Universidade Federal do Paraná

João Manoel Pampanini Filho

Implementação do CP1 do MIPS
Unidade de ponto flutuante com Tomasulo

Curitiba, PR
2017

João Manoel Pampanini Filho

Implementação do CP1 do MIPS
Unidade de ponto flutuante com Tomasulo

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Prof. Dr. Roberto André Hexsel.

Curitiba, PR
2017

Resumo

O uso de números reais é cada vez mais frequente nas aplicações, portanto é necessário que processadores atendam a esta demanda, fornecendo unidades para cálculo com ponto flutuante seja em *half*, *single* ou *double precision.* A utilização de ponto flutuante torna-se um problema quando inserida em processadores segmentados, pois as unidades funcionais dedicadas para este fim demoram normalmente mais que um ciclo do relógio. O Algoritmo de Tomasulo se mostra uma solução eficaz para este impasse, possibilitando um bom escalonamento de instruções e gerenciamento de diversas unidades funcionais distintas, com latências diferentes, podendo ser utilizadas em uma unidade à parte, como um coprocessador ou um periférico, ou implementado como um processador como um todo. Este trabalho apresenta uma implementação do Algoritmo de Tomasulo com um coprocessador, usado para as instruções de ponto flutuante, e propondo um conjunto de instruções para sua utilização. São apresentados testes estruturais que mostram funcionamento correto das partes.

Palavras-chave: MIPS, Tomasulo, Ponto Flutuante.

Abstract

The use of real numbers is becoming more frequent in applications, so it is necessary for processors to meet this demand, providing units for calculating with floating point in half, single or double precision. Floating-point utilization becomes a problem when inserted into segmented processors because the dedicated units of work for this purpose usually take longer than one clock cycle. The Tomasulo Algorithm is an effective solution to this impasse, allowing a good scheduling of instructions and management of several different functional units, with different latencies, and can be used in a separate unit, as a coprocessor or peripheral, or implemented as a processor as a whole. This work presents an implementation of the Tomasulo Algorithm with a coprocessor, used for the floating point instructions, and proposes a set of instructions for its use. Structural tests are presented that show correct functioning of the parts.

Keywords: MIPS, Tomasulo, Floating Point.

Sumário

1 Introdução **6**

2 Componentes **9**
2.1 Execução em ordem e fora de ordem 9
2.2 Algoritmo de Tomasulo 11
2.3 O padrão IEEE-754 12

3 Implementação **14**
3.1 Interface com CPU 14
3.2 Estágio de decodificação 15
3.3 Estágio de despacho 18
3.4 Estágio de execução 18
3.4.1 Multiplicador 19
3.4.2 Somador 19
3.4.3 Memória 19
3.5 Estágio de *write-back* 20

4 Testes **22**

5 Conclusões e Trabalhos futuros **32**

Referências Bibliográficas **33**

Capítulo 1

Introdução

Muitos programas utilizam operações com uma representação para números reais visando maior precisão em seus dados e resultados, como, por exemplo, para apresentar uma estatística ou para cálculos complexos de programas científicos, sendo necessário que processadores ofereçam suporte a operações com números em ponto flutuante e ponto fixo. Operações nas representações com ponto flutuante, definidos pelo padrão IEEE-754 [IEE08], são tratados de forma diferente daquelas com inteiros, e portanto precisam de *hardware* dedicado.

Ao tentarmos implementar uma unidade de ponto flutuante em um processador segmentado, incorremos em problema de escalonamento porque distintas operações em ponto flutuante demoram tempos distintos e, possivelmente, maiores que um ciclo do clock, e, por isso, interferem na execução das demais instruções. Portanto, pode ser necessária a utilização de uma unidade funcional separada do processador de inteiros para atender a essa demanda.

Este trabalho descreve a implementação do co-processador 1 (CP1) do MIPS de 32 bits [MIP05a, MIP05b, MIP05c], responsável por operações de ponto flutuante, por isso também referenciado como *unidade de ponto flutuante*, utilizando unidades funcionais de soma e muitiplicação padrão IEEE-754 e o algoritmo de Tomasulo (AoT) definido em [Tom67].

Para simplificar a implementação e os testes, a *unidade de ponto flutuante* é implementada como um periférico, e emprega um conjunto de instruções próprio e *Direct Memory Access* (DMA) para acesso à memória. A implementação do algoritmo de Tomasulo é baseada em Hennessy e Patterson [HP11], que apresenta uma unidade de re-ordenação das instruções e uma de renomeação de registradores. Os motivos para a escolha do algoritmo de Tomasulo são o paralelismo em nível de instrução, a re-ordenação dinâmica das instruções e o aumento do número de registradores disponíveis. A re-ordenação dinâmica das instruções permite a exploração do paralelismo em nível de instrução, quando algumas instruções podem ser executadas em uma ordem distinta daquela do programa.

A Figura 1.1 mostra um diagrama de blocos do co-processador 1. A unidade de ponto flutuante consiste das unidades funcionais de soma e produto, de uma unidade de acesso à memória, de um conjunto de registradores para ponto flutuante, um conjunto de registradores para endereços e duas estações de reserva para cada unidade funcional. Um *buffer* de reordenação é usado para permitir que as instruções executem fora de ordem.

Quando as instruções chegam ao co-processador, são direcionadas para uma estação de reserva, junto com tudo que é necessário para a operação ser realizada. Caso algum operando não esteja disponível, a operação fica à espera até esse seja apresentado

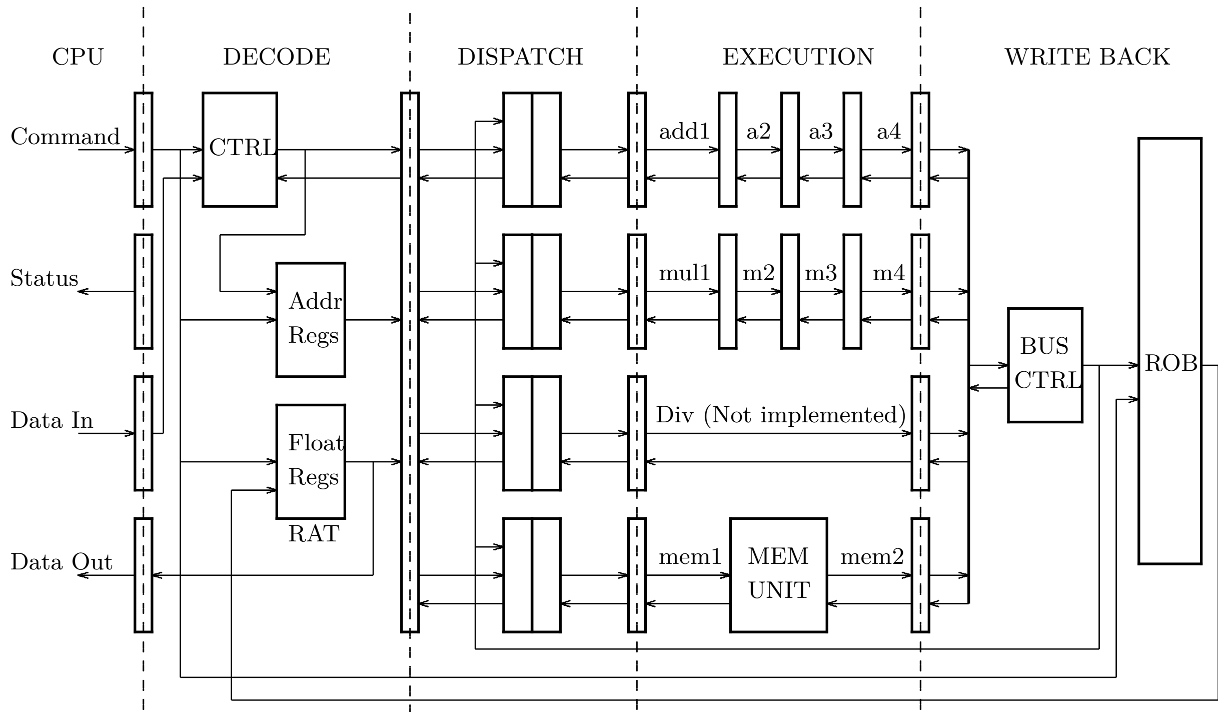

Figura 1.1: Diagrama de blocos do coprocessador 1.

no barramento. Assim que uma instrução dispõe dos recursos para sua execução, ela é executada, sem esperar por outras que chegaram antes na *unidade de ponto flutuante*, mas que ainda estão impossibilitadas de executar por falta de operandos. A ordem de execução das instruções é alterada e isso pode diminuir o tempo total de execução de um programa.

Após uma instrução completar, seu resultado é colocado no *buffer* de reordenação (*re-order buffer*, ROB), que garante que as instruções sejam confirmadas na ordem do programa. O resultado é enviado também para as estações de reserva que estão esperando pelo resultado.

Definimos renomeação como o ato de trocar o nome do registrador de destino de uma operação para um identificador de resultado, e trocando tambem todos os as dependencias neste registrador de destino para o identificador. A renomeação provê registradores adicionais. Quando uma instrução chega para a *unidade de ponto flutuante*, seu registrador de destino é trocado para uma indicação de onde a unidade funcional vai escrever seu resultado, usando para isso um índice do ROB. Quando uma instrução chega e tem dependência de dados em outras instruções, em vez de ficar esperando para que o valor seja alterado no registrador de destino, ela espera pelo valor que será escrito no ROB, o que possibilita a reutilização dos registradores e reduz o tempo de execução.

Foram utilizados as unidades de soma e multiplicação descritos em [TDD+15]. As unidades aritméticas são compostas por 3 estágios, portanto suportam até 3 instruções de soma e 3 de multiplicação executando ao mesmo tempo. A simulação de acesso a memória ocorre através da leitura e escrita em um arquivo, e estas operações demoram 1 ciclo. A unidade de divisão em ponto flutuante não foi implementada, sendo utilizado uma unidade "fake" em seu lugar, sendo que o quociente e o resto são sempre zero.

A unidade de ponto flutuante foi implementada como um periférico e para acessá-la foi definido um conjunto de instruções, seguindo um padrão similar às instruções do MIPS para um fácil entendimento de sua utilização.

O texto está organizado da seguinte maneira. No capítulo 2 são apresentados os conceitos envolvidos no projeto. No capítulo 3 é descrita a implementação do coprocessador. No capítulo 4 são apresentados testes e amostras dos diagramas de tempo da execução de instruções. No capítulo 5 são apresentadas as conclusões e os trabalhos futuros.

Capítulo 2

Componentes

Este capítulo apresenta os conceitos utilizados neste trabalho. A seção 2.1 explica a execução em ordem e fora de ordem. A seção 2.2 explica o algoritmo de Tomasulo. A seção 2.3 descreve a representação em ponto flutuante e explica a sua utilização.

2.1 Execução em ordem e fora de ordem

Execução em ordem é a execução na ordem do código gerado pelo compilador. O hardware projetado para executar "em ordem", como um pipeline por exemplo, faz um escalonamento estático das instruções na ordem em que o compilador gerou as instruções. As instruções são execudas e confirmadas uma após a outra, na ordem do código. A implementação desse tipo de escalonador é simples, o que facilita o projeto do hardware. Outra vantagem é a facilidade de depurar, pois a execução é "bem organizada".

Ocorre uma *dependência de dados* entre instruções i e j quando a instrução j usa o resultado de i como um de seus operandos. A instrução j não pode executar até que o resultado de i seja confirmado. Diz-se que existe um *risco* (*hazard*) se é possível que j execute antes de i confirmar seu resultado, possivelmente gerando um resultado errado

Definimos *ciclo por instrução* (CPI) como a razão entre o número de ciclos usados para executar um programa e a quantidade de instruções deste programa. Quanto menor o CPI, maior o desempenho do processador.

Bolhas (*stalls*) são as instruções inseridas para acomodar a execução das instruções na presença de riscos causados por dependências. As bolhas tem como função distanciar as instruções conflitantes, para que o risco desapareça, ao custo de um aumento do CPI.

Quando ocorre dependência entre instruções, todas as demais devem esperar o risco ser resolvido, caso não haja circuito adicional para resolvê-lo. Isso faz as instruções demorarem mais que o desejado e o CPI aumentar.

A Figura 2.1 mostra um trecho de código na sequência gerada pelo compilador. As setas indicam as dependências entre as instruções. Os *NOPs* representam as bolhas necessárias para resolver as dependências.

Na execução *fora de ordem* as instruções são buscadas na ordem do programa, mas sua execução pode ou não ser nesta ordem. Em processadores *fora de ordem* mais antigos, as instruções eram confirmadas fora de ordem também, enquanto que, nos mais atuais, a confirmação é feita em ordem. O *hardware* faz o escalonamento dinâmico das instruções, o que pode adiantar instruções para execução. Quando temos instruções paradas, por risco ou outro motivo, uma instrução que acabou de chegar pode passar para o estágio de execução antes. Esse comportamento possibilita uma melhora de desempenho, e esconde

```
# ordem do programa
MUL r2, r0, r1
MUL r5, r3, r4
ADD r9, r2, r5
MUL r8, r6, r7
ADD r0, r8, r1
# execução em ordem
MUL r2, r0, r1
MUL r5, r3, r4
NOP
NOP
ADD r9, r2, r5
MUL r8, r6, r7
NOP
NOP
ADD r0, r8, r1
```

Figura 2.1: Exemplo de execução *em ordem*.

parte da latência que riscos geram, fazendo com que o processador fique menos tempo ocioso, proporcionando uma performace melhor e uma maior utilização das unidades funcionais.

A Figura 2.2 mostra um trecho de código que pode ser executado fora de ordem. Supondo que cada instrução executada em 3 ciclos, o código da Figura 2.2 é executada em 6 ciclos, ao invés dos 9 da Figura 2.1

```
# ordem do programa
MUL r2, r0, r1
MUL r5, r3, r4
ADD r9, r2, r5
MUL r8, r6, r7
ADD r0, r8, r1
# execução fora de ordem
MUL r2, r0, r1
MUL r5, r3, r4
MUL r8, r6, r7
ADD r9, r2, r5
NOP
ADD r0, r8, r1
```

Figura 2.2: Exemplo de troca de ordem de execução.

2.2 Algoritmo de Tomasulo

O algoritmo de Tomasulo foi publicado no *Journal* da IBM em janeiro de 1967, [Tom67], e descreve uma forma de otimização de performance das unidades aritméticas, em particular as de ponto flutunte. Sua primeira implementação foi na unidade de ponto flutuante do *IBM System/360 Model 91.* O algoritmo de Tomasulo pode ser usado para qualquer sistema em que são utilizadas várias unidades funcionais simultaneamente, proporcionando escalonamento dinâmico e execução fora de ordem das instruções, além do aumento do número de registradores disponíveis.

O algoritmo de Tomasulo faz a renomeação dos registradores de destino, que recebem os valores que são produzidos nas unidades funcionais. As instruções chegam ao processador e são direcionadas para as estações de reserva das unidades funcionais que forem utilizar. Cada uma dessas instruções recebe um *tag* como nome de seu resultado. Deste modo o registrador de destino é renomeado para essa *tag* e o nome lógico (de programa) do registrador é esquecido, deixando o mesmo livre para utilização por outras instruções. Todas as instruções que chegam ao processador, e que dependem de um resultado, passam a depender da sua *tag* ao invés de depender do nome lógico do registrador.

```
#  sem renomeação
       lw   r2, 0(r4)
       add  r2, r2, r1
       mul  r1, r2, r1

#  com renomeação
       lw   t0, 0(r4)
       add  t1, t0, r1
       mul  t2, t1, r1
```

Programa 2.1: Exemplo de código com e sem renomeação.

Quando as instruções chegam ao processador, elas são direcionadas às estações de reserva de suas respectivas unidades funcionais, sendo feita a renomeação de seus registradores neste momento. Se as estações estiverem cheias então o processador deve esperar. Uma vez na estação de reserva, as instruções esperam que a unidade funcional fique livre para uso e que seus operandos estejam disponíveis. Com as duas condições satisfeitas, é feito o despacho para a execução. Esse despacho pode não ocorrer na ordem de chegada das instruções no processador, uma vez que não necessariamente uma instrução que entrou antes vai estar pronta para execução pois, por exemplo, a mesma pode depender de um acesso à memória e ficar à espera da carga de uma variável.

```
#  ordem de entrada
     lw   r1, 0(r0)
     add  r2, r1, r3
     add  r4, r1, r2
     mul  r5, r3, r6

#  ordem de execução
     lw   r1, 0(r0)
     mul  r5, r3, r6
     add  r2, r1, r3
     add  r4, r1, r2
```

Programa 2.2: Exemplo de código com escalonamento dinâmico.

Ao completar uma operação, a unidade funcional pede o uso do barramento comum (CDB). Se o barramento não estiver livre, a unidade deve esperar até que esteja. No momento que a unidade funcional consegue o barramento, ela escreve seu resultado em todas estações de reserva que estiverem esperando pela *tag* do valor, bem como no registrador de destino que a *tag* substituiu, caso o registrador lógico não tenha sido reutilizado e contenha um valor válido, ou esteja esperando por um valor de outra *tag*.

2.3 O padrão IEEE-754

O padrão IEEE-754 [IEE08] é um padrão técnico para a computação com ponto flutuante que foi estabelecido em 1985 pelo *Institute of Electrical and Electronics Engineers* (IEEE), sendo sua versão mais atual a de 2008. Atualmente o padrão define formatos binários para 32 bits, 64 bits e 128 bits.

O formato *float* de 32 bits é definido com 1 bit de sinal, 8 bits de expoente e 23 bits de fração, representado na Figura 2.3 e na Equação 2.1 respectivamente por S, EXP e FRAC.

$$-1^S * 1.FRAC * 2^{EXP-127} \tag{2.1}$$

Figura 2.3: Organização do formato *float*.

No padrão estão definidos alguns tipos de valores especiais que um *float* pode assumir. Dentre eles estão os valores *finitos*, *zero*, *denormals*, *NaN* (*not a number*) e ± *infinito*. O valor *finito* é o representado na Equação 2.1, mas o valor de EXP deve estar entre "00000001" e "11111110" inclusive, e FRAC é adicionado a um bit 1 implicito. O valor *zero* é representado quando o EXP é cheio de 0's e a FRAC também. O valor *infinito* tem seu sinal determinado por S, sua representação é dada por EXP cheio de 1's e FRAC cheio de 0's. O valor *NaN* é representando por EXP cheio de 1's e FRAC com ao menos um bit em 1. O valor *denormal* representa números bem próximos de *zero*, mas que não pertencem à representação dos *finitos*. Este formato é representado como na Equação 2.2,

onde temos um 0 implicito ao invés do 1, o valor em EXP é cheio de 0's, mas seu valor para cálculos efetivamente é "00000001".

$$-1^S * 0.FRAC * 2^{-126} \tag{2.2}$$

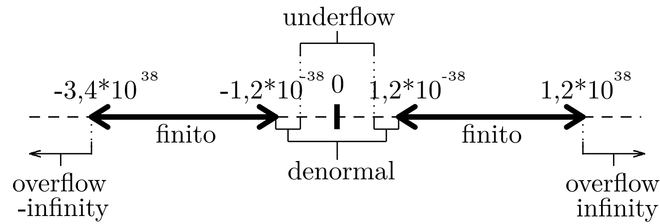

Figura 2.4: Faixa de valores representados em *float*.

A Figura 2.4 mostra a faixa de valores representados no formato *float*. *Underflow* é quando o resultado de uma operação é muito pequeno para a representação, sendo representado como *zero* e sinalizado como *underflow*. *Overflow* é quando o resultado de uma operação é muito grande para a representação, sendo representado como $\pm$*infinito* e sinalizado como *overflow*.

Capítulo 3

Implementação

Este capítulo apresenta a implementação da *unidade de ponto flutuante* com o algoritmo de Tomasulo. A seção 3.1 descreve a interface com o processador. A seção 3.2 explica o estágio de decodificação e seus componentes. A seção 3.3 descreve o estágio de despacho. A seção 3.4 explica o estágio de execução. A seção 3.5 explica o estágio de *write-back* e mecanismo de reordenação.

3.1 Interface com CPU

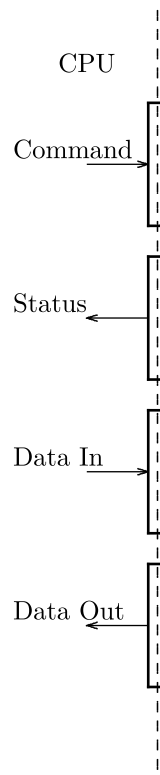

Figura 3.1: Diagrama de bloco da interface com a cpu.

A comunicação com o processador se dá através dos comandos, propostos nesse trabalho, e uma interface de 4 registradores de 32 bits, como mostra a Figura 3.1. Esses registradores têm a função de receber informações da CPU e mandar informações para a mesma. O registrador CMD é o registrador de comando, nele são recebidas as instruções

que o processador está requisitando que a unidade de ponto flutuante execute. Essas instruções são descritas na Figura 3.4, e possuem dois formatos. O primeiro formato, descrito na Figura 3.2, é o da maioria das instruções e contempla todas as instruções funcionais do CP1. O segundo formato, descrito na Figura 3.3, é utilizado para carregar endereços de memória para a unidade de ponto flutuante, para que posteriormente sejam usados em instruções de acesso à memória.

opcode	A	B	D	IMM
7 bits	3 bits	3 bits	3 bits	16 bits

Figura 3.2: Formato de instruções Aritméticas

opcode	dest	addr
1 bit	3 bits	28 bits

Figura 3.3: Formato da intrução de carga de endereços

Na Figura 3.4 temos a descrição dos comandos com três instruções de aritmética de ponto flutuante, sendo as de soma e multiplicação implementadas e a de divisão não, duas instruções de acesso a memória, uma de leitura e outra de escrita, duas de acesso do processador, e uma instrução para carga de endereços de memória. As instruções aritméticas e de memória são similares às do MIPS. As aritméticas usam dois registradores de operando e um para destino. As de memória usam dois registradores, um para valor/destino e um para endereço, e um valor imediato para incremento ou decremento do endereço. As instruções de acesso da CPU servem para que um valor possa ser inserido nos registradores de ponto flutuante (RAT), e buscarem um valor para uso pela CPU. A instrução que insere valores faz uso do registrador DataIn, um registrador de 32 bits onde é colocado o número ponto flutuante que deseja inserir na unidade. A instrução que busca valores utiliza o registrado DataOut de 32 bits, de onde o valor desejado estará disponível para uso pela CPU.

O registrador de status não tem uso no momento pois como o CP1 foi utilizado como um periférico, o processador requisita operações à unidade de ponto flutuante e continua seu funcionamento normal. Por esse motivo a tarefa de tratar exceções se torna uma tarefa complicada. Como um trabalho futuro, este registrador será usado para relatar o status das instruções à medida que elas forem confirmadas, ou caso algum erro ocorra durante uma instrução, avisando que ocorreu *overflow*, por exemplo.

3.2 Estágio de decodificação

O estágio de decodificação, mostrado na Figura 3.5, tem como função receber as instruções da interface com o processador e organizar os operandos e sinais de controle para que a instrução seja executada na unidade funcional apropriada. Ao chegar uma instrução na interface, os sete bits mais significativos vão para uma unidade de controle (CTRL). Ao receber o opcode, a unidade decodifica a instrução e liga sinais que indicam se há escrita em registradores, qual unidade funcional será utilizada, se há escrita no ROB, e se há leitura ou escrita pela CPU. O controle também recebe sinais vindos do estágio de despacho, sinalizando se alguma estação de reserva está cheia. Caso alguma esteja e a

instr.	opcode	formato	descrição
nop	0000000	nop	Instrução para inserir uma bolha no processador
add	0000001	add D, A, B	RAT[D] := RAT[A] + RAT[B] # Soma de ponto flutuante
mul	0000010	mul D, A, B	RAT[D] := RAT[A] * RAT[B] # Multiplicação de ponto flutuante
div	0000011	div D, A, B	RAT[D] := RAT[A] / RAT[B] # Divisão de ponto flutuante, não implementado
lw	0000100	lw D, imm(B)	RAT[D] := MEM[REG_ADDR[B] + SignalEXT[imm]] # Carrega p. f. da mem. para regs.
sw	0000101	sw A, imm(B)	MEM[REG_ADDR[B] + SignalEXT[imm]] := RAT[A] # Carrega p. f. dos regs. para a mem.
rd	0000110	rd A	DataOut := RAT[A] # Carrega valor em p. f. direto nos registradores
wr	0000111	wr D	RAT[D] := DataIn # Busca valor de um registrador especifico
nop	0001xxx	nop	Instrução para inserir uma bolha no processador
nop	001xxxx	nop	Instrução para inserir uma bolha no processador
nop	01xxxxx	nop	Instrução para inserir uma bolha no processador
la	1xxxxxx	la dest, addr	REG_ADDR[dest] := b"00"& addr & b"00"# Carrega end. base de usúario, para uso da unidade

Figura 3.4: ISA da interface com cpu.

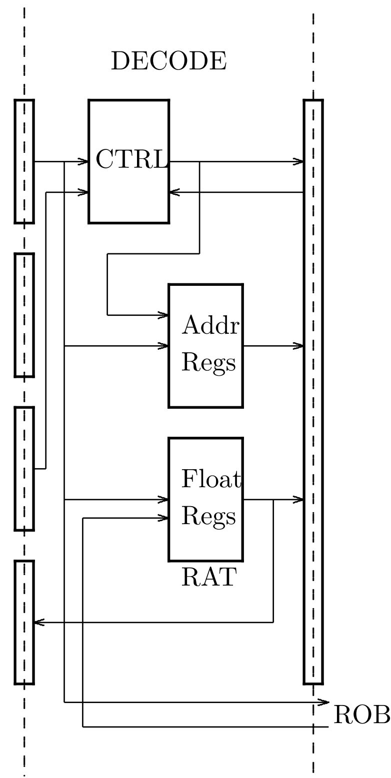

Figura 3.5: Diagrama de bloco do estágio de decodificação.

instrução atual vai utilizá-la, o controle trava o estágio atual e envia um *NOP* no lugar da instrução.

O banco de registradores de endereços (Addr Regs) é um conjunto de oito registradores de 32 bits, Esses registradores servem para armazenar os endereços de memória usados pelas instruções de load e store. Ao chegar a instrução de carga de endereço, os 28 bits menos significativos de CMD recebem dois 0's na parte alta e dois na parte baixa, antes de o endereço ser inserido no banco. Os 0's na parte alta indicam que os endereços utilizados são de usuário [MIP05a, MIP05b, MIP05c], e os dois 0's na parte baixa indicam que o endereço está alinhado na memória. Ao receber uma intrução de acesso à memória, o endereço base vem do banco de endereços. O endereço efetivo é o resultado da soma do imediato (com extensão de sinal) e o endereço base, e esse valor é enviado para o estágio de despacho.

O banco de registradores de ponto flutuante (Float Regs) é um conjunto de oito registradores de 36 bits, sendo 32 de valor, 3 de índice no ROB e 1 bit de registrador valido. O banco de registradores de ponto flutuante tem como função, além de armazenar os valores, renomear os registradores, sendo denominado de *Register Alias Table* (RAT). No momento em que chega uma instrução que atualizará um registrador no RAT, este registrador é marcado como inválido e é inserido o índice do ROB ao qual a instrução foi alocada. Esse índice é o novo nome do valor válido daquela entrada, e é passado para o estágio de despacho.

Os valores de operandos e endereços são então enviados para o estágio de despacho. Se um operando necessário para a instrução está inválido, é feita uma comparação pelo índice do ROB (*tag*) com valor no ROB. Caso o valor esteja disponível, ele é enviado para o estágio de despacho. Caso contrário é enviado o índice (*tag*) e é sinalizado para o despacho que o valor ainda está inválido.

3.3 Estágio de despacho

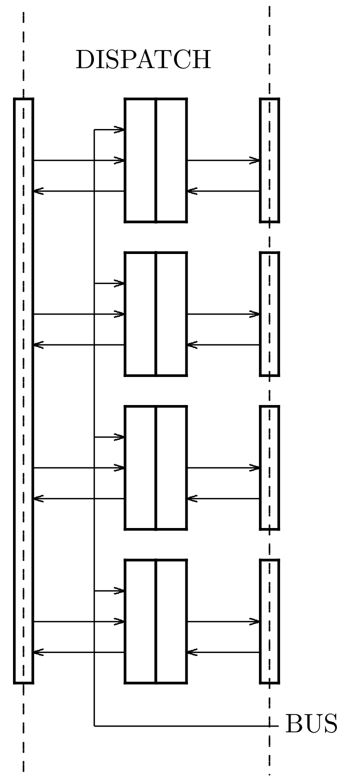

Figura 3.6: Diagrama de bloco do estágio de despacho.

O estágio de despacho, mostrado na Figura 3.6, é composto por 4 estações de reserva (*Reservation Station*, RS), e tem como função preparar e organizar os operandos e demais informações necessárias na execução, sendo responsável também pelo escalonamento dinâmico das instruções. Há uma estação de reserva para cada unidade funcional, e cada RS tem capacidade para duas instruções, nomeadas internamente de 1 e 0, sendo que 0 é a mais recente e 1 a mais antiga. Quando uma instrução chega e está esperando algum operando, ela fica na estação de reserva até que o(s) operando(s) que esta(ão) faltando apareça(m) no barramento. Caso mais de uma instrução esteja pronta para executar, e a instrução mais antiga ainda não tenha seus operandos, a instrução mais recente passa para a execução.

As estações de reserva ativam sinais de "cheio" caso estejam com os dois espaços cheios e estiverem aguardando operandos, ou se a unidade funcional estiver ocupada. Esses sinais são enviados para a unidade de controle, no estágio de decodificação.

3.4 Estágio de execução

O estágio de execução, com seu diagrama de blocos mostrado na Figura 3.7, é responsável pelo processamento das instruções nas unidades funcionais. O CP1 contém 3 unidades funcionas implementadas e uma não implementada. As unidades funcionais aritméticas utilizadas são as descritas em [TDD+15], e a unidade funcional de acesso à memória é a versão de simulação da RAM utilizada no cMIPS [Hex15].

A unidade de divisão não foi implementada, portanto foi fixado no registrador de interface com o estágio de write-back o valor ’0’ para o resultado.

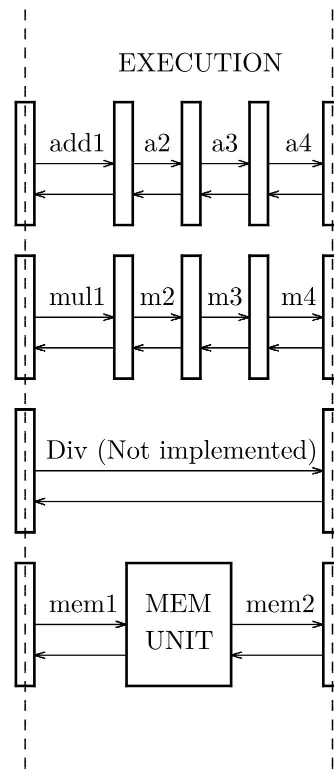

Figura 3.7: Diagrama de bloco do estágio de execução.

3.4.1 Multiplicador

A unidade funcional de multiplicação foi implementada com base no Apendice J de [HP11], seguindo o padrão IEEE-754, e consiste de três estágios. A unidade pode conter até três operações sendo executas concorrentemente. Caso haja um valor válido no estágio final da execução, mas o barramento esteja ocupado, a unidade propaga um sinal de espera vindo do estágio de write-back. O sinal de espera recebido só é propagado caso o pipeline da unidade esteja cheio. Nesse caso, um sinal é enviado para o estágio de despacho, avisando que a unidade está ocupada e não pode receber instruções.

3.4.2 Somador

A unidade funcional de soma é similar à de multiplicação.

3.4.3 Memória

Como já dito, o modelo para a memória foi retirado do cMIPS [Hex15]. A latência de acesso à memória, tanto leitura quanto escrita, é de um ciclo. Não há propagação de sinal de espera na unidade de memória, pois o barramento do estágio de *write-back* é sempre liberado com prioridade para a memória. A geração de sinais de espera do barramento é descrita na seção 3.5.

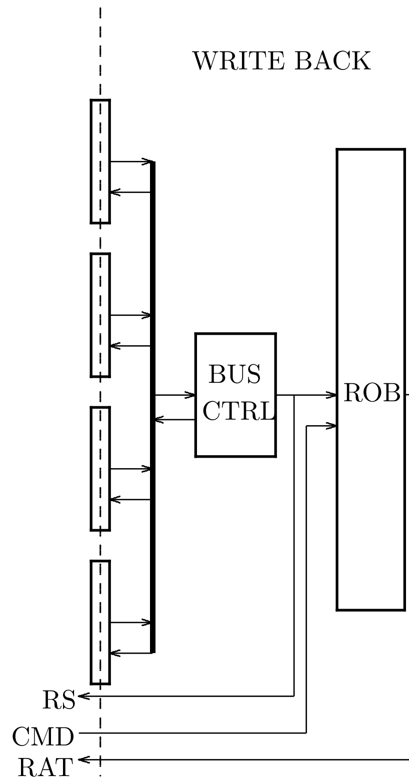

Figura 3.8: Diagrama de bloco do estágio de *(*write-back).

3.5 Estágio de *write-back*

O estágio de *write-back*, mostrado na Figura 3.5, tem como função priorizar a saída das unidades funcionais e a re-ordenação da confirmação das instruções, para a ordem do programa. Este estágio tem três componentes, o barramento comum de dados, o controle do barramento (Bus Ctrl) e o *buffer* de reordenação (*reorder buffer*, ROB).

As unidades funcionais tentam acessar o barramento envianto um sinal de pronto para o controle do barramento. O controle recebe os sinais das unidades, arbitrando qual irá usar o barramento. Essa escolha é feita estaticamente por ordem de prioridade, sendo dada a maior prioridade à memória, seguido de soma, multiplicação e divisão. Quando uma unidade recebe a posse do barramento, as demais recebem um sinal para esperarem. O controle então liga os sinais para que os valores da unidade escolhida sejam visíveis no barramento, sendo direcionado para as *estações de reserva* e ROB, na posição para o qual este valor é endereçado (*tag*).

O *buffer* de re-ordenação é implementado como um banco de registradores com 8 registradores de 38 bits, sendo 32 bits do valor de ponto flutuante, 3 bits de destino (índice da RAT em qual deve ser inserido o valor), 1 bit de validade do registrador, 1 bit de habilitação de escrita na RAT, e 1 bit de pronto, que indica que o registrador tem tudo o que é necessario para confirmar. O ROB usa dois registradores de 3 bits, denominados de *head* e *tail*, para apontar o próximo registrador a ser confirmado, e o próximo registrador livre para uso, respectivamente. Quando uma instrução chega, o controle aloca um registrador da ROB, que é o registrador apontado por *tail*, que então é incrementado.

Quando o registrador da ROB apontado por *head* está com o bit de pronto ligado, este pode ser confirmado. Quando é feita a confirmação, o valor é inserido na RAT, caso o

bit de escrita na RAT esteja ligado, o índice de ROB que estiver na RAT seja o mesmo que o *head* e a entrada da RAT estiver inválida. Após a confirmação, o registrador *head* é incrementado.

Capítulo 4

Testes

Este capítulo apresenta testes gerados para verificar o modelo do circuíto do CP1 implementado. O modelo foi implementado em VHDL, utilizando GHDL e GTKWAVE para compilar e visualizar, respectivamente. O ambiente de teste montado feito utilizando o *coprocessador 1* como um componente dentro de uma modelo de simulação. A verificação é feita pré-calculando valores em um computador pessoal, com um *i5 3ª geração* e rodando num Debian 9. Os cálculos foram computados no processador, e essas mesmas operações foram repetidas na unidade e conferidas por meio de comandos de leitura (RD), fazendo uso de *assert* e comparando com o valor resultante do PC. No total foram 3824 linhas de código, incluindo comentários, onde 3550 são em VHDL e 274 em C.

O Programa 4.1 mostra o vetor de operações realizadas para testar a unidade. Este teste estrutural utiliza todas as instruções propostas, e também utiliza todos os componentes implementados dentro da CP1. Primeiro são feitos testes de leitura e inserção de dados pelo processador. Depois são realizados testes das unidades funcionais de soma, multiplicação, e leitura e escrita na memória.

```
(x"0e000000",x"3f800000",x"00000000",'0'), -- rat[0]        <= 1.0
(x"0e010000",x"40000000",x"00000000",'0'), -- rat[1]        <= 2.0
(x"0e020000",x"40800000",x"00000000",'0'), -- rat[2]        <= 4.0
(x"0e030000",x"41000000",x"00000000",'0'), -- rat[3]        <= 8.0
(x"0c000000",x"00000000",x"00000000",'0'), -- dataOut       <= rat[0]
(x"0c400000",x"00000000",x"00000000",'0'), -- dataOut       <= rat[1]
(x"0c800000",x"00000000",x"3f800000",'0'), -- dataOut       <= rat[2]
(x"0cc00000",x"00000000",x"40000000",'0'), -- dataOut       <= rat[3]
(x"04100000",x"00000000",x"40800000",'0'), -- rat[0]        <= rat[0] x rat[2] ( 4.0)
(x"04510000",x"00000000",x"41000000",'0'), -- rat[1]        <= rat[1] x rat[2] ( 8.0)
(x"020a0000",x"00000000",x"00000000",'0'), -- rat[2]        <= rat[0] + rat[1] (12.0)
(x"80010000",x"00000000",x"00000000",'0'), -- r_addr[0]     <= 0x00040000
(x"0e070000",x"41800000",x"00000000",'0'), -- rat[7]        <= 16.0
(x"0BC00000",x"00000000",x"00000000",'0'), -- M[r_addr[0]]  <= rat[7]
(x"08030000",x"00000000",x"00000000",'0'), -- rat[3]        <= M[r_addr[0]] (16.0)
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"00000000",x"00000000",x"00000000",'0'),
(x"0c000000",x"00000000",x"00000000",'0'), -- dataOut   <= rat[0]
(x"0c400000",x"00000000",x"00000000",'0'), -- dataOut   <= rat[1]
(x"0c800000",x"00000000",x"40800000",'0'), -- dataOut   <= rat[2]
(x"0cc00000",x"00000000",x"41000000",'0'), -- dataOut   <= rat[3]
(x"00000000",x"00000000",x"41400000",'0'),
(x"00000000",x"00000000",x"41800000",'0'),
(x"00000000",x"00000000",x"00000000",'0')
```

Programa 4.1: Vetor de teste estrutural.

A Figura 4.1 mostra o diagrama de tempo do teste das instruções RD e WR. Após o sinal de *reset* o processador envia quatro instruções WR para o CP1, carregando no RAT

os valores 1, 2, 4 e 8 nas posições de 0 a 3. Assim que uma instrução chega no estágio de decodificação, ela é separada em opcode, índices dos operandos (*opa e opb*), índice de destino (*dest*) e imediato *imm*, e é mandada então para o controle que ativa os sinais de controle de escrita na RAT (*ctrl_wr*). Com o sinal de escrita ativo, na borda de subida do relógio o valor do registrador *data input* é inserido no registrador da RAT apontado por *dest*. E em seguida são mandadas quatro instruções WR lendo os registradores de 0 a 3 do RAT. O sinal de controle ativo para instrução é o de escrita do registrador *data out*, denotado por *ctrl_out_en*. Com o sinal de controle ativo, na borda de subida do relógio o valor da RAT apontado por *opa* é enviado para o processador e então é feita a conferência do resultado.

As Figuras 4.2 a 4.8, mostram os diagramas de tempo de teste das unidades funcionais e do algoritmo de Tomasulo em si. Quando uma instrução aritmética chega no estágio de decodificação, (Figura 4.2), são ativados os sinais de controle para escrita na RAT, escrita na ROB (*wr_on_rob*), e o sinal de qual unidade funcional vai ser utilizada, *ctrl_add* para soma e *ctrl_mul* para multiplicação. Com os sinais de controle ativos, na borda de subida do relógio os valores dos registradores do RAT apontados por *opa* e *opb* são enviados para o estágio de despacho, o registrador apontado por *dest* é marcado como inválido e então é incrementado o valor de *tail* ao RAT.

As intruções de memória funcionam quase da mesma forma que as de aritméticas. É ativado o sinal de controle para indicar que a unidade funcional é a de memória (*ctrl_mem*), e o que diferencia a instrução LW da de SW é o sinal de controle de escrita na RAT.

A Figura 4.3 mostra as instruções no estágio de despacho. Elas são guardados nas estações de guardas que o sinal de controle habilita. No próximo ciclo do relógio, se os operandos tiverem carregados, estes seguem para as unidades funcionais das respectivas estações de reserva.

Na Figura 4.4 temos o diagrama de tempo do estágio de execução. As unidades funcionais, ao terminarem sua execução, sinalizam para o estágio de *write-back* que terminaram, para que o BUS_CTRL organize o uso do barramento.

Nas Figura 4.5, 4.6 e 4.7 temos o diagrama de tempo do estágio de *write-back*, que mostra, no destino do BUS, que as intruções foram executadas fora de ordem, mas a confirmação foi feito na ordem, mostrado pelo valor de *head*.

Na Figura 4.8 vemos o envio dos valores calculados ao processador para a verificação.

Além dos testes estruturais, foram executados testes de *stress* na unidade, utilizando uma multiplicação de matrizes 10x10 para este fim. Estes testes apresentam erros na sincronização das unidades e componentes do coprocessador, quando as estações de guarda ou ROB ficam "cheios" gerando *stall's* até que haja espaço livre em um deles. Este erro se mostrou de difícil depuração, pois, como não temos uma noção exata da ordem em que as operações acontecem, é difícil isolar o problema e replicar o erro em um teste "mais bem comportado".

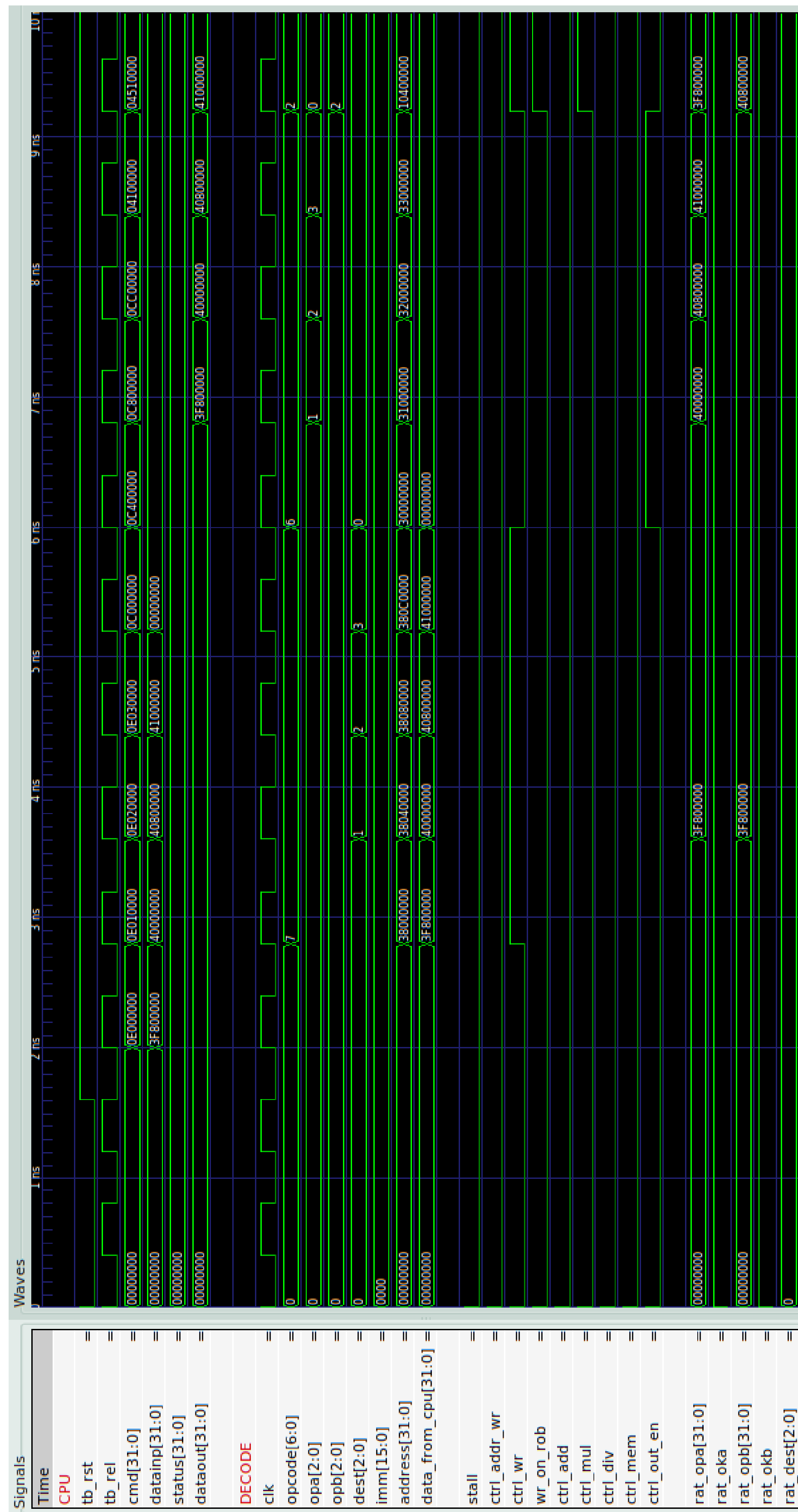

Figura 4.1: Interface com CPU e decodificação
Teste de instruções RD e WR

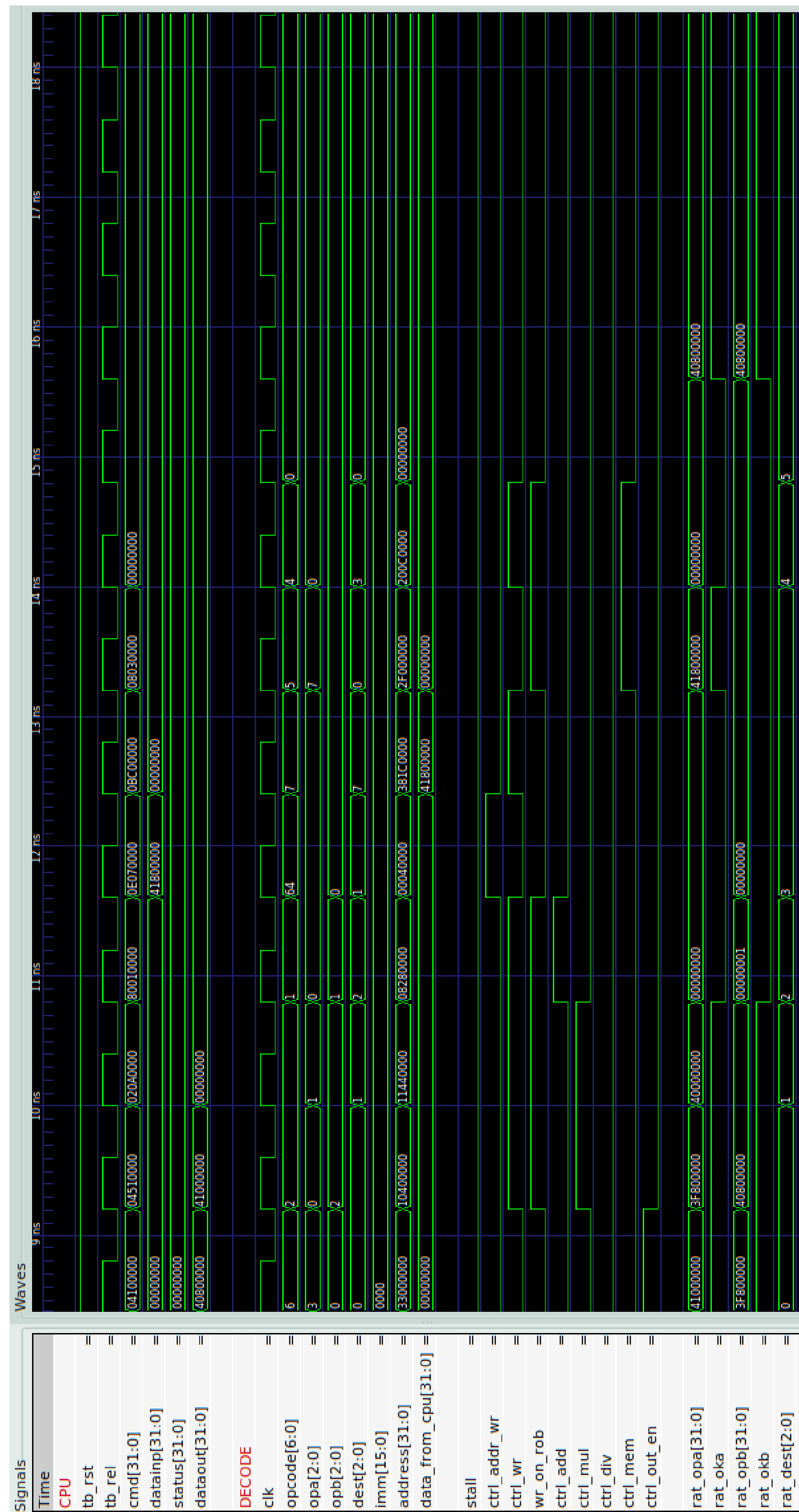

Figura 4.2: Interface com CPU e decodificação
Teste de LW, SW, ADD e MUL

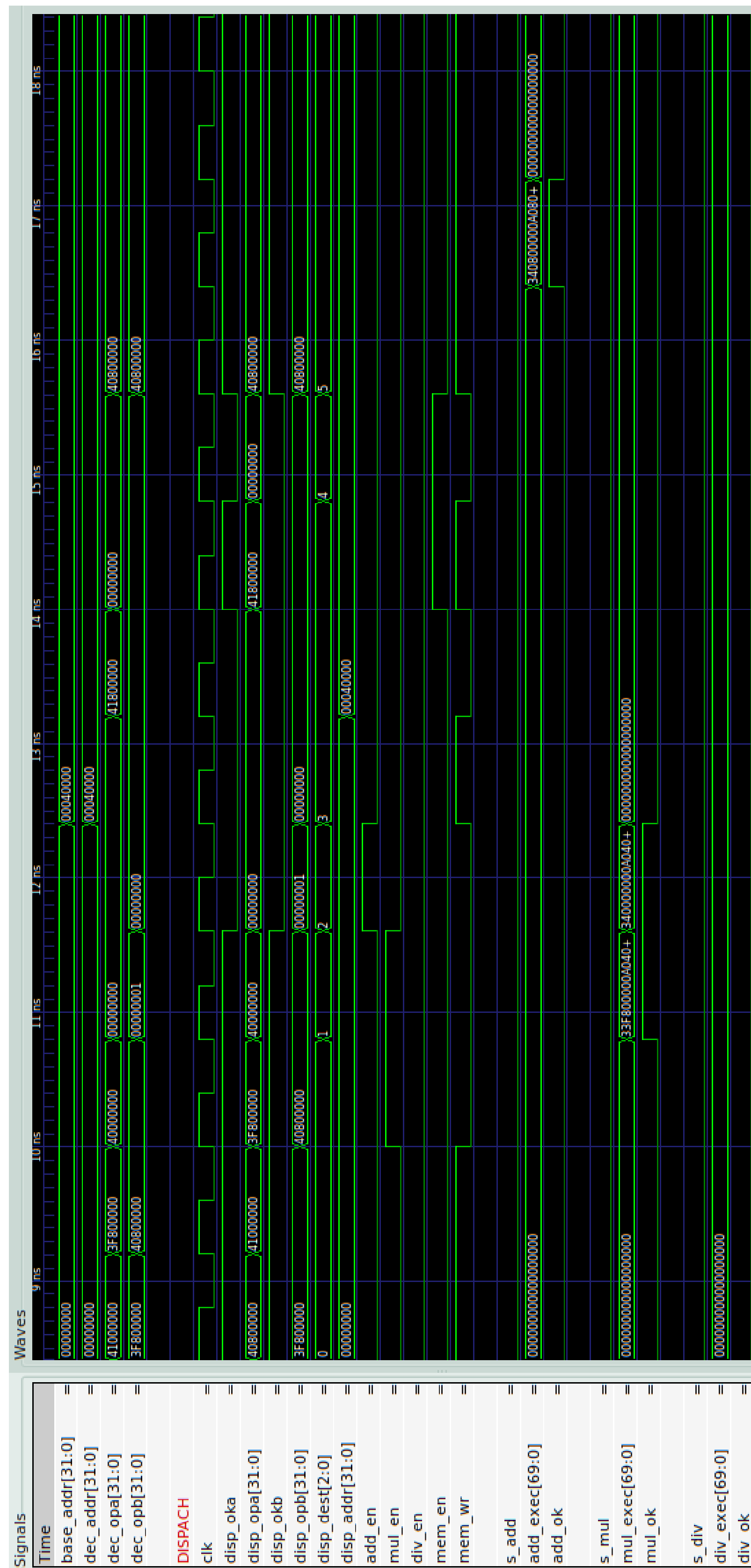

Figura 4.3: Decodificação e despacho
Teste de LW, SW, ADD e MUL

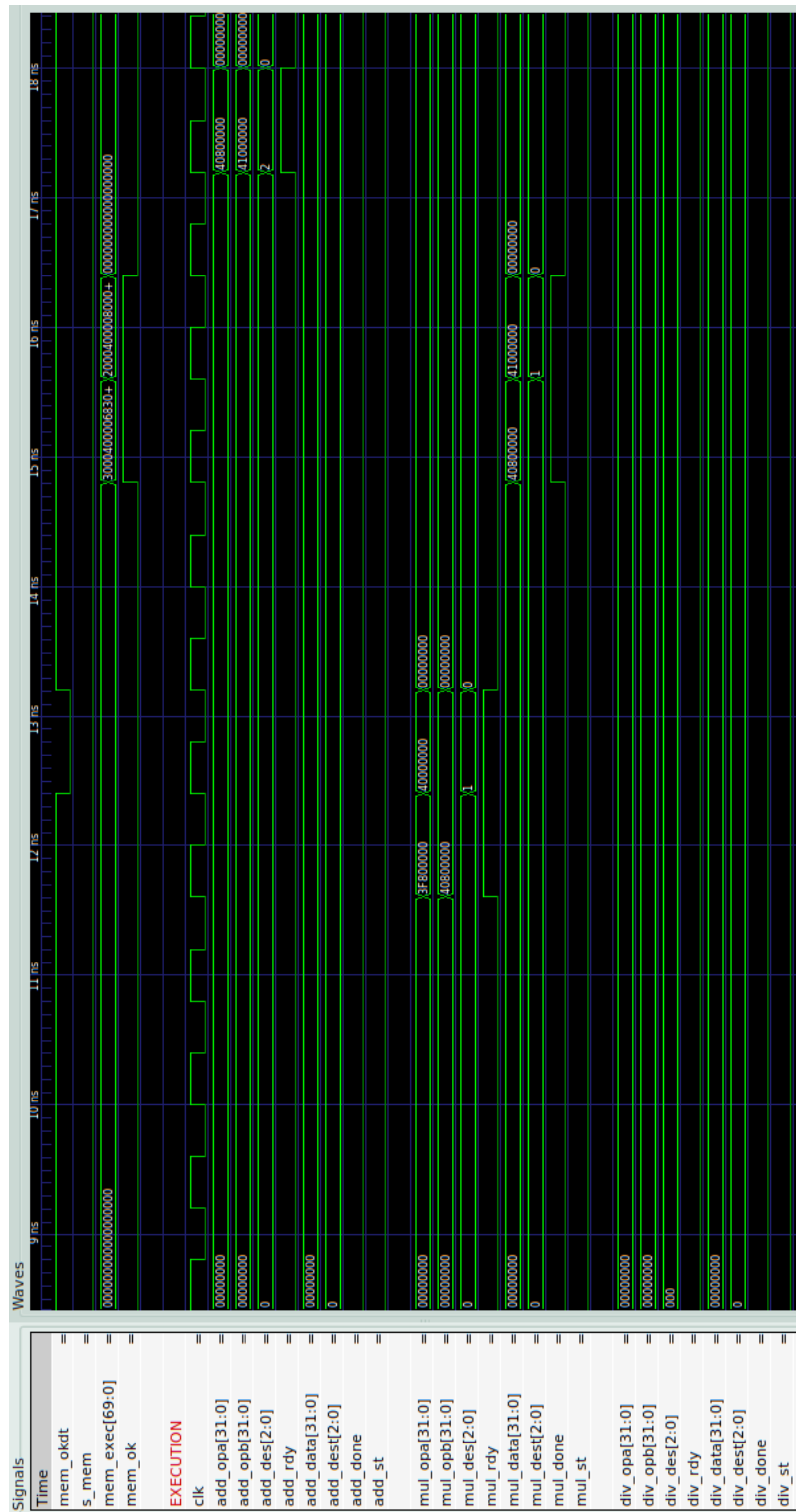

Figura 4.4: Despacho e execução
Teste de LW, SW, ADD e MUL

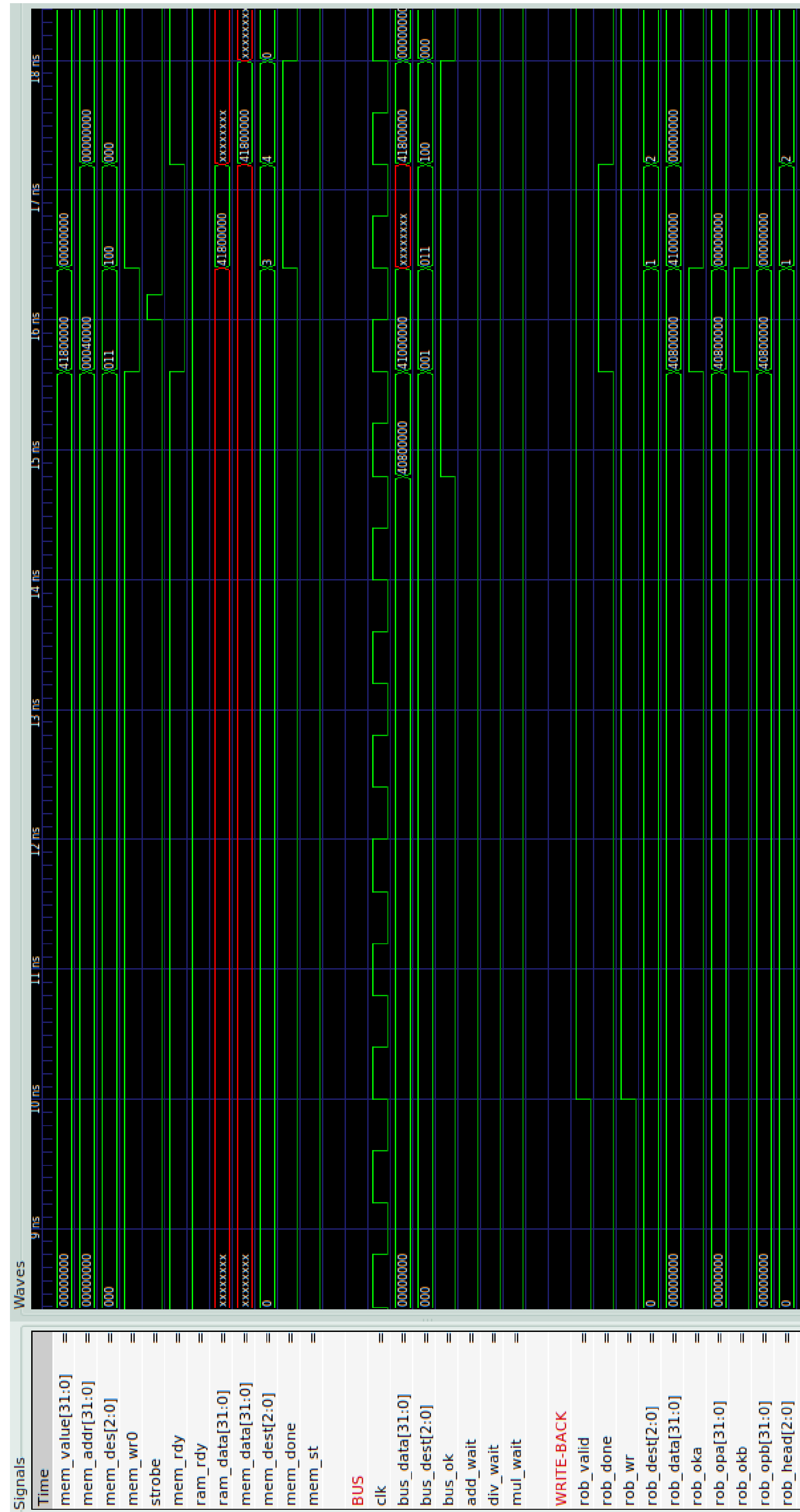

Figura 4.5: Execução e *write-back*
Teste de LW, SW, ADD e MUL

Figura 4.6: *Write-back*
Teste de LW, SW, ADD e MUL

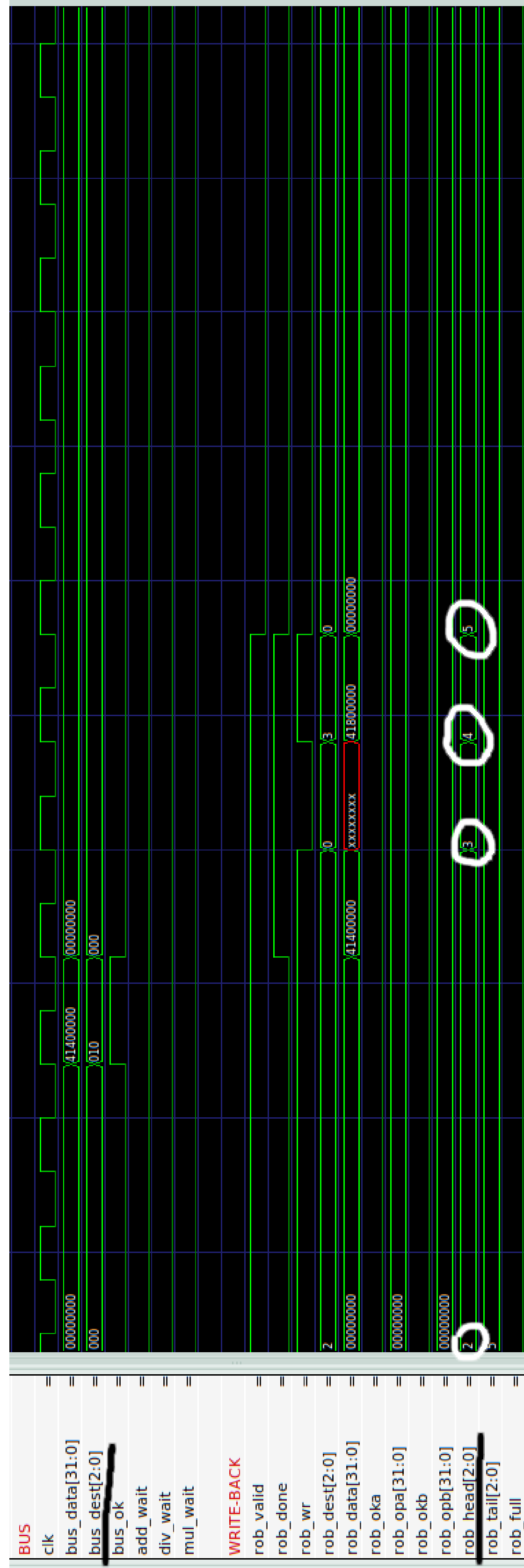

Figura 4.7: *Write-back* 2
Teste de LW, SW, ADD e MUL

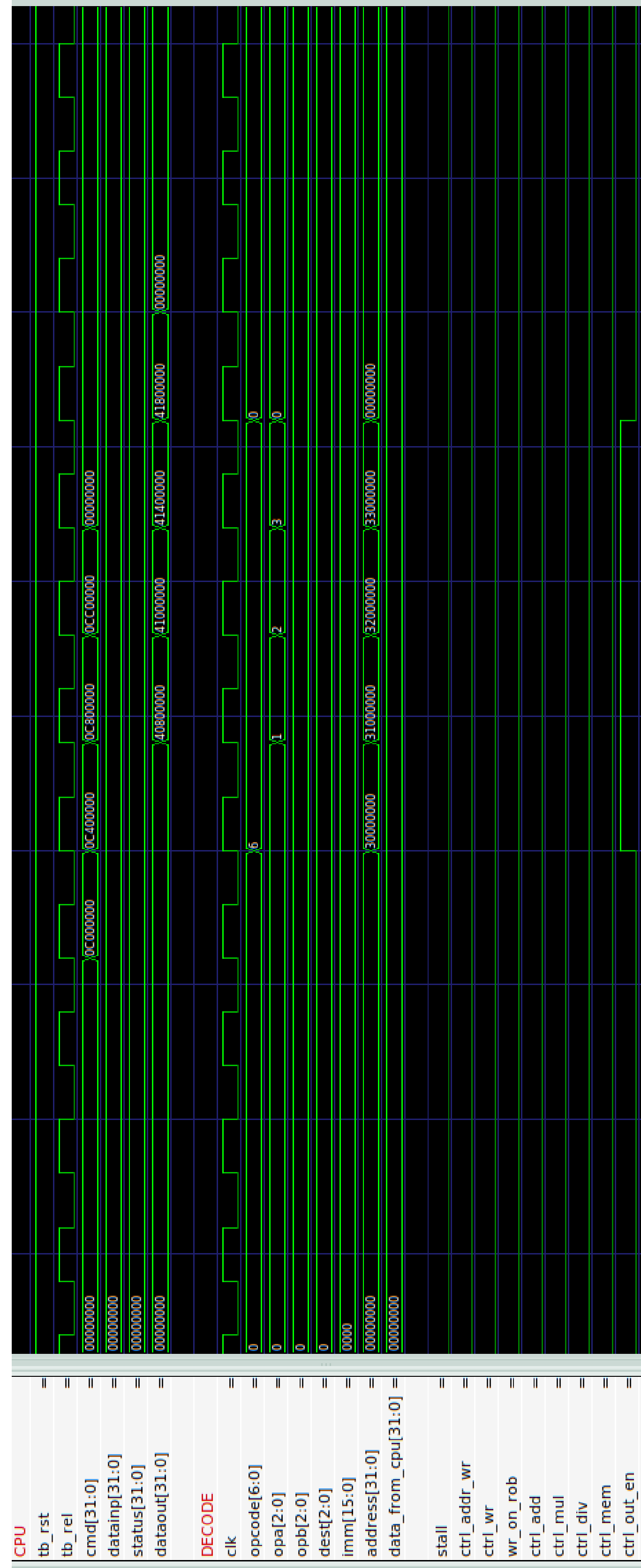

Figura 4.8: Interface com CPU e decodificação 2
Teste de LW, SW, ADD e MUL

Capítulo 5

Conclusões e Trabalhos futuros

O algoritmo de Tomasulo se mostra uma útima opção para conciliar diversas unidades funcionais com latências diferentes, mantendo uma boa performance e possibilitando certo ganho com o escalonamento dinâmico e execução *fora de ordem*. Com esses ganhos advém uma grande dificuldade de acompanhar o que acontece dentro do circuito, dificultando muito a depuração de erros descobertos no decorrer de testes. Nos testes mostrados no Capítulo 4, foram apresentados sequências visando executar uma parte específica do *hardware*, mostrando que individualmente os componentes estão funcionando corretamente.

Como trabalhos futuros é necessário depurar os problemas encontrados no coprocessador, adicionar uma unidade de divisão, colocar a unidade funcional no processador cMIPS e sintetizar em um FPGA, e testar a inserção de uma unidade para aritmética e memória de inteiros, visando contruir um processador superescalar utilizando o algoritmo de Tomasulo.

Referências Bibliográficas

[Hex15] Roberto A Hexsel. cMIPS – a synthesizable VHDL model for the classical five stage pipeline. Repositório de *software*, Depto de Informática, UFPR, 2015. Disponível em <https://github.com/rhexsel/cmips>.

[HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[IEE08] IEEE-754. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[MIP05a] MIPS. MIPS32 architecture for programmers, volume I: Introduction to the MIPS32 architecture. Rev. 2.50, MIPS Technologies, Inc, 2005.

[MIP05b] MIPS. MIPS32 architecture for programmers, volume II: The MIPS32 instruction set. Rev. 2.50, MIPS Technologies, Inc, 2005.

[MIP05c] MIPS. MIPS32 architecture for programmers, volume III: The MIPS32 privileged resource architecture. Rev. 2.50, MIPS Technologies, Inc, 2005.

[TDD+15] Cainã C. Trevisan, Clara D. H. Daru, Jean C. K. Diogo, João M. P. Filho, and Roberto A. Hexsel. Modelagem e implementação em vhdl de unidade aritmética de ponto flutuante segundo o padrão ieee-754. *Workshop de Iniciação Científica, no XVI WSCAD-SSC'15*, pages 1–6, October 2015.

[Tom67] R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.*, 11(1):25–33, January 1967.