O Processador

Arquitetura de Computadores

Emilio Francesquini
e.francesquini@ufabc.edu.br

2021.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de Arquitetura de Computadores na UFABC.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- O conteúdo destes slides foi baseado no conteúdo do livro *Computer Organization And Design: The Hardware/Software Interface*, 5th Edition.

Lógica combinacional

Basicão de design lógico

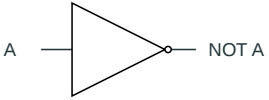
UFABC

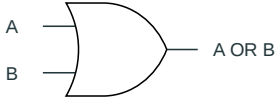
- Informação é codificada em binário
  - Tensão baixa = 0, tensão alta = 1
  - Um fio por bit
  - Dados multi-bit são transmitidos em barramentos multi-fios
- Elemento combinacional
  - Opera nos dados
  - Saída depende da entrada
- Elementos de estado (sequenciais)
  - Armazenam informação

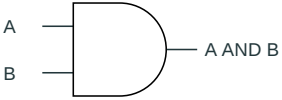
UFABC

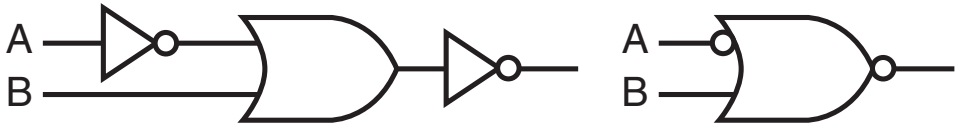
- Nesta aula revisamos brevemente alguns conceitos fundamentais que utilizaremos no restante das aulas.
- Esta disciplina não trata destes assuntos diretamente (outras disciplinas como Circuitos Digitais e Sistemas Digitais servem a este propósito) mas utilizaremos alguns dos conceitos que elas apresentam.
- Informações mais detalhadas podem ser vistas em [PH]: Appendix B.

Portas lógicas





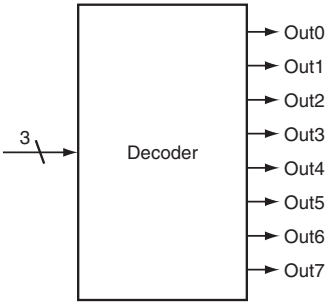




Pergunta

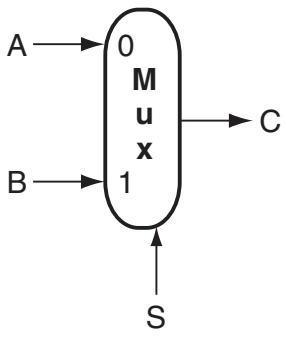
Que expressão lógica os circuitos acima representam?

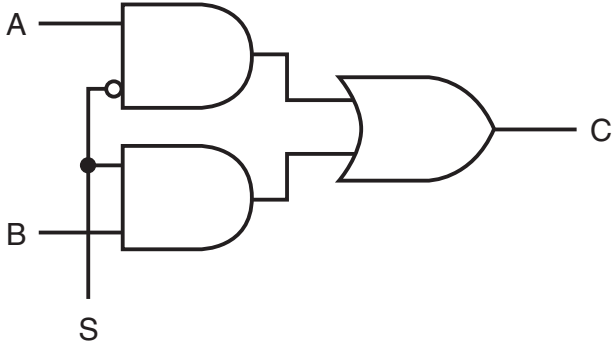
- **Decoders** escolhem uma dentre 2^n saídas baseando-se em uma entrada de n bits.
- Exitem também os **encoders** que fazem o caminho inverso.

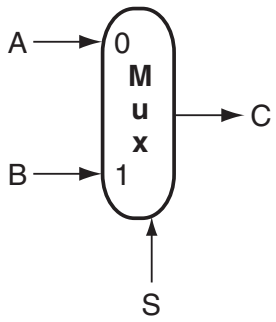


Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- Permitem escolher uma entrada baseando-se num seletor

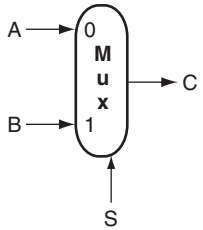






A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

- Como montar um?
- Usamos uma tabela verdade!
- Como transformar em um circuito?

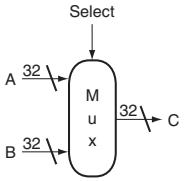


A	B	S	C
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

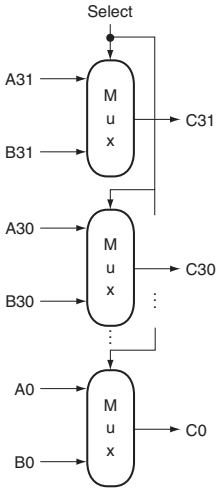
A tabela ao lado é equivalente a:

$$C = \overline{A}.B.S + A.\overline{B}.\overline{S} + A.B.\overline{S} + A.B.S =$$
$$\overline{A}.B.S + A.(\overline{B}.\overline{S} + B.\overline{S} + B.S) =$$
$$\overline{A}.B.S + A.(\overline{S}(\overline{B} + B) + B.S) =$$
$$\overline{A}.B.S + A.(\overline{S} + B.S) =$$
$$\overline{A}.B.S + A.\overline{S} + A.B.S =$$
$$A.\overline{S} + B.(\overline{A}.S + A.S) =$$
$$A.\overline{S} + B.(S.(\overline{A} + A)) =$$
$$A.\overline{S} + B.S$$

Multiplexadores de 32 bits



a. A 32-bit wide 2-to-1 multiplexor

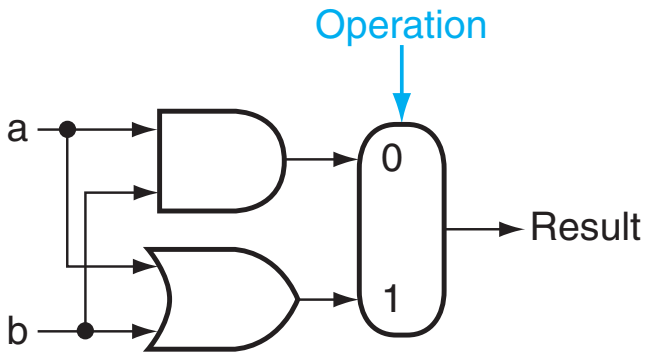


b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

Construindo uma ULA básica de 1 bit

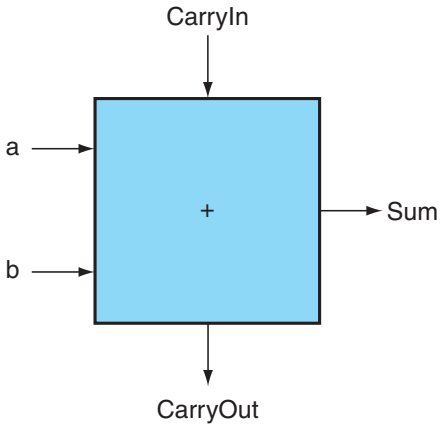
UFABC

- Se queremos chegar até um MIPS, temos várias instruções que precisam ser implementadas
- Vamos começar com as mais básicas
  - AND, OR, ADD, SUB, SLT, BEQ, ...
- Para as duas primeiras já temos as portas lógicas prontas! Vamos começar por elas.



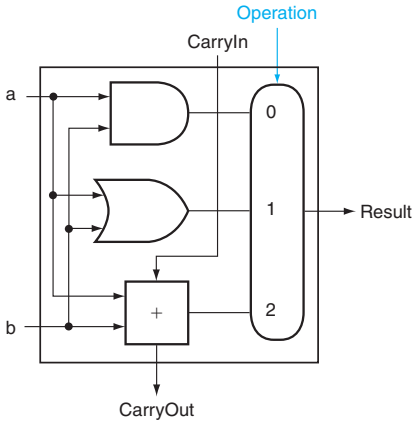
- Vamos adicionar **ADD**...

Um somador de 1 bit



- Um somador de 1 bit pode ser facilmente implementado usando o método que usamos para fazer um multiplexador.
- Vamos integra-lo na nossa ULA.

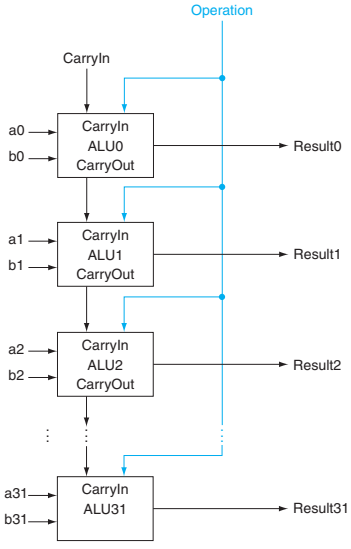
ULA de 1 bit com **AND**, **OR** e **ADD**



Pergunta

Como fazemos para criar uma ULA de 32 bits?

ULA de 32 bits com **AND**, **OR** e **ADD**

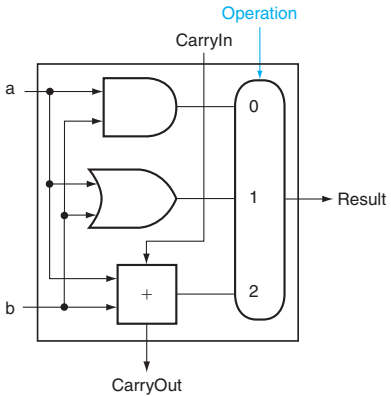


Incluindo a operação **SUB**

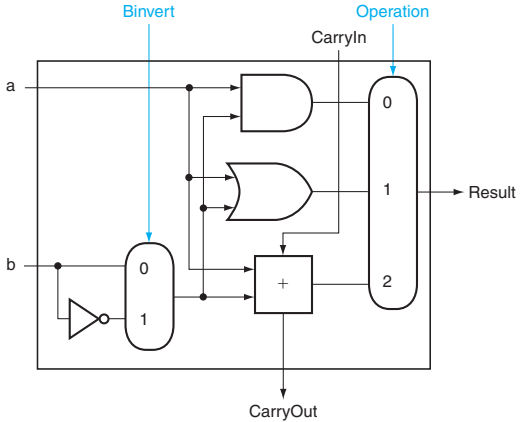
- Nas aulas anteriores vimos que para números em complemento de 2, a subtração é muito parecida com a adição.
- Para calcular $A - B$, invertemos todos os bits de B, somamos 1 ao resultado e em seguida somamos com A.

Note

Conseguimos adaptar o que já temos pronto para fazer isso?



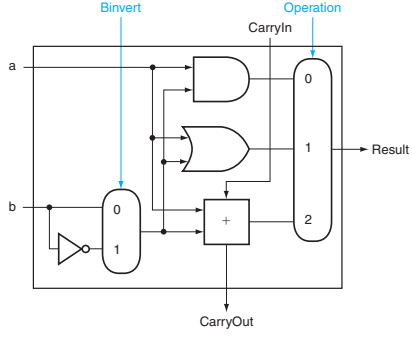
Incluindo a operação **SUB**



- Basta então fornecer os sinais de controle corretamente.

Incluindo a operação NOR

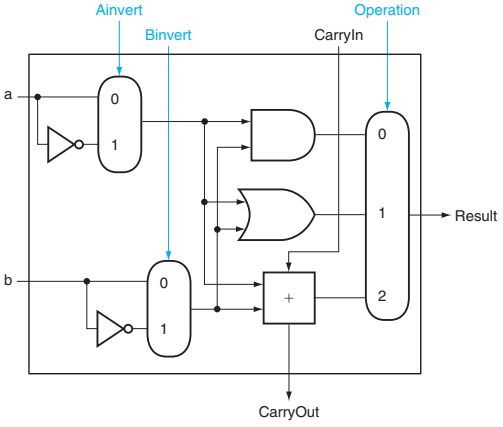
- De maneira semelhante, com pouquíssimas modificações já conseguimos adicionar a operação NOR
  - `NOR A B = NOT(A OR B) = (NOT A) AND (NOT B)`



Pergunta

Como alteramos o circuito acima para fazer isso?

Incluindo a operação **NOR**



Pergunta

Quais são os sinais de controle para cada uma das operações que já implementamos (`AND OR ADD SUB NOR`)?

UFABC

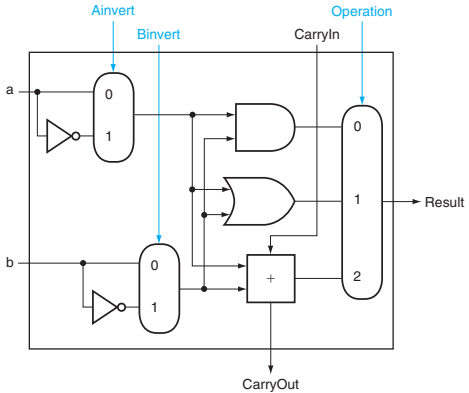
	Operation	Carry in	Ainvert	Binvert
AND	00	_	0	0
OR	01	_	0	0
NOR	00	_	1	1
ADD	10	0	0	0
SUB	10	1	0	1

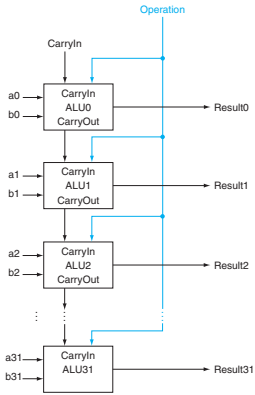
Tip

Conseguimos implementar diretamente mais uma operação só com o hardware que temos? Se sim, qual?

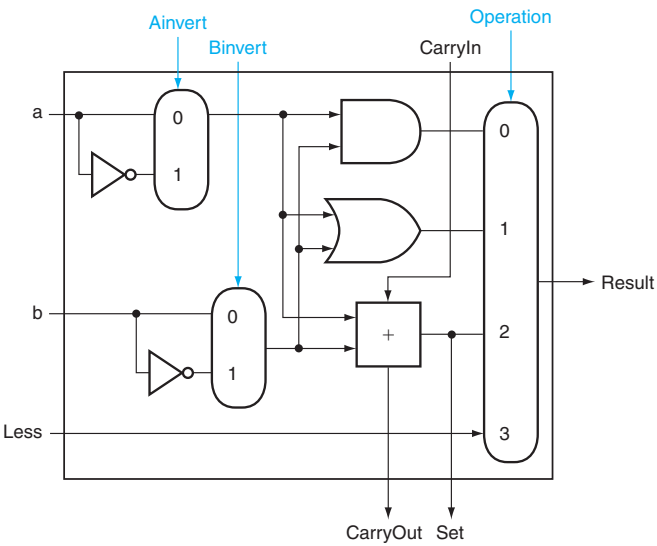
Mas ainda está faltando algo...

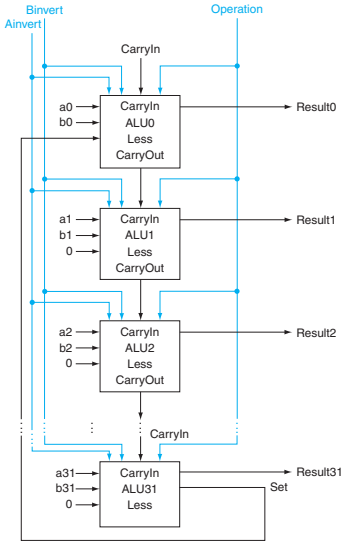
- Para criarmos o nosso processador MIPS, ainda temos 2 operações que são essenciais: `slt` e `beq`
- A ALU tem que ser capaz de verificar se `A < B` e se `A = B`
- O que dá para fazer com que já temos?





Implementando o `slt`





UFABC

- Para saber se $A = B$ podemos fazer $A - B$. Se for 0, então $A = B$. Basta negar o resultado!
- Vamos consertar o circuito! Mas antes vamos olhar novamente a tabela de sinais de controle

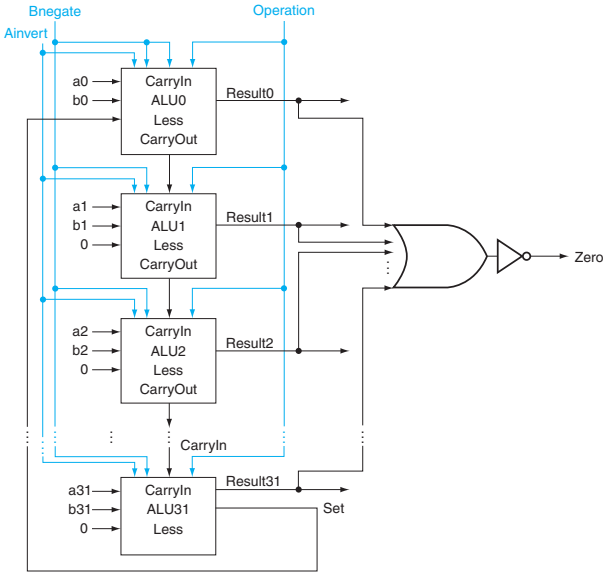
	Operation	Carry in	Ainvert	Binvert
AND	00	_	0	0
OR	01	_	0	0
NOR	01	_	1	1
ADD	10	**0**	0	**0**
SUB	10	**1**	0	**1**
SLT	11	**1**	0	**1**

UFABC

- Podemos juntar o *Carry in* e *Binvert*. Vamos chamar de **Bnegate**

	Operation	Bnegate	Ainvert
AND	00	**0**	0
OR	01	**0**	0
NOR	01	**1**	1
ADD	10	**0**	0
SUB	10	**1**	0
SLT	11	**1**	0

- Voltando ao **beq**...

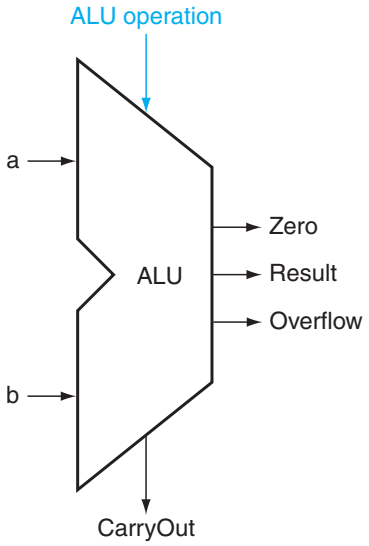


- Criamos uma nova saída da ALU, chamada **`Zero`**
- Quando **`Zero`** for 1, *A* e *B* são iguais.

Note

Quais são os bits de controle para **`beq`**?

Finalmente temos uma ALU



O Processador

UFABC

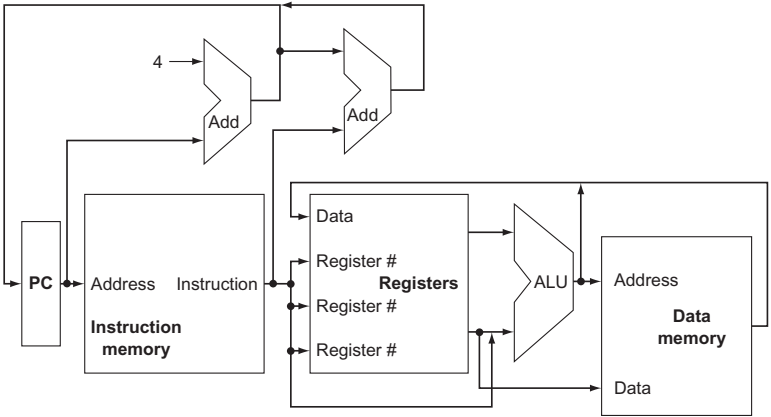
- O desempenho da CPU é influenciado por
  - Número de instruções executadas
    - Determinado pela ISA e pelo compilador
  - CPI e Frequência
    - Determinada pelo hardware
- Vamos estudar duas implementações de MIPS
  - Uma versão simplificada monociclo
  - Uma versão mais realista, com pipelines
- Vamos nos concentrar em um subconjunto das instruções que, apesar de simples, demonstram a maior parte dos detalhes
  - Acesso à memória: `lw`, `sw`
  - Aritimética e lógica: `add`, `sub`, `and`, `or`, `slt`
  - Branches e saltos: `beq`, `j`

Execução de instruções

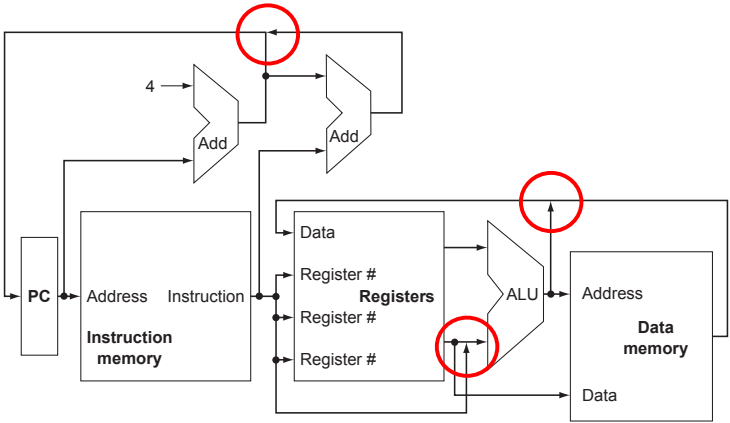
UFABC

- PC $\rightarrow$ memória de instruções, carrega instrução
- Números de registradores $\rightarrow$ banco de registradores, leitura de registradores
- Dependendo to tipo de instrução
  - Usa a ALU para calcular
    - Resultado aritmético
    - Endereço de memória para load/store
    - Endereço para branch
  - Acessa a memória de dados para load/store
  - PC $\leftarrow$ endereço de destino ou PC + 4

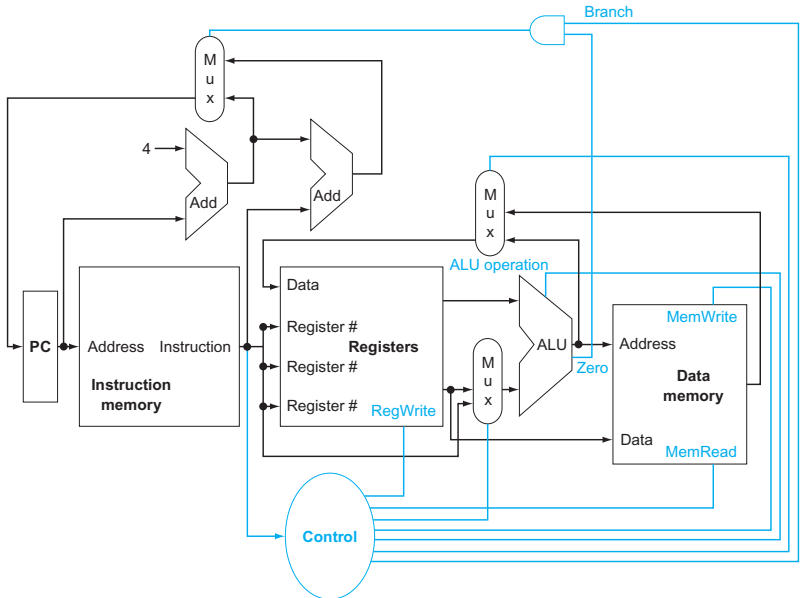
Uma visão geral do processador



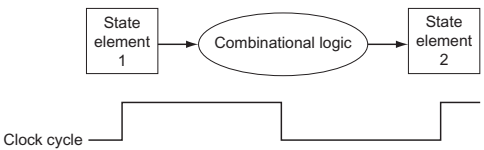
- Elementos combinacionais que operam nos dados
- Elementos sequenciais que armazenam o estado

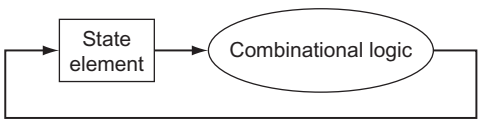


- Não podemos sair juntando fios. Precisamos de multiplexadores para selecionar qual sinal é de interesse.


Branch
M
u
x
4
Add
Add
M
u
x
ALU operation
PC
Address
Instruction
Instruction
memory
Data
Register #
Registers
Register #
Register #
RegWrite
M
u
x
ALU
Zero
MemWrite
Address
Data
memory
Data
MemRead
Control

- Lógica combinacional transforma os dados durante os ciclos de clock
  - Entre os bordas do clock
  - Entrada de elementos de estado, saída para elemento de estado
  - O atraso mais longo determina o período do clock

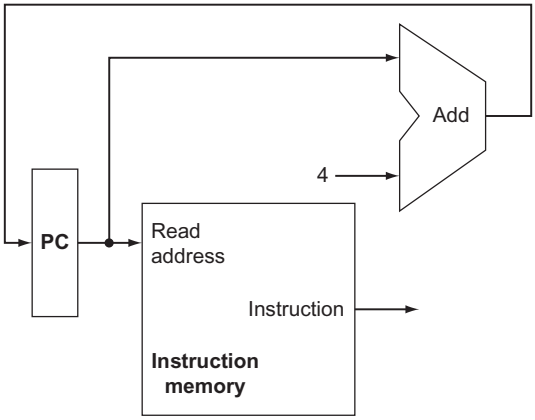




Datapath Monociclo

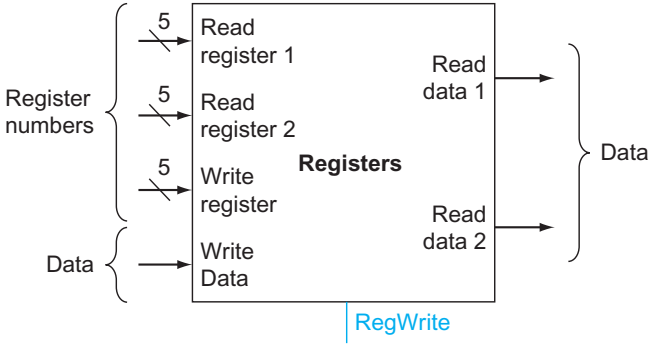
UFABC

- Datapath
  - Elementos que processam os dados e endereços na CPU.
    - Registradores, ALUs, muxes, memórias, ...
- Vamos construir um datapath de MIPS de maneira incremental.
  - Refinaremos pouco a pouco o design geral que vimos.

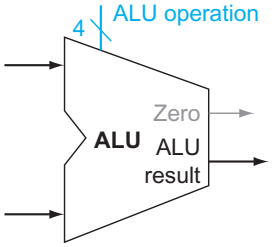


- PC é um registrador de 32 bits.
- ALU soma 4 no PC para pular para a próxima instrução.

- Leem 2 operandos de registradores
- Fazem uma operação lógica/aritmética
- Escrevem o resultado em um registrador

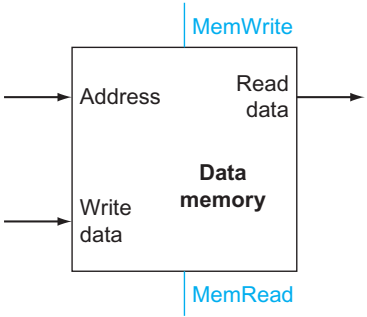


a. Registers

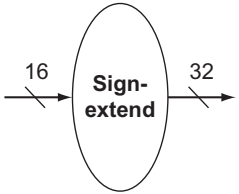


b. ALU

- Lê os operandos dos registradores
- Calcula o endereço usando um offset de 16 bits
  - Usa a ALU, mas faz a extensão de sinal do offset
- Load: lê a memória e atualiza o registrador apropriado
- Store: escreve o valor do registrador na memória

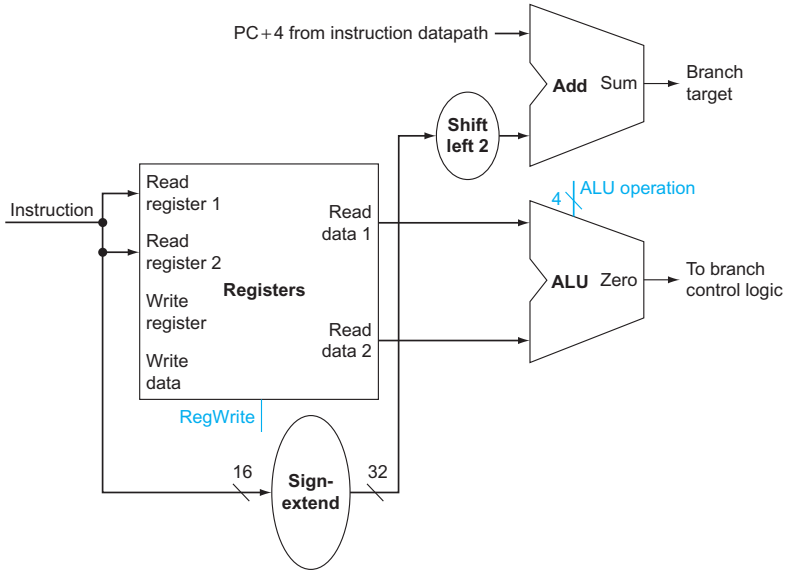


a. Data memory unit



b. Sign extension unit

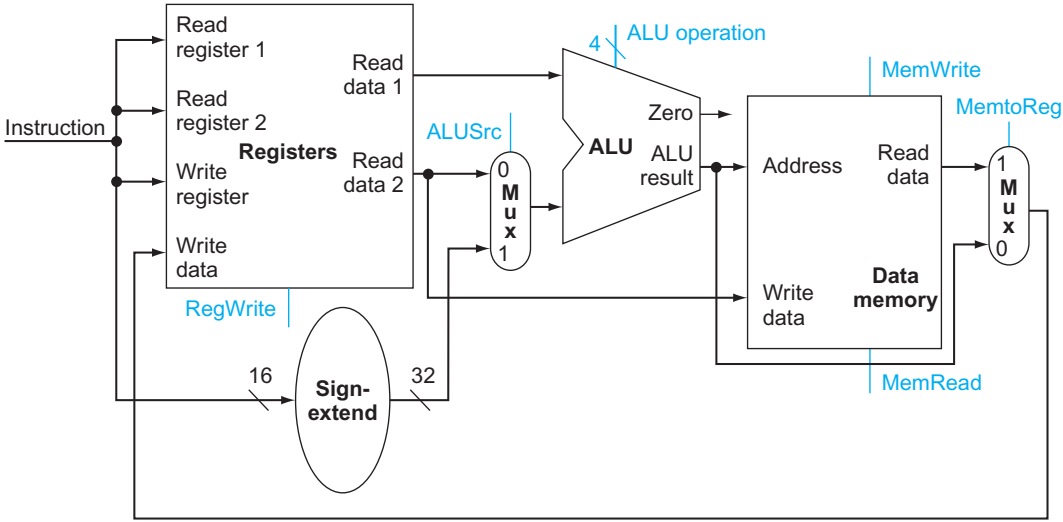
- Lê os operandos dos registradores
- Compara os operandos
  - Usa a ALU, subtrai os operando e usa a saída `Zero`
- Calcula o endereço alvo
  - Estende o deslocamento (com sinal)
  - Faz um shift de 2 (uma palavra)
  - Adiciona 4 ao PC
    - Já feito pelo passo de fetch de instruções


PC + 4 from instruction datapath
Add
Sum
Branch target
Shift left 2
4
ALU operation
Instruction
Read register 1
Read register 2
Write register
Write data
Registers
Read data 1
Read data 2
ALU
Zero
To branch control logic
RegWrite
16
Sign-extend
32

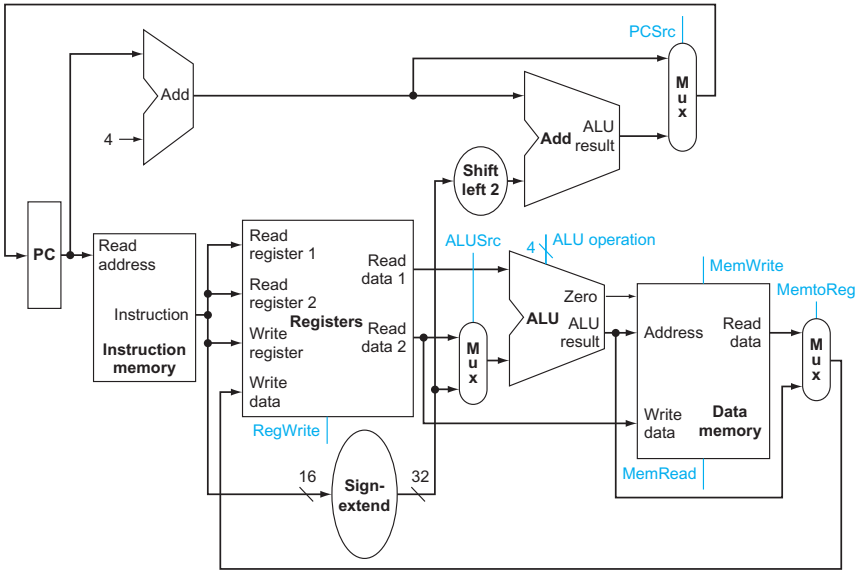
Juntando tudo

UFABC

- O Datapath executa uma instrução por ciclo
  - Cada elemento do datapath só pode fazer uma função por vez
  - Logo precisamos separar as memórias de instruções e de dados
- Emprega-se multiplexadores nos locais onde fontes de dados alternativas são usadas por diferentes instruções.


Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
RegWrite
ALUSrc
0
M
u
x
1
4
ALU operation
ALU
Zero
ALU result
MemWrite
MemtoReg
Address
Read data
Write data
Data memory
MemRead
1
M
u
x
0
16
Sign-extend
32

O Datapath completo



UFABC

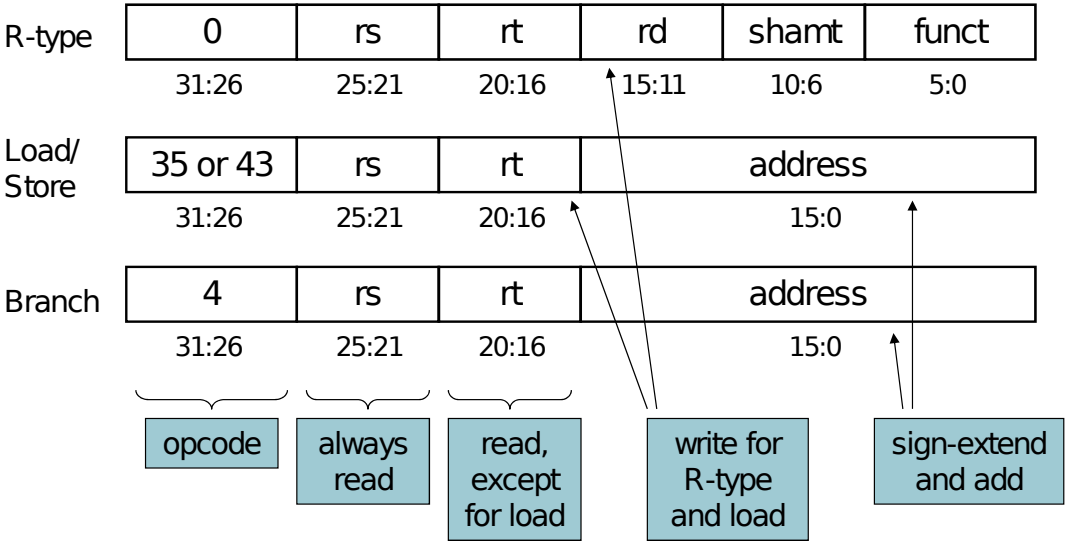
- ALU usada para
  - Load/Store: Função = **ADD**
  - Branch: Funcão = **SUB**
  - R-Type: Função depende do campo `funct`

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

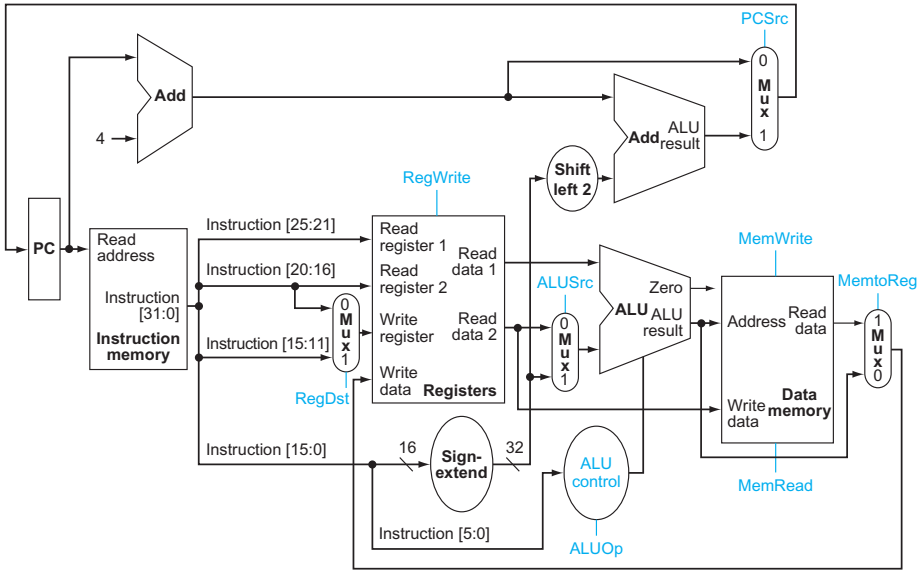
- Assuma que **ALUOp** de 2 bits é derivada do opcode
  - Lógica combinacional é usada par derivar o controle da ALU

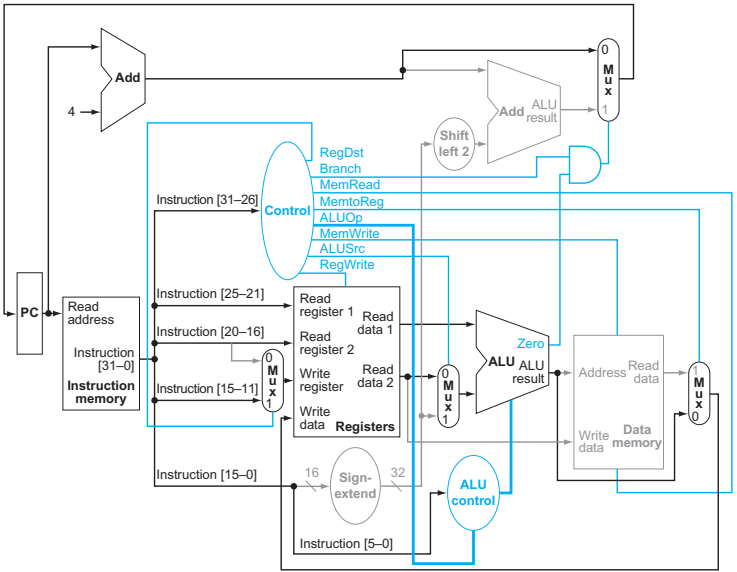
Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	XXXXXX	add	0010
SW	00	store word	XXXXXX	add	0010
Branch equal	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	set on less than	101010	set on less than	0111

A unidade de controle principal

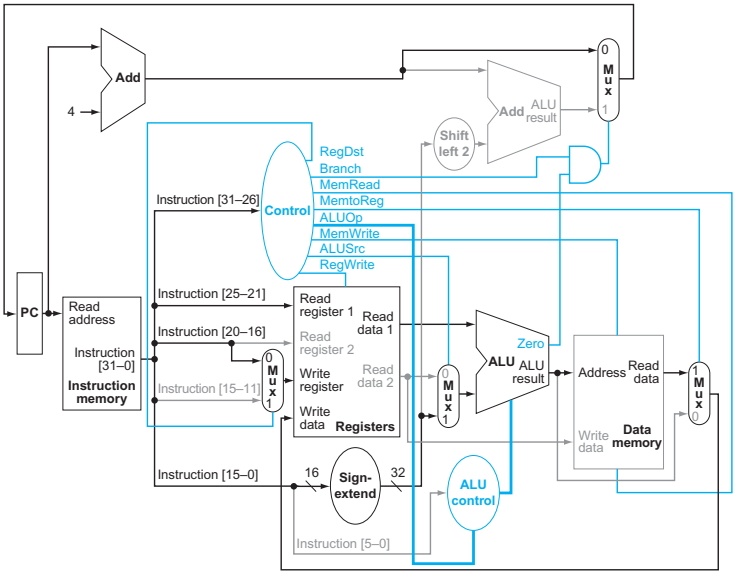


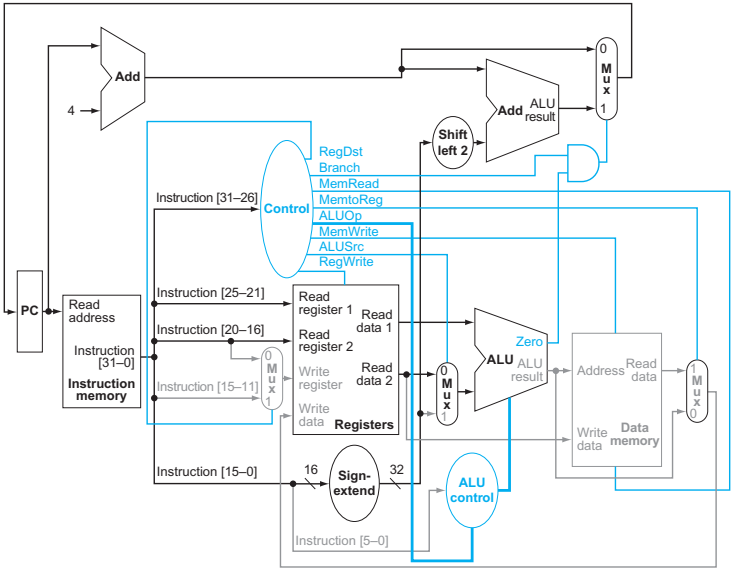
O Datapath com a unidade de Controle




Add
4
M
u
x
0
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign-extend
32
ALU control

Instruções Load

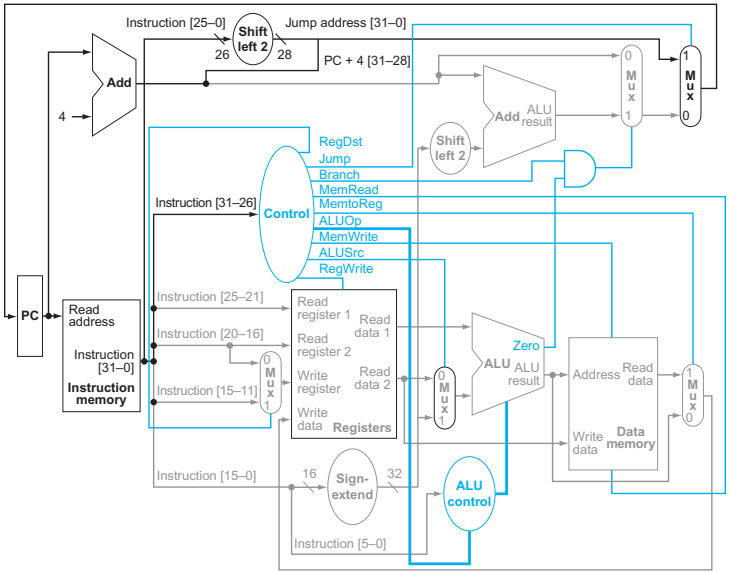



Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
16
Sign-extend
32
ALU control
Zero
ALU
ALU result
Address
Read data
Write data
Data memory

Implementando Jumps

- Jump usa o endereço de uma palavra
- Atualiza o PC com a concatenação dos
  - 4 bits mais altos do PC
  - Endereço de 26 bits da instrução de jump
  - 00
- Precisa de um sinal de controle extra (que pode ser gerado pelo opcode)

Datapath com Jumps



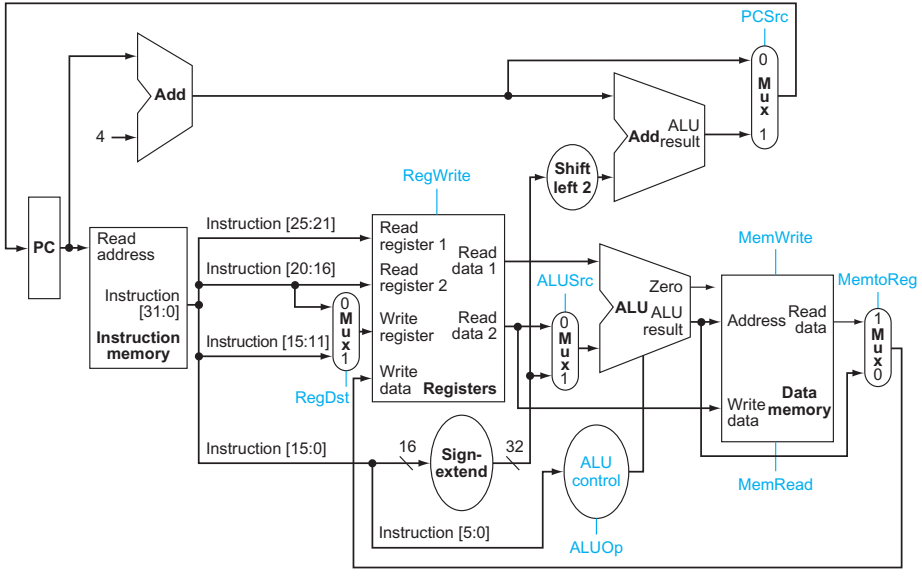
Porque este tipo de Datapath não é usado hoje em dia?

UFABC

- O delay mais longo determina o período do clock
  - Caminho crítico - Load Instruction
  - Memória de instrução $\rightarrow$ banco de registradores $\rightarrow$ ALU $\rightarrow$ memória de dados $\rightarrow$ banco de registradores
- Não é razoável alterar o período a cada instrução
- Viola o princípio de tornar rápido o caso mais comum
- Veremos como melhorar o desempenho através de pipelining

Pipelining

O Datapath com a unidade de Controle

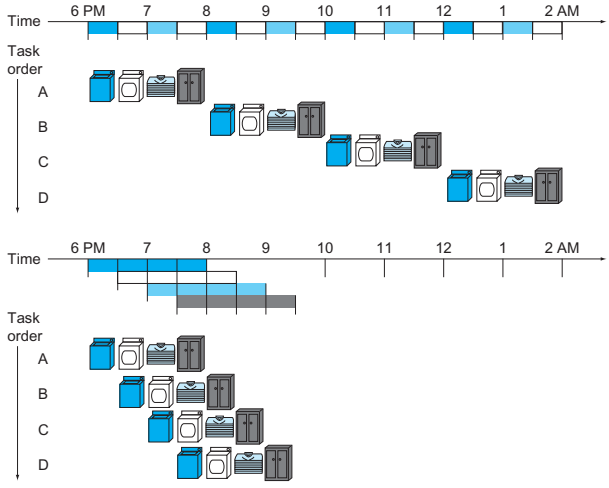


Porque este tipo de Datapath não é usado hoje em dia?

- O delay mais longo determina o período do clock
  - Caminho crítico - Load Instruction
  - Memória de instrução $\rightarrow$ banco de registradores $\rightarrow$ ALU $\rightarrow$ memória de dados $\rightarrow$ banco de registradores
- Não é razoável alterar o período a cada instrução
- Viola o princípio de tornar rápido o caso mais comum
- Veremos como melhorar o desempenho através de pipelining

Uma analogia com lavar roupas

Execuções paralelas de tarefas para aumento de desempenho



- 4 cargas na máquina
- Tempo total: 8 horas
- Versão com pipeline
  - Tempo total: 3.5h, ou 2.3x + rápido
  - *Speedup* = $2n/0.5n + 1.5 \approx 4$ = número de estágios!

UFABC

Cinco estágios, um passo por estágio

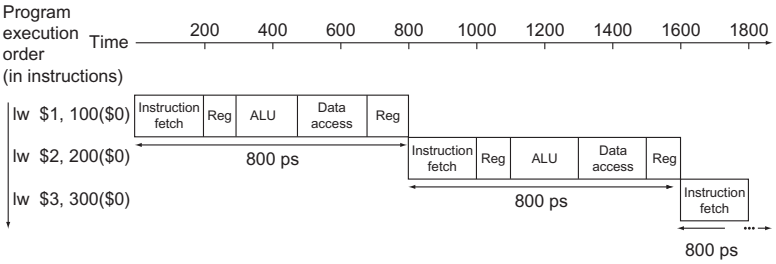
1. `IF` *Instruction Fetch* - Busca instrução na memória.
2. `ID` *Instruction Decode* - Decodifica a instrução e lê os registradores.
3. `EX` *Execute* - Executa a operação dada pela instrução e calcula endereços.
4. `MEM` *Memory* - Acesso aos operandos em memória.
5. `WB` *Write Back* - Escreve o resultado da operação de volta ao registrador.

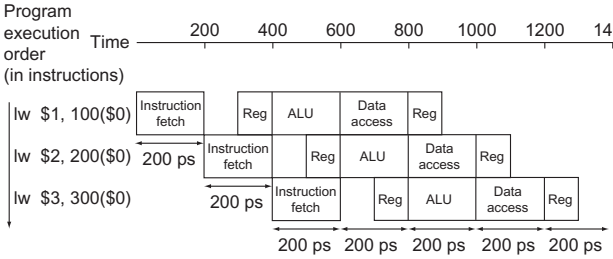
UFABC

- Assuma que o tempo por estágio do pipeline seja de
  - 100ps para leitura ou escrita de registradores
  - 200ps para os demais
- Compare o desempenho de um datapath monociclo com um datapath com pipeline

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (`lw`)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (`sw`)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (`add`, `sub`, `AND`, `OR`, `slt`)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (`beq`)	200 ps	100 ps	200 ps			500 ps

Desempenho do pipeline





- Monociclo
  - $T_c = 800$ ps
- Pipelined
  - $T_c = 200$ ps

UFABC

- Se todos os estágios estiverem equilibrados (*i.e.* levam o mesmo tempo)

$$\text{Tempo entre instruções}_{\text{Pipeline}} = \frac{\text{Tempo entre instruções}_{\text{Sem pipeline}}}{\text{Número de estágios}}$$

- Se não estiverem balanceados, então o *speedup* é menor
- O *speedup* é devido ao aumento da **vazão** (*en:* ***throughput***)
  - Contudo, o tempo total no início ao fim da execução de uma instrução, ou a **latência** (*en:* ***latency***) não se altera.

UFABC

- A ISA do MIPS foi criada para ser executada em um pipeline.
  - Todas as instruções tem 32 bits.
    - Facilita o *fetch* e *decode* em um ciclo.
    - Compare com o x86 que tem instruções de 1 a 17 bytes.
  - Poucos formatos e, dentre eles, com regularidade.
    - É possível fazer a decodificação e a leitura de registradores em um único passo.
  - Endereçamento apenas por load e stores.
    - Permite calcular o endereço no 3^o estágio do pipeline e fazer. os acessos no 4^o.
  - Operandos de instruções de acesso à memória devem ser alinhados.
    - Garante que os acessos à memória tomem apenas um ciclo.

Pipeline Hazards

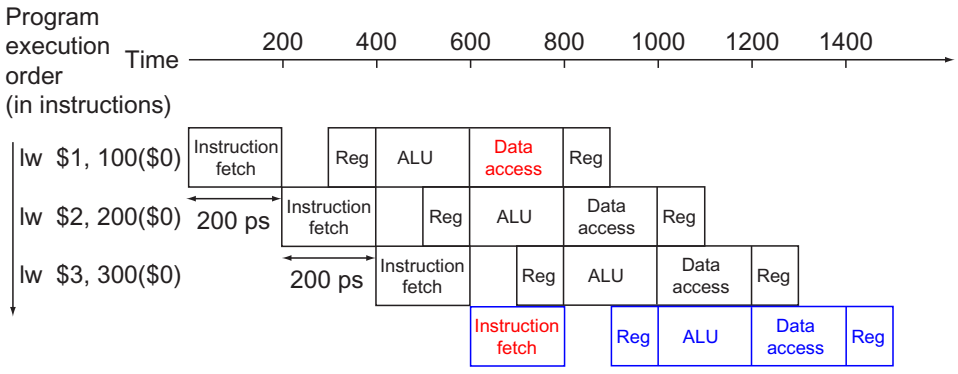
UFABC

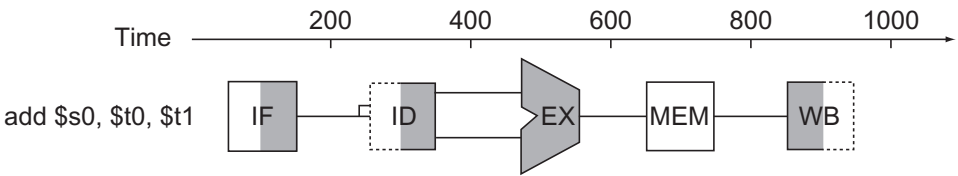
- Em certas situações não é possível iniciar a execução de uma instrução no ciclo imediatamente após a instrução anterior
- Essas situaçoes são chamadas de ***hazards*** e são, comumente, divididas em 3 tipos:
  - *Structure hazards*
    - Ocorrem quando um recurso necessário está ocupado.
  - *Data hazards*
    - Ocorrem quando é necessário aguardar que uma instrução anterior termine sua leitura/escrita de dados.
  - *Control hazards* ou *Branch hazard*
    - Ocorrem quando uma decisão de controle de execução depende de uma instrução anterior.

Structure hazards

UFABC

- Ocorrem quando há conflito por um recurso
- Possível no MIPS, se houver uma única memória
  - Load/Store precisam acessar dados
  - Busca de instruções terá que **adiar** (*en:* ***stall***) o ciclo
    - Acaba causando uma bolha no pipeline
  - Pipeline stalls também são conhecidos como **bolhas** (***bubbles***)
- Logo, datapaths com pipelines precisam de memórias separadas para instruções e dados para evitar atrasos
  - Ou, de maneira equivalente, caches independentes para dados e instruções


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
lw $1, 100($0)
lw $2, 200($0)
lw $3, 300($0)
Instruction fetch
Reg
ALU
Data access
Reg
200 ps
200 ps



- Cada estágio é representado por um bloco
- Os sombreados indicam que aquele estágio é utilizado pela instrução sendo executada
  - Sombreados à direita indicam leitura e à esquerda escrita
- No exemplo, a instrução `add` não faz acesso à memória, logo a caixa `MEM` não está sombreada

UFABC

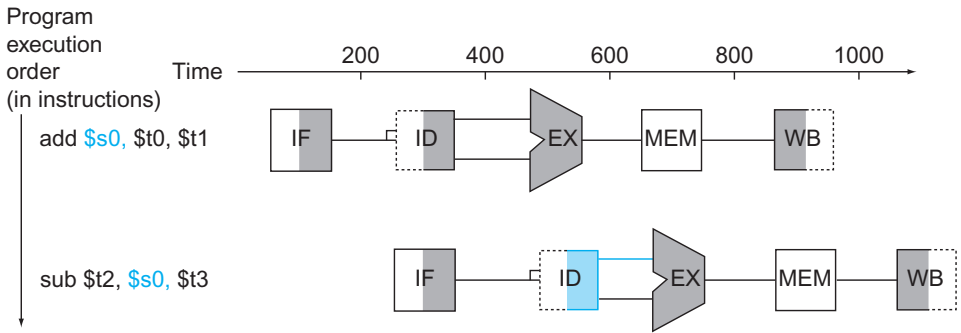
- Uma instrução depende do término do estágio de acesso aos dados de uma instrução anterior

```
1 add     $s0, $t0, $t1
2 sub     $t2, $s0, $t3
```

- Ambas instruções usam `$s0`
  - `add` escreve em `$s0`
  - `sub` depende do resultado de `add`

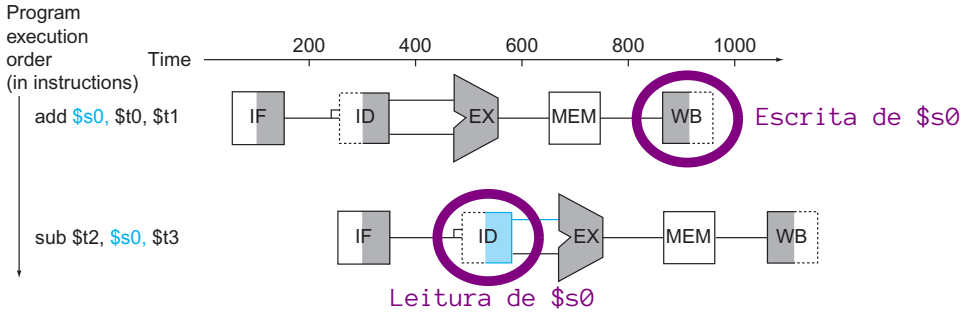
UFABC

```
1  add         $s0, $t0, $t1
2  sub         $t2, $s0, $t3
```

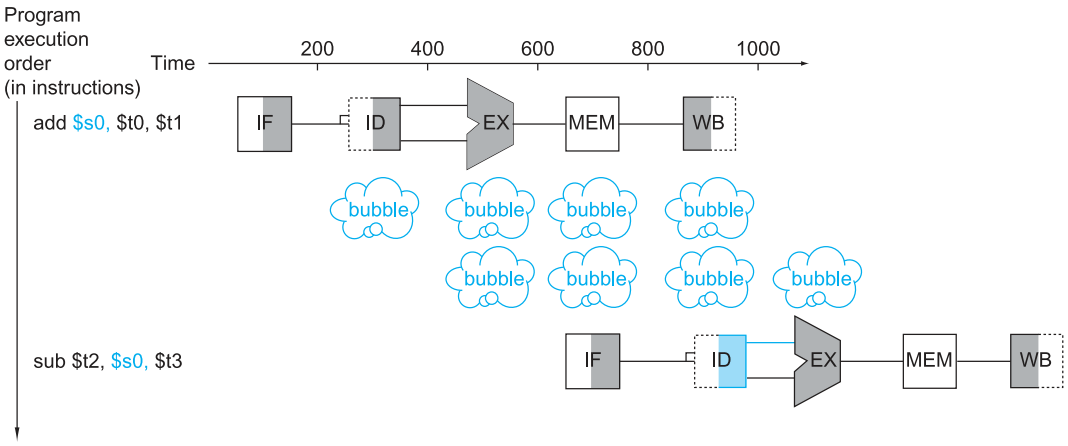


UFABC

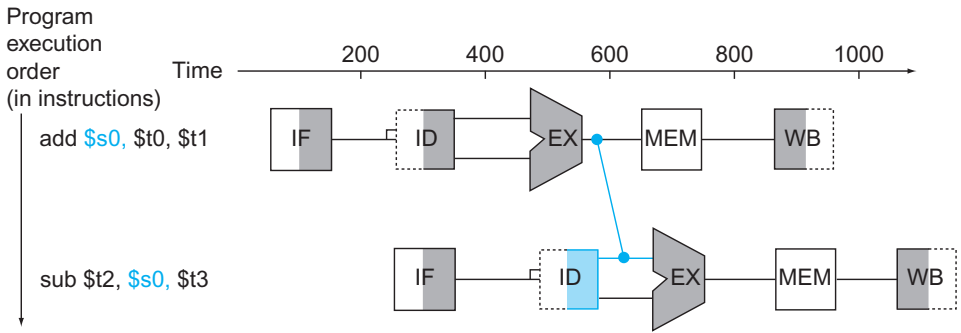
```
1  add         $s0, $t0, $t1
2  sub         $t2, $s0, $t3
```



- Inserimos bolhas no pipeline para garantir que o valor necessário para a instrução já esteja disponível



- Adiantamos o valor logo depois do estágio de execução para a próxima instrução
- Chamado de ***forwarding*** ou ***bypassing***
- Exige hardware adicional no datapath



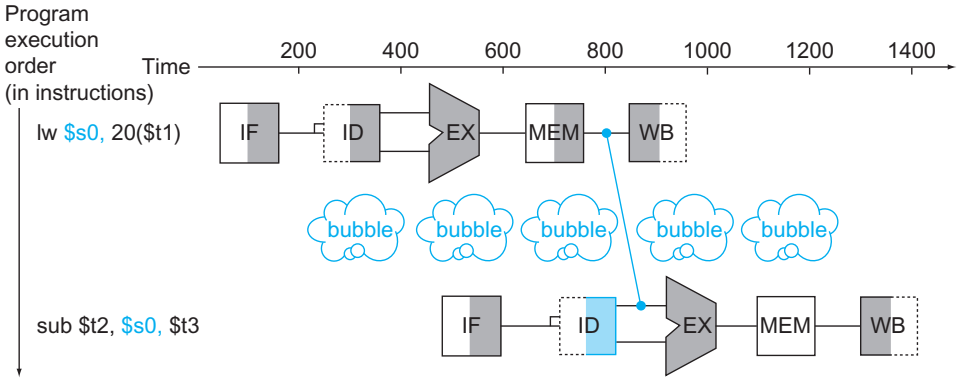
UFABC

- Nem sempre apenas o fowarding é suficiente para resolver um hazard

```
1 lw $s0, 20($t1)
2 sub $t2, $s0, $t3
```

- O valor só vai estar disponível após o estágio **MEM**
- Não dá para fazer forward voltando no tempo! :D

- Possível solução é combinar *forwarding* com *pipeline stalls*



Reordenando instruções para evitar bolhas

- Código em C:

```
1  a = b + e;
2  c = b + f;
```

- Código MIPS (variáveis acessíveis a partir de `$t0`)

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

- Quais são os hazards? Quais se resolvem com forwarding?

Reordenando instruções para evitar bolhas

- Código em C:

```
1  a = b + e;
2  c = b + f;
```

- Código MIPS (variáveis acessíveis a partir de `$t0`)

```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  add $t3, $t1, $t2
4  sw  $t3, 12($t0)
5  lw  $t4, 8($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

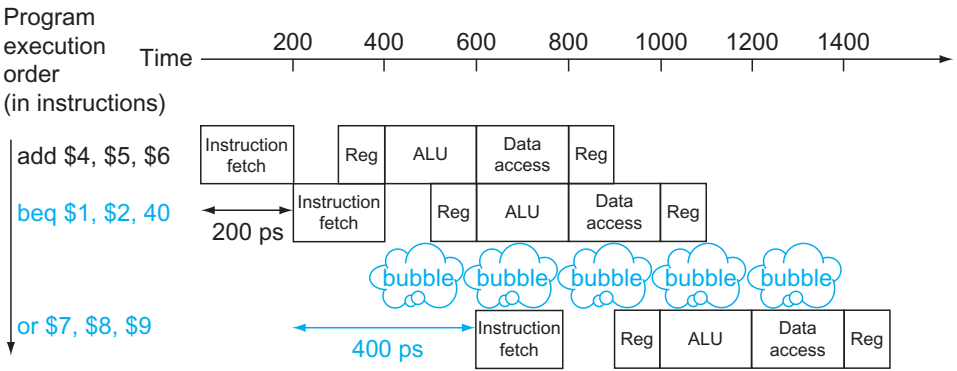
```
1  lw  $t1, 0($t0)
2  lw  $t2, 4($t0)
3  lw  $t4, 8($t0)
4  add $t3, $t1, $t2
5  sw  $t3, 12($t0)
6  add $t5, $t1, $t4
7  sw  $t5, 16($t0)
```

- Quais são os hazards? Quais se resolvem com forwarding?

UFABC

- Branches determinam o controle do fluxo do programa
  - A busca da próxima instrução depende da avaliação do branch
  - O pipeline não é capaz de sempre buscar a instrução correta
    - Ainda está trabalhando na identificação da instrução (estágio `ID`)
- No MIPS
  - Necessário comparar registradores e computar o destino do branch nos primeiros estágios do pipeline
  - O MIPS tem hardware para fazer isto no estágio `ID`

- Espera o resultado do branch antes de buscar a próxima instrução



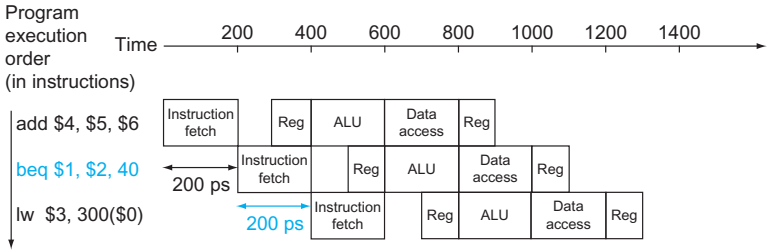
UFABC

- Pipelines mais longos não conseguem determinar o resultado de um branch de maneira adiantada como o MIPS
  - O custo de inserir bolhas se torna proibitivo
- Saída? Chuta-se (cof cof), **predizemos** o resultado do branch
  - Se o chute for errado, é causado um stall
- No MIPS
  - É capaz de predizer branches não tomados (ele sempre chuta que não é tomado)
  - Logo, carrega, sempre, a instrução seguinte ao branch

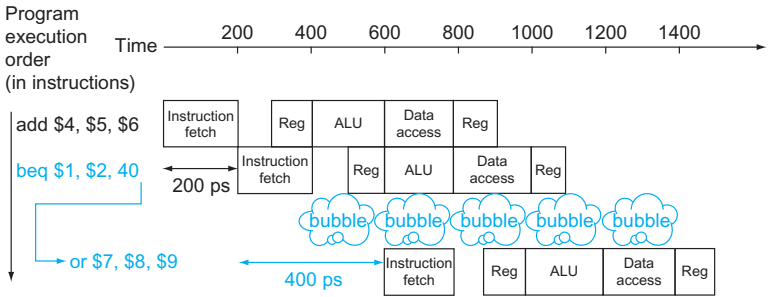
Branch Predictor no MIPS (Not Taken)

UFABC

Predição correta



Predição incorreta



UFABC

Preditor de branches estático

- Baseado no comportamento típico do branch
- Exemplo: laços e ifs
  - Prediz que branches para trás são tomados
  - Prediz que branches para frente não são tomados

Preditor de branches dinâmico

- Exige hardware adicional para medir o comportamento do branch
  - Normalmente armazena informações sobre o comportamento mais recente de cada um dos branches
  - Assume que os comportamentos passados são indicativos do comportamento futuro
    - Quando erra, insere uma bolha, carrega a instrução correta e atualiza o histórico
    - Usado em processadores modernos e tem acerto superior a 90%

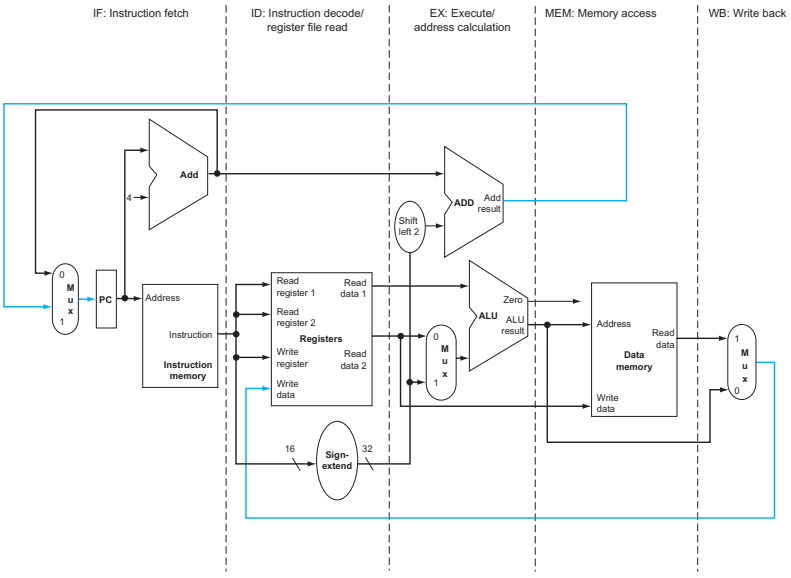
Resumo sobre pipelining

UFABC

- Pipelining aumenta o desempenho pelo aumento da vazão
  - Mas não faz nada quanto à latência de cada instrução
  - Baseia-se na execução em paralelo de múltiplas instruções
- Pode sofrer de hazards
  - Estruturais, de dados ou de controle
- A ISA pode influenciar significativamente a complexidade de implementação de um pipeline

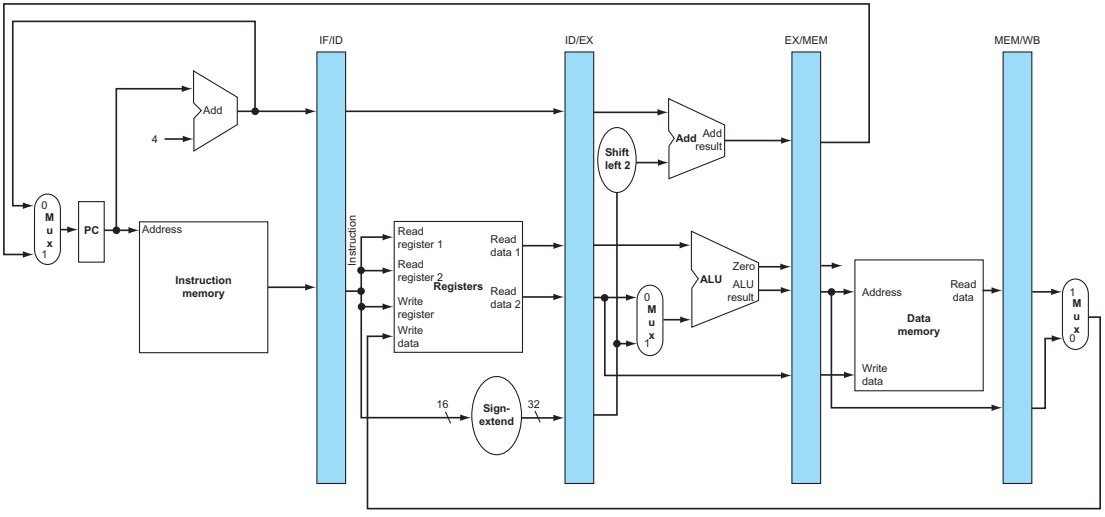
Pipelining no MIPS

MIPS Pipelined Datapath



- Fluxo direita → esquerda pode causar hazards

Registradores de pipeline

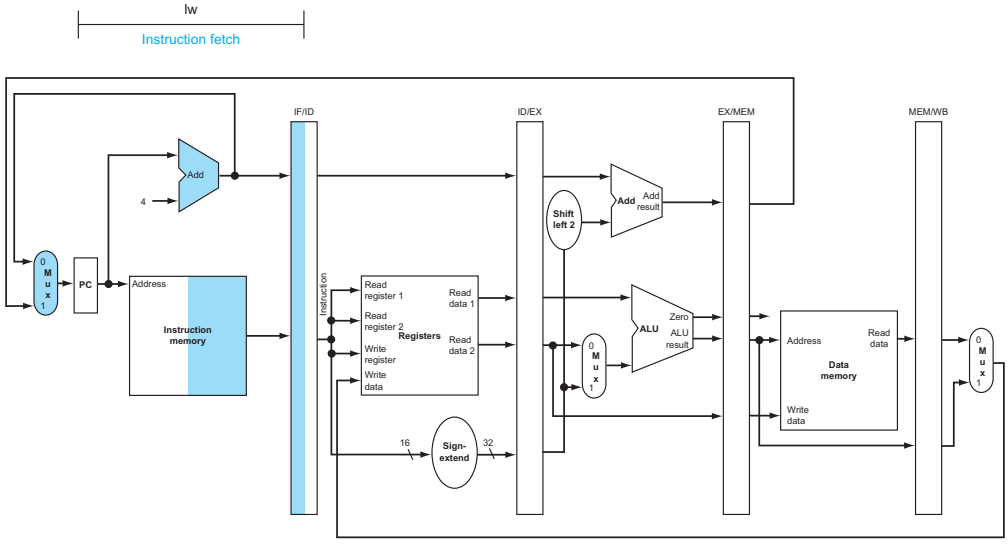


Operação do pipeline

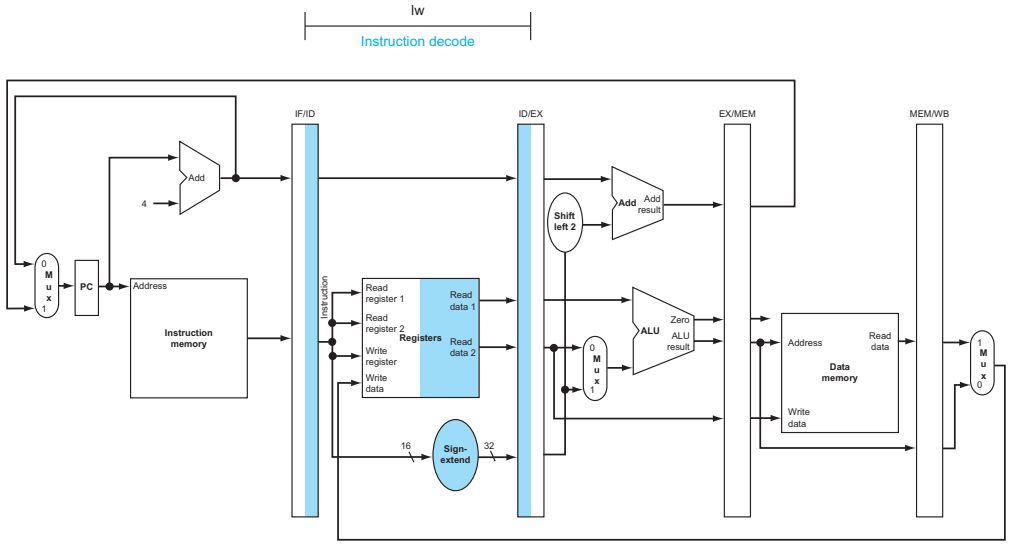
UFABC

- Ciclo a ciclo as instruções são passadas pelos estágios do datapath
- Há duas maneiras comuns de se mostrar o funcionamento do pipeline
  - *"Single-clock-cycle"pipeline diagram*
    - Mostra o uso do pipeline em um único ciclo
    - Destaca os recursos em uso
  - *"Multi-clock-cycle"pipeline diagram*
    - Mostra a operação durante o tempo
- Vamos começar dando uma olhada no *single-clock-cycle* para as operações de *load* e *store*

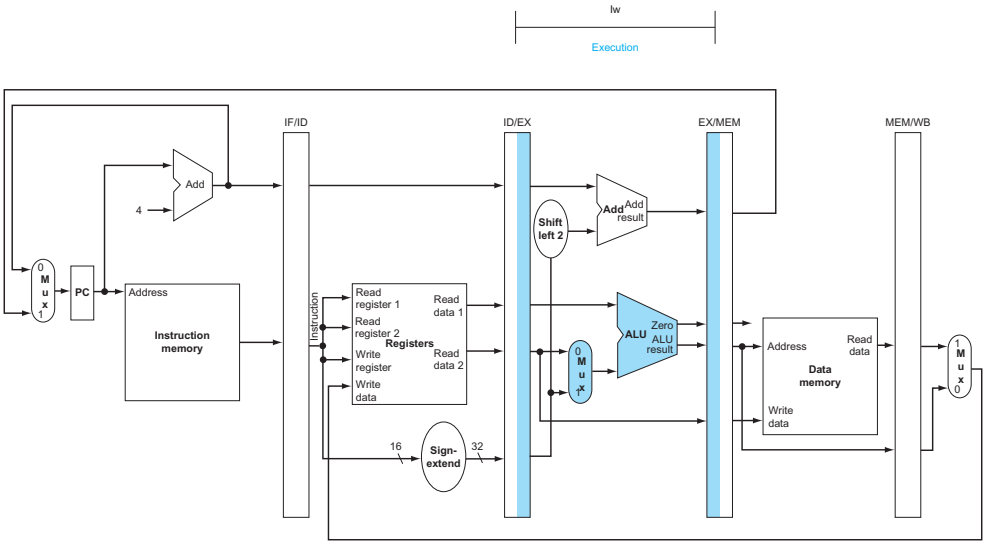
IF para *load* e *store*



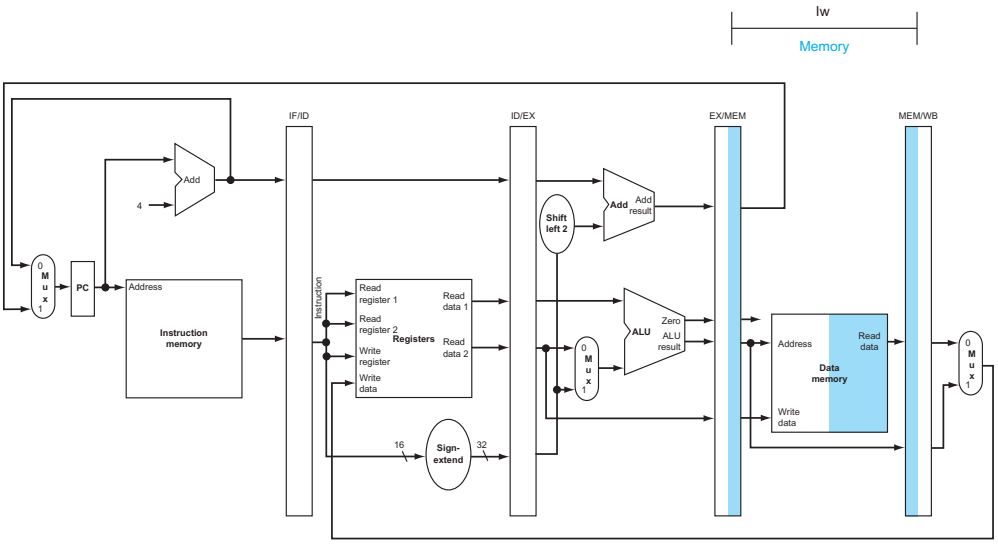
ID para *load* e *store*

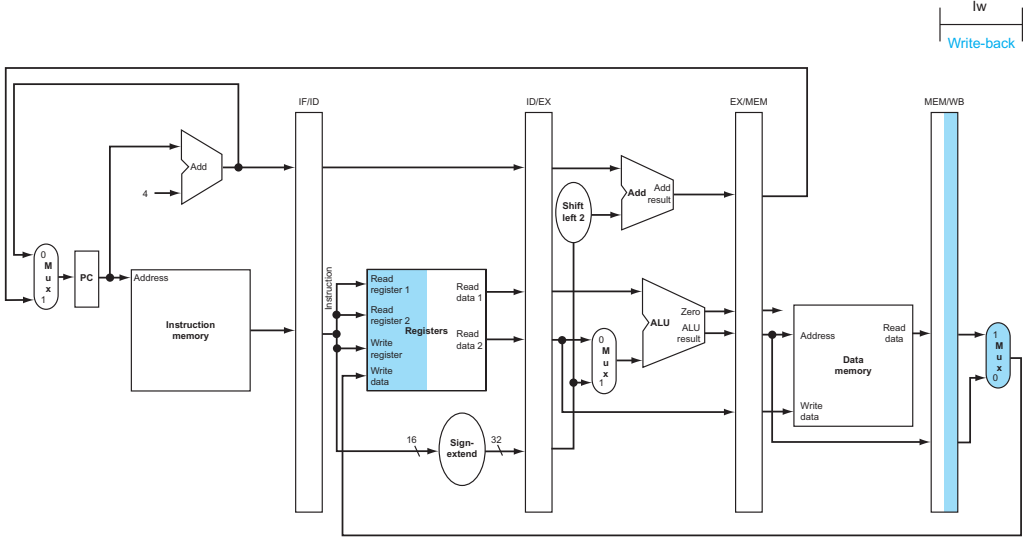


EX para *load*


lw
Execution
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
M
u
x
0
1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
Shift left 2
Add
Add result
ALU
Zero
ALU result
Address
Data memory
Read data
Write data
16
Sign-extend
32

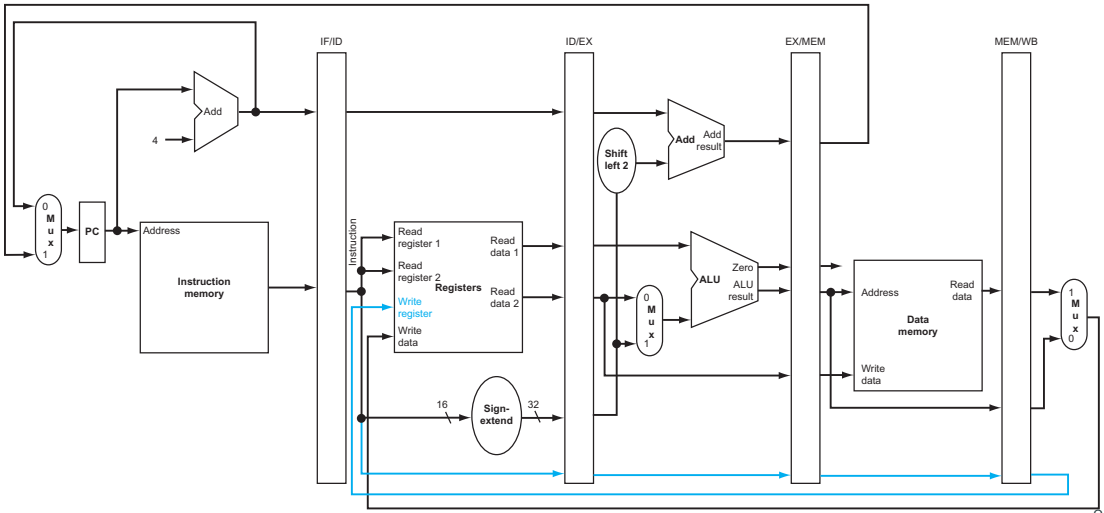
MEM para *load*



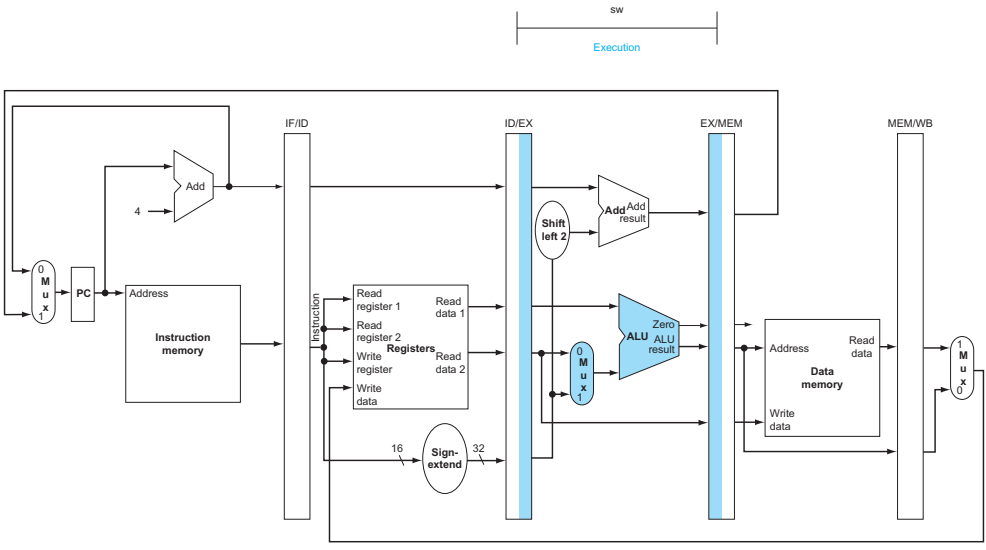


Possível número de registrador incorreto

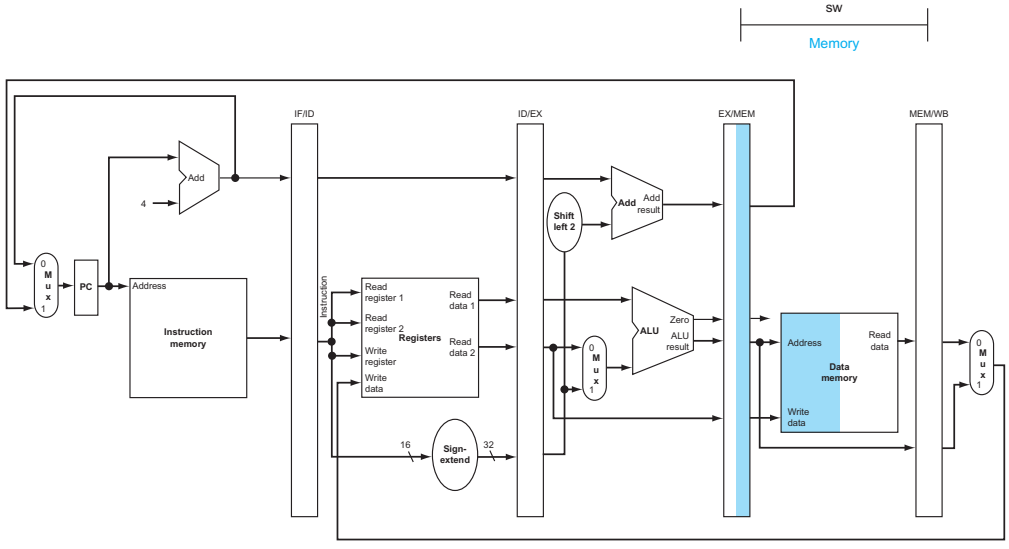
Datapath corrigido para *load*



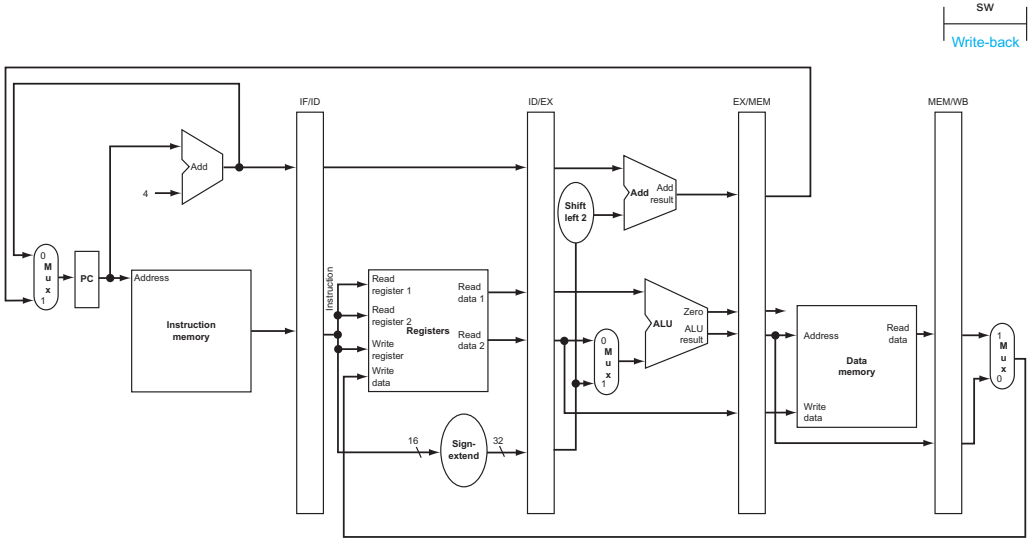
EX para *store*



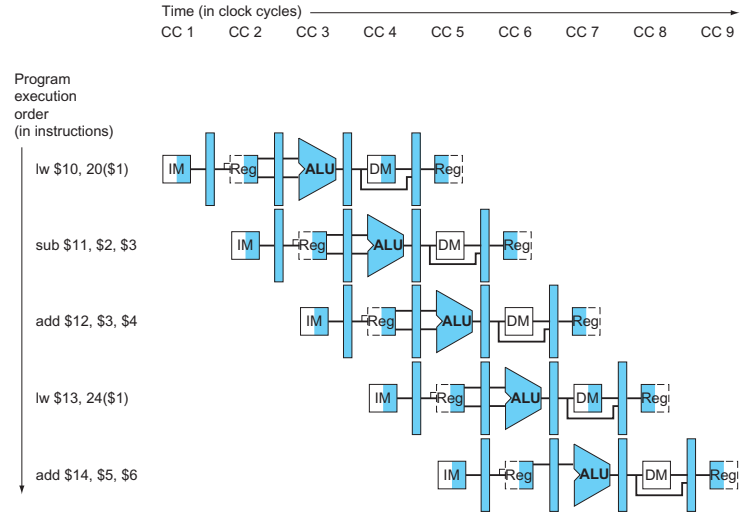
MEM para *store*



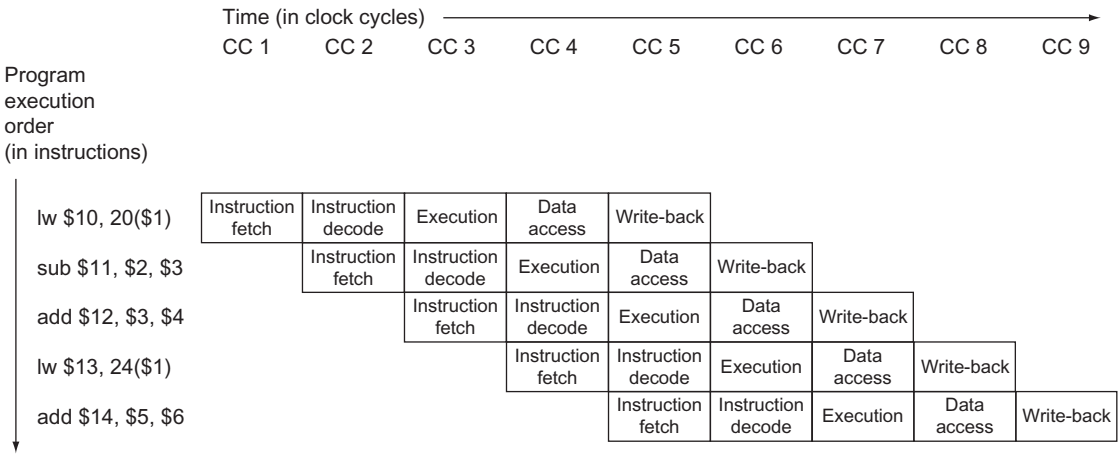
WB para *store*



Pode ser desenhado mostrando o uso dos recursos

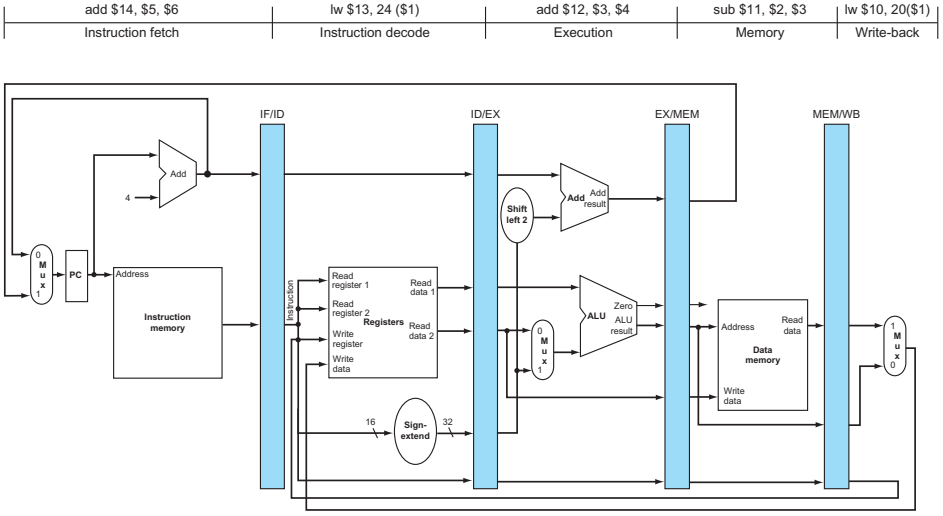


Ou mostrando o uso dos estágios

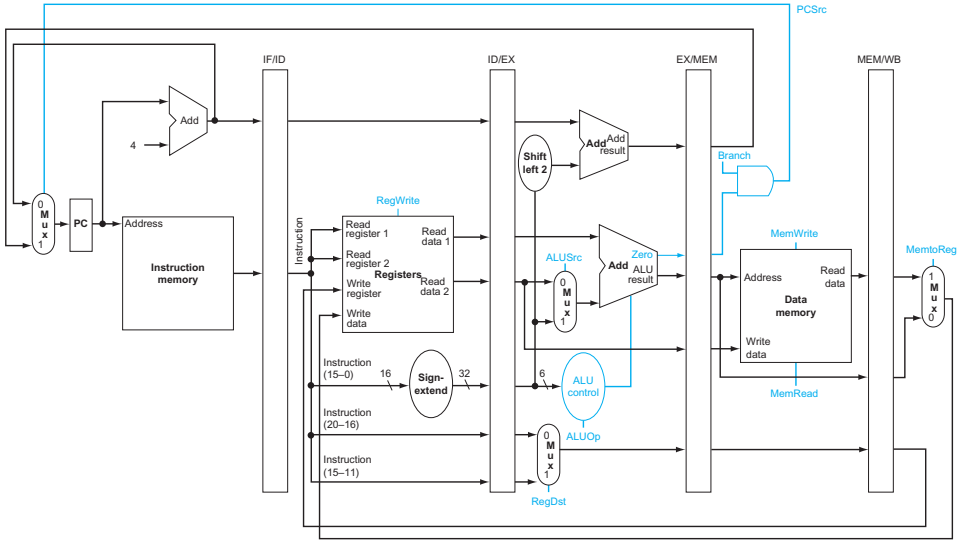


UFABC

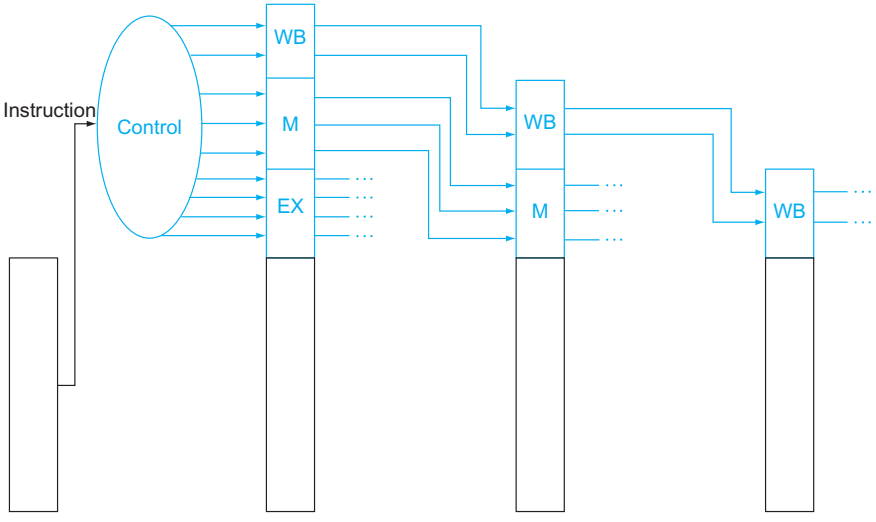
Mostrando o ciclo 5 das figuras anteriores



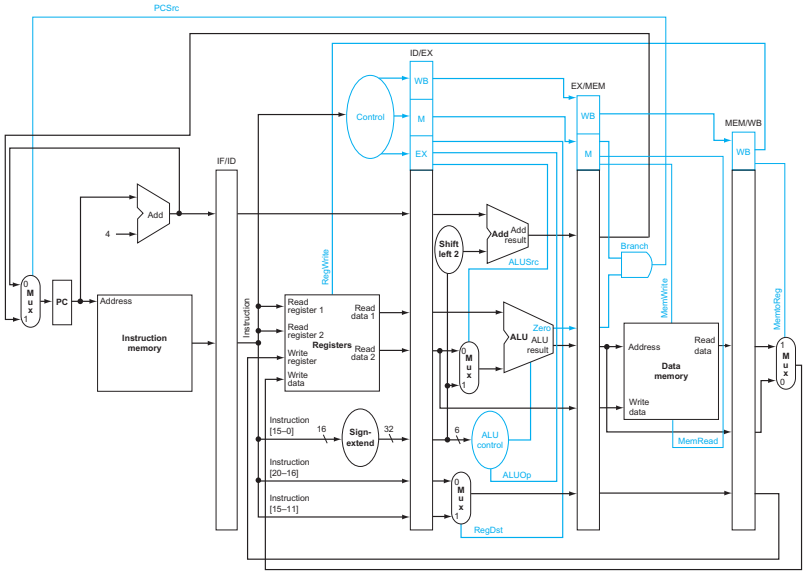
Controle do pipeline (simplificado)



Controle do pipeline



Controle do pipeline



Para saber mais sobre pipeline e hazards de dados, controle e exceções

UFABC

- O livro traz na sua seção 4.7 como o pipeline deve ser alterado para levar em consideração os hazards e fazer o forwarding/bypassing.
- A seção 4.9 descreve o tratamento de interrupções e exceções.
- Aqui não vamos entrar nestes detalhes, mas caso queira ainda mais detalhes do que no livro [PH] vejam o livro [HP] disponível na bibliografia da disciplina.
- Vamos dar apenas uma rápida olhada sobre o branch predictor.

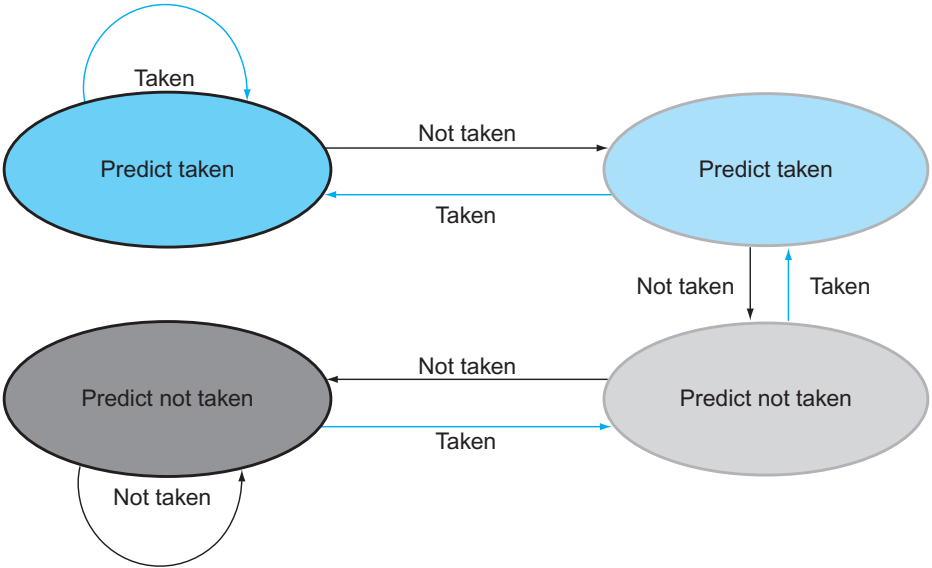
Dynamic branch predictor

UFABC

- Em processadores com um pipeline longo, o custo de chutar um branch incorreto pode ser muito caro
- A solução é usar algo mais certeiro que sempre chutar que um branch será tomado ou não
- Dynamic branch predictors usam uma predição dinâmica
  - Branch predictor buffer (ou branch predictor table)
  - Indexado pelos endereços dos branches mais recentes
  - Guarda histórico do branch: se foi tomado ou não
- Para executar um branch
  - Verifica a tabela, chuta o mesmo resultado
  - Começa a execução da instrução apontada pelo chute
  - Se incorreto, limpa (***flush***) o pipeline e atualiza a tabela

- Têm um problema de errar a predição duas vezes em laços aninhados!
  - Erram na última iteração do laço
  - Erram na primeira iteração do laço

```
1 outer: ...
2        ...
3 inner: ...
4        beq ..., ..., inner
5        ...
6        beq ..., ..., outer
```


Taken
Predict taken
Not taken
Taken
Predict taken
Not taken
Taken
Predict not taken
Not taken
Taken
Predict not taken
Not taken

- Correlating predictors
  - Mantem 2 automatos, um para quando um branch foi tomado pela última vez e outro para quando não foi tomado. Mostra-se que isso traz um aumento significativo do acerto.
- Tournament predictors
  - Mantém 2 preditores para cada branch. Conforme o branch vai executando, os preditores são atualizados e a cada momento o que teve mais acertos históricos é utilizado para prever o resultado. Alguns processadores recentes usam este esquema.

- Uma maneira de aumentar o paralelismo é aumentar o comprimento do pipeline (mas isto traz, como já vimos, outros problemas)
- Alguns processadores são capazes de executar mais de uma instrução ao mesmo tempo no mesmo estágio do pipeline
  - Para isto, eles incluem hardware adicional para evitar conflitos.
  - Isso, na nossa analogia, é equivalente a instalar diversas máquinas de lavar, secar, ...
  - O nome geral que se dá a essa técnica é ***multiple issue***

Multiple issue

UFABC

- **Issue slots** são cada uma das posições nas quais as instruções podem ser selecionadas para execução em um único ciclo.
- **Issue packet** é o conjunto de instruções que será selecionada para execução em um único ciclo
- Estático
  - As decisões de quais instruções serão executadas ao mesmo tempo é definida em tempo de compilação.
  - Em arquiteturas **VLIW** (***very long instruction word***) as diversas instruções independentes são empacotadas em uma única instrução (tipicamente com diversos campos opcode diferentes)
- Dinâmico
  - As decisões de quais instruções serão executadas ao mesmo tempo é definida pelo processador em tempo de execução.
  - Normalmente o compilador já fez um primeiro passo de otimização.

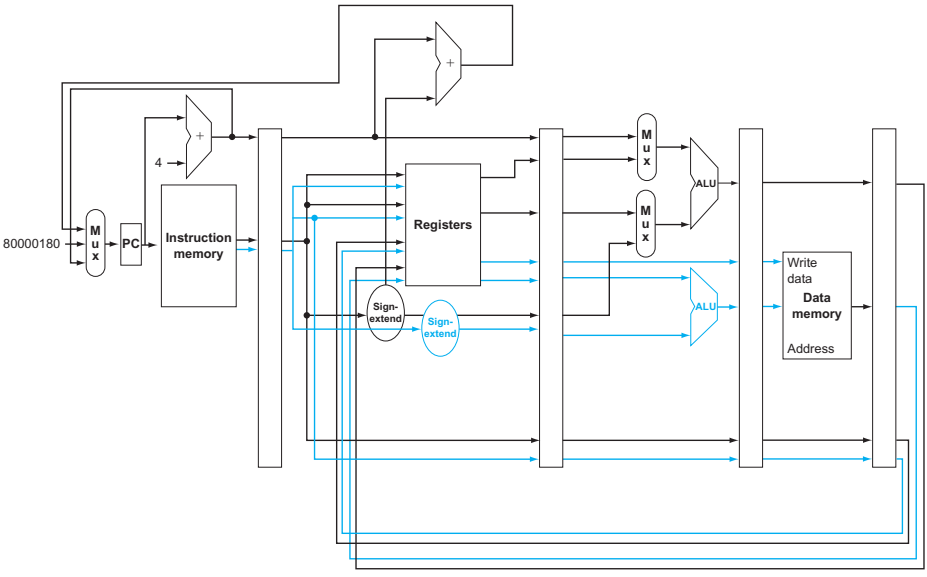
UFABC

- O compilador é o responsável por remover todos (ou quase) os hazards
  - Precisa reordenar instruções dentro de issue packets
  - Não pode haver dependêcias dentro de um pacote
  - Talvez seja possível haver dependências entre pacotes
    - Mas depende da ISA! O compilador precisa saber dos detalhes
  - Completa pacote com `nops` caso necessário

- Pacotes com dois slots
  - Um para instruções ALU/Branch
  - Um para instruções load/store
  - Precisam estar alinhados em fronteiras de 64 bits

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

MIPS com dual-issue estático



Hazards no MIPS com dual-issue

UFABC

- Mais instruções executando em paralelo.
- EX data hazard
  - Forwarding resolveu os stalls com um sistema de single-issue
  - Agora não podemos usar o resultado da ALU em um load/store no mesmo pacote

```
1 add  $t0, $s0, $s1
2 load $s2, 0($t0)
```

  - É preciso quebrar em dois pacotes (essencialmente estamos fazendo um stall)
- Load-use hazard
  - Ainda há uma latência para usarmos de um ciclo, mas passa a ser uma espera de 2 instruções
- É necessário um mecanismo de escalonamento mais agressivo por parte do compilador

Exemplo de escalonamento

UFABC

Queremos escalonar:

```
Loop: lw    $t0, 0($s1)    # $t0=array element
      addu  $t0,$t0,$s2# add scalar in $s2
      sw    $t0, 0($s1)# store result
      addi  $s1,$s1,-4# decrement pointer
      bne   $s1,$zero,Loop# branch $s1!=0
```

Versão otimizada:

	ALU or branch instruction	Data transfer instruction	Clock cycle
Loop:		lw $t0, 0($s1)	1
	addi $s1,$s1,-4		2
	addu $t0,$t0,$s2		3
	bne $s1,$zero,Loop	sw $t0, 4($s1)	4

- IPC = 5/4 = 1.25 (ideal IPC = 2)

UFABC

- Desenrolar laços (*loop unroling*) permite que aproveitemos melhor os recursos do processador
- Replicamos o corpo do laço para expor mais paralelismo
  - Reduz o overhead do controle do laço
- Utilizamos diferentes registradores por iteração
  - Chamado de ”register renaming”
  - Evita anti-dependências *loop-carried*
    - Ex. Store seguido de load de um mesmo registrador
    - Também chamado de dependência de nome
    - Ocorre quando se reusa o nome de um registrador

Desenrolando laços - Exemplo

	ALU or branch instruction		Data transfer instruction		Clock cycle
Loop:	addi	$s1,$s1,-16	lw	$t0, 0($s1)	1
			lw	$t1,12($s1)	2
	addu	$t0,$t0,$s2	lw	$t2, 8($s1)	3
	addu	$t1,$t1,$s2	lw	$t3, 4($s1)	4
	addu	$t2,$t2,$s2	sw	$t0, 16($s1)	5
	addu	$t3,$t3,$s2	sw	$t1,12($s1)	6
			sw	$t2, 8($s1)	7
	bne	$s1,$zero,Loop	sw	$t3, 4($s1)	8

- IPC = 14/8 = 1.75 (ideal IPC = 2)
  - Estamos mais próximos de 2, mas pagamos em número de registradores usados e tamanho do código

UFABC

- São processadores com multiple issue dinâmico
- O próprio processador decide se deve executar 0, 1, 2, ... instruções por ciclo
  - Também é responsável por cuidar de hazards estruturais e de dados
- Diminui a pressão no escalonamento feito pelo compilador
  - Apesar de ainda ser útil
  - A semântica do código é garantida pela CPU
- Algumas CPUs são capazes de escolher as instruções a serem executadas fora de ordem (*out-of-order*)
  - Mas ainda dão a ilusão ao usuário que as instruções estão sendo executadas em ordem

Para que fazer escalonamento dinâmico?

UFABC

- Se não fazer simplificaria o hardware e o escalonamento poderia ser feito pelo compilador?
  - Alguns stalls não podem ser determinados em tempo de compilação
    - Ex. Falhas de cache
  - Não é possível fazer o escalonamento no caso de branches
    - Já que por definição é determinado dinamicamente
  - Isto tudo é muito dependente da ISA em questão.

Multiple-issue funciona?

UFABC

- Sim! Mas não tanto quanto gostaríamos.
- Programas têm dependências que limitam o ILP.
  - Algumas são dificílimas de eliminar.
    - Ex. Pointer aliasing.
- Alguns tipos de paralelismo são difíceis de expor.
  - Tamanho de janela limitada durante o issue de instruções.
- Há atrasos no acesso à memória e uma banda limitada.
  - É difícil de manter o pipeline cheio.
- Especulação pode ajudar muito, se for bem feita.

- A complexidade de escalonamentos dinâmicos e especulação exigem gasto de energia.
- Dependendo da aplicação, múltiplos núcleos simples podem ser mais vantajosos.

Microprocessor	Year	Clock Rate	Pipeline Stages	Issue Width	Out-of-Order/ Speculation	Cores/ Chip	Power	
Intel 486	1989	25 MHz	5	1	No	1	5	W
Intel Pentium	1993	66 MHz	5	2	No	1	10	W
Intel Pentium Pro	1997	200 MHz	10	3	Yes	1	29	W
Intel Pentium 4 Willamette	2001	2000 MHz	22	3	Yes	1	75	W
Intel Pentium 4 Prescott	2004	3600 MHz	31	3	Yes	1	103	W
Intel Core	2006	2930 MHz	14	4	Yes	2	75	W
Intel Core i5 Nehalem	2010	3300 MHz	14	4	Yes	1	87	W
Intel Core i5 Ivy Bridge	2012	3400 MHz	14	4	Yes	8	77	W

Arm Cortex-A8 e Intel i7

UFABC

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128 - 1024 KiB	256 KiB
3rd level cache (shared)	–	2 - 8 MiB