

## ② Lei de Amdahl

$$\text{Speed-up} = \frac{1}{(1-p) + \frac{p}{N}}$$

p - fração paralela.

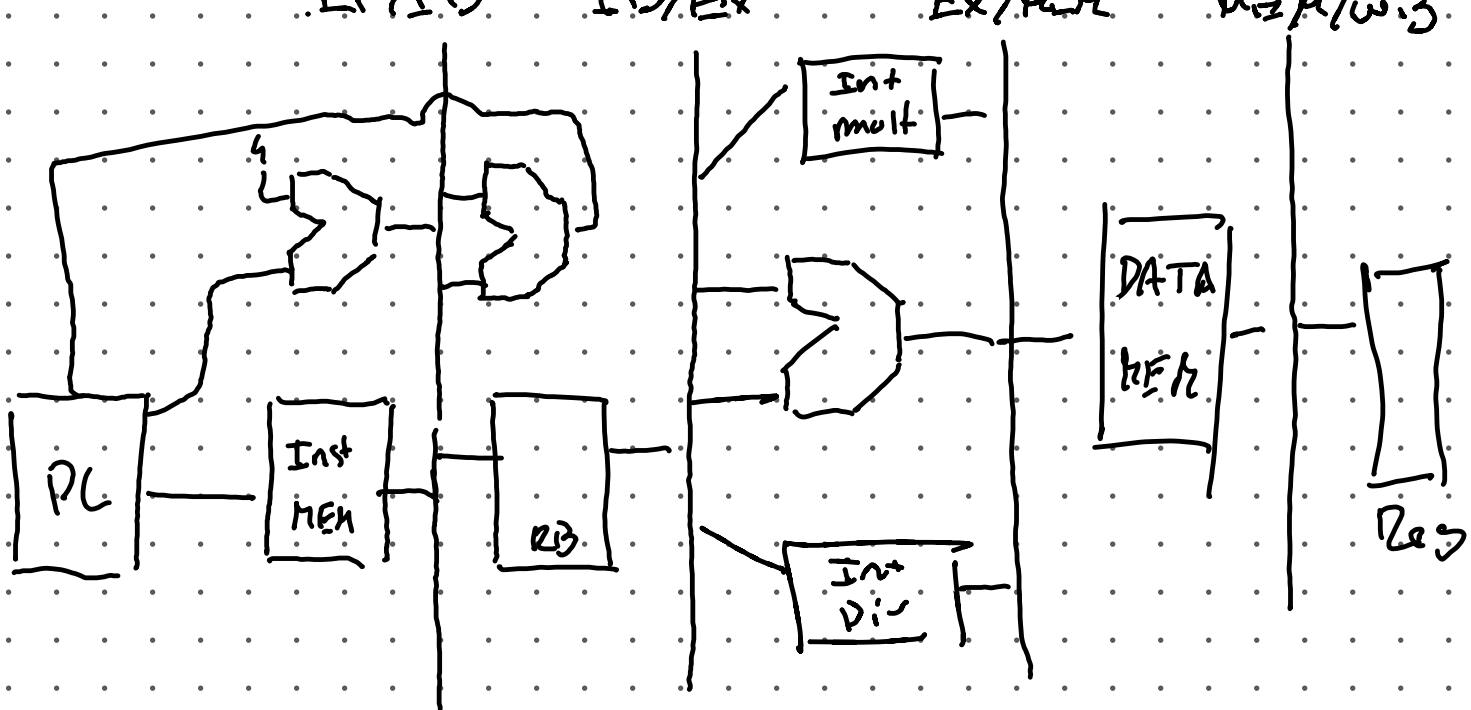
N - fator de otimização

A lei de "diminishing returns" diz que com o aumento do fator de paralelização obtinguemos um máximo no fator de Speed-up. Isto acontece porque as paralelizadas instruções têm um overhead na execução da tarefa. Apesar de executarmos as sub-tasks mais rapidamente o overhead total vai aumentar (overhead tem tempo constante).

③ Uma estrutura pipeline consegue subdividir uma "super-task" em K sub-tasks, ou estados, que executam tarefas independentes mas demoram o mesmo tempo. Neste caso pipelining uma instrução é largada para execução a cada ciclo de reloj e também são finalizadas a cada ciclo (executa a inicial). Dito isto non-pipelined demora  $n \times K$  enquanto que pipelined demora  $K + n - 1$ .

Os hazards são estruturais (conflieto de uso do mesmo hardware ao mesmo tempo, por 2 instruções), hazards de dados (em que uma instrução i+1 usa um valor ainda por obter/calcular na instrução i), hazard de controle em que o cálculo de uma operação branch necessita de um ciclo extra. A 1º é resolvida usando 2 memórias (1 para dados e outra para inst), visto que é o único elemento que pode ser requerido duas vezes no mesmo tempo, para além disso também necessitamos de slots para pipelines que usam FP units, a segunda é resolvida usando forward do estado KEM  $\rightarrow$  KEM, KEM  $\rightarrow$  EX e EX  $\rightarrow$  EX, o 3º é resolvido usando branch delayed slot, método que consiste em inserir uma instrução independentemente do resultado de branch, depois da instrução branch.

(4) uma instrução de multiplicação é pipelined porque a multiplicação é comutativa, logo, podemos calcular independentemente. A divisão não é e, por isso, necessita de ser sequencial.



Unha arquitectura pipelined non é capaz de executar inst. out of order porque non ha scheduling e para isso executar inst. fora de orden crean denominados hazards, principalmente a de datos.

$$\textcircled{1} \quad 3674_{10}, 5932_{10} \quad w = 32 \text{ bits}$$

$$\begin{array}{r}
 3674'4'6'116 \\
 -32 \\
 \hline
 47 \quad -16 \quad 14 \quad 16 \\
 -32 \\
 \hline
 154 \quad -24 \quad (14) \quad 1 \\
 -144 \\
 \hline
 10
 \end{array}$$

$$16 \times 9 = 144$$

$$16 \times 4 = 64$$

$$16 \times 3 = 48$$

$$16 \times 1 = 16$$

$$3674_{10} = 0x0000\text{E5A}_{16}$$

$$\begin{array}{r}
 5932'1'16 \\
 -48 \\
 \hline
 113 \quad -32 \quad (18) \\
 -112 \\
 \hline
 12 \quad -48 \quad (1) \quad 0
 \end{array}$$

$$5932_{10} = 0x0000\text{172C}_{16}$$

R: Ambos son alinhados porque acaba en 00.

(5) Usamos associatividade (direto, 2-way, ...). Desta modo, conseguimos associar que  $k$  endereços de memória mapeiam para  $k$  endereços de cache ( $k = \text{associatividade}$ ). Implementamos políticas de replacement (LRU, FIFO, Random), deste modo, generalizamos que trocas constantes de endereços no mesmo set  $k$  são mitigados.

Têm que ser mais elaborado