MC723 - Projeto 03
Grupo 7[1]
Sistemas Multicore e Offload para Hardware
26 de Junho de 2015

Daniel Vatanabe Pazinato RA116545
Ruan Ramos Monteiro Silva RA104034
Murilo Frônio Bássora RA094236

# 1 Introdução

Neste experimento implementamos um processador multicore no simulador MIPS implementado em ArchC (8 cores) com instruções para controle de concorrência e para acelerar o desempenho do software de teste escolhido. O programa de teste escolhido foi a soma de dois vetores de inteiros de mesmo tamanho *in-place*, ou seja, a soma de cada elemento do primeiro vetor com o elemento de mesmo índice do segundo vetor, guardando o resultado no primeiro vetor. Nas próximas seções daremos detalhes de implementação de cada parte do projeto.

# 2 Controle de Concorrência

Os diferentes dispositivos de controles de concorrência utilizados em software (mutex, semáforos) necessitam de uma implementação em hardware para garantir que seja feita de forma atômcica. Esta implementação é possível através de duas instruções atômicas, o load-link and store-conditional [1].
**Load-Link:** Carrega uma palavra da memória de forma atômica
Formato: LL rt, offset(base)
Descrição: rt = memory[base+offset]

**Store Conditional:** Guarda a palavra na memória se n
Formato: SC rt, offset(base)
Descrição: if (atomic update) then memory[base+offset] = rt, rt = 1 else rt = 0

Com essas duas instruções é possível criar um mutex. Então para implementar um mutex acquire podemos usar as as seguintes duas instruções:
LL lock, mutex
SC one, mutex
if (!lock  one) return 1 // conseguiu mutex

Para mutex release é só liberar o mutex fazendo: *mutex = 0;

As duas instruções LL e SC foram implementadas no mips_isa.cpp para poderem ser chamadas pelo simulador. Também foi usado assembly inline, para colocar as instruções diretamente no código c.

[1]Repositório disponível em https://github.com/danielvatanabe/mc723-lab3

# 3 MultiCore

Para transformar o simulador em multicore (com oito núcleos) iniciamos oito processadores mips e, antes de começar a simulação do programa, pausamos sete dos oito processadores (com o auxílio da função PauseProcessor() fornecida).

Para que os programas executados pudessem ligar ou desligar cores, criamos um novo periférico *controller* que conseguisse pausar ou continuar a execução de algum core (utilizando os métodos *PauseProcessor*() e *ResumeProcessor*()).

A forma com que o programa de execução se comunica com o *controller* é através de endereços de memória especiais. Reservamos uma faixa de endereços que não estava sendo usada para a memória e uma porta no barramento que leva as instruções que acessam esses endereços especiais para o nosso periférico. Mapeamos cada core para dois endereços: um no qual o core é ligado e outro no qual o core é desligado. Desta forma, com o auxílio de variáveis globais, os programas de teste foram capazes de ligar e desligar cores específicos.

O código do periférico criado encontra-se na pasta controller.

# 4 Software Offloading

Criamos um instrução que adiciona um valor a uma posição de mémoria. Chamamos essa instrução de las (load-add-store). Ela é muito útil no exemplo que escolhemos pois assim a soma de duas posições no vetor é direta.
Assinatura da instrução:
Formato: LAS rt, offset(base)
Descrição: memory[base+offset] = memory[base+offset] + rt

Essa instrução foi implementada no arquivo mips_isa.cpp, junto com a LL e SC. Para conseguir rodar a instrução no simulador foi usado o upcode da instrução SDL do MIPS III. Essa instrução ainda não tinha sido implementado no simulador então não há problema compatibilidade.

# 5 Programa de Teste - Soma de vetores

Como já dito anteriormente, escolhemos como programa de teste a soma de dois vetores de inteiros de mesmo tamanho. A entrada do nosso programa é um arquivo txt contendo três linhas. A primeira contém um número que representa o tamanhos dos dois vetores. As duas outras linhas são os elementos de cada um dos vetores, separados por espaço simples. Geramos diferentes arquivos de entrada.

A nossa escolha do programa se deu principalmente por dois fatores: pela simplicidade deste tipo de implementação em ser paralelizada e também por conseguirmos tirar o máximo proveito de nossa implementação de software offloading, já que nosso programa usa i vezes a operação que implementamos, onde i é o tamanho do vetor para a soma.

Sem a implementação do mutex, nosso programa paralelizado imprimia desordenadamente os valores da soma e com alguns valores faltantes, devido ao compartilhamento do mesmo recurso da tela por 8 núcleos. Foi preciso então usar o mutex para conseguir fazer a impressão dos valores computados por cada core na tela. O mutex também foi necessário para definir o ID de cada processador. Utilizando apenas uma variável global que era incrementada em cada execução foi fonte de erro em alguns casos.

O código do programa de teste *soma_vetores.c* encontra-se na pasta sw/ do projeto. Os arquivos de entrada estão nomeados como input_soma.dat, input_large1.dat, input_large2.dat.

# 6 Conclusão

Apesar da implementação simplificada de um processador multicore, tivemos contato com diversas técnicas que tem como fim a implementação deste tipo de processador em uma situação real. Problemas como a divisão de memória e de recursos entre os núcleos tem de ser solucionados com controle de concorrência usando mutex, e é preciso de um controlador que consiga administrar estes núcleos de maneira efetiva.

Com as rodadas de teste foi possível perceber a importância de haver uma instrução atômica para o mutex, pois sem ela (utilizando variáveis globais por exemplo) o funcionamento não era o esperado.

# Referências

[1] Mips Instructions Set: http://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf