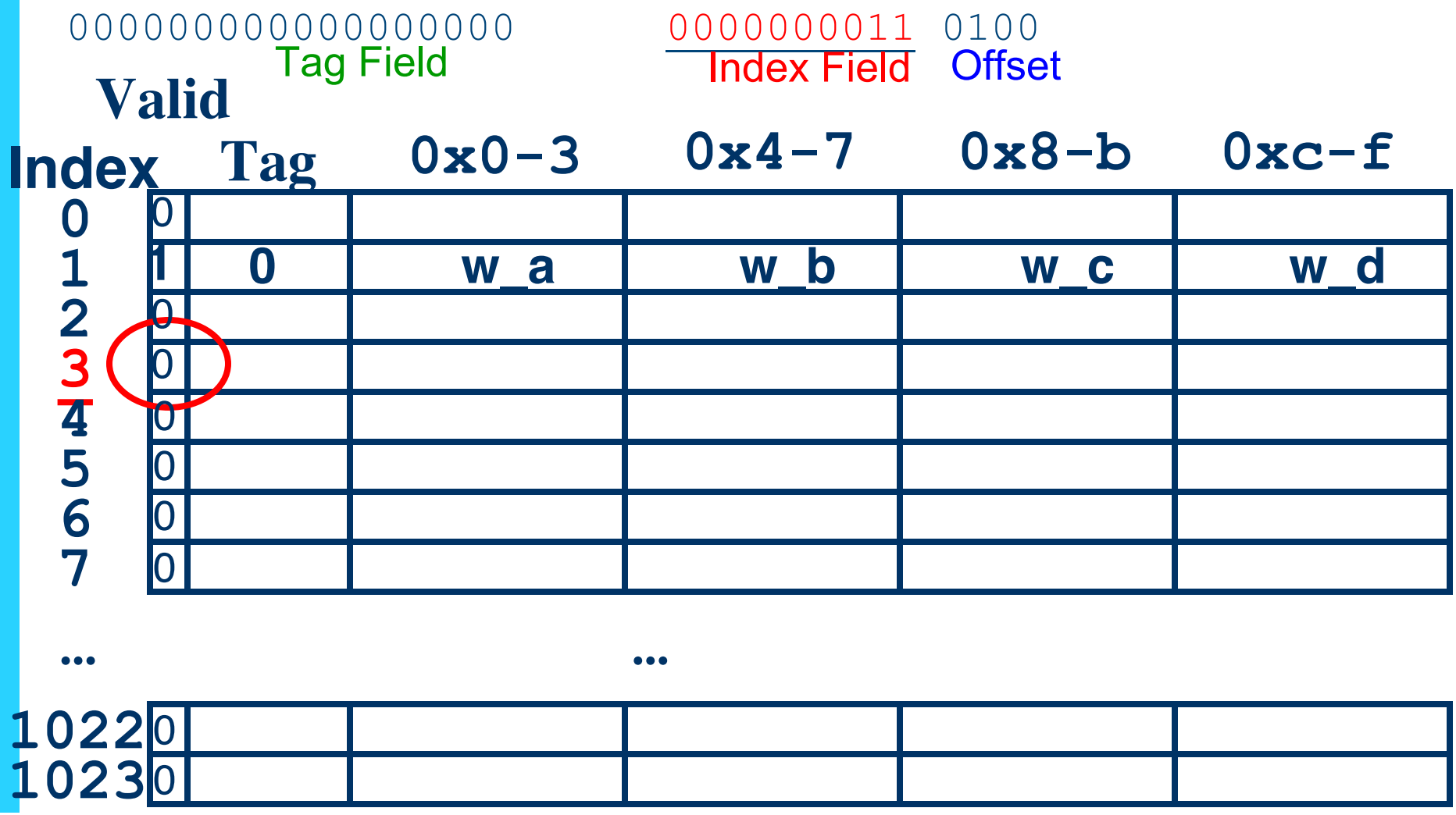
Tag Field | Index Field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	w_a	w_b	w_c	w_d
2	0					
3	0					
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

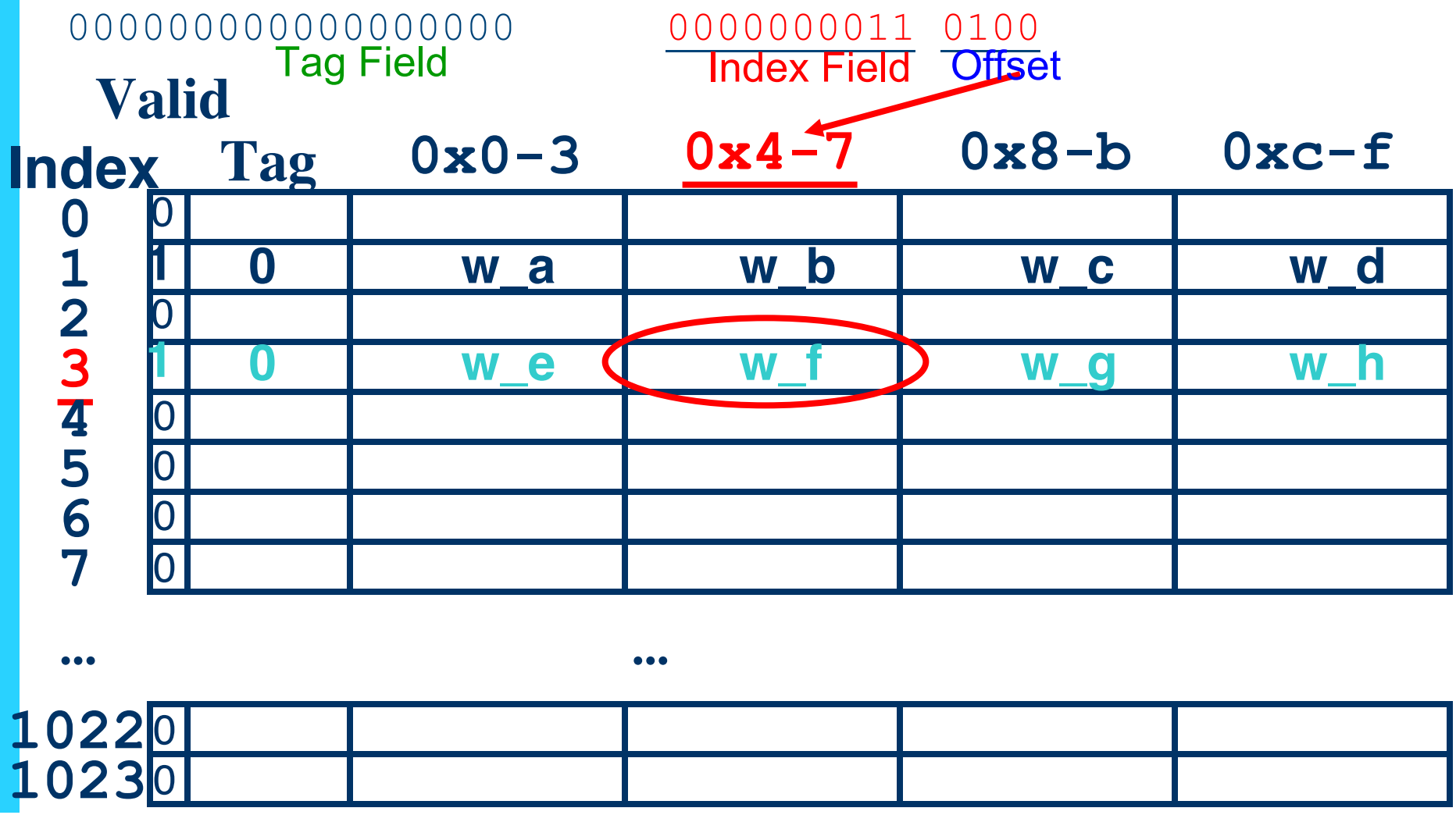
So read block 3



No valid data - Cache Miss



Load that cache block, return word f



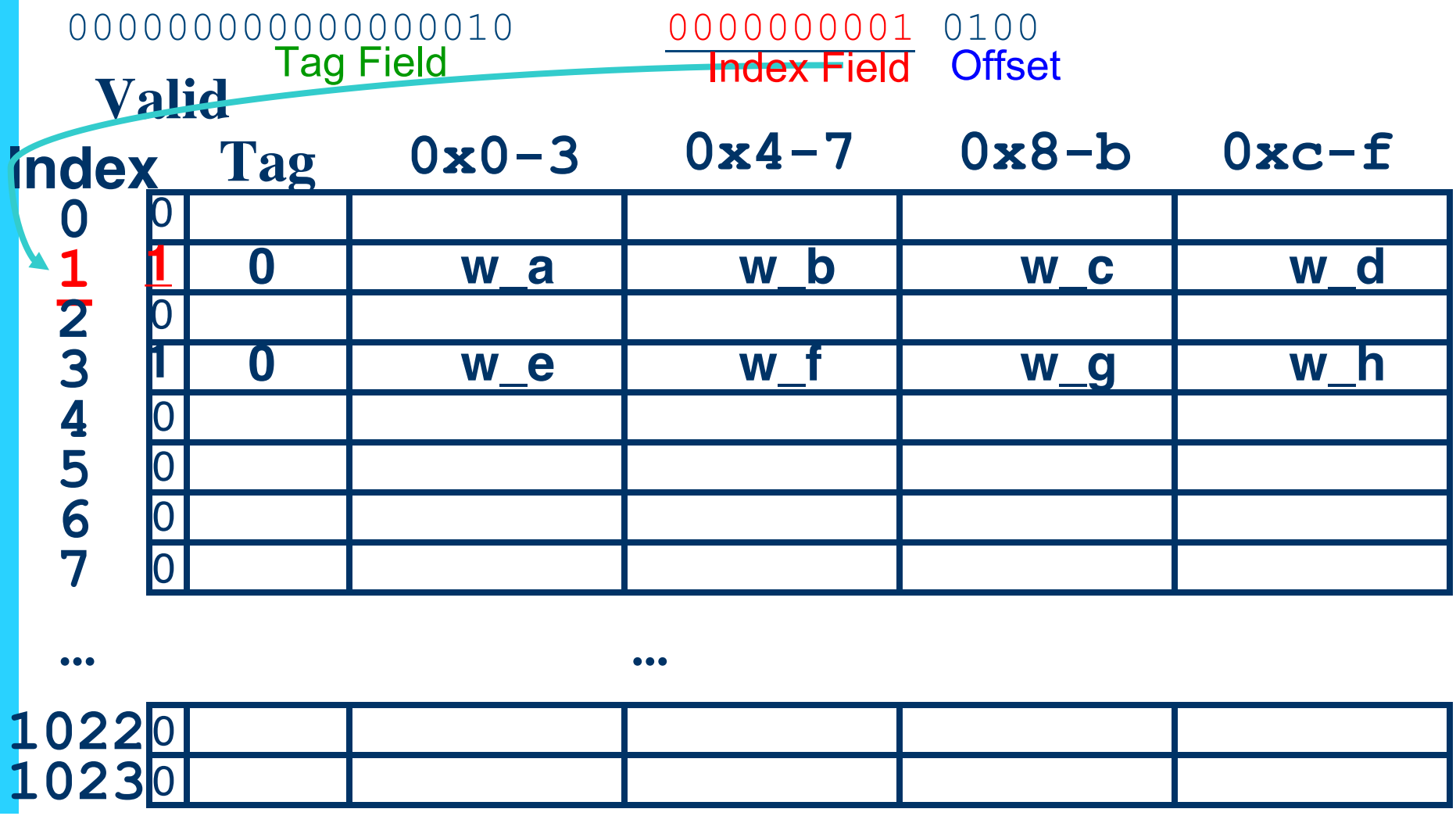
4. Read 0x00008014 = 0…10 0..001 0100

0000000000000010 0000000001 0100

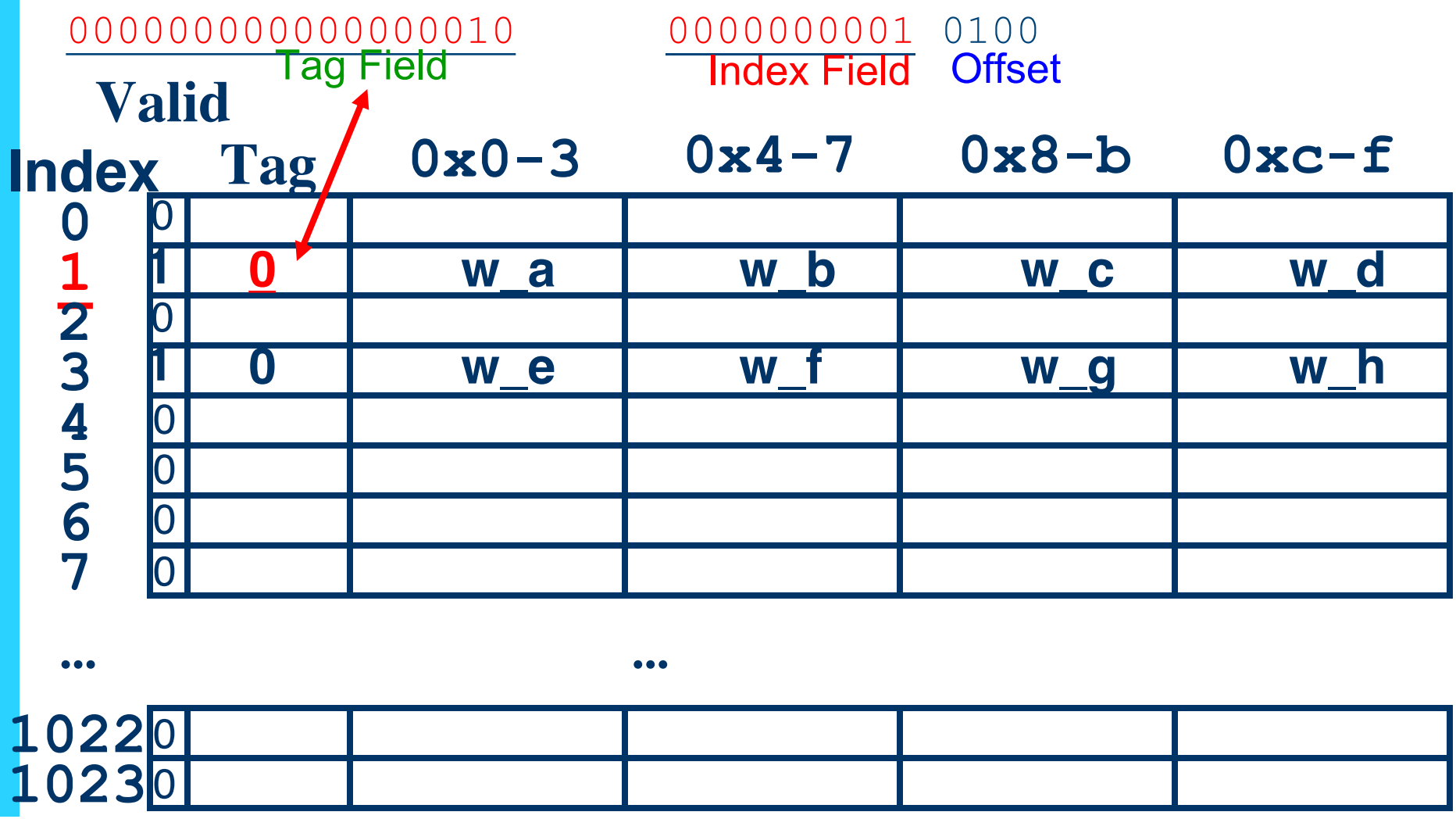
Tag Field | Index Field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	0	w_a	w_b	w_c	w_d
2	0					
3	1	0	w_e	w_f	w_g	w_h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

So read Cache Block 1, Data is Valid



Tag does not match (0 != 2) - Cache Miss w/ Rep.



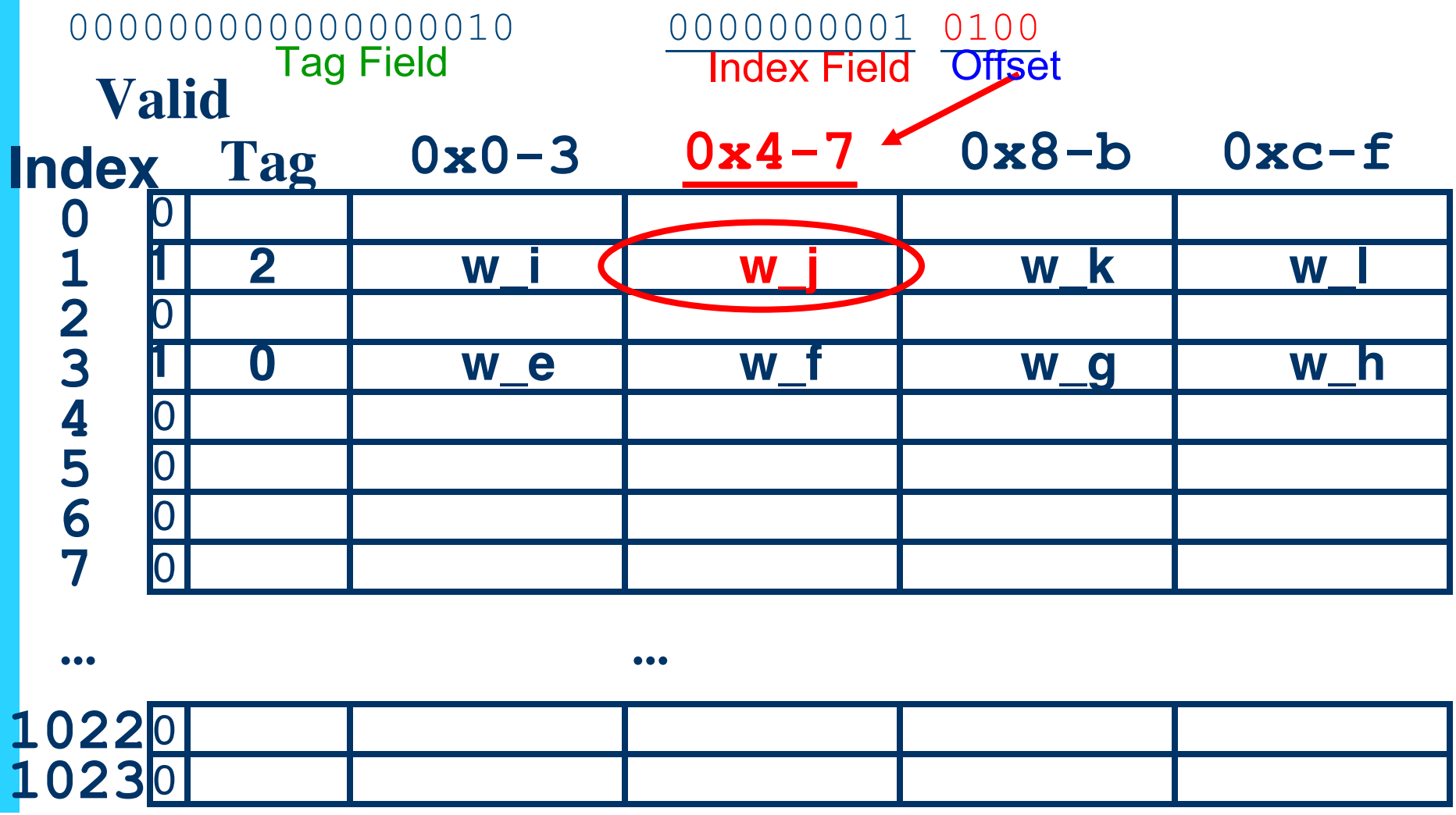
Replace block 1 with new data & tag

0000000000000010 0000000001 0100

Tag Field | Index Field | Offset

Index	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	w_i	w_j	w_k	w_l
2	0					
3	1	0	w_e	w_f	w_g	w_h
4	0					
5	0					
6	0					
7	0					
…				…		
1022	0					
1023	0					

And return word j



Do an example yourself. What happens?

Choose from: Cache: Hit, Miss, Miss w. replace
Values returned: a ,b, c, d, e, ..., k, l

Read address 0x00000030 ? `0000000000000000 0000000011 0000`

- Cache Hit; Returns e

Read address 0x0000001c ? `0000000000000000 0000000001 1100`

- Cache Miss with Block Replacement

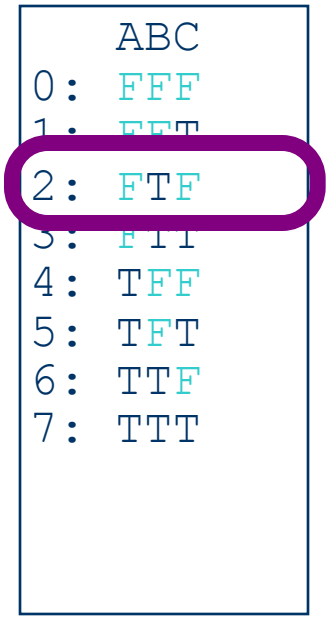
	Valid	Tag	0x0-3	0x4-7	0x8-b	0xc-f
0	0					
1	1	2	w_i	w_j	w_k	w_l
2	0					
3	1	0	w_e	w_f	w_g	w_h
4	0					
5	0					
6	0					
7	0					
…				…		

QUIZ

A - All caches take advantage of spatial locality.

B - All caches take advantage of temporal locality.

C - On a read, the return value will depend on what is in the cache.



What to do on a write hit?

Write-through

- update the word in cache block and corresponding word in memory

Write-back

- update word in cache block
- allow memory word to be “stale”

⇒ add ‘dirty’ bit to each block indicating that memory needs to be updated when block is replaced

⇒ OS flushes cache before I/O…

Performance trade-offs?

Block Size Tradeoff (1/3)

Benefits of Larger Block Size

- Spatial Locality: if we access a given word, we're likely to access other nearby words soon
- Very applicable with Stored-Program Concept: if we execute a given instruction, it's likely that we'll execute the next few as well
- Works nicely in sequential array accesses too

Block Size Tradeoff (2/3)

Drawbacks of Larger Block Size

- Larger block size means
  larger miss penalty
  - on a miss, takes longer time to load a new block from next level
- If block size is too big relative to cache size, then there are too few blocks
  - Result: miss rate goes up

In general, minimize Average Memory Access Time (AMAT)

= Hit Time + Miss Penalty x Miss Rate

Block Size Tradeoff (3/3)

Hit Time = time to find and retrieve data from current level cache

Miss Penalty = average time to retrieve data on a current level miss (includes the possibility of misses on successive levels of memory hierarchy)

Hit Rate = % of requests that are found in current level cache

Miss Rate = 1 - Hit Rate

Extreme Example: One Big Block

Cache Size = 4 bytes Block Size = 4 bytes

- Only ONE entry (row) in the cache!

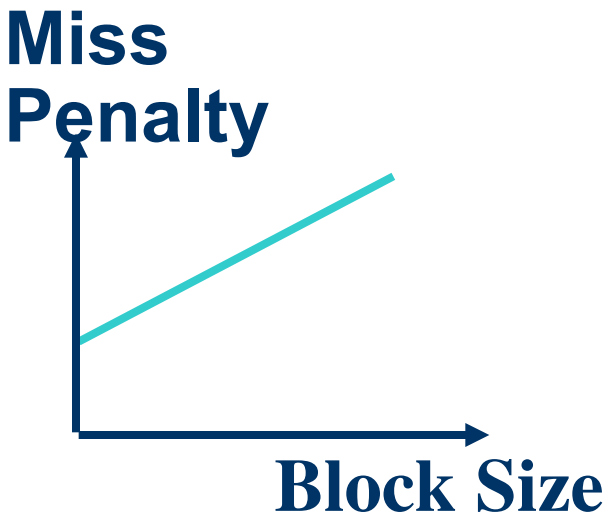
If item accessed, likely accessed again soon

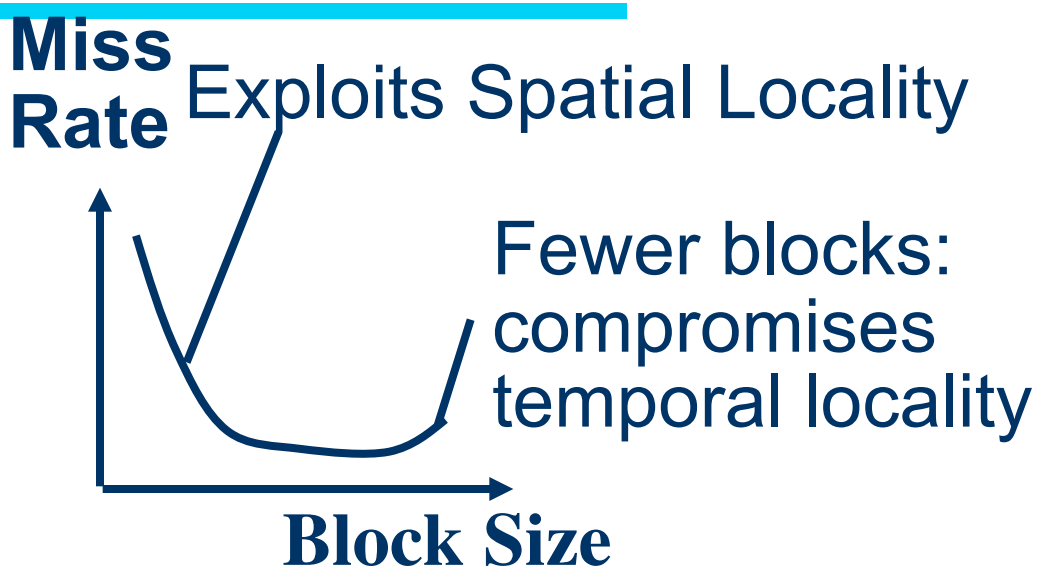
- But unlikely will be accessed again immediately!

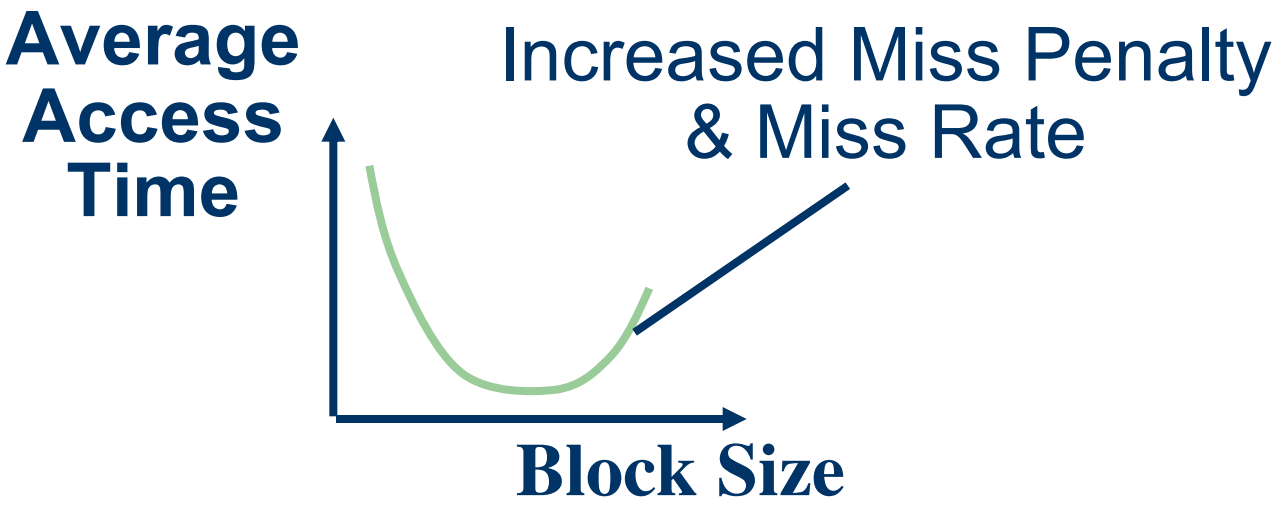
The next access will likely to be a miss again

- Continually loading data into the cache but discard data (force out) before use it again
- Nightmare for cache designer: Ping Pong Effect

Block Size Tradeoff Conclusions







Types of Cache Misses (1/2)

"Three Cs" Model of Misses

1st C: Compulsory Misses

- occur when a program is first started
- cache does not contain any of that program's data yet, so misses are bound to occur
- can't be avoided easily, so won't focus on these in this course

2nd C: Capacity Misses

- miss that occurs because the cache has a limited size
- miss that would not occur if we increase the size of the cache
- sketchy definition, so just get the general idea

Types of Cache Misses (2/2)

3rd C: Conflict Misses

- miss that occurs because two distinct memory addresses map to the same cache location
- two blocks (which happen to map to the same location) can keep overwriting each other
- it is a waste in case there are other free blocks corresponding to mem addresses that are not being accessed
- big problem in direct-mapped caches!
- how do we lessen the effect of these?

Dealing with Conflict Misses

- Solution 1: Make the cache size bigger
  - Fails at some point
- Solution 2: Can multiple distinct blocks fit in the same cache Index?

Fully Associative Cache (1/3)

Memory address fields:

- Tag: same as before
- Offset: same as before
- Index: non-existant

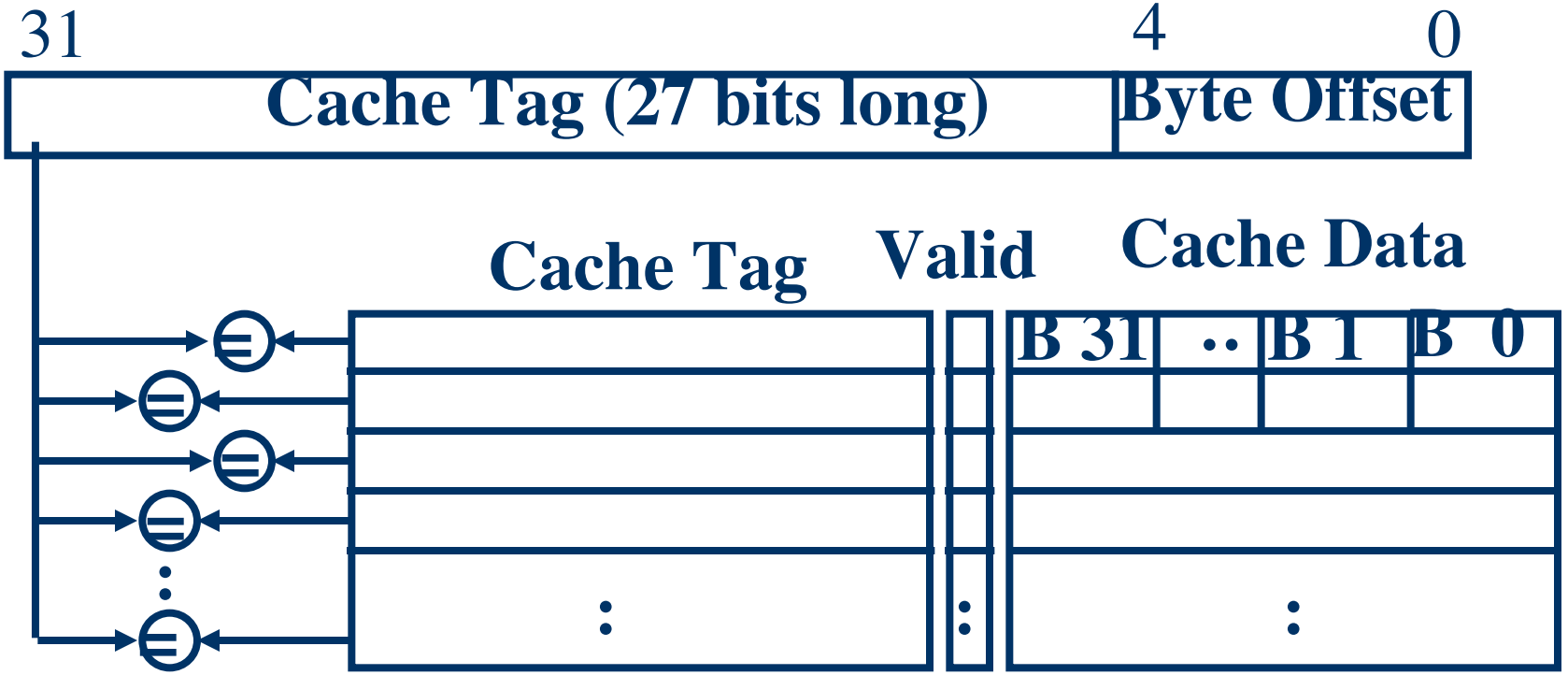
What does this mean?

- no “rows”: any block can go anywhere in the cache
- must compare with all tags in entire cache to see if data is there

Fully Associative Cache (2/3)

Fully Associative Cache (e.g., 32 B block)

- compare tags in parallel



Fully Associative Cache (3/3)

Benefit of Fully Associative Cache

- No Conflict Misses (since data can go anywhere)
- The primary type of miss is Capacity Miss

Drawbacks of Fully Associative Cache

- Need hardware comparator for every single entry: if we have a 64KB of data in cache with 4B entries, we need 16K comparators: infeasible

N-Way Set Associative Cache (1/3)

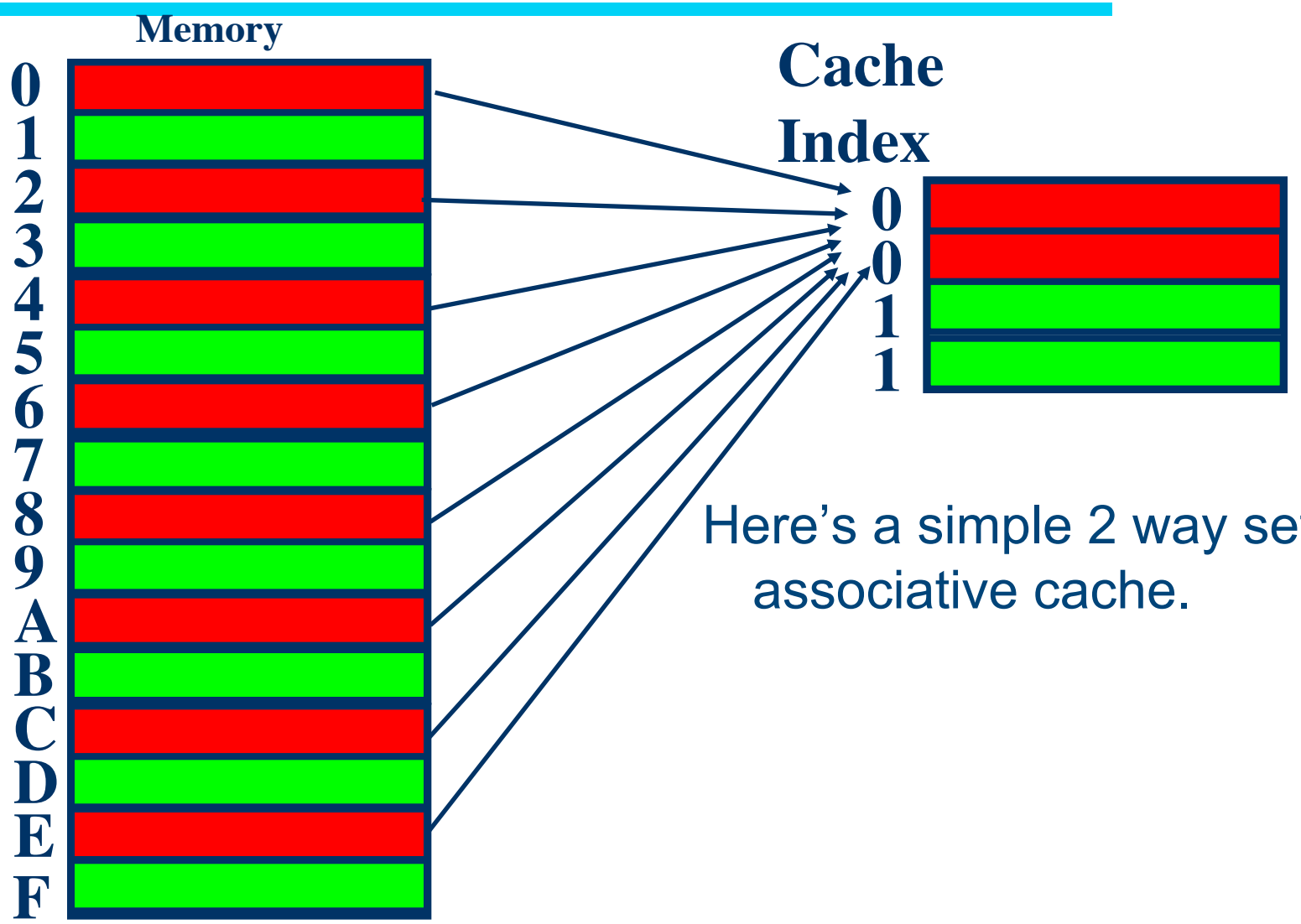
Memory address fields:

- Tag: same as before
- Offset: same as before
- Index: points us to the correct “row” (called a set in this case)

So what’s the difference?

- each set contains multiple blocks
- once we’ve found correct set, must compare with all tags in that set to find our data

Associative Cache Example



Here's a simple 2 way set associative cache.

N-Way Set Associative Cache (2/3)

Basic Idea

- cache is direct-mapped w/respect to sets
- each set is fully associative
- basically N direct-mapped caches working in parallel: each has its own valid bit and data

Given memory address:

- Find correct set using Index value.
- Compare Tag with all Tag values in the determined set.
- If a match occurs, hit!, otherwise a miss.
- Finally, use the offset field as usual to find the desired data within the block.

N-Way Set Associative Cache (3/3)

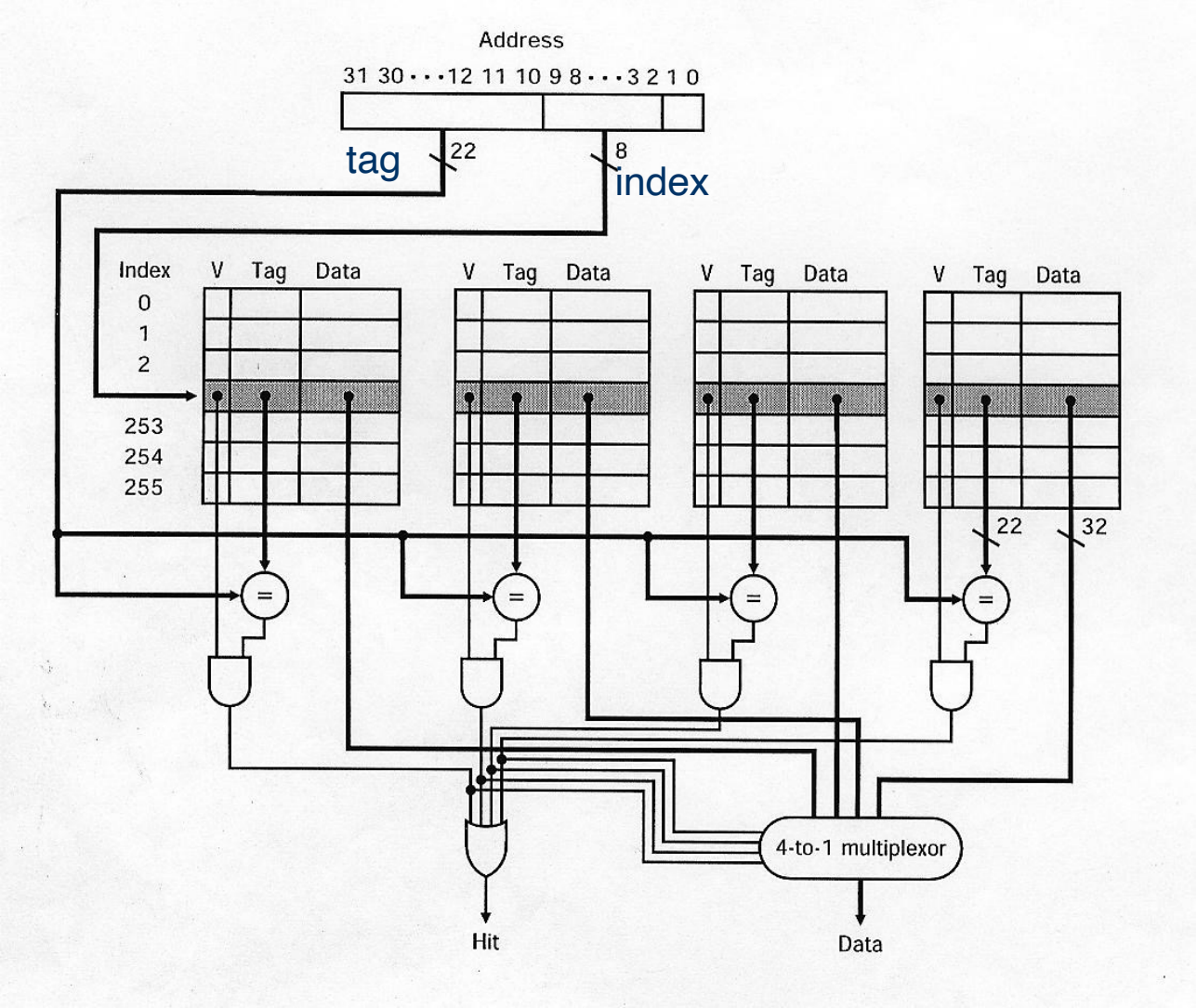
What's so great about this?

- even a 2-way set assoc cache avoids a lot of conflict misses
- hardware cost isn't that bad: only need N comparators

In fact, for a cache with M blocks,

- it's Direct-Mapped if it's 1-way set assoc
- it's Fully Assoc if it's M-way set assoc
- so these two are just special cases of the more general set associative design

4-Way Set Associative Cache Circuit



Block Replacement Policy

Direct-Mapped Cache: index completely specifies which position a block can go in on a miss

N-Way Set Assoc: index specifies a set, but block can occupy any position within the set on a miss

Fully Associative: block can be written into any position

Question: if we have the choice, where should we write an incoming block?

- If there are any locations with valid bit off (empty), then usually write the new block into the first one.
- If all possible locations already have a valid block, we must pick a replacement policy: rule by which we determine which block gets "cached out" on a miss.

Block Replacement Policy: LRU

LRU (Least Recently Used)

- Idea: cache out block which has been accessed (read or write) least recently
- Pro: temporal locality $\Rightarrow$ recent past use implies likely future use: in fact, this is a very effective policy
- Con: with 2-way set assoc, easy to keep track (one LRU bit); with 4-way or greater, requires complicated hardware and much time to keep track of this

Big Idea

How to choose between associativity, block size, replacement & write policy?

Design against a performance model

- Minimize:

Average Memory Access Time = Hit Time + Miss Penalty x Miss Rate

- influenced by technology & program behavior

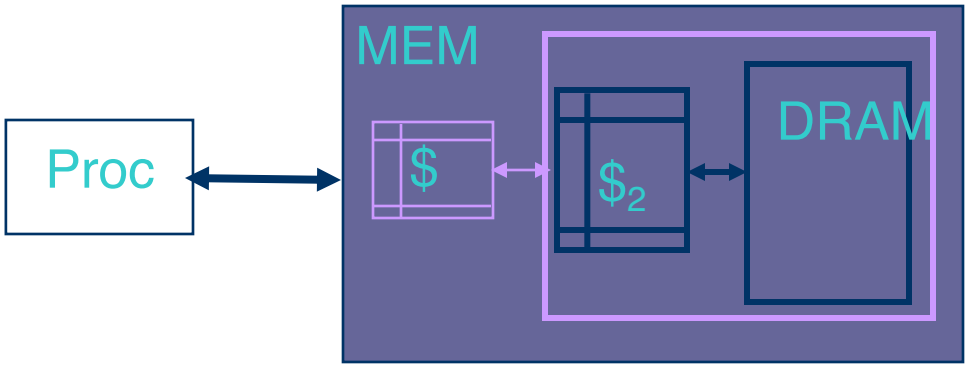
Create the illusion of a memory that is large, cheap, and fast - on average

How can we improve miss penalty?

Improving Miss Penalty

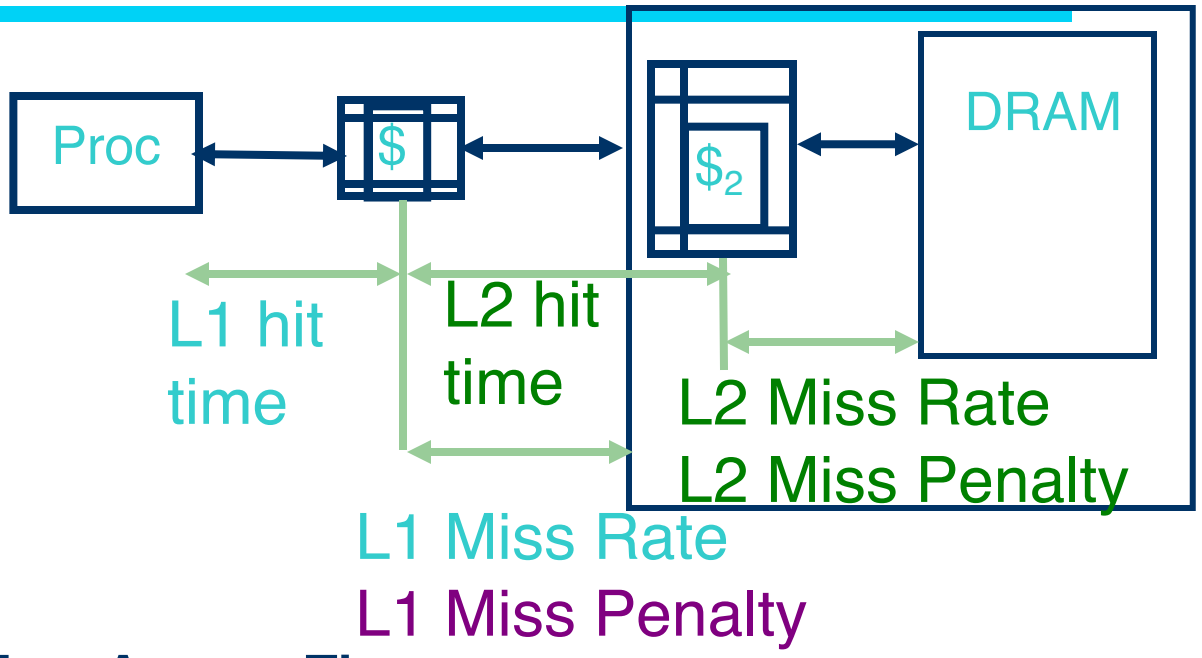
When caches first became popular, Miss Penalty ~ 10 processor clock cycles

Later, for 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM
⇒ 200 processor clock cycles!



Solution: another cache between memory and the processor cache: Second Level (L2) Cache

Analyzing Multi-level cache hierarchy



Avg Mem Access Time =

L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =

L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =

**L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)**

Example: with L2 cache

Assume

- L1 Hit Time = 1 cycle
- L1 Miss rate = 5%
- L2 Hit Time = 5 cycles
- L2 Miss rate = 15% (% L1 misses that miss)
- L2 Miss Penalty = 200 cycles

L1 miss penalty = 5 + 0.15 * 200 = 35

Avg mem access time = 1 + 0.05 x 35

= 2.75 cycles

Example: without L2 cache

Assume

- L1 Hit Time = 1 cycle
- L1 Miss rate = 5%
- L1 Miss Penalty = 200 cycles

Avg mem access time = 1 + 0.05 x 200

= 11 cycles

4x faster with L2 cache! (2.75 vs. 11)

Conclusion

We've discussed memory caching in detail. Caching in general shows up over and over in computer systems

- Filesystem cache
- Web page cache
- Game databases / tablebases
- Software memorization
- Others?

Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.

Cache design choices:

- Write through v. write back
- size of cache: speed v. capacity
- direct-mapped v. associative
- for N-way set assoc: choice of N
- block replacement policy
- 2nd level cache?
- 3rd level cache?

Use performance model to pick between choices, depending on programs, technology, budget, ...

Para saber mais ...

P&H - Capítulo 7.3