

Caminho de Dados parte 1 e 2: MIPS de números inteiros completo, com 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

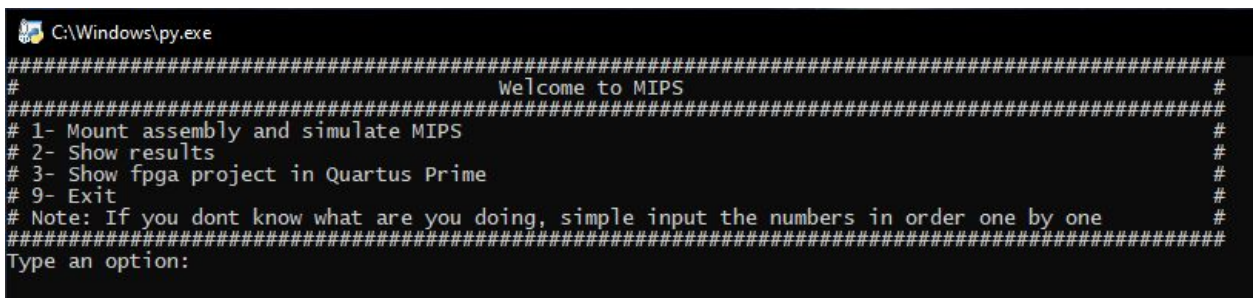
Resumo. *Este trabalho consiste em arquitetar um processador MIPS com pipeline em SystemVerilog que consiga executar as instruções propostas e várias adicionais e implementá-lo na FPGA DE2-115 da Altera. Para isso, nós dividimos o processador em estágios. Também criamos um pequeno makefile em Python 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”.



```
C:\Windows\py.exe
#####
#                               welcome to MIPS                               #
#####
# 1- Mount assembly and simulate MIPS                                     #
# 2- Show results                                                         #
# 3- Show fpga project in Quartus Prime                                  #
# 9- Exit                                                                #
# Note: If you dont know what are you doing, simple input the numbers in order one by one #
#####
Type an option:
```

Imagem 1.0: Menu do trabalho

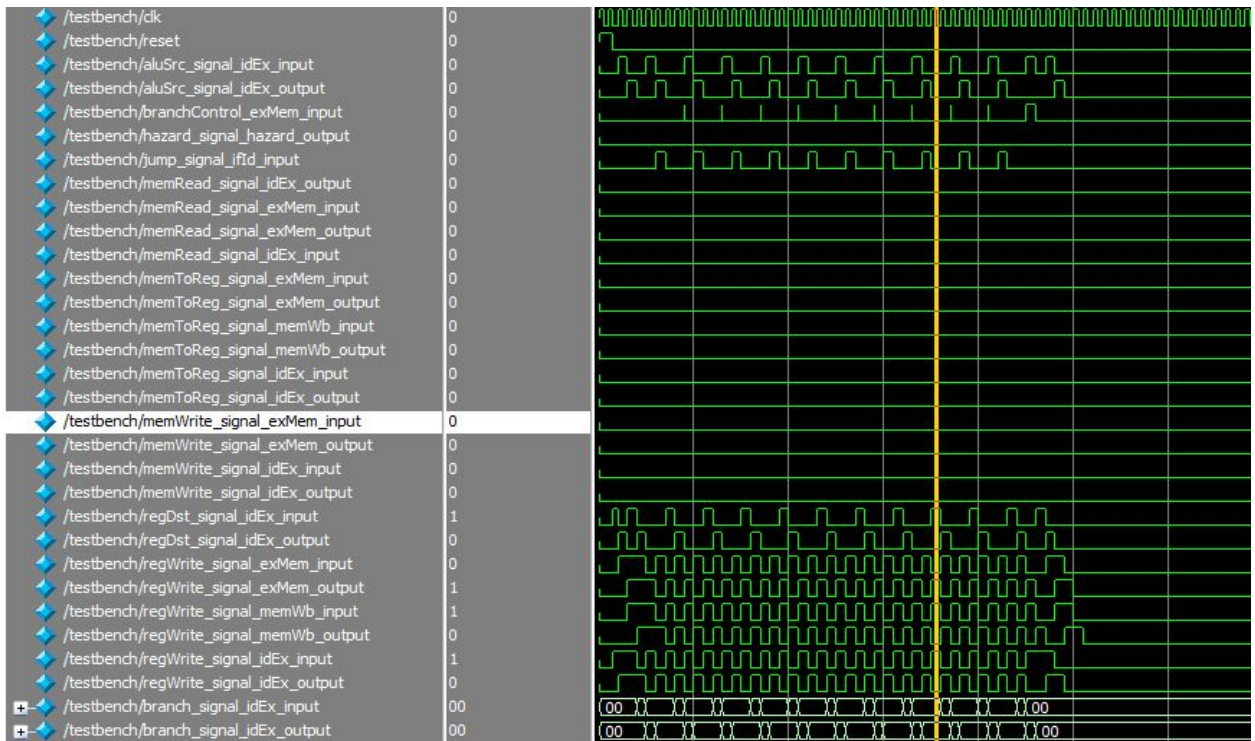


Imagem 1.1: Simulação de ondas do MIPS pipeline no ModelSim

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

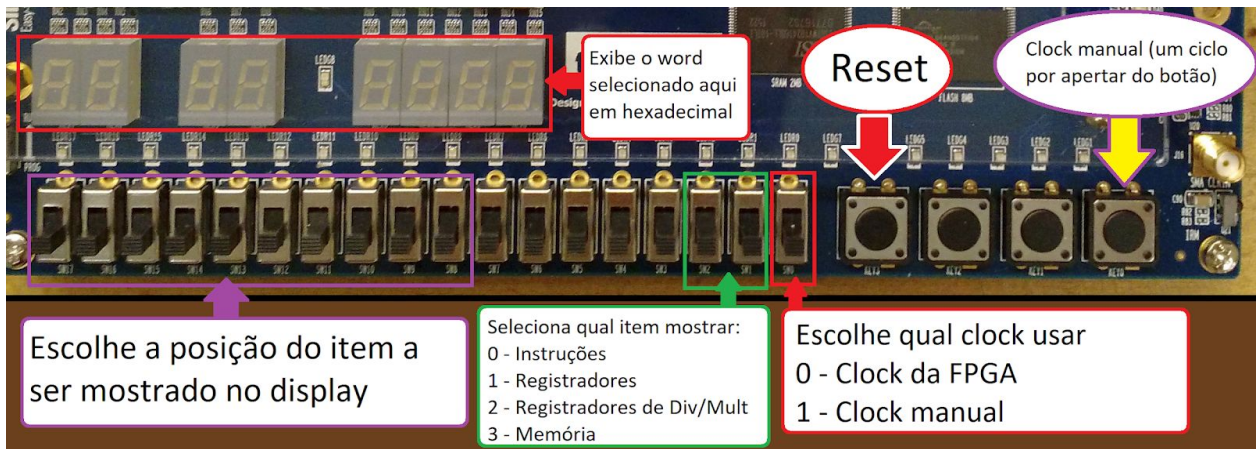


Imagem 1.2: Manual de como utilizar o MIPS na FPGA

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
SUB	SW
AND	BEQ
OR	BNE
SLT	LUI
SLLV	ADDI
SRLV	ANDI
SRAV	ORI
XOR	XORI
J	
SLTI	

Imagem 3.0: Instruções suportadas pelo nosso MIPS

4. A estrutura do processador

O processador é dividido em cinco estágios, sendo eles: Instruction-Fetch, Instruction-Decode, Executing, Memory e Write-Back. Cada um desses estágio, com exceção do Instruction-Fetch, possui um conjunto de registradores nomeados de registradores de pipeline. Além desses componentes, o processador contém também uma unidade de Hazard e uma de Forwarding. Cada um dos itens citados serão posteriormente explicados e, se necessário, detalhados nos tópicos seguintes.

Além disso, também seguimos um padrão para nomenclatura dos fios que conectam os módulos do processador. Por exemplo, **memToReg_exMem_input**, string que representa um fio, pode ser dividida da seguinte forma: **[Sinal]_[Estágio no Pipeline]_[Sentido]**. O **Sinal** indica que tipo de informação o fio transporta, o **Estágio no Pipeline** a origem do fio e o **Sentido** indica se o fio está entrando ou saindo do estágio de origem. Aplicando esse conhecimento no exemplo citado anteriormente, podemos deduzir que o fio sinaliza se deve ser escrita alguma informação da memória em algum registrador, diz também que ele é fornecido pelo estágio **exMem** e que ele, o fio, está entrando nesse estágio.

Quanto aos arquivos do trabalho, todos estão divididos em pastas com o mesmo nome. Os registradores de pipeline estão dentro da pasta *pipelineRegisters* e os estágios estão dentro da pasta *stages*, 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: a unidade Hazard, unidade de Forwarding, 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.

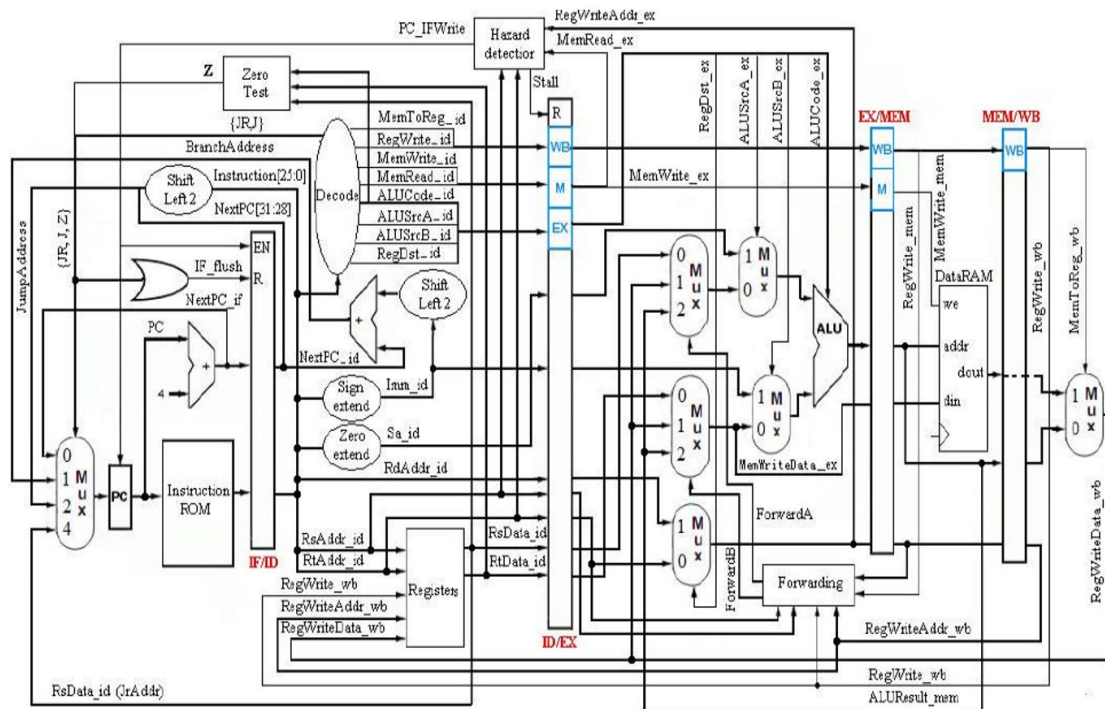


Imagem 4.0: Estrutura do MIPS feito por nós

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 funco, códigos de controle da AL, e etc, para fins de melhoria na legibilidade e redigibilidade do código.

O controlador geral do mips, chamado controller, presente no estágio Instruction-Decode, recebe os seis bits menos significativos de cada instrução fornecidos pelo estágio Instruction-Fetch, ou seja, o opcode e gera a partir disso os sinais de controle que serão carregados nos registradores de pipeline do estágio Executing no próximo ciclo de clock. Esses sinais são cruciais para os próximos estágios realizarem qualquer operação.

4.1. Os registradores de pipeline

Cada estágio, com exceção do Instruction-Fetch, possui seu respectivo módulo de registradores de pipeline. Esses registradores são, provavelmente, o item mais importante para o sucesso do pipeline, pois são eles os protagonistas no deslocamento das instruções ao longo dos estágios. Nosso MIPS conta com 4 deles, sendo: IF_ID (precede o estágio Instruction-Decode), ID_EX (precede o estágio Executing), EX_MEM (precede o estágio Memory) e MEM_WB (precede o estágio Write-Back).

Embora existam quatro módulos de registradores de pipeline, todos cumprem a mesma função de guardar os valores necessários para a função a ser executada no estágio respectivo. Devido a essa similaridade, é desnecessário falar sobre todos individualmente neste tópico. No tópico 4.7, os módulos IF_ID e ID_EX serão citados novamente pois possuem uma pequena peculiaridade que não cabe a este tópico explicar.

4.2. O estágio Instruction-Fetch

O estágio busca a instrução correta, sendo ela um salto ou não, e a passa para os registradores do próximo estágio no próximo ciclo de clock. Como sinais de controle, o Instruction-Fetch, doravante IF, recebe **jump_signal_ifId_input** e **branchControl_exMem_input**. Como saídas, IF tem 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 subida do clock, o valor de adderProgramCounter que é **PC+4** e a próxima instrução é carregada na saída do estágio.

Entretanto, se o endereço da próxima instrução é fornecido por uma instrução de salto condicional (uma instrução **BNE** ou **BEQ**), o sinal de controle **branchControl_exMem_input** será “1” e então programCounter, ao invés de **PC+4**, receberá o valor de **pcBranch_exMem_input**, fio de 32 bits que contém o endereço do salto.

Como última possibilidade, se a instrução for um salto incondicional, ou seja, um **J**, o estágio recebe 1 no sinal de controle **jump_signal_ifId_input** e programCounter recebe o valor de **pcJump_signal_ifId_input**, fio de 32 bits que contém o endereço do desvio.

4.3. O estágio Instruction-Decode

Este estágio é responsável por decodificar a instrução, ou seja, preparar os dados que serão usados no Executing, 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 **regWrite_signal_memWb_output** e tem como saídas os dados de dois registradores, o imediato de uma possível instrução do tipo **I** com sinal estendido para 32 bits e todos os

sinais do controller. 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_signal_memWb_output** diz se é preciso fazer uma gravação ou não naquela borda de subida.

Além disso, esse estágio também é responsável por verificar se a instrução atual é do tipo J. Se for, o endereço da próxima instrução no estágio Instruction-Fetch será definido por ela.

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.

4.4. 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 4 módulos e dentre eles estão aritimeticalControl e alu que merecem certo destaque.

O módulo aritimeticalControl é o controlador usado para decidir qual operação deve ser realizada pela alu. 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**. Além da saída convencional, a alu também possui a saída **isAluOutputZero**. Esse sinal indica se o resultado da operação executada é igual a zero ou não. Para instruções de desvio condicional, esse dado é extremamente importante, pois ele é responsável por decidir se o imediato desse tipo de instrução definirá a próxima instrução no estágio Instruction-Fetch,

4.5. O estágio **memory**

É o estágio responsável por gerenciar a memória do processador. Como único estágio de controle, ele contém o fio **memWrite_signal_exMem_output**, 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.

4.6. 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_signal_memWb_output** e usa um multiplexador genérico para fazer isso.

4.7. A unidade de hazard

Apesar do MIPS ser um dos projetos mais bem feitos quando se trata de arquitetura RISC e não possuir hazards estruturais, infelizmente possui alguns tipos hazards de dados.

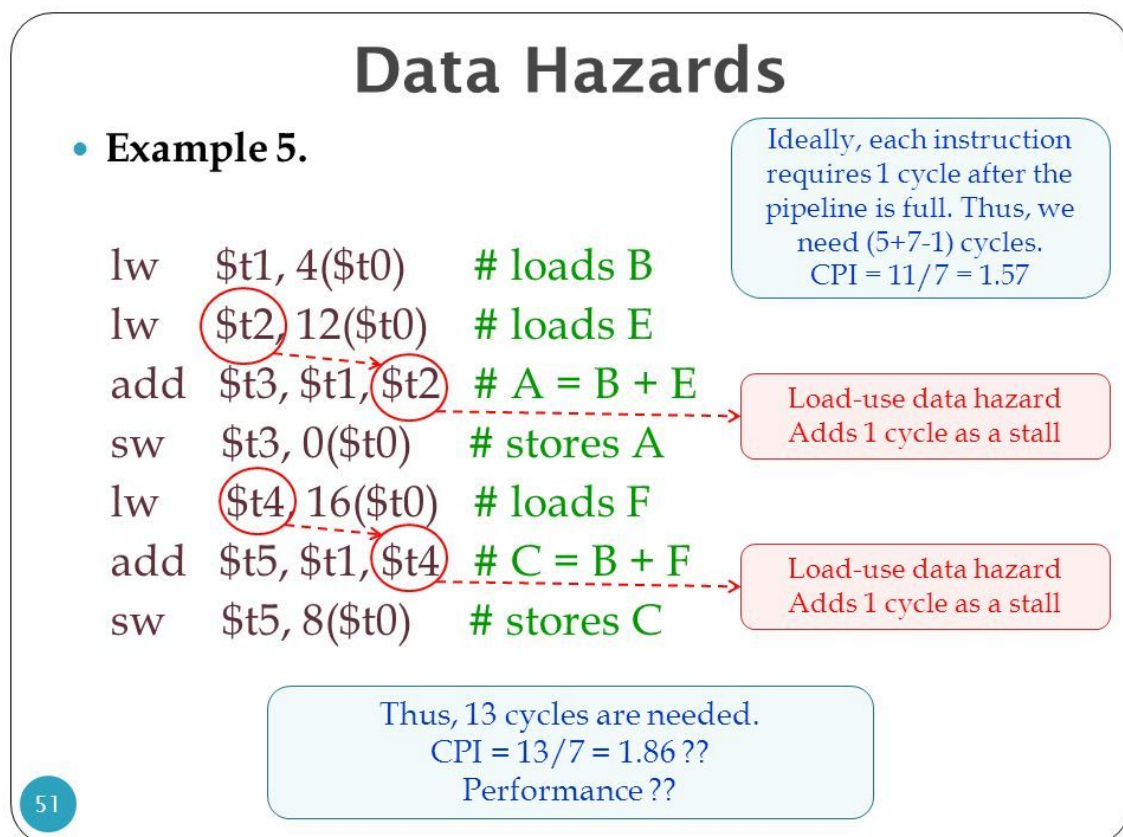


Imagem 4.6.0:Exemplos de hazard

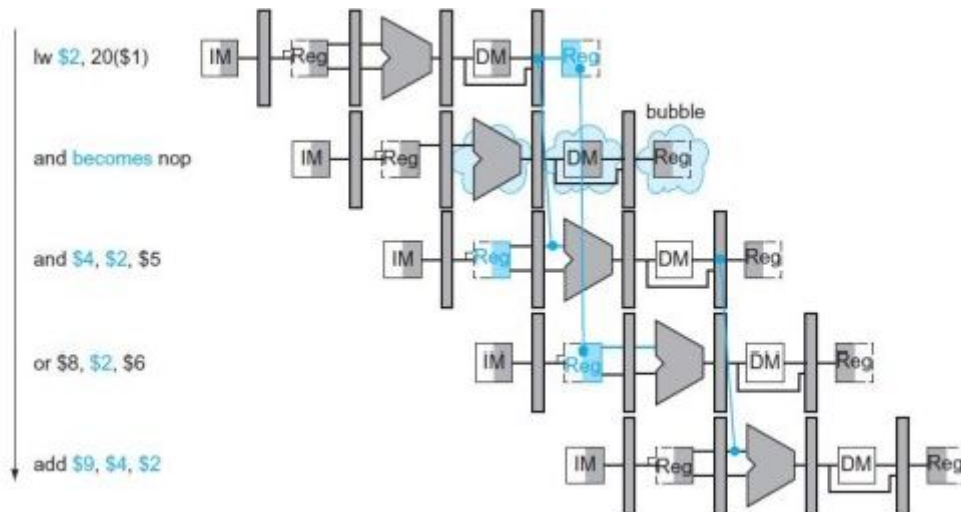


Imagem 4.6.1: Exemplo de hazard e forwarding

Como é possível notar na figura 4.6.0., embora a segunda instrução tenha, teoricamente, modificado o registrador \$t2, a terceira instrução utilizou o valor que havia antes da suposta modificação. Isso acontece porque a terceira instrução já está no estágio Executing enquanto a segunda ainda está no Memory, ou seja, ela sequer ainda armazenou o novo valor no registrador utilizado pela terceira instrução. Para resolver esse problema são utilizados dois módulos: Hazard e Forwarding. Porém, como estamos no tópico “**A unidade de hazard**”, por enquanto falaremos apenas do primeiro.

Parte da resolução desse problema consiste em incluir uma bolha entre as duas instruções dependentes, isto é, durante um ciclo de clock apenas, continuar deslocando a mais antiga e manter a mais recente onde está. No entanto, como é possível notar na imagem 4.6.1 o problema ainda não foi resolvido e seria necessário mais 2 bolhas para ter sucesso, porém, isso tornaria a execução de quase todos algoritmos lento demais. No próximo tópico será apresentada a unidade responsável por resolver a outra parte desse problema de forma eficiente.

4.6.8. A unidade de forwarding

Ao observar a imagem 4.6.1 é possível notar que ao ser aplicada uma bolha entre duas instruções dependentes, ainda não foi o resolvido o problema apresentado no tópico 4.6.7., mas também é possível notar que, embora os dados que a instrução AND precisam não estejam no banco de registradores, eles já estão prontos, só que em outro estágio. Portanto, o que a unidade de forwarding faz quando existe uma dependência, e a “distância” entre as instruções dependentes é de um estágio apenas, é encaminhar o valor que está causando a dependência por um caminho alternativo. Esse caminho alternativo é sempre entre o estágio Executing e Memory ou entre o estágio Executing e Write-Back. Uma observação importante é que nem todas as instruções dependentes precisam de utilizar hazard e forwarding, mas toda instrução dependente utiliza a

unidade de forwarding, ou seja, a unidade de hazard é utilizada apenas quando a “distância” entre as instruções dependentes é de dois estágios.

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 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 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 mais uma vez a ideia que tínhamos do funcionamento de um computador/processador com pipeline. Depois do trabalho, percebemos que o funcionamento do pipeline, pelo menos nos processadores RISC, é relativamente simples e muito literal, pois funciona basicamente como um deslocador.

Além disso, tivemos a segunda 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 conseguimos imaginar o quão difícil e complicado deve ser o dia-dia de projetistas de processadores que são utilizados pelas pessoas cotidianamente nos computadores, isto é, os AMD64 que são CISC. Um projeto como o nosso, que é extremamente pequeno se comparado com outros processadores, no seu final, ficou extremamente complicado de analisar no simulador. Se não tivéssemos o costume de programar e debugar estaríamos em maus lençóis.