Arquitetura de um computador

1. Caminho de dados de um processador:

- Implementação de um processador MIPS:
  - Versão simplificada (monociclo)
  - Versão pipeline
- Execução de uma instrução:

FETCH → DECODE → EXECUTE

  - PC → Acesso à memória de instruções, obtém instrução **(fetch)**
  - **Decodifica a instrução** e **obtém os dados** necessários para a execução.

→ O decode é feito pela interpretação dos formatos tipo R, I e J

→ Após o decode, há o acesso aos registradores (obtenção dos dados) e realização das operações (processamento).

  - Dependendo da classe da instrução:
    - Usa uma ULA para cálculos:
      - Instruções aritméticas: resultado da operação
      - Instruções de acesso à memória: **Cálculo do endereço**
      - Instruções de desvio: **Endereço do desvio.**
    - Faz acesso à memória (load/store word)
  - PC = Destino do desvio ou PC+4 (porque cada instrução ocupa 4 bytes na memória)
    - Obs: Destino de desvio: operações (beq, jump, etc…)

PC+4: Instruções sequenciais do código

Entendendo um processador

- Princípios de design lógico:
  - informações sempre codificada em binário
- Elementos combinacionais
- Elementos sequenciais
- Metodologia de clock
  - A lógica combinacional transforma os dados durante um ciclo de clock

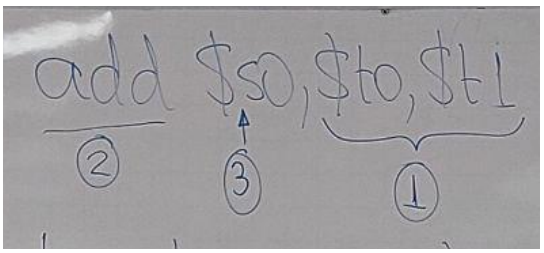
FETCH

- (imagem)
- O PC já é atualizado (PC += 4) logo após ocorrer a leitura dele pela memória de instrução.

DECODE

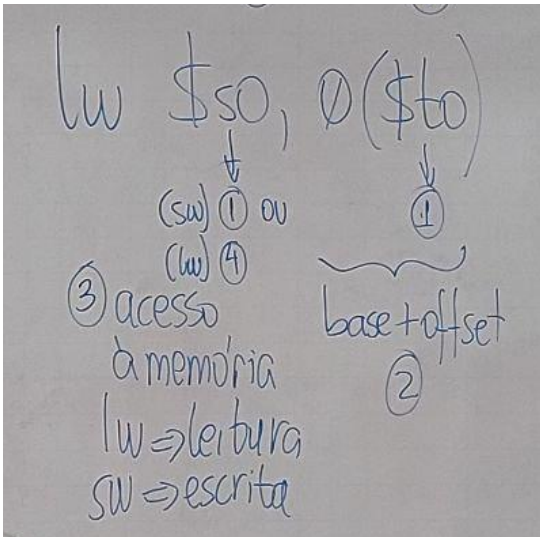
Instruções do tipo R

1. Lê dois registradores operandos
2. Faz a operação lógica/aritmética
3. Escreve o resultado num registrador



Instruções load/store

1. Lê os registradores operandos
2. Calcula o endereço de memória usando o offset de 16 bits
    - Usa a ULA e também um extensor de sinal do offset (para transformar o offset de 16 bits em um dado de 32 bits e conseguir ser lido pela ULA)
3. Load: lê da memória e escreve no registrador
4. Store: lê do registrador e escreve na memória



Boa questão de prova: Por que existem duas flags na memória de acesso, já que elas são complementares? (MemWrite & MemRead)
→Resposta: **Por conta das instruções tipo R,** que não irão nem ler, nem escrever na memória

Instruções de desvio

1. Lê os registradores operandos
2. Compara os operandos
    - Usa ULA para subtrair e verifica se a saída é zero
3. Calcula o endereço de destino do desvio
    - Extensão de sinal
    - 2 shifts à esquerda (palavra: 4 bytes)
    - Soma a PC+4 (já que isso já foi calculado no FETCH)

beq $s0,$s1,label

$s0 e $s1 sao registradores pra comparar ($s0 == $s1?)
label é a constante (deslocamento em instruções)
Endereço de desvio = PC + deslocamento*4

lw → instrução que mais consome tempo, pq ela faz tudo.
controladora→ Unidade que decide qual caminho de dados deve seguir.

- Controlador:
  ula→ Analisa o campo op e se for do tipo R analisa o campo funct
  nem toda instrução faz leitura ou escrita na memória → por isso temos dois sinais de controle, por isso elas não podem ser ambas ao mesmo tempo (boa questão de prova)

Boa questão de prova: Por que existem duas flags na memória de acesso, já que elas são complementares? (MemWrite & MemRead)
→Resposta: **Por conta das instruções tipo R,** que não irão nem ler, nem escrever na memória.
label→deslocamento relativo ao pc(instruções).
Endereço de destino do desvio(boa questão de prova)→pc + 4*label (pois estou convertendo instruções para bytes).
a ula só sabe trabalhar com números de 32 bits, por isso precisamos transformar esses números fazendo a **extensão de sinal** , do qual nós prolongamos o bit mais significativo até chegarmos nos 32 bits.
Não se usa memória para uma instrução beq
Um sistema operacional não deixa executar um programa se não houver memória suficiente para ele todo, pois isso prejudicaria a lógica de montagem.
se o beq deu certo dizemos que **o desvio foi tomado (ou não)**
controle (decode) tipo R:

- reg dst:1
- branch/PCSrc:0
- memRead:0
- memWrite:0
- memtoReg:0
- AluSrc:0
- RegWrite:1

Controle load:

- RegDst:0 (qualquer coisa p store)
- Branch:0
- memRead:1 (0 p store)
- memWrite:0 (1 p store)
- memtoReg:1 (qualquer coisa p store)
- AluSrc:1
- RegWrite:1 (0 p store)

Controle beq:

- RegDst: x (tanto faz)
- Branch:1
- memRead:0
- memWrite:0
- memtoReg:x (tanto faz)
- AluSrc:0
- regWrite:0

- Memória de dados é a fase mais demorada do ciclo de clock
- memória de instrucao → banco de registradores → ULA → memória de dados → banco de registradores (as 5 fases)

Etapas de execução → As 5 etapas

- Recuperação da instrução. →IF
- Leitura no banco de registradores. → ID
- Operação aritmética → Ex
- Acesso à memória de dados. → MEM
- Escrita no banco.-->wb

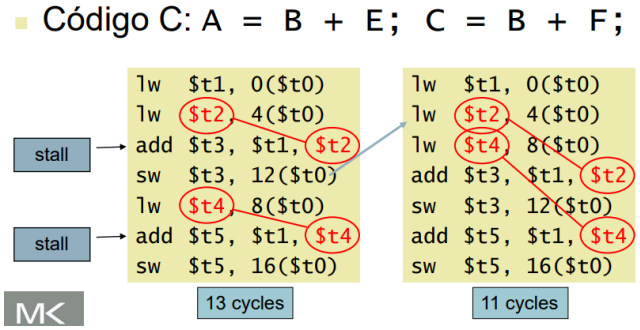
obs: Nem toda instrução faz as 5 etapas. → boa questão de prova, saber quantas etapas cada instrução faz. (tem no slide de pipeline)

Pipeline

(Técnica de hardware) não melhora o tempo de execução de cada instrução, mas otimiza a execução em paralelo
Cinza a direita(leitura)
Cinza à esquerda(escrita)
Em branco(não passa pela etapa)
Hazard de dados -> pode acontecer uma bolha/stalls entre duas funções pois uma depende da outra:
Ex: add $t0,$t1,$t2
sub $t3,$t0,$t4

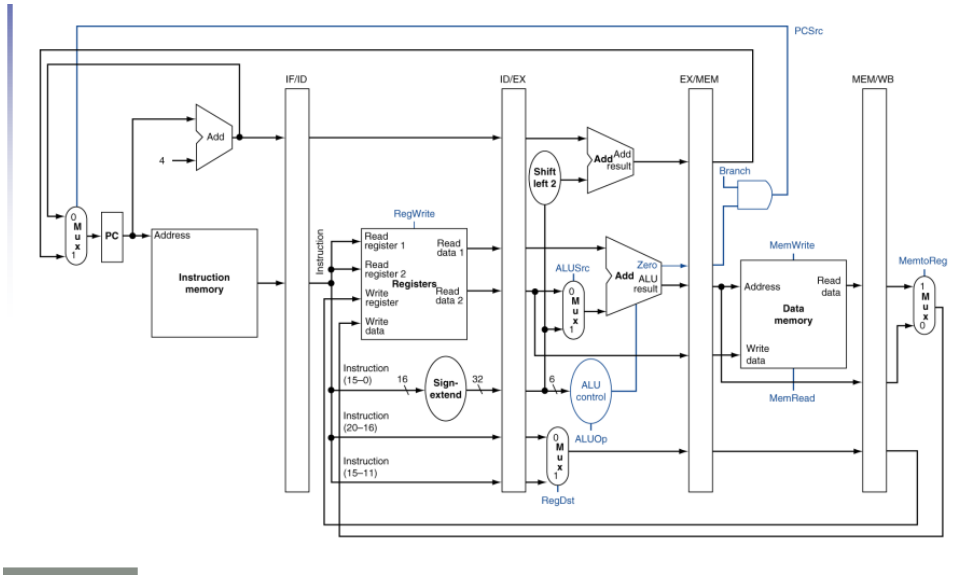
- Para resolver usa-se Forwarding(Bypassing): já usa um resultado assim que for calculado, não espera ser armazenado no registrador, requer conexões extras
- Hazard de dados Load-Use: quando o dado vem da memória, ai não é possível impedir a bolha

* compilador ou o desenvolvedor pode mudar a ordem das operações para otimizar o tempo (evitar stalls)



Stalls em desvios: espera até que a ULA calcule a instrução de desvio para que ele seja tomado (ainda gera uma bolha). Para resolver isso usa-se a predição de desvio->pipeline tenta adivinhar qual será a próxima execução, exemplo: num loop depois de rodar algumas vezes ele prediz o próximo desvio, caso não seja tomado esse caminho o resultado é jogado fora
Predição estática: gera a predição após algumas instruções repetidas, não muda
Predição dinâmica: vê o histórico das instruções e prevê o desvio, caso o desvio não seja tomado ele gera uma bolha para descartar o resultado
Registradores pipeline: usa 4 registradores na arquitetura do hardware para armazenar os dados entre os ciclos de clock para não alterar os dados que são encaminhados de outras instruções

Representação gráfica do pipeline “multi-clock-cycle”



questão de prova: perceber a diferença de lw e sw pelo lado pintado de cinza na memória
lw -> lê os registradores, lê a memória e escreve no registrador
sub e add -> lê os registradores, não usam memória, escreve no registrador
sw -> le registradores, escreve na memória e não usa o último registrador

agora existem vários multiplexadores e a controladora não configura todo mundo de uma vez, ela joga os sinais de controle nos registradores intermediários e a cada etapa ele reconfigura ou seja: novas instruções são passadas a cada ciclo para o registrador intermediário e vão sendo repassados para os outros registradores intermediários. Cada registrador dificilmente passa de 20 bytes.

Hierarquia de Memória

- **Memória**: unidade persistente para armazenamento de dados de usuário e software.
    - Acesso aos dados consome o maior tempo.
    - Mais rápida → mais cara(monetário)
    - A memória trabalha de acordo com dois princípios de localidade:
        - **Tempora**l: um dado acessado tende a ser acessado novamente em breve.
        - **Espacial**:se um dado foi acessado, dados próximos devem ser acessados em breve.

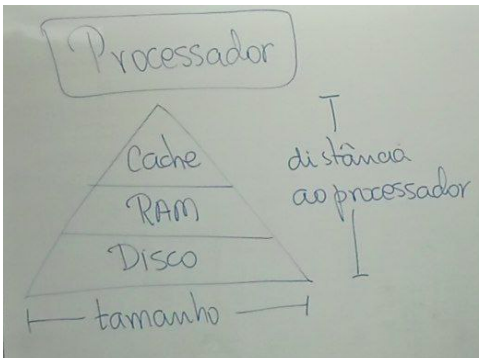
    Esses princípios sugerem uma **organização hierárquica** de uma memória.

    Processador: quanto mais próximo do computador mais rápido é de trazer o dado e mais caro se torna. Ex: cache é a memória mais próxima e mais rápida mas é extremamente cara por isso só tem 5MG em média. Quanto mais barata coloca em maior quantidade e quanto mais cara em menor quantidade no hardware.

    - Os níveis hierárquicos são divididos de tal forma que:
        - O custo-benefício seja o mesmo nos níveis e a memória mais rápida fique mais próxima ao processador.
    - Conceitos
        - Os dados são mantidos no nível mais baixo e as cópias são feitas apenas entre os níveis entre níveis adjacentes.
        - O processador acessa dados apenas do nível mais alto.

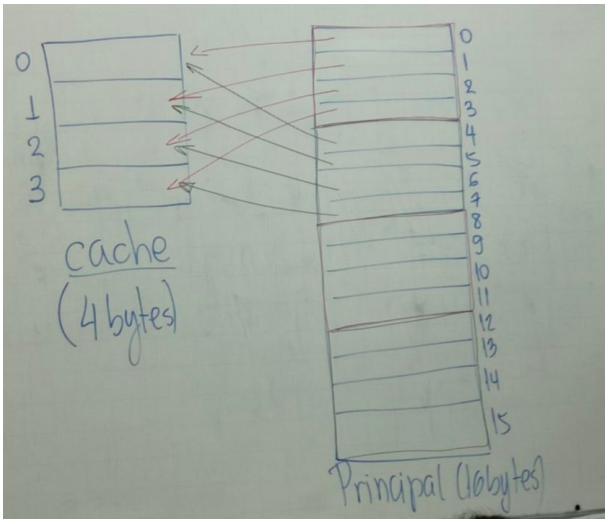
- Se um dado requisitado estiver no nível mais alto, dizemos que houve um **acerto.** Caso contrário, houve uma **falha.**
- Taxa de acerto (taxa de falha) é a fração de acessos, que resultam em acerto (falha).
- Tempo de acerto (penalidade de falha) é o tempo para acessar um dado quando houve acerto(falha).
- SRAM →cache *static* ($5000 por GB)
  DRAM→RAM *dinamic* ($20 por GB)
  FLASH →HD ($5 por GB)

● Em um processador temos 3 níveis de hierarquia de memória:



→obs: temos muita memória disco em nossas máquinas e menos memória RAM e muito menos ainda memória cache por conta dos altos valores financeiros.
→Um processador solicita um dado pelo **endereço.**
→Cada dado na memória deve possuir um endereço de memória **único.**Esse endereço corresponde ao de maior unidade, no caso disco.
→É melhor buscar esses endereços pela memória **Disco** pois ela é maior e com certeza teria todos os endereços.
→ Um dado é transferido apenas entre níveis adjacentes (Cache vai para RAM e RAM para cache, assim como disco para RAM e RAM para disco…nunca de disco para Cache e vice versa). Logo um dado vai para o processador apenas da memória **Cache.**
→Objetivo da memória hierárquica é maximizar **a taxa de acerto**(Usando os princípios de localidade).

● Quando o processador solicita um dado, como saber se ele está presente na cache ?
● Mapeamento de endereços Cache ←→Memória Principal
→Por via de regra, a memória cache sempre será menor que a principal.



- Dado um endereço X da memória principal, como saber qual endereço será mapeado na memória cache ?
  → y = x % 4, ou genericamente y = x % c, onde c é o tamanho da cache.Olhamos o resto.
- Dado um dado com endereço na cache, como saber qual o endereço ele ocupa na memória principal ?
  →fatia = x // C (divisão inteira)
  →Endereço na memória principal = C*fatia + linha

bit de validade	tag (fatia da memória principal)	dado (blocos de 8 bytes)
0		| | | | | | |
1		| | | | | | |
2		| | | | | | |
3		| | | | | | |

memória principal = 2^t bytes
memória cache = 2^n bytes

Endereço de memória principal:

tag(t-n bits) - Fatia	linha(n bits)
t bits ——----------	——----------

- Resumo -

- Mapeamento associativo
  →Endereço da memória principal (x) → y = x % C, onde C é o tamanho da memória cache.
  x = tag(fatia) * C + linha…onde tag são os dois números mais importantes da memória e a linha os dois últimos da memória cache.

Achar a posição na memória principal a partir da matriz (cache):
p(i,j) na matriz(m,n), dado o tamanho da matriz e a posição na matriz
tamanho da memória princial = m*n
posição = posicaoLinha(i)*tamColunas(n) + posicaoColuna(j)

Achar a posição na matriz(cache) a partir da memória principal:
p(i,j) na memória com n posições, dado p
i = p/n
j = p%n

——----------

Exemplo:

cache tem 4 linhas => 2^n = 2^2
memória principal tem 128 bytes => 2^t = 2^7
bloco tem 8 bytes => 2^b = 2^3

Agora, dado uma posição p=60 da memória principal de 128 bytes:
fatias de 32 bits
60 / 32 = 2, está na 2º fatia (32 a 63 bits)
60 % 32 = 28
para mapear numa matriz de 3x8
i = 28 / 3 = 3
j = 28 % 8 = 4
p(3,4) = 60

Com a fórmula:
coluna cache = p % 2^b, em que os blocos tem 8 bytes resultando em 2^3, b = 3
“ = 60 % 8 = 4
linha cache = (p % 2^n+b) / 2^b = (60 % 32) / 8

Memória Cache:

		000	001	010	011	100	101	110	111
00									
01									
10									
11						x			

Outra forma:
Em binário 60 é 00111100
Endereço de memória principal -

tag = 001	linha = 11	bloco(coluna) = 100
t-n-b bits = 2	n bits = 2	b bits = 3

Para fazer o mapeamento:

1. Determinar n, t e b (a partir dos tamanhos da memória cache, principal e bloco)
2. Segregar o end memória principal: [tag (t-n-b) | linha(n) | coluna(b)] (t bits)

Resumo do mapeamento direto:

Endereço memória principal

tag	linha cache	bloco
t-n-b bits	n bits	b bits

obs: tudo isso equivale a t bits.

- Tamanho real da cache
  →Quando dizemos o tamanho de uma memória cache, nos referimos ao total de dados que ela pode armazenar.

→ Para calcular o tamanho real de uma cache, é necessário incluir o bit de validade e a tag. Logo, para determinar esse tamanho:

1) Determinar o tamanho, em bytes (i) da memória principal e (ii) de um bloco da cache.Por último, determinar o total de linhas da cache.(Em outras palavras queremos determinar **t,n** e **b)**.

2) Determinar o tamanho da **tag = t - n - b (bits)**

3) **Tamanho real da cache = 2^n.(1+tag+dados por linha) → O ideal é converter tudo para bits para pois pois a tag e o bit de validade já vem em bits…então melhor converter os dados para bits também e se necessário converter o tamanho real da cache para gigas ou o que for necessário.**

Obs: São dados: O tamanho(dados) da cache, o tamanho de um bloco e o tamanho da memória pirncipal.

Lembrando que:

- 1 byte = 8 bits
- 1 Kib = 2^10bytes
- 1 mib = 2^10 kib = 2^20 B
- 1 Gib = 2^10 mib = 2^20 kib = 2^30 B
- 1 Tib = 2^10Gib = 2^20 mib = 2^30 Kib = 2^40 B

- Quantos bits são necessários para uma memória cache diretamente mapeada com 16 Kib de dados e blocos de 32 B(ou seja 32 colunas), considerando que a memória principal possui 4 Gib ? **Boa questão de prova**

  1) 2^t : tamanho da memória principal (B)
     → 4Gib = 4.2^30 =2^2.2^30= 2^32 B **concluimos t = 32**
     2^b:bloco (B)
     →32 B = 2^5 B **concluimos b = 5**
     2^n:total de linhas da cache
     →16Kib/32B = (2^4 . 2^10B)/2^5 B = 2^9 linhas **concluimos n = 9**
     16 Kib = quantidade de dados
  2) tag = t - n - b
     tag = 32 - 9 - 5 = **18 bits**
  3) Tamanho real = 2^9 .(1+18+2^(5+3))
     = 2^9(1+18+256)
     = 2^9(275)
     = 275.2^9 bits
     = (275.2^9)/(2^3 . 2^10) Kib
     =275/2^4 Kib
     = **17,18 Kib** → Muito próximo aos 16 Kib como esperado

  obs: ele teria que dar muito próximo a estes 16 kib, se der muito longe disso algo está errado