UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

DEPARTAMENTO DE TECNOLOGIA

TEC499 MI DE SISTEMAS DIGITAIS

PROBLEMA 02 – MICROARQUITETURA DE PROCESSADORES

Beatriz de Brito Santana, Franklin Lázaro Santos de Oliveira

Tutor: Thiago Cerqueira de Jesus

1 INTRODUÇÃO

Este documento descreve uma visão geral da arquitetura e das análises do projeto Microarquitetura de Processadores. O projeto em questão visou uma nova fase de desenvolvimento de um microprocessador baseado na arquitetura RISC; dessa vez focando na sintetização das especificações desenvolvidas na etapa anterior.

Nesta fase, foi solicitada a entrega da descrição do processador em linguagem Verilog, com os seus respectivos *test benches*. Como requisitos deveria ser implementadas as instruções presentes nos programas de teste da fase anterior; com uma taxa de operação do processador de 100 MIPS e a execução paralela de cinco instruções a cada ciclo de *clock*.

O processador possui um *datapath* definido baseado no modelo de funcionamento MIPS, executa instruções de forma paralela, possui tratamento para conflitos de controle e de dados, com testes para cada bloco. Além da descrição das configurações de ambiente e uma breve descrição da arquitetura utilizada, este documento possui também uma análise dos componentes desenvolvidos.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 VERILOG

É uma linguagem de descrição de *hardware* que permite a realização da descrição de sistemas digitais. Desenvolvida em 1980 pela empresa Gateway, é utilizada universalmente. Sua sintaxe é similar à linguagem de programação em C e possui como grande diferencial das outras linguagens comuns o fato de que a execução não é feita completamente de forma sequêncial (como em blocos de repetição, por exemplo), mas a sua maior parte é executada de forma paralela.

2.2 TEST BENCH

É um código de descrição de hardware (HDL) que permite a geração de estimulos e entradas para os módulos de forma a permitir a análise dos resultados. Para isso ele pode conter desde uma variável de *clock*, a registradores, entradas, saídas e estruturas de repetições para a realização do teste.

2.3 CONFLITOS/HAZARDS

Por conta da utilização de uma estrutura de *pipeline* no processador, é comum que ocorram conflitos entre as instruções, os chamados *hazards*.

Um tipo de conflito que pode existir é o conflito de dados. Este ocorre quando mais de uma instrução está tentando utilizar um mesmo registrador, como forma de solução pode-se usar o chamado *forwarding* ou "adiantamento de dados", onde o valor do registrador é passado para a instrução seguinte antes de ser gravada, ou seja, após cálculada no estágio *EX* do pipeline.

Existe também o conflito de controle, neste caso temos em específico as intruções de *branch*, que modificam o valor do PC para pular para uma instrução mais adiante dependendo do resultado de uma comparação entre dois registradores. Uma opção é causar uma bolha para esperar o resultado dessa comparação, outra é realizar uma predição considerando que ele sempre será tomado ou não, dependendo do desejo do projetista (neste projeto foi usado a predição de desvio "nunca tomado"). Neste último método é necessário desfazer as ações tomadas caso a predição esteja errada.

3 METODOLOGIA

3.1 VISÃO GERAL DO PROCESSADOR

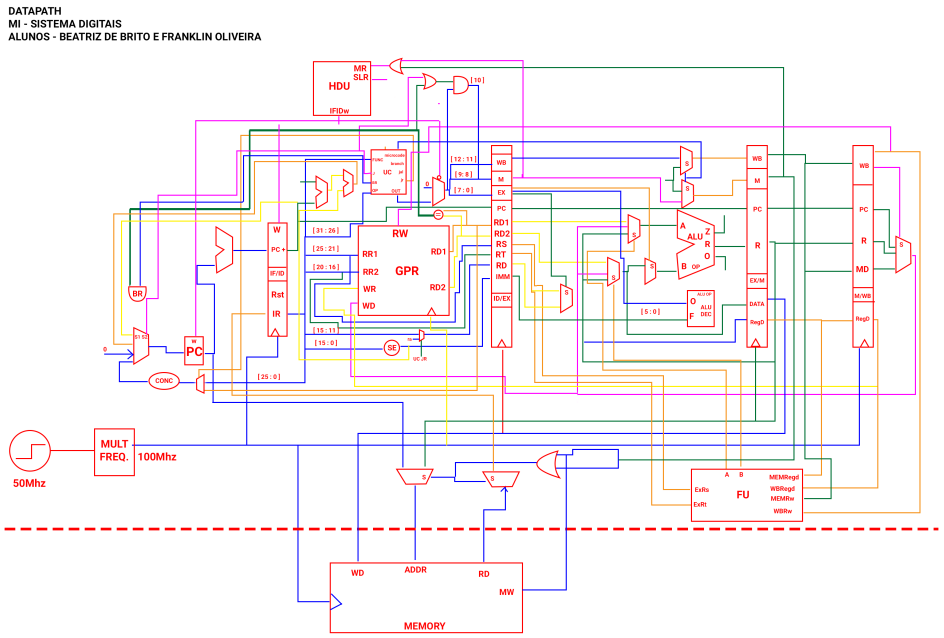
Para implementação inicial, nesta fase em questão, foi solicitado a implementação de apenas as instruções utilizads nos 6 programas de testes desenvolvidos na fase anterior: *bubble sort*, fatorial, fibonacci, potência de um número, primos e cálculo de raíz. Para isto, foram implementadas as seguintes instruções:

- ADD
- SUB
- MFHI
- DIV
- MUL
- SLT

Essas instruções seriam utilizadas durante a execução dos programas de teste. Como requisito foi solitictado permitir uma execução de forma paralela, ou seja, mais de uma instrução seria executada a cada clico de *clock* (mais precisamente, 5 instruções por ciclo). Para tal feito foi necessário a implementação de um processador em pipeline, causando algumas alterações no *datapath* anterior, já que seriam necessárias as implementações de registradores entre os estágios e também de uma unidade de forwarding e uma unidade de hazard (estruturas necessárias para o tratamento de conflitos de dados e conflitos de controle). O novo *datapath* pode ser visualizado na Figura 1. A imagem pode ser visualizada de forma ampliada neste link: (<https:

//goo.gl/uoOiF9>). As cores foram escolhidas de forma a facilitar a visualização do caminho de linhas distintas que se cruzam.

Figura 1 – Datapath



Fonte: próprio autor

3.2 ANÁLISES

3.2.1 Caminho crítico

O caminho crítico do projeto segue o mesmo caminho do MIPS. A instrução de load, responsável por carregar o dado de uma memória para um registrador é a que leva o maior tempo de execução por usar todos os estágios do pipeline: IF, ID, EX, MEM e WB.

3.2.2 Frequência de operação

No problema, foi proposto que o processador deve executar 100 milhões de instruções por segundo (MIPS). Com essa informação, foi possível definir a frequência do mesmo a partir dos cálculos a seguir:

Tabela 1 – Informações para cálculo de frequência

Número de instruções	**tempo gasto(s)**
100.000.000	1
1	t

t = 1/100.000.000 = 10e-9s = **10ns**

f = 1/t = 1/10e-9 = **100Mhz**

Foi calculado então que o processador deve operar a uma frequência de 100Mhz, onde cada ciclo de clock dura 10ns. Porém, pelo fato do FPGA Cyclone IV EP4CE30 possuir um clock máximo de 50Mhz, é necessário o uso de um multiplicador de frequência para atingir 100Mhz.

3.2.3 Throughput

A cada ciclo de *clock* temos um acesso à memória, seja ele no estágio de *instruction fetch* ou *memory access*. Com a velocidade de operação de 100 MIPS para o circuito e transferência de dados de 32 bits, pode-se encontrar o *throughput* da seguinte forma:

throughput = 32 x 100.000.000 = 3.200.000.000 bits/s = **3,2 Gbits/s**

3.2.4 Síntese lógica

A Tabela 2 faz referência a LUT de acordo com quantidade de elementos lógicos utilizados. Já a Tabela 3 mostra a quantidade desses elementos lógicos dividivo entre o modo normal e o modo aritmético, sendo que a Tabela 4 exibe o total desses elementos.

Tabela 2 – Elementos lógicos por entrada

Logic element usage by number of LUT inputs	
<=2 input functions	266
3 input functions	1956
4 input functions	2538

Tabela 3 – Total de Elementos por modo

Logic elements by mode	
arithmetic mode	739
normal mode	4021

Tabela 4 – Estimativa de LE's

Estimated Total logic elements	
Total combinational functions	4,760
Dedicated logic registers	319
Total	4,860

3.2.5 Síntese física

A Tabela 5 faz referência aos elementos físicos que seriam utilizados caso o projeto fosse implementado. Podemos obsevar por exemplo a quantidade de pinos da FPGA, a quantidade de

memória, etc.

Tabela 5 – Análise dos componentes físicos

Total registers	319
Total pins	66 / 329 (20 %)
Total virtual pins	0
Total memory bits	524,390 / 608,256 (86 %)
Embedded Multiplier 9-bit elements	6 / 132 (5 %)
Total PLLs	0 / 4 (0 %)

4 RESULTADOS E DISCURSÃO

4.1 AMBIENTE DE EXECUÇÃO E TESTES

O ambiente utilizado para o desenvolvimento e teste do projeto em questão seguiu as seguintes configurações:

- Sistema Operacional: Windows 10;
- Quartus Prime 16.0 Lite Edition + Cyclone IV device support;
- ModelSim-Altera 10.4d (Quartus Prime 16.0);

Para atingir os resultados esperados, é recomendável a instalação do Quartus e ModelSim no diretório *C:/altera/16.0*. Para compilar e realizar as análise no Quartus usa-se os passos *File > Open Project... > ../PBL2/PBL.qpf* e para simular o processador no ModelSim deve-se seguir os passos *Jumpstart > Open a Project > ..PBL2/PBL*. Na pasta do projeto encontram-se todos os códigos em Verilog e os respectivos *test benches*.

4.2 SIMULAÇÃO DOS PROGRAMAS TESTES NO MODELSIM

1. Executar o Assembler.exe e montar o programa desejado;
2. Simular o programa com o Simulator.exe;
3. Simular o arquivo SoC_tb no modelsim(**comando**: vsim work.SoC_tb);
4. Rodar a simulação até a finalização da execução do programa(**comando**: run -all);
5. *Are you sure you want to finish?* NO;
6. Verificar os valores na memória (Memory List > /SoC_tb/CHIP/MEMORY/../mem_data) e compará-los com os simulados em *Simulator.exe*;

7.Verificar os valores no GPR (Memory List > /SoC_tb/CHIP/MEMORY/GPR/registers) e compará-los com os simulados em *Simulator.exe*;

Observações:

- No processador foi incluída a instrução HALT (32'b1 ou -1), então existirá sempre uma linha de discrepância entre a memória do processador e a simulada em Simulator.exe, porém o resultado não é afetado;
- Os arquivos Assembler.exe e Simulator.exe estão dentro da pasta Montador-Simulador, que fica dentro da pasta do projeto;

4.3 TESTES

Como solicitado pelo cliente, inicialmente, foram implementadas apenas as instruções que estavam presentes nos testes da fase anterior. Para a realização dos testes dessas instruções no módulo da ULA, foi usado um laço de repetição que, com a função *random*, gerava números aleatórios que eram calculados e o resultado comparado com o resultado que saía ULA para o mesmo cálculo. A utilização dessa função permitiu a verificação do teste para um conjunto de números variados.

Além da simulação do arquivo SoC_tb (que é o arquivo *top-level* do projeto), também foram testados os módulos de forma separada através dos *tests bench*, o que garantiu uma melhor confiabilidade no código e uma maior facilidade na hora de realizar a integração dos módulos. o projeto pode ser visualizado em todos os seus módulos através no ModelSim do arquivo stagesWave_out.do.

5 CONCLUSÃO

O projeto se mostrou bastante desafiador, visto que a implementação de um processador com *pipeline* envolve vários elementos como a unidade de controle de conflito, o forwarding, os registradores intermediários, etc.

A implementação foi iniciada com o desenvolvimento dos blocos individuais e o teste dos mesmos, afim de evitar a dificuldade em localizar erros quando os blocos fossem integrados. Os *test bench* se mostraram de grande ajuda para a identificação de erros e falhas na implementação, como por exemplo a correção da utilização errada de entradas e saídas da ULA.

Foi possível expandir os conhecimentos além da base teórica, aprendendo novas ferramentas como o uso do ModelSim e a implementação de códigos em linguagem HDL. Em conclusão, o projeto trouxe bastante conhecimento tanto teórico quanto prático, o que beneficia em entender como aplicar os conceitos aprendidos e como lidar com erros comuns e distintos que ocorreram durante a implementação.

REFERÊNCIAS

PATTERSON, D. A. HENNESSY, J. L. **Computer Organization and Design, Fifth Edition: The Hardware/Software Interface**. Ed. Elsevier, 5a ed., 2014.