Etapa 1: Instruction Fetch

- A word de 32-bits na qual a instrução é codificada tem de ser sempre lida a partir da memória (instruction fetch)
- Para além disso o PC (program counter) tem de ser sempre incrementado para apontar para a instrução seguinte (PC = PC + 4)

Etapa 2: Instruction Decode

- Depois do fetch, é necessário fazer a descodificação da instrução e obter os dados associados a cada campo
- Primeiro, ler o opcode para determinar o tipo de instrução e o tamanho dos campos
- Segundo, ler os dados de todos os registos indicados de forma a definir os operandos
  - Para o `add`, lê-se dois registos
  - Para o `addi`, lê-se um único registo
  - Para o `jal`, não é necessário ler-se registos

Etapa 3: ALU (Unidade de Lógica e Aritmética)

- Na maior parte das instruções o trabalho efetivo é feito neste nível: aritmética (+, -, *, /), deslocamento, lógica (&, |), comparações (`slt`)
- E quanto aos loads e stores?
  - `lw $t0, 40($t1)`
  - Repare que é necessário calcular o endereço final através da adição de 40 (imediato) ao contéudo do registo `$t1`
  - A adição para o cálculo do endereço é feita nesta etapa

Etapa 4: Memory Access

- Somente as instruções load e store é que fazem trocas de informação com a memória (leitura e escrita); todas as outras instruções ficam inativas (idle) durante esta etapa.
- Este é uma etapa incontornável para a implementação dos loads e stores. Assim, e apesar das outras instruções não terem este passo, o datapath tem de conter esta etapa.

Etapa 5: Register Write

- A maioria das instruções escreve o resultado de uma determinada operação num registo destino.
- exemplos: operações aritméticas e lógicas, deslocamentos, loads, `slt`
- E quanto aos stores, jumps e branches?
  - Estas instruções não escrevem nenhum resultado num registo destino
  - São instruções que permanecem inativas durante esta etapa.

```
add   $r3,$r1,$r2    # r3 = r1+r2
```

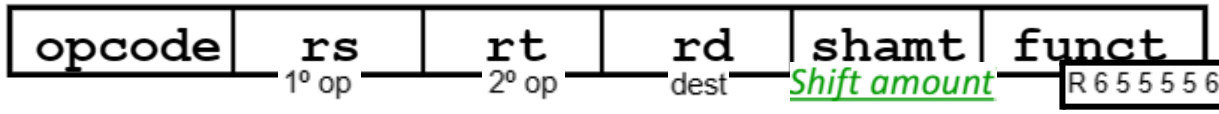
- Etapa 1: instruction fetch, inc. PC
- Etapa 2: descodificação para determinar que é um `add`. Leitura dos registos `$r1` e `$r2`
- Etapa 3: soma dos dois valores provenientes da etapa 2
- Etapa 4: idle (não há qualquer leitura/escrita de memória)
- Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

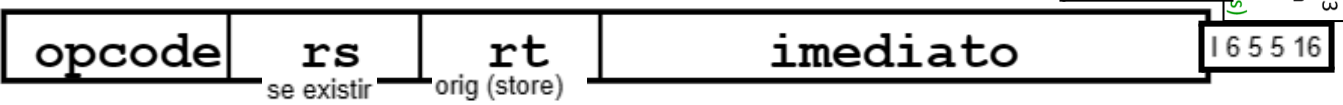
```
slti   $r3,$r1,17
```

- Etapa 1: fetch da instrução, inc. PC
- Etapa 2: descodificação para descobrir que é um `slti`. Leitura do registo `$r1`
- Etapa 3: comparação do valor proveniente da Etapa 2 com o inteiro 17
- Etapa 4: idle
- Etapa 5: escrita do resultado da etapa 3 no registo `$r3`

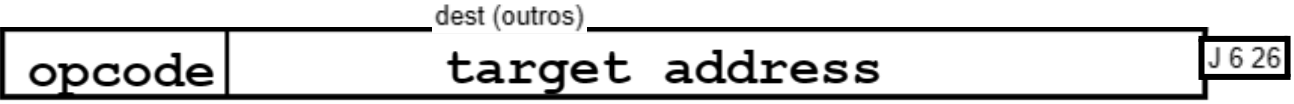
```
sw   $r3, 16($r1)
```

- Etapa 1: fetch da instrução, inc. PC
- Etapa 2: descodificação para saber que é um sw. Leitura dos registos $r1 e $r3
- Etapa 3: soma de 16 ao valor do registo $r1
- Etapa 4: escrita do valor que reside no registo $r3 (proveniente da Etapa 2) na posição de memória com o endereço calculado na Etapa 3
- Etapa 5: idle (não há nada a escrever nos registos)





1 bit sinal



2^26 words -> 2^28 bytes

Mnemonic	Meaning	Type	Opcode	Funct
add	Add	R	0x00	0x20
addi	Add Immediate	I	0x08	NA
addiu	Add Unsigned Immediate	I	0x09	NA
addu	Add Unsigned	R	0x00	0x21
and	Bitwise AND	R	0x00	0x24
andi	Bitwise AND Immediate	I	0x0C	NA
beq	Branch if Equal	I	0x04	NA
bne	Branch if Not Equal	I	0x05	NA
div	Divide	R	0x00	0x1A
divu	Unsigned Divide	R	0x00	0x1B
j	Jump to Address	J	0x02	NA
jal	Jump and Link	J	0x03	NA
jr	Jump to Address in Register	R	0x00	0x08
lbu	Load Byte Unsigned	I	0x24	NA
lhu	Load Halfword Unsigned	I	0x25	NA
lui	Load Upper Immediate	I	0x0F	NA
lw	Load Word	I	0x23	NA
mfhi	Move from HI Register	R	0x00	0x10
mflo	Move from LO Register	R	0x00	0x12

Mnemonic	Meaning	Type	Opcode	Funct
mult	Multiply	R	0x00	0x18
multu	Unsigned Multiply	R	0x00	0x19
nor	Bitwise NOR (NOT-OR)	R	0x00	0x27
xor	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
or	Bitwise OR	R	0x00	0x25
ori	Bitwise OR Immediate	I	0x0D	NA
sb	Store Byte	I	0x28	NA
sh	Store Halfword	I	0x29	NA
slt	Set to 1 if Less Than	R	0x00	0x2A
slti	Set to 1 if Less Than Immediate	I	0x0A	NA
sltiu	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
sltu	Set to 1 if Less Than Unsigned	R	0x00	0x2B
sll	Logical Shift Left	R	0x00	0x00
srl	Logical Shift Right	R	0x00	0x02
sra	Arithmetic Shift Right	R	0x00	0x03
sub	Subtract	R	0x00	0x22
subu	Unsigned Subtract	R	0x00	0x23
sw	Store Word	I	0x2B	NA

```
add   $8,$9,$10
```

0	9	10	8	0	32

```
addi  $21,$22,-50
```

8	22	21	-50

mal/tal:

MAL (MIPS Assembly Language): conjunto de instruções que o programador pode utilizar para fazer código para o MIPS; isto inclui as pseudo-instruções.

TAL (True Assembly Language): conjunto de instruções que são traduzidas directamente para uma instrução linguagem máquina de 32 bits

```
MAL                                TAL
---                                ---

move $4, $3                        add $4, $3, $0

add $4, $3, 15 # not $15           addi $4, $3, 15
                                   # also andi, ori, etc.

mul $8, $9, $10                    mult $9, $10  # $HI || $LO <-- product
                                                 # never overflow
                                   mflo $8       # $8 <-- $LO
                                                 # ignore $HI!

div $8, $9, $10                    div $9, $10   # $LO <-- quotient
                                                 # $HI <-- remainder
                                   mflo $8

rem $8, $9, $10                    div $9, $10
                                   mfhi $8

branches:
bltz,bgez,blez,bgtz,beqz,bnez,     bltz,bgez,blez,bgtz,
blt,bge,ble,bgt,beq,bne            beq,bne
beqz $4, loop                      beq $4, $0, loop

blt $4, $5, target                 slt $at, $4, $5 # $at is 1 if $4 < $5
                                                   # $at is 0 otherwise
                                   bne $at, $0, target

addi   $t0,$t0, 0xABABCDCD         lui     $at, 0xABAB
                                   ori     $at, $at, 0xCDCD
                                   add     $t0,$t0,$at
```

BRANCHES COM CONSTANTES E BRANCHES COMPOSTOS (BLT, BLE, ETC) SÃO MAL!

SUB É MAL!

pipelining:

1) IFtch: Instruction Fetch, Incrementa PC
2) Dcd: Instruction Decode, Lê Registos
3) Exec:
   Mem-ref: Calcula endereços
   Arith-log: Executa a operação
4) Mem:
   Load: Leitura de dados da memória
   Store: Escrita de dados para a memória
5) WB: Write Data Back to Register

Conflitos Estruturais (*structural hazards*): O HW físico não permite suportar determinadas combinações de instruções (e.g. uma única pessoa não pode dobrar e arrumar a roupa simultaneamente)

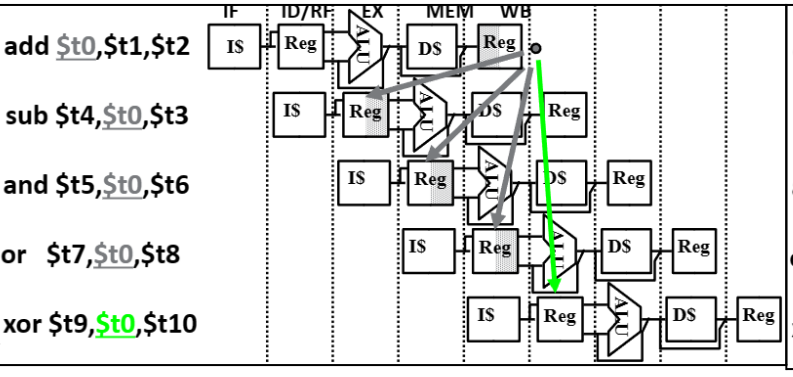
solução conflito w/r em regs: write na 1º ½ e read dps
solução: RegFile permite read after write

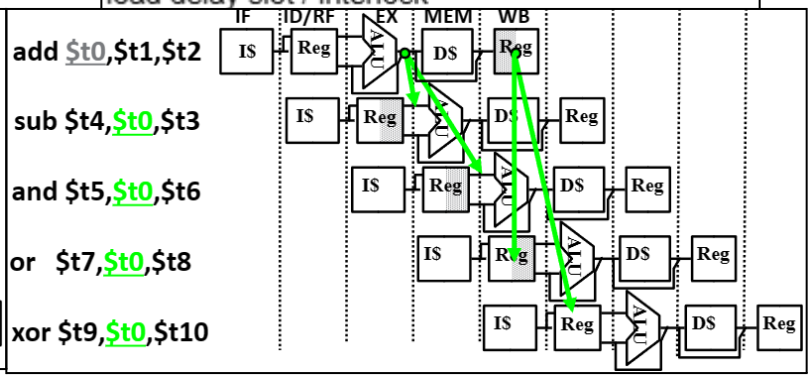
Conflitos de Controlo (*control hazards*): Quando aparecem saltos potenciais no fluxo de execução (instruções de *branch* e *jump*) existe incerteza quanto às instruções que se seguem. Isto causa paragens e poderá levar a uma limpeza do pipeline e retrocesso na execução (“*flush*”).

solução branch: nop/op útil abaixo (branch-delay slot)

Conflitos de Dados (*data hazards*): Instruções que dependem do resultado de outras instruções que ainda estão no *pipeline* (o caso do par de peúgas)

solução dados: forwarding, por vezes. se não der, load delay slot / interlock





com loads: load delay slot é o slot com a instruçao após o load, interlock é equivalente a um stall nesse slot
com branches: branch delay slot é o slot com a instruçao após o branch, a instrução a seguir é sempre executada, porque o branch avança para debaixo desta

na pag 3, todos os titulos referem-se à assemblagem. na pag 4, os titulos referem à etapa acima delas.

Compilação

Input: Código fonte escrito numa linguagem de alto nível
(e.g., C, Java como `foo.c`)

Output: Código em linguagem *assembly*
(e.g., `foo.s` para o MIPS)

Nota: O *output* pode conter pseudo-instruções

Pseudo-instruções: instruções que o *assembler* compreende mas que não fazem parte do "*instruction set*" do processador. Por exemplo
- `move $s1,$s2` => `add $s1,$s2,$zero`

Assemblagem

Input: Código em linguagem *assembly*
(e.g., `foo.s` para o MIPS)

Output: Código objecto, tabelas
(e.g., `foo.o` para o MIPS)

Lê e utiliza Directivas
Substitui pseudo-instruções (MAL para TAL)
Produz código máquina
Cria Ficheiro de Código Objecto

inst. tipo r resolvidas na assemblagem
relativo: branches / absolutos: jumps
endereços relativos: resolvidos na assemblagem
endereços absolutos: resolvidos na linkagem

Substituição de Pseudo-Instruções

O assembler não só considera como pseudo-instruções instruções que manifestamente não fazem parte do ISA, como rectifica variações cujo sent é claro.

Pseudo:	Real:
`subu $sp,$sp,32`	`addiu $sp,$sp,-32`
`sd $a0, 32($sp)`	`sw $a0, 32($sp)` `sw $a1, 36($sp)`
`mul $t7,$t6,$t5`	`mult $t6,$t5` `mflo $t7`
`addu $t0,$t6,1`	`addiu $t0,$t6,1`
`ble $t0,100,loop`	`slti $at,$t0,101` `bne $at,$0,loop`
`la $a0, str`	`lui $at,left(str)` `ori $a0,$at,right(str)`

Directivas do Assembler

Dá indicações ao assembler, mas não é traduzido em instruções máquina

- `.text`: Colocar o que vem a seguir no segmento de texto do utilizador (a ser traduzido em código máquina)
- `.data`: Colocar o que vem a seguir no segmento de dados do utlizador
- `.globl sym`: declarar `sym` como "label" global que pode ser referenciado a partir de outros ficheiros
- `.asciiz str`: Armazenar a string `str` em memória terminada por null
- `.word w1,…,wn`: Armazenar os *n* elementos de 32-bit em words sucessivas de memória

Geração de Código Máquina (1/3)

- Casos Simples
  - Instruções aritméticas e lógicas (add, sub, sll, or, etc.)
  - Toda a informação necessária está codificada na própria instrução

E quanto aos "branches" condicionais?
- Salto relativo ao valor do PC
- Só podemos saber o tamanho real do salto relativo, depois de as pseudo-instruções terem sido substituídas

No caso dos "branches" a assemblagem requer duas passagens
As instruções de "*branch*" podem fazer referência a "*labels*" que estão à frente no código

```
    or    $v0,$0,$0
L1: slt   $t0,$0,$a1
    beq   $t0,$0,L2
    addi  $a1,$a1,-1
    j     L1
L2: add    $t1,$a0,$a1
```

A tradução para código máquina da instrução "beq" é feita em 2 passagens
- A primeira passagem determina a posição do *label*
- A segunda passagem usa a posição do *label* para fazer a tradução

E quanto aos *jumps* (`j` e `jal`)?
- Os *jumps* funcionam em termos de endereços absolutos.
- Só é possível gerar a instrução máquina depois de se saber a posição do *label* em memória (o salto não é relativo)
- Isto só pode ser resolvido depois da linkagem

E quanto às referências a dados?
- `la` é desdobrado num `lui` e `ori`
- Estes precisam de saber o endereço de 32 bits dos dados … (mesmo problema que os *jumps*)

Tabela de Símbolos
- Lista os "itens" do "ficheiro .o" que podem ser referenciados deste ou de outros "ficheiros .o".
- Que itens são estes?
  - *Labels*: e.g. chamada de funções
  - Dados: qualquer coisa da secção `.data`; variáveis que podem ser acedidas a partir de outros ficheiros

Tabela de Realocação
- Lista os"itens" que o "ficheiro .o" referencia e do qual não tem o endereço porque são externos (estão noutro ficheiro) ou serão resolvidos em "*runtime*".
  - Os "labels" usados nos `j` ou `jal`
    - internos
    - externos (includindo ficheiros .lib)
  - Dados
    - Por exemplo, a instrução `la`

TR: 1 entrada por j/jal,
1 entrada por la (+1 se o registo >16bit)
Como isto só se sabe depois da assemblagem, precisamos de criar duas tabelas …

Formato dos ficheiros .o (código objecto)

Cabeçalho: posição e tamanho dos diferentes componentes do ficheiro objecto.

Segmento de texto: código máquina
Segmento de dados: representação binária dos dados e estruturas declarados no código fonte (normalmente declarações globais)

Tabela de realocação: identifica as linhas de código onde há endereços a ser resolvidos
Tabela de símbolos: lista de "*labels*" internos que podem ser referenciados, quer a partir do próprio ficheiro, quer a partir de ficheiros externos.

Informação de *debug*: (lembre-se da *flag* –g do *gcc*)
Um formato standard é o ELF (Executable and Linkable Format), excepto nas ferramentas Microsoft.

Linker (1/3)

Input: Ficheiros código objecto, tabelas (e.g., `foo.o`, `libc.o` para o MIPS)

Output: Código executável (e.g., `a.out` para MIPS)

Combina vários ficheiros (.o) num único executável ("linking")

A técnica permite a compilação separada de diferentes ficheiros

- Alterações num ficheiro fonte não requerem a recompilação de todo o programa (lembra-se do makefile?)

Passo 1: Concatenação dos segmentos de texto de cada ficheiro .o

Passo 2: Juntar os segmentos de dados de cada ficheiro .o e concatená-los com o segmento de texto

Passo 3: Resolver as referências

- Ver as tabelas de realocação e resolver cada entrada
- Definir os endereços absolutos em relação ao início do programa

Tipos de Endereçamento

Endereçamento em relação ao PC (`beq`, `bne`): não é usada realocação

Endereçamento absoluto (`j`, `jal`): realocação sempre

Referências externas (normalmente `jal`): realocação sempre

Referência a dados (normalmente `lui` e `ori`): realocação sempre

Loader (1/2)

Input: Código Executável (e.g., `a.out` para MIPS)

Output: (programa a correr)

Os ficheiros executáveis estão armazenados em disco.

Quando o executável é chamado, o "loader" tem a tarefa de o carregar em memória e iniciar a execução.

Normalmente o "loader" é o próprio OS

Lê o cabeçalho dos executáveis para determinar o tamanho e posição dos segmentos de texto e dados

Cria um espaço de endereçamento para o programa capaz de receber o texto, dados e pilha (e eventualmente "*heap*")

Copia os dados e instruções do executável para o espaço de endereçamento criado

Copia os argumentos de chamada para a pilha (lembre-se do *argc* e *argv* no C)

Inicializa os registos do processador

- A maioria dos registos são colocados a 0, mas o "*stack pointer*" fica a apontar para a 1ª *frame* livre

Salta para a rotina de "*start-up*" (ainda OS) que copia os argumentos do programa e faz o set do PC

Se a rotina principal (*main*) regressar, a rotina de "*startup*" termina o programa com uma chamada a exit.

Numa ***Direct-Mapped Cache*** cada endereço de memória é associado a um único bloco de memória na *cache*.

Precisamos apenas de verificar um único local para confirmar se os dados existem ou não na *cache*.

O *Bloco* é a unidade mínima de transferência entre o *cache* e a memória

tttttttttttttttt	iiiiiiiiii	oooo

Index: especifica o índice da cache (em qual "linha"/bloco da cache devemos procurar)

Offset: depois de encontramos o bloco correto, especifica qual o byte dentro do bloco que queremos

Tag: os bits restantes são usados para identificar quais os endereços de memória que são mapeados no mesmo bloco da cache

O número de bits do *tag* depende apenas do tamanho da cache. Nunca depende do tamanho de cada bloco.

Se souber o tamanho da cache do seu computador então pode frequentemente fazer com que o seu código corra mais rápido.

As hierarquias de memória tiram partido da localidade temporal ao manter sempre os dados mais recentes próximos do processador.

ABC

Suponha que temos uma memória *cache* de mapeamento direto com 16 KB com blocos de 16 bytes.

Determine o tamanho dos campos de *tag*, *index* e *offset* se estivermos a utilizar uma arquitetura de 32 bits

Offset

- especifica o byte correto dentro de um bloco
- Cada bloco contém 16 bytes

  16 bytes = 2^4 bytes
- Vão ser necessários 4 bits para especificar o byte dentro do bloco

Index: (basicamente especifica o endereço de cada bloco na cache)

- A *cache* contém 16 KB = 2^{14} bytes
- Cada bloco contém 2^4 bytes (16 bytes)
- O número de blocos na *cache* será:

$$Nº\ Blocos = \frac{Tamanho\ Cache}{Nº\ Bytes\ por\ Bloco} = \frac{2^{14}}{2^4} = 2^{10}$$

- São necessários 10 bits para especificar este número de blocos
- Logo o campo *tag* são os 18 bits mais à esquerda do endereço de memória

Quando tentamos ler a memória, três cenários podem acontecer:

cache hit: o bloco de *cache* é válido e contém o endereço apropriado, então basta ler a *word* desejada da *cache*;

cache miss: nada no *cache* no bloco apropriado, então carregar o bloco da memória principal;

cache miss, block replacement: o bloco na *cache* não tem a *tag* certa, então substituir o bloco pelo correcto;

Benefícios de um tamanho de Bloco maior:

- Localidade Espacial: se acedermos a uma determinada palavra, muito provavelmente acederemos a palavras próximas de seguida
- Na execução de programas ao executarmos um determinada instrução, é muito provável que também executemos as próximas de seguida.
- Funciona também muito bem em acessos sequenciais como por exemplo em tabelas

Desvantagens de um tamanho de Bloco maior

- Tamanho de bloco maior significa maior penalização no caso de um *miss* (*miss penalty*)
  - em caso de falha, leva mais tempo para carregar um novo bloco do próximo nível de memória
- Se o tamanho do bloco for muito grande em relação ao tamanho da cache, então há poucos blocos
  - Resultado: a *miss rate* aumenta

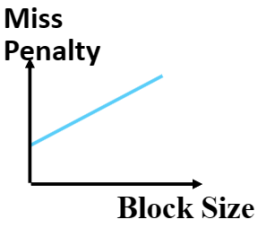
Em geral pretende-se minimizar o **tempo médio de acesso à memória** ou *Average Memory Access Time* (AMAT)

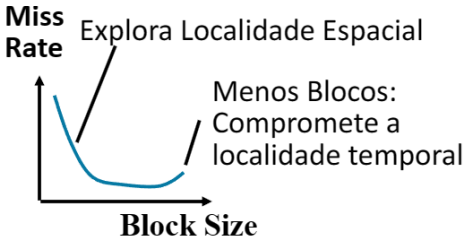
= Hit Time + Miss Penalty x Miss Rate

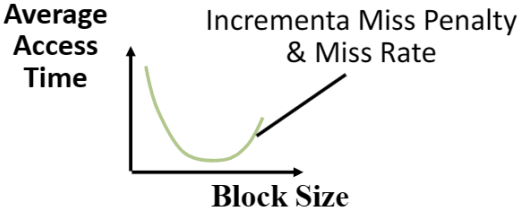
Hit Time = tempo necessário para encontrar e recuperar dados da *cache* de nível atual

Miss Penalty = tempo médio para recuperar dados em uma falha no nível atual da *cache* (inclui a possibilidade de perdas em níveis sucessivos da hierarquia de memória)

Hit Rate = % de pedidos que são correspondidos no *cache* de nível atual







Cache Misses

1st C: Compulsory Misses
- Ocorre quando um programa é iniciado pela primeira vez
- A *cache* não contém nenhum dos dados desse programa ainda, então podem ocorrer falhas
- Não pode ser evitado facilmente. A solução está fora do âmbito da disciplina

2nd C: Capacity Misses
- Falha que ocorre porque a cache tem um tamanho limitado
- Falha que não ocorreria se aumentássemos o tamanho da *cache*

3rd C: Conflict Misses
- Falha que ocorre quando dois endereços de memória distintos são mapeados no mesmo bloco da *cache*
- Pode ocorrer sobretudo quando dois blocos que por acaso são mapeados para o mesmo local na cache são utilizados de forma alternada
- É um desperdício caso existam outros blocos livres na cache, mas que correspondem a outros blocos de memória que não estão a ser acedidos no momento
- Este é o grande problema das caches de mapeamento directo!
- Como atenuamos as consequências dete problema?

Cache Fully Associative

Campos no Endereço de Memória
- ***Tag***: tal como anteriormente
- ***Offset***: tal como anteriormente
- ***Index***: não existe!

Não há "linhas": qualquer bloco na memória principal pode ser mapeado em qualquer bloco na *cache*

Tem de se comparar com todas as *tags* na *cache* para descobrir onde o bloco da memória foi mapeado

Vantagens:
- Não ocorrem *Conflit Misses* (já que os dados podem ir parar a qualquer lugar da *cache*)
- O principal tipo de falha é o Capacity Miss

Desvantagens:
- Precisamos de ter um comparador em hardware para cada entrada na cache: se tivermos 64KB de dados numa cache com 4B de em cada bloco, precisaríamos de ter 16K comparadores: **impraticável**

Cache N-Way Set Associative

Agrupar blocos na cache em conjuntos de N blocos, que se comportam como uma cache *Fully Associative*.

Como **N** é normalmente pequeno então o hardware necessário não é complexo.

Resolve os *Conflict Misses* uma vez que no mesmo conjunto vamos ter espaço para vários blocos.

Tag: tal como anteriormente, um identificador único de cada bloco em memória.

Offset: tal como anteriormente

Index: indica qual **o conjunto** de blocos na *cache* onde o bloco de memória pode ser mapeado

cada conjunto na cache pode conter vários blocos

Assim que encontramos o conjunto correto na cache, temos de comparar com todas as *tags* desse conjunto para encontrar o local onde o bloco já está mapeado.

Cache N-Way Set Associative

Ideia Basica:

- A *cache* é *direct-mapped* se pensarmos em termos de *sets;*
- Cada conjunto é *fully associative;*
- Basicamente as *caches N-way Set Associative* trabalham em paralelo dentro de cada conjunto: cada bloco tem o seu próprio *valid bit* e dados.

Dado um endereço de memória:

- Encontrar o conjunto correto na cache usando o valor do campo ***Index***.
- Comparar o ***Tag*** do bloco em memória com todos os ***Tags*** no conjunto determinado anteriormente.
- Se ocorrer um ***match***, então temos um ***hit***!, caso contrário, um ocorre um ***miss***.
- Finalmente, usar o campo de ***offset*** como de costume para encontrar os dados desejados dentro do bloco.

Quais as grandes vantagens?

- Mesmo uma cache 2-way Set Associative Evita muitos ***conflict misses***;
- O custo do hardware não é significativo: apenas são necessários N comparadores.

De facto uma *cache* com M blocos:

- É ***Direct-Mapped*** se for uma *cache 1-way Set Associative*
- É ***Fully Associative*** se for uma *cache M-way Set Associative*
- Então estes dois exemplos são apenas casos especiais do modelo concepetual de uma cache N-Way Set Associative

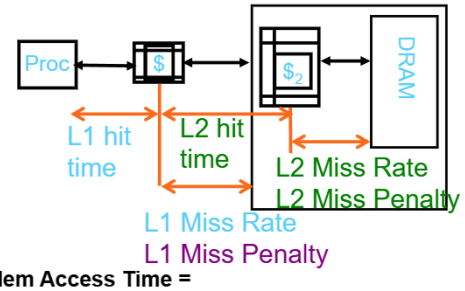
AMAT = Hit Time + Miss Penalty x Miss Rate

ganho de execução **Ge** da parte melhorada (acelerada) do sistema (**Ge**>1);

fracção de tempo **Ft** que a parte modificada (acelerada) do sistema ocupa na execução global (**Ft**≤1).

$$speedup = \frac{1}{(1-Ft)+\frac{Ft}{Ge}}$$

Analisando uma Hierarquia de Memória *Cache* Multinível



Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty =
L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time =
L1 Hit Time + L1 Miss Rate *
(L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)

Qual é o bloco que o `malloc()` escolhe?

best-fit: escolhe o bloco mais pequeno que satisfaça os requisitos de espaço

first-fit: Escolhe o primeiro bloco que satisfaça os requisitos

next-fit: semelhante ao *first-fit*, mas lembra-se onde terminou a pesquisa da última vez, e retoma-a a partir desse ponto (não volta ao início)

Suponha que uma determinada memória cache é **20 vezes mais rápida** do que a memória principal e suponha ainda que a cache **hit rate é igual a 75%**. Qual será o máximo ganho (***speedup***) obtido através do uso de cache no sistema?

Ft = 0.75, Ge = 20

Determine o tempo médio de acesso (***average access time***) de um sistema em que o acesso à memória principal **requer 80 ns**, enquanto que o acesso à **cache é 10 vezes mais rápido** e tem um **hit rate de 80%**.

8 + 0.2 * 80

Considere um sistema baseado num processador com um valor de **CPI REAL igual a 6.6**. O sistema possuí **duas caches**, uma para instruções e outra para dados. A **hit rate da cache de instruções é de 85%** e a da **cache de dados é igual a 75%**. **Apenas 40% das instruções envolvem um acesso à memória de dados**. Se considerarmos que o **miss penalty é igual a 10 ciclos** de relógio em ambas as caches, qual será o **CPI IDEAL** deste sistema?

$$CPI_{Real} = CPI_{Ideal} + MissRate_{Inst} * MissP_{Inst} + PInst_{Dados} * MissRate_{Dados} * MissP_{Dados}$$

Assuma que uma memória tem 32 blocos e a cache consiste em 8 blocos. Determine onde será encontrado o 13º bloco de memória na cache para:

a) Direct mapped cache — 12 % 8 = 4

b) 2-way set-associative cache — 12 % 4 = 0 (1º ou 2º)

c) Fully associative cache — qualquer

Considere a seguinte hierarquia de memória (memória principal + memória cache) em que:

- Tamanho da memória principal é 2 Giga bytes;
- Tamanho da memória cache é 1 Mega bytes;
- Tamanho de bloco da memória cache é 128 bytes.

Considere os seguintes tipos de arquitecturas de caches:

i. Direct mapped cache — a) 2^13
ii. 2-way set-associative cache — b) 2^12
iii. 8-way set-associative cache — 2^10
iv. Fully associative cache — 1

T	I	O
11	13	7
12	12	7
14	10	7
24	0	7

a) Calcule o número de blocos a considerar nestas caches.
b) Calcule o número de sets das set-associative caches
c) Calcule a estrutura de endereçamento para cada um dos tipos de caches.

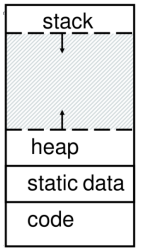
bit: ⅛ byte
kb: 2^{10} byte
mb: 2^{20} byte
gb: 2^{30} byte

```
char str[] = "One STRING";

void main(){
 int i;
 char *temp;

 temp=(char *)malloc(strlen(str)+1);

 for(i=0;i<=strlen(str);i++)
    temp[i]=str[i];
}
```



A variável ***str*** vai ser armazenada na zona de dados estáticos (global)
As variáveis ***i*** e ***temp*** vão ser armazenadas na pilha.
A variável ***temp*** vai apontar para uma zona de memória no *heap*.

Static Storage: onde ficam as variáveis globais que podem ser lidas/escritas por qualquer função do programa. Este espaço está alocado permanentemente durante todo o tempo em que o programa corre (daí o nome estático)

A Pilha/Stack: armazenamento de variáveis locais, parâmetros, endereços de retorno, etc.

A Heap (dynamic malloc storage): os dados são válidos até ao instante em que o programador faz a desalocação manual com `free()`.

Como o processador interage com o seu ambiente?

– Panorâmica sobre a unidade de entrada/saída (I/O)

Como trocar informação com os dispositivos?

– I/O Programada ou I/O Mapeado em Memória

Como lidar com eventos?

– *Polling* ou Interrupções

Como transferir grandes quantidades de dados?

– Acesso Directo à Memória (*DMA Direct Memory Access*)

Parâmetros de *Bus*

Largura = Número de Fios

Tamanho da Transferência = Nº de palavras por transação no bus

Síncrono (utilizando o relógio do *bus*) ou Assíncrono (sem utilizar o relógio do bus / "relógio próprio")

Bus Processador – Memória ("*Front Side Bus, QPI*)

– Curto, rápido e largo

– Topologia tipicamente fixa, desenhada por "*chipset*"

• CPU + Caches + Interligações + Controlador de Memória

Bus Periféricos e I/O (PCI, SCSI, USB, LPC, …)

– Mais comprido, mais lento e mais estreito

– Topologia flexível, ligações múltiplas/variadas

– Interoperabilidade entre vários dispositivos

– Liga-se ao bus processador-memória através de uma ponte (*bridge*)

Unidades de I/O diferentes requerem uma organização hierárquica das interligações entre componentes. A tendência é cada vez mais optar-se por uma topologia com ligações ponto-a-ponto para privilegiar a velocidade.

APIs para dispositivos de I/O

Registos de Comando

– Uma escrita neste registo permite o dispositivo realizar uma dada tarefa

Registos de Estado

– Leitura permite indicar o que está a ser feito, códigos de erro, etc.

Registos de Dados

– Escrita: transfere dados para o dispositivo

– Leitura: transfere dados do dispositivo

Prós? Contras? (referente às interruptions)

– Mais eficiente na gestão de tempo da CPU: apenas interrompe quando o dispositivo está pronto

Menos eficiente em termos de recursos – salvaguarda de contexto
• Contexto do: PC, SP, registos, etc.

Contra: fluxo de execução de Código de temporização imprevisível uma vez que pode ser interrompido por acções externas

Interfaces de Comunicação

Q: Como pode o processador comunicar com o dispositivo?

R: Introdução de instruções especiais para I/O

I/O Programado ← Interage directamente com os registos de estado, dados e comando

– inb $a, 0x64 ← registo de estado do teclado

– outb $a, 0x60 ← registo de dados do teclado

– Especifica: dispositivo, dados, direcção de transferência

– Protecção: instruções apenas permitidas em modo de kernel

Q: Como pode o processador comunicar com o dispositivo?

A: Mapear registos num espaço de endereçamento virtual

Memory-mapped I/O ← Mais rápido

– O acesso a certos endereços de memória são redirecionados para os dispositivos

– Os dados circulam no bus da memória

I/O Programado

– Requer instruções especiais

– Pode requerer hardware dedicado para fazer a interface com os dispositivos

– Mecanismos de proteção via acesso restrito ao *kernel* às instruções de I/O

– A virtualização pode ser difícil

Memory-Mapped I/O

– Utiliza as instruções normais de *load/store*

– Utiliza as interfaces standard com a memória

– Mecanismos de protecção à custa dos esquemas de proteção de memória

– A Virtualização é possível através dos esquemas normais de virtualização de memória

Métodos de Comunicação

Como é que o programa sabe se o dispositivo está pronto ou terminou a tarefa?

Polling: Verificar periodicamente o registo de estado

– *If device ready, do operation*

– *If device done, …*

– *If error, take action*

```
char read_kbd(){
  do {
    sleep();
    status = inb(0x64);
  } while(!(status & 1));

  return inb(0x60);
}
```

Prós? Contras?

– Temporização previsível e é barato

– Mas: desperdiça ciclos de relógio com o CPU a não fazer nada

– Eficiente apenas se não existe mais nada a fazer em paralelo

Comum em sistemas embebidos ou de tempo-real

Interrupções: O dispositivo envia um pedido de interrupção ao CPU

– Existem registos específicos para identificar a causa da interrupção e o dispositivo

– Define-se uma rotina de resposta a interrupções que decide as acções a tomar

Esquema de prioridades

– Eventos urgentes podem interromper o tratamento de interrupções de menor prioridade

– O Sistema Operativo pode desactivar (ou deferir) interrupções

Transferência de dados Programada:

Dispositivo ←→ CPU ←→ RAM

for (i = 1 .. n)
- CPU emite um pedido
- O Dispositivo coloca os dados no bus e a CPU lê-os para os registos
- A CPU escreve os dados na memória
- ***Não é eficiente***

Muito Lento!!

Acesso Direto à Memória (DMA, Direct Memory Access)

1) O Sistema Operativos indica o endereço de início e o comprimento da transferência
2) O controlador (ou o dispositivo) transfere os dados para a memória autonomamente
3) É gerada uma interrupção após a conclusão ou a ocorrência de um erro

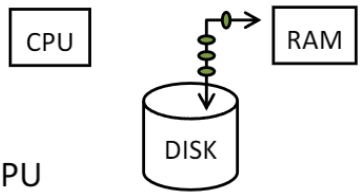
Transferência utilizando DMA:

Device ←→ RAM

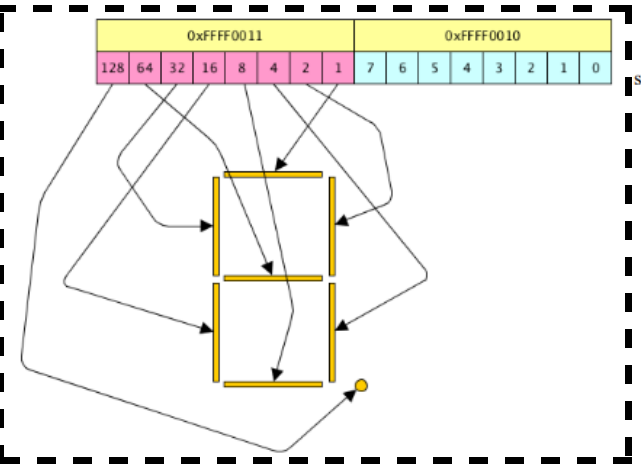
A CPU inicia um pedido DMA

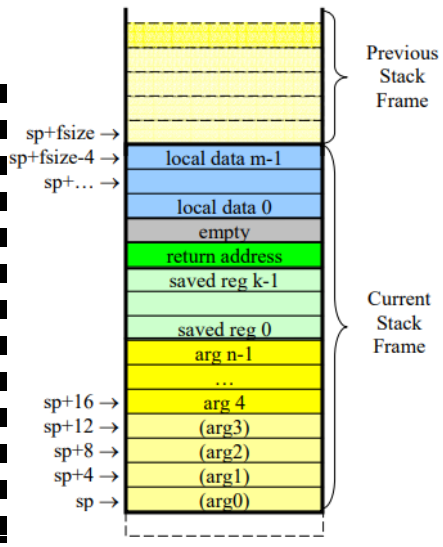
for (i = 1 ... n)
O dispositivo coloca os dados no bus e a memória aceita-os

O dispositivo interrompe a CPU após o final da transferência ou após a ocorrência de um erro.



formato makefile
target: dependencies
operation





ÍNDICE
T - top / B - bottom / R - right / L - left / C - center

Essential Commands

gdb *program* [*core*]	debug *program* [using coredump *core*]
b [*file*:]*function*	set breakpoint at *function* [in *file*]
run [*arglist*]	start your program [with *arglist*]
bt	backtrace: display program stack
p *expr*	display the value of an expression
c	continue running your program
n	next line, stepping over function calls
s	next line, stepping into function calls

Starting GDB

gdb	start GDB, with no debugging files
gdb *program*	begin debugging *program*
gdb *program core*	debug coredump *core* produced by *program*
gdb --help	describe command line options

Display

print [/*f*] [*expr*]	show value of *expr* [or last value $] according to format *f*
p [/*f*] [*expr*]	
info all-reg [*rn*]	display all enabled expressions on list
info reg [*rn*]...	register values [for regs *rn*] in selected frame; **all-reg** includes floating point
info locals	local variables of selected frame
info args	arguments of selected frame
info frame [*addr*]	describe selected frame, or frame at *addr*
down *n*	select frame *n* frames down
up *n*	select frame *n* frames up

Program Stack

backtrace [*n*]	print trace of all frames in stack; or of *n* frames—innermost if *n*>0, outermost if *n*<0
bt [*n*]	
frame [*n*]	select frame number *n* or frame at address *n*; if no *n*, display current frame
info display	numbered list of display expressions
enable disp *n*	enable display for expression(s) number *n*
disable disp *n*	disable display for expression(s) number *n*
undisplay *n*	remove number(s) *n* from list of automatically displayed expressions
display	display all enabled expressions on list
display [/*f*] *expr*	show value of *expr* each time program stops [according to format *f*]

Automatic Display

.c -> .s -> .o -> executável (.exe, …)

flags compilação:
-c: criar ficheiro objetos
-o: criar executável (para dar run)
-g: debug
-S: cria ficheiro assembly
-E: cria ficheiro assembly (tal)
-O / -O(1-3): otimização

O teclado simula um teclado normal em que as teclas são lidas através de um mecanismo de "*scan*". Isto significa que temos de ir testando linha a linha do teclado para ver se há alguma tecla premida. Assim para ler o teclado teremos de em ciclo ir enviando para o endereço 0xFFFF0012 o número da linha a testar, enviando sucessivamente cada um dos bits entre 0 e 3 activos. De seguida vamos ler no endereço 0xFFFF0014 se há alguma tecla premida nessa linha. O resultado lido se houver tecla premida será o composto por o número da linha (*nibble* menos significativo) e o número da coluna (*nibble* mais significativo), ou seja:

tecla 0 → 0x11, tecla 1 → 0x21, tecla 2 → 0x41, tecla 3 → 0x81, tecla 4 → 0x12..., tecla f → 0x88