Aulas 20, 21, 22 e 23

- *Pipelining*
  - Definição - exemplo prático por analogia
  - Adaptação do conceito ao caso do MIPS
  - Problemas da solução *pipelined*
- Construção de um *datapath* com *pipelining*
  - Divisão em fases de execução
  - Execução das instruções
- *Pipelining hazards*
  - *Hazards* estruturais: replicação de recursos
  - *Hazards* de controlo: *stalling*, previsão, *delayed branch*
  - *Hazards* de dados: *stalling*, *forwarding*
- *Datapath* para o MIPS com unidades de *forwarding* e *stalling*

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Introdução

- ***Pipelining*** é uma técnica de implementação de arquiteturas do *set* de instruções (ISA), através da qual múltiplas instruções são executadas com algum grau de **sobreposição temporal**

- O objetivo é aproveitar, de forma o mais eficiente possível, os recursos disponibilizados pelo *datapath*, por forma a **maximizar a eficiência global do processador**

Pipelining - exemplo por analogia

- O exemplo de *pipelining* que iremos observar de seguida apoia-se num conjunto de tarefas simples e intuitivas: o processo de tratamento da roupa suja ☺

- Neste exemplo, o tratamento da roupa suja desencadeia-se nas seguintes quatro fases:
  1. Lavar uma carga de roupa na máquina respetiva
  2. Secar a roupa lavada na máquina de secar
  3. Passar a ferro e dobrar a roupa
  4. Arrumar a roupa dobrada no guarda roupa respetivo

Pipelining - exemplo por analogia

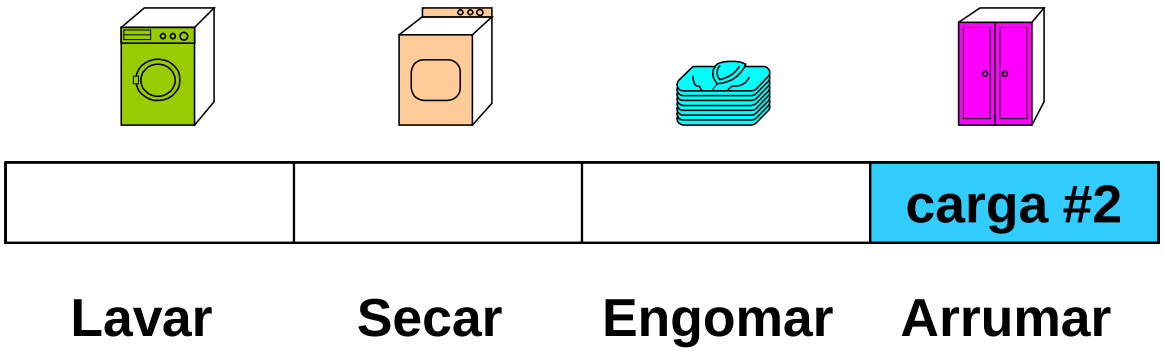
- Numa versão não *pipelined*, o processamento de N cargas de roupa seria:

carga #1

carga #2

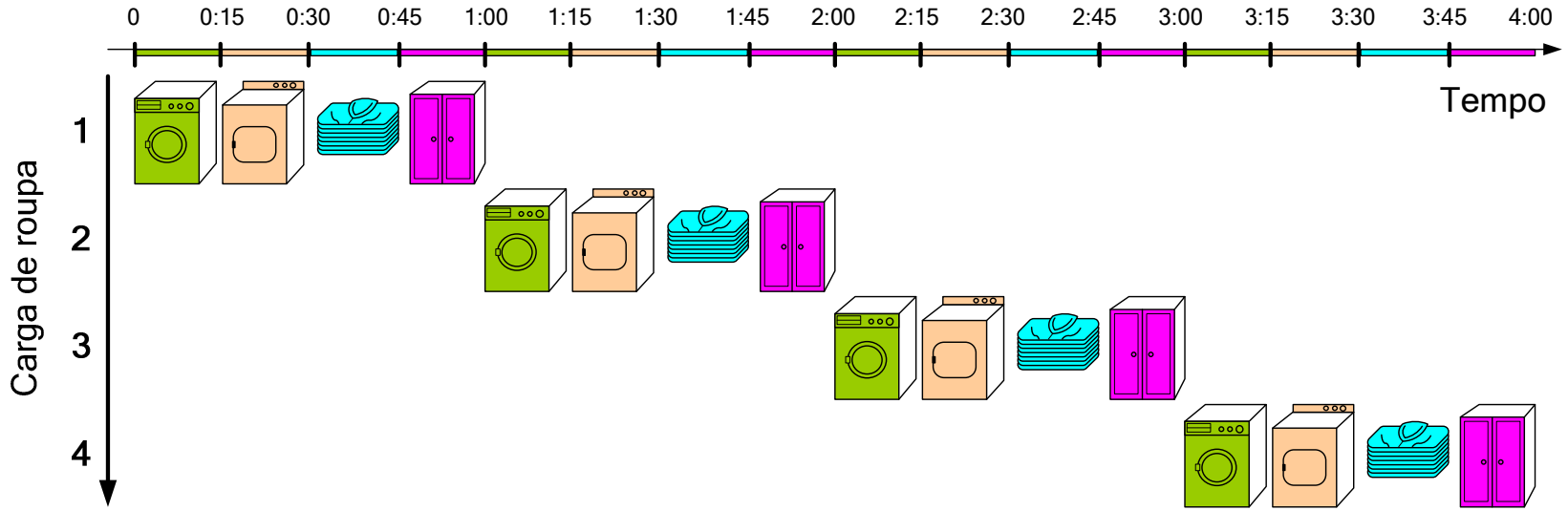
(…)

carga #n



Pipelining - exemplo por analogia

- Este processo pode então ser descrito temporalmente do seguinte modo:



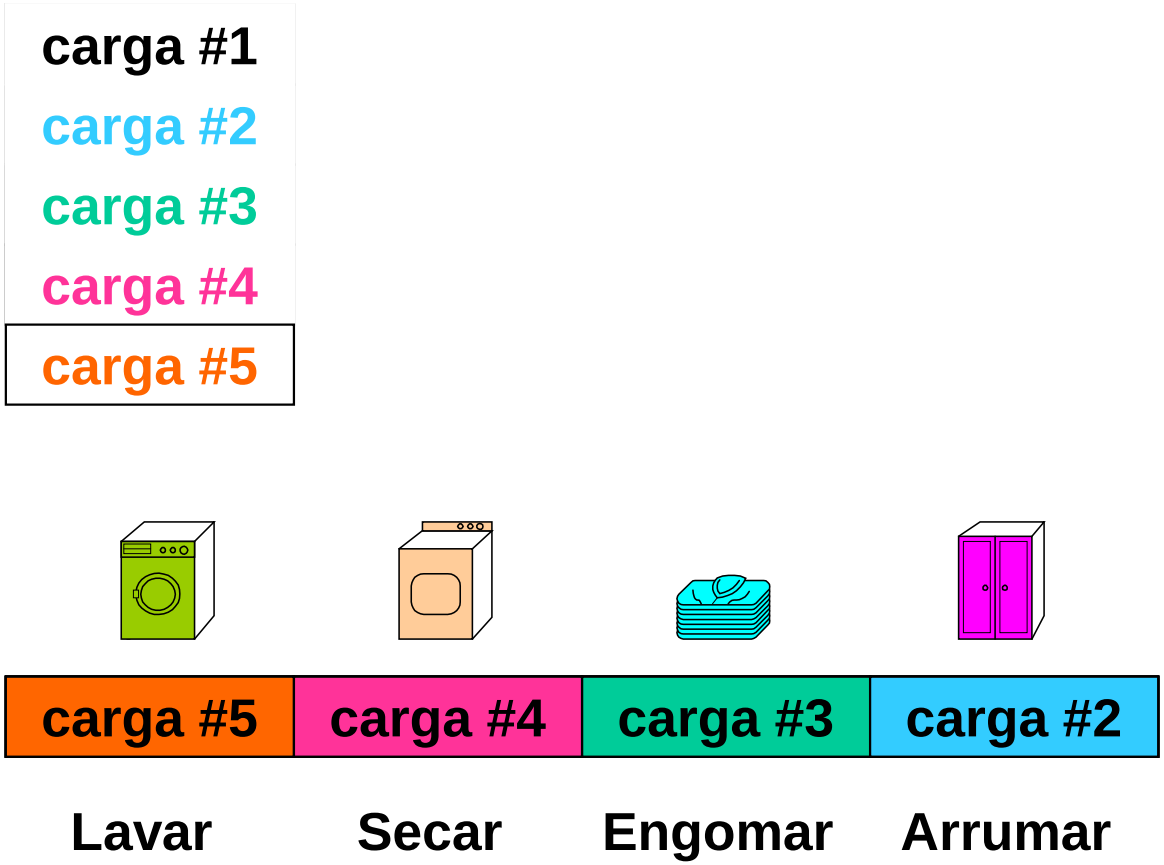
Se o tempo para tratar uma carga de roupa for uma hora, tratar quatro cargas demorará **quatro horas.**

Pipelining - exemplo por analogia

- Na versão *pipelined*, aproveita-se para carregar uma nova carga de roupa na máquina de lavar mal esteja concluída a lavagem da primeira carga
- O mesmo princípio se aplica a cada uma das restantes três tarefas
- Quando se inicia a arrumação da primeira carga, todos os passos (chamados **estágios** ou **fases** em *pipelining*) estão a funcionar em paralelo
- Maximiza-se assim a utilização dos recursos disponíveis

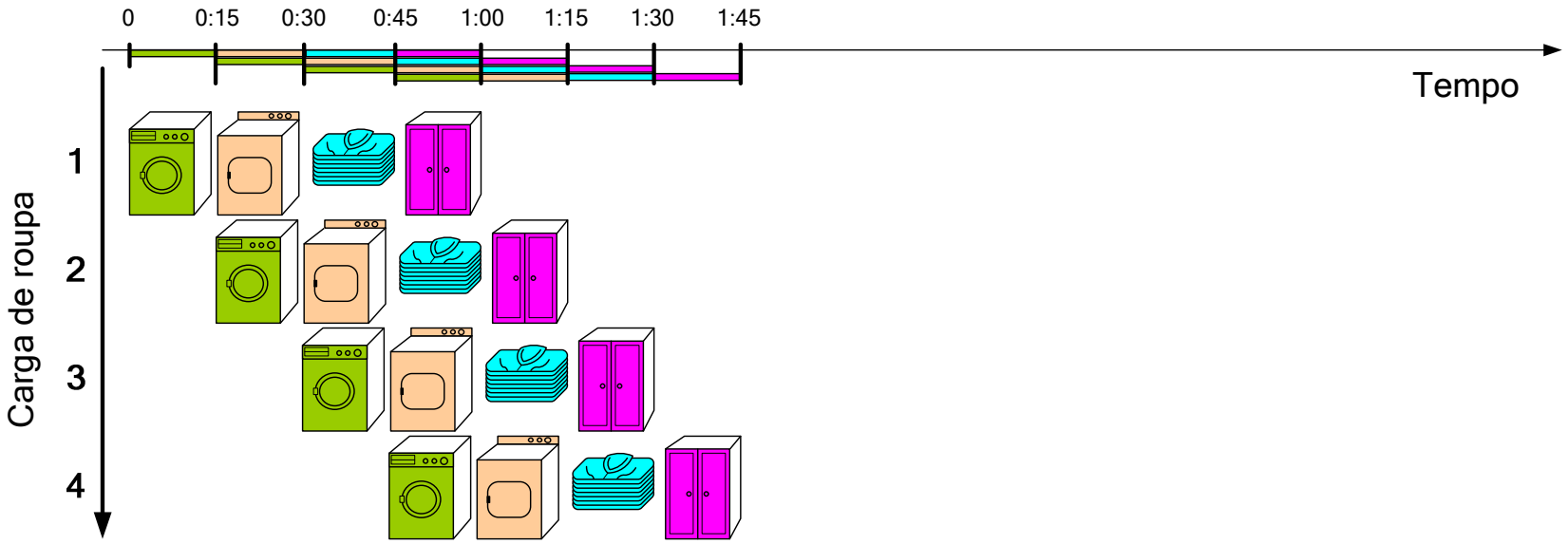
Pipelining - exemplo por analogia

- Na versão *pipelined*, o processamento das cargas de roupa seria (admitindo tempo nulo entre a comutação de tarefas):



Pipelining - exemplo por analogia

- O processo de tratamento da versão *pipelined* pode então ser descrito temporalmente do seguinte modo:



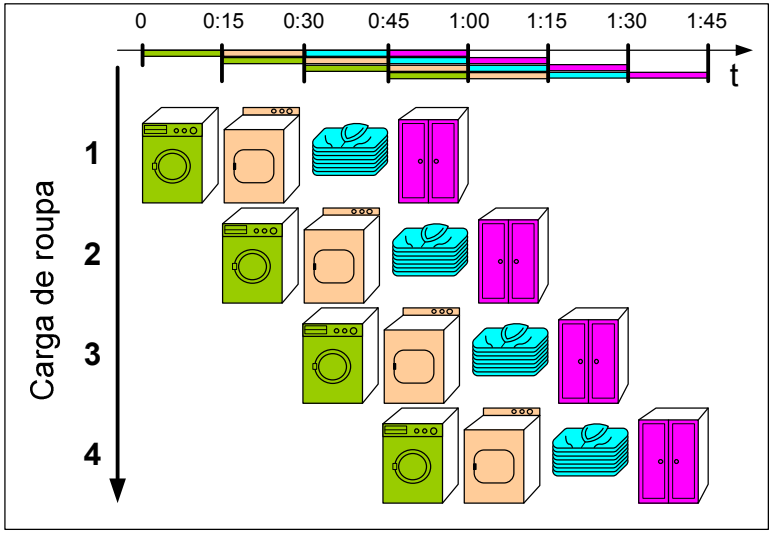
Na versão *pipelined*, o tempo total para tratar quatro cargas será de 1h45. Ou seja 135 minutos menos (240 – 105).

Pipelining - exemplo por analogia

- O paradoxo aparente da solução *pipelined* é que o tempo necessário para o processamento completo de uma carga de roupa não difere do tempo de execução da solução não *pipelined*
- A eficiência da solução com *pipelining* decorre do facto de, para um número grande de cargas de roupa, todos os passos intermédios estarem a executar em paralelo
- O resultado é o aumento do número total de cargas de roupa processadas por unidade de tempo (***throughput***)
- Qual o **ganho de desempenho** que se obtém com o sistema *pipelined* relativamente ao sistema normal?

Pipelining – ganho de desempenho

- O tratamento de N cargas de roupa num sistema com F fases demorará idealmente (admitindo que cada fase demora 1 unidade de tempo):



Sistema não *pipelined*: $T_{NON\text{-}PIPELINE} = N \times F$

Sistema *pipelined*: $T_{PIPELINE} = F + (N - 1) = (F - 1) + N$

Ganho obtido com a solução *pipelined*:

$$\frac{Desempenho_{PIPELINE}}{Desempenho_{NON\text{-}PIPELINE}} = \frac{T_{NON\text{-}PIPELINE}}{T_{PIPELINE}} = \frac{N \times F}{(F - 1) + N}$$

Se N >> (F-1), então:

$$Ganho \approx \frac{N \times F}{N} = F$$

Pipelining – ganho de desempenho

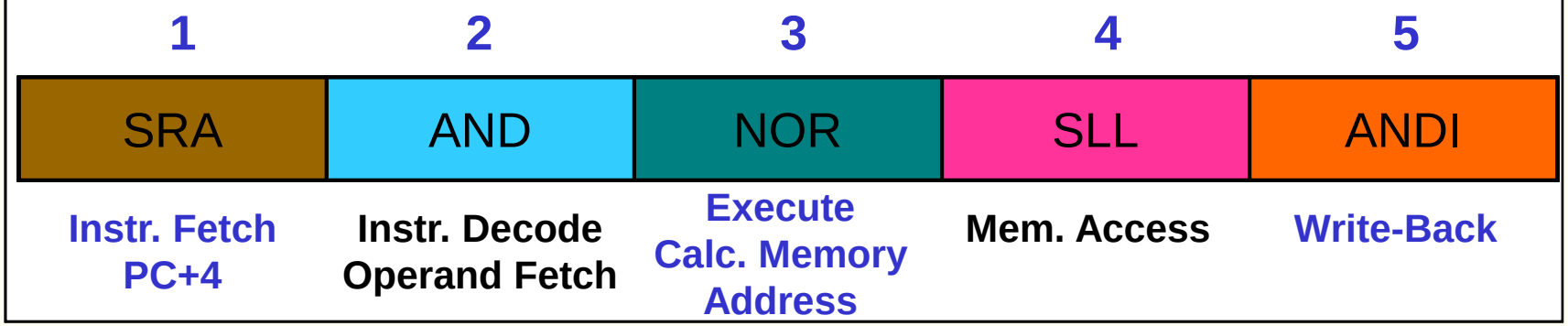
- No limite, para um número de cargas de roupa muito elevado, o ganho de desempenho (medido na forma da razão entre os tempos necessários ao tratamento da roupa, num e noutro modelo) é da ordem do **número de tarefas realizadas em paralelo** (isto é, igual ao número de fases do processo)
- Genericamente, poderíamos afirmar que o ganho em velocidade de execução é igual ao número de estágios do *pipeline* (F)
- No exemplo observado, o limite teórico estabelece que a solução *pipelined* é quatro vezes mais rápida do que a solução não *pipelined*
- A adopção de *pipelines* muito longos (com muitos estágios) pode, contudo, como veremos mais tarde, limitar drasticamente a eficiência global

Pipelining no MIPS

- Os mesmos princípios que observámos para o caso do tratamento da roupa, podem igualmente ser aplicados aos processadores
- No caso do MIPS, como já sabemos, as instruções podem ser divididas genericamente em **cinco fases** (estágios, etapas):
  1. **Instruction fetch** (ler a instrução da memória), incremento do PC
  2. **Operand fetch** (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
  3. **Execute** (executar a operação ou calcular um endereço)
  4. **Memory access** (aceder à memória de dados para leitura ou escrita)
  5. **Write-Back** (escrever o resultado no registo destino)
- Parece assim razoável admitir a construção de uma solução *pipelined* do *datapath* do MIPS que implemente cinco estágios distintos, um para cada fase da execução das instruções

Pipelining no MIPS

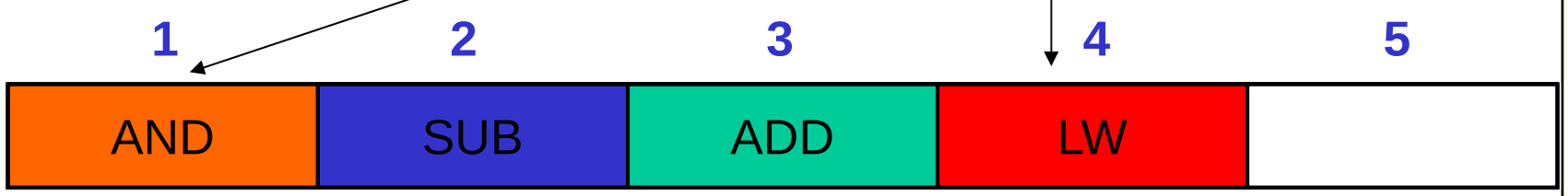
```
                or   $t1, $s3, $t9
                sub  $t2, $t9, $s3
                add  $s0, $a1, $a2
5(WB)           andi $s3, $t5, 0x01
4(MEM)          sll  $a1, $a2, 5
3(EX)           nor  $a2, $a2, $t1
2(ID)           and  $a3, $t2, $t1
1(IF)           sra  $s0, $t2, 1
                (…)
```



Pipelining – Problemas (exemplo 1)

```
lw   $t1, 0($t9)
add  $t2, $t3, $t4
sub  $t3, $t4, $t5
and  $t4, $t5, $t6
lw   $t5, 16($t9)
```

- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efetuar, simultaneamente, um acesso à memória para **leitura de dados** e para o ***instruction fetch***
- Se existir apenas uma memória para dados e código, as duas operações não podem ser executadas em paralelo

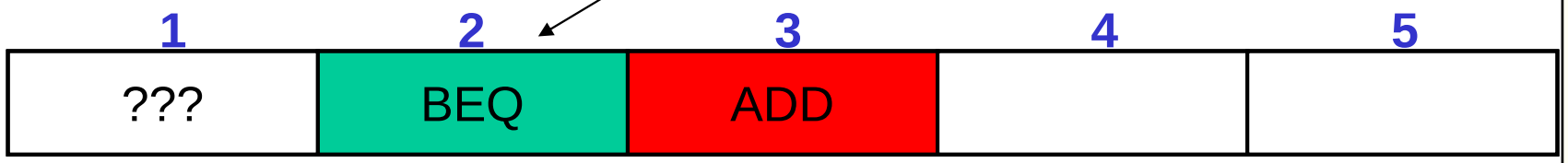


Instr. Fetch PC+4	Instr. Decode Operand Fetch	Execute Calc. Memory Address	Mem. Access	Write-Back

Pipelining – Problemas (exemplo 2)

```
        add   $t4, $t5, $t6
        beq   $t1, $t2, Z1
        lw    $s3, 300($a0)
        (…)
Z1:     or    $t7, $t8, $t9
```

- Qual a próxima instrução a ler da memória (**LW** / **OR**) assumindo que a decisão do *branch* só é tomada no 3º estágio do *pipeline*?
- Mesmo admitindo que existe h/w dedicado para avaliar a condição do *branch* logo no 2º estágio, a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória.

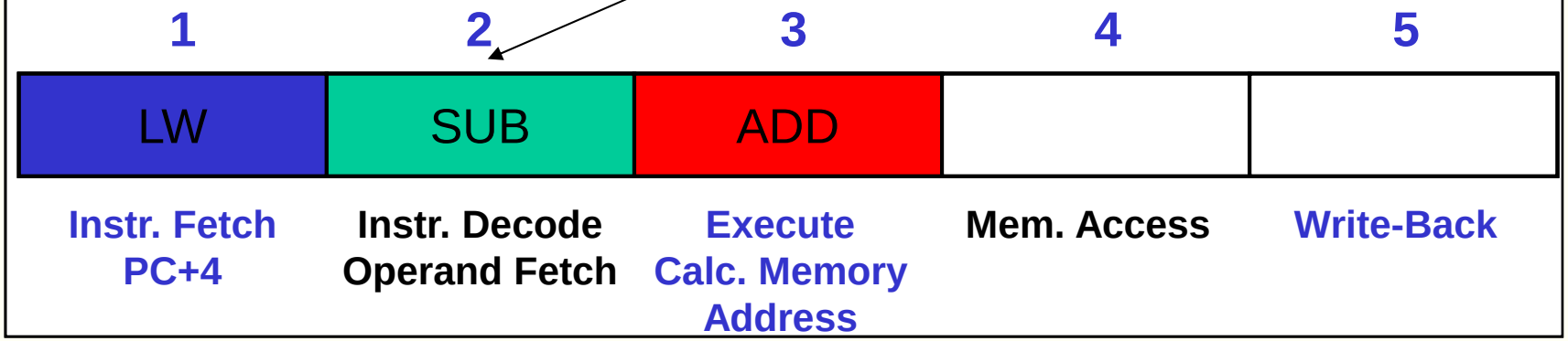


1	2	3	4	5
Instr. Fetch PC+4	Instr. Decode Operand Fetch	Execute Calc. Memory Address	Mem. Access	Write-Back

Pipelining – Problemas (exemplo 3)

```
add  $s0, $t0, $t1
sub  $t2, $s0, $t3
lw   $t4, 0($s2)
```

A instrução de subtração não pode avançar para o estágio seguinte uma vez que o seu operando **$s0** ainda não foi calculado e armazenado no registo destino pela instrução anterior.



Datapath pipelined para o MIPS

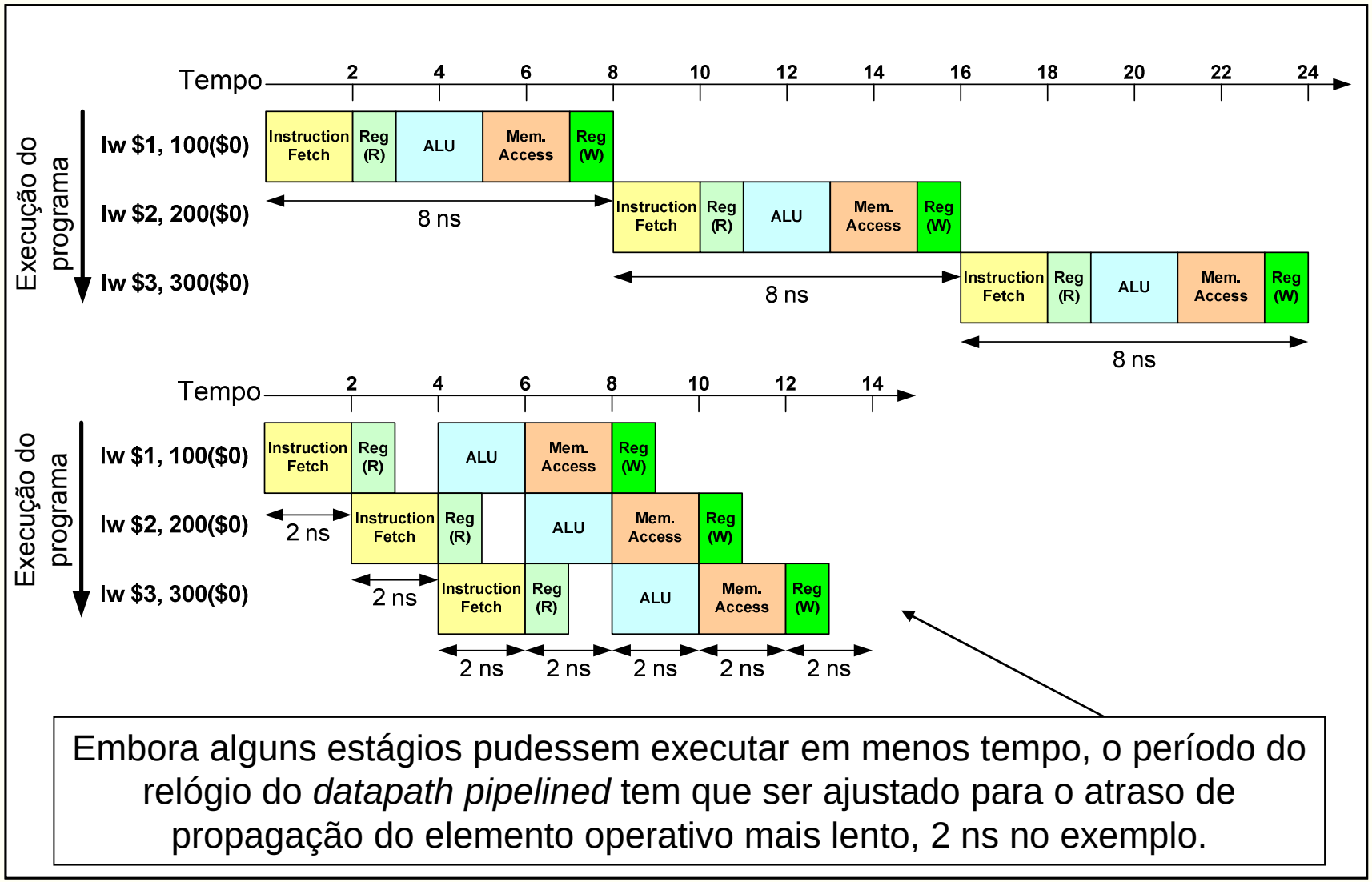
- Para tornar a discussão mais concreta, vamos construir um *datapath* que implemente um *pipeline*, que suporte as instruções que já considerámos anteriormente, isto é:
  - acesso à memória: load word (**lw**) e store word (**sw**)
  - instruções tipo R: **add**, **sub**, **and**, **or** e **slt**
  - Instruções imediatas: **addi** e **slti**
  - branch if equal (**beq**)
- Começamos por comparar os tempos necessários à execução destas instruções num *datapath single cycle* e num *datapath pipelined*, tomando como referência os seguintes tempos de execução de cada um das fases:

Instruction	Instruction Fetch	Register Read	ALU Operation	Memory Access	Register Write
Load word (lw)	2 ns	1 ns	2 ns	2 ns	1 ns
Store word (sw)	2 ns	1 ns	2 ns	2 ns	
R-Type (add, sub, and, or, slt)	2 ns	1 ns	2 ns		1 ns
Branch (beq)	2 ns	1 ns	2 ns		
Immediate (addi, slti)	2 ns	1 ns	2 ns		1 ns

Datapath pipelined para o MIPS

- De acordo com a tabela fornecida, e para a solução *single cycle*, teremos que ajustar o período do relógio ao tempo necessário para executar a instrução mais lenta (lw)
- Ou seja, na solução *single cycle* todas as instruções, independentemente do tempo mínimo que poderiam durar, serão executadas num tempo de 8ns
- Para verificarmos como comparar o tempo de execução de um trecho de código por cada uma das soluções (*pipelined* e não *pipelined*), observemos o exemplo do slide seguinte

Datapath pipelined para o MIPS



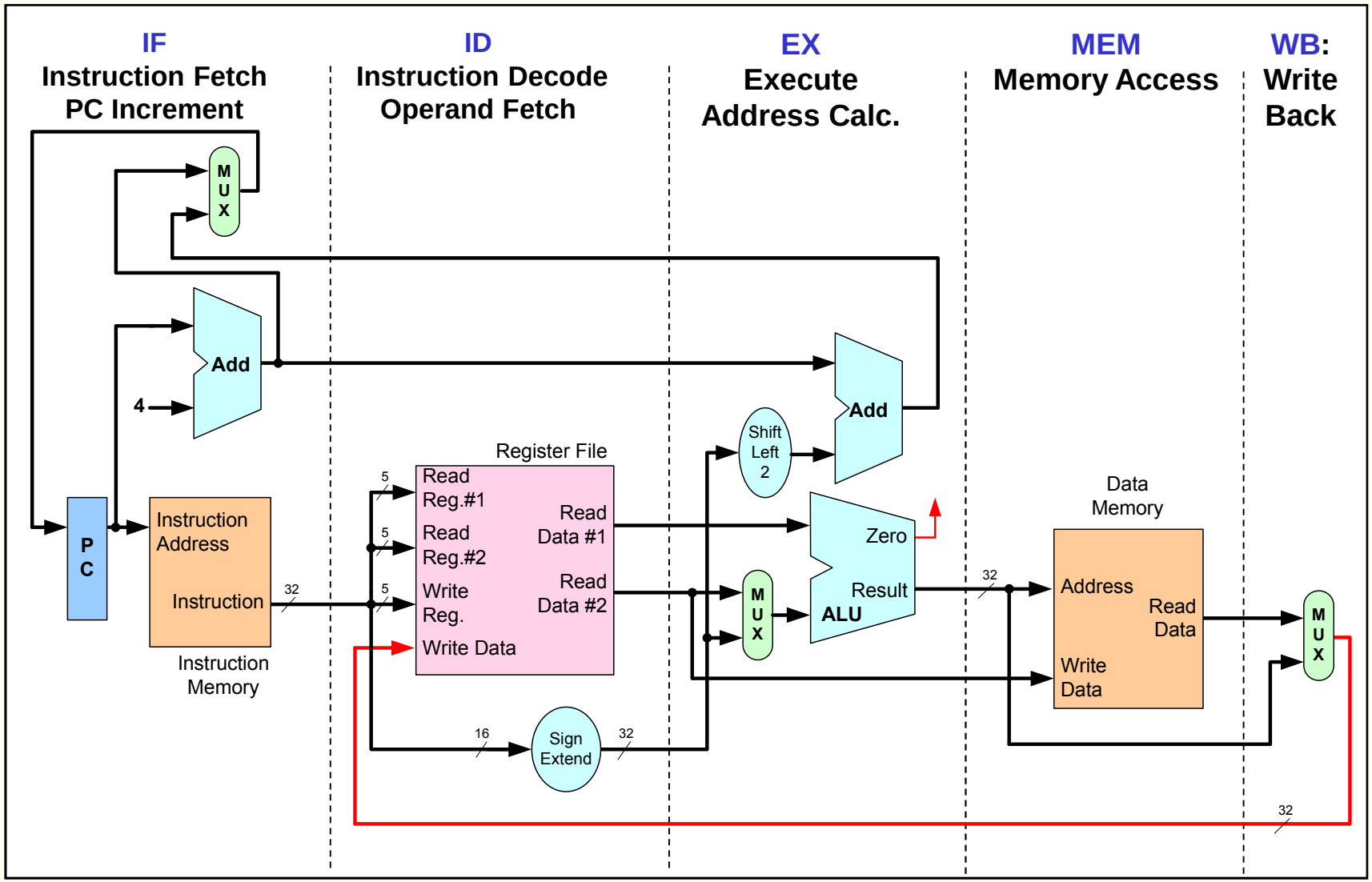
Datapath pipelined para o MIPS

- O *instruction set* do **MIPS** (***M****icroprocessor without* ***I****nterlocked* ***P****ipeline* ***S****tages*) foi concebido para uma implementação em *pipeline*. Os aspetos fundamentais a considerar são:
  - **Instruções de comprimento fixo**. *Instruction Fetch* e *Instruction Decode* podem ser feitos em estágios sucessivos uma vez que a unidade de controlo não tem que se preocupar com a dimensão da instrução descodificada
  - **Poucos formatos de instrução**, com a referência aos registos a ler sempre no mesmo campo. Isto permite que os registos sejam lidos no segundo estágio ao mesmo tempo que a instrução é descodificada pela unidade de controlo
  - **Referências à memória só aparecem em instruções de load/store**. O terceiro estágio pode assim ser usado para executar a instrução ou para calcular o endereço de memória, permitindo o acesso à memória no estágio seguinte
  - Os **operandos em memória têm que estar alinhados**. Desta forma qualquer operação de leitura/escrita da memória pode ser feita num único estágio

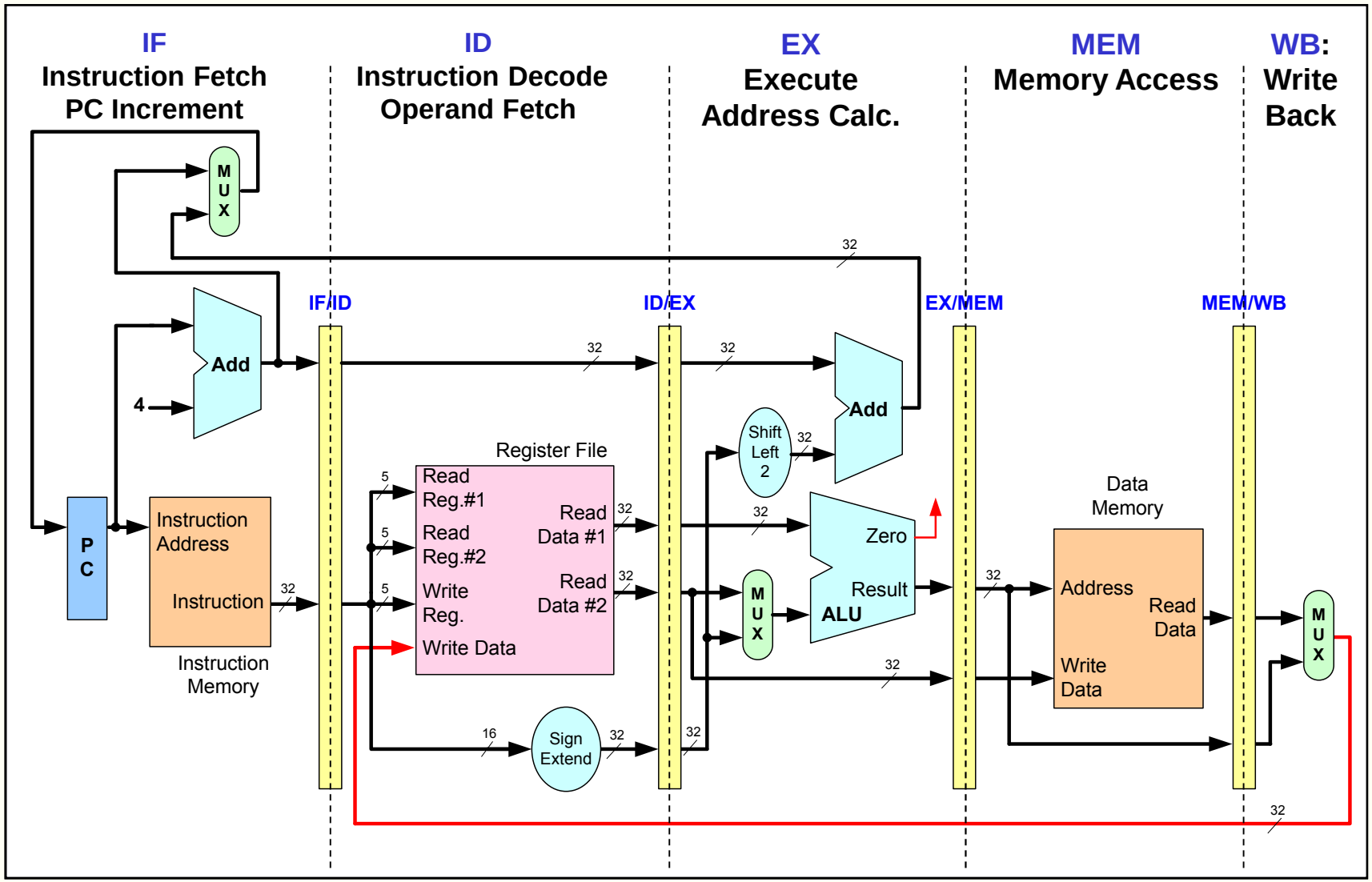
Datapath pipelined para o MIPS

- A solução *pipelined* para o MIPS parte do modelo do *datapath single-cycle*
- A organização implementa as cinco fases sequenciais em que são decomponíveis as instruções:
  1. (**IF**) - *Instruction fetch* (ler a instrução da memória), incremento do PC
  2. (**ID**) - *Operand fetch* (ler os registos) e descodificar a instrução (o formato de instrução do MIPS permite que estas duas tarefas possam ser executadas em paralelo)
  3. (**EX**) - Executar a operação ou calcular um endereço
  4. (**MEM**) - *Memory access* (aceder à memória de dados para leitura ou escrita)
  5. (**WB**) - *Write-back* (escrever o resultado no registo destino)
- Na solução apresentada no slide seguinte não são identificados os sinais de controlo nem a respetiva unidade de controlo

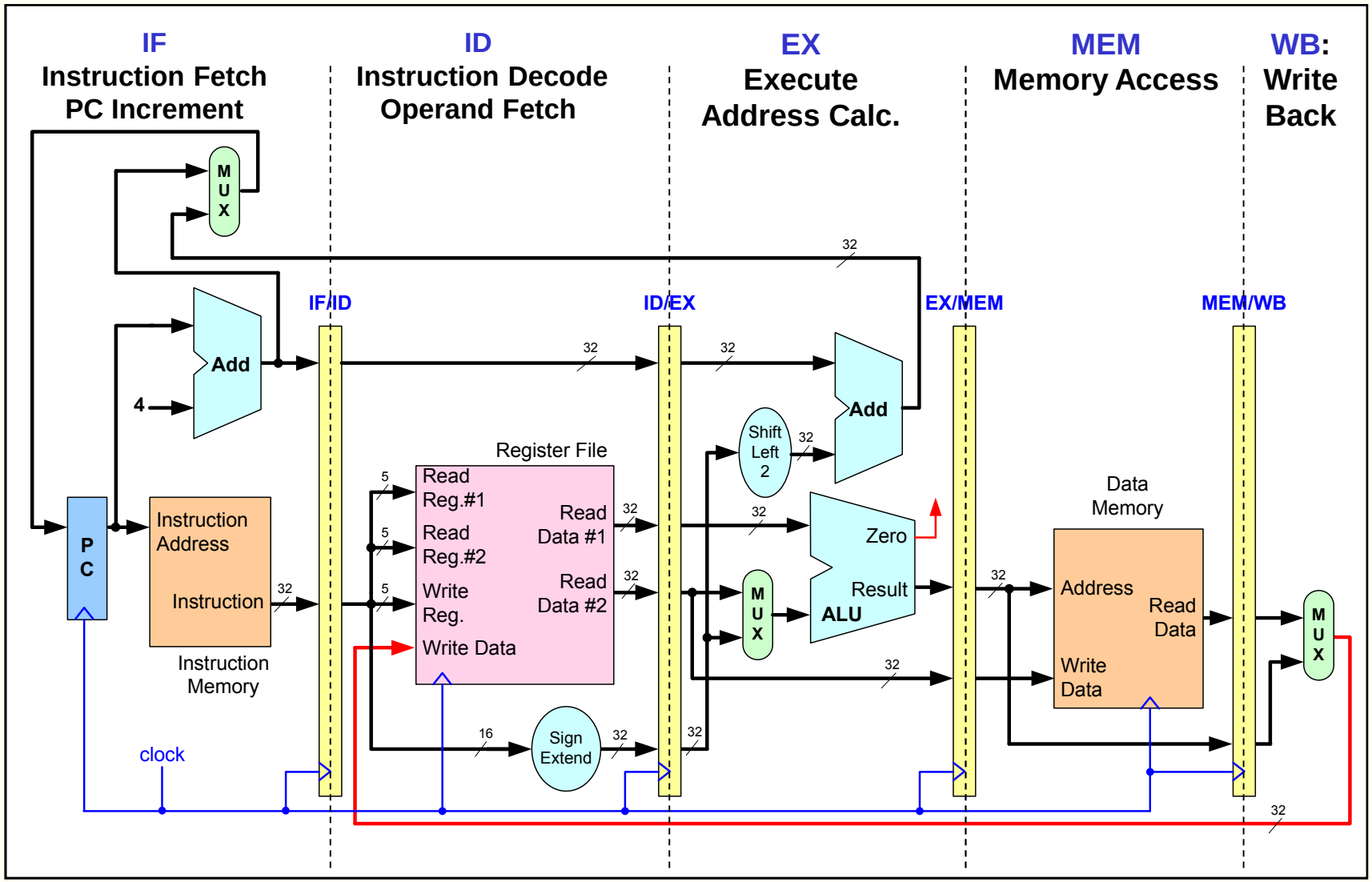
Divisão em fases de execução



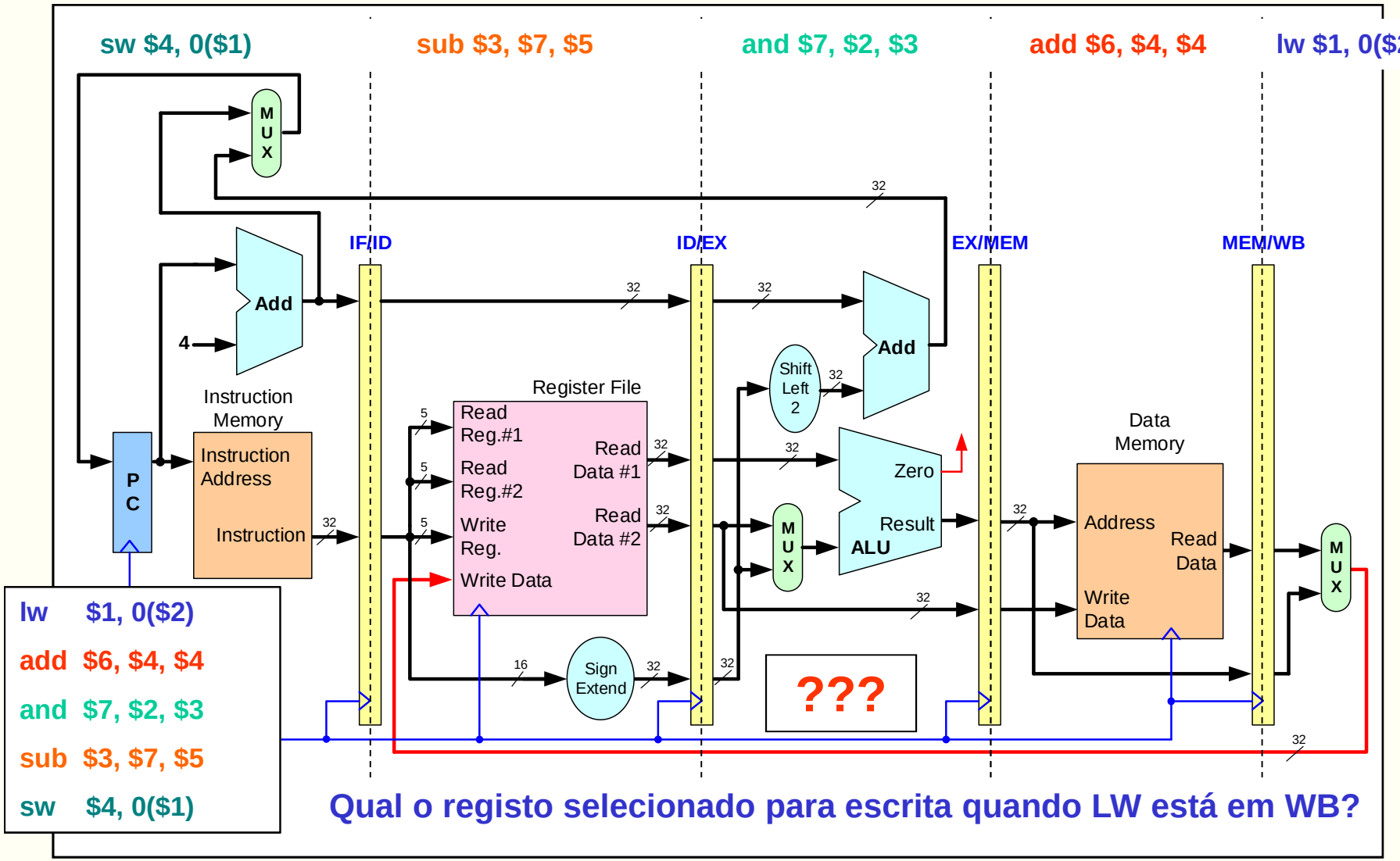
Divisão em fases de execução



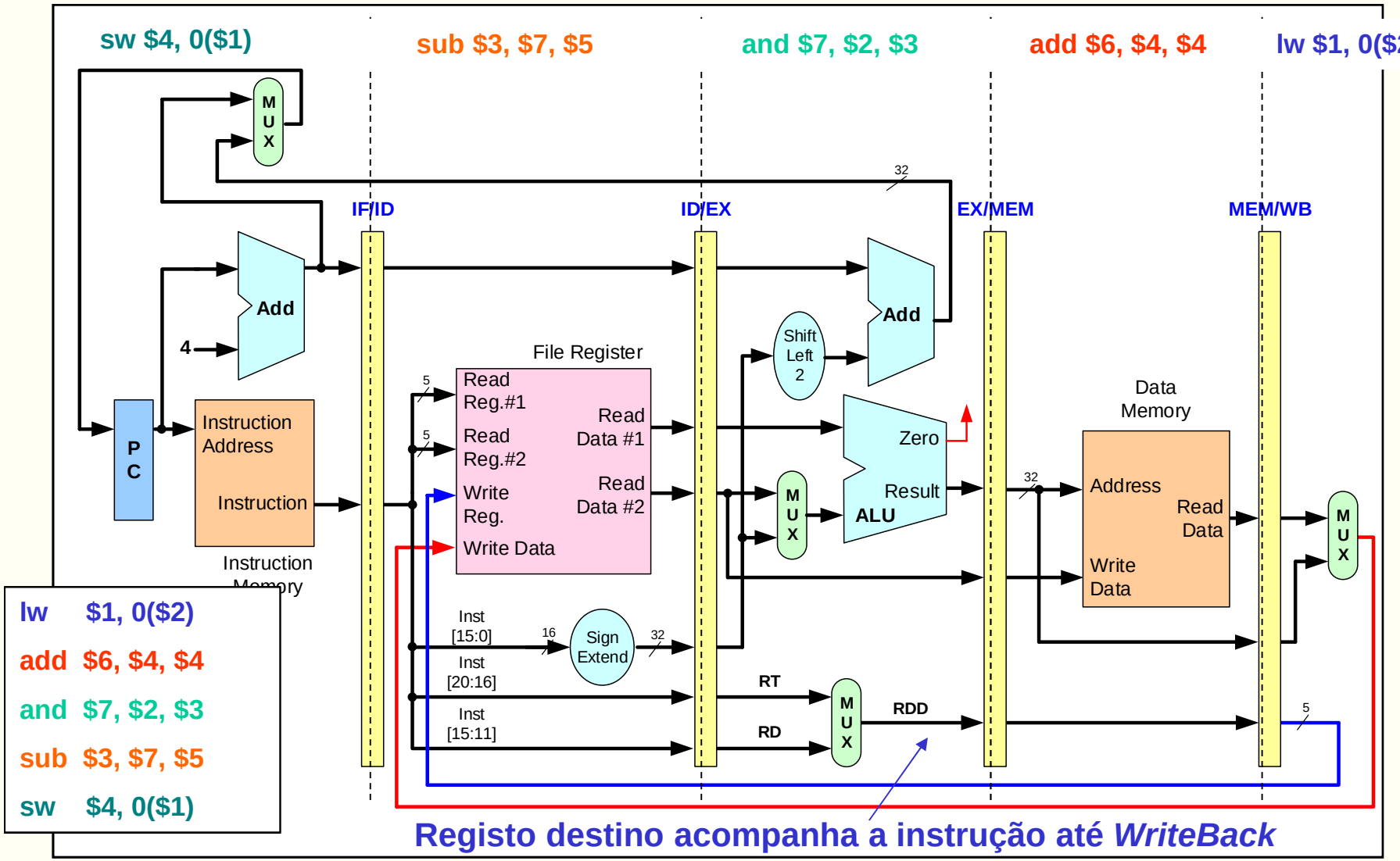
Divisão em fases de execução (com o sinal de relógio)



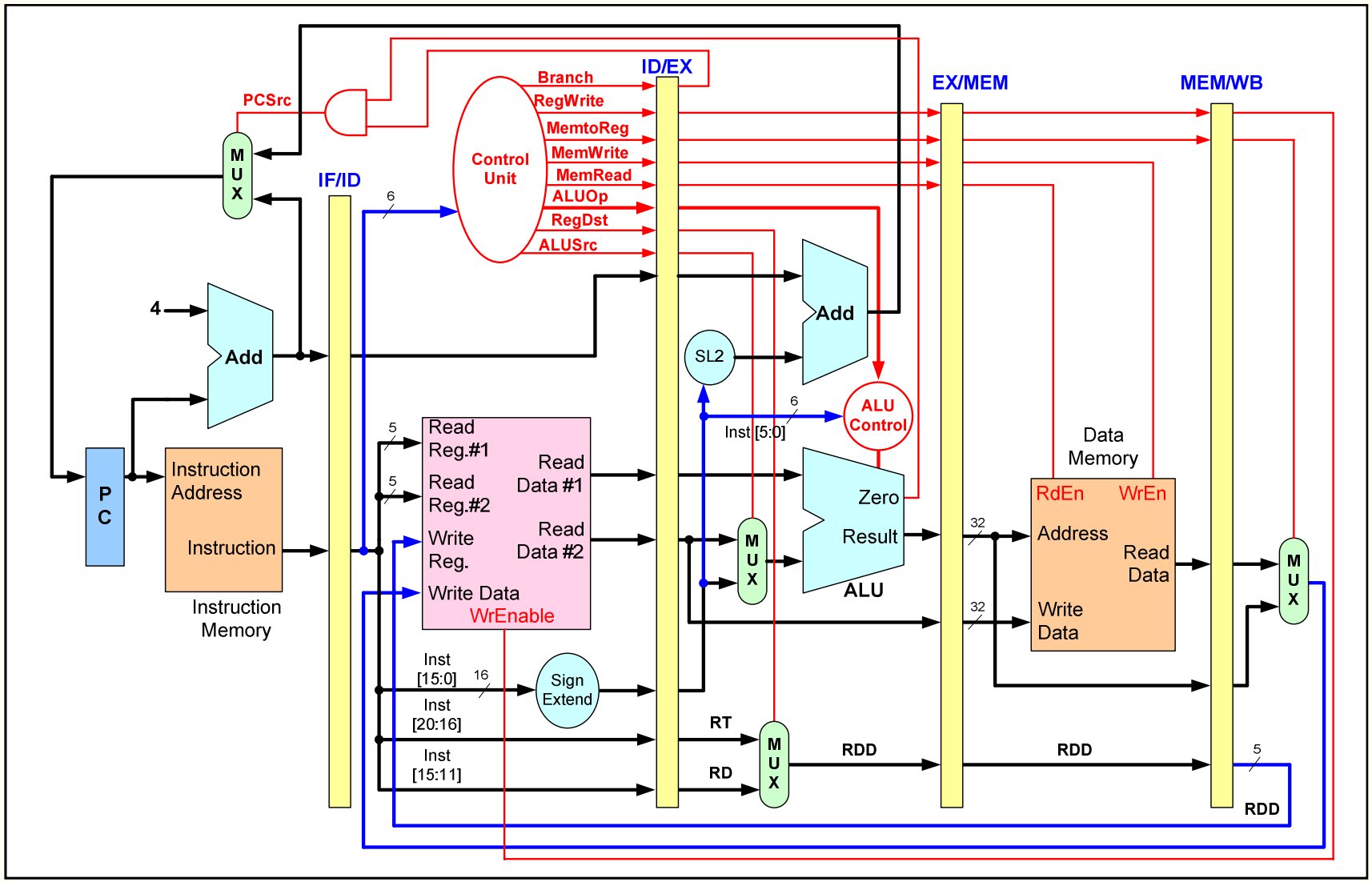
Execução de instruções



Datapath pipelined – 1ª versão



Unidade de controlo



Unidade de controlo

- A implementação *pipeline* do MIPS usa os mesmos sinais de controlo da versão *single-cycle*
- A unidade de controlo é, assim, uma **unidade combinatória** que gera os sinais de controlo em função do código da instrução (6 bits mais significativos da instrução, i.e., *opcode*) presente na fase ID
- Os sinais de controlo relevantes avançam no *pipeline* a cada ciclo de relógio (assim como os dados) estando, portanto, sincronizados com a instrução
- O sinal **RegWrite** é propagado até *WriteBack* e daí controla a escrita no *Register File* (fase ID)
- O sinal **Branch** é propagado até à fase EX (nesta versão o *branch* é resolvido nessa fase)

Exercício 1

- Determine o número de ciclos de relógio que o trecho de código seguinte demora a executar num *pipeline* de 5 fases, desde o instante em que é feito o *Instruction Fetch* da 1ª instrução, até à conclusão da última:

```
add   $1,$2,$3
lw    $2,0($4)
sub   $3,$4,$3
addi  $4,$4,4
and   $5,$1,$5  #"and" em ID, "add" já terminou
sw    $2,0($1)  #"sw" em ID, "add" e "lw"
                #  já terminaram
```

Nr_Cycles = F + (Number_of_executed_instructions – 1)

= 5 + (6 – 1) = 10 T

Num *datapath single-cycle* o mesmo código demoraria 6 ciclos de relógio a executar. Então porque razão é a execução no *datapath pipelined* mais rápida?

Quantos ciclos de relógio demora a execução num *datapath multi-cycle*?

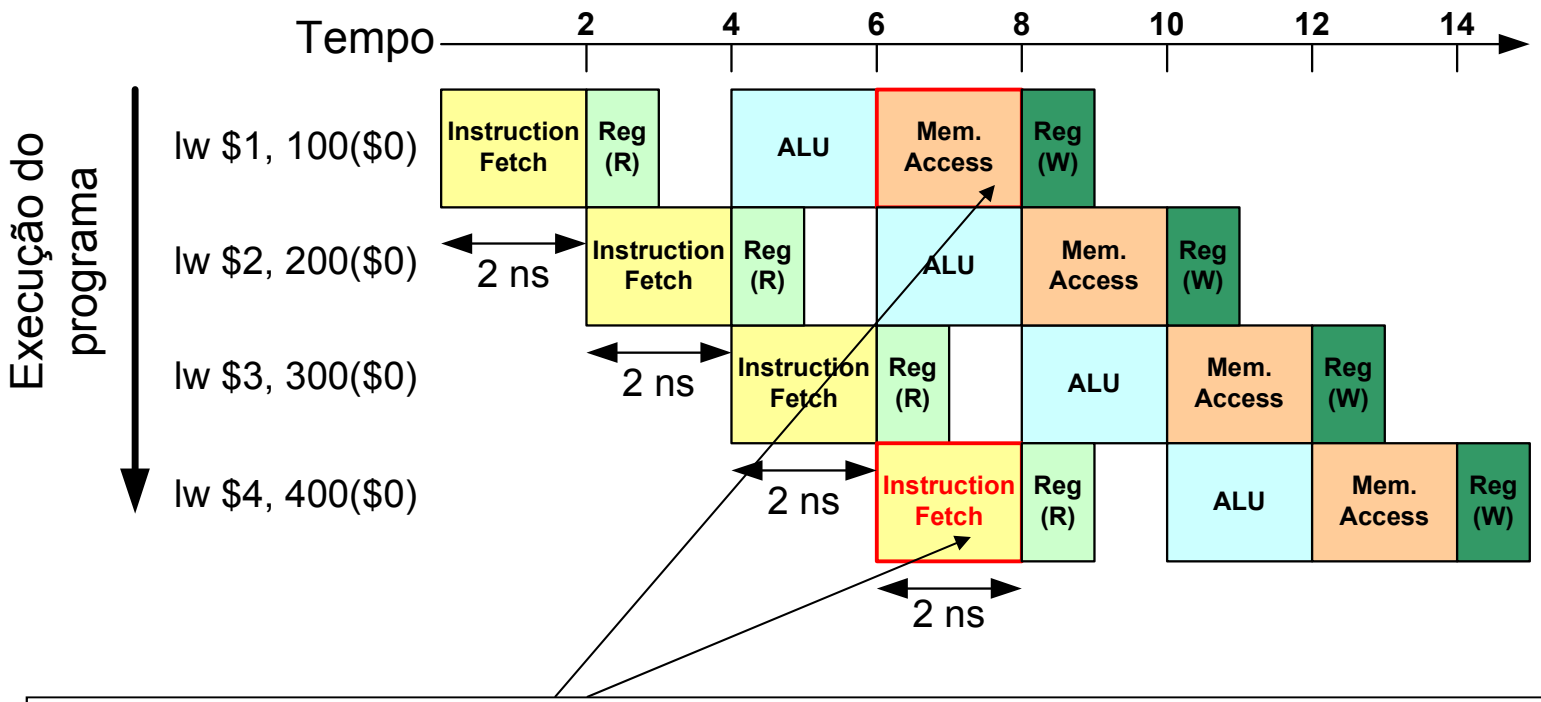
Pipeline Hazards

- Existe um conjunto de situações particulares que podem condicionar a progressão das instruções no *pipeline* no próximo ciclo de relógio
- Estas situações são designadas genericamente por ***hazards***, e podem ser agrupadas em três classes distintas:
  - ***Hazards* estruturais**
  - ***Hazards* de controlo**
  - ***Hazards* de dados**
- Nos próximos slides serão discutidas, para cada tipo de *hazard*, as origens e as consequências, mapeando depois esses aspetos ao nível da arquitetura *pipelined* do MIPS

Hazards Estruturais

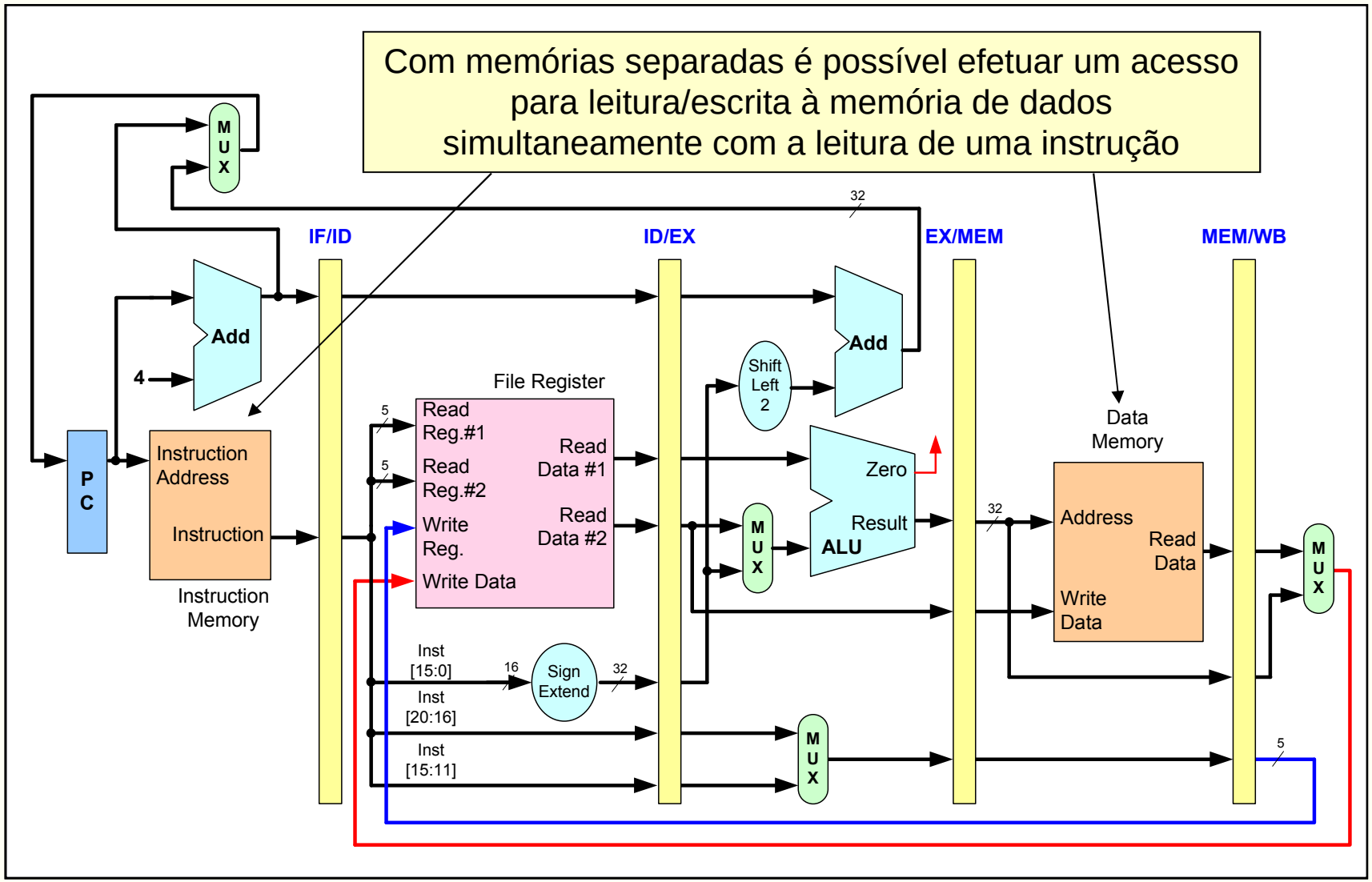
- Um ***hazard* estrutural** ocorre quando mais do que uma instrução necessita de aceder ao mesmo hardware
- Ocorre quando: 1) apenas existe uma memória ou 2) há instruções no *pipeline* com diferentes tempos de execução
- No primeiro caso o *hazard* estrutural é evitado duplicando a memória, i.e., uma memória de programa e uma memória de dados (acesso em IF não conflitua com possível acesso em MEM)
- O segundo caso está fora da análise feita nestes slides; como exemplo pode pensar-se na implementação de uma instrução mais complexa que demore 2 ciclos de relógio na fase EX

Hazards Estruturais



- No quarto estágio da primeira instrução e no primeiro da quarta instrução é necessário efetuar, simultaneamente, um acesso à memória para leitura de dados e para o *instruction fetch*
- **A não existência de memórias separadas determinaria, neste caso, a ocorrência de um *hazard* estrutural**

Hazards Estruturais



Hazards de Controlo

- Um *hazard* de controlo ocorre quando é necessário fazer o *instruction fetch* de uma nova instrução e existe numa etapa mais avançada do *pipeline* uma instrução que pode alterar o fluxo de execução e que ainda não terminou
- No caso do MIPS, as situações de *hazard* de controlo surgem com as instruções de salto, (*jumps* e *branches*)
- Exemplo:

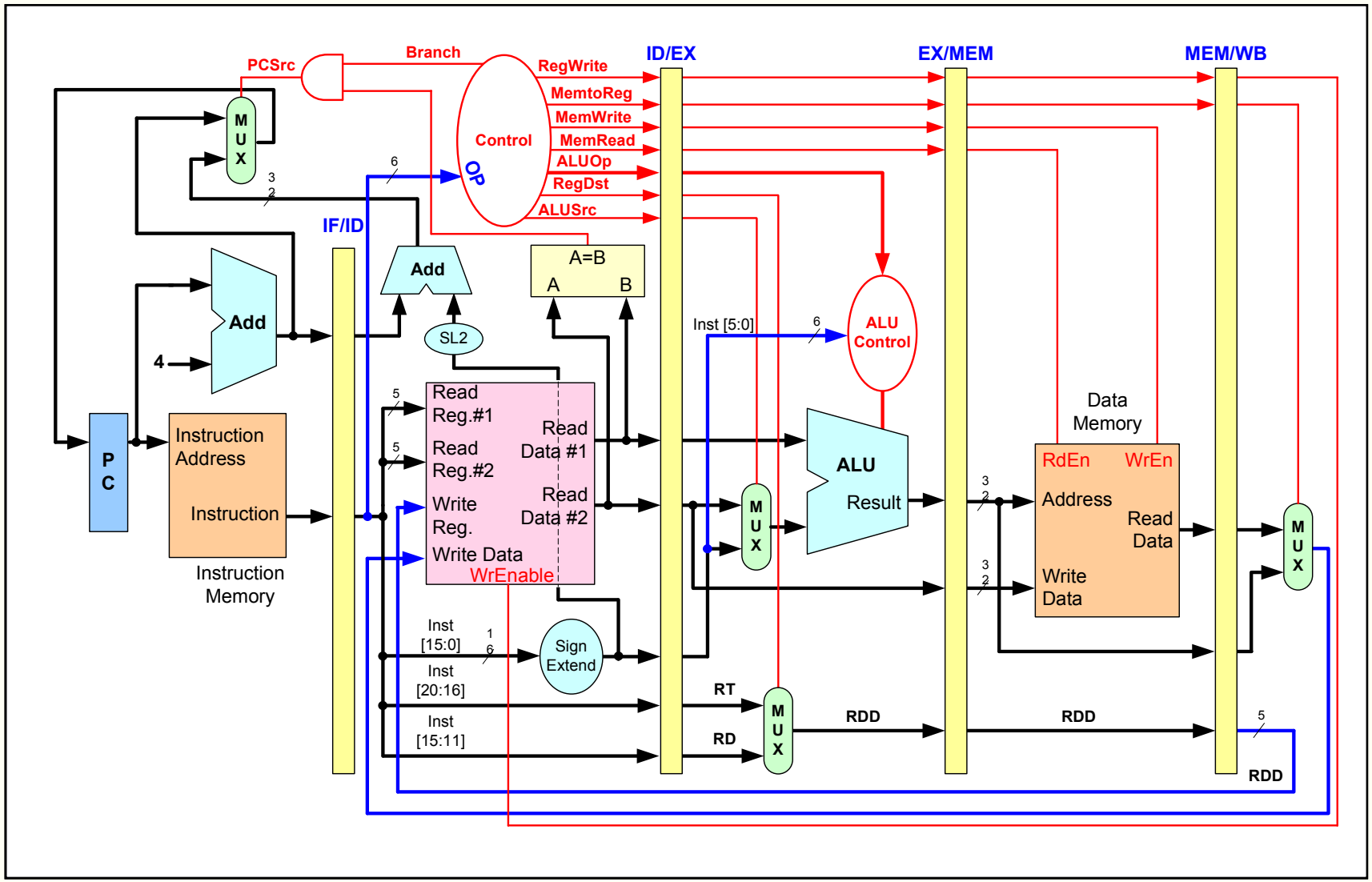
```
        beq $5,$6,next
        add $2,$3,$4
        ...
next:   lw  $3,0($4)
        ...
```

Qual a instrução que deve entrar no *pipeline* a seguir à instrução "beq"?

Hazards de Controlo

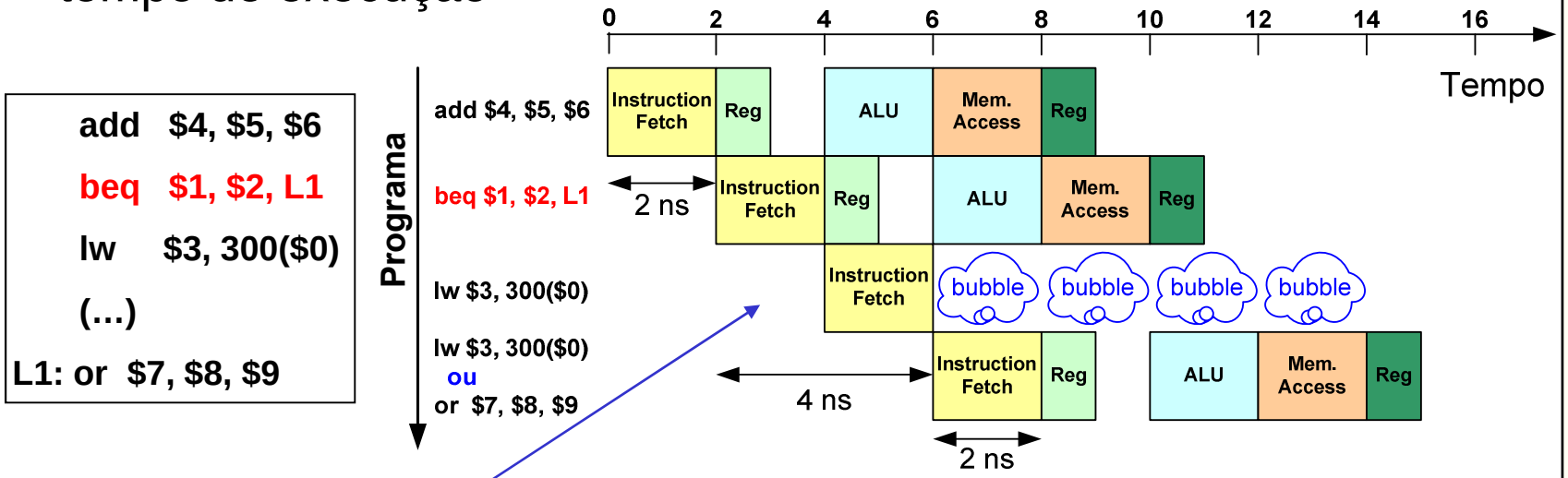
- Na versão do *datapath* apresentada anteriormente os *branches* são resolvidos em EX (3º estágio)
- Mesmo admitindo que existe hardware dedicado para avaliar a condição do *branch* logo no 2º estágio (ID), a unidade de controlo terá sempre que esperar pela execução desse estágio para saber qual a próxima instrução a ler da memória de código
- Na análise que se segue supõe-se que a **comparação dos operandos é efetuada no 2º estágio (ID)**, através de hardware adicional
- Do mesmo modo, **o cálculo do *Branch Target Address* passa também a ser efetuado em ID**

Datapath com *branches* resolvidos em ID



Hazards de Controlo

- Há mais do que uma solução para lidar com os *hazards* de controlo. A primeira que vamos analisar é designada por ***stalling*** ("parar o progresso de…")
- Nesta estratégia a unidade de controlo atrasa a entrada no *pipeline* da próxima instrução até saber o resultado do *branch* condicional
- É uma solução conservativa que tem um preço em termos de tempo de execução



Pipeline ***stall***.
Também conhecido por *bubble*

Bubble: A instrução é convertida em **NOP** (*no operation*. Exemplo: **sll $0,$0,0**)

Hazards de Controlo - *Stalling*

- Se 15% das instruções de um dado programa forem *branches*, qual o efeito desta estratégia no desempenho da arquitetura, admitindo que os *branches* são resolvidos em ID?

  Sem *stalls*: CPI = 1

  Com *stalls*: CPI = 1 + 1 * 0,15 = 1,15

  Relação de desempenho = 1 / 1,15 = 0,87

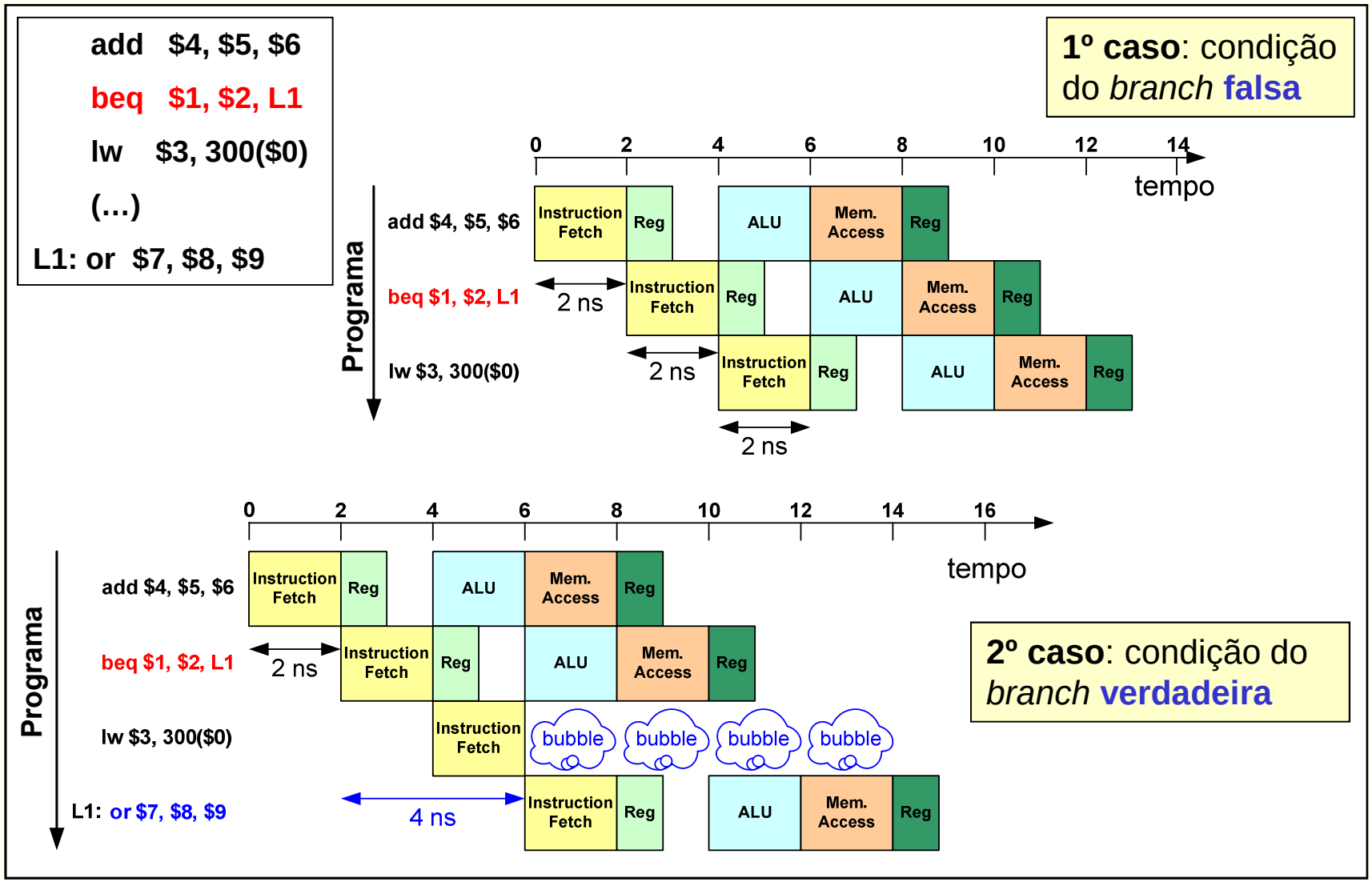
- A degradação do desempenho é tanto maior quanto mais tarde for resolvida a instrução de *branch.* Na mesma situação, se o *branch* for resolvido em EX, a relação passa a ser:

  Relação de desempenho = 1 / (1 + 2 * 0,15) = 0,77

Hazards de Controlo

- Uma solução alternativa ao *pipeline stalling* é designada por **previsão** (*prediction*):
  - Assume-se que a condição do *branch* é falsa (*branch not taken*), pelo que a próxima instrução a ser executada será a que estiver em PC+4 – estratégia designada por **previsão estática *not taken***
  - Se a previsão falhar, a instrução entretanto lida (a seguir ao *branch*) é descartada (convertida em **nop**), continuando o *instruction fetch* na instrução correta
- Se a previsão estiver certa esta estratégia permite poupar tempo; para o exemplo anterior, se a previsão for correta 50% das vezes, a relação de desempenho passa a ser: Ganho = 1 / (1 + 1 * 0,15 / 2) = 0,93

Hazards de Controlo – previsão *not taken*



Hazards de Controlo – previsão

- Os previsores usados nas arquiteturas atuais são mais elaborados
- **Previsores estáticos**: o resultado da previsão não é dependente do resultado da execução das instruções:
  - **Previsor *Not taken***
  - **Previsor *Taken***
  - **Previsor *Backward taken, Forward not taken*** (BTFNT)
- **Previsores dinâmicos**: o resultado da previsão depende da história de *branches* anteriores:
  - Guardam informação do resultado *taken/not taken* de *branches* anteriores e do *target address*
  - A previsão é feita com base na informação guardada

Hazards de Controlo – a solução do MIPS

- Uma outra alternativa para resolver os *hazards* de controlo, adotada no MIPS, é designada por ***delayed branch***
- Nesta abordagem, o processador **executa sempre a instrução que se segue ao *branch***, independentemente de a condição ser verdadeira ou falsa
- Esta técnica é implementada com a ajuda do **compilador/assembler** que:
  - organiza as instruções do programa por forma a trocar a ordem do *branch* com uma instrução anterior (desde que não haja dependência entre as duas), ou
  - não sendo possível efetuar a troca de instruções introduz um **NOP** ("no operation"; ex.: **sll, $0, $0, 0**) a seguir ao *branch*
- Não é uma técnica comum nos processadores modernos

Hazards de Controlo – *delayed branch*

- Esta técnica **é escondida do programador** pelo compilador/assembler:

Código original

```
    add   $4, $5, $6
    beq   $1, $2, L1
    lw    $3, 300($0)
    (…)
L1: or  $7, $8, $9
```

Assembler troca a ordem das duas 1ªs instruções

Código reordenado

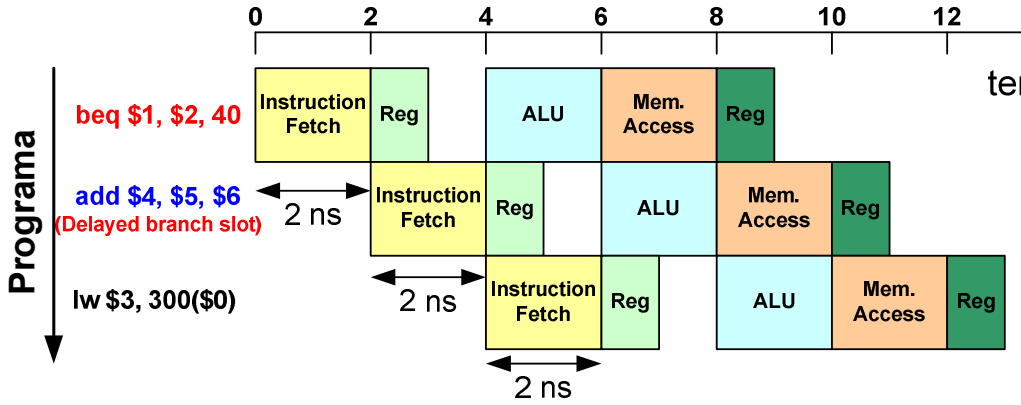
```
    beq   $1, $2, L1
    add   $4, $5, $6
    lw    $3, 300($0)
    (…)
L1: or  $7, $8, $9
```

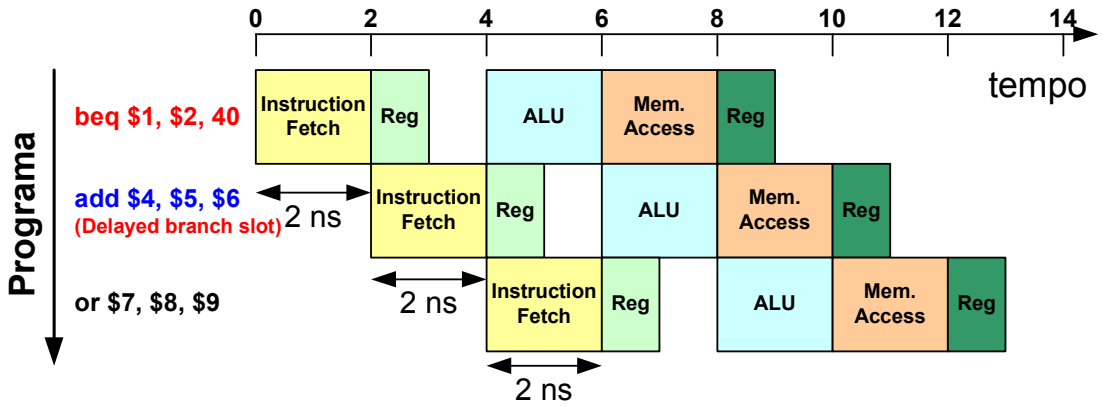
- Neste exemplo a instrução "**beq**" não depende do resultado produzido pela instrução "**add**", logo a troca das duas não altera o resultado final do programa
- A instrução "**add**" é executada independentemente do resultado do "**beq**"

Hazards de Controlo – *delayed branch*

```
beq   $1, $2, L1
add   $4, $5, $6
lw    $3, 300($0)
(…)
L1: or  $7, $8, $9
```

1º caso: condição do *branch* **falsa**





2º caso: condição do *branch* **verdadeira**