Apontamentos de ACA

Versão “Tenho de safar o exame em 8 horas”

v 1.0

Paulo Vasconcelos
fevereiro 2020

Disclaimer:

Estes apontamentos foram criados por forma a tentar auxiliar as pobres almas que se lixaram à força toda no semestre mais complicado do MIECT. Por favor, não tomem este documento como único material de estudo se pretendem obter uma nota minimamente decente.

Para os interessados em aprofundar mais o seu conhecimento sobre a cadeira ou que simplesmente queiram rever alguns conceitos relevantes que possam estar menos presentes na memória, é deixado no final de cada capítulo uma recomendação bibliográfica complementar ao conteúdo abordado no capítulo. Estas recomendações bibliográficas foram retiradas diretamente dos materiais fornecidos pelo Professor António Rui Borges.

Dito isto, bom estudo e boa sorte.

Paulo Vasconcelos
fevereiro 2020

Paulo Vasconcelos
fevereiro 2020

Capítulo 1 – Computer Abstractions and Technology

Definições:

Computer Architecture – Atributos de um sistema visíveis para o programador. Inclui o *instruction set*, o número e tamanho dos registos internos do processador, os formatos dos diferentes tipos de *data*, os modos de endereçamento da memória e os mecanismos de I/O.

Computer Organization – Organização interna de um sistema, não visível para o programador. Inclui os sinais de controlo, as interfaces entre componentes do sistema (como processador, memória e I/O) e a tecnologia usada na memória.

Stored-program – Tratamento das instruções a executar por um processador como informação, permitindo assim que sejam armazenadas em memória. Base da arquitetura von Neumann.

RISC - *Reduced Instruction Set Computers*. Computadores com design focado no aproveitamento do *instruction level parallelism* (primeiro com *pipelining* e, depois, com lançamento simultâneo de múltiplas instruções) e aplicação sistemática de *caches* para melhorar os tempos de acesso do processador a informação e instruções.

Um computador, por se tratar de um sistema extremamente complexo, requer vários níveis de abstração. Para o contexto de cada nível, é necessário estar consciente da estrutura (componentes existentes e ligações entre si) e função (operação de cada componente e sua contribuição para o todo).

Funções básicas de um computador:

- Processamento de informação;
- Armazenamento de informação;
- Controlo;
- Transferência de informação (interna, memória, ou externa, I/O).

A Lei de Moore diz que o número de *transistors* que é possível colocar num chip duplica a cada ano. Desde que a observação foi feita, em 1965, o ritmo abrandou para duplicar a cada dois anos e meio.

Estrutura e função (base) de um computador:

- Processador - Controlo do sistema;
- Memória principal - Armazenamento durante processamento, volátil;
- Memória de massa - Armazenamento entre processos, maior, não volátil;
- I/O - Transferência de informação entre o sistema computacional e o exterior;

- *System interconnection* - *Bus* para transferência interna de informação.

Tipos de instruções presentes tipicamente num *instruction set*:
- Transferência de informação - Entre registos e memória ou controlador de I/O;
- Aritmética/lógica - *add*, *sub*, …, *shift* e *rotate*;
- Salto - Condicional ou incondicional;
- Invocação de subrotinas - Condicional ou incondicional.

Diferentes mercados para sistemas computacionais:
- Dispositivos móveis pessoais - Dispositivos *wireless* como smartphones e tablets;
- *Desktop computing* - Portáteis e *workstations*;
- Servidores;
- *Clusters*/*warehouse-scale computers* - Supercomputadores;
- *Embedded* - Pensados para correr *firmware* próprio. Preço reduzido.

É possível haver dois tipos de paralelismo ao nível de aplicações: paralelismo ao nível de informação e ao nível de tarefas. Estes podem ser explorados pelo hardware das seguintes formas:
- *instruction-level parallelism*;
- *thread-level parallelism*;
- *request-level parallelism*;
- hardware especial/dedicado - GPUs, por exemplo.

Classificação de Michael Flynn:
- SISD - *singlecore*;
- SIMD - arquitetura vetorizada, GPUs;
- MISD - nenhuma aplicação comercial;
- MIMD - *multicore*.

O desenho de sistemas computacionais deve:
- Aproveitar paralelismo;
- Aproveitar o princípio da localidade - reutilização de instruções e informação recentemente utilizadas;
- Focar os casos de utilização mais frequentes.

$$disponibilidade = \frac{t_{disponível}}{t_{total}} = \frac{t_{disponível}}{t_{disponível} + t_{recuperação}} = \frac{MTTF}{MTTF+MTTR}$$

$$speedup_{total} = \frac{t_{sem\ melhoria}}{t_{com\ melhoria}} = \frac{1}{(1-frac_{melhorada}) + \frac{frac_{melhorada}}{speedup_{parcial}}}$$ (Lei de Amdahl)

$$t_{execução} = \sum_i (frac_i * CPI_i) * clock\ cycle\ time$$

Sugestões de leitura:

Computer Architecture: A Quantitative Approach, Hennessy J.L., Patterson D.A., 5th Edition, Morgan Kaufmann, 2012

- Chapter 1: Fundamentals of Quantitative Design and Analysis
- Appendix L: Historical Perspectives and References

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

- Chapter 1: Introduction
- Chapter 2: Computer Evolution and Performance

Capítulo 2 – Instruction-Level Parallelism (Principles)

Definições:

Pipelining - Técnica de implementação que consiste na divisão da execução uma tarefa em subtarefas independentes possibilitando assim operar simultaneamente sobre objetos sucessivos do *stream*. Cada *pipe stage/segment* é executada sequencialmente.

Precise exceptions - Exceções lançadas por uma determinada instrução que permitem às instruções a jusante no *pipeline* terminarem a sua execução permitindo também à própria instrução e todas as que se encontrem a montante serem recomeçadas do zero.

Considerando *T* o tempo de execução de uma tarefa, *N* o número de etapas do *pipeline*, *t* o tempo de execução de cada *pipe stage*, *M* o número de tarefas executadas a tender para infinito e $T \simeq N * t$:

$$t_{non-pipelined} = M * T$$

$$throughput_{non-pipelined} = \frac{1}{T}$$

$$t_{pipelined} = (N - 1 + M) * t \simeq M * t$$

$$throughput_{pipelined} = \frac{1}{t}$$

$$speedup_{pipeline\ com\ N\ etapas} = \frac{M*T}{(N-1+M)*t} \simeq \frac{M*N*t}{(N-1+M)*t} = \frac{N}{N-1} = N$$

Nas arquiteturas RISC, as únicas operações que afetam a memória são as operações de *load* e *store*. As únicas operações aplicáveis a informação (presente em registos) são aritméticas e lógicas. Tende a haver poucos formatos de instruções, tipicamente com o mesmo tamanho.

Pipeline clássico de 5 etapas:

1. *Instruction fetch* - IF;
2. *Instruction decode & register fetch* - ID;
3. *Execution* - EX;
4. *Memory access* - MEM;
5. *Write-back* - WB.

Por forma a não haver um *hazard* estrutural derivado de dois acessos à memória simultâneos que ocorrem caso haja um acesso à memória e se tente fazer um *Instruction fetch*, é necessário separar a memória de dados e de instruções em duas memórias separadas.

Para evitar o *hazard* estrutural derivado de dois acessos ao banco de registos em simultâneo (leitura no *Instruction decode & register fetch* e escrita no *Write-back*) a leitura é realizada na primeira metade do ciclo do relógio e a escrita na segunda.

A computação do *Branch Target Address* (BTA) deve ser feita na fase de ID.

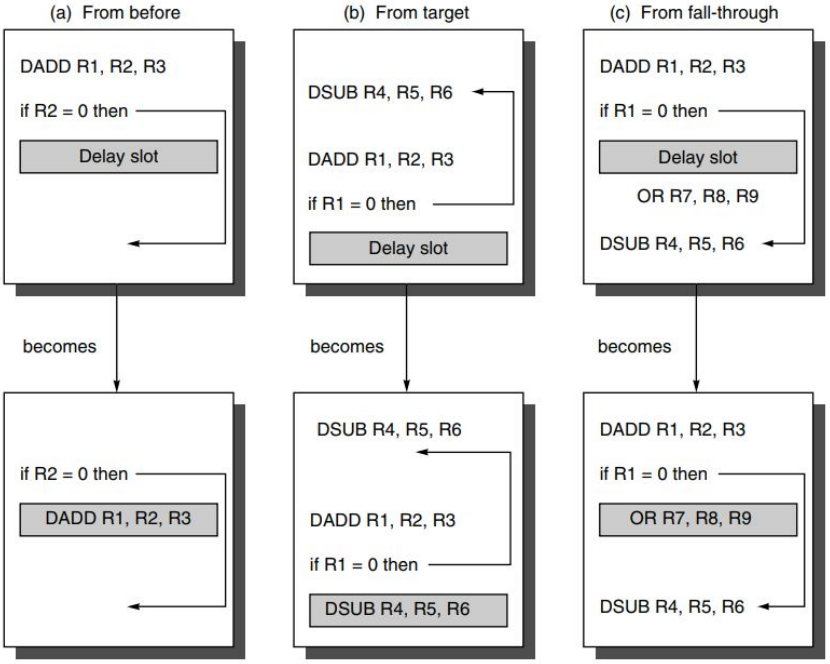
Existem três tipos possíveis de *hazards*:

- *Hazards* estruturais - Conflitos de hardware;
- *Hazards* de dados - Dependências de resultados de instruções prévias;
- *Hazards* de controlo - *Branches*.

Hazards estruturais podem exigir que se faça um *stall* do *pipeline*.

Hazards de dados requerem uma análise cuidada. Há que verificar se o problema pode ser resolvido com *forwarding*. Só caso não possa ser resolvido com *forwarding* (*load* seguido de uma instrução que tente ler do registo escrito) é que é introduzido *stall* no *pipeline* (um ciclo caso seja uma instrução da ALU, duas caso seja um *beq*).

O impacto dos *hazards* de controlo pode ser reduzido através de técnicas de predição de *branch*. Pode ser uma coisa tão rudimentar como prever, independentemente de tudo o resto, que o *branch* vai ser *taken* ou *not taken*. Por outro lado, o compilador pode tentar otimizar o processo usando uma técnica denominada de ***delayed branch***. Esta técnica consiste na existência de um ***branch delay slot*** que deve ser ocupado com uma instrução que não quebre o fluxo do *pipeline*.



Para além disto, pode ainda ser realizada uma predição dinâmica (predição em *runtime*) através de preditores baseados em máquinas de estados. Pode ainda ser usado um preditor de torneio que usa múltiplos preditores com uma máquina de estados extra para tomar a decisão relativa a que previsão (de que preditor) selecionar (funcionando, basicamente, como um mutex de preditores).

Categorias de exceções:

- Síncronas vs Assíncronas - Exceções síncronas ocorrem sempre no mesmo sítio da execução do programa com as mesmas condições (dados e alocação de memória);
- *User requested* vs *Coerced* - Pedidas pelo utilizador ou não;
- *Maskable* vs *Non-maskable* - As exceções são *maskable* se o programa puder escolher escolher o momento no qual lhes dá resposta;
- Dentro vs Entre instruções - Exceções assíncronas dentro de instruções tendem a ser causadas por falhas catastróficas e levam o programa a terminar sempre.
- Resumir vs Terminar - *Aftermath* da exceção.

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Visto que operações de *Floating Point* tendem a demorar múltiplos ciclos de relógio, podemos alterar a implementação tradicional do *pipeline* de 5 etapas para uma com múltiplas unidades funcionais (e instruções que ficam mais de um ciclo na etapa de EX). Com isto, são introduzidos dois novos conceitos ao *pipeline*: **latência** - número de ciclos de relógio necessários entre uma instrução que produza um determinado resultado e uma instrução que utilize esse mesmo resultado; **iniciação** ou **intervalo de repetição** - número de ciclos de relógio necessários entre duas operações do mesmo tipo.

A não ser que sejam dadas indicações em contrário, devem ser considerados os seguintes valores:

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

De notar que uma instrução de *load* tem uma latência de 1 mas uma instrução de *store* tem uma latência de -1.

Esta implementação é mais propícia a *hazards* estruturais (ter em atenção acessos à memória ao completar os ciclos necessários em cada unidade funcional) e *hazards* de dados dado, uma vez mais, as diferentes latências das instruções.

Para evitar *hazards* estruturais sob a forma de múltiplos acessos à memória em simultâneo, pode-se optar por computar o ciclo de relógio no qual vai ocorrer ainda na fase de ID e fazer *stall* em caso de necessidade. Esta implementação necessita de um *shift register* para ir mantendo o registo dos diversos ciclos de acesso à memória de todas as instruções presentes no *pipeline*.

Outra abordagem ao problema passa por fazer *stall* numa das instruções conflituosas assim que esta tenta entra na fase na qual haverá problemas (MEM ou WB). Esta implementação é mais simples em termos de implementação mas implica que a deteção seja feita em dois sítios distintos (antes da fase MEM e antes da fase WB) em oposição à implementação anterior na qual a única deteção feita era na fase ID.

Hazards de dados podem ser resolvidos de qualquer uma de duas formas: atrasar o lançamento da segunda instrução; impedir a primeira instrução de alterar o conteúdo do registo destino. Ambas são aceitáveis dada a raridade da ocorrência.

A MIPS R4000 é uma família de processadores com o *instruction set* MIPS64 mas com um pipeline de 8 etapas (*superpipelining*). Isto permite-lhe atingir um *clock rate* superior através da decomposição dos acessos à memória. Esta implementação aumenta a lógica necessária para implementar *forwarding* mas também altera os atrasos.

- 0 ciclos para *branch not taken*;
- 2 ciclos para *branch taken*;
- 2 ciclos para *load* seguido de uma operação da ALU que use o registo destino.

Para além destes, podem ocorrer ainda *stalls* relacionados com a Float Point Unit por um dos operandos ainda não ter sido computado (*hazard* de dados) ou por as etapas necessárias dentro da Float Point Unit para realizar a operação não estão disponíveis (*hazard* estrutural).

Sugestões de leitura:

Computer Architecture: A Quantitative Approach, Hennessy J.L., Patterson D.A., 5th Edition, Morgan Kaufmann, 2012

Appendix C: Pipelining: Basic and Intermediate Concepts (Sections 1 to 6)

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

Chapter 14: Processor Structure and Function

Paulo Vasconcelos
fevereiro 2020

Capítulo 3 – Memory Hierarchy Design

Definições:

Cache – Nome dado aos níveis da hierarquia de memória localizados entre o processador e a memória principal.

Cache line – Quantidade mínima de informação possível de transferir entre dois níveis de *cache* ou armazenada ao mais baixo nível da *cache*.

Block – Quantidade mínima de informação que pode estar ou não presente numa hierarquia de dois níveis.

Hit rate/ratio – Fração das referências de memória encontradas no nível superior (*hit*).

Miss rate/ratio – Fração das referências de memória não encontradas no nível superior (*miss*).

Hit time – Tempo para aceder ao nível superior de memória, verificando se se trata de um *hit* ou um *miss*.

Miss penalty – Tempo para trocar o bloco em causa no nível superior de memória.

Memory access time – Tempo entre o *issue* de um pedido de leitura ou escrita o instante em que a informação fica disponível ou é armazenada.

Memory cycle time – Tempo de intervalo mínimo entre pedidos.

Princípio de otimalidade (*principle of optimality*) – Escolher a linha do grupo cujos dados não serão mais referenciados ou, se forem, que serão referenciados daí a mais tempo.

Tipos de **localidade**:

- **Espacial** - Referenciar informação próxima (em termos de endereço de memória) em breve;
- **Temporal** - Referenciar informação novamente em breve.

Existem vários níveis de *cache* para que seja possível apresentar ao programador o máximo de memória possível (*cache L3*) ao mesmo tempo que se fornece acesso à mais rápida velocidade (*cache L1*).

Hierarquia de memória:

- Banco de registos;
- *Cache*;
- Memória principal (RAM);
- *Swapping area*.

Tipos de organização de *cache*[1]:

- **Mapeamento direto (*Direct mapped*)** - Cada bloco tem apenas um sítio na cache no qual pode ser colocado;
- **Mapeamento associativo (*Fully associative*)** - Cada bloco pode ser colocado em qualquer sítio;
- **Mapeamento associativo por conjunto (*Set associative*)** - Cada bloco tem um conjunto de posições (*set*) no qual pode ser colocado.

Para além do conteúdo da memória principal (a informação em si), uma *cache line* contém também uma *tag field* (usada para verificar se há correspondência aquando de um acesso à memória) e um *bit* de validação. Pode ainda haver um *dirty bit* caso a política de escrita seja *write-back* e mais algum/alguns bit(s) relativos à política de substituição caso esta seja LRU ou FIFO.

Estratégias para seleção/substituição de uma linha da *cache*[2]:

- ***Random***;
- ***Least Recently Used (LRU)***;
- ***First In, First Out (FIFO)***.

Políticas de escrita:

- ***Write-through*** - Os dados são escritos para a linha da *cache* e para a memória imediatamente abaixo na hierarquia de memória (*cache* L2 escreve na *cache* L3, *cache* L3 escreve na memória principal, ...). Mais simples de implementar, maior garantia de coerência de dados com a desvantagem que uma operação de escrita passa a demorar mais tempo (passa a demorar o tempo de escrita para a memória principal em vez do tempo de escrita para a *cache*);
- ***Write-back*** - Os dados são escritos para a linha da *cache*. Só são escritos para a memória quando o bloco é substituído. Sem atrasos desnecessários mas com menor robustez.

Na política de ***write-through***, por forma a evitar o *stall* do processador durante o processo de escrita para a memória mais lenta, pode ser implementado um ***write buffer*** que permite a continuação normal do funcionamento do processador assim que a escrita é feita na *cache* passando a realizar a escrita para memória em simultâneo com a execução do processador.

Um endereço de memória na *cache* divide-se em *block address* e *offset* sendo que o *block address*, por sua vez, se pode subdividir em *tag field* e *index field*[3]. Verificam-se as seguintes condições[4]:

---

[1] Nota do autor: No caso da matéria relativa aos tipos de organização de *caches*, dada no segundo ano do curso, estar esquecida, recomendo vivamente o Capítulo 4 do livro de Stalling.

[2] Estas estratégias só se aplicam a *caches* com uma organização do tipo *Fully associative* ou *Set associative*. No caso da organização *Direct Mapping*, como só existe uma linha da *cache* na qual o bloco pode ser colocado, o processo de substituição de uma linha na *cache* é, basicamente, verificar se o bloco que está nessa linha é o pretendido e, caso não seja, trocar.

[3] A divisão do *block address* em *tag field* e *index field* verifica-se caso a organização da *cache* seja do tipo *direct mapping* ou *set associativity*. No caso de ser *fully associativity* o *block address* corresponde ao *tag field* (não existe *index field*).

[4] sib = *size in bits*; siB = *size in bytes*

$$offset\ sib = log_2(block\ siB) \equiv block\ siB = 2^{offset\ sib}$$
$$block\ address\ sib = memory\ address\ sib - offset\ sib$$
$$memory\ positions\ in\ main\ memory = 2^{memory\ address\ sib}$$
$$blocks\ in\ main\ memory = 2^{block\ address\ sib}$$
$$index\ field\ sib = log_2(\frac{cache\ siB}{set\ associativity * block\ siB})$$

De notar que a última expressão se mantém válida para qualquer modo de organização de *cache*.[5]

Tipos de *cache misses*:

- ***Compulsory misses*** - *Misses* inevitáveis que ocorrem na primeira vez que se tenta aceder a um determinado pedaço de informação (seja *byte* ou *word*);
- ***Capacity misses*** - *Misses* existentes em *fully associative caches* causados por tamanho insuficiente da *cache* o que leva a que um pedaço de informação seja descartado (para dar lugar a outro que tenha sido chamado) e depois tenha de ser novamente copiado da memória para poder voltar a ser usado;
- ***Conflict misses*** - *Misses* causados pela organização interna da *cache* (demasiados blocos mapeados para o mesmo set). Não aplicado a uma *fully associative cache*.

Sugestões de leitura:

Computer Architecture: A Quantitative Approach, Hennessy J.L., Patterson D.A., 5th Edition, Morgan Kaufmann, 2012

Chapter 2: Memory Hierarchy Design
Appendix B: Review of Memory Hierarchy

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

Chapter 4: Cache Memory
Chapter 5: Internal Memory
Chapter 6: External Memory

[5] Como foi dito acima, no caso de a organização da *cache* ser do tipo *fully associativity* o *index field* não existe (tem tamanho igual a 0 bits). Isto porque, ao preencher esta expressão, deve considerar-se o *set associativity* igual a 1 no caso de *direct mapping* e igual a *block siB / cache siB* no caso de *fully associativity*. No caso de uma *cache* organizada segundo o modo *set associativity* o valor da variável deve ser indicado no enunciado.

Paulo Vasconcelos
fevereiro 2020

Capítulo 4 – Instruction-Level Parallelism (Complements)

Definições:

Basic block (of a program) – Segmento de código sem nenhum *branch in* sem ser o ponto de entrada e nenhum *branch out* sem ser o ponto de saída.

Tournament predictor – Preditor de *branch* que usa um preditor local e um global juntamente com a sua própria máquina de estados por forma a fazer melhores previsões que qualquer um dos outros preditores usados individualmente.

Static scheduling – Controlo/ajuste do lançamento de instruções por software (antes da execução do programa). Uma otimização feita por um compilador é considerado *static scheduling*.

Dynamic scheduling – Controlo/ajuste do lançamento de instruções por hardware (durante a execução do programa). de forma a minimizar stall que ocorrem de Data Hazards.

reoordenar instruções

Tipos de *data hazards*:

- **RAW** - Ler um operando antes de ele ter sido escrito por uma instrução anterior;
- **WAW** - Escrever um operando antes de uma operação de escrita anterior o escrever, ou seja, a operação que devia ter escrito em primeiro lugar vai ser a última a escrever e o valor que vai ser guardado não vai ser válido. Este *hazard* apenas é possível em *pipelines* que permitam conclusão de instruções fora de ordem;
- **WAR** - Escrever um valor antes de este ser lido. Como no *hazard* anterior, este também não ocorre em qualquer *pipeline*. Neste caso, o *hazard* só ocorre em *pipelines* nos quais seja possível escrever resultados numa das primeiras fases do *pipeline*.

tipos de Hazards só em pipilines com suporte para conclusão de instruções fora de ordem

Vantagens do *dynamic scheduling*:

- Código compilado pode ser executado noutra máquina (*pipeline*, possivelmente, diferente) sem haver problema pois o próprio hardware faz o ajuste do lançamento das instruções aos recursos que tem disponíveis;
- Gestão de dependências desconhecidas aquando da compilação (como referências de memória);
- Adaptação a imprevistos (como *cache misses*).

Scoreboarding:

1. *Issue*;
2. *Read operands*;
3. *Execution*;

4. *Write result* - Antes de completar, verifica se há WAR.

Algoritmo de Tomasulo:

1. *Issue*;
2. *Execute*;
3. *Write result*.

[Ver slides 4-43 a 4-94]

Scoreboarding:

- *Issue in-order*;
- *Completion out-of-order* derivado de múltiplas unidades funcionais com diferentes números de ciclos de relógio necessários para o processamento.

  http://bnrg.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture10.pdf

Tomasulo:

- *Issue in-order*;
- *Completion out-of-order* derivado de múltiplas unidades funcionais com diferentes números de ciclos de relógio necessários para o processamento.

  https://people.eecs.berkeley.edu/~pattrsn/252F96/Lecture04.pdf

  https://www.youtube.com/watch?v=y-N0Dsc9LmU

  https://www.youtube.com/watch?v=YH2fFu-35L8

ao Algoritmo de Tomasulo

A implementação de ***hardware speculation*** acrescenta a fase de ***commit*** e o ***reorder buffer***. Esta técnica permite um melhor aproveitamento do processador aquando de uma instrução de *branch* pois continua a execução do programa normalmente mas não faz *commit* do resultado dessas instruções sem ter a certeza de que acertou na sua predição de branch.

Tipos de *multiple issue processors*:

- *statically scheduled superscalar processors*;
- *dynamically scheduled superscalar processors*;
- *very long instruction word processors*.

Sugestões de leitura:

Computer Architecture: A Quantitative Approach, Hennessy J.L., Patterson D.A., 5th Edition, Morgan Kaufmann, 2012

Chapter 3: Instruction-level Parallelism and its Exploitation (Sections 1 to 11 and 13)

Appendix C: Pipelining: Basic and Intermediate Concept (Section 7)

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

Chapter 16: Instruction-Level Parallelism and Superscalar Processors

Capítulo 5 – Data-Level Parallelism

Definições:

Stride – Distância que separa dois elementos adjacentes que se pretende inserir em *vector registers*.

Sparse matrices – Matrizes em que a maioria dos seus elementos são 0.

Variações de SIMD*s*:

- ***Vector architectures***;
- ***Multimedia instruction set extensions***;
- **GPUs**.

Principais componentes da arquitetura **VMIPS**:

- ***Vector registers*** - Registos vetoriais. Basicamente iguais aos registos da arquitetura MIPS a que estamos acostumados mas usados para cálculo vetorial;
- ***Vector functional units*** - Unidades funcionais vetoriais. Unidades funcionais capazes de realizar operações em vetores de dados. Estas unidades são também capazes de realizar operações sobre escalares;
- ***Vector load/store units*** - Tal como nos anteriores, as unidades de *load* e *store* apresentam muitas semelhanças às da arquitetura MIPS sendo a única diferença a capacidade de operar sobre a memória com vetores. Para isto, é transferida, de forma *pipelined*, uma *word* por ciclo de relógio. Estas unidades são também capazes de realizar operações sobre escalares;
- ***Scalar registers***.

Passem uma vista de olhos por isto:
http://taco.cse.tamu.edu/classes/614/CAQA6e_ch4.pptx

Operações sobre matrizes esparsas (*sparse matrices*) são muitas vezes computadas através de operações *gather-scatter* com o auxílio de um vetor de índices. Estas operações podem ser vistas como três passos:

1. Fazer o *gather* dos dados - Comprimir os dados num vetor mais pequeno contendo apenas os valores diferentes de zero. Isto é possível de se fazer através do vetor de índices que contém a referência para as posições com valores diferentes de zero;
2. Operar vetorialmente sobre os dados - Operação que era inicialmente pretendida (aplicada de forma vetorial);
3. Fazer o *scatter* dos dados - Descomprimir os dados, enviando-os de volta para as posições iniciais usando, novamente, o vetor de índices.

Sugestões de leitura:

Computer Architecture: A Quantitative Approach, Hennessy J.L., Patterson D.A., 5th Edition, Morgan Kaufmann, 2012

Chapter 4: Data-Level Parallelism in Vector, SIMD and GPU Architectures

Computer Organization and Architecture: Designing for Performance, Stalling W., 9th Edition, Prentice Hall, 2013

Chapter 17: Parallel Processing (Section 7)

Apêndice A – Correção do Exame (Época Normal, 2020)

1. *A give processor presents a 44-bit address bus. What is the maximum number of bytes and 64-bit words, expressed approximately as a power of 10, it may address? Justify your claim in detail. (2 points)*

44-bit address permite o armazenamento de 2^{44} bytes
64-bit word = 8 bytes = 2^3 bytes

$$\frac{2^{44}}{2^3} = 2^{41}$$

$$2^{44} = 2^4 * (2^{10})^4 \simeq 2^4 * (10^3)^4 = 1.6 * 10^{13}$$

$$2^{41} = 2^1 * (2^{10})^4 \simeq 2 * (10^3)^4 = 2 * 10^{12}$$

2. *In the 1960s, Michael Flynn studied the parallel computing efforts made so far and found a classification that is still popular today. Which are the characteristics of computer systems he has used to base his classification? How did he name the different categories? Take as an example a quad-core processor. In which category would you place it? (2 points)*

(slide 1-36)
Michael Flynn considerou o paralelismo nas instruções e nos streams de dados e criou 4 categorias:

- SISD (Single Instruction, Single Data)
- MISD (Multiple Instruction, Single Data)
- SIMD (Single Instruction, Multiple Data)
- MIMD (Multiple Instruction, Multiple Data)

Single | Instruction, Single | Data
Multiple | Multiple |

gpu
processadores desktop comuns

Um processador quad-core insere-se na última categoria, Multiple Instruction Multiple Data.

3. *Pipelining is a technique universally used nowadays to speed up instruction execution in a processor. Explain in detail how it works. (1.5 points)*

(slide 2-3)
Pipelining é uma técnica de implementação na qual se divide uma tarefa em subtarefas por forma a que as subtarefas possam ser executadas em simultâneo em vários objetos diferentes.
Apesar desta técnica aumentar o tempo de processamento de cada instrução individual, o throughput de um processador com pipelining passa de 1/T (sendo T o tempo de execução de uma instrução numa implementação non-pipelined) para 1/t (sendo t o um ciclo de clock da versão pipelined) o que se traduz num speedup de até N (sendo N o número de etapas do pipeline).

4. *Dealing with exceptions in a pipelined implementation of a processor is a lot harder than in a non-pipelined implementation. Why it is so? (1.5 points)*

(slide 2-78)
Há dois tipos de exceção que faz sentido considerar ao analisar esta questão: exceções within e between.
Exceções between: num pipeline, ao surgir a exceção, a mesma é ativada após efetuar o write back.
Exceções within: todas as etapas do pipeline, com exceção do write back, podem gerar exceções, ou seja, podem surgir várias em simultâneo. Devem ser colocados nops nas etapas a montante da instrução que lançou a exceção exceção mais a jusante no pipeline e na própria, completando as que estiverem a jusante. Completion out of order é um problema (caso de múltiplas unidades funcionais). Para solucionar, deve ser feito agendamento dinâmico com especulação (buffer de reordenação).
*falar de exceções precisas e exceções imprecisas

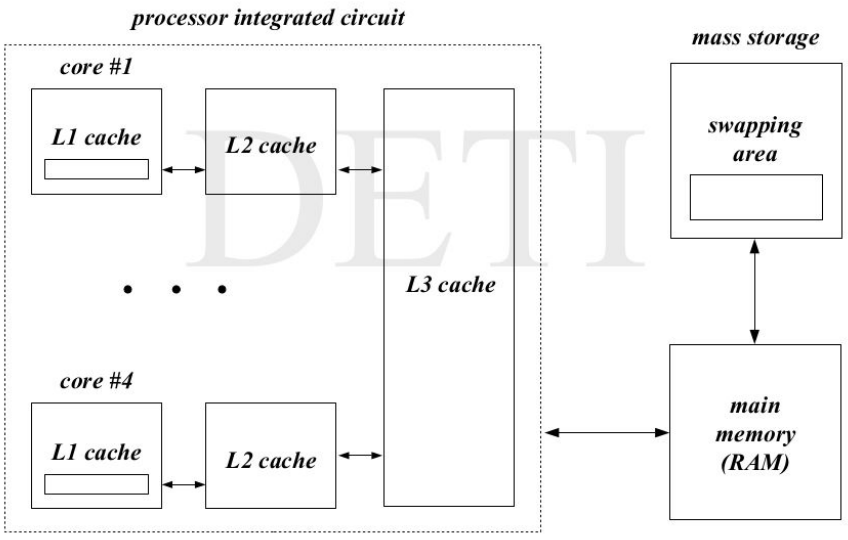
5. *Explain why data hazards of the type WAW can never happen in a processor implementing the 5-stage classical pipeline with a single integer execution unit. (1 point)*

(slide 4-14)
Um pipeline clássico de 5 andares faz static scheduling e tem apenas uma unidade funcional pelo que uma instrução nunca pode "ultrapassar"[6] outra.

instruções são executadas sequencialmente e demoram todas o mesmo número de ciclos para concluir (5 ciclos)

6. *The diagram below depicts the cache hierarchy for a multicore processor. (2 points)*

Typical memory hierarchy for a 4-core processor system



[6] Por "ultrapassar uma instrução" entenda-se "completar antes de uma instrução que tenha sido lançada antes".

a. *Why three cache levels are typically used?*

L3 -> unificação do acesso à memória principal (e sincronismo entre cores)
L2 -> cache maior que L1 com um bus de dados grande mas que não se encontra no die do core
L1 -> cache ultra rápida presente no próprio die do core

b. *Why level 1 is usually divided in an instruction and a data cache?*

(slide 2-29)
Evitar hazard estrutural no acesso à memória simultâneo que pode ocorrer quando uma instrução está na fase de MEM e outra na fase de IF.

c. *What kind of write policy is usually applied to them?*

L1 e L2 -> write-through (mecanismo de sincronização - há que garantir coerência entre cores portanto a informação tem de ser propagada até L3)
L3 -> write-back (atraso incomportável caso fosse write-through - tempo de acesso à memória principal é de outra ordem de grandeza)

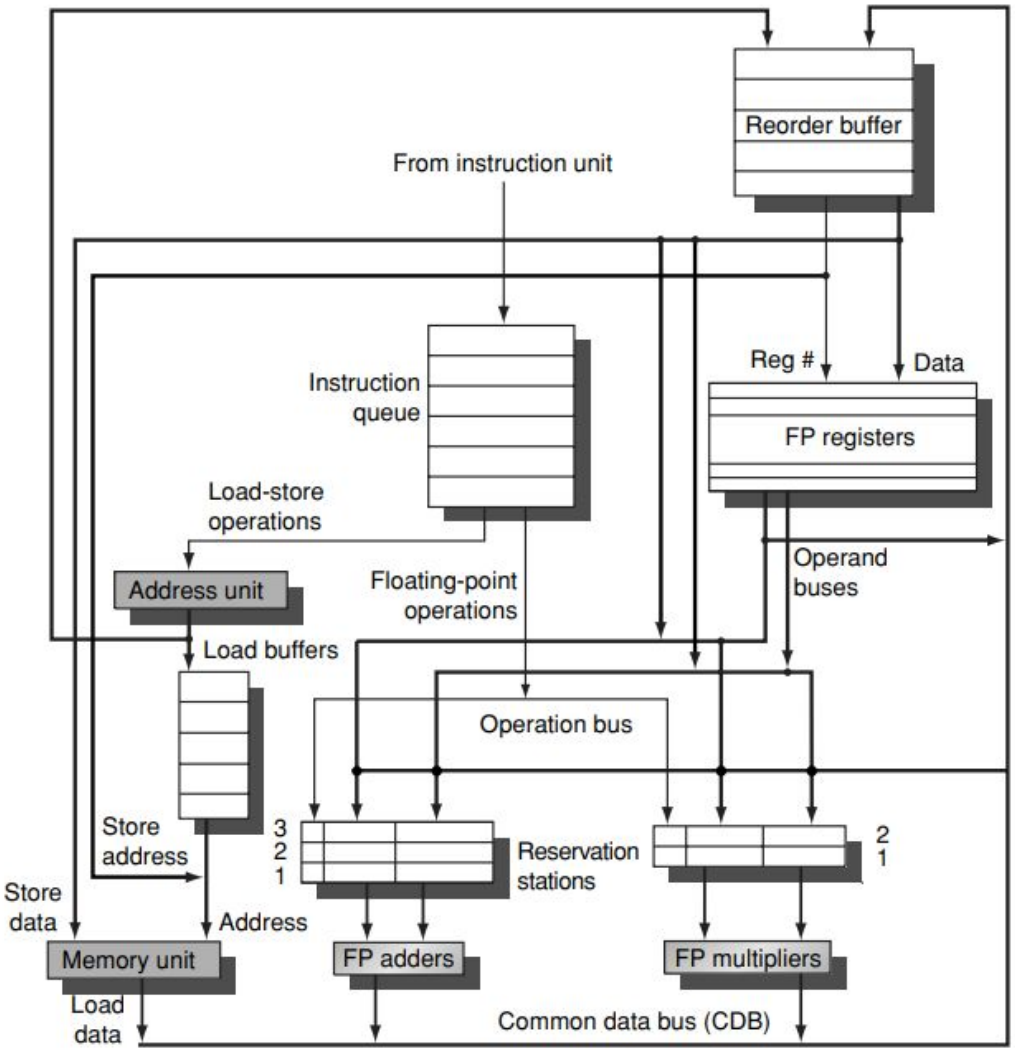
7. *Typically, a cache is organized in lines, each capable of storing the contents of a given memory block. A line, however, does not store only the contents of the memory block. Other information must be present. Which is it? (2 points)*

(slides 3-21, 3-26 e 3-30)
bloco de dados + tag + valid bit + LRU/FIFO bits + dirty bit (write-back)
tag field- address na main memory do bloco que está naquela linha de cache
validation bit - verificar se a linha presente na cache é útil/ tem significado
dity bit - (write back writting policy) states if thats block was written to. If written needs to be replaced in main men.

8. *Distinguish static from dynamic instruction scheduling. What are the advantages the latter has over the former? (2 points)*

(slide 4-38)
Static scheduling - lançamento de instruções in-order.
Dynamic scheduling - rearranjo da ordem de lançamento das instruções feito por hardware por forma a tentar aproveitar instruções sem dependências para maximizar a eficiência do pipeline. e diminuir o número de stalls.
Para além disto, o dynamic scheduling permite que um binário não tenha de ser adaptado para ser corrido noutra máquina (noutro pipeline) pois o próprio hardware encarrega-se de adaptar a ordem das instruções ao hardware disponível.

9. *The diagram below depicts the basic organization of a floating point unit using the Tomasulo's algorithm extended to handle speculation. (2 points)*



a. *Explain what is speculation and how it is dealt with in this organization.*

(slide 4-78)

Especulação é uma técnica que tenta prever o resultado de um branch (taken ou not taken) por forma a tentar maximizar a eficiência do pipeline. Nesta organização, a especulação é feita com recurso ao reorder buffer.

b. *With speculation, a further step on instruction execution has to be introduced. Which is it?*

(slide 4-86)

A fase de commit. Nesta fase, os dados estão no reorder buffer a aguardar o sinal de "ready" para poderem ser escritos no banco de registos ou na memória. Esta fase permite descartar valores inválidos sem comprometer a integridade/coerência dos dados.

10. *In a vector computer, if the loop length is greater than the size of the vector registers a technique, known as strip-mining, is applied by the compiler when generating the code. Explain in what it consists. (2 points)*

(slide 5-24)
(https://www.youtube.com/watch?v=Z9koSZ9R0xw)
Strip-mining é uma técnica aplicada pelo computador em arquiteturas vetoriais quando o tamanho do loop (N) é superior ao tamanho do vector registers (K). Neste caso, é criado um novo loop com ceil(N/K) iterações para que seja possível aproveitar ao máximo os registos presentes no vector registers. Num loop de tamanho 50 e um vector registers de tamanho 8, ao aplicar strip-mining irá ser criado um loop com 7 iterações sendo que nas 6 primeiras serão utilizados todos os registos do vector registers enquanto que na última serão utilizados apenas 2.

11. *When writing a program in CUDA, three optimization criteria must be considered for the program to run efficiently in a GPU. Which are they? (2 points)*

Definir o tamanho do grid por forma a fazer uso de todos os SIMDs disponíveis.
Definir o tamanho do bloco por forma a ser um múltiplo de warp size.
Minimizar os cache misses através de fine tuning do programa.

Apêndice B – Correção do Exame (Época de Recurso, 2019)

1. *Take the following memory addresses expressed in decimal: 6742_{10} and 5472_{10}. Convert them to hexadecimal assuming a 32-bit address length. Which one of them <u>can not</u> represent the location of a properly aligned 32-bit operand? Justify your claim in detail. (2 points)*

1) Converter de decimal para hexadecimal (ou para binário).

$6742_{10} = 1A56_{16} = 1\ 1010\ 0101\ 0110_2$

$5472_{10} = 1560_{16} = 1\ 0101\ 0110\ 0000_2$

2) Perceber quantos bits dos menos significativos vão ter de estar a 0.

32-bit operand = 4-bytes operand => 4 address positions

$log_2(4) = 2$

3) Tirar conclusões.

Visto que os bits menos significativos das addresses são, respetivamente, 10 e 00, podemos concluir que a address 6742_{10} não pode representar a localização de um operando de 32 bits corretamente alinhado.

2. *In the 1960s, Michael Flynn studied the parallel computing efforts made so far and found a classification that is still popular today. Which are the characteristics of computer systems he has used to base his classification? How did he name the different categories? Take as an example a quad-core processor. In which category would you place it? (2 points)*

(slide 1-36)

Michael Flynn considerou o paralelismo nas instruções e nos streams de dados e criou 4 categorias:

- SISD (Single Instruction, Single Data)
- MISD (Multiple Instruction, Single Data)
- SIMD (Single Instruction, Multiple Data)
- MIMD (Multiple Instruction, Multiple Data)

Um processador quad-core insere-se na última categoria, Multiple Instruction Multiple Data.

3. *Explain why pipelining can speed up instruction execution in a processor and why one may say that pipelining is an example of implementing parallelism at instruction level. (1 point)*

(slide 2-3)

Pipelining é uma técnica de implementação na qual se divide uma tarefa em subtarefas por forma a que as subtarefas possam ser executadas em simultâneo em vários objetos diferentes.

Apesar desta técnica aumentar o tempo de processamento de cada instrução individual, o throughput de um processador com pipelining passa de 1/T (sendo T o tempo de execução de uma instrução numa implementação non-pipelined) para 1/t (sendo t o um ciclo de clock

da versão pipelined) o que se traduz num speedup de até N (sendo N o número de etapas do pipeline).
Por estar a executar várias instruções diferentes em simultâneo (em fases diferentes), esta técnica é um exemplo de paralelismo ao nível de instruções.

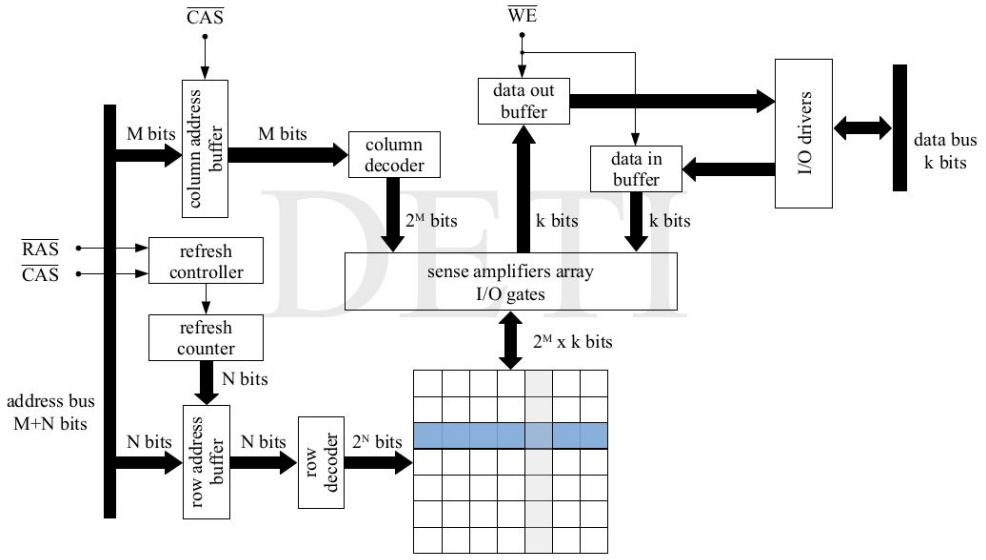
4. *However, for pipelining to be really effective, that is, approaching as much as possible the nominal throughput in normal operation, one has to deal with hazard resolution. Explain in detail how structural hazards may be dealt with for a processor implementing the 5-stage classical pipeline with a single integer and multiple floating point execution units. (2 points)*

(slide 2-94)
Visto que estamos a considerar uma implementação de um pipeline clássico de 5 etapas, vamos considerar que o problema de dois acessos simultâneos à memória (fases IF e MEM) já foi corrigido separando as caches de instruções e de dados.
Os hazards estruturais que podem surgir desta implementação advêm das múltiplas unidades funcionais (fase EX). Os hazards podem não surgir na fase EX em si mas na fase de WB. Por exemplo, uma instrução de divisão FP que seja lançada pode terminar ao mesmo tempo que uma instrução de adição de inteiros que tenha sido lançada depois. Isto causará dois acessos em simultâneo ao banco de registos.
Para além do caso acima, podemos ainda considerar que as unidades funcionais de FP necessitam de mais de um ciclo de relógio para completar a sua execução (como fizemos em cima) e podemos pressupor que não estão pipelined. Ou seja, podemos ter múltiplas instruções do mesmo tipo a tentar executar mas não existirem unidades funcionais disponíveis para processar a instrução.
Para solucionar estes problemas deve-se tentar o mais cedo possível no pipeline a situação e introduzir stalls de acordo.

5. *In the example of question 4, explain why data hazards of the type WAR can never happen. (1 point)*

(slide 4-15)
Para hazards do tipo WAR poderem ocorrer num pipeline, as escritas têm de ser feitas antes das leituras no pipeline. Isto não ocorre na maioria dos pipelines nos quais se engloba o pipeline referido na alínea anterior.

6. *The schematics below depicts the major building blocks of and asynchronous DRAM. Based on it, describe how a read operation takes place. (1.5 points)*



(slides 3-69 e 3-70)
1) O sinal de Write Enable fica a High, sinalizando uma leitura;
2) É colocada a linha que se pretende ler no row address buffer sendo que esse valor é lido no falling edge do RAS;
3) A informação presente na linha escolhida é colocada no sense amplifiers array;
4) É colocada a coluna que se pretende ler no column address buffer sendo que esse valor é lido no falling edge do CAS;
5) A informação presente na coluna escolhida da linha que foi colocada no sense amplifiers array é colocada no data out buffer.

7. *Cache misses may be modeled into three basic types. Which are they? (1.5 points)*

(slide 3-59)
Compulsory misses - miss que acontece da primeira vez que se tenta aceder a um bloco de dados.
Capacity misses - miss existente em *fully associative caches* causados por tamanho insuficiente da *cache* o que leva a que um pedaço de informação seja descartado (para dar lugar a outro que tenha sido chamado) e depois tenha de ser novamente copiado da memória para poder voltar a ser usado
Conflict misses - miss causado por haver demasiados blocos mapeados para o mesmo set da cache.

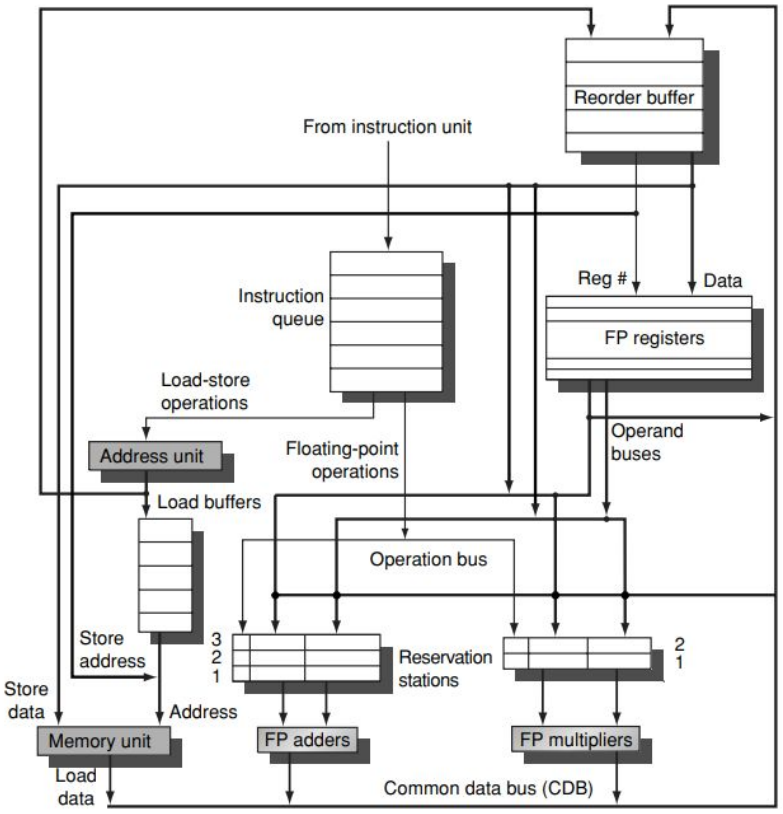
8. *Explain which of the basic types of cache misses are affected by the size of the memory block stored in each line. (1 point)*

(slide 3-62)
Ao aumentar o tamanho do memory block, reduz-se o número de compulsory misses (ao aceder ao um bloco pela primeira vez estamos a aceder a mais informação portanto, pelo princípio da localidade espacial, irá haver menos compulsory misses).
Por outro lado, ao aumentar o tamanho do bloco está também a reduzir-se o número de linhas o que pode aumentar o número de capacity e conflict misses.

9. *Distinguish static from dynamic instruction scheduling. Give an example of each. (2 points)*

(slide 4-38)
Static scheduling - lançamento de instruções in-order. Exemplo: pipeline clássico de 5 etapas sem hardware speculation.
Dynamic scheduling - rearranjo da ordem de lançamento das instruções feito por hardware por forma a tentar aproveitar instruções sem dependências para maximizar a eficiência do pipeline. Exemplo: pipeline com hardware speculation.
Para além disto, o dynamic scheduling permite que um binário não tenha de ser adaptado para ser corrido noutra máquina (noutro pipeline) pois o próprio hardware encarrega-se de adaptar a ordem das instruções ao hardware disponível.

10. *The diagram below depicts the basic organization of a floating point unit using the Tomasulo's algorithm extended to handle speculation.*



a. *Explain what is speculation and how it is dealt with in this organization. (1 point)*

(slide 4-78)
Especulação é uma técnica que tenta prever o resultado de um branch (taken ou not taken) por forma a tentar maximizar a eficiência do pipeline. Nesta organização, a especulação é feita com recurso ao reorder buffer.

b. *With speculation, a further step on instruction execution has to be introduced. Which is it? (1 point)*

(slide 4-86)
A fase de commit. Nesta fase, os dados estão no reorder buffer a aguardar o sinal de "ready" para poderem ser escritos no banco de registos ou na memória. Esta fase permite descartar valores inválidos sem comprometer a integridade/coerência dos dados.

11. *In a vector computer, if the loop length is greater than the size of the vector registers, a technique, known as strip-mining, is applied by the compiler when generating the code. Explain in what it consists. (2 points)*

(slide 5-24)
(https://www.youtube.com/watch?v=Z9koSZ9R0xw)
Strip-mining é uma técnica aplicada pelo computador em arquiteturas vetoriais quando o tamanho do loop (N) é superior ao tamanho do vector registers (K). Neste caso, é criado um novo loop com ceil(N/K) iterações para que seja possível aproveitar ao máximo os registos presentes no vector registers. Num loop de tamanho 50 e um vector registers de tamanho 8, ao aplicar strip-mining irá ser criado um loop com 7 iterações sendo que nas 6 primeiras serão utilizados todos os registos do vector registers enquanto que na última serão utilizados apenas 2.

12. *When writing a program in CUDA, two optimization steps must be carried out so that the program runs efficiently in a GPU. Which are they? (2 points)*

Definir o tamanho do grid por forma a fazer uso de todos os SIMDs disponíveis.
Definir o tamanho do bloco por forma a ser um múltiplo de warp size.
Pode/deve também ser considerado como um dos critérios a ter em conta: minimizar os cache misses através de fine tuning do programa.

Apêndice C – Correção do Exame (Época Normal, 2019)

1. *State the law of Amdahl. Assume that for a given program, 60% of the code may run in parallel. What is the speed up that can be achieved if the program is run in a quad-core processor computer? And if the program is run in a ten processor computer? Present the speed up results as fractions. Based on the numbers you have computed, explain why it is frequently said that Amdahl's law is a law of diminished returns. (2 points)*

$$speedup_{total} = \frac{t_{sem\ melhoria}}{t_{com\ melhoria}} = \frac{1}{(1-frac_{melhorada}) + \frac{frac_{melhorada}}{speedup_{parcial}}}$$

$$speedup_{quad-core} = \frac{1}{(1-0.6)+\frac{0.6}{4}} = \frac{1}{0.4+\frac{0.6}{4}} = \frac{1}{\frac{1.6+0.6}{4}} = \frac{4}{2.2} (\simeq 1.8)$$

$$speedup_{ten-core} = \frac{1}{(1-0.6)+\frac{0.6}{10}} = \frac{1}{0.4+\frac{0.6}{10}} = \frac{1}{\frac{4+0.6}{10}} = \frac{10}{4.6} (\simeq 2.2)$$

A lei de Amdahl é uma lei de "deminishing returns" pois os ganhos não são proporcionais ao aumento na capacidade de processamento. Isto é, de um single-core para um quad-core verifica-se um speedup de aproximadamente 1.8. Em contrapartida, de um quad-core para um ten-core (mais do dobro do número de cores) verifica-se um speedup de apenas cerca de 2.2/1.8 = 1.2.

2. *When discussing the subject of computer systems, one often speaks about architecture and organization. What is the difference between these two concepts? How are they related in the context of a family of computers produced by the same or different manufacturers? (2 points)*

(slides 1-12 e 1-13)

"Computer architecture" é o conjunto dos atributos de um sistema computacional visíveis para o programador. Neste conjunto incluem-se o instruction set, o número e tipo de registos do processador, os modos de endereçamento da memória e os mecanismos de I/O. Por outro lado, "computer organization" refere-se aos detalhes de hardware que não são conhecidos pelo programador.

Empresas que produzem computadores podem produzir vários modelos com a mesma arquitetura mas organizações diferentes o que justifica, em grande parte, as diferenças em performance e preço.

3. *Pipelining has become a popular technique in implementing instruction-level parallelism for the design of RISC processors. What is a RISC processor and why it is so? (1 point)*

(slide 1-27)

Um processador RISC é um processador que implemente um instruction set com um reduzido número de instruções. Estes processadores permitiram a exploração do paralelismo de instruções pelo que o pipelining se tornou uma técnica de implementação muito popular com estes processadores.

4. *However, for pipelining to be really effective, that is, approaching as much as possible the nominal throughput in normal operation, one has to deal with hazard resolution. Explain in detail how data hazards may be dealt with for a processor implementing the 5-stage classical pipeline with a single integer and multiple floating point execution units. (2 points)*

(slides 2-94, 2-95 e 2-99)

Há dois tipos de hazards de dados a considerar: WAW e RAW. Os hazards do tipo RAW não têm nada de diferente relativamente ao que se passa no pipeline clássico de 5 etapas com apenas uma unidade funcional. No entanto, os hazards do tipo WAW vão poder ocorrer neste pipeline caso as diferentes unidades funcionais demorem diferentes números de ciclos de relógio para completarem as suas operações.

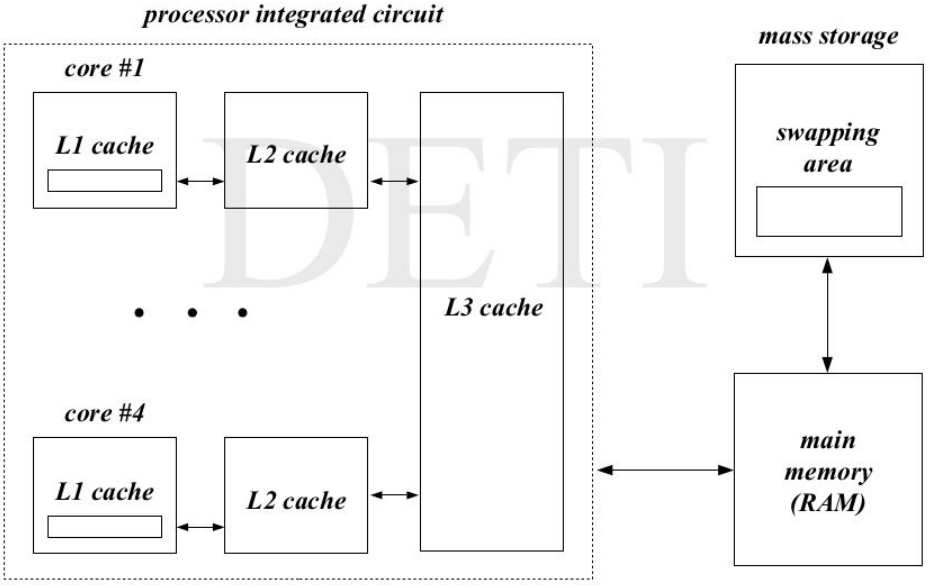
Alguns hazards podem ser atenuados através de forwarding. No entanto, será necessário em muitos casos a inserção de stalls ou mesmo descartar instruções no pipeline por forma a dar resposta aos hazards.

5. *In the example of question 4, one says "the instructions are executed in-order, but may be completed out-of-order". Why it is so? (1 point)*

(slide 2-94)

Esta implementação de pipeline, por ter múltiplas unidades funcionais, pode ter unidades funcionais que demorem diferentes números de ciclos de relógio a completar a sua execução o que poderá resultar em instruções que foram lançadas depois de outras, completarem a sua execução antes. Veja o exemplo de uma instrução de divisão FP seguida de uma adição de inteiros. É provável que as unidades funcionais que vão realizar as operações tenham números de ciclos de relógio diferentes e que a divisão FP demore muito mais que a adição de inteiros. Assim, é razoável prever que a segunda instrução terminará antes da primeira.

6. *In order to speed up memory access, a high speed special memory, called cache, is placed between the processor and main memory. Taking the figure below as a schematics of the organization, explain how the different components operate both for cache hits and cache misses. (1.5 points)*



(slide 3-8 e … cenas)
Por forma a aproveitar o princípio da localidade e a hierarquia de memória, quando uma instrução tenta fazer um acesso à memória, esse acesso não é feito diretamente à memória principal. É feito um acesso à cache (incomparavelmente mais rápido que um acesso à memória principal) por forma a ver se a cache contém a informação pretendida.
No caso de a cache ter o conteúdo pedido pelo programa (cache hit), realiza-se a operação de leitura ou de escrita.
No caso de a cache não ter o conteúdo pretendido, dá-se um cache miss e a serão feitos acessos aos níveis inferiores da hierarquia de memória por forma a obter o bloco de informação de informação necessário. Para isto será necessário trocar uma linha de cache por outra de acordo com as políticas de mapeamento (direct mapping, fully associative ou set associative) e seleção (random, LRU ou FIFO) implementadas.
Se for uma operação de escrita (e tendo em consideração que as políticas de escrita das caches L1 e L2 são de write-through e a política de escrita da cache L3 é de write-back), é ainda realizada uma escrita para L2 e para L3.

7. *Typically, a cache is organized in lines, each capable of storing the contents of a given memory block. A line, however, does not store only the contents of the memory block. Other information must be present. Which is it? (1.5 points)*

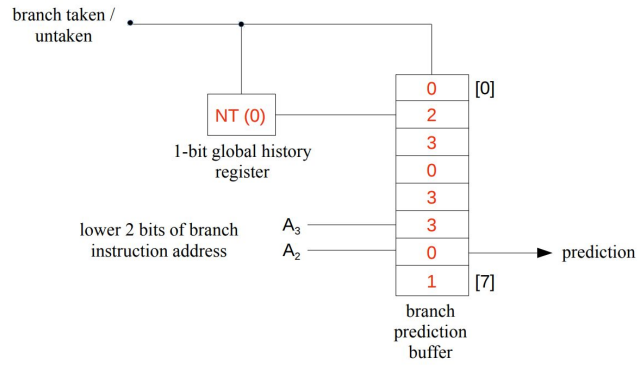
(slides 3-21, 3-26 e 3-30)
bloco de dados + tag + valid bit + LRU/FIFO bits + dirty bit (write-back)

8. *Explain why a cache line does not store a single data word but always a data block consisting of several data words. (1 point)*

Tendo em consideração a ordem de grandeza da latência no acesso à memória principal e o princípio da localidade espacial, a implementação mais lógica no que toca às transferências de dados entre cache e memória principal é transferir um conjunto de words (todas na mesma zona da memória, bloco) por forma a reduzir o número de acessos à memória necessários (cache misses) e aproveitando ao máximo o bus de dados existente.

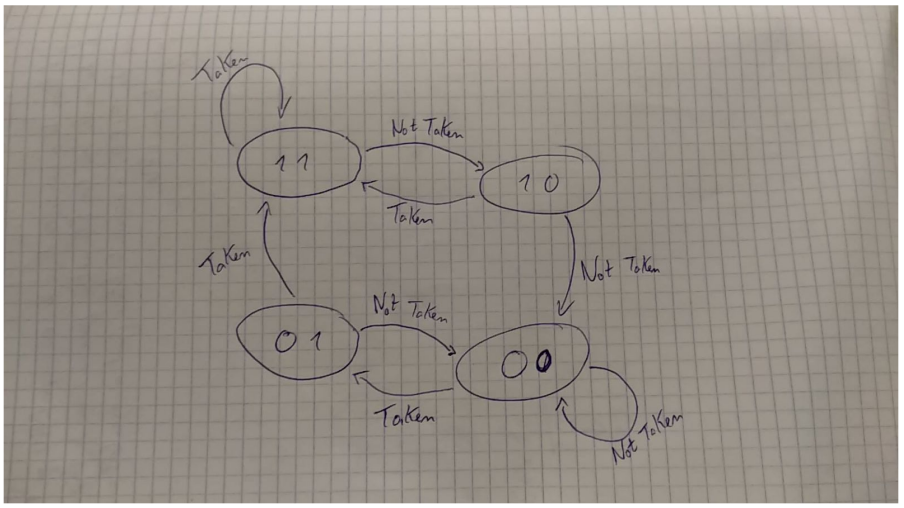
9. *Branch execution hinders pipeline performance because enforcing control dependencies generate hazards that require stalling the pipeline progress in many situations. So, advanced branch prediction has become standard practise. Sketch the organization of a correlating branch predictor and explain in detail how it works. (2 points)*

Desenhar qualquer coisa como isto:

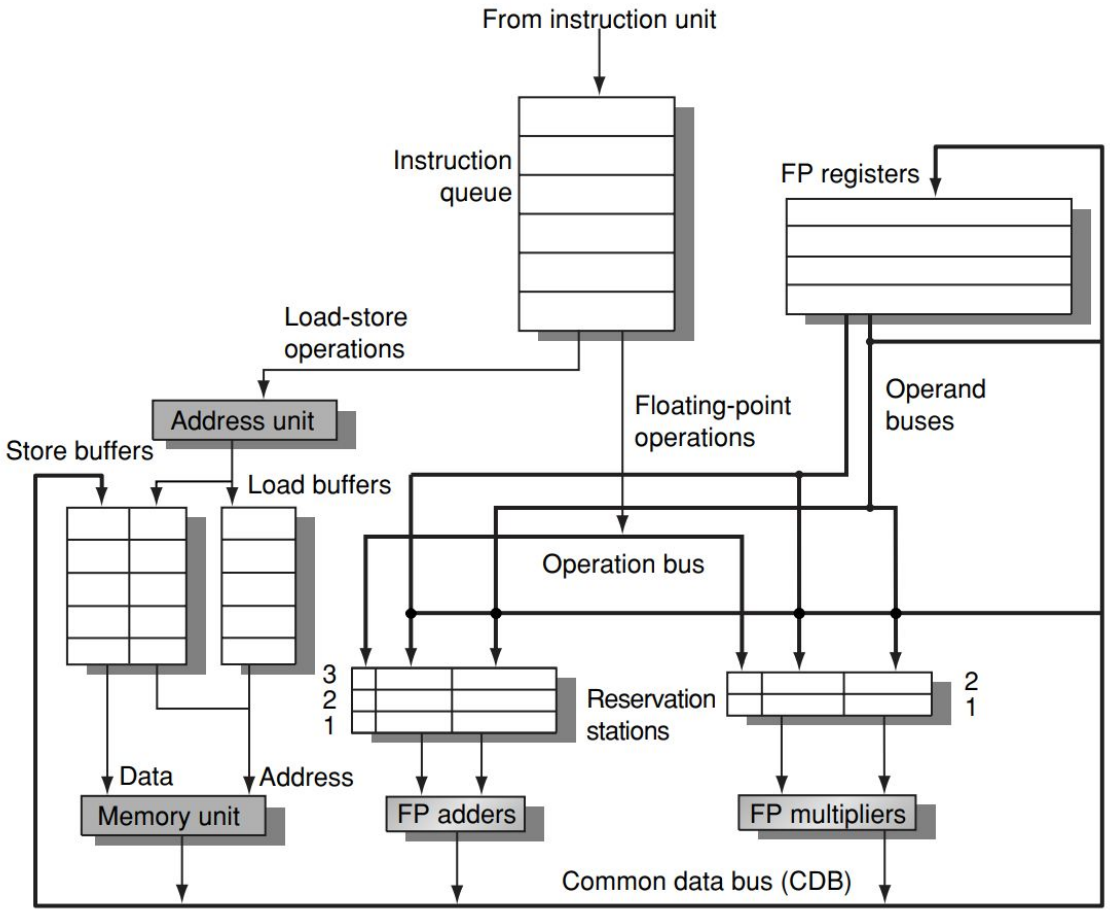


(1,2) correlating branch predictor

Um (m,n) correlating branch predictor usa n bits menos significativos do endereço da instrução de branch concatenados com m bits de histórico global (últimos m branches terem sido taken ou not taken) para selecionar uma posição no branch prediction buffer. Esta posição do buffer contém o estado da máquina de estados associado àquela posição. Esse estado é que vai ditar a predição feita. Numa máquina de estados de 2 bits, caso o estado seja 0 ou 1 a predição será not taken enquanto que se o estado for 2 ou 3 a predição será taken. Depois, conforme o branch seja, de facto, taken ou not taken, o valor no branch prediction buffer poderá ter de ser atualizado. Tomando, novamente, o exemplo de uma máquina de estados de 3 bits, o comportamento terá de ser de acordo com o seguinte esquema:



10. *The diagram below depicts the basic organization of a MIPS floating point unit using the Tomasulo's algorithm.*



a. *Explain what is register renaming and how it is dealt with in this organization. (1 point)*

Register renaming é a alteração do nome dos registos de destino por forma a eliminar hazards de dados WAR e WAW causados por out-of-order completion.
Nesta organização, o register renaming é feito através das reservation stations que servem de buffer a operações à espera para serem executadas e que permitem que um dos operandos seja o resultado de uma reservation station em vez de um registo. Isto permite que os operandos sejam fetched logo que disponíveis, evitando ter de esperar pela escrita no banco de registos.

b. *Explain how out-of-order execution takes place in this organization. (1 point)*

Como as operações que estão à espera para serem executadas ficam em buffer nas reservation stations e vão buscar os operandos logo que estes sejam computados por outras reservation stations, é possível que uma instrução que seja issued primeiro, seja executada depois caso os seus operandos só estejam disponíveis mais tarde. Isto deve-se também aos tempos variáveis das unidades funcionais (neste caso, os FP adders e FP multipliers).

11. *What is the main difference between a vector computer and a GPU-based computer? (2 points)*

Um vector computer é um computador cuja arquitetura de operação é vetorizada (está pensado quase que exclusivamente para operações vetorizadas) enquanto que um GPU-based computer normalmente tem a GPU como um coprocessador de um processador convencional (computação heterogénea). Neste último caso, o processador convencional (construído para operações escalares) encarrega-se das operações escalares e “offloads” o que puder ser computado vetorialmente para a GPU por este ter essas capacidades. Pode parecer que esta implementação é o melhor de ambos os mundos mas no cálculo da performance tem de ser equacionado o tempo de transferência entre o processador e a GPU.

12. *Describe in detail what is the CUDA computation model and how it runs in a GPU-based computer. (2 points)*

CUDA é uma plataforma de computação paralela e modelo de programação que permite fazer o offloading do esforço computacional para uma GPU da NVIDIA. Assim sendo, só é possível fazer uso desta plataforma num sistema computacional que contenha uma GPU da NVIDIA (um GPU-based computer).
Isto permite a um computador com um CPU convencional fazer uso de paralelismo de dados.
Para executar um programa de CUDA, ao compilar o código tem que ser separado entre código para ser executado no CPU (C) e código para ser gerado na GPU (CUDA C)