Pipelining

Luís Nogueira

luis@dei.isep.ipp.pt

Departamento Engenharia Informática

Instituto Superior de Engenharia do Porto

Análise de performance

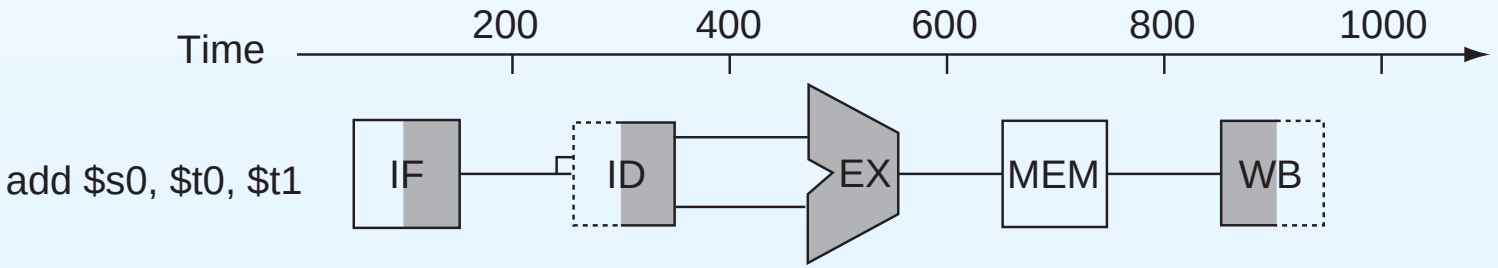
- Desenho “ciclo único” de relógio é ineficiente
  - Todas as instruções demoram 1 ciclo de relógio
  - Determinado pelo caminho mais longo nos circuitos (`lw`)
  - Viola princípio de optimização de casos frequentes
- Alternativa 1: Variar comprimento do ciclo de relógio
  - Todas as instruções demoram 1 ciclo de relógio
  - No entanto, frequência de relógio variável
  - Extrema dificuldade de implementação
  - Overhead pode facilmente superar vantagens obtidas

Análise de performance

- Alternativa 2: Desenho “ciclo múltiplo”
  - Ciclo de relógio menor
  - Instruções divergem nos ciclos de relógio necessários
  - Logo, partilham unidades funcionais
  - Temos que redesenhar o nosso processador!
- Alternativa 3: Pipelining
  - Dividir execução da instrução em fases
  - Ciclo de relógio menor (fase em vez de instrução)
  - Todas as instruções demoram mesmo número de ciclos
  - Sobrepor processamento de várias instruções em fases distintas
  - Melhorar performance global sem diminuir tempo de execução da instrução

Divisão da instrução em fases

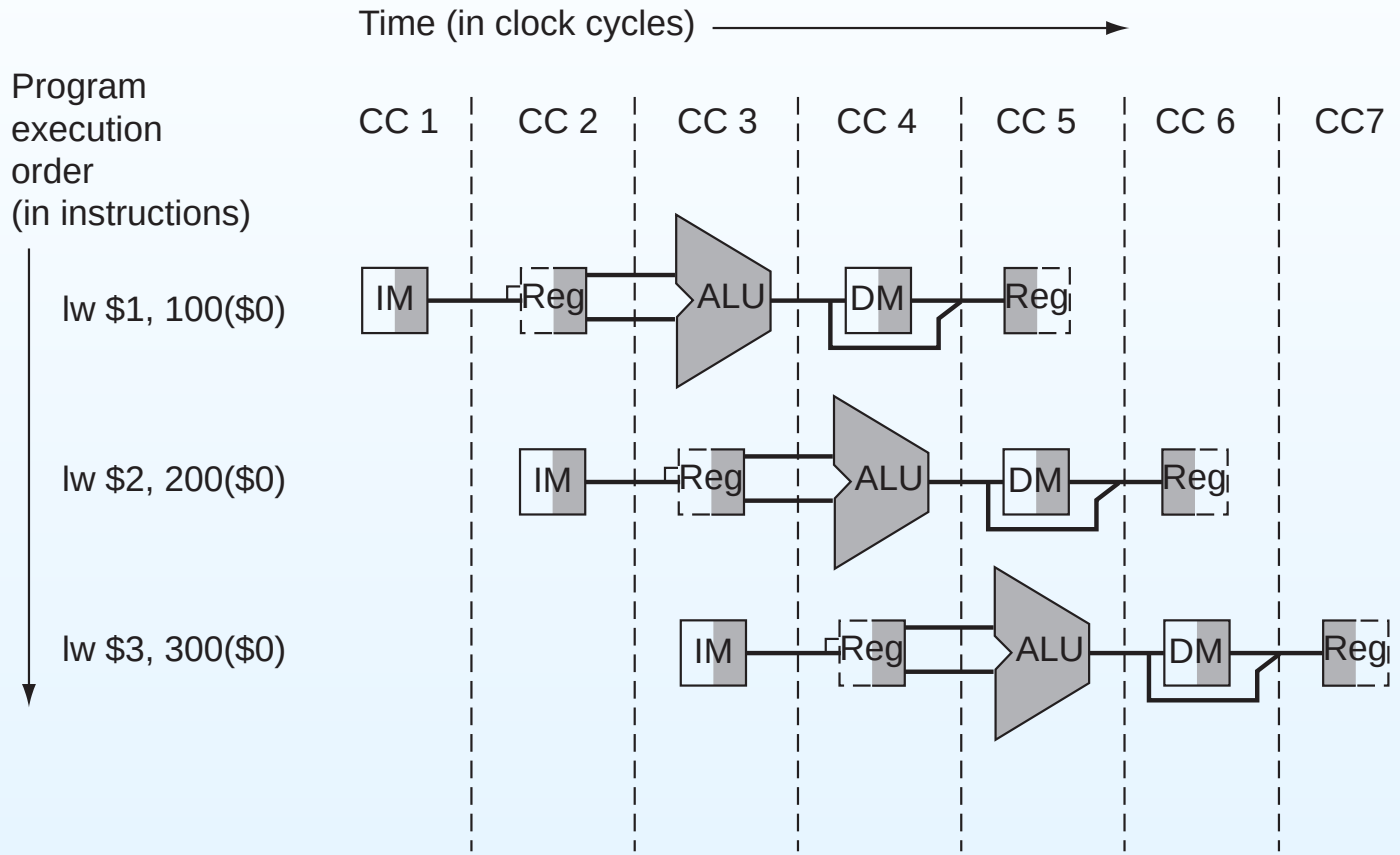
- **Instruction Fetch** instrução é transferida da memória
- **Instruction Decode** tipo e operandos são determinados
- **Execute** operação executada/endereço calculado
- **Memory Access** acesso a memória em lw/sw
- **Write Back** resultado escrito em registo



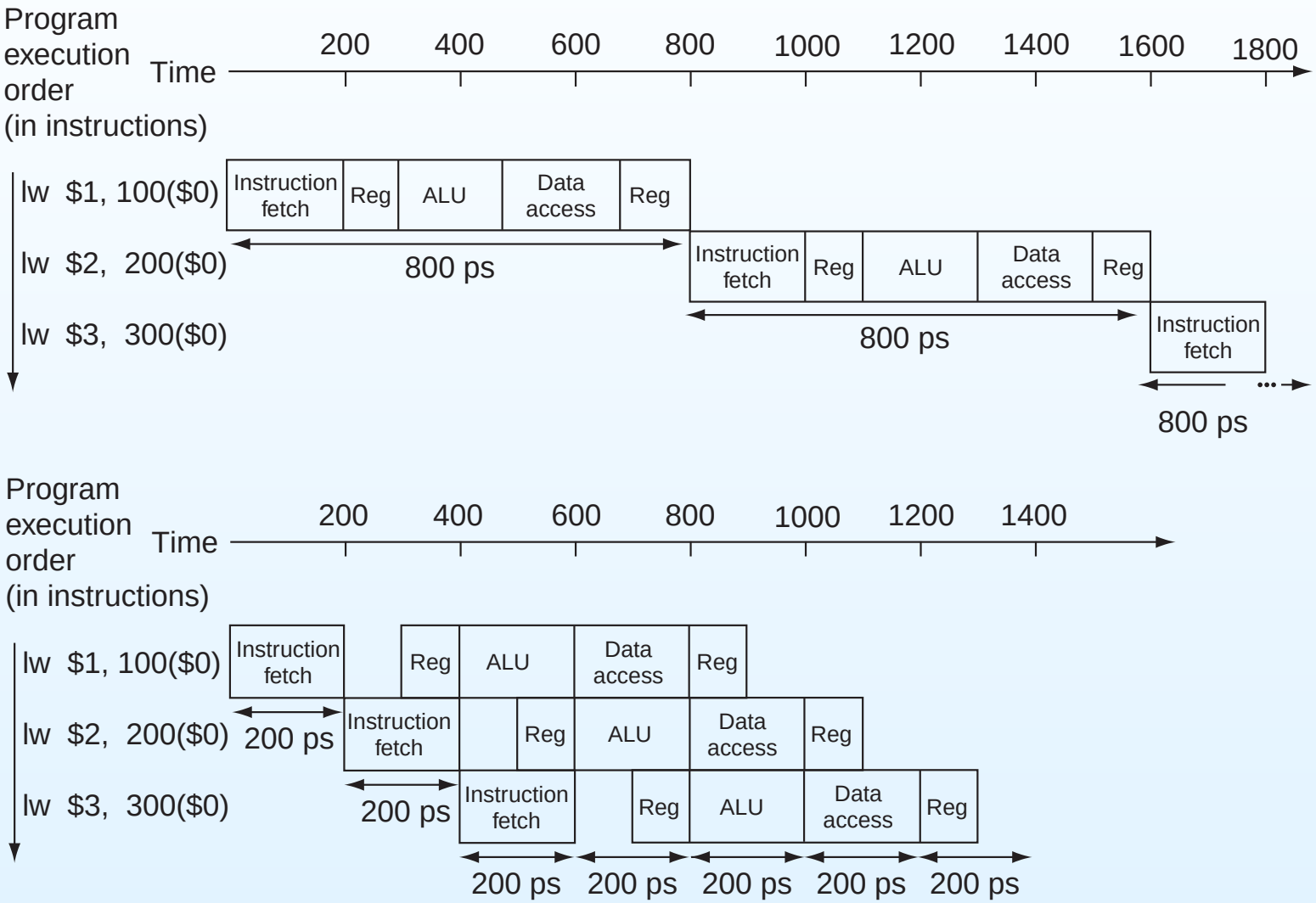
Sobreposição de execução

- Cada fase demora 1 ciclo de relógio
  - Ciclo de relógio menor
  - Instruções demoram 5 ciclos
- Como aumentar performance?
- Sobrepondo execução das instruções
  - Em fases distinctas
  - Sem partilha de hardware na mesma fase
- Semelhante a uma linha de montagem

Sobreposição de execução



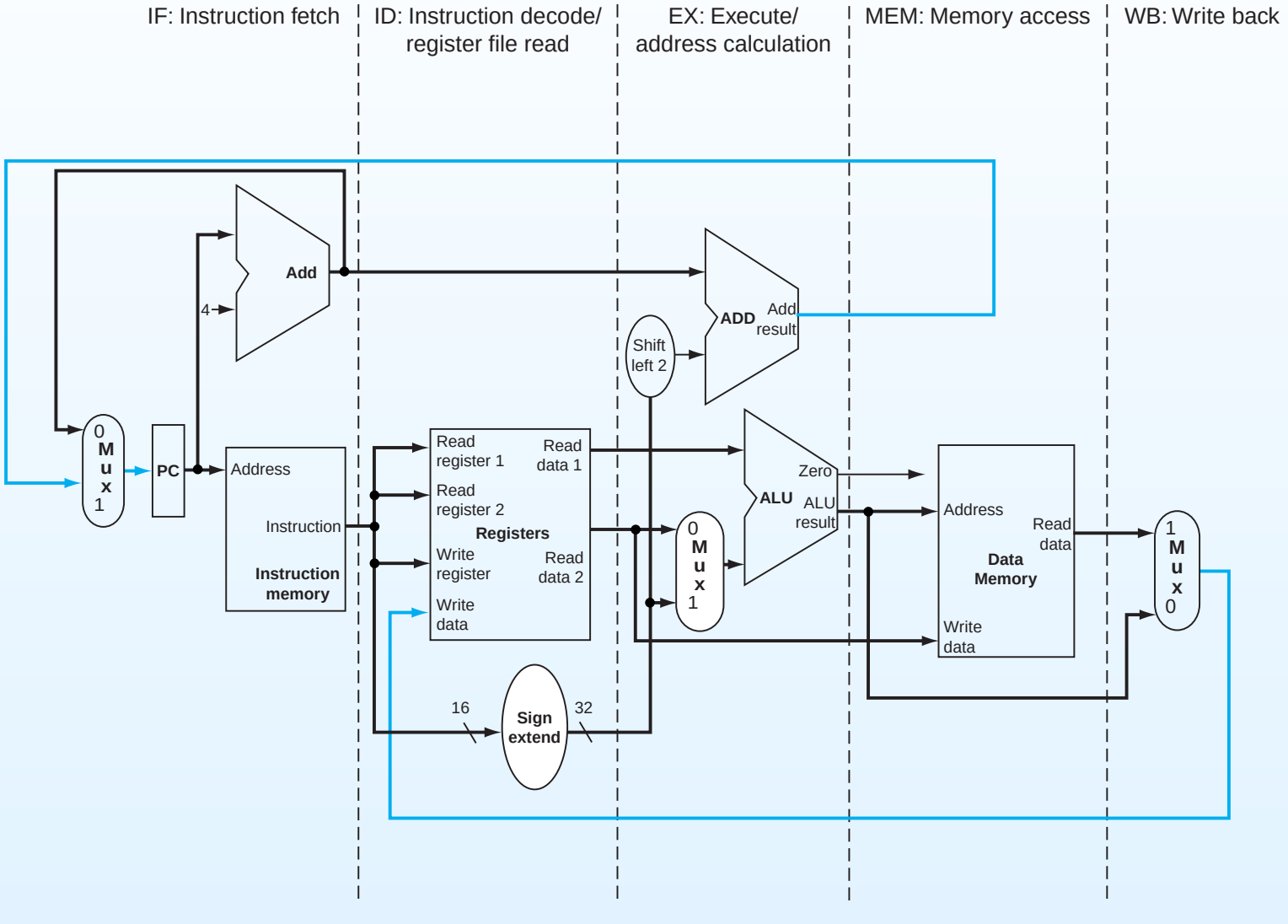
Ciclo único vs Pipelining



Ciclo único vs Pipelining

- Ciclo único
  - Ciclo de relógio determinado pela instrução mais lenta
- Pipelining
  - Ciclo de relógio determinado pela fase mais lenta
  - Tempo de execução por instrução é o mesmo
  - No entanto, nº instruções por segundo aumenta
    - Executando instruções em paralelo (↑ `IPC`)
    - Diminuindo ciclo de relógio (↑ `frequência`)
  - Depois de superada a latência (nº ciclos para encher pipeline)
    - 1 instrução por ciclo (IPC = 1)

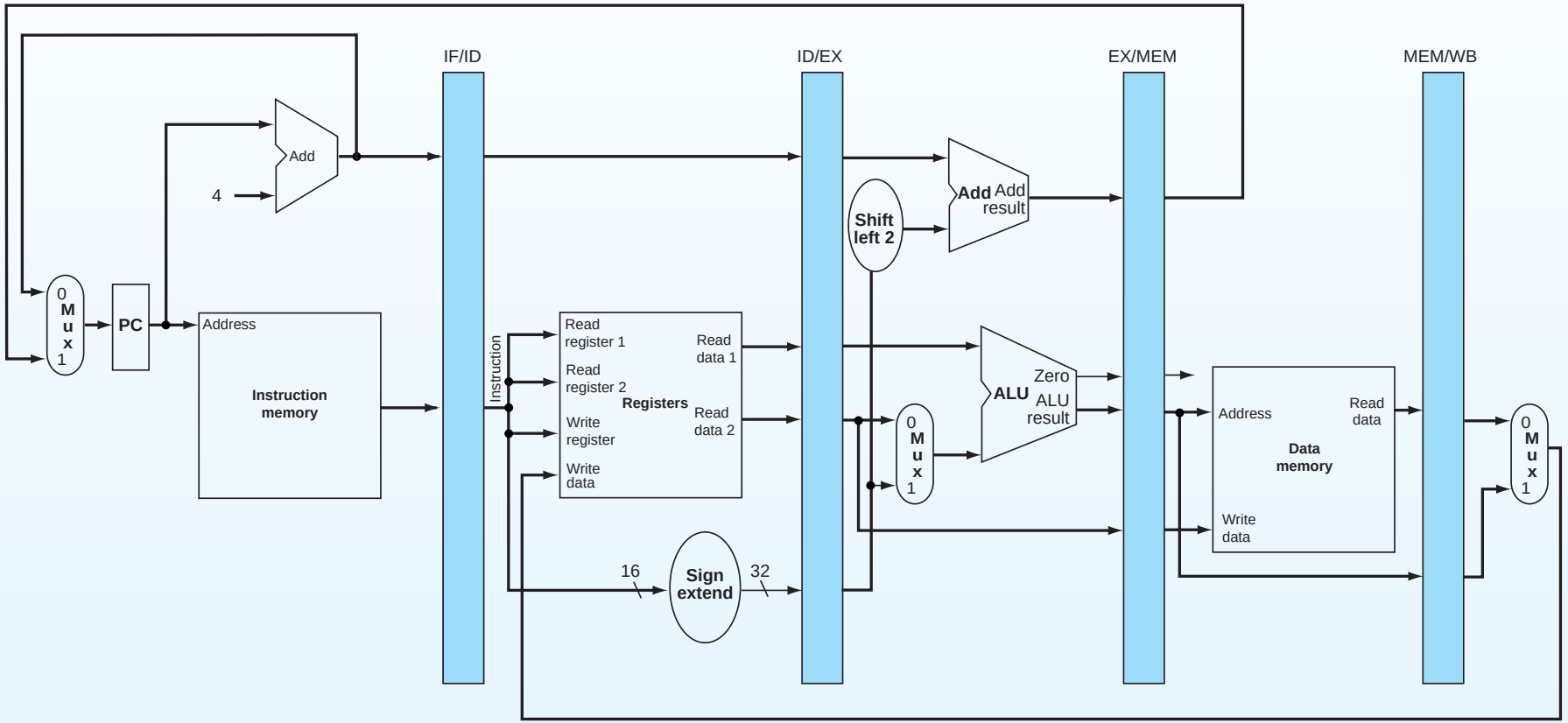
Divisão dos componentes em fases



Implementação

- No mesmo ciclo de relógio
  - Instruções em fases distintas
  - Partilham fluxo de informação
- Necessário armazenar informação intermédia
- Adicionar registos entre fases do pipeline
  - **IF/ID** informação que passa de IF para ID
  - **ID/EX** informação que passa de ID para EX
  - **EX/MEM** informação que passa de EX para MEM
  - **MEM/WB** informação que passa de MEM para WB

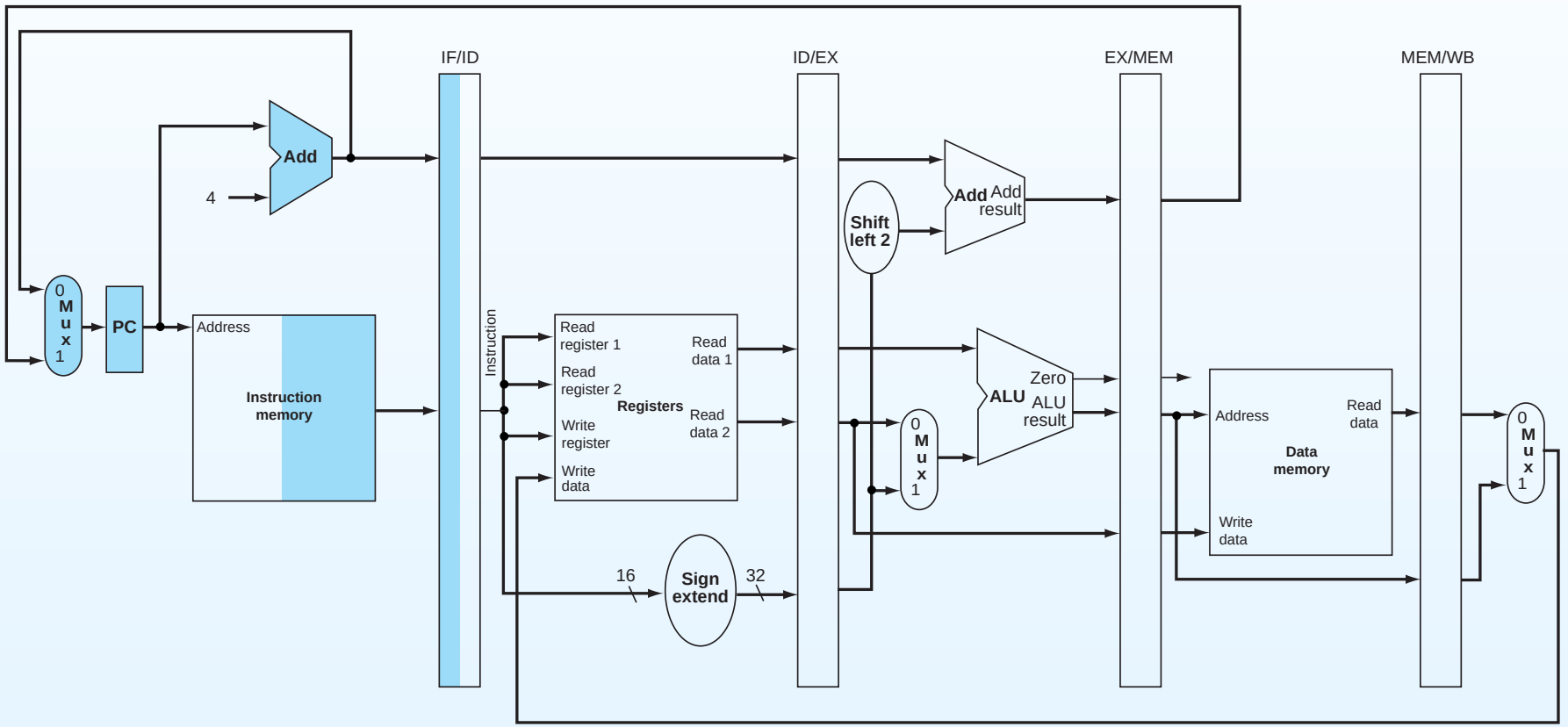
Registos do pipeline



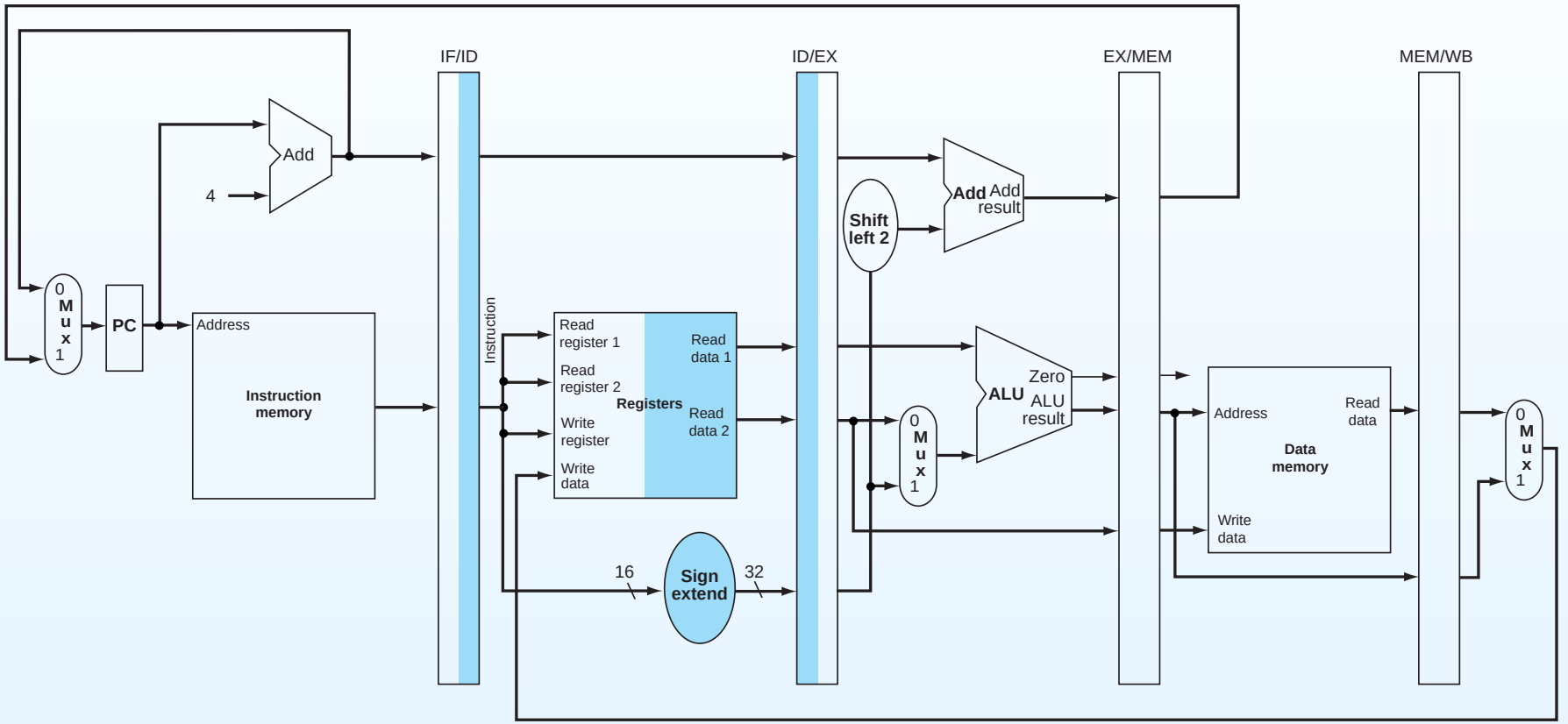
Exemplo do fluxo de informação

- Instrução de acesso à memória ($lw\$1, offset(\$2)$)
  - Activa as 5 fases
- Componentes nos próximos esquemas
  - Lado direito realçado $\rightarrow$ leitura
  - Lado esquerdo realçado $\rightarrow$ escrita

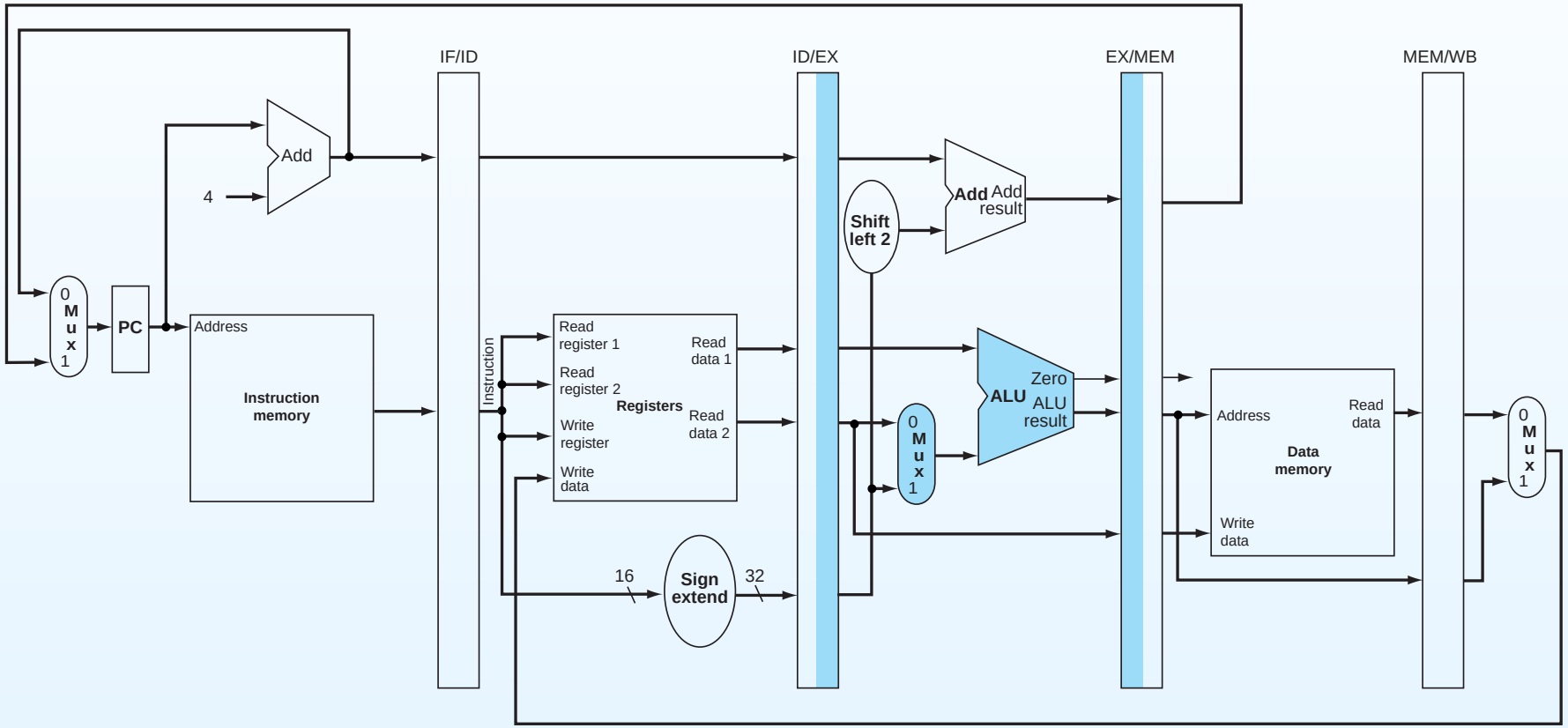
Instruction Fetch - `lw $1,offset($2)`



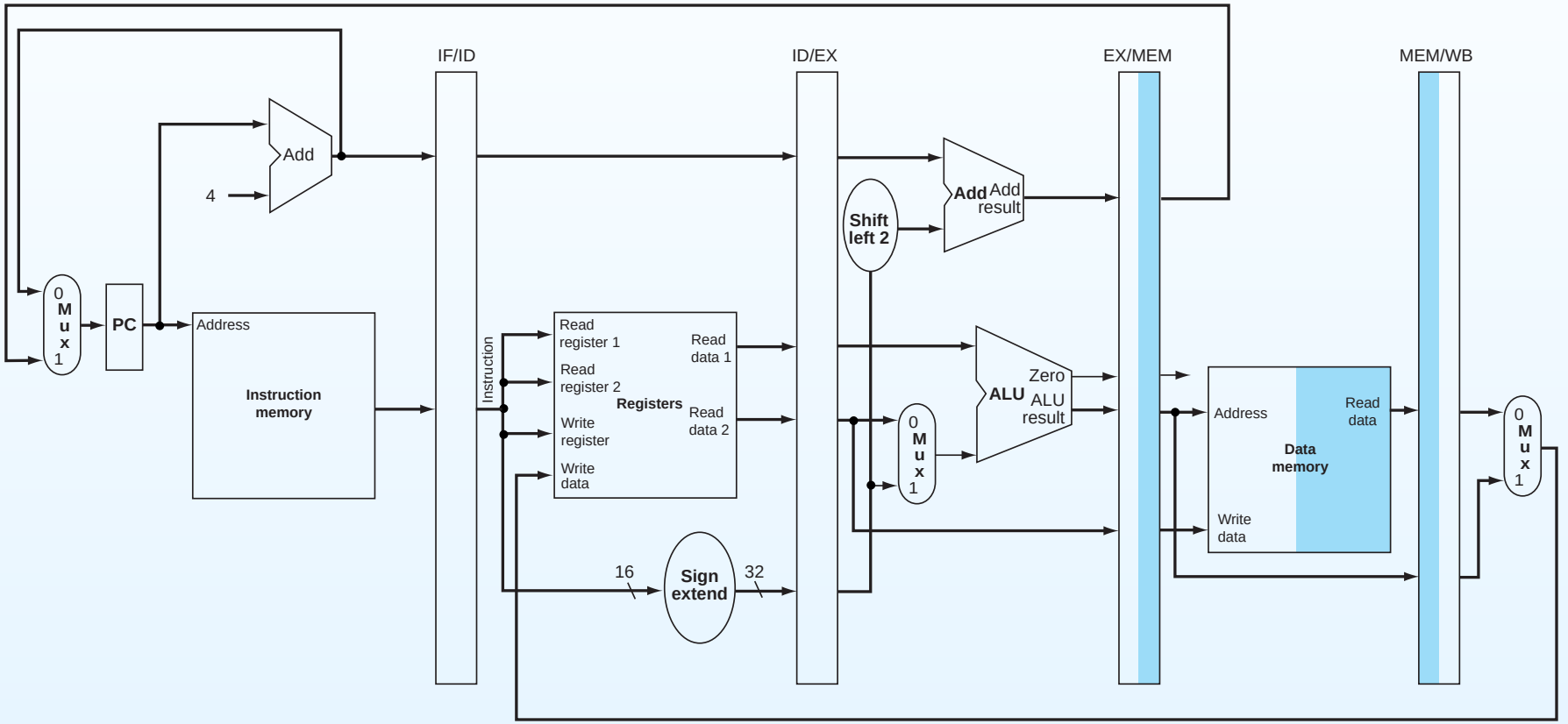
Instruction Decode - `lw $1,offset($2)`



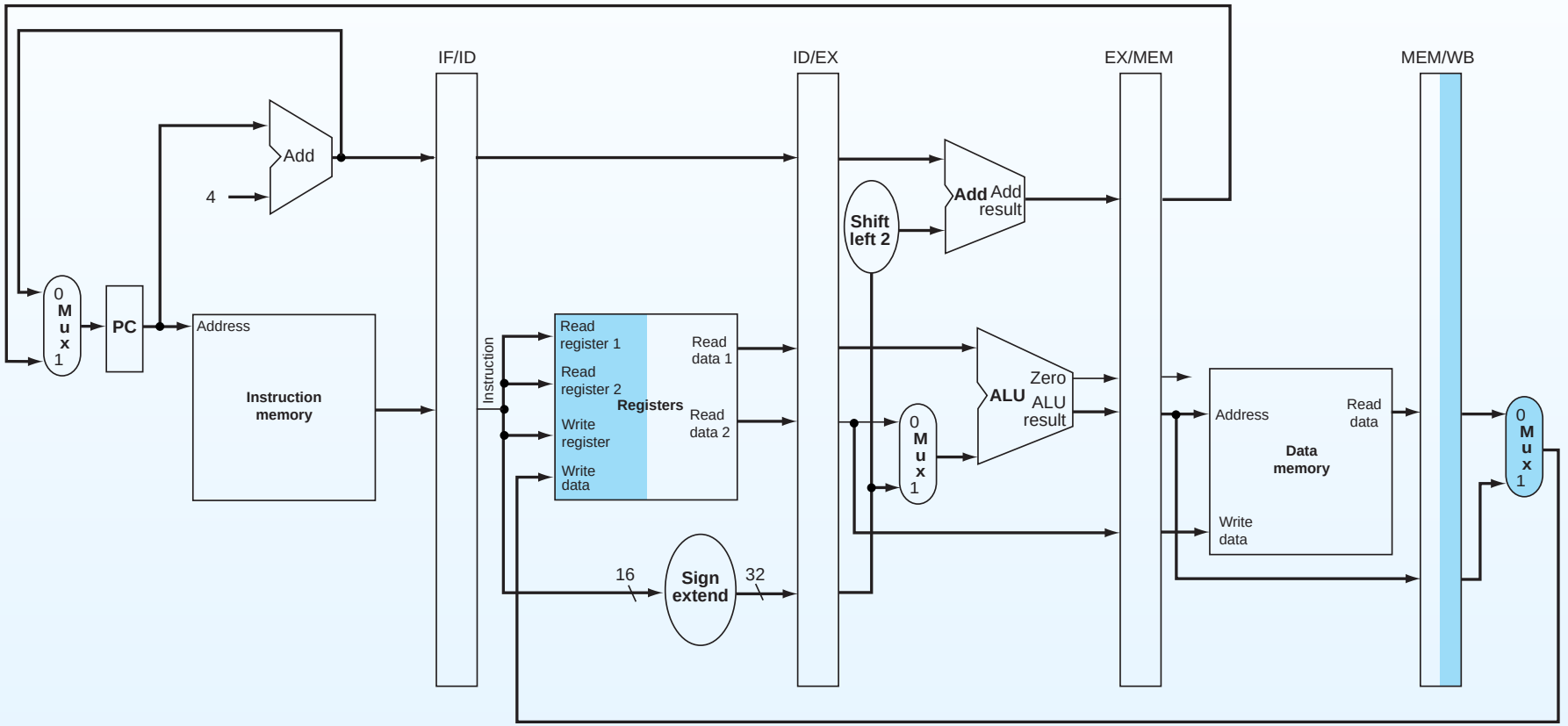
Execute - `lw $1,offset($2)`



Memory Access - `lw $1,offset($2)`



Write Back - `lw $1,offset($2)`

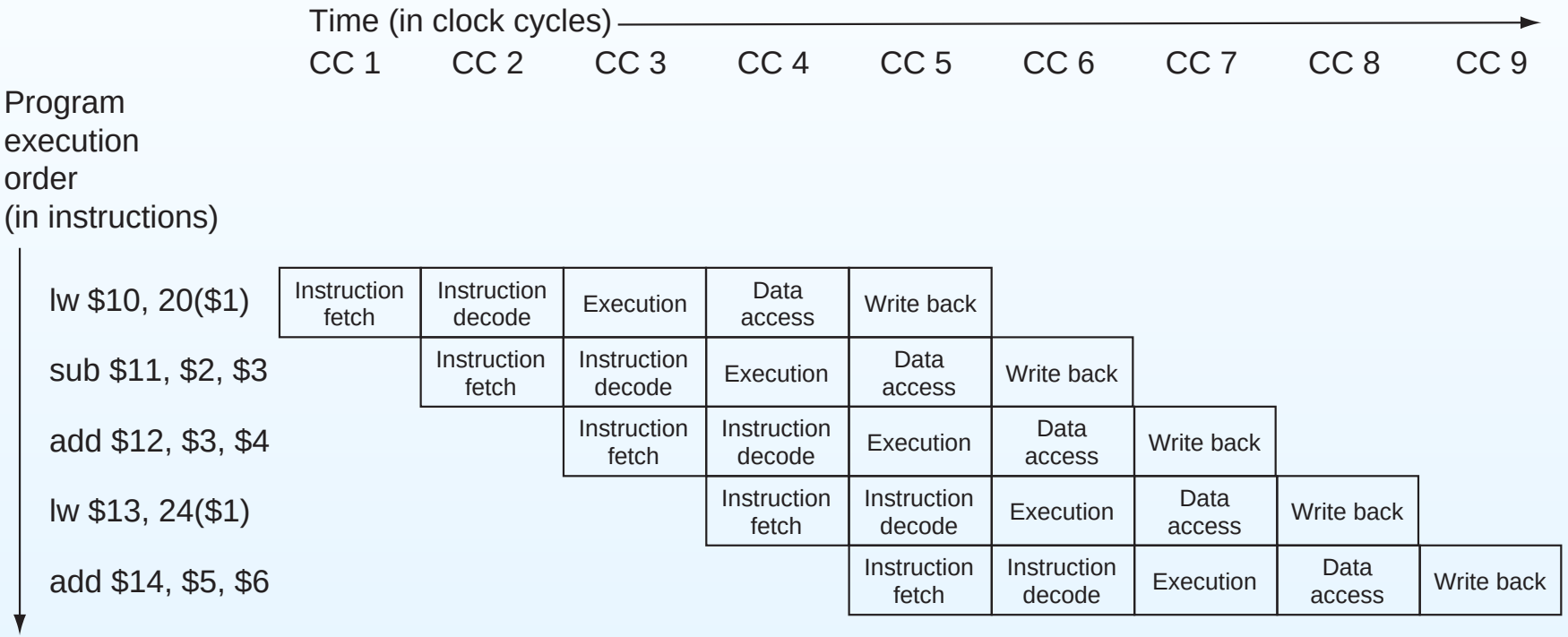


Fluxo de informação - sequência de instruções

```
lw $10,20($1)
sub $11,$2,$3
add $12,$3,$4
lw $13,24($1)
add $14,$5,$6
```

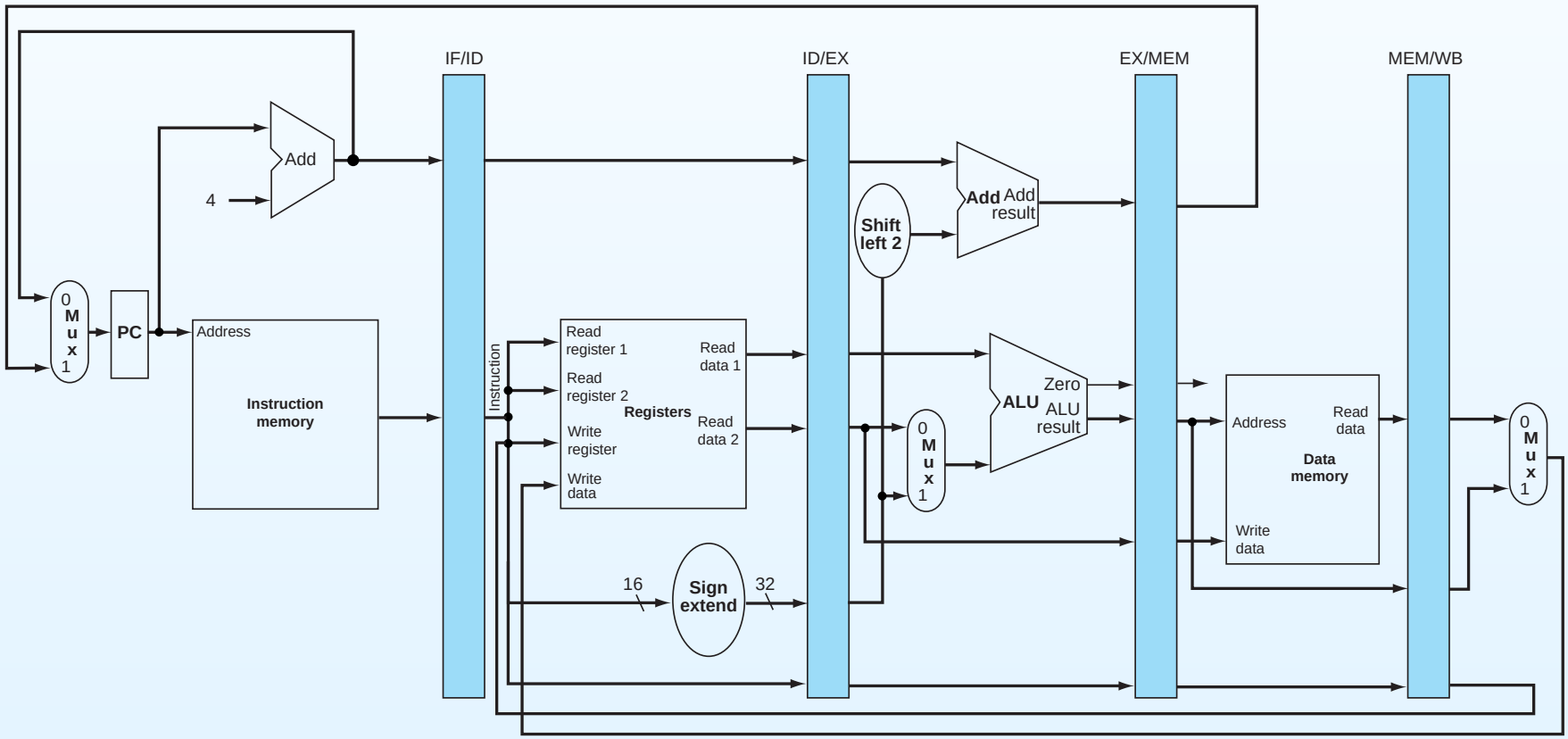
- Execução de instruções em paralelo
- Superada a latência do pipeline
  - Todos os componentes activos num ciclo de relógio
  - IPC = 1

Fluxo de informação - sequência de instruções



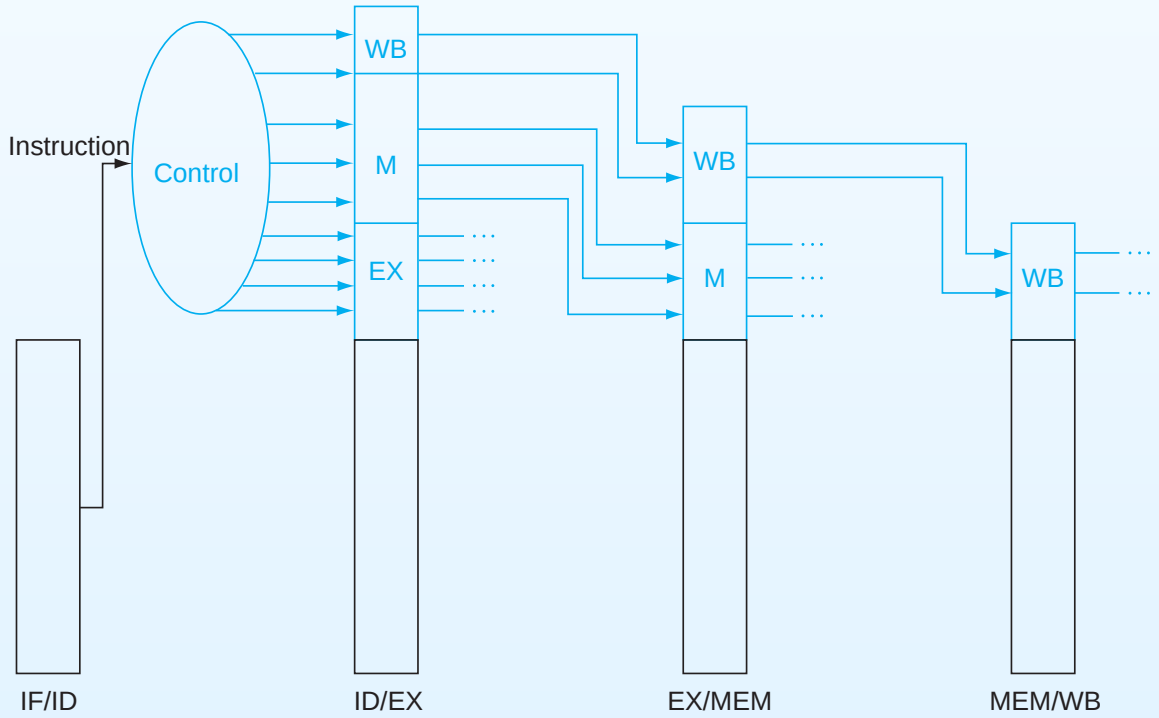
Fluxo de informação - sequência de instruções (CC5)

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4, $11	sub $11, $2, $3	lw$10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write back

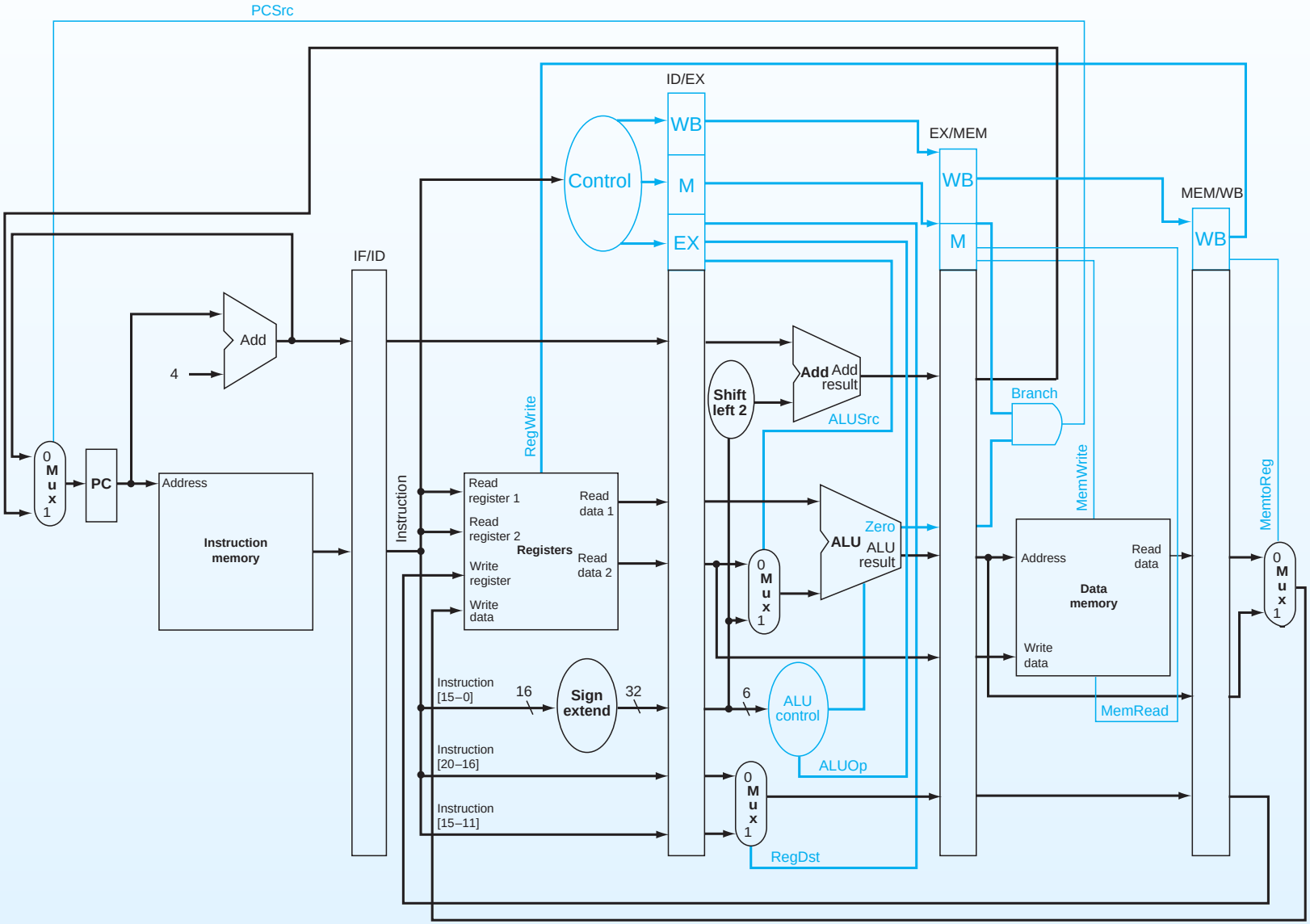


Controlo do fluxo de informação

- **EX** assinalar `RegDest, ALUop, ALUSrc`
- **MEM** assinalar `PCSrc, MemRead, MemWrite`
- **WB** assinalar `MemtoReg, RegWrite`



Controlo do fluxo de informação



Problemas em pipelining

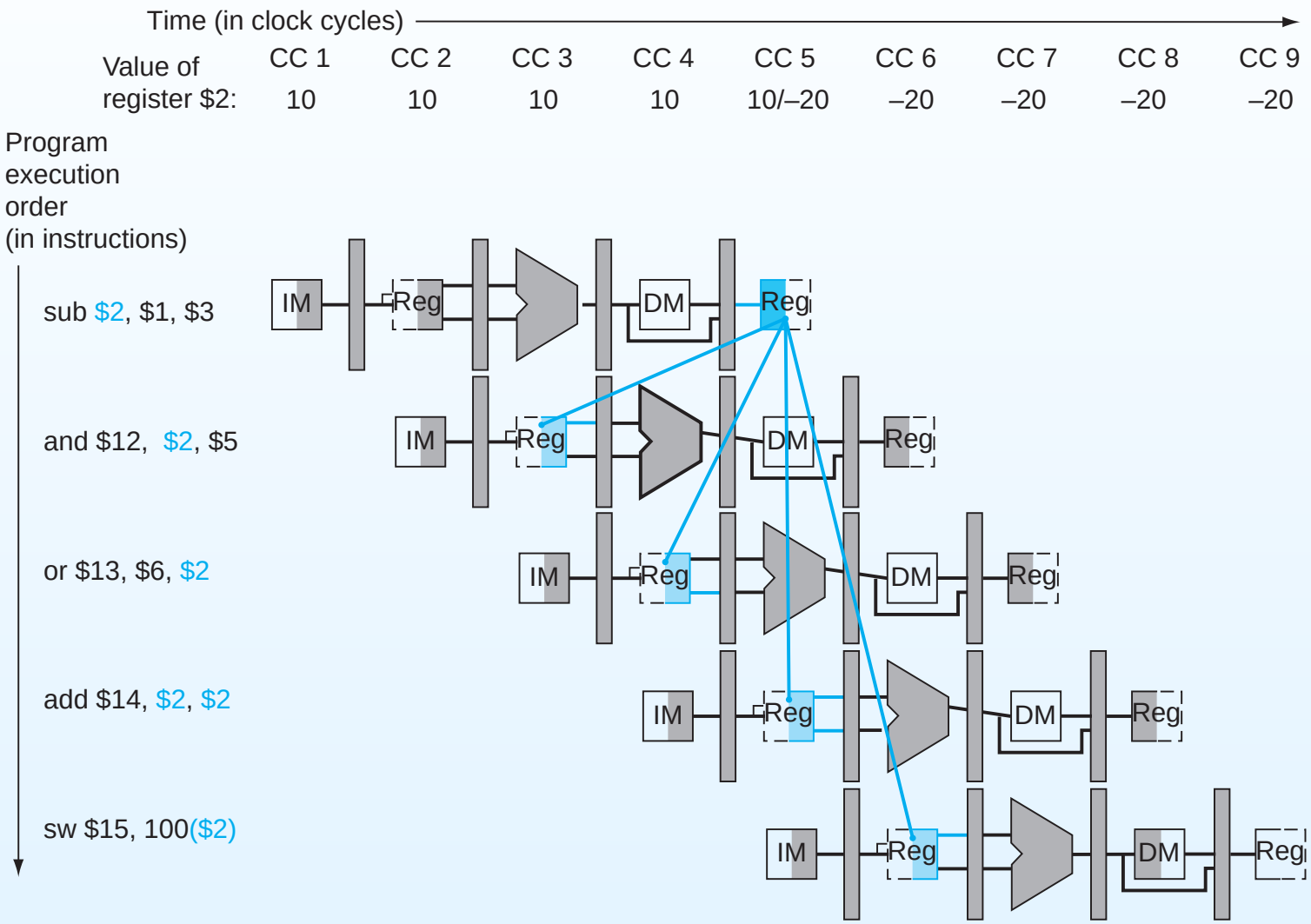
- Objectivo é manter pipeline cheio (IPC = 1)
- Mas por vezes a próxima instrução não pode executar no próximo ciclo de relógio
- Problemas
  - Dependências de dados entre instruções sequenciais
  - Acesso à memória tem latência elevada (`lw`, `sw`)
  - Instruções de salto alteram fluxo de execução
- Não temos problemas estruturais no nosso pipeline
  - Não há partilha de hardware na mesma fase
  - Separamos memória de dados da memória de instruções

Dependências entre instruções

```
sub $2,$1,$3
and $1,$2,$5
or $13,$6,$2
add $14,$2,$2
sw $15,100($2)
```

- Valor de $2 em and é necessário antes de EX
- Só é actualizado na fase WB por sub!
  - Apenas no 5º ciclo de relógio
- Mesmo problema para or e add

Dependências entre instruções



Dependências entre instruções

- Solução 1
  - Parar pipeline até valor de $2 ser actualizado em WB
- Como executar código sem parar pipeline?
- Análise atenta do esquema anterior revela
  - Valor de $2 produzido no CC 3
  - Necessário valor de $2 em CC 4 e CC 5
- Solução 2
  - Propagar valores mal são conhecidos (forwarding)

Forwarding

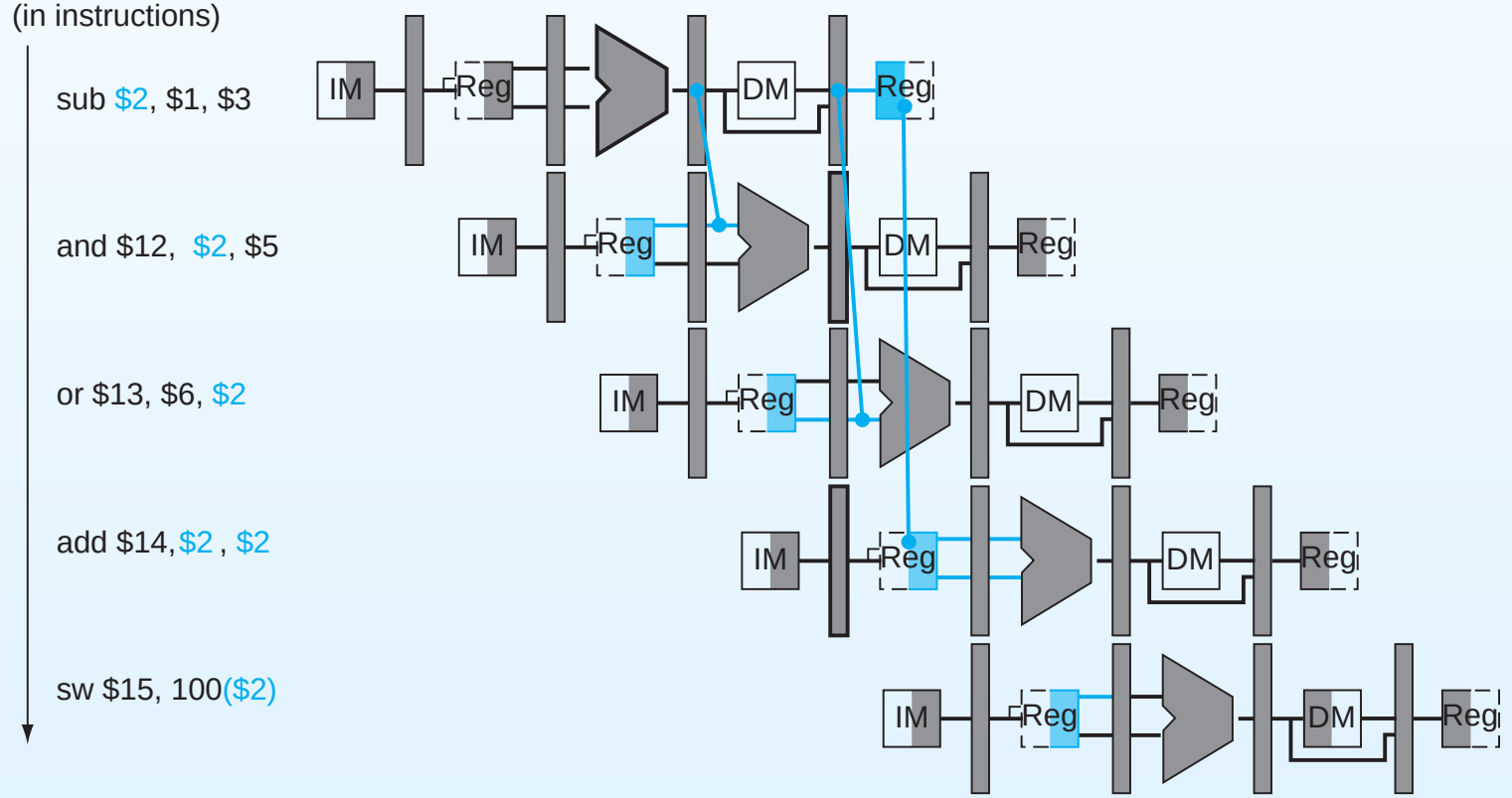
- Detectar dependência em *run time*
  - EX/MEM.RegisterRd = ID/EX.RegisterRs
  - EX/MEM.RegisterRd = ID/EX.RegisterRt
  - MEM/WB.RegisterRd = ID/EX.RegisterRs
  - MEM/WB.RegisterRd = ID/EX.RegisterRt
- Propagar valor
  - Valor é propagado mal é conhecido (fim de EX)
  - Input ALU passa a vir de qualquer registo do pipeline
  - Sem paragens no pipeline
  - Exige hardware adicional

Forwarding - Propagação de valores

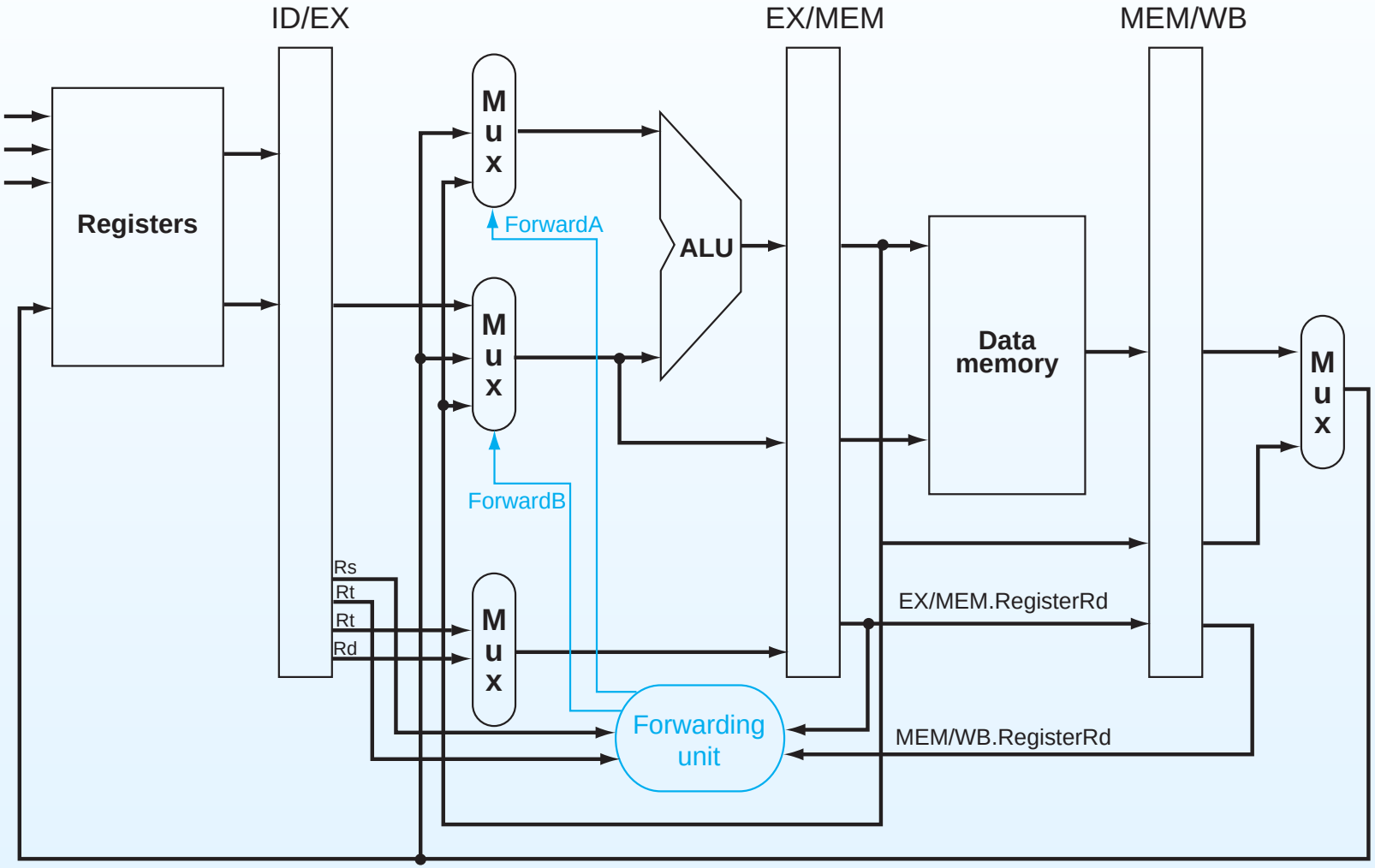
Time (in clock cycles) →

	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
Value of register $2:	10	10	10	10	10/–20	–20	–20	–20	–20
Value of EX/MEM:	X	X	X	–20	X	X	X	X	X
Value of MEM/WB:	X	X	X	X	–20	X	X	X	X

Program execution order (in instructions)



Forwarding - Hardware adicional

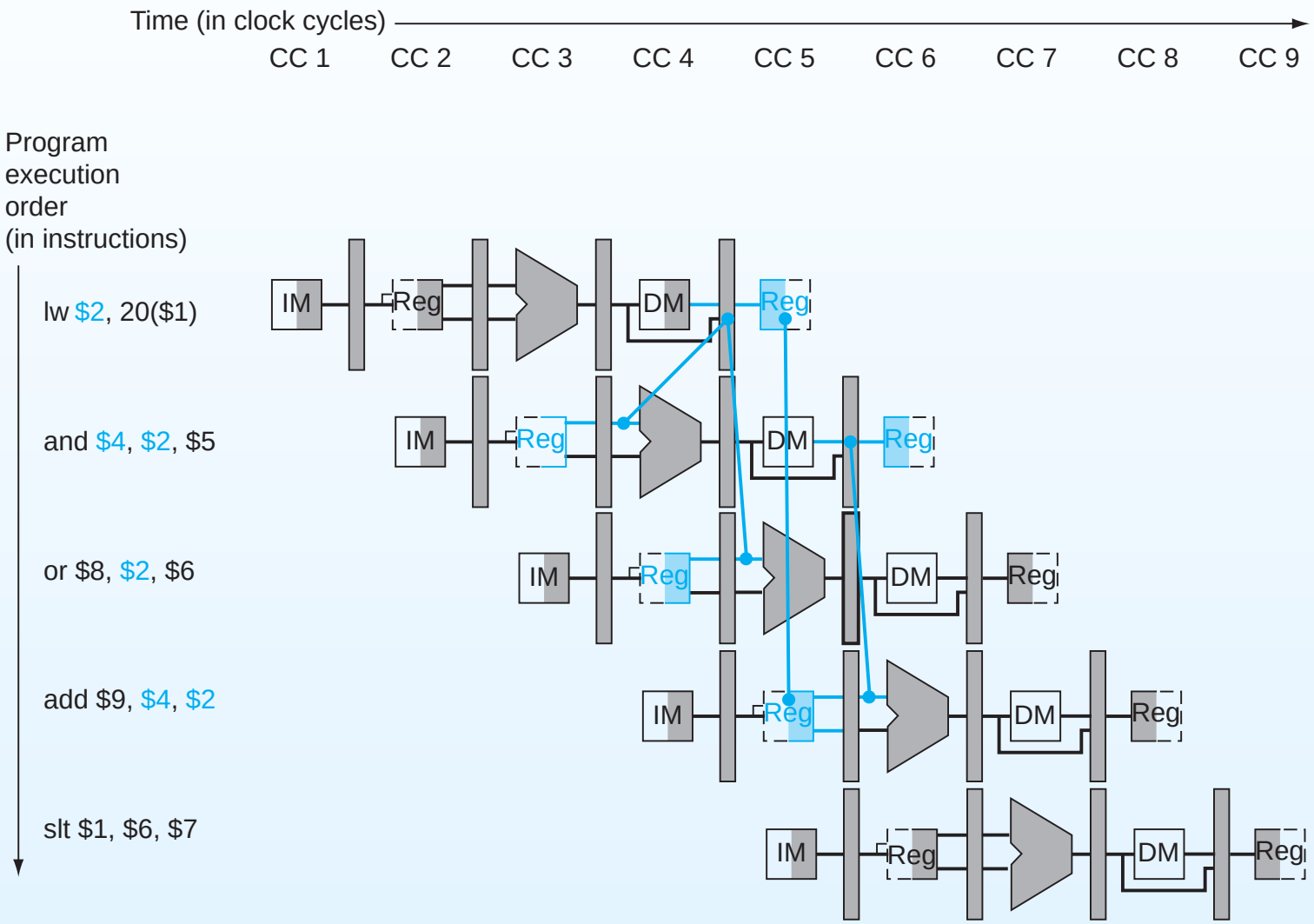


Latência no acesso à memória

```
lw $2,20($1)
and $4,$2,$5
or $8,$2,$6
add $9,$4,$2
slt $1,$6,$7
```

- Acesso à memória tem latência elevada
- and necessita valor de $2 antes de EX
- Valor de $2 só é conhecido depois de MEM
- Forwarding não resolve problema!

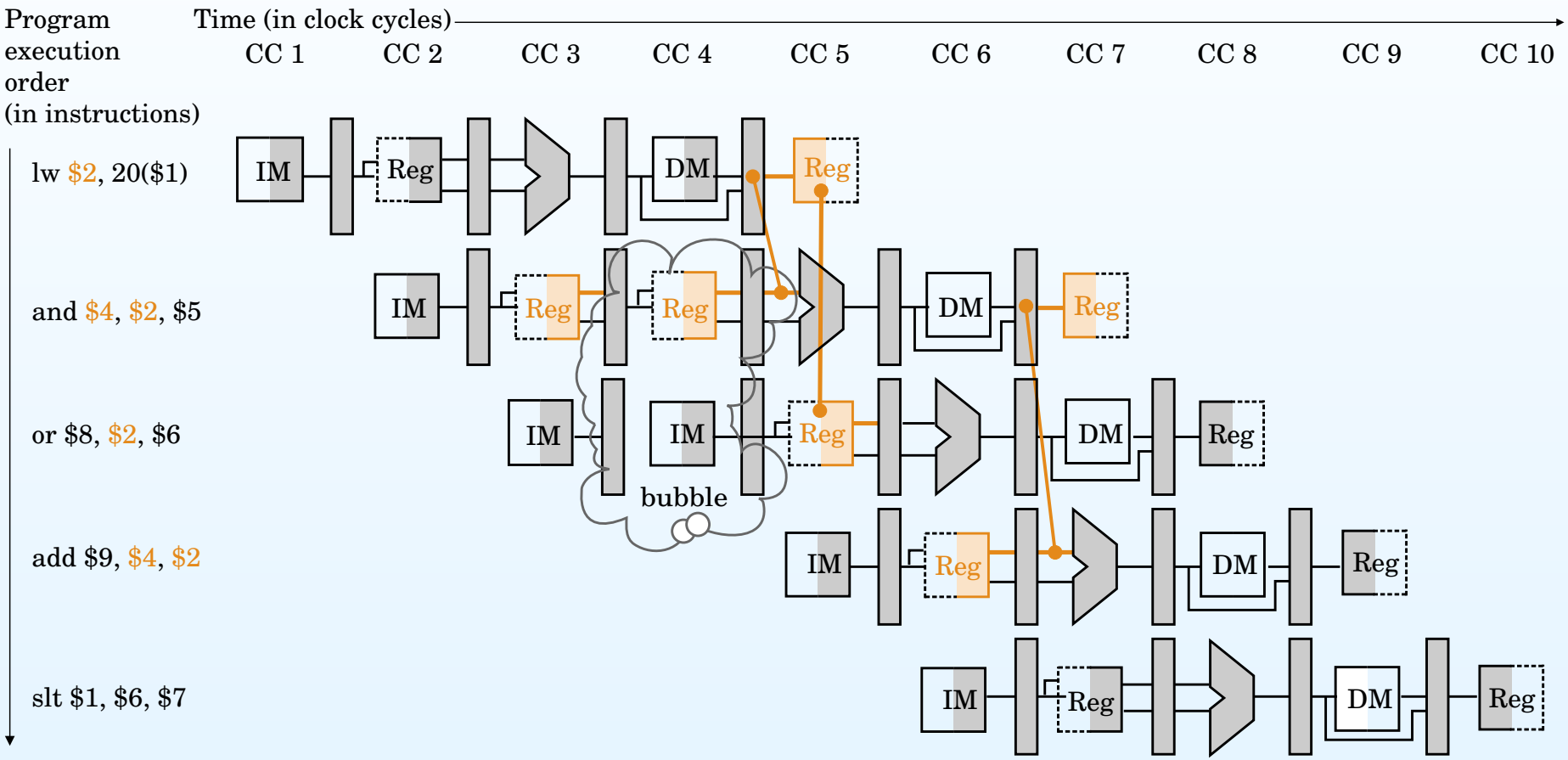
Latência no acesso à memória



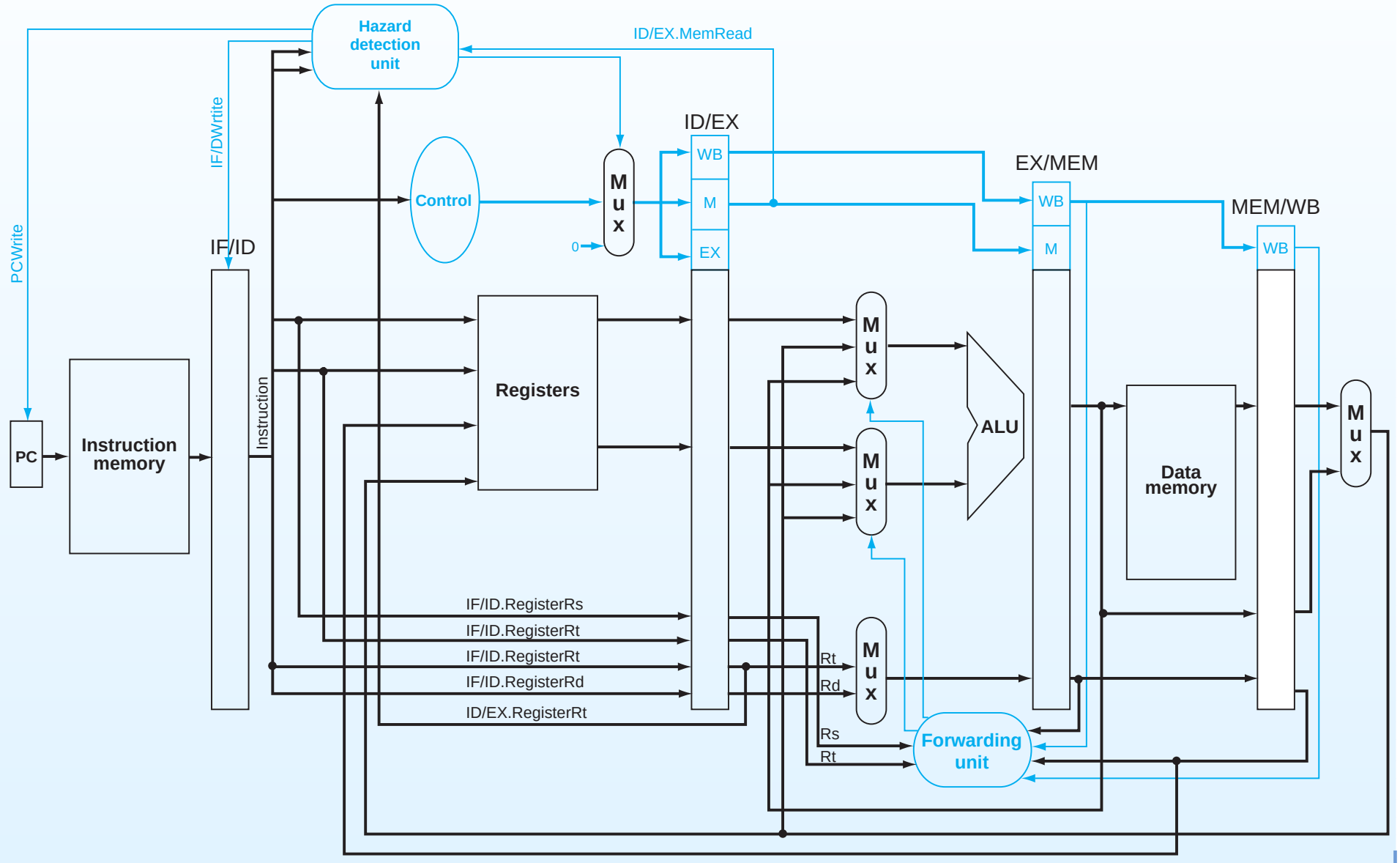
Latência no acesso à memória

- Soluções
  - Compilador
  - Hardware
- Compilador
  - Reordena instruções
  - Instrução seguinte a `lw` não depende desta
- Hardware
  - Detectar dependência em *run time*
  - Inserir estados de espera no pipeline

Estados de espera



Estados de espera - hardware adicional

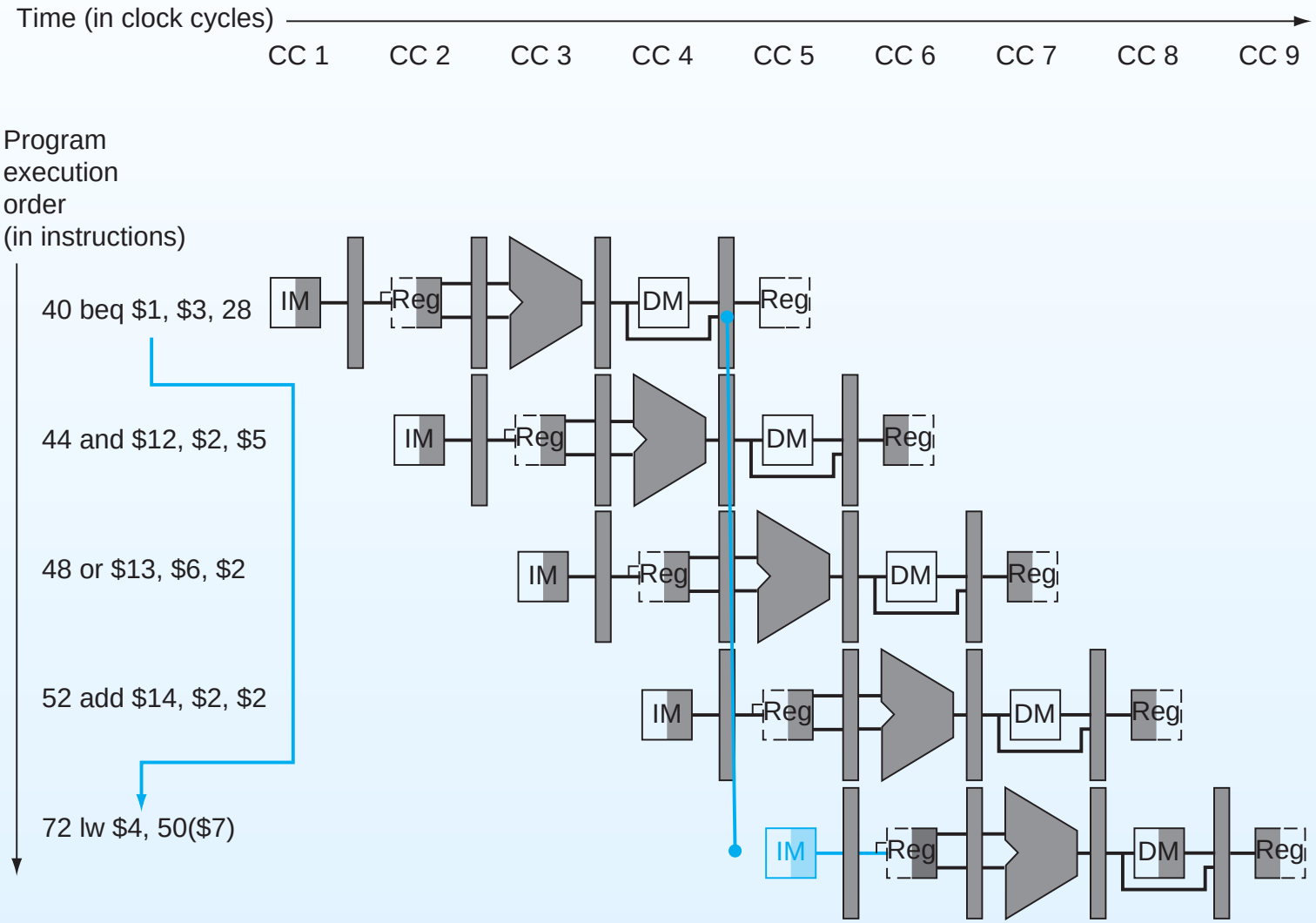


Instruções de salto

```
beq $1,$3,28
and $12,$2,$5
or $13,$6,$2
add $14,$2,$2
lw $4,50($7)
```

- IF ocorre em cada ciclo de relógio
- E se a instrução a ir buscar depender de um salto condicional?
  - Só conhecemos resultado do salto em MEM!

Instruções de salto



Instruções de salto

- Soluções
  - Assumir que salto não é efectuado
  - Reduzir latência da decisão de salto
  - Branch delay slot (compilador)
  - Branch predication

- Nas próximas aulas
  - Previsão dinâmica de saltos
  - Execução especulativa

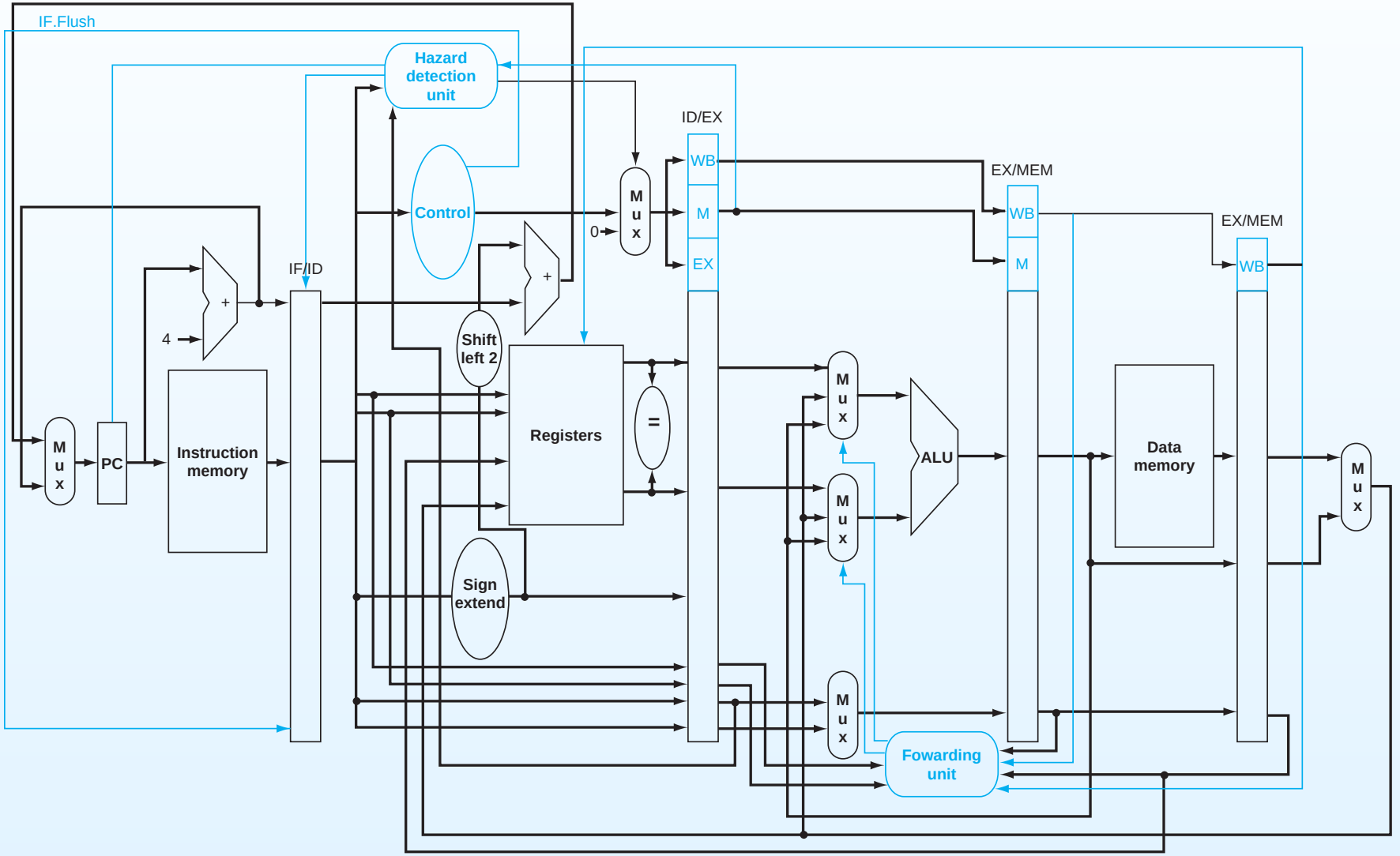
Assumir que salto não é efectuado

- Parar pipeline até conhecer resultado do salto é ineficiente
- Assume-se que salto nunca é efectuado
  - Execução sequencial das instruções
- Se salto for efectuado
  - Descartar instruções em IF, ID e EX
  - IF da instrução correcta
- Se saltos não forem realizados 50% das vezes
  - Optimização de 50%

Reduzir latência da decisão de salto

- Antecipar decisão de salto do MEM para ID
  - Apenas 1 instrução descartada
- Calcular endereço do salto
  - Mover incrementador PC de MEM para ID
    - PC e endereço conhecidos em ID
- Decisão do salto (mais difícil)
  - Mover decisão para ID implica hardware adicional
    - Forwarding e detecção de dependências
  - Descodificar, comparar e propagar valores no mesmo ciclo

Antecipar decisão de salto - Alterações no hardware



Branch delay slot

- Antecipando a decisão do salto para ID
  - Apenas 1 instrução em dúvida
- Compilador tenta introduzir instrução útil nesse slot
  - Instrução independente da decisão de salto
- Limitações
  - Restrições impostas pelas instruções do programa
  - Dificuldade em prever resultado do salto durante a compilação
- Solução é eficaz para pipeline com 5 estágios
  - Pipelines mais profundos $\rightarrow$ branch delay slot maior
  - Actualmente técnica é combinada com previsão dinâmica de saltos por hardware

Branch predication

C	MIPS		Branch predication
`if(i == j)`		`bne $1,$2,ELSE`	`(p) add $3,$4,$5`
`  f = g + h;`		`add $3,$4,$5`	`(~p) sub $3,$3,$1`
`else`		`j EXIT`	
`  f = f - i;`	`ELSE:`	`sub $3,$3,$1`	
	`EXIT:`		

- Eliminar saltos condicionais com lógica de predicados
  - Instruções dependem de predicados (verdadeiro/falso)
- Mais rápido se blocos condicionais são pequenos
- ISA tem de suportar instruções predicadas
  - Mais bits por instrução
- Predicado falso $\rightarrow$ instrução convertida em `nop`

Resumo

- Pipelining melhora o tempo de execução médio das instruções
  - Melhora performance global
  - Não diminui tempo de execução de uma instrução
- ISA influencia implementação do pipeline
  - ISA MIPS desenhada para pipelining
- Problemas em pipelining
  - Dependências de dados entre instruções sequenciais
  - Acesso à memória tem latência elevada (`lw`, `sw`)
  - Instruções de salto alteram fluxo de execução