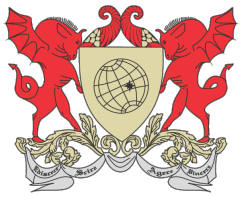
Universidade Federal de Viçosa
Campus Florestal
Ciência da Computação



Documentação MIPS Com Pipeline

1ª Parte do 1º Trabalho de Organização de Computadores 2

Grupo Ruan Evangelista Formigoni (02661) e Vitor Guerra Veloso (02658)
Professor José Augusto Miranda Nacif

Belo Horizonte, 2 de setembro de 2017

Sumário

Resumo

Nesse trabalho iremos desenvolver o famigerado MIPS Simplificado com Pipeline, o qual terá suporte a hazard de dados, instruções do tipo R e instruções LW/SW. Para isso utilizaremos a linguagem de descrição de hardware Verilog e sintetizaremos/simularemos sua execução com Icarus Verilog. Por fim os resultados serão exibidos em formas de ondas utilizando a ferramenta GtkWave.

Abstract

At this work we'll develop the Simplified MIPS Processor with Pipeline. Our MIPS will support data hazards, type R instructions and LW/Sw instructions. For achieving this goal we'll use the hardware description language Verilog and we'll synthesize/simulate the execution with Icarus Verilog. Finally the results will be shown in wave forms at the GtkWave tool.

1 Introdução

O trabalho consiste na implementação do Pipeline em um MIPS Simplificado, podendo ser este uma implementação previamente desenvolvida por um dos integrantes do grupo. A implementação ocorrerá completamente via código na linguagem de descrição de hardware Verilog e deverá estar acompanhada de seu devido *testbench*[1]. A entrega foi dividida em 2 etapas, a primeira consiste em: documentação parcial e código em Verilog simulável no Icarus Verilog; a segunda consiste em: documentação total e código sintetizável para uma FPGA real.

Utilizamos - como implementação base - o código desenvolvido pelo integrante Ruan Formigoni na disciplina Organização de Computadores 1. Esse código - que consiste no MIPS Simplificado sem pipeline - será adaptado para receber os módulos de pipeline e tratamento de hazard. Para nos organizarmos melhor utilizamos as ferramentas Git e Overleaf para, respectivamente, Gerenciar versões e redigir a documentação de forma descentralizada.

[1] Módulo de teste desenvolvido na mesma linguagem

2 Revisando o Design do Projeto anterior

Como o trabalho que nos baseamos era uma implementação de MIPS sem pipeline funcional, todas modificações que fizemos foram aquelas relacionadas a inclusão de um pipeline e o tratamento de hazards. Grande parte dos módulos individuais auxiliares se mantiveram intactos desde a outra implementação de MIPS. Mudanças foram necessárias no PC, Control, banco de registradores... Todas essas mudanças tinham relação a implementação do Tratamento de Hazards.

Além disso foram incluídos módulos como os Registradores de Pipeline e as unidades de tratamento de hazards. Essas adições serão discutidas melhor nas próximas seções.

3 Desenvolvendo o MIPS com Pipeline

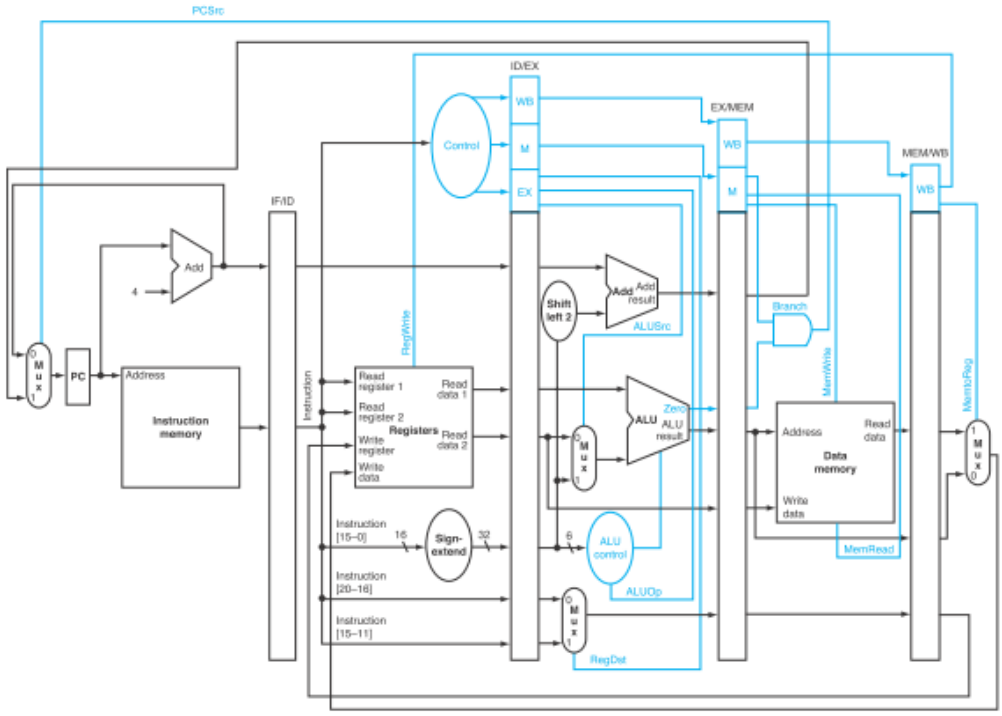


Figura 1: Caminho de Dados do MIPS com Pipelines

3.1 Decisões de Projeto

Optamos por separar o Pipeline em 2 partes: a primeira consiste nos registradores de pipeline e a segunda nos módulos de tratamento de hazards; cada integrante ficou com uma dessas partes. Ao término da elaboração inicial dos novos módulos, foi decidido que um integrante trataria a integração das partes e o outro adiantaria a documentação.

3.2 Uma Visão Ampla da Implementação

Um MIPS com Pipeline difere do sem pipeline por ter múltiplas instruções sendo executadas ao mesmo tempo e isso exige não apenas inclusão de novos módulos, mas também adaptação de alguns que já existiam: como os sinais de controle, que passam a ser individuais para cada instrução em andamento. Exigindo que sejam incluídos registradores que armazenem o sinais de controle para cada instrução nos diferentes estágios do pipeline ao qual ela vai passar.

3.3 Uma Visão Detalhada da Implementação

3.3.1 Unidade de Tratamento de Hazards

Deve detectar lw para ocasionar os stalls quando necessário. Para isso ele se atenta a flag LeMem do estágio EX.

- Se a flag LeMem anunciar algum load, então a unidade de detecção de hazard confere os registradores Rs e Rt da instrução do estágio ID, comparando-as com o registrador Rt da instrução de load. Se forem iguais é STALL.

Como funciona o stall? Basicamente ele força no próximo ciclo do relógio que todos sinais de controle em ID/EX tenham valor 0 e que os registradore PC e IF/ID tenham o mesmo valor do último ciclo. Atrasando os ciclos

Entradas:

1. ID/EX.LeMem
2. ID/EX.registradorRt
3. IF/ID.registradorRs

4. IF/ID.registradorRt

Saidas:

1. PCWrite
2. IF/ID.Write
3. Mux.ID/EX.Write

3.3.2 Unidade de Forwarding

Deve detectar se uma instrução irá escrever em registrador que será lido no próximo ciclo. Para isso ele se atenta as flags EscreveReg dos sinais de controle nos pipelines MEM/WB e EX/MEM.

- Se a flag EscreveReg está ativa para o registrador EX/MEM, compara-se o EX/MEM.registradorRd com 0 e com os registradores de entrada Rs da próxima instrução (ID/EX). Se EX/MEM.registradorRd não for 0:
  - e tem valor similar a ID/EX.registradorRs... Forward! (ForwardA=10)
  - e tem valor similar a ID/EX.registradorRt... Forward! (ForwardB=10)
- Se a flag EscreveReg está ativa para o registrador MEM/WB, compara-se o MEM/WB.registradorRd com 0 e com os registradores de entrada Rs da instrução 2 ciclos seguintes (ID/EX). Se MEM/WB.registradorRd não for 0:
  - confere se não há Forward em MEM antes.[2] Caso não tenha Forward em Mem:
    - e tem valor similar a ID/EX.registradorRs... Forward! (ForwardA=01)
    - e tem valor similar a ID/EX.registradorRt... Forward! (ForwardB=01)
- Caso não haja Forward nenhum então (ForwardB=00)(ForwardA=00)

[2]Conferindo se EX/MEM.EscreveReg está inativo ou se EX/MEM.registradorRd é 0. Indicando que talve haja Forward em Mem. Então confere se o Registrador Rd de escrita em EX/MEM se assemelha com algum Registrador de entrada de ID/EX, Rs ou Rt.

Entradas

1. ID/EX.registradorRs
2. ID/EX.registradorRt
3. EX/MEM.registradorRd
4. MEM/WB.registradorRd
5. EX/MEM.EscreveReg
6. MEM/WB.EscreveReg

Saidas

1. ForwardB
2. ForwardA

3.3.3 Registradores de Pipeline

Os registradores de pipeline possuem uma grande quantidade de entradas e saídas, mas possuem pouca lógica. Sua função é basicamente receber uma entrada em um clock e passá-la adiante no próximo ciclo de clock.

```
1     //EX REGS
2    input wire PIPEIN_EX_ALUSrc,
3    input wire[1:0] PIPEIN_EX_ALUOp,
4    input wire PIPEIN_EX_RegDst,
5    output reg PIPEOUT_EX_ALUSrc,
6    output reg[1:0] PIPEOUT_EX_ALUOp,
7    output reg PIPEOUT_EX_RegDst,
8    //MEM REGS
9    input wire PIPEIN_MEM_Branch,
10   input wire PIPEIN_MEM_MRead,
11   input wire PIPEIN_MEM_MWrite,
12   output reg PIPEOUT_MEM_Branch,
13   output reg PIPEOUT_MEM_MRead,
14   output reg PIPEOUT_MEM_MWrite,
15   //WB REGS
16   input wire PIPEIN_WB_RegWrite, input wire PIPEIN_WB_MemtoReg↩
       ,
```

```
17      output reg PIPEOUT_WB_RegWrite, output reg ↩
            PIPEOUT_WB_MemtoReg,
18
19      //INPUT ON ID FASE
20      input wire[31:0] PIPEIN_PCPlus4, input wire[31:0] ↩
            PIPEIN_ReadData1,
21      input wire[31:0] PIPEIN_ReadData2, input wire[31:0] ↩
            PIPEIN_SignExt,
22      input wire[4:0] PIPEIN_RS, input wire[4:0] PIPEIN_RT, input ↩
            wire[4:0] PIPEIN_RD,
23      //OUTPUT ON EX FASE
24      output reg[31:0] PIPEOUT_PCPlus4, output reg[31:0] ↩
            PIPEOUT_ReadData1,
25      output reg[31:0] PIPEOUT_ReadData2, output reg[31:0] ↩
            PIPEOUT_SignExt,
26      output reg[4:0] PIPEOUT_RS, output reg[4:0] PIPEOUT_RT, ↩
            output reg[4:0] PIPEOUT_RD
```

4 Testando

4.1 Unidade de Forwarding

As diferentes formas de redirecionar os dados na pipeline são implementadas com base no que foi mostrado em 3.3.2, recapitulando, os principais casos apontados em que o redirecionamento de dados é necessário são:

1. EX/MEM.RegisterRd = ID/EX.RegisterRs, RegWrite está ativo e o registrador é diferente de $0.

2. EX/MEM.RegisterRd = ID/EX.RegisterRt, RegWrite está ativo e o registrador é diferente de $0.

3. MEM/WB.RegisterRd = ID/EX.RegisterRs, RegWrite está ativo e o registrador é diferente de $0.

4. MEM/WB.RegisterRd = ID/EX.RegisterRt, RegWrite está ativo e o registrador é diferente de $0.

Serão testados os tratamentos para os hazards de dados utilizando a seguinte sequencia de instruções:

```
1         add $s0 $t0 $t1 // 8 + 4 = 12
2         sub $s1 $s0 $t2 // 12 - 10 = 2
3         add $s2 $s0 $s1 // 12 + 2 = 14
4         add $s3 $s2 $s1 // 14 + 2 = 16
```

Os registradores possuem os valores iniciais mostrados abaixo:

```
1       registers[8] = 8;       //$t0
2       registers[9] = 4;       //$t1
3       registers[10] = 10;     //$t2
```

Na linha 2 há um hazard do tipo 1, o valor de $s0 vem do estágio *MEM* e não da pipeline ID/EX.
Sinais Esperados: ForwardA = 10 — ForwardB = 00.

Na linha 3 há dois hazards dos tipos 2 e 3, o valor de $s0 vem do estágio *WB* e não da pipeline ID/EX (tipo 3). E o valor de $s1 vem do estágio *MEM* (tipo 2).
Sinais Esperados: ForwardA = 01 — ForwardB = 10.

Na linha 4 há dois harzards dos tipos 1 e 4, o valor de $s2 vem do estágio *MEM* e não da pipeline ID/EX (tipo 1). E o valor de $s1 vem do estágio *WB* (tipo 4).
Sinais Esperados: ForwardA = 10 — ForwardB = 01.

A seguir serão mostradas as formas de onda geradas pelo software GTKWave, e referentes ao comportamento do processador ao executar as instruções acima.

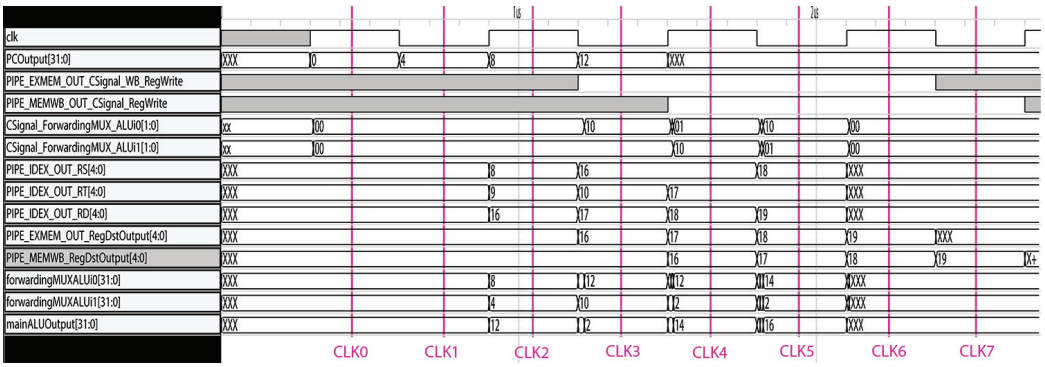


Figura 2: Formas de onda referentes às instruções listadas em 4.1.

- O primeiro estágio de execução acontece em *CLK2*, não há dependências pois esta é a primeira instrução executada.
- O segundo estágio de execução acontece em *CLK3*, o valor da entrada i0 da ALU vem do estágio MEM.
- O terceiro estágio de execução acontece em *CLK4*, o valor da entrada i0 da ALU vem do estágio WB e o valor da entrada i1 da ALU vem do estágio MEM.
- O quarto estágio de execução acontece em *CLK5*, o valor da entrada i0 da ALU vem do estágio MEM e o valor da entrada i1 da ALU vem do estágio WB.

4.2 Unidade de Tratamento de Hazads

No caso em que um registrador tenha seu valor carregado da memória e uma instrução subsequente necessite utilizá-lo, é necessário parar o pipeline até que o valor tenha sido carregado da memória como foi mostrado em 3.3.1, recapitulando, a forma de fazer isto é atribuindo valores nulos *nops* a todos os sinais de controle, o que é definido a partir da seguinte condição:

Se no estágio de execução um o sinal de leitura de memória está ativado e o registrador a ter seu valor atualizado é o mesmo utilizado pela instrução subsequente, deixe-a parada por um ciclo no estágio de decodificação até que seja acessada a memória e o valor esteja disponível para ser utilizado no

estágio de execução com o auxílio da unidade de forwarding.

Serão testados os tratamentos para os hazards de dados utilizando a seguinte sequencia de instruções:

```
1          sw $t3 4($0)     //Guarda 132 na memoria
2          lw $s0 4($0)     //Carrega 132 da memoria
3          add $s1 $s0 $t1     // 132 + 4 = 136
```

Os registradores possuem os valores iniciais mostrados abaixo:

```
1       registers[9] = 4;          //$t1
2       registers[10] = 10;        //$t2
3       registers[11] = 132;       //$t3
```

Na linha 1 o valor 132 é armazenado na memória. Na linha 2 o valor 132 é carregado da memória. Na linha 3 o valor carregado da memória é somado com 4

A seguir serão mostradas as formas de onda geradas pelo software GTKWave, e referentes ao comportamento do processador ao executar as instruções acima.

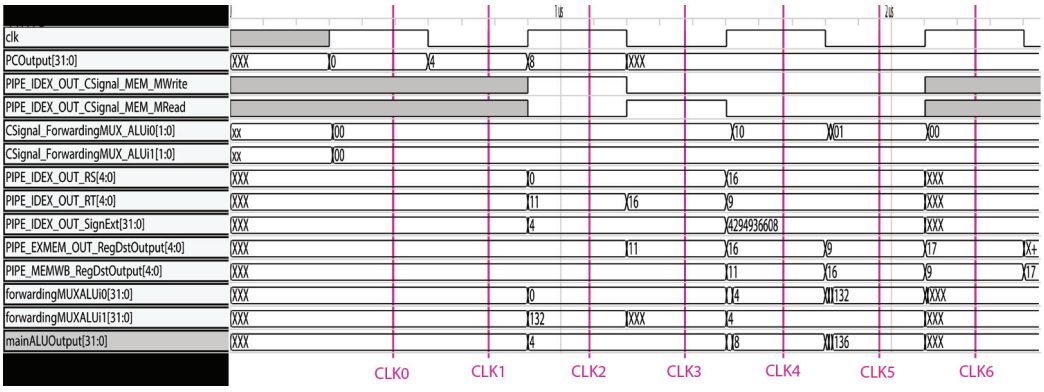


Figura 3: Formas de onda referentes às instruções listadas em 4.2.

- O primeiro estágio de execução acontece em *CLK2* onde o endereço de memória é calculado para armazenar o valor 132 na memória, ambos este endereço e o valor a ser armazenado são transferidos para o estágio MEM.

- O segundo estágio de execução acontece em *CLK3* onde o endereço de memória do valor a ser transferido para o registrador $s0 é calculado. Como será necessária uma leitura de memória, o sinal *MemRead* é ativado. No módulo de controle de hazards a condição em que o registrador a ser lido é o mesmo utilizado por uma instrução subsequente retorna um valor verdadeiro.

- O terceiro estágio de execução acontece em *CLK4* onde uma instrução *nop* é executada, a instrução anterior continua no estágio de decodificação.

- O quarto estágio de execução acontece em *CLK5*, onde a terceira instrução é executada. O valor da primeira entrada da ALU é buscado do estágio WB pela *forwarding unit* e a operação de adição é executada.

5 Conclusão

Com esse trabalho pudemos desenvolver e assim compreender na prática como é o funcionamento de um paralelismo em nível de instrução executado em um processador MIPS com Pipeline. A notável vantagem em termos de desempenho computacional é evidenciada, principalmente, quando testamos a simulação do processador com um grande número de instruções, isso ocorre devido ao fato de que cada clock agora é responsável pelo avanço de 3 instruções e a conclusão de uma. Desta forma em um ciclo de clock menos dispositivos são acionados por instrução sendo executada, o que reduz o período necessário para os ciclos de clock em uma taxa considerável.

Apesar de implementarmos um processador já existente e muito bem documentado pelo livro Organização de Computadores, de Patterson, aprendemos muito com esse trabalho e pudemos ver como se pode aumentar o desempenho de um processador sem ter que alterar a frequência de clock ou aumentar o número de núcleos - o que levaria ao superaquecimento e ao aumento considerável no custo de produção, respectivamente.