# Paralelismo: Conceitos básicos

Yuri Kaszubowski Lopes

UDESC

Anotações



## No processador MIPS estudado

- Temos um paralelismo a nível de instrução
  - Através do pipeline, múltiplas instruções são executadas “ao mesmo tempo na CPU”
    - em estágios diferentes

  Porém:
  - Somente uma instrução é enviada a cada ciclo de clock
  - Somente uma instrução é (pode ser) completada a cada ciclo
  - Ganho está na redução do tempo de cada ciclo de clock
- Note também que as instruções operam em apenas um dado
  - Fazem a operação e armazenam o resultado em um registrador

Anotações



## SISD

- A CPU MIPS estudada é um exemplo de uma CPU SISD
- **SISD: Single Instruction Single Data**
  - Uma instrução para um dado
  - Processadores que executam uma instrução por vez, e cada instrução é capaz de operar em apenas um dado

Anotações

## Multiprocessadores

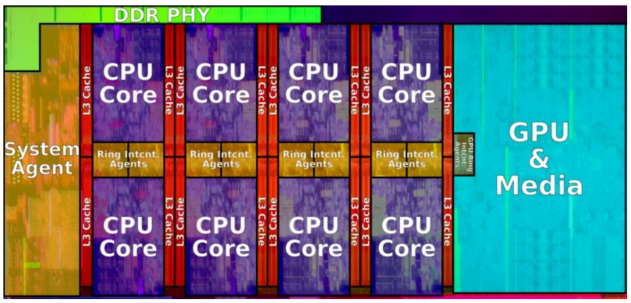
- Um multiprocessador é um processador composto de múltiplos processadores
  - Nossas máquinas x86-64 atuais são exemplos de multiprocessadores
  - A indústria os chama de **microprocessadores multicore**

Anotações



## i9-9900k

- Multiprocessador composto de 8 processadores (8 cores)




Anotações



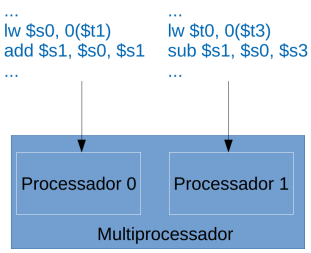
## Multiprocessadores

- Em um sistema multiprocessado, cada processador pode executar uma tarefa independente
  - Paralelismo a nível de tarefa/processo
- Se você escrever um programa sequencial, ele vai usar somente **um** dos n processadores disponíveis
  - Saber criar programas que executam em paralelo não é opcional
    - fork
    - Pthreads (POSIX Threads)
  - Em Python temos que considerar o Global Interpreter Lock (GIL)!
    - Efetivamente só usa uma CPU
    - Pois (quase) todo código em Python só pode ser executado se "em controle" do GIL
    - Então, devemos dividir o trabalho entre multiplos processos e não multiplas threads
    - `concurrent.futures.ProcessPoolExecutor`
    - `multiprocessing`: API de mais baixo nível

Anotações

## MIMD

- **MIMD: Multiple Instruction Multiple Data**
  - Múltiplas instruções e múltiplos dados
- Multiprocessadores podem ser considerados MIMD
  - Possuem n processadores, cada um operando uma instrução diferente
    - Multiple Instruction
    - Paralelismo a nível de instrução
  - Cada instrução opera em um dado diferente
    - Multiple data
    - Paralelismo a nível de dados






Anotações

## MISD

- **MISD: Multiple instruction single data**
  - Múltiplas instruções operando em um dado único
- E.g., uma instrução que executa múltiplas operações em um único dado
- **Não existem computadores puramente MISD atualmente**



Anotações

## SIMD

- **SIMD: Single instruction Multiple Data**
  - Uma instrução operando em múltiplos dados
- E.g., uma instrução que soma um valor imediato em múltiplos registradores ao mesmo tempo
- Comum em nossos processadores atuais



Anotações

## Resumo e exemplos

- Taxonomia de Flynn

| | | Dado | |
|---|---|---|---|
| | | **Único** | **Múltiplos** |
| **Instrução** | **Única** | SISD: MIPS estudado | SIMD: um único “core” de uma CPU i7 atual |
| | **Múltiplas** | MISD: ???? | MIMD: Intel i7 |



Anotações

## SIMD

- Como poderíamos adicionar capacidades SIMD no processador MIPS estudado
- É comum termos de carregar múltiplos endereços de memória para os registradores, somar um imediado, e depois armazenar os resultados
  - Comum quando fazemos operações em vetores ou matrizes
- Um exemplo dessas operações em 4 endereços consecutivos de memória
  - Como podemos criar instruções que operam com os dados de 4 em 4?
    - Uma instrução que carrega os dados da memória de 4 em 4
    - Uma instrução que adiciona um imediato em 4 registradores
    - Uma instrução que armazena os dados de 4 registradores na memória



Anotações

## SIMD

```
1  lw $s0, 0($t0)
2  lw $s1, 4($t0)
3  lw $s2, 8($t0)
4  lw $s3, 12($t0)
5  addi $s0, $s0, 1
6  addi $s1, $s1, 1
7  addi $s2, $s2, 1
8  addi $s3, $s3, 1
9  sw $s0, 0($t0)
10 sw $s1, 4($t0)
11 sw $s2, 8($t0)
12 sw $s3, 12($t0)
```



Anotações

## SIMD

- Uma primeira ideia
  - Vamos adicionar um s no final dos mnemônicos de nossas instruções para indicar que são SIMD
  - As instruções podem ficar na forma:

```
1 lws $s0, $s1, $s2, $s3, 0($t0)
2 addis $s0, $s1, $s2, $s3, $s0, $s1, $s2, $s3, 1
3 sws $s0, $s1, $s2, $s3, 0($t0)
```

- **lws**
  - `$s0 = MEMÓRIA[$t0 + 0]`
  - `$s1 = MEMÓRIA[$t0 + 4]`
  - ...
- **addis**
  - `$s0 = $s0 + 1`
  - `$s1 = $s1 + 1`
  - ...
- **sws**
  - `MEMÓRIA[$t0 + 0] = $s0`
  - `MEMÓRIA[$t0 + 4] = $s1`
  - ...



Anotações

## Problema

- `lws $s0, $s1, $s2, $s3, 0($t0)`
- **Problemas com uma instrução desse tipo?**
- opcode, regBase, reg1, reg2, reg3, reg4, imediato
- $6 + 5 \times 5 + 16 = 47$ bits
- Temos apenas 32 bits por instrução!



Anotações

## Criando registradores “grandes”

- Outra solução é adicionar registradores “grandes” em nossa CPU
- Exemplo:
  - Registradores xmm0, xmm1, ... xmm7
  - Cada registrador com capacidade para 128 bits
- Agora temos a operação
  - `lws xmm0, 0($t0)`
  - ... que carrega 128 bits (16 bytes) a partir de `$t0 + 1`
- $6 + 5 + 5 + 16 = 32$ bits
- **Problemas?**
  - Sim: com 5 bits para endereço dos registradores no máximo 32 registradores, e já estão todos ocupados:
    `$zero, $at, $s0, ..., $s7, $t0,..., $t9, $sp, $ra, ...`
  - Solução: o opcode dessas instruções (final s) indica o uso de outro banco de registradores



Anotações

## Instruções SIMD

- Utilizando os registradores de 128 bits do exemplo
  - Carregamos os 128 bits para o registrador
    - `lws xmm0, 0($t0)`
  - Agora podemos realizar uma adição de imediato em paralelo
    - `addis xmm0, xmm0, 10`
  - Como `addis` faz os cálculos para inteiros, ele pode pegar os primeiros 32 bits de `xmm0`, somar com 10, e armazenar nos primeiros 32 bits de `xmm0`
    - O primeiro inteiro foi salvo nos primeiros 32 bits
  - Os próximos 32 bits de `xmm0` são somados com 10, e armazenados nos próximos 32 bits de `xmm0`
  - Repetimos ...
  - A instrução segmenta o registrador e guarda cada resultado “em pedaços” (de 32 bits) no registrador

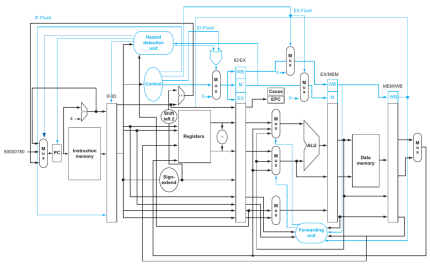
```
1 lws xmm0, 0($t0)
2 addis xmm0, xmm0, 10
3 sws xmm0, 0($t0)
```



Anotações

## Instruções SIMD

- `addis xmm0, xmm0, 10`
- Para que o `addis` efetivamente faça os cálculos em paralelo, precisamos adicionar hardware
- **O que precisamos modificar/adicionar na CPU MIPS?**






Anotações

## Instruções SIMD

- **O que precisamos modificar/adicionar na CPU MIPS?**
- Na CPU MIPS vamos precisar no mínimo:
  - Adicionar os registradores SIMD no banco de registradores
    - Novo banco de registradores
  - Precisamos de 4 ALUS, para que cada ALU faça o cálculo de um dos inteiros
    - Caso contrário criaríamos uma fila na ALU, executando o cálculo de um inteiro por vez



Anotações

## Instruções SIMD

- **É exatamente essa a estratégia utilizada nas CPUs x86 atuais**
  - Um dos conjuntos de instruções especializados do x86-64 para realizar esse tipo de operação é o SSE (Streaming SIMD Extensions)
- Execute **lscpu** em sua máquina e verifique a versão do SSE que ela é capaz de executar
- Então se olharmos para um único processador (core) de seu multiprocessador x86-64, veremos um processador com capacidades SIMD

Anotações



## SSE

- SSE do x86-64 funciona utilizando 8 registradores de 128 bits
  - xmm0, xmm1, ..., xmm7
- Dependendo da operação que executamos nesses registradores, podemos operar em paralelo:
  - 2 doubles ou
  - 2 longs ou
  - 4 floats ou
  - 4 ints ou
  - ...
- As operações SSE exigem que o vetor no qual estamos operando comece em um endereço de memória múltiplo de 16
  - Restrição de alinhamento de memória

Anotações



## SSE: Exemplo

```
1 float* vetor;
2 int ret = posix_memalign((void**)&vetor, 16, TAM_VETOR * sizeof(float));
```

- `posix_memalign` é uma função que aloca a memória (`malloc`) de maneira alinhada, e armazena o endereço no ponteiro vetor
- Devemos passar o endereço do vetor
- A função retorna 0 em caso de sucesso, ou `EINVAL`/`ENOMEM` em caso de erro
  - Ver códigos em `errno.h`
- Mais detalhes: `man posix_memalign`
- Funções e tipos prontos para lidar com SSE em C estão disponíveis na biblioteca `emmintrin.h`
- O tipo `__m128` representa uma variável de 128 bits, que o compilador vai carregar em um dos registradores `xmm`

Anotações

## SSE: Exemplo

```
printf("Tamanhos: %lu %lu %lu\n",
    sizeof(__m128),
    sizeof(float),
    sizeof(__m128) / sizeof(float));
```

- Para compilar: `gcc sse.c -o sse -lm -O3 -msse2`
- Imprimindo na tela o tamanho dos tipos das variáveis que serão envolvidas no programa
- `__m128` ocupa 16 bytes (128 bits)
  - Cabem 4 floats dentro de um `__m128`

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <errno.h>
#include <emmintrin.h>

#define TAM_VETOR 1073741824
```



Anotações

## SSE: Exemplo

```
void normal(float* a, int N) {
    for (int i = 0; i < N; ++i)
        a[i] = sqrt(a[i]);
}

void sse(float* a, int tamanho) {
    int numBlocos = tamanho / 4;
    __m128* ptr = (__m128*)a;
    for (int i = 0; i < numBlocos; ++i, ++ptr)
        _mm_store_ps((float*)ptr, _mm_sqrt_ps(*ptr));
}

int main(int argc, char** argv) {
    float* vetor;
    int ret = posix_memalign((void**)&vetor, 16,  TAM_VETOR
        * sizeof(float));

    for (int i = 0; i < TAM_VETOR; ++i)
        vetor[i] = 3141592.65358;

    // normal(vetor, TAM_VETOR);
    sse(vetor, TAM_VETOR);
    printf("%f %f %f\n", vetor[0], vetor[10], vetor[TAM_VETOR-1]);
}
```



Anotações

## Observações

- As instruções SIMD são executadas em uma única CPU
  - Único “núcleo de processamento”
- Se você possui, por exemplo, um i7 com 6 processadores (núcleos), somente 1 deles está operando
  - Os demais estão “sem fazer nada” (a respeito deste código)
  - Imagine quanto poderíamos otimizar se executarmos o programa em paralelo em múltiplos processadores, levando em consideração o pipeline, instruções SIMD, memória cache,
- dados são totalmente independentes uns dos outros, e nossa operação é simples
  - Temos independência de dados e um problema trivialmente paralelizável
- Entenda como lidar com problemas mais interessantes e também com paralelismo via múltiplos processos/threads nas disciplinas de:
  - Sistemas Operacionais
  - Processamento Paralelo



Anotações

## Referências


- D. Patterson; J. Henessy. **Organização e Projeto de Computadores: Interface Hardware/Software**. 5a Edição. Elsevier Brasil, 2017.
- J. Henessy; D. Patterson. **Arquitetura de computadores: Uma abordagem quantitativa**. 6a Edição, 2017.
- STALLINGS, William. **Arquitetura e organização de computadores**. 10. ed. São Paulo: Pearson Education do Brasil, 2018.
- software.intel.com/sites/landingpage/IntrinsicsGuide/
- docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/kcwz153a(v=vs.100)
- felix.abecassis.me/2011/09/cpp-getting-started-with-sse




Anotações

Anotações

Anotações