Relatório Intermediario Pocket Mips
Laboratório de Hardware

Diogo Oliveira Roman[1], **Rodrigo Gonsalvez de branco**[1]

[1]Faculdade de Computação – Universidade Federal do Mato Grosso do Sul (UFMS)

{diogo.or,rodrigo.g.branco}@gmail.com

Abstract.

Resumo. *Este trabalho consiste na implementação em linguagem de descrição de hardware (VHDL) de um processador baseado em MIPS. Foi implementado os componentes básicos para se fazer operações ariméticas, saltos condicionais ou não e carregamento entre memória e registrador. Esta implementação foi baseada na versão monociclo do MIPS.*

1. Introdução

Este trabalho objetiva implementar em VHDL um processador baseado em MIPS monociclo, além disso para a completa resolução do mesmo seria necessário a prototipação na placa Alteras. Para esta implementação foi necessário relembrar o funcionamento do caminho de dados do processador MIPS monociclo, e algumas de suas particularidades de clock.

Na seção 2 será feita uma explicação detalhada sobre os componentes do Pocket-MIPS e na 3 como as implementações do mesmo foram feitas. Na seção 4 será mostrado o resultado de um programa escrito para ser executado sobre esse processador e por fim na seção 5 a conclusão desse trabalho.

2. Caminho de dados do Pocket-MIPS

O processador Pocket-MIPS foi idealizado a fim de ser implementado em linguagem de descrição de hardware, este processador é monociclo, ou seja, cada instrução é executada em um único ciclo de clock e não há o uso de pipeline. Além disso, assim como o MIPS, as escritas são realizadas apenas na borda de subida do clock.

Além disso o Pocket-MIPS possui 4 registradores de propósito geral e 2 acumuladores. Dessa maneira todas as instruções suportam um único registrador como operando, pois o outro é o acumulador. Cada registrador armazena 1 byte e a memória máxima endereçável é de 256 bytes, e o comprimentos dos endereços são 8 bits.

Os circuitos do caminho de dados são:

- ULA: A unidade lógica aritmética deve realizar soma e subtração. Além do resultado da operação, a alu deverá retornar um outro sinal chamado *zero*, este é 1 caso os dois operandos sejam iguais e 0 caso contrário.
- Unidade de Extensão de Sinal: As operações imediatas, como ADDI, têm em seu operando um número de 5 bits, contudo este caminho de dados é baseado

em operandos de 8 bits (1 byte) e dessa maneira este operando imediato deve ser estendido de 5 bits para 8 bits, isto é feito replicando o bit de sinal até completar estes 8 bits.
- Somadores Adicionais: Um somador (restrito para fazer soma) deve ser usado para fazer o incremento do PC, como o endereçamento é feito em bytes, este incremento deve ser feito de uma unidade, ou seja, PC = PC + 1.
- Gerador de clock: Componente para simular o clock da placa.
- Registrador PC: Um registrador para armazenar o endereço da instrução corrente.
- Conjunto de Registradores: Controla a escrita e leitura dos registradores, para isso é composto por alguns sinais de controle e entradas para os endereços dos registradores e para os dados a serem escritos e 2 saídas para os dados lidos.
- Memória de Instruções: Armazena as instruções a serem executadas, sua leitura é feita a partir do endereço do PC. Assim como um banco de registradores deve ter portas para escrita e leitura dos dados e alguns sinais de controle.
- Memória de Dados: Semelhante à memória de instruções porem armazena apenas os dados.
- Unidade de controle principal: A unidade controle faz a definição de todos os sinais de controle do caminho de dados, ela faz isto a partir *opcode* da instrução.

3. Implementação do Pocket-MIPS em VHDL

Todas as componentes do caminho de dados foram implementadas, esta foi feita utilizando a ferramente GHDL no sistema operacional Linux. Além disso utilizamos svn para controle de versão.

Para o completo entendimento dessa seção é necessário estar com o código ao lado, pois ela é uma descrição da implementação do Pocket-MIPS. Além disso todas as palavras em *italico* representam nomes de variáveis do sistema.

3.1. Gerador de Clock

A arquitetura foi implementada com dois processos:

Um é encarregado de gerar o clock com uma frequencia de 2 ns, este utiliza um sinal auxiliar da arquitetura identificado como *clock*, dessa maneira o clock ainda não é propagado para fora do circuito. Um detalhe desse processo é que ele só fará a modificação no sinal *clock*, caso o sinal *canHalt* for igual a zero, dessa maneira é possível interromper a geração de clock, atribuindo '1' a *canHalt*.

O outro é encarregado por modificar o sinal da arquitetura *canHalt* e o sinal do circuito *Clk*. Além disso sua lista de sensibilidade é composta pelos sinais *clock* e *Halt*, onde o primeiro é interno à arquitetura e outro pertence ao circuito. Caso as condições sejam satisfeitas o sinal de *clock* é atribuito para *Clk* e assim é propagado para fora do circuito, pois *Clk* é o sinal de saída. Além disso, caso o sinal do circuito *Halt* sofrer um evento e mudar para '1' deve-se interromper a geração de clock, como mencionado anteriormente isso pode ser feito atribuindo o valor '1' à variável *canHalt*.

3.2. Registrador PC

Esta arquitetura é composta por um único processo sincronizado pelo clock, sempre que o clock esta na borda de subida o valor do sinal *NewPC* é atribuído ao *CurrentPC*, mesmo quando o valor não muda esta atribuição deve acontecer, pois isso garante o funcionamento dessa entidade como um registrador.

3.3. Somador de 1 Bit

Esta arquitetura é totalmente estrutural, ou seja sem processo, ela simplesmente implementa o circuito somador de 1 bit. Como esse circuito é bem conhecido não entraremos nos detalhes de sua implementação neste trabalho, visto que a forma estrutural traduz exatamente o circuito para a linguagem VHDL, dessa maneira nenhum detalhe de implementação diferente do circuito foi necessário.

3.4. Somador Completo

Esta arquitetura é totalmente estrutural, e consiste simplesmente da ligação entre os somadores de 1 bit. A posições dos vetores de bits *A*, *B* e *Sum* correspondem aos sinais *A*, *B* e *Sum* em cada somador de 1 bit. Ja as posições do também vetor de bits *carry* correspondem ao sinal *carryIn* e *carryOut* de cada somador de 1 bit.

A implementa desse somador foi feita seguindo o padrão ja conhecido na literatura, portando não entraremos em detalhes. A única observação a ser feita é que a ligação entre o *carryOut* e o *carryIn* de somadores de 1 bit respectivos é feita através do vetor de bits *carry* dessa arquitetura. Cada posição desse vetor corresponde ao *carryOut* dos somadores de 1 bit na respectiva posição, ou seja, o somador de 1 bit para os bits da posição 5 dos operandos armazenam o *carryOut* também na posição 5 do vetor *carry*, dessa maneira basta atribuir ao *carryIn* a posição anterior no vetor *carry* para fazer a ligação entre o *carryIn* do somador atual com o *carryOut* do somador anterior.

3.5. Unidade Logico Aritimetica

A implementação dessa arquitetura é simplesmente a instanciação de um somador completo e a manipulação dos seus operandos B e carryIn. Esta manipulação se deu da seguinte maneira:

1. Caso seja soma: Atribui os operandos da ALU para o somador sem nenhuma alteração.
2. Caso seja subtração: Nega bit a bit o operando *B* e atribui um para o *carryIn* da ALU.

3.6. Memória de Dados

Implementamos a memória de dados utilizando um único processo, onde na primeira parte foi calculado o endereço a ser acessado na memória, isso nada mais é que a conversão do sinal *Address* de binário para decimal. Isso foi necessário para acessar o índice do vetor bi-dimensional *Memory*, que justamente é a representação da memória.

A leitura da memória é feita em qualquer momento, independente do clock, já a escrita é feita apenas na borda de subida.

3.7. Memória de Instruções

Esta memória poderia ser exatamente uma copia da memória de dados, contudo para isso deveríamos fazer outra entidade para carregar as instruções do arquivo para ela. Pela simplicidade preferimos fazer o carregamento das instruções dentro dessa entidade mesmo.

Dessa forma foi feito dois processos, um faz apenas a leitura da memória de acordo com o sinal de entrada *ReadAddress* e retorna a instrução pela variável *Instruction*.

Já o outro processo controla o carregamento da memória, ele deve carregar todas as instruções do arquivo especificado na entrada padrão para a memória de instruções antes de começar a execução do processador como um todo. Dessa maneira inserimos dois sinais de controle, para que fosse possível iniciar o processamento assim que a memória de instruções estivesse totalmente carregada, isso foi preciso pois como não sabemos a quantidade de instruções que serão carregada não podemos prever quantos ciclos de clock o este carregamento pode levar. Isso pode causar o incremento do PC antes da hora, pois este esta sincronizado apenas com o clock. Portando enquanto carregamos a memória o gerador de clock está parado.

3.8. Banco de Registradores

Um problema que encontramos ao implementar o banco de registradores foi como definir em qual registrador acumulador os dados deveram ser escritos ou lidos. Para soluciona-lo inserimos dois sinais de controle identificados como *WriteHidden* e *WhichHidden*, onde o primeiro indica se deverá ser feita uma escrita em algum acumulador e o segundo em qual acumulador essa escrita será feita, caso seja 1 o acumulador BC deve ser carregado, caso contrário o AC que deverá receber o valor. Além dessas variáveis de controle inserimos um vetor de bits para retornar qual o valor lido nos acumuladores, a esse vetor demos a identificação de *ReadHidden*.

3.9. Extensor de sinal

Esta entidade é bem simples, de forma que foi trivial faze-la. Fizemos com um processo que apenas copia os dados de um vetor de 5 bits para outro de 8 bits, os 5 bits de mais baixa ordem do vetor menor vão para as cinco posições iguais no vetor maior, os bits que sobram são preenchidos com o bit de sinal, ou seja, da posição 5 até à 7 recebe a posição 4 do vetor menor.

3.10. Unidade de Controle

Apesar do código extenso dessa entidade, ela não oferece muitas dificuldades. Nela fazemos controle de todo o processador. Antes de qualquer ação uma verificação na condição da memoria de instruções é feita, ou seja, verifica se esta já esta totalmente carregada, e caso afirmativo o processamento da unidade de controle continua e faz uma séride de atribuições aos diversos sinais de controle. Estas atribuições dependem apenas dos sinais de entrada *aluOperation* e *Func*, onde o primeiro é o código da operação que se deve executar e o segundo é usado apenas nas operações MTA, MFA, MTB, MFB.

Fazemos o controle da memória de instruções através de um sinal de inicio de clock, ou seja, enquanto não for recebido o sinal de que a memória de instruções esta carregada a geração de clock não inicia.

3.11. Caminho de Dados (System)

Esta entidade é o caminho de dados, nela fazemos a ligação entre todas as outras entidades citadas e ainda entre circuitos primitivo como portas and e multiplexadores. Foi necessário a instanciação de vários sinais para fazer a ligação entre as entidades. Além disso incluímos outro processo que intercepta o sinal retornado pela memória de instruções e retorna na saída padrão a informação das instruções lida em ascii.

4. Experimentos e Resultados

Programa teste:

1. 00100110
2. 11100000
3. 00100100
4. 00001000
5. 01001000
6. 11100011

Este programa faz o seguinte:

1. AC = AC + 6
2. r1 = AC + 0
3. AC = AC + 4
4. AC = AC + r1
5. AC = AC - r1
6. Interrompe o processo

No fim o registrador AC deve armazenar o valor 10. Isto pode ser visto na imagem de ondas abaixo.

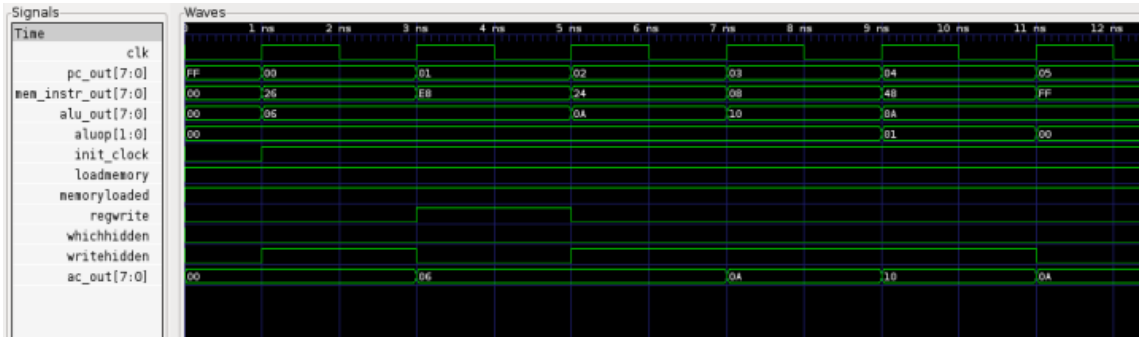

Figure 1. Ondas geradas na execução do programa

No primeiro ciclo de clock a memória de instruções é carregada, porem isso foi feito muito rápido de maneira que foi imperceptível, nesse mesmo ciclo o PC é iniciado em 0 e a unidade de controle faz a leitura da primeira instrução, e ajusta os sinais de controle, é possível ver isso através da variável *writehidden*. Na borda de subida do segundo ciclo de clock o valor 6 é atribuído ao acumulador AC, o PC é incrementado e dessa maneira a instrução MFA é carregada, note que o sinal *writehidden* volta ao valor 0, isto indica que a instrução recem carregada não escreverá nos acumuladores. O processamento continúa de maneira analoga até o fim do programa. A variável *ac_out* armazena o valor do acumulador AC, dessa maneira pode-se visualizar a corretude do processador nesse teste.

5. Conclusão

As principais dificuldades encontradas foram questões praticas de simulação. Em alguns momentos perdemos muito tempo para descobrir alguns erros pois estávamos pensando

no sistema como qualquer outro programa que estamos acostumado a fazer, principalmente no fato de termos que reatribuir valores a variáveis mesmo quando esses valore não mudam, algo desnecessário na programação comum.

Este trabalho nos ajudou a identificar alguns problemas inerentes dos projetos de hardware, além disso foi possível entender e superar as dificuldade comuns ao se programar em linguagem de descrição de hardware.