Caminho de Dados parte 1 e 2: MIPS de números inteiros completo, sem pipeline, em SystemVerilog e implementado na FPGA DE2-115 Altera

Leandro Lazaro Araújo Vieira (3513), Mateus Pinto da Silva (3489)

Ciência da Computação – Universidade Federal de Viçosa - Campus Florestal (UFV-caf) – Florestal – MG – Brasil

{leandro.lazaro, mateus.p.silva}@ufv.br

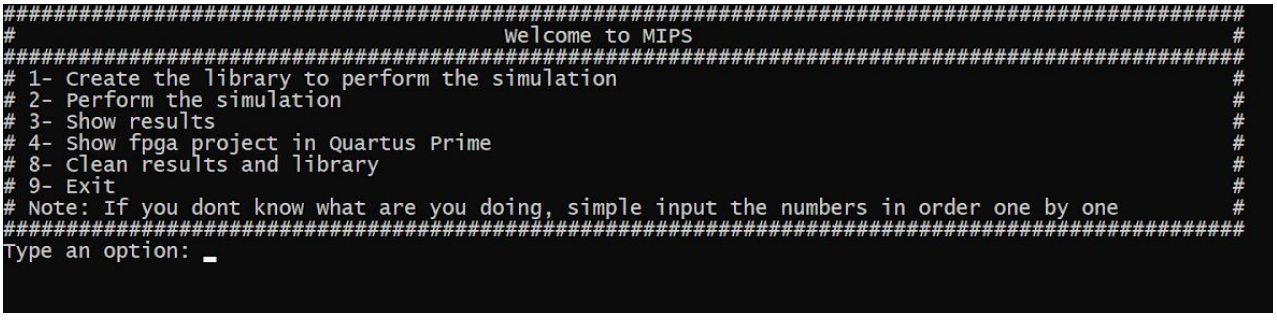
Resumo. *Este trabalho consiste em arquitetar um processador MIPS sem pipeline em SystemVerilog que consiga executar as instruções propostas e várias adicionais e implementá-lo na FPGA DE2-115 da Altera. Para isso, nós tomamos como base a divisão do processador com pipeline e aplicamos um módulo adicional de divisão e multiplicação. Também criamos um pequeno makefile em Python para executar os comandos, devido à ausência de make nativo para Windows, sistema operacional que executa a grande maioria dos softwares de design de arquitetura, inclusive os que usamos.*

1. Como simular e o funcionamento na FPGA

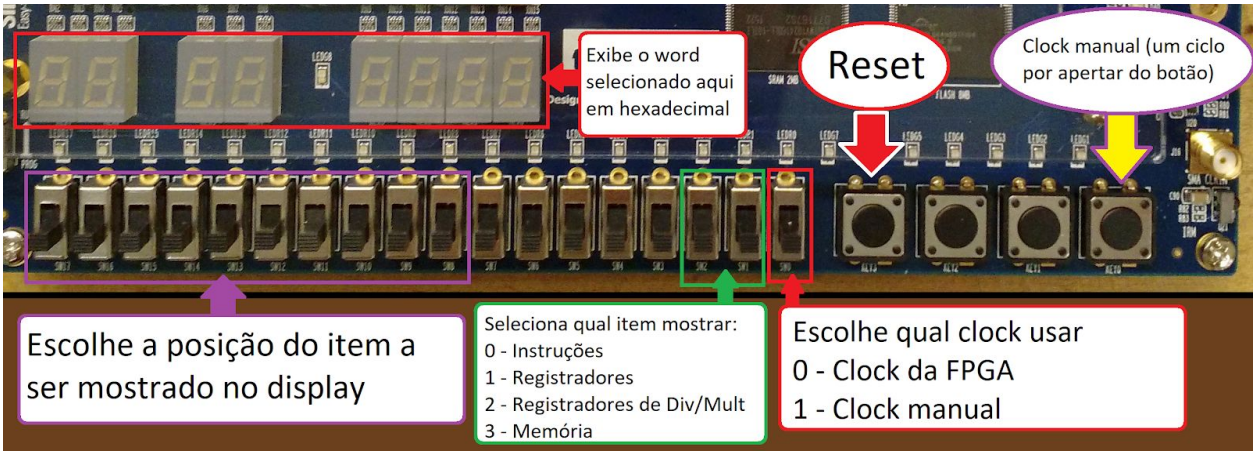
Para a execução correta do trabalho, é necessário:

- ModelSim PE Student Edition
- Quartus Prime Lite Edition (para testes na DE2-115)
- FPGA DE2-115 Altera
- Python 3 (para execução do makefile)
- Windows (todos os testes foram feitos usando a versão 10 PRO)

Para executar, apenas dê dois cliques no arquivo “make.py”.



Depois de transferir o arquivo sintetizado para a FPGA através do Quartus Prime, a placa deverá se comportar da seguinte forma:



2. A escolha da linguagem de descrição e as melhorias obtidas

Como permitido pelo Professor Nacif, preferimos usar a linguagem SystemVerilog por ser menos verbosa, apresentar uma incrível capacidade de descrever lógica combinacional de forma simples e por permitir criar blocos always específicos, o que permite sintetizar o código sem erros.

3. As instruções propostas e as adicionais implementadas

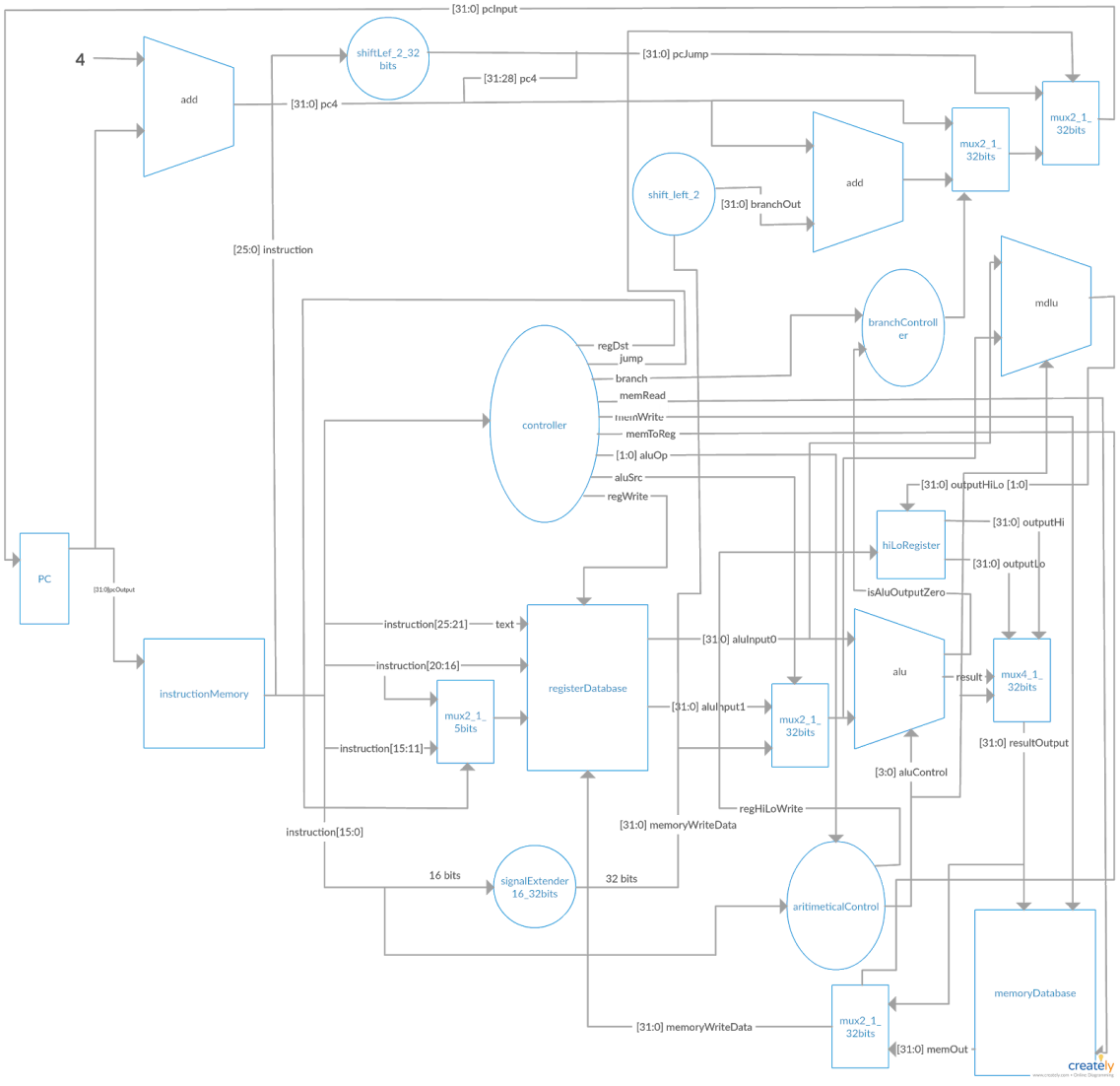
As instruções em fundo branco foram as pedidas na documentação, já as de fundo azul são as bônus:

ADD	LW	J
SUB	SW	MFLO
AND	BEQ	
OR	BNE	
SLT	LUI	
SLLV	ADDI	
SRLV	ANDI	
SRAV	ORI	
XOR	XORI	
MULT	SLTI	
DIV	MFHI	

4. A estrutura do processador

Como são muitos módulos, e o próximo trabalho prático será desenvolver o mesmo processador, porém usado pipeline, adotamos previamente a divisão dos cinco estágios com algumas pequenas mudanças no estágio de Instruction-Fetch e os incrementos no Execute.

Todos os arquivos do trabalho estão divididos em pastas com o mesmo nome dos estágios dentro da pasta *cores*, e cada estágio tem um arquivo de código com o respectivo nome, usado para referenciar os módulos e encapsular o estágio. As únicas exceções são: o controlador do processador, uma biblioteca (chamado de pacote na linguagem SystemVerilog) que contém todas as constantes do processador como o opcode das instruções e os módulos genéricos (como multiplexadores e etc). Além disso, há uma pasta com módulos de integração com fpga que serão explicados no tópico de Integração com FPGA, e um módulo que encapsula todo o MIPS.

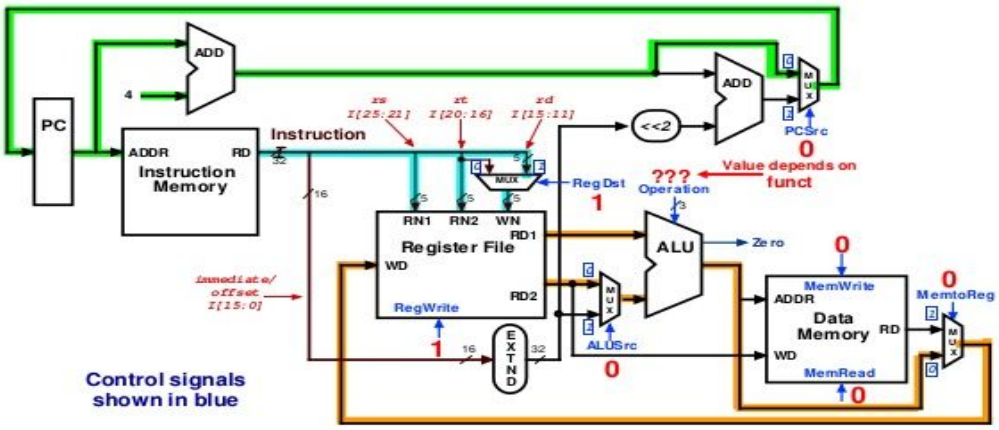


Estrutura do MIPS feito por nós

4.0. O controlador geral e as constantes do MIPS

O arquivo *libMips* contém todas as constantes usadas no processador, como o opcode de cada instrução, os funcs, códigos de controle da ALU e da MDLU, e etc, para fins de melhoria na legibilidade e redigibilidade do código.

O controlador geral do mips, chamado controller, recebe os seis bits menos significativos de cada instrução do Instruction-Fetch, ou seja, o opcode e gera a partir disso os sinais de controle, preparando o processador para realizar alguma operação.



Exemplo: sinais de controle de uma instrução do tipo R mostrados em azul.

4.1. O estágio Instruction-Fetch

O estágio busca a instrução correta e a passa para o próximo estágio, sendo ela um salto ou não. Como sinais de controle, o Instruction-Fetch, doravante IF, recebe **jump** e **branchC**. Como saídas, IF têm os 32 bits correspondentes a instrução que está sendo/será executada. Além disso, o estágio tem uma memória com todas as 256 instruções contidas no processador, e um IO para mostrá-las na FPGA.

Caso o fluxo de execução seja de uma instrução que não é um desvio, programCounter recebe, na borda de descida do clock, o valor de adderProgramCounter que é **PC+4** e a próxima instrução é carregada na saída do estágio.

Entretanto, se a instrução for um salto condicional, o controlador geral do processador enviará para o estágio 1 ou 2 no sinal de controle **branchC**, que são referentes a instrução **BNE** e **BEQ** do MIPS. Esse sinal é recebido pelo branchController, que combinando-o com o sinal de controle **isAluOutputZero** da saída da ALU, ativa um multiplexador que diz se o estágio deve saltar para a próxima instrução ou realizar o desvio condicional. Para fazer essa análise, branchController, caso receba sinal de controle para realizar um **BEQ** (branch if equals), avalia se **isAluOutputZero** é igual a 1, ou seja, se a subtração dos dois parâmetros a serem avaliados resultou em zero, o que significa que são iguais. Para **BNE**, algo similar é realizado, porém é verificado se o sinal **isAluOutputZero** é igual a 0, ou seja, se sao diferente.

Como última possibilidade, se a instrução for um salto incondicional, ou seja, um **J**, o estágio recebe 1 no sinal de controle **jump**, os 26 bits correspondentes ao

endereço são deslocados dois bits para a esquerda por um deslocador genérico e são concatenados com os quatro bits mais significativos do programCounter. Então um multiplexador escolhe entre a saída do salto condicional e o endereço do salto incondicional, usando o sinal de controle jump.

4.2. O estágio Instruction-Decode

Este estágio é responsável por decodificar a instrução, ou seja, preparar os dados que serão usados na execução, que é o próximo estágio. Para isso, o estágio transforma os endereços dos registradores em dados. Ademais, o estágio é responsável por salvar informações nesses mesmos registradores. Como sinais de controle, o estágio recebe **regDst** e **regWrite**, e tem como saídas os dados de dois registradores e o imediato de uma possível instrução do tipo **I** com sinal estendido para 32 bits. Além disso, o estágio contém saídas para IO de todos os registradores para integração com FPGA.

O módulo mais importante do estágio é o registerDatabase, que realiza leituras de forma assíncrona e gravações síncronas (em borda de subida do clock), o que possibilita fazer as duas coisas simultaneamente. **regWrite** diz se é preciso fazer uma gravação ou não naquela borda de subida.

Não muito menos importante, **regDst** diz qual parte do código da instrução corresponde ao registrador de destino, ou seja, o registrador que terá sua memória escrita. Esse tratamento é feito por um demultiplexador genérico de 32 bits ligado à entrada do endereço de escrita de registerDatabase.

4.3. O estágio executing

Executing é o estágio responsável por realizar todas operações lógicas aritméticas e é utilizado em quase todas as instruções, exceto as todo tipo j, como o jump, por exemplo. Ele contém 6 módulos e dentre eles estão aritimeticalControl, alu, mdlu e hiLoResgister que merecem certo destaque.

O módulo aritimeticalControl é o controlador usado para decidir qual operação deve ser realizada pela alu ou a mdlu e se os registradores especiais **HI** e **LO** devem ser escritos ou não. Para decidir isso, esse controlador avalia o sinal de um fio de quatro bits proveniente do controller e, na maioria dos casos, os seis bits menos significativos da instrução. É importante esclarecer que o fio aluOp possui quatro bits para que os seis bits menos significativos de algumas instruções seja ignorado e a decisão de qual operação aritmética executar seja tomada exclusivamente por ele.

A alu módulo é utilizado para realizar quase todas as operações reconhecidas pelo aritimeticalControl, sendo elas **ADD**, **SUB**, **AND**, **OR**, **XOR**, **NOR**, **LUI**, **SLL**, **SLT**, **SRA** e **SRL**, não sendo responsável apenas por realizar as operações **MULT** e **DIV**. Também é importante relembrar que a saída **isAluOutputZero**, já citada anteriormente, é parte responsável pela decisão de desvios condicionais e está presente neste módulo. Como isso já foi detalhado anteriormente no tópico **4.1.**, não será explicado novamente.

O mdlu é o módulo capaz de realizar **MULT** e **DIV**, operações não incluídas na lista de execução do alu módulo. Embora que para alguns pareça ser um equívoco dedicar uma unidade lógica apenas para duas operações, isso foi uma escolha

inteligente, pois mesmo que essas duas operações estejam sendo executadas de forma combinacional, devido a sua natureza, elas possuem a peculiaridade de salvar seus resultados em registradores específicos chamados **HI** e **LO**, operadores que não são utilizados por qualquer outra instrução, a não ser a **MFHI** e a **MFLO** que os extraem para um registrador da lista de registradores acessíveis às demais instruções.

hiLoResgister é o módulo responsável por controlar os registradores especiais **HI** e **LO**. Esse módulo apenas são escritos quando uma operação **MULT** ou **DIV** está sendo executada na mdlu, então, como é de se imaginar, o aritimeticalControl dedica um fio para este módulo que, baseado no seu sinal, autoriza ou não a escrita.

4.4. O estágio memory

É o estágio responsável por gerenciar a memória do processador. Como único estágio de controle, ele contém o fio **memWrite**, que diz se é necessário salvar o valor na memória, dado o conteúdo a ser salvo e o endereço, de forma similar ao banco de registradores do Instruction-Fetch. Há uma saída para IO de todas as posições da memória para integração com FPGA.

Também conta a possibilidade de ler um valor da memória dado seu endereço de forma assíncrona, tornando-o capaz de realizar suas duas funções de forma simultânea.

4.5. O estágio Write-Back

Este estágio é responsável por dizer se o valor a ser salvo no banco de registradores será o da saída da ALU ou da saída da leitura da memória. Recebe como sinal de controle o **memToReg** e usa um multiplexador genérico para fazer isso.

5. Implementação em FPGA, os problemas enfrentados e suas soluções

Para execução na DE2-115, foi necessário definir como exibir as informações. Escolhemos mostrar todas as instruções, registradores (normais e de multiplicação/divisão) e posições da memória do MIPS em hexadecimal usando o display de sete segmentos da placa. Como um word do processador contém 32 bits, seriam necessários oito números hexadecimais, ou seja, exatamente a quantidade de displays da FPGA. Entretanto, desreferenciar todas essas informações diretamente no projeto seria impossível, visto que a criação de links (matrizes de fios) não pode ser realizada pelo Quartus Prime, então foi criado um módulo em SystemVerilog para isso.

Outro problema seria definir qual clock usar. O mais rápido possível, usando como referência apenas o caminho crítico, exibiria muito bem o desempenho da lógica, porém seria muito complicado debugar um código no MIPS (ou debugar o próprio processador). Como solução, usamos um multiplexador chamado clockChooser (localizado na pasta *fpgaIntegration/fpgaController*) que permite escolher entre o clock máximo e um clock manual feito por um botão na FPGA, o que permite usar o processador em modo normal ou modo debug. Graças ao software de síntese, não é necessário nenhum tratamento adicional, visto que ele reconhece que é um demultiplexador de clock e faz esse tratamento automaticamente.

O projeto do Quartus Prime está na pasta raiz, e o arquivo chama-se *simpleMips.qpf*.

5.1. Criando as saídas para a FPGA

Todos os IOs do processador (instruções, registradores normais e de multiplicação/divisão e memória) são multiplexados para serem exibidos no display da FPGA. Para isso, é usado o módulo infoChooser (que está na pasta *fpgaIntegration/fpgaController*). Usando como entradas **select** (que seleciona entre memória, registradores, etc) e **derreference** (que escolhe a posição do objeto selecionado), uma saída de 32 bits com o word desejado é carregado usando lógica combinacional.

Depois disso, o módulo displaySevSegm converte esse word em saídas compatíveis com os oito displays da DE2-115. Para isso, são usados oito módulos conversores simples de binários de quatro bits para hexadecimal em display de sete segmentos chamados binaryToHexSevSegm. Ambos módulos estão na pasta *fpgaIntegration/displaySevSegm*.

6. Conclusão

Através deste trabalho, desmistificamos a ideia que tínhamos do funcionamento de um computador/processador. Antes, víamos o funcionamento no nível de hardware como algo mágico. Depois do trabalho, percebemos que é um funcionamento relativamente simples e muito literal, pelo menos nos processadores RISC. Outra coisa importante foi perceber que tudo o que é executado em um computador, no fim, são operações matemáticas elementares, como somas, subtrações e operações lógicas.

Além disso, tivemos a chance de trabalhar com o Verilog atual usado pela Intel, o SystemVerilog, que resolve grandes problemas do Verilog comum, como os blocos always que às vezes não são lidos de forma correta na linguagem mais antiga. Além disso, ele possui a facilidade de identificar automaticamente e diferenciar fios de registradores, falhando unicamente em criar links, como foi percebido durante as semanas de desenvolvimento do trabalho. Também aprendemos a usar o ModelSim, simulador gratuito para estudantes dessa linguagem.

Também achamos extremamente interessante a aplicação prática dos conceitos RISC que são muito teóricos quando vistos em sala e até aparentemente pouco importantes. Porém, quando usados na prática, mostram que são muito bem-vindos, tornando o desempenho do processador muito mais rápido e seu desenvolvimento mais simples.