Universidade Federal de Minas Gerais
Instituto de Ciências Exatas da UFMG
Departamento de Ciência da Computação

# Relatório segunda entrega: Processador MIPs Multiciclo

Data:08/05/2016
Disciplina: Organização de computadores II
Professor: Omar Paranaiba Vilela Neto
Monitor: Luiz Sardinha

Grupo: Alexandre Alphonsos
Vinicius Borges
Felipe Marcelino
Lucas Furtini

## 1. Introdução

O objetivo deste trabalho foi implementar um processador MIPs multiciclo em verilog e sintetizá-lo em uma FPGA. Para realizar esta tarefa, recebemos um modelo de processador já sintetizável no qual teríamos que incluir o módulo da ALU, ALUControl e registradores, bem como modifica-lo para incluir mais instruções e também adicionar uma segunda memória, a memória de dados.

## 2. Implementação

**ALU, ALUControl e Banco de Registradores:**
Utilizamos a implementação feita na entrega anterior, porém fizemos alterações para corrigir alguns erros e incluir as instruções de LOAD e STORE em memória de dados.

**Mem32_with_rst_data:**
Este módulo foi criado para ser a memória de dados. Replicamos o modelo já existente e criamos um initializer novo para alocar o espaço necessário na memória da FPGA.
Na FPGA ambas as memórias possuem palavras de 32 bits e tamanho de 1MB cada.

**MIPs:**
Neste módulo criamos a chamada do módulo da memória de dados (Mem32_with_rst_data) e enviamos os sinais necessários para seu correto funcionamento (por exemplo o endereço que a memória de dados recebe é calculado pela ALU, por isso ele recebe alu_res como sinal).
Modificamos também cada estado do multiciclo (IF, ID, EX, MEM e WB) para acrescentar e modificar os sinais de controle necessários para as novas instruções incluídas e a nova memória de dados. Também fizemos a modificação de somente ser permitida a leitura do registrador R0, escrita não é mais possível nesse registrador.

**Demais módulos:**
Os demais módulos do processador não precisaram ser modificados para que as modificações necessárias funcionassem.

# 3. Testes

Os testes são feitos utilizando dois arquivos (inst.mif e inst2.mif), o primeiro arquivo contém a sequência de instruções que o processador realizará e o segundo arquivo contém o endereço e valor de cada posição da memória.

Para facilitar a geração de casos teste, criamos um programa em java que recebe como entrada a instrução MIPs, o numero do registrador ou constantes que se deseja utilizar para cada instrução do MIPs e gera um arquivo. mif com as instruções em hexa. O programa também inicializa o arquivo .mif para os valores que se quer colocar na memória. Em anexo um exemplo de entrada do programa.

Ao longo do desenvolvimento realizamos vários testes individuais para conferir o funcionamento de cada instrução implementada. Depois realizamos testes mais completos envolvendo várias instruções, esses testes estão na pasta 'Testes', os arquivos que possuem a terminação 'data' significam os valores da memória de dados (inst2.mif), os que não possuem essa terminação são os arquivos de instruções (inst.mit). Cada par (instrução e dado) deve ser substituído de acordo. Seguem alguns exemplos:

**Teste 1:** inicializamos a memória de dados com os valores 150 e 250, nas posições 0 e 1 respectivamente. Nas instruções realizamos um LOAD da posição 0 da memória para o registrador R1, depois outro LOAD da posição 1 da memória para o registrador 2, depois realizamos um ADD de R1 com R2 e salvamos em R3, depois damos um STORE do valor de R3 na memória na posição 3. Depois damos um novo LOAD da posição 3 para o registrador R4 para verificar a escrita em memória.

**Teste 2:** inicializamos a memória de dados com o valor 5 na posição 0. Para as instruções fazemos um LOAD da posição 0 da memória no registrador R1, depois fazemos outro LOAD da posição 0 para o registrador R2, depois realizamos um BEQ de R1 e R2 e pulamos

duas instruções. E então realizamos um ADDI de 1 no R1 e depois um ADDI de 2 no registrador 2.

**Teste 3:** neste teste realizamos teste com várias operações do tipo R. A primeira instrução é um ADDI de 10 em R1, a segunda é um ADD de R1 + R0 e salva em R2, a terceira é um ADDI de 5 no registrador R5, depois fazemos um SUB de R2 – R5 e salvamos em R3, depois fazemos um ADDI de 7 no R4, depois fazemos um AND, depois um OR, depois um XOR, depois um NOR, depois um SLL, depois um SRL, depois um STORE e por fim um LOAD.

**FIBBONACCI:** o próximo teste nós implementamos um Fibbonnacci, calculado utilizando apenas registradores. O número pedido foi o da posição 12 da sequência, que é 144 (90 em hexa), o valor final é salvo no registrador R2. Nessa função usamos BEQ e J para realizar o loop necessário.

**FIBBONACCI 2:** nesse teste modificamos o fibbonacci anterior para fazer um LOAD da memória de dados dos valores inicias e do valor que desejamos saber, utilizamos a função JAL para salvar o início da execução do fibbonacci e pularmos para os LOADS, depois utilizamos um JR para voltar ao início do fibbonacci e realizarmos a execução normal. Para esse teste também queremos saber do valor da 12ª posição do fibbonacci (144 em decimal e 90 em hexa). Salvamos o valor final no registrador R4.

# 4. Conclusão

O desenvolvimento do projeto teve duas grandes dificuldades. A primeira foi configurar e fazer com que o código dado funcionasse na FPGA, juntamente com entender o que estava implementado. A segunda grande dificuldade foi implementar a segunda memória, de dados, e inicializa-la na FPGA e criar as instruções LOAD e STORE. As demais instruções que tivemos que adicionar foram implementadas sem maiores dificuldades. Os testes sugerem que as instruções estão funcionando corretamente.