PROJETO FINAL – DESENVOLVIMENTO DA UNIDADE OPERATIVA MULTICICLO MIPS E IMPLEMENTAÇÃO EM VHDL – KIT DE2-70

Organização e Arquitetura de Computadores – Turma C – 2/2016

Arthur Henrique Aguiar Pereira – 10/0054013 – Engenharia de Computação (E-mail: arthurpereira @hotmail.com)
Pedro Yan Ornelas de Oliveira – 14/0158995 – Engenharia de Computação (E-mail: pedroyane@gmail.com)
Pietro Bertarini de Carvalho Mota – 14/0159118 – Engenharia de Computação (E-mail: pietromota@hotmail.com)

Professor: Ricardo Pezzuol Jacobi

Brasília, 19 dezembro de 2016
Universidade de Brasília

Resumo Este trabalho descreve uma implementação Multiciclo de um subconjunto da ISA MIPS em VHDL, utilizando o Kit Altera DE2-70.

Palavras Chaves: Processador Multiciclo, MIPS, controle

1 OBJETIVOS

Juntar os módulos principais da unidade operativa e de controle que compõe a arquitetura MIPS Multiciclo, de forma que estes módulos sejam capazes de processar conjuntos de instruções binárias de 32 bits pré-compiladas em formato MIF – Memory Init File. Além disso, acrescentar um conjunto determinado de instruções ao projeto que não foram implementadas previamente.

Conjunto da ISA MIPS a ser implementada
ORI rs, rt, imediato
SLL rd, rt, shamt
SLTi rd, rs, imediato
JR label
JAL label

Tabela 1 – Conjunto da ISA MIPS a ser implementada

2 INTRODUÇÃO

MIPS é uma arquitetura computacional RISC (Reduced Instruction Set Computer) com um acrônimo para Microprocessor without Interlocked Pipelines (Microprocessador sem Estágios Interligados de Pipeline) desenvolvida pela MIPS Computer Systems. Essa arquitetura trabalha principalmente com 32 bits, sendo recentemente expandida para 64 bits.

Em uma implementação MIPS Uniciclo as instruções são divididas em uma série de etapas correspondentes às operações das unidades funcionais necessárias, onde todas as instruções serão executadas em um único ciclo de clock. A duração de tal ciclo deverá ser no mínimo o valor da duração da instrução mais lenta. Desta forma, instruções que poderiam ser executadas em períodos mais curtos de clock deverão ser "penalizadas" com o período de clock da instrução mais lenta, para que se respeite o seu conceito de principal.

Uma primeira ideia para resolver este "problema" seria idealizar uma implementação que executasse instruções com clocks de períodos variáveis. Infelizmente, este tipo de projeto é inviável. Portanto, uma alternativa ao clock de período variável se dá com a utilização dos conceitos de uma implementação Multiciclo.

2.1 MIPS Multiciclo

Em uma implementação Multiciclo, cada instrução levará um determinado ciclo de clock, em que a duração mínima de cada clock pode ser calculada a partir da medida do tempo de execução da etapa mais lenta. Diferentemente da implementação uniciclo, na qual os ciclos levam sempre o mesmo número de clock da maior instrução (load word), o multiciclo é muito mais rápido pois não ocorrem "desperdícios" de clock em instruções menores que o load word. Esse compartilhamento reduz a quantidade de hardware necessária quando comparada a outras implementações.

A capacidade de permitir que instruções usem diferentes números de ciclos de clock é a principal vantagem de um projeto Multiciclo. O caminho de dados da uma implementação MIPS Multiciclo foi realizado baseado na implementação do livro Computer Organization and Design, 4ª edição de David Patterson.

3 IMPLEMENTAÇÃO

3.1 Instruções

Com o que havia sido feito até então do projeto, ele já poderia ser executado de maneira a compilar alguns conjuntos de instruções simples. Dentre essas instruções, pode-se descatar:

- Operações da ULA: ADD, SUB, AND, OR, SLT, e consequentemente as instruções BEQ e BNE, visto que elas são realizadas mediante uma operação de subtração;

- Leitura e escrita de dados: LW e SW;

- Jump.

3.1.1 ORI e SLTI

Para realizar as operações ORI e SLTI, foram criadas as microinstruções no arquivo Rom.vhd, dentro da pasta Control:

```
constant ORI : microInstrucao_T := (Alu_or & "01" & SRC_2_Extend & "00000" & "000" & "0000", DISPATCH_2);
```

```
constant SLTI : microInstrucao_T:= (Alu_slt & "01" & SRC_2_Extend & "00000" & "000" & "0000", DISPATCH_2);
```

Foram adicionados, como pode ser observado acima, no campo de tipos ALU Cntr(3), a fim de ajustar os sinais de controle,

```
constant Alu_or : std_logic_vector(2 downto 0) := "101";
```

```
constant Alu_slt : std_logic_vector(2 downto 0):= "110";
```

No arquivo AdressLogic.vhd, dentro da pasta Control, também foram adicionadas em dispastch1:

```
"01011" when "001101",
```

```
"01100" when "001010";
```

E em dispatch2:

```
"00111" when "001000",
```

```
"00111" when "001101";
```

Assim, ambas vão para a microinstrução:

```
constant WriteBack : microInstrucao_T := ("000"&"00"&"000" & Reg_writeAlu & "000"&"0000", FETCH);
```

3.1.2 SLL

Para realizar a operação SLL, foi necessária a criação de um novo hardware, CheckR. Através do campo funct, a máquina seleciona o estado, pois a instrução, apesar de ser do tipo-R, necessita de tratamento diferenciado em relação as instruções lógico-aritméticas.

```
if(Opc="00110") then
        if(funct="000000") then
                AddressOut <= "01110";
        else AddressOut <= OPc;
        end if;
else AddressOut <= OPc;
end if;
```

Foram criadas as micrinstruções no arquivo Rom.vhd, dentro da pasta Control:

```
constant SLL1 : microInstrucao_T := (funcCode & "10" & SRC_2_B & "00000"&"000" & "0000", SEQ);
```

```
constant SLL2 : microInstrucao_T := ("000"&"00"&"000" & Reg_writeAlu & "000"&"0000", FETCH);
```

Foi adicionado também, no campo de tipos SRC2:

```
constant SRC_2_Shamt : std_logic_vector(2 downto 0) := "111";
```

3.1.3 JAL

Para realizar a instrução JAL, foi criada a seguinte microinstrução no arquivo Rom.vhd, dentro da pasta Control:

```
constant JAL : microInstrucao_T := ("000" & "00" & "000" & Reg_writeJAL & "000" & "0000", DISPATCH_2);
```

Foi adicionado também, como pode ser observado acima, no campo de tipos Register, a fim de ajustar o controle,

```
constant Reg_writeJAL : std_logic_vector(4 downto 0) := "11010";
```

Além disso, no arquivo AdressLogic.vhd, dentro da pasta Control, também foi adicionada em dispatch1:

```
"10001" when "000011";
```

E em dispatch2:

```
"01000" when "000011";
```

3.1.4 JR

Para realizar a instrução JR, foi criada a seguinte microinstrução no arquivo Rom.vhd, dentro da pasta Control:

```
constant JR : microInstrucao_T:= (ADD & "01" & SRC_2_B & "00000" & "000" & PC_ALU, FETCH);
```

Assim como a SLL, ela usa CheckR:

```
if(Opc="00110") then
        if(funct="001000") then
                AddressOut <= "10000";
        end if;
        else AddressOut <= Opc;
end if;
```

4 MEMÓRIA

A memória tem apenas 256 palavras de 32 bits, de forma que

apenas 8 bits do PC foram utilizados no seu endereçamento, assim o endereço do dado foi gerado,

```
AdressDado<= '1'& saidaULA_2(8 downto 2);
```

Além disso, o clock foi invertido:

```
clk_inv<= not(clk);
```

Para a instrução Jump, foi feita uma alteração no arquivo Memoria.mif em relação ao arquivo teste.asm, onde foi trocada a instrução **0x08100015** pra **0x08000012**, devido ao formato diferente entre o .mif e o simulador MARS.

5 RESULTADOS

Para testar os resultados da implementação, foi criado um testbench para ser simulado no pacote ModelSim Altera, disponível no Quartus. Todas as instruções funcionaram perfeitamente como esperado.

6 IMPLEMENTAÇÃO NO KIT ALTERA DE2-70

Na segunda etapa o projeto no Quartus deveria ser implementado no Kit Altera DE2-70, disponível no Laboratório de Informática da Universidade de Brasília.

Figura 1 – FPGA Altera DE2-70

A pinagem foi feita usando o molde disponibilizado na plataforma Aprender, juntamente com o guia da placa também disponibilizado na plataforma referida anteriormente.

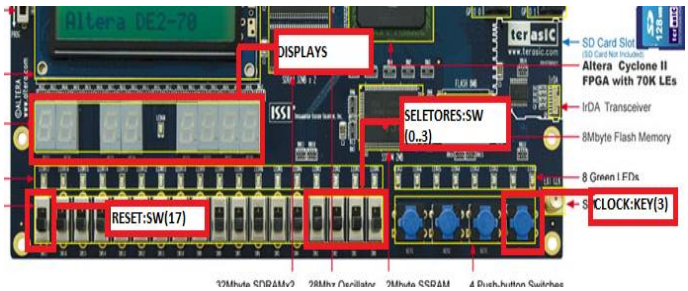

Figura 2 – Pinagem na FPGA

Na tabela abaixo estão indicados quais sinais serão mostrados nos displays baseado na seleção dos bits do seletor.

Seletor	Sinal nos displays
000	Saída da ULA
001	Endereço (PC)
010	Instrução
011	RDM

Tabela 2 – Sinais mostrados nos displays

A implementeção funcionou perfeitamente como esperado e foi demonstrada durante apresentação do projeto para o professor Ricardo Jacobi.

7 CONCLUSÕES

A implementação do processador Multiciclo funcionou conforme o esperado, não somente as instruções que já funcionavam com a montagem mais simples do processador, mas também as que precisaram ser projetadas, como ORI, SLTI, SLL, JAL e JR. Por fim, analisando por um viés mais menos técnico, pode-se dizer que o trabalho foi satisfatório por ter dado aos alunos uma ótima noção e entendimento sobre a arquitetura de diversos componentes internos de um processador, como memória, unidade lógico-aritmética, registradores e controle.

Além disso, as habilidades de projeto de novas codificação em VHDL também melhoraram consideravelmente.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Patterson, D. A. e Hennessy, J. L. (2005). Computer Organization and Design. Fourth Edition;

[2] Jacobi, R.P – Slides de Aula, Universidade de Brasília (2016);

[3] Site Computer Organization Instructive Demonstration: http://u.cs.biu.ac.il/~wiseman/Computer%20Organization.swf ;

[4] Altera DE2-70 User Manual – Terasic Technologies;