RAFAEL ROCHA DE CARVALHO

ANÁLISE DE ESCALABILIDADE NO *BUFFER BCD* DO *CLUPA*

Trabalho de Graduação apresentado à disciplina de CI083 - Trabalho de Graduação em Arquitetura e Organização de Computadores II como requisito à conclusão do curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas Universidade Federal do Paraná.
Orientador: Roberto A Hexsel.

CURITIBA
2016

SUMÁRIO

LISTA DE FIGURAS iii

LISTA DE TABELAS iv

RESUMO v

ABSTRACT vi

1 INTRODUÇÃO 1

2 MIPS32 2

2.1 Estágios do *MIPS32* 2

2.2 Registradores MIPS 3

2.3 Formato das instruções do *MIPS* 4

2.4 Usos do *MIPS* 5

3 CMIPS 6

4 CLUPA 8

4.1 Hardware 8

4.2 Buffer 10

4.2.1 Verificação da disponibilidade do *Buffer* 13

4.3 Escrita no *Buffer* 13

4.4 Leitura do *Buffer* 14

5 AVALIAÇÃO DE DESEMPENHO DO *BUFFER* DO CLUPA. 15

5.1 Cargas de simulação 15

5.2 Programa de simulação 17

5.3 Métricas das simulações 17

5.4 Problemas iniciais 18

5.5 Geradores de números da sequência de *Fibonacci* e primos 18

5.6 Resultados 21

5.6.1 Simulação 1 21
5.6.1.1 Tamanho de fila 1 22
5.6.1.2 Tamanho de fila 2 23
5.6.1.3 Tamanho de fila 4 24
5.6.1.4 Tamanho de fila 8 25
5.6.1.5 Tamanho de fila 16 26
5.6.1.6 Tamanho de fila 32 27
5.6.1.7 Tamanho de fila 64 28
5.6.1.8 Tamanho de fila 128 a 256 29
5.6.1.9 Análise dos resultados da simulação 1 29
5.6.2 Simulação 2 30
5.6.2.1 Análise dos resultados da simulação 2 31
5.6.3 Simulação 3 32
5.6.3.1 Tamanho de fila 1 32
5.6.3.2 Tamanho de fila 2 à 256 33
5.6.3.3 Análise dos resultados da simulação 3 34
5.6.4 Simulação 4 34
5.6.5 Simulação 5 35

6 CONCLUSÃO **37**

7 APÊNDICES **38**
7.1 Script de Testes 38
7.1.1 Código 38
7.1.2 Diretórios 40

BIBLIOGRAFIA **42**

LISTA DE FIGURAS

2.1 Modelo de execução do Pipeline do *MIPS*. 3
2.2 Registradores do *MIPS*. 4
2.3 Tipo de instruções do *MIPS*. 4

3.1 Diagrama demonstrando a implentação do cMIPS em um *testbench*. 6

4.1 Visão geral do sistema CLupa. 8
4.2 Hardware do cLUPA. 9
4.3 *Buffer* de comunicação direta. 11

5.1 Números primos e série de Fibonacci(a escala do gráfico é logarítmica). 19
5.2 Tempo para gerar séries de primos e fibonacci em execução no núcleo *cMips-0*. . 20
5.3 Teste de corretude do *Buffer*. 21
5.4 *Buffer* com fila de tamanho 1. 22
5.5 *Buffer* com fila de tamanho 2. 23
5.6 *Buffer* com fila de tamanho 4. 24
5.7 *Buffer* com fila de tamanho 8. 25
5.8 *Buffer* com fila de tamanho 16. 26
5.9 *Buffer* com fila de tamanho 32. 27
5.10 *Buffer* com fila de tamanho 64. 28
5.11 *Buffer* com fila de tamanho 128 à 256. 29
5.12 Uso do *Buffer* em porcentagem em relação com o tamanho da fila. 30
5.13 *Buffer* com fila de tamanho 1 a 256. 31
5.14 *Buffer* com fila de tamanho 1. 32
5.15 *Buffer* com fila de tamanho 2 á 256 e tempo simulado de 1500 μs. 33
5.16 *Buffer* com fila de tamanho 1 a 256. 34
5.17 Comportamento *Buffer* com fila de tamanho 256 e tempo de execução de 2 ms. . 35

LISTA DE TABELAS

5.1 Cargas de simulação. 16

RESUMO

Este trabalho investiga o comportamento da comunicação entre dois processadores através de uma fila implementada em *hardware*. Deseja-se estudar escabilidade da fila de comunicação para determinar seu tamanho, em função de possiveis aplicações.

Foram realizados testes simulando diferentes cargas nos dois processadores. Os programas de teste foram escritos em C, e simulam diferentes taxas de inserção e de remoção de inteiros na fila.

Para estes programas de teste, filas pequenas, com capacidade para um ou dois elementos, são adequadas segundo a métrica de utilização.

Palavras-chave: Escabilidade, MultiProcessadores, MIPS, CMips.

ABSTRACT

This work investigates the behavior of a communication system between two processors, consisting of a hardware queue. The queue size is to be scaled as a function of the demand imposed by the candidate applications.

Simulation tests were performed for different processing loads on the two processors. The test programs were written in C, and simulate differing rates of insertion and removal of items from the queue.

For the set of test programs, it was found that queues of capacity one or two are adequate, when considering queue utilization.

Keywords: Scalability, MultiProcessors, MIPS, CMips.

CAPÍTULO 1

INTRODUÇÃO

Esta monografia descreve a avaliação do comportamento da comunicação através de filas no *CLupa*. *CLupa* é uma ferramenta criada por Edmar A. Bellorini, em conjunto com os professores Roberto André Hexsel e Marcio S. Oyamada. Esta ferramenta, descrita no capítulo 4, tem como objetivo ampliar documentos e alterar a cor de visualização destes, para pessoas com necessidades especias no campo da visão [1].

Este trabalho tem por objetivo possibilitar, através de uma análise dos dados, qual tamanho a ser usado pelo *Buffer*, gerando assim conhecimento nas áreas de hardware como por exemplo através da compreensão do funcionamento de multiprocessadores, o entendimento da utilização de uma fila entre dois processadores, e a compreensão de utilização do espaço de armazenamento em hardware.

O *CLupa* foi desenvolvido em cima do *CMips* e emprega dois processadores desse modelo interligados por uma fila. O *CMips* é um modelo projetado para melhorar o ensino das matérias que trabalham com arquitetura de computadores e seus detalhes são explanados no capítulo 3. O *CMips* é uma versão dos processadores *MIPS* [2] e a estrutura, definições e uso destes processadores são explanados no capítulo 2.

Para avaliar o comportamento do *BCD* foram utilizadas simulações com diferentes tipos de cargas. Cada uma dessas simulações foi analizada utilizando-se de métricas a fim de obter-se qual o melhor dos tamanhos para o *BCD*, baseado em seu comportamento. As simulações e seus resultados se encontram no capítulo 5. Além dos resultados, são apresentados outros detalhes envolvendo os testes, tais como problemas encontrados no desenvolvimento. Detalhes tais como estruturas de pastas e execução do *script* de testes, são explanados no capítulo 7.

CAPÍTULO 2

MIPS32

MIPS é uma arquitetura de processadores da classe RISC [2]. Essa arquitetura tem como fundamento duas técnicas, que são a exploração de paralelismo em nível de instrução e o uso de caches [3]. O *MIPS* tem como características um grande número de registradores e uma variedade de instruções com estilos bem definidos [2].

O processador *MIPS* foi desenvolvido por John Hennessy em 1981 e tinha como objetivo aumentar o desempenho com o uso de *pipelines* para as instruções [4], segmentando o processador em cinco estágios de execução [5].

Os processadores *MIPS* possuem 32 registradores para uso geral permitindo assim que o compilador otimize o código gerado, mantendo os dados frequentemente usados em registradores [5]. As características desses registradores são descritas na sessão 2.2.

Quanto às instruções, o *MIPS* possui um conjunto que inclui operações aritméticas, tais como soma, multiplicação, etc... de acesso a memória e de controle de fluxo. As instruções estão descritas na seção 2.3.

2.1 Estágios do *MIPS32*

A implementação "clássica" do conjunto de instruções *MIPS32* é dividida em cinco estágios, e o objetivo dessa divisão é permitir que o processador possa conter várias instruções ativas ao mesmo tempo, iniciando uma nova instrução a cada ciclo de *clock*. Toda instrução do *MIPS* executa nos cinco estágios, mesmo que essa instrução não utilize algum desses estágios [6]. Cada estágio é definido a seguir:

Instruction Fetch (IF) Neste estágio o processador lê a próxima instrução da memória usando o endereço contido no registrador PC (Program Counter) e salva a instrução no registrador IR (Instruction Register).

Instruction Decode (ID) Decodifica a instrução recebida do estágio anterior, calcula o próximo PC e verifica os operandos necessários para instrução que foi recebida.

Execution (ALU) Executa a instrução do estágio anterior e qualquer operação que envolve a

ULA (Unidade de lógica e aritimética) é executada nesse estágio.

Memory Access (MEM) Estágio que executa acessos à memória. É nesse estágio em que instruções de Load e Store são executados; outras instruções que não envolvam memória passam por esse estágio sem fazer nada.

Write Back (WB) Se uma instrução tiver algum resultado que deve ser armazenado em um registrador, a escrita ocorre nesse estágio.

O modelo de execução dos estágios é mostrado na figura 2.1 [1], que indica como o *MIPS* trabalha com as instruções, executando as múltiplas instruções ao mesmo tempo.

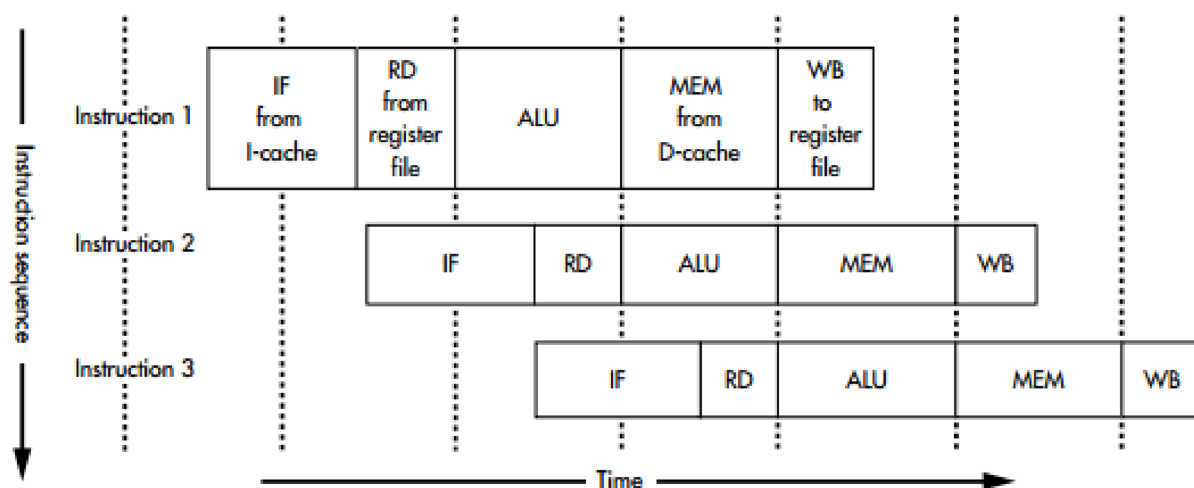


Figura 2.1: Modelo de execução do Pipeline do *MIPS*.

Na figura 2.1 é mostrado como o processador lida com as instruções, e cada uma completa um estágio a cada ciclo de relógio, fazendo com que o processador execute várias instruções ao mesmo tempo.

Na seção a seguir será explanado o funcionamento dos registradores do *MIPS* e as suas características.

2.2 Registradores MIPS

Os processadores *MIPS* possuem um conjunto de 32 registradores, e estes são divididos em 12 categorias. Os registradores são numerados de 0 a 31 e cada qual possui um determinado uso, os quais são mostradas na tabela 2.2 [5].

Registradores	Nome	Descrição
0	zero	Sempre contém o valor zero.
1	at	*Assembler Temporary*: usado pelo montador.
2-3	v0-v1	Valor do retorno de uma chamada de função.
4-7	a0-a3	Quatro primeiros parâmetros para uma chamada de função.
8-15	t0-t7	Variáveis temporárias; não precisam ser preservadas por funções.
16-23	s0-s7	Variáveis salvadas; devem ser preservadas.
24-25	t8-t9	Variáveis temporárias.
26-27	k0-k1	Registradores para o uso do sistema operacional.
28	gp	*global pointer*.
29	sp	*stack pointer*.
30	fp	Apontador para registro de ativações ou variáveis de subrotinas.
31	ra	Endereço de retorno da última chamada de subrotina.

Figura 2.2: Registradores do *MIPS*.

2.3 Formato das instruções do *MIPS*

As instruções dos processadores *MIPS* são divididas em três formatos. A figura 2.3 [1] mostra quais são esses tipos e o seus formatos.

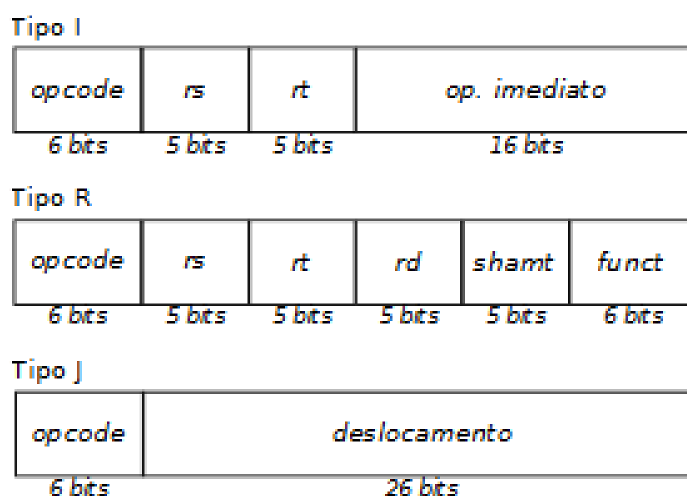


Figura 2.3: Tipo de instruções do *MIPS*.

A figura 2.3 mostra os três tipos de instrução. Todas as instruções tem 32 *bits* com um *opcode* de 6 *bits* que são usados para a identificação da instrução.

Instrução do Tipo I Tipo de instruções que operam com constantes, com a memória ou saltos condicionais, sendo o campo *rs* o registrador fonte da instrução e o registrador *rt* o registrador de destino, e o campo op.imediato é codificado em complemento de dois sempre que necessário.

Instrução do Tipo R Tipo de instruções que fazem operações com registradores envolvendo lógica ou aritmética. O campo *rs* representa o registrador que é o primeiro operando da operação, o campo *rt* representa o registrador que é segundo operando, o campo *rd* representa o registrador de destino. O campo *funct* define a operação a ser executada, e em casos de operações que utilizam-se de deslocamento, o campo *shamt* define quantos *bits* serão deslocados.

Instrução do Tipo J Tipo de instruções de saltos incondicionais, que contém apenas o *op-code* da instrução e um campo *deslocamento* que determina o valor que será atribuído no registrador *PC*.

2.4 Usos do *MIPS*

Os processadores *MIPS* tem sido utilizados em vários projetos desde que foram criados, sendo bastante utilizado em sistemas embarcados, roteadores, videogames e em computadores [4]. Este uso se dá pelo fato da arquitetura ser amplamente suportada, com um conjunto de ferramentas bem documentadas e eficiente, com um bom custo-beneficio [2]. Os exemplos a seguir mostram alguns projetos que utilizaram-se dos processadores *MIPS*.

Roteadores 7600 CISCO Criado para ser altamente escalável, com alta performance como exigida para serviços de video, voz e dados entre outros, emprega como CPU um processador *MIPS* de 600 *MHz* [5].

Nintendo 64 Console de videogame criado em 1996 pela empresa *Nintendo* tinha como CPU um processador *MIPS R4300i* de 93.75 *MHz* [7].

PlayStation Console de videogame criado em 1994 pela empresa *Sony* tinha em sua CPU um processador de 32-bit *MIPS R3000A* de 33.8 *MHz* [8].

Sonda *New Horizon* Sonda criada pela *NASA* a fim de explorar o planeta anão Plutão, tem em seu CPU um modelo baseado no processador *MIPS R3000* chamado de *Mongoose-V* de 12*MHz*. Foi utilizado para fazer com que a sonda comunique-se com a Terra, corrigir problemas de contato com os operadores na Terra entre outros usos [9].

Atualmente, processadores *MIPS* são empregados em aplicações embarcadas que necessitam de processadores com baixo dispêndio de energia.

CAPÍTULO 3

CMIPS

cMIPS é um modelo escrito em *VHDL* do processador *MIPS32r2*. Este modelo implementa todo o conjunto de instruções e segue o design que é proposto no livro *"Computer Organisation and Design"*. Todas as definições apresentadas no cápitulo 2 aplicam-se a este modelo. As definições de conjunto de instruções, os seus formatos, o pipeline do *MIPS*, são reproduzidos em VHDL no *cMIPS* [10].

O *cMIPS* foi escrito com o objetivo de auxiliar no ensino de arquitetura de computadores, criando assim um modelo em que os alunos possam estudar programação em um nível mais "próximo ao *Hardware*" [10].

A figura 3.1 descreve a implementação do *TesteBench* do *cMIPS*, possuindo cinco periféricos e mais os componentes *RAM* e *ROM*. Os periféricos utilizados são: Um *STDOUT* para exibir resultados das simulações; um utilizado para a leitura de um arquivo de entrada, e outro para a escrita em um arquivo de saída; um contador utilizado para geração de interrupção após um determinado número de ciclos do *clock*, e uma *UART* comunicando-se com uma *UART* remota, sendo estas não mostradas no diagrama [10].

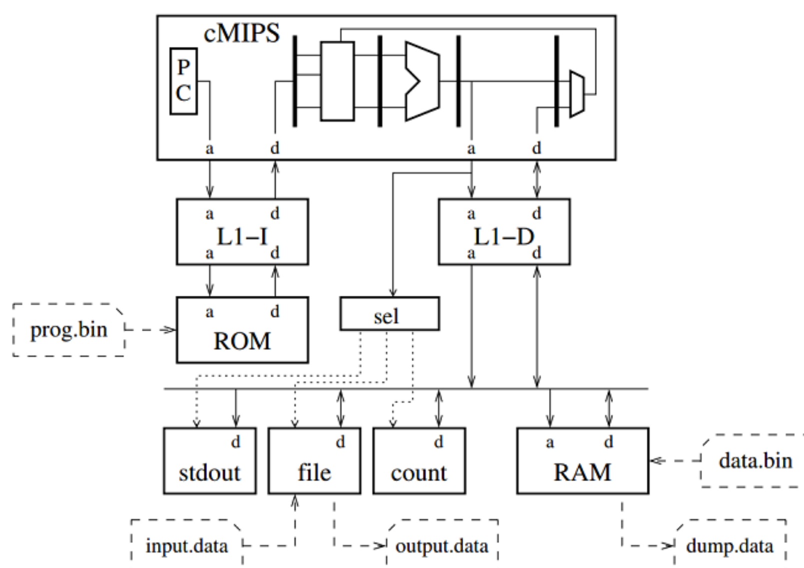


Figura 3.1: Diagrama demonstrando a implentação do cMIPS em um *testbench*.

Os arquivos que são necessários para simular os programas em **C** e **Assembly** são mostrados

no diagrama. O código do programa é carreagado na *ROM* através do arquivo *prog.bin* e a **RAM** pode ser inicializada pelo arquivo *data.bin*. A geração desses arquivos ocorre na compilação dos programas em **C** ou *Assembly*. O *cMIPS* pode ler dados contidos no arquivo *input.data* e escrever resultados binários no arquivo *output.data*. A **RAM** pode ser copiada para um arquivo de *dump* chamado *dump.data* [10].

CAPÍTULO 4

CLUPA

cLUPA é um ampliador de documentos impressos baseado no *xLupa* Embarcado e tem como objetivo ajudar a leitura de tais documentos por pessoas com necessidades especiais na área da visão [1].

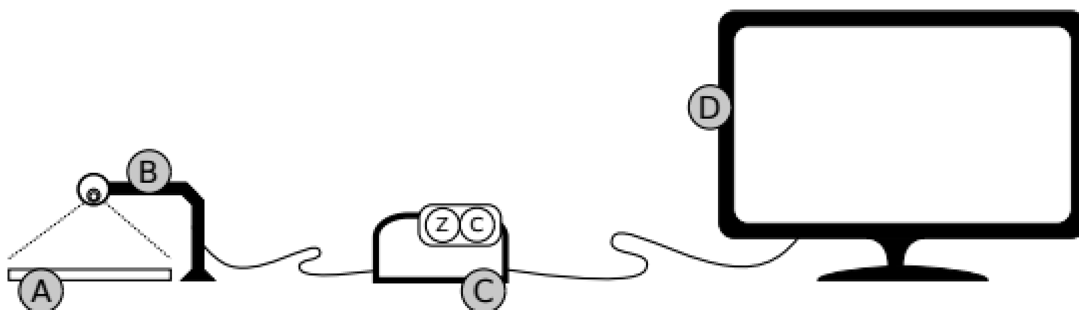


Figura 4.1: Visão geral do sistema CLupa.

A Figura 4.1 mostra quatro componentes que são a base do *cLUPA* [1]:

A: Documento que vai ser ampliado pelo *cLUPA*, o mesmo deve ser posicionado abaixo da câmera pelo usuário.

B: Braço de sustentação da câmera, esta realiza a leitura do documento.

C: Módulo de processamento em que a imagem é tratada. Possui dois botões para ampliação da imagem e alterar o contraste da mesma, estes são representados pelas letras Z e C. Este módulo se comunica com a câmera através de uma conexão *USB* e transmite a imagem para o monitor através de uma conexão *VGA*.

D: Monitor onde a imagem tratada pelo Componente C é apresentada.

Descrito o funcionamento geral do *cLUPA*, será descrito na seguinte subseção o seu *Hardware*, explanando seu funcionamento interno e detalhando o componente *Buffer* que é o objeto deste trabalho.

4.1 Hardware

O *hardware* do *cLUPA* é composto de três partes como mostra a Figura 4.2 [1]: (a) Câmera que filma o documento, (b) Módulo de processamento da plataforma *cLUPA* e (c) *Display* de Vídeo

que mostra o resultado do processamento.

O módulo de processamento recebe a imagem que é enviada pela câmera, então executa o processamento de *Zoom* e/ou Contraste, de acordo com a requisição do usuário, a qual é feita nos botões Z e C e, envia a imagem para o *display* de vídeo.

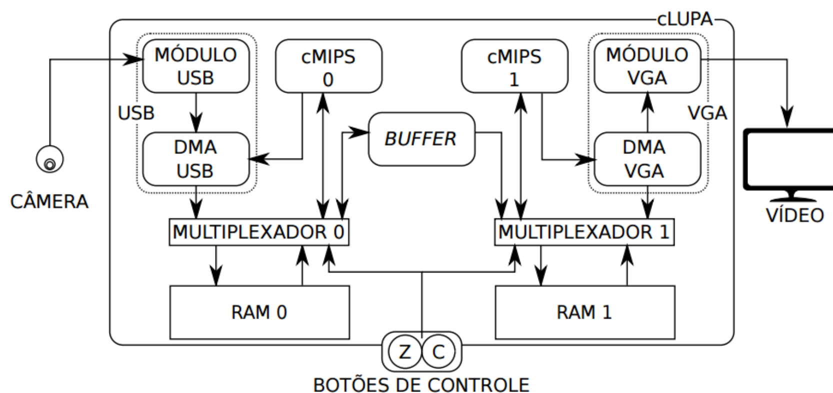


Figura 4.2: Hardware do cLUPA.

O módulo de processamento da imagem tem seu funcionamento interno listado abaixo [1]:

1. É feita a leitura da imagem através do componente Câmera.
2. A imagem é então transmitida para o Módulo *USB*.
3. O componente *DMA-USB* lê 4 *bytes* do Módulo, que representam um pixel.
4. Os dados do componente *USB* passam pelo Multiplexador 0 e são escritos na RAM 0.
5. O núcleo *CMips-0* lê um conjunto de *píxeis*.
6. O *CMips-0* aplica então o contraste a cada um dos *píxeis* lidos.
7. Após esse processamento os *píxeis* são armazenados no *BUFFER*.
8. O núcleo *CMips-1* lê cada um desses *píxeis* e aplica a ampliação, armazenando o resultado na RAM-1
9. O componente *VGA* lê esse conjunto de *píxeis* que estão na RAM 1 e o transmite para o vídeo.

4.2 Buffer

O *Buffer* de Comunicação Direta (BCD), mostrado na Figura 4.3, é o componente que faz a comunicação entre os núcleos *cMIPS-0* e *cMIPS-1.* Seu uso se dá pela transmissão de um pixel processado no núcleo 0 para o uso do mesmo no núcleo 1. Este pixel é codificado em 32 *bits* ou um inteiro.

O *Buffer* se comporta como um periférico que está ligado no MULTIPLEXADOR-0 e MULTIPLEXADOR-1 e desta forma liga-se aos núcleos *cMIPS* como demonstrado na figura 4.3 [1]. Cada um dos núcleos realiza uma tarefa especifica com o *buffer*: o núcleo *cMIPS-0* realiza a escrita de palavras ou leitura de estados, e o núcleo *cMIPS-1* realiza a leitura de palavras ou leitura de estados [1].

A Figura 4.3 mostra os componentes do *Buffer* estes são descritos abaixo.

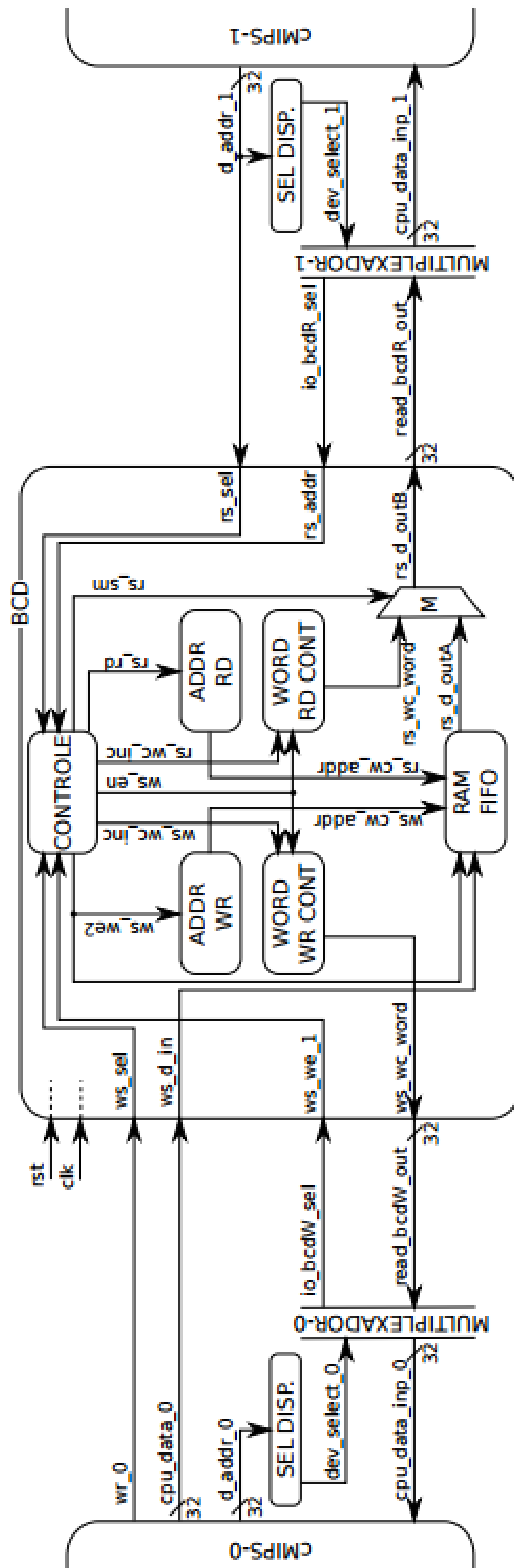


Figura 4.3: *Buffer* de comunicação direta.

O componente *RAM-FIFO* corresponde à memória com comportamento *FIFO (First-In-First-Out)* e usa dois sinais para controle de gravação e outro de leitura, correspondendo respectivamente aos sinas *ws_d_in* e *rs_d_out_A*. Também possui mais dois sinais de endereçamento usados para obter o endereço da palavra escrita ou lida. Estes sinais correspondem respectivamente aos sinais *ws_cw_addr* e *rs_cw_addr*. E por fim possui sinais de controle para indicar quando uma gravação ou leitura devem ocorrer, indicados respectivamente pelos sinais *ws_we2* e *rs_rd* [1].

O *Buffer* possui os componentes *ADDR-WR* e *ADDR-RD* que servem respectivamente como contadores cíclicos de endereços para gravação e leitura, e estes se ligam à *RAM-FIFO* através dos sinais *ws_cw_addr* e *rs_cw_addr*, e são controlados pelos sinais *ws_we2* e *rs_rd* os quais definem quando devem ser incrementado os endereços. O endereço de leitura sempre deve corresponder a um endereço anterior ao de gravação, e ser também menor que um endereço de gravação. Não há no hardware um controle para identificar se os endereços estão corretos; este controle deve ser feito pelo programador, que deve gerenciar a leitura e gravação utilizado para esse gerenciamento a leitura de estado do *buffer*.

Além dos componentes acima descritos, o *Buffer* possui dois componentes que são utilizados para o gerenciamento de leituras e escritas, que respectivamente correspondem ao componente *WORD-WR-CONT* e *WORD-RD-CONT*. O componente *WORD-WR-CONT* é um contador que armazena o número de palavras disponíveis na *RAM-FIFO* que ainda podem ser gravadas, e é incrementado cada vez que uma palavra é lida, ou decrementado cada vez que uma palavra é gravada. Quanto ao componente *WORD-RD-CONT*, este funciona também como um contador, porém destinado a leitura: ele armazena o número de palavras que estão gravadas na *RAM-FIFO* e que podem ser lidas, sendo incrementado quando uma gravação é realizada, e decrementado quando uma leitura ocorre. O componente *WORD-RD-CONT* é concorrente com a *RAM-FIFO* para acesso ao componente *MULTIPLEXADOR-1*. Suas saídas de dados são selecionadas pelo componente multiplexador M.

Outro componente do *Buffer* é o *CONTROLE*. Este lê os sinais que controlam a ativação da leitura e escrita representados pelos sinais *w_sel* e *r_sel*, os sinais de endereçamento representado pelo sinal r_addr e o sinal de gravação do *Buffer* representado por *w_we*, todos estes sinais se originam dos núcleos. O *Buffer* seleciona os sinais de controle de gravação da *RAM-FIFO* representado pelo sinal *ws_we2*, os sinais de incremento dos componentes de endereçamento representados pelos sinais *ws_we2* e *rs_rd*, os sinais de incremento ou decremento dos geren-

ciadores de estados, representados pelos sinais *wc_en*, *ws_cw_inc* e *rs_cw_inc*, e o sinal de seleção de saída, representado pelo sinal *rs_sm*, que é utilizado no multiplexador M, conforme a operação de escrita e/ou leitura no *Buffer*.

As seguintes subseções mostram como é feito o gerenciamento de leitura e escrita nos núcleos.

4.2.1 Verificação da disponibilidade do *Buffer*

No programa 4.1, a variável *bcd_max_aux* recebe um valor que indica se o *Buffer* possui espaço para a escrita através da função *bcdWst()*, que consulta o componente *WORD-WR-CONT*. Se o retorno dessa função for igual a zero, o programa permanece em um *loop* aguardando o momento da escrita. Para verificar o espaço do *buffer* para leitura o procedimento é igual ao do programa 4.1: deve-se apenas trocar a função *bcdWst()* pela função *bcdRst()*, essa função consulta o componente *WORD-RD-CONT* esperando o mesmo retornar um valor maior que 0 que significa que há algo a ser lido do *buffer*.

Programa 4.1: Verificação de espaço para escrita no *Buffer*.

```
do{
    bcd_max_aux = bcdWSt();
}while(bcd_max_aux<=0);
```

O programa 4.2 mostra como a escrita no *Buffer* deve ser feita em C. A função *bcdWWr* recebe um inteiro como parâmetro, este então é escrito na *RAM-FIFO* através do sinal *ws_d_in* e após a leitura, os componentes *ADDR-WR*, *ADDR-RD*, *WORD WR-CONT* e *WORD RD-CONT* serão atualizados. O programa 4.2 deve sempre ser executado depois do programa 4.1, pois a escrita deve esperar a existência de espaço para escrever no *Buffer*.

4.3 Escrita no *Buffer*

Programa 4.2: Escrita de dados no *Buffer*.

```
bcdWWr(valor_a_ser_escrito);
```

Através do programa 7.1 é mostrado como a leitura do *buffer* é feita. Para esta ser realizada, deve-se chamar a função *bcdRRd()*, esta consulta o componente *RAM-FIFO* o qual responde com o resultado no sinal *rs_d_outA*. Após a leitura, os componentes são atualizados. O programa 7.1

segue o princípio do programa 4.2 e deve ser executado após o código 4.1 a fim de esperar que existam valores a serem lidos do *Buffer*.

4.4 Leitura do *Buffer*

Programa 4.3: Leitura de valor do *Buffer*.

```
valor_lido = bcdRRd();
```

CAPÍTULO 5

AVALIAÇÃO DE DESEMPENHO DO *BUFFER* DO CLUPA.

Para a avaliação do desempenho do *Buffer* do cLUPA foram realizados simulações de carga com diferentes tamanhos de *Buffer*, que estão descritos na seção 5.1. A métrica utilizada se encontra na seção 5.3 e os resultados na seção 5.6.

Nos gráficos mostrados neste capítulo, o eixo X representa a sequência das operações, e o eixo Y o tempo em que cada uma destas sequências levou para ser escrita ou lida. O núcleo *CMips-0* é chamado de *produtor* dado que este gera os dados que são escritos no *Buffer*, e o núcleo *CMips-1* é chamado de *consumidor*.

5.1 Cargas de simulação

Para avaliar o desempenho do *Buffer* foram realizadas simulações para tamanhos variados de *Buffer*. Para essas simulações foram utilizadas as funções de *Fibonacci* e a função de Crivo de Eratóstenes para computar números primos, a qual será chamada de "primo". A função de *Fibonacci*, executada de forma iterativa, é usada para simular uma alta taxa de leitura e/ou escrita por parte do produtor e/ou consumidor. A função de primo é usada para simular o comportamento do *Buffer* quando a taxa de leituras e/ou escritas é baixa por parte do consumidor e/ou produtor.

Com relação às simulações, foram usadas 5 combinações a fim de simular as possibilidades de carga e a descrição de cada simulação se encontra na tabela 5.1.

Para cada simulação foram utilizados tamanhos de *Buffer* que variam de 2^1 a 2^8. Tais tamanhos foram determinados por representarem, de forma significativa, o comportamento do *Buffer*, já que tamanhos maiores que 2^8 representam um resultado igual entre eles para os testes que foram executados. Também foi utilizado para as simulações um tempo máximo de execução, que é de 1500 microsegundos (tempo simulado) deste caso, uma simulação demora 46 minutos e 2.090 segundos em um computador *Acer Intel Core* I5 de 2.50 GHz.

Simulações	Núcleo 0	Núcleo 1	Objetivo
Simulação 1	Gera um número de *Fibonacci.*	Verifica se o número resultante de Fibonacci no no núcleo *cMips-0* é primo.	Verifica o comportamento do *Buffer* no caso em que o produtor tem uma taxa elevada de escritas na fila e o consumidor uma baixa taxa de leituras.
Simulação 2	Procura um número primo.	Processa Fibonacci do número primo recebido do núcleo *cMips-0.*	Verifica o comportamento do *Buffer* no caso em que o produtor tem uma taxa de escritas na fila muito baixa e o consumidor uma taxa elevada de leituras.
Simulação 3	Gera um número de Fibonacci.	Gera um número de Fibonacci assim que um resultado é recebido do núcleo *cMips-0.*	Verifica o caso em que tanto produtor quanto consumidor provocam alta taxa de utilização do *Buffer.*
Simulação 4	Procura um número primo.	Verifica se o número recebido do núcleo *cMips-0* é primo	Verifica o caso em que tanto produtor quanto consumidor provocam baixa taxa de utilização do *Buffer.*
Simulação 5	Gera um valor aleatório e esse valor é gerado em intervalos também aleatórios	Espera um tempo aleatório para retirar um elemento do *buffer*	Observa a taxa de utilização do *Buffer* para comportamento aleatório de produtor e de consumidor.

Tabela 5.1: Cargas de simulação.

5.2 Programa de simulação

Cada simulação foi executada utilizando-se de um *script* de simulação, este executa as simulações com a seguinte linha de comando:

Programa 5.1: Descrição da linha de execução do *Script*.

```
>./teste.sh simulacao_inicial simulacao_final
            tamanho_inicial_BCD tamanho_final_BCD
```

Os parâmetros passados são respectivamente, simulação inicial, simulação final, tamanho inicial do *Buffer* e tamanho final do *Buffer*: de qual simulação deve-se iniciar até qual simulação deve-se executar e qual o tamanho inicial do *Buffer* até o tamanho máximo do *Buffer* deve ter para executar. O exemplo a seguir mostra uma linha de execução possível do *script*:

Programa 5.2: Exemplo de linha de execução do *Script*.

```
>./teste.sh 1 4 1 256
```

O programa 5.2 executa as simulações de 1 a 4, conforme as definições da tabela 5.1, e cada uma dessas simulações será executada com um *Buffer* de tamanho 1 até tamanho 256.

Definidos os parâmetros, a simulação executa da seguinte forma:

1. A partir da simulação inicial definida nos parâmetros inicia um *loop* até a última simulação definida.
2. Cada simulação executa um tamanho de fila do *Buffer* indo do tamanho inicial até o tamanho máximo definido.
3. Em cada simulação, o tempo simulado é limitado pelo número de primos/*Fibonacci* gerados; sendo gerados no máximo 312 números na sequência de *Fibonnaci* e 48 números primos, porque estes são os valores máximos representáveis em 32 bits.

O apêndice 7.1.1 contém os detalhes sobre o *script teste.sh*, com a sua implementação e estruturas de pastas.

5.3 Métricas das simulações

Para avaliar o comportamento do *buffer* foi utilizada a métrica utilização baseada na teoria das filas. Primeiramente são definidas algumas quantidades: A_i que representa a quantidade de escritas no *Buffer*, C_i que representa a quantidade de leituras.

A métrica observada é a utilização U do *Buffer* em relação a leitura de dados. esta métrica é dada pela equação:

$$U_i = \frac{C_i}{A_i} \cdot 100 \tag{5.1}$$

Com a utilização é possível verificar a taxa de leitura do *Buffer* em relação ao tamanho da fila, e pode-se dizer se o gerador de dados está ficando muito tempo ocioso em relação às leituras isto é, quanto menor a utilização mais tempo os dados da leitura ficam enfileirados no *Buffer*.

Nas seguintes seções são descritos os problemas iniciais para executar as simulações e como estes foram resolvidos. Breves descrições sobre o comportamento da função de Fibonacci e função dos números primos e os resultados das simulações são também apresentados.

5.4 Problemas iniciais

Ao se iniciar as simulações, com a simulação 1, foi notada a existência de um *bug* na leitura do *Buffer*. Esse ocorria sempre que a fila do *buffer* estava com tamanho 8. Tal fato ocorria quando a fila do *Buffer* estava com mais de quatro elementos gravados, quando então os contadores de endereço da leitura pulavam, lendo valores à frente daqueles que deveriam ser lidos.

Ao se fazer uma análise sobre a simulação, com o *GTKwave*, descobriu-se que a definição do contador com tamanho 8 estava errada: o contador que deveria ir do index 0 até o index 7 e então voltar a 0, estava indo de index 0 a index 3, causando o pulo na leitura dos dados.

5.5 Geradores de números da sequência de ***Fibonacci*** e primos

Antes de descrever os resultados das simulações deve-se fazer algumas considerações quanto às propriedades das funções de *Fibonacci* e primos. Para tais considerações será utilizado primeiramente o gráfico 5.1, que mostra as sequências geradas.

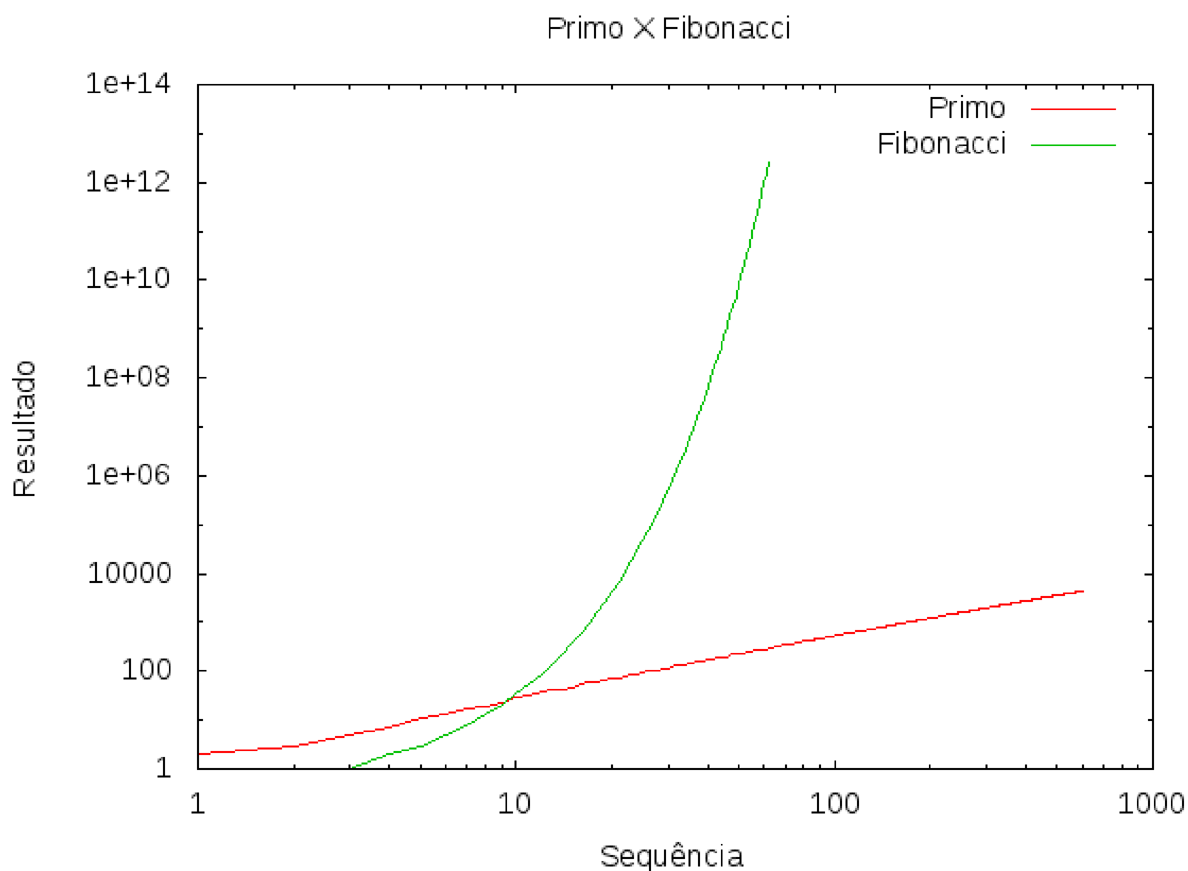


Figura 5.1: Números primos e série de Fibonacci(a escala do gráfico é logarítmica).

Na Figura 5.1 nota-se que a função de *Fibonacci* tende a crescer mais rápido que a função primo, fato esse que ocorre pois a função de *Fibonacci* é uma sucessão de somas, diferente dos números primos que não apresentam um padrão que cresce de maneira uniforme.

No Figura 5.2 é apresentado o comportamento das funções de *Fibonacci* e *Primo* no *cLUPA*, lembrando que para essa execução o *Buffer* não foi utilizado e por isso o resultado apresentado refere-se apenas ao comportamento desta função no *cMIPS*. Para a simulação foi usado o tempo de simulação de 500 ns e executada no núcleo *cMips-0*.

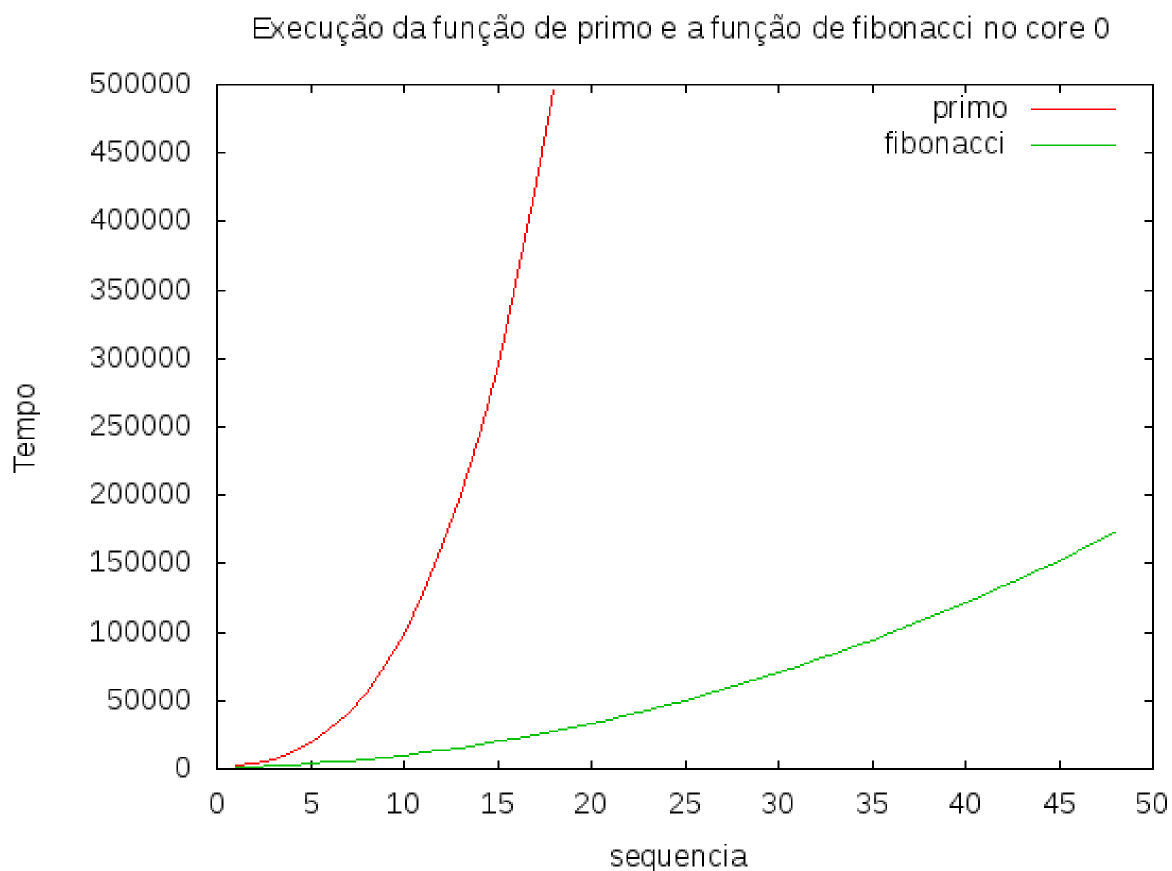


Figura 5.2: Tempo para gerar séries de primos e fibonacci em execução no núcleo *cMips-0*.

Nota-se que o tempo para processar os números de *Fibonacci* é muito menor que para processar os primos, isso ocorre pelo fato de que na função usada para descobrir se um número é primo ocorrem sucessivas divisões e comparações ocasionando assim um tempo maior de processamento.

Por outro lado, a função de *Fibonacci* possui um tempo de execução menor por utilizar-se apenas de somas, e não necessita de operações mais complexas como multiplicações e divisões.

Para verificar se o comportamento do *Buffer* é o esperado, ou seja escrevendo e lendo os valores normalmente, foi executado um programa com um tempo simulado de 40.000 ciclos em que cada ciclo dura 20ns. Este programa gera um número aleatório no core *cMips-0* e este é consumido pelo core *cMips-1* assim que é escrito no *Buffer*. O resultado é mostrado na figura 5.3, para um *Buffer* com capacidade para 256 inteiros.

Figura 5.3: Teste de corretude do *Buffer*.

A Figura 5.3 mostra que o *Buffer* funciona de acordo com o esperado, pois as linhas de *write* e *read* permanecem sobrepostas. Qualquer anomalia que o **BCD** viesse a ter, o gráfico mostraria através da separação das linhas.

5.6 Resultados

Os resultados das simulações são apresentados a seguir.

5.6.1 Simulação 1

$$\textit{fibonacci} \rightarrow \textit{é primo ?}$$

A simulação 1 está descrita na tabela 5.1 e com o resultado da execução foram montados gráficos correspondentes a cada um dos tamanhos do *Buffer* investigados: 1, 2, 4, 8, 16, 32, 64, 128 e 256. O tempo simulado é de 1500 μs, dado que este tempo é o suficiente para obter-se uma noção do que ocorre no *Buffer* com este tipo de simulação. Cada simulação está separada por tamanho da fila nas seguintes subseções.

5.6.1.1 Tamanho de fila 1

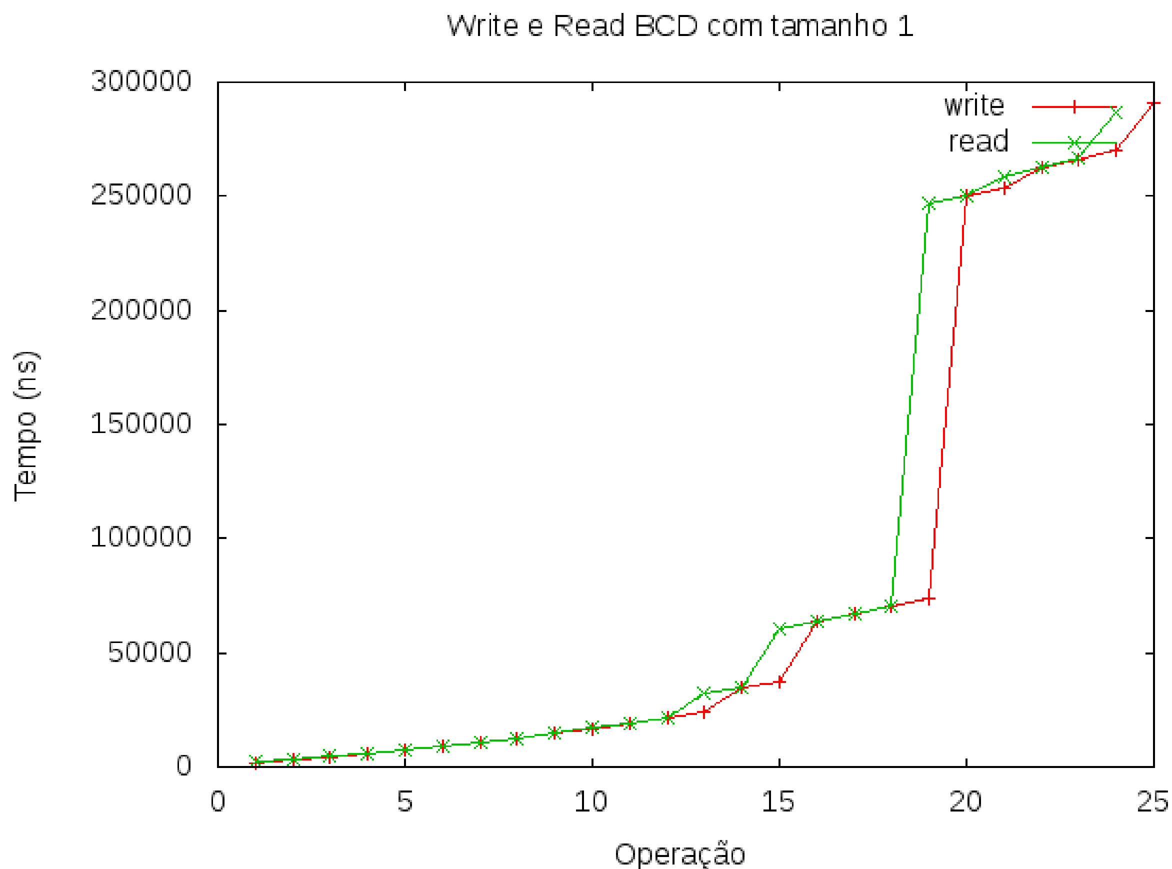


Figura 5.4: *Buffer* com fila de tamanho 1.

A Figura 5.4 mostra o comportamento do *Buffer* com apenas um espaço de armazenamento. Nota-se uma proximidade entre as linhas de *writes* e *reads*. Isso ocorre pois o produtor só escreve no *Buffer* assim que o consumidor ler deste. Dado que o tamanho do *Buffer* é de 1 elemento, assim que o produtor escreve, ele entra em estado de espera até que o elemento seja retirado do *Buffer*, com isso a linha de *write* tende a ser sempre próxima da linha de *read*.

Quando o consumidor leva um tempo maior para ler algum elemento na fila, começa então a haver uma separação entre as linhas de *writes* e *reads*, porém dado o tamanho de 1 elemento do *Buffer* as operações ficam atrasadas em apenas 1 operação.

Utilizado-se da métrica de utilização do *Buffer*, consta-se que com tamanho de 1 elemento para o *Buffer*, obtem-se uma taxa de **96%**, pois foram feitas 25 escritas e 24 leituras, esse número ocorre pois dado que o consumidor toma um tempo muito grande para processar alguns dos valores gerados pelo produtor, a simulação é parada devido ao limite imposto no tempo simulado de 1500 μs.

5.6.1.2 Tamanho de fila 2

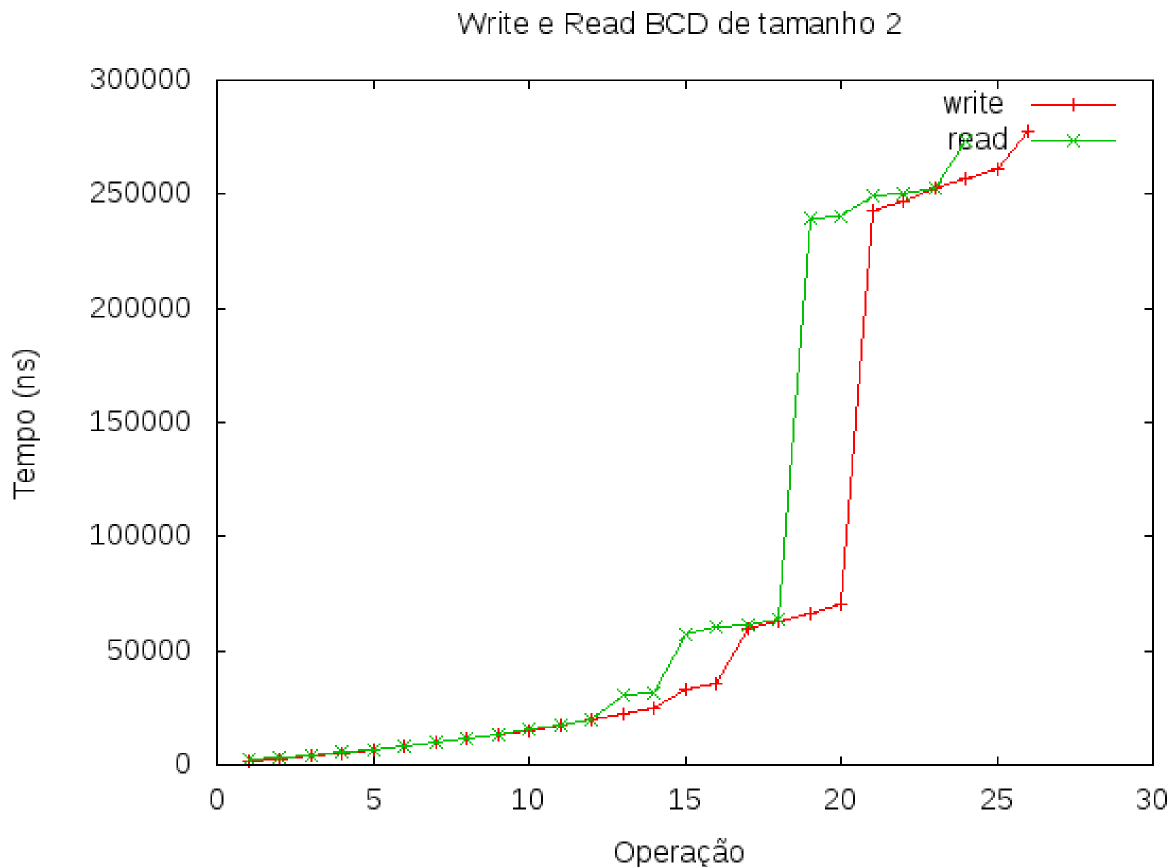


Figura 5.5: *Buffer* com fila de tamanho 2.

A Figura 5.5 mostra o relacionamento entre *writes* e *reads* no *Buffer*. Nota-se que em dadas operações existe uma diferença entre o *write* da operação e o seu respectivo *read*. Isso ocorre por que nessas operações a função de *Fibonacci* começa a produzir valores muito altos como mostrado na figura 5.1, com isso a função primo passa a levar um tempo maior para verificar se esses valores são ou não primos, resultando então na separação das linhas, como por exemplo na operação 13. A separação das linhas nesta figura mostra que o produtor preenche o *Buffer* com dois elementos e assim que o consumidor lê um valor o produtor logo adiciona um novo elemento na fila, fazendo com que essa fique, a partir da operação 13, constantemente cheia até que o produtor pare de gerar novos valores complexos e volte a gerar valores simples para o processamento do consumidor e, desse momento em diante (operação 20), as linhas tendem a se encontrar. Os saltos que ocorrem na linha de *writes* ocorre pelo fato que o produtor deve esperar o consumidor ler um valor do *buffer* que está cheio no momento e nesse período o produtor fica em espera sem gerar novos valores.

Utilizado a métrica previamente definidas obtém-se que a utilização do *Buffer* foi de **92%** já que tivemos 26 escritas contra 24 leituras, esses números foram previamente explicados na Sub subseção 5.6.1.1 e o *Buffer* possui dois elementos na sua fila.

5.6.1.3 Tamanho de fila 4

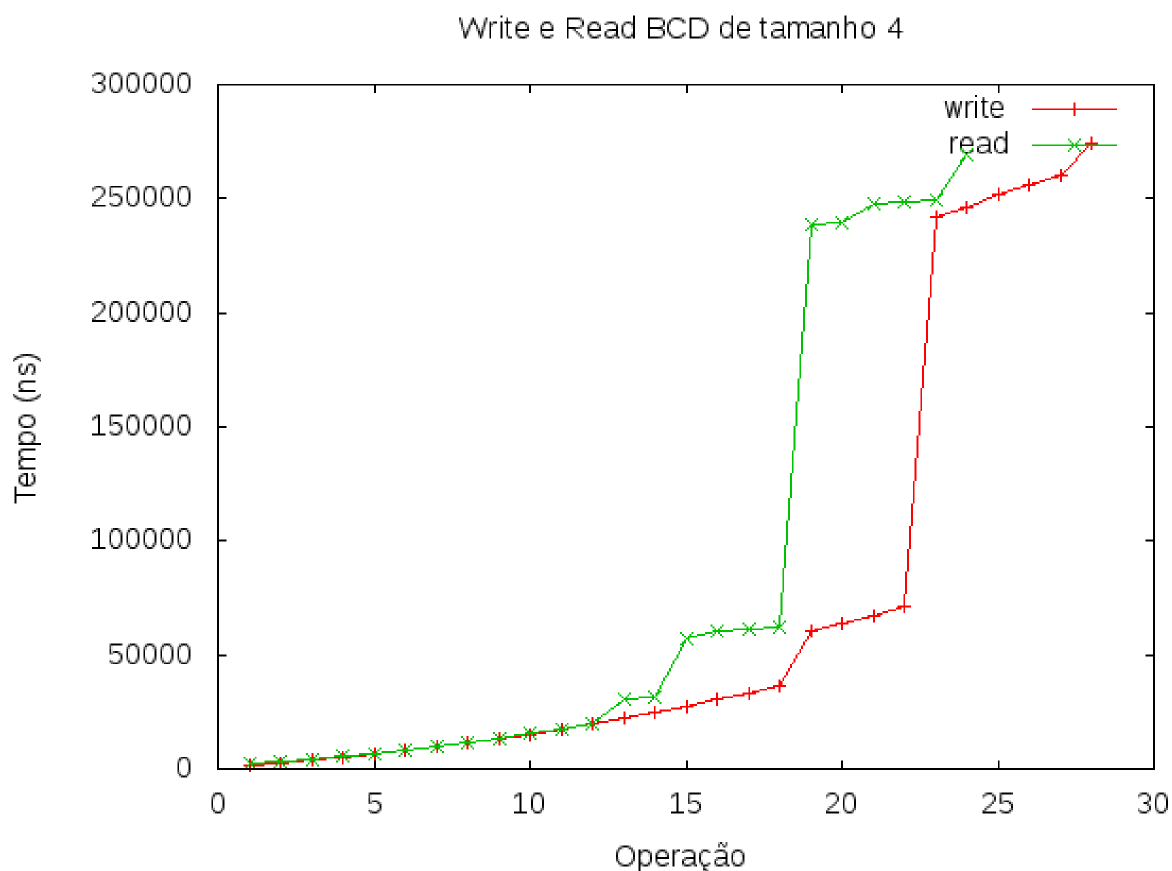


Figura 5.6: *Buffer* com fila de tamanho 4.

A Figura 5.6 mostra o quanto o tamanho acaba por influenciar a separação das linhas de *writes* e *reads*. As linhas não voltam a se encontrar enquanto o produtor continuar a gerar novos valores, e estes não se encontram pois quando ocorre a separação, iniciada a partir do valor gerado pelo produtor na operação 12, enquanto o consumidor está processando esse novo valor, o produtor continua a escrever no *Buffer*, até atingir o limite de 4 elementos e logo após atingir esse limite fica aguardando a retirada de apenas um elemento da fila para escrever um novo, consequentemente a fila tende a ficar sempre cheia, até que o produtor pare de gerar novos valores demorados e volte a gerar valores rápidos e assim então o consumidor poderá processar os valores contidos no *Buffer* de forma eficiente.

A taxa de utilização com esse tamanho foi de aproximadamente **86%**, pois tivemos 28 escritas contra apenas 24 leituras e com 4 elementos no *Buffer*.

5.6.1.4 Tamanho de fila 8

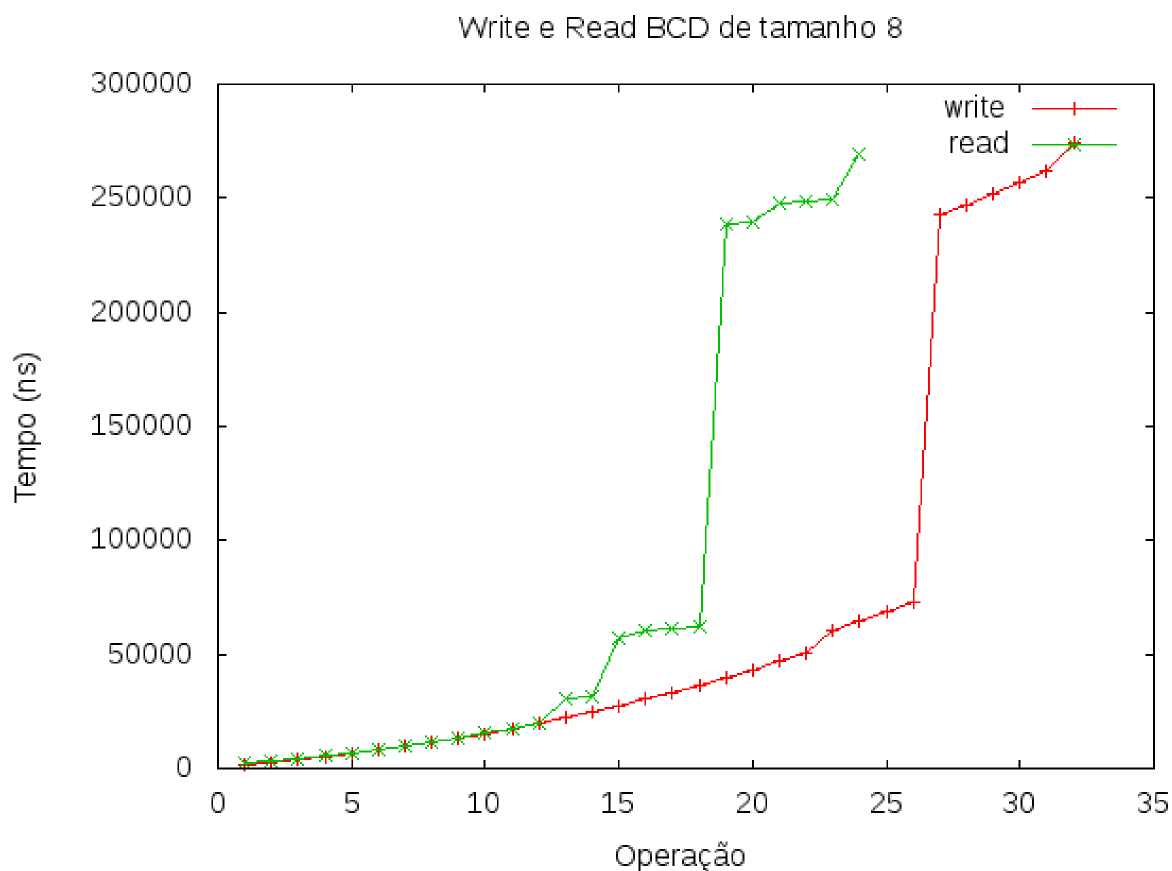


Figura 5.7: *Buffer* com fila de tamanho 8.

A Figura 5.7 mostra o que está explanado na subseção 5.6.1.3, em que devido ao fato de o consumidor levar um tempo maior para processar os dados, o produtor começa a preencher o *Buffer* até o seu limite, ocasionando assim a separação de linhas, e no caso da Figura 5.7, a separação começa a ficar grande, e portanto o encontro das mesmas acontece quando os valores gerados pelo produtor forem rápidos o suficiente para que o processamento do consumidor seja rápido, como os valores entre a operação 1 a 11 e isto não ocorre porque o tempo de simulação foi limitado em 1500 μs.

Na simulação com tamanho 8, obtém-se uma taxa de utilização de **75%**, pois foram feitas 32 escritas 24 leituras, sendo esses números de escritas e leituras já previamente explicados em subseções anteriores e com 8 elementos no *Buffer*.

5.6.1.5 Tamanho de fila 16

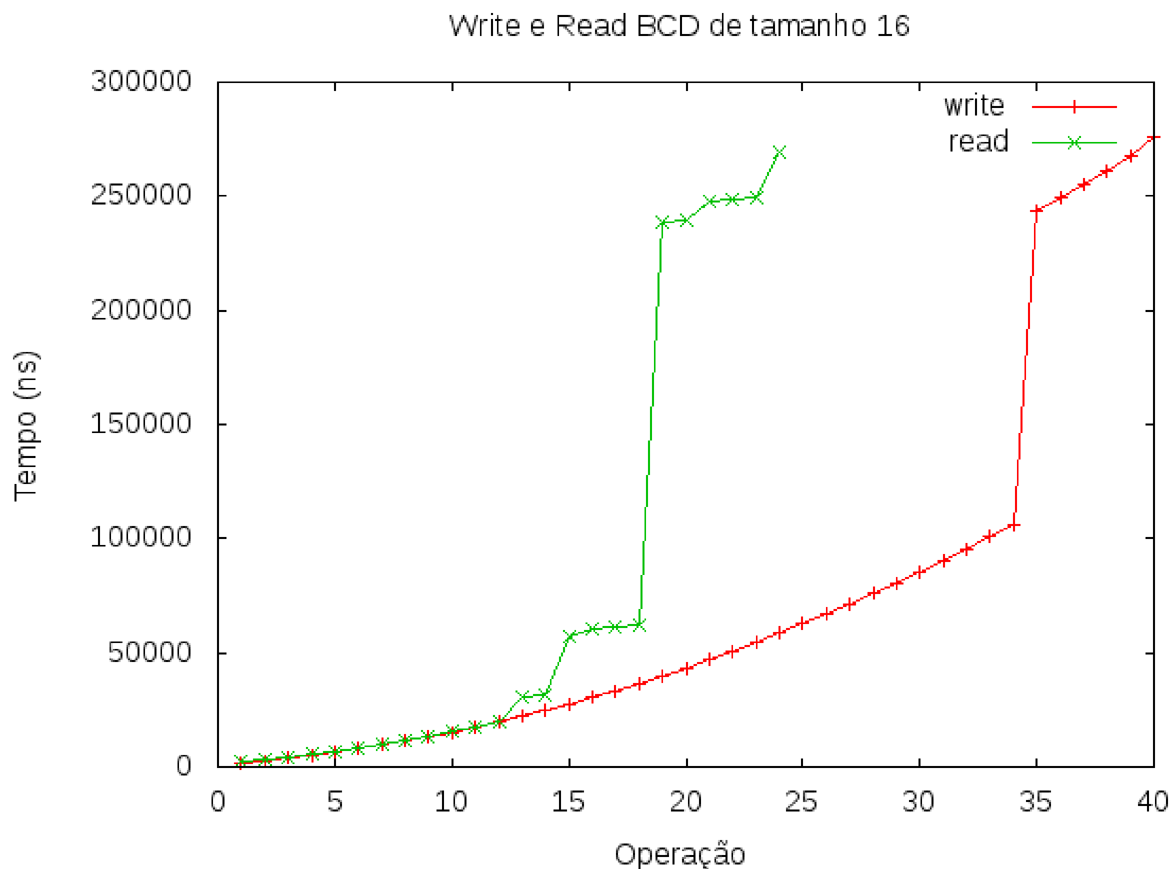


Figura 5.8: *Buffer* com fila de tamanho 16.

A Figura 5.8 mostra a mesma situação de quando o consumidor demora muito tempo para processar os seus dados e o produtor continua a gerar dados até o limite do *Buffer* e cada vez mais obtém-se uma separação maior entre as linhas *write* e *read* e nunca voltando a juntar as duas linhas de novo pois os períodos dos laços do *Fibonacci* e do *Primo* são diferentes; a fila garante então que eles fiquem dessincronizados até o final da execução.

Em relação a métrica a taxa de utilização do *Buffer* ficou em **60%** com 40 escritas e 24 leituras e tendo 16 elementos no *Buffer*.

5.6.1.6 Tamanho de fila 32

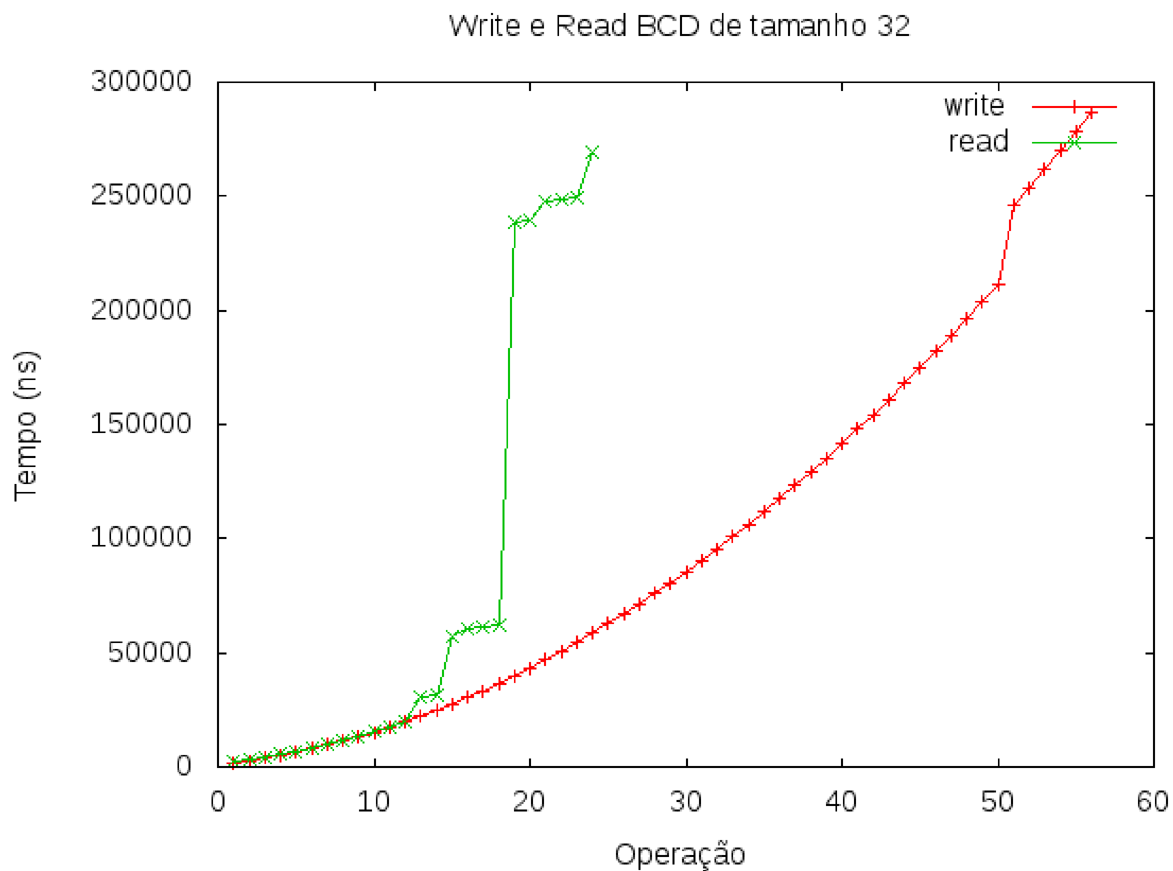


Figura 5.9: *Buffer* com fila de tamanho 32.

Continuando com o padrão descrito nas simulações anteriores, a distância entre *read* e *writes* apenas aumenta, e a cada aumento que há no tamanho do *Buffer* a linha *write* fica mais parecida com o gráfico de *Fibonacci* mostrado na Figura 5.1, no caso do gráfico 5.9 existe um único salto que ocorre na operação 50, isto ocorre pelo fato de não haver mais espaço no *Buffer* para escrita.

Verificando o gráfico com a métrica definida obtém-se que a taxa de utilização do *Buffer* ficou em aproximadamente **43%** sendo feitas 56 escritas e 24 leituras, e com o *Buffer* contendo 32 elementos em seu interior.

5.6.1.7 Tamanho de fila 64

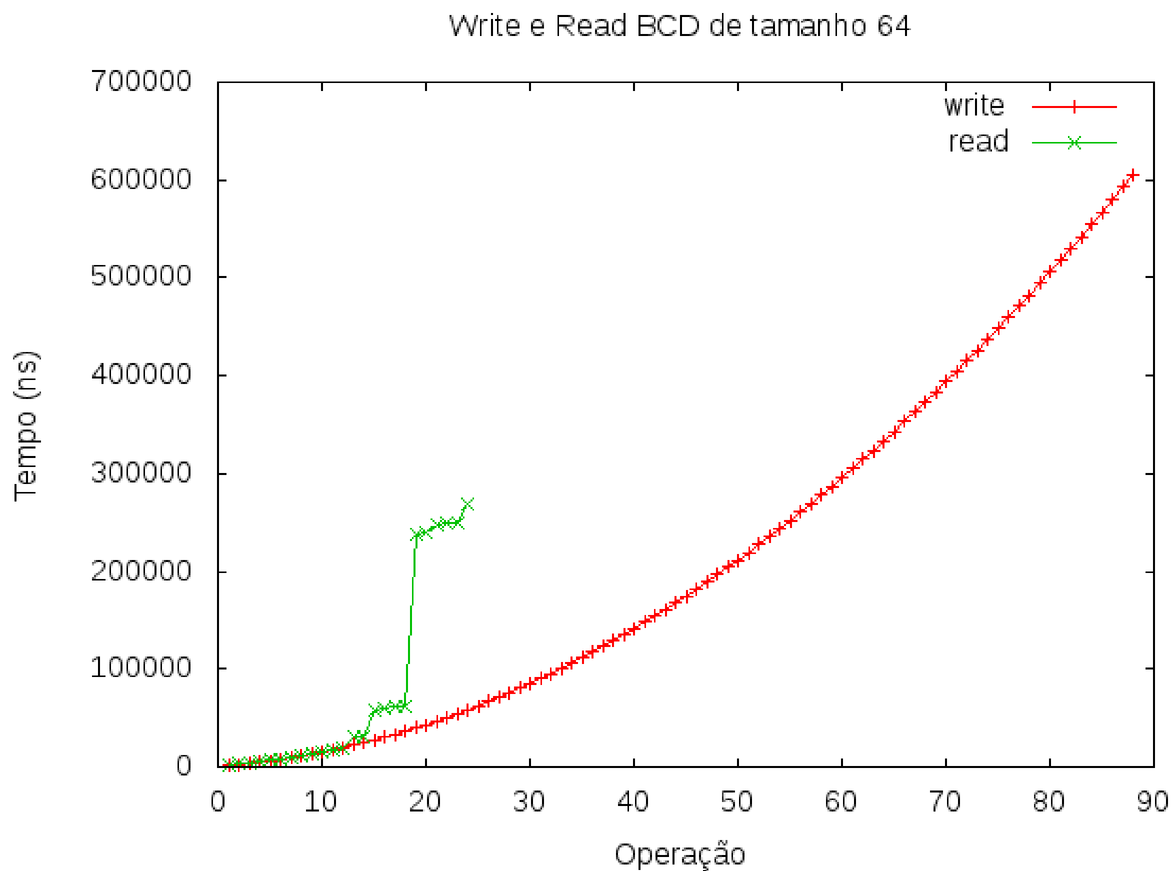


Figura 5.10: *Buffer* com fila de tamanho 64.

A partir do tamanho 64 o gráfico do *write* nota-se que a linha de *write* está semelhante a linha da sequência de *Fibonacci* mostrada na Figura 5.1 para o tempo simulado de 1500 μs e como demonstrado nas simulações anteriores a distancia entre *reads* e *writes* aumentou e estas linhas não voltarão a se encontrar como explicado na subseção 5.6.1.5.

Quanto as métricas, obteve uma taxa de utilização de aproximadamente **27%** com 88 escritas e 24 leituras e contendo 64 elementos no *Buffer*.

5.6.1.8 Tamanho de fila 128 a 256

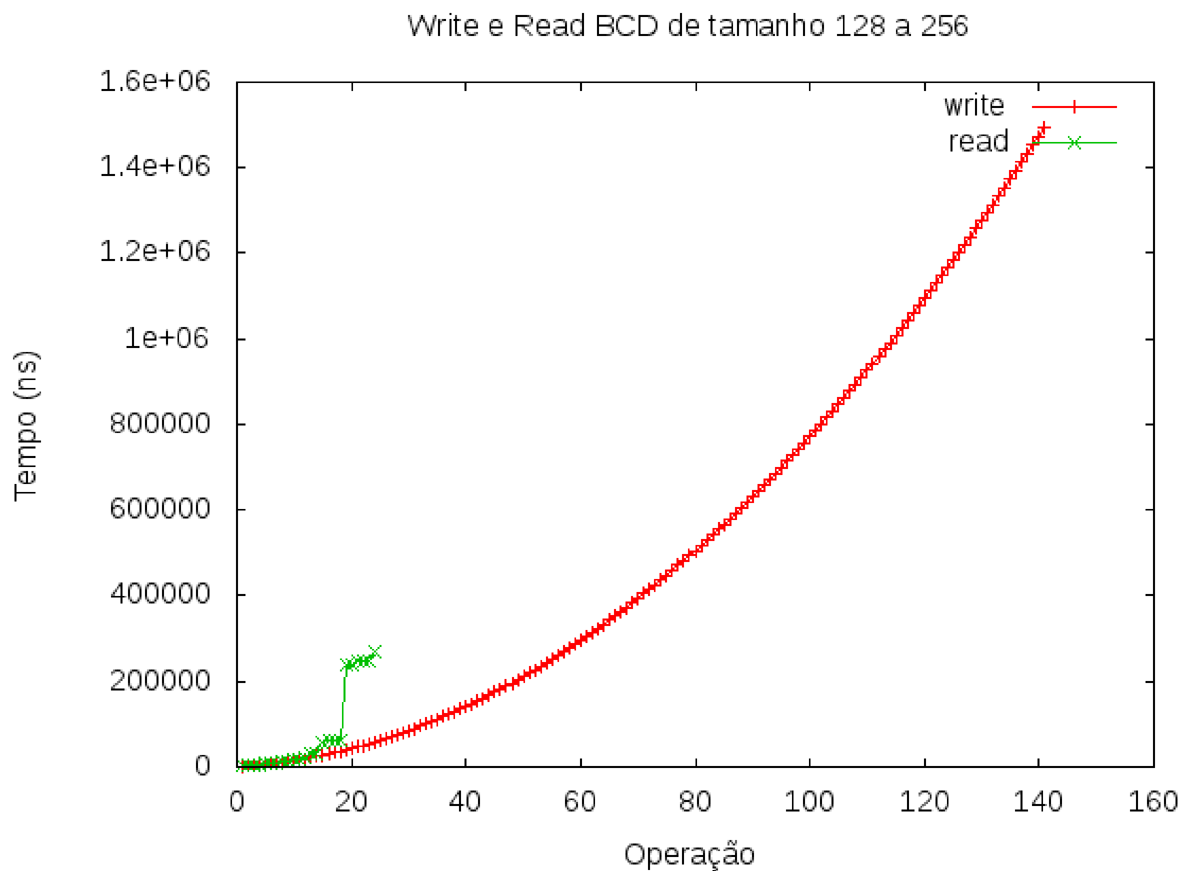


Figura 5.11: *Buffer* com fila de tamanho 128 à 256.

Os gráficos dos tamanhos 128 e 256 são idênticos, e este fato ocorre devido a limitação de tempo de simulação de 1500μs. Em relação a métrica, a taxa de utilização do *Buffer* para o tamanho 128 e 256 é de aproximadamente **21%** com 141 escritas no *Buffer* e 24 leituras e contendo 117 elementos no *Buffer*.

5.6.1.9 Análise dos resultados da simulação 1

Através dos resultados das simulações obtem-se o gráfico 5.12, que mostra a utilização do *Buffer*.

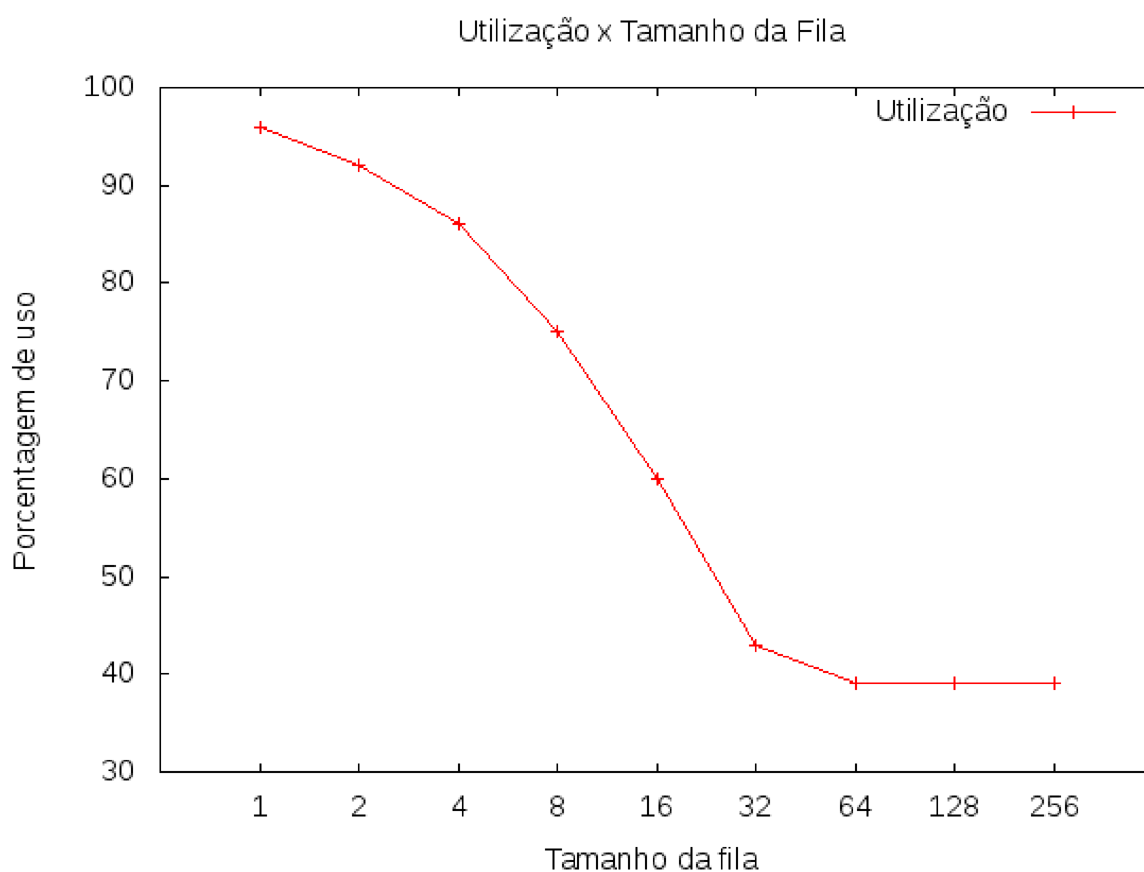


Figura 5.12: Uso do *Buffer* em porcentagem em relação com o tamanho da fila.

A figura 5.12 mostra que, quanto menor a fila melhor a utilização do *Buffer*, pois com filas pequenas obtem-se uma sincronização entre o produtor e o consumidor e entre filas grandes o produtor, que gera novos valores rapidamente, acaba por ter que esperar que o consumidor, que processa lentamente os valores gerados, ocasionando assim a baixa taxa de utilização. Com isso para a simulação 1, entende-se que filas de menores tamanhos devem ser utilizadas e observando os resultados obtidos através da métrica de utilização do *Buffer*, obtem-se que o tamanho 1, com 96% de taxa de utilização, é o melhor tamanho para este caso.

5.6.2 Simulação 2

$$primo \rightarrow fibonnaci\ do\ primo$$

A simulação 2 apresenta um gráfico em que as linhas se assemelham a linha de números primos mostrada na Figura 5.1, pois a mesma é a inversão de cargas da simulação 1, fazendo com que o núcleo *cMips-0* processe menos dados e o núcleo *cMips-1* consuma muitos dados. Os resultados são mostrados na imagem 5.13.

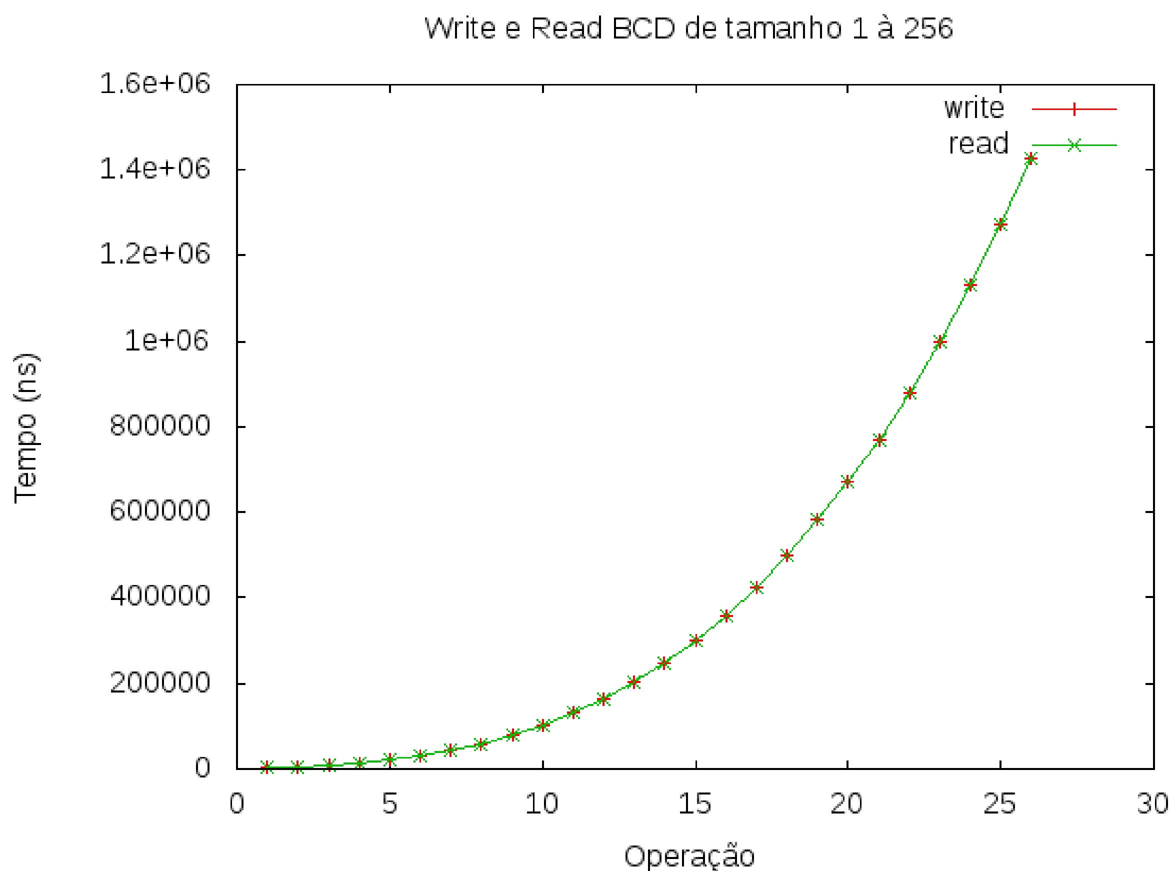


Figura 5.13: *Buffer* com fila de tamanho 1 a 256.

O gráfico 5.13 mostra que, independentemente do tamanho da fila, o número de *writes* e *reads* sempre são muito próximos. Isso ocorre pelo fato que, na simulação 2, a carga no produtor é muito alta, já que processar números primos no *cMIPS* toma mais tempo. Por outro lado o processamento no consumidor é baixo, sendo assim para cada valor computado que ocorre no produtor, a sua leitura ocorre quase que instantaneamente, como indica a Figura 5.13.

Quanto à utilização esta foi de **100%**, porém deve-se salientar que o consumidor passa a maior parte do tempo em espera ocupada, ocasionado uma ociosidade no consumidor, fazendo com que ele espere um longo tempo para poder ler algo vindo do produtor.

5.6.2.1 Análise dos resultados da simulação 2

Dado que todos os tamanhos atingiram a taxa de utilização de 100% devido ao fato que o produtor é mais lento que o consumidor para processar os seus dados, então pode-se dizer que qualquer tamanho para o *Buffer* serve, mas deve-se levar em conta que como os valores gerados pelo produtor são rapidamente processados pelo consumidor então para evitar um desperdício

de espaço em *Hardware* com uma fila muito grande em que apenas um elemento é guardado, opta-se por escolher o tamanho de fila 1 para o *Buffer*.

5.6.3 Simulação 3

fibonacci → *fibonnaci do fibonacci*

Esta simulação apresenta um gráfico de execução muito próximo ao gráfico 5.13, porém com sutis diferenças que serão explanadas a seguir. Primeiramente quanto a simulação, essa segue a descrição feita na subsessão 5.1: produtor e consumidor apresentam uma alta utilização do *Buffer* tanto para escrever quanto para ler e para mostrar os resultados dessa execução que estão divididos nas duas subseções a seguir.

5.6.3.1 Tamanho de fila 1

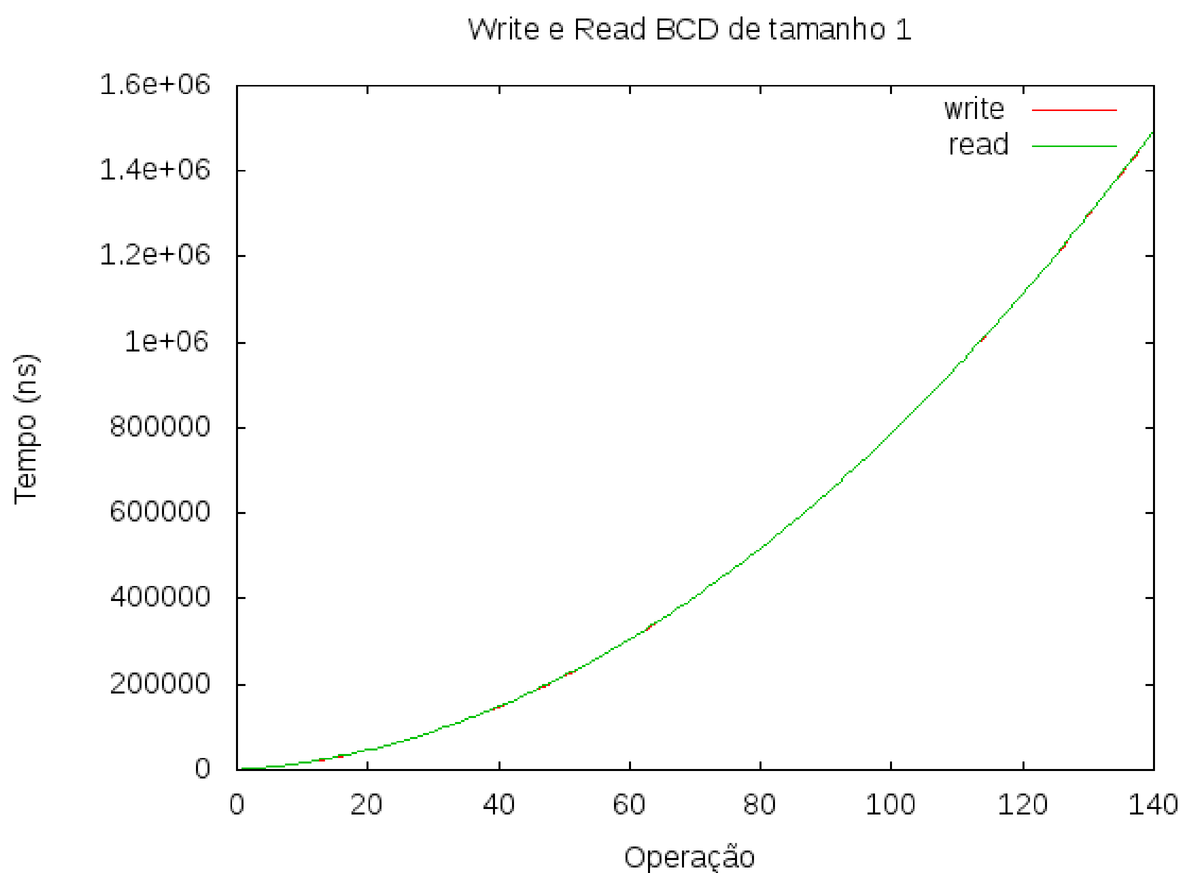


Figura 5.14: *Buffer* com fila de tamanho 1.

A Figura 5.14 mostra que o produtor é muito rápido para gerar seus valores e o consumidor é muito rápido para consumir os seus valores, porém como a operação entre eles é a de que o

consumidor deve gerar um *fibonacci* de um outro *fibonacci* já gerado pelo produtor, ocasionando assim uma demora por parte do consumidor para poder fazer uma leitura, já que esse utiliza-se de um tempo maior de processamento para gerar esse *fibonacci* do *fibonacci*. A separação entre as duas linhas tende aumentar até que o produtor volte a gerar valores de *fibonacci* menores.

Em relação a métrica, a taxa de utilização do *Buffer* foi de 100% sendo feitas 140 escritas e leituras.

5.6.3.2 Tamanho de fila 2 à 256

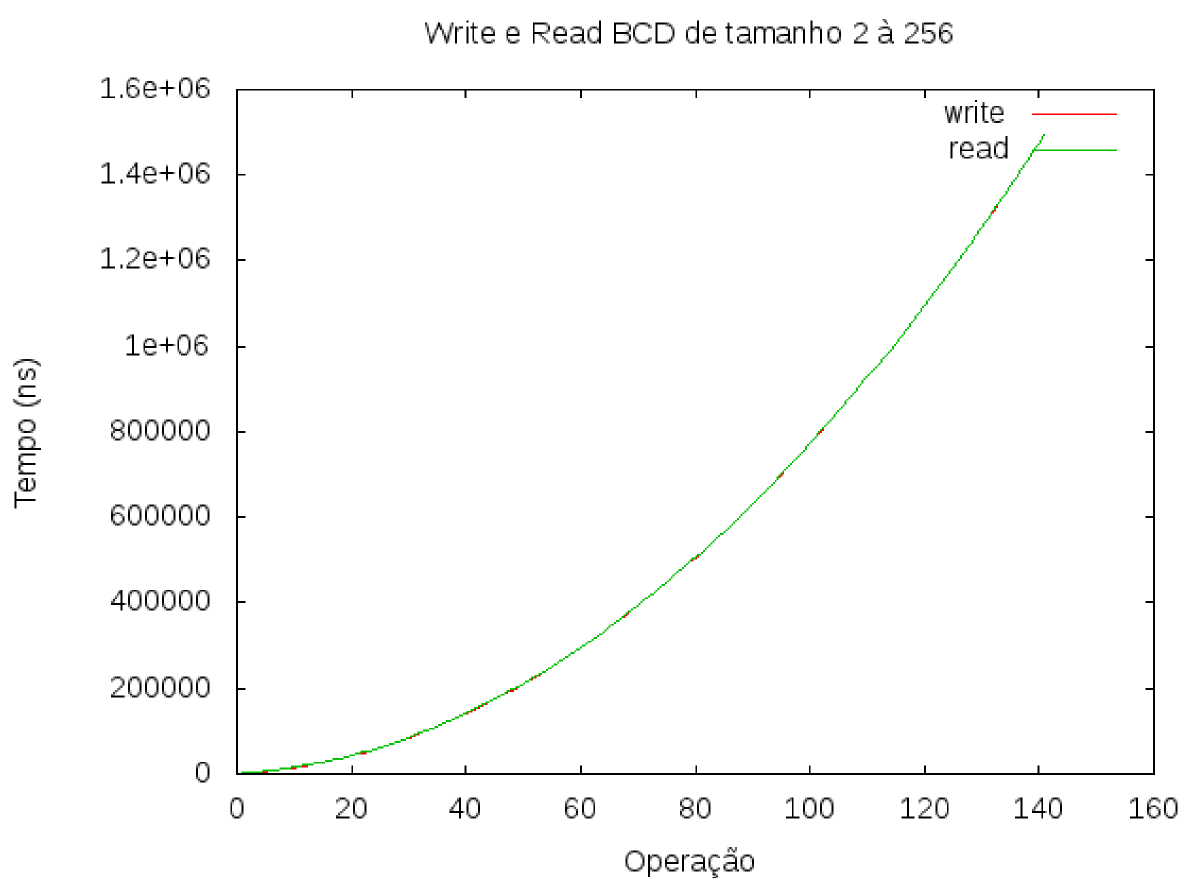


Figura 5.15: *Buffer* com fila de tamanho 2 á 256 e tempo simulado de 1500 μs.

A Figura 5.15 mostra que existe a separação entre as linhas pelo fator já explanado na subseção 5.6.3.1 porém com a diferença que nessas faixas de tamanhos foi possível gerar um valor a mais que com o tamanho 1.

Sobre a métrica, a taxa de utilização do *Buffer* foi de 100% como ao do tamanho 1, porém sendo feitas 141 escritas e leituras.

5.6.3.3 Análise dos resultados da simulação 3

Quanto à questão de utilização do *Buffer*, qualquer tamanho é viável, porém levando em conta o espaço em *hardware* ocupado, deve-se então optar por um tamanho menor de *Buffer* e como o tamanho 1 gerou 140 resultados e o tamanho 2, 141, então é definido que para este tipo de simulação de cargas, o tamanho 2 é o que possuí o melhor comportamento do *Buffer*.

5.6.4 Simulação 4

primo → *é primo ?*

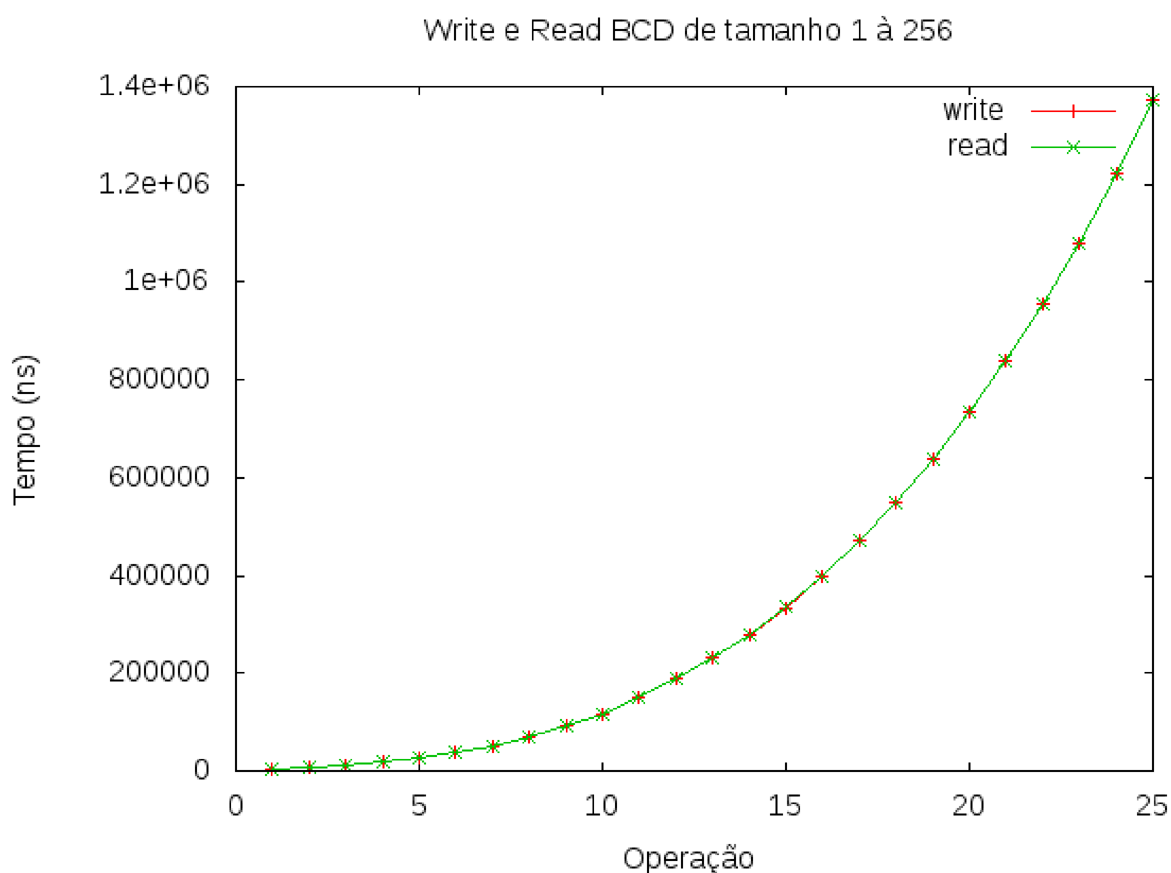


Figura 5.16: *Buffer* com fila de tamanho 1 a 256.

A simulação produz um resultado similar ao da simulação 2, em que, os tempos para escritas e leituras são muito próximos e que as linhas seguem o formato da linha de números primos mostrada na Figura 5.1. Obtém-se da análise dessa simulação que um *Buffer* de tamanho 1 tende a ser a melhor escolha em questão de custo x desempenho pois além de possuir a taxa de utilização em 100% também não gera desperdício de *Hardware* como fora explanado na subseção 5.6.2.1.

5.6.5 Simulação 5

$$random \rightarrow random$$

A simulação 5 foi feita a fim de obter-se o comportamento do *buffer* em um caso de aleatoriedade, em que leituras e escritas ocorrem de forma imprevisível. Também foi definido que esta simulação deveria ocorrer em um tempo muito maior que as demais, a fim de obter-se o comportamento da fila em um grande período e que esta deveria ser do maior tamanho testado nas simulações a fim de verificar o comportamento do *Buffer* em um caso mais extremo. Para este fim foi utilizado um tempo que equivale a 40.000 ciclos do processador, o que equivale a um tempo de simulação de 2 milisegundos. O resultado é mostrado na Figura 5.17.

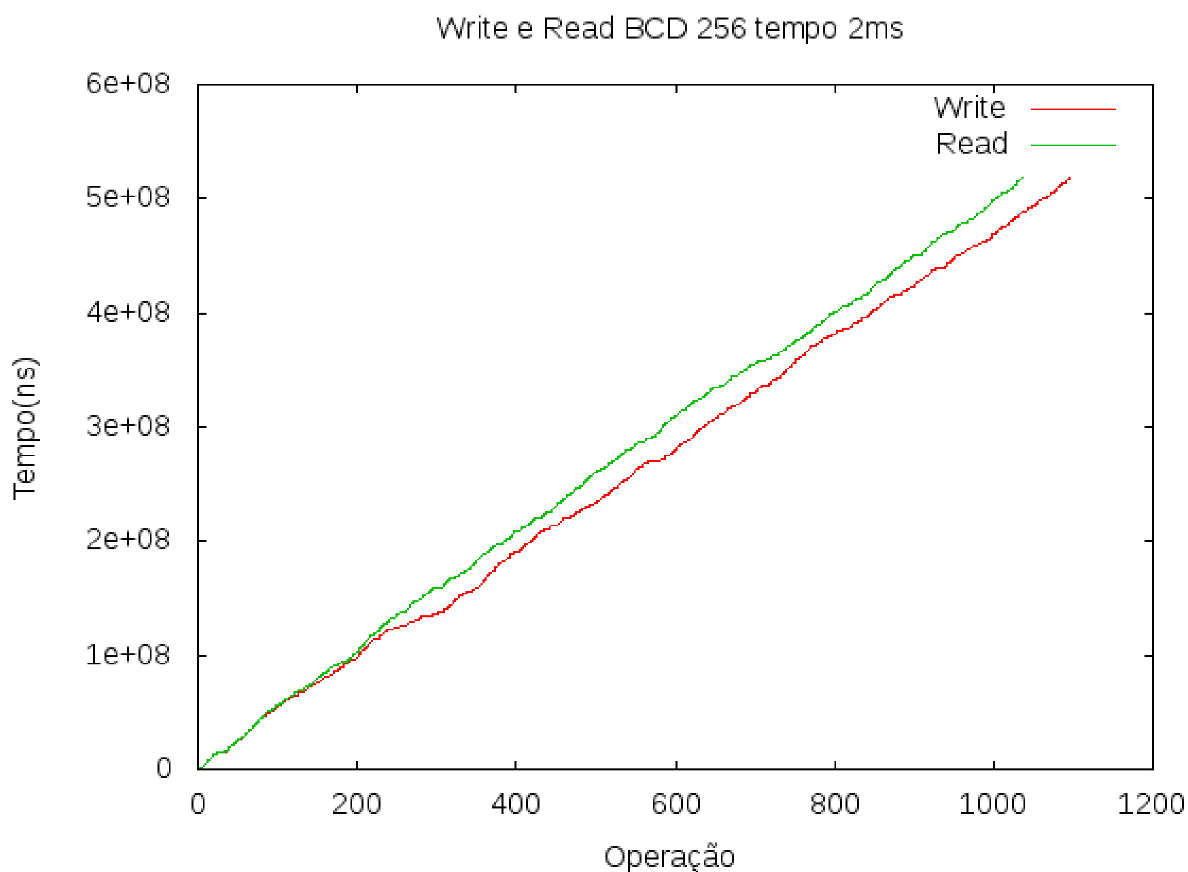


Figura 5.17: Comportamento *Buffer* com fila de tamanho 256 e tempo de execução de 2 ms.

A figura 5.17 mostra o comportamento do *Buffer* quando o acesso a ele é de forma aleatória, em que um produtor gera valores em tempos aleatórios para o *Buffer* e o consumidor lê em tempo aleatório os valores escritos. Com isso nota-se na figura que a partir do momento em que o consumidor leva um tempo um pouco maior para poder processar algo, ele causa a separação das linhas entre *writes* e *reads*. Essa separação significa que enquanto o consumidor está processando

algum valor o produtor, por ter espaço na fila, continua a gerar novos valores, até que a fila fique cheia.

Foram feitas 1036 leituras e 1096 escritas obtendo assim, uma taxa de utilização de **94%** do *Buffer*, e com 60 elementos na fila para serem processados ao final da simulação. Com esses fatos, afirma-se que tamanhos menores que 256 conseguem utilizar-se do *Buffer* de forma mais eficiente, pois no momento em que há a separação de linhas, os tamanhos menores seriam capazes de sincronizar as duas linhas novamente, diferentemente do que com os tamanhos maiores, que tendem apenas a aumentar esse espaço entre elas, com isso a fim de evitar desperdício de *Hardware* escolhe-se então o tamanho 1 para o *Buffer*.

CAPÍTULO 6

CONCLUSÃO

Esta monografia apresenta os conceitos sobre os processadores *MIPS*, a sua implementação no modelo *CMips*, o uso do deste no *CLupa*, descreve o conceito do *Buffer* **BCD**, seus detalhes quando ao seu *hardware* e implementação, e investiga o comportamento do **BCD** através de simulações de cargas de escritas e leituras neste *Buffer*, e segundo a métrica de utilização do *Buffer* baseadas na teoria de filas, conclui-se o melhor tamanho para cada simulação executada. Para encontrar a melhor utilização foram usadas funções para simular as cargas utilizadas tanto no núcleo *CMips-0* chamado de *produtor* quanto no núcleo *CMips-1* chamado de *consumidor*, a primeira função gera um número da sequência de *Fibonacci*, outra função gera números primos utilizando o crivo de Eratóstenes, e uma função que faze com que produtor e consumidor esperem um tempo aleatório .

Conclui-se que quando o produtor está com uma alta taxa de escritas no *Buffer* então o tamanho 1 é a melhor opção, porém no caso em que o produtor está com uma baixa taxa de escritas e o consumidor com uma alta taxa de leituras do *Buffer* então o tamanho 2 é melhor, conclui-se assim que tamanhos pequenos de *Buffer* são de melhores utilização pois sincronizam o produtor e consumidor. Tamanhos grandes tendem a obter sempre uma baixa taxa de utilização do *Buffer*. Conclui-se então desse trabalho que dado o tipo de carga os dois tamanhos: 1 e 2, são passiveis de serem utilizados entre o produtor e o consumidor do *cLupa* de forma eficiente e baixo custo.

CAPÍTULO 7

APÊNDICES

7.1 Script de Testes

O script de teste como mencionado na seção 5.2 é utilizado para executar as simulações propostas variando o tamanho do *Buffer*. Será demonstrado nas subseções a seguir o código desse script de teste, seu funcionamento e os diretórios de pastas usado por ele.

7.1.1 Código

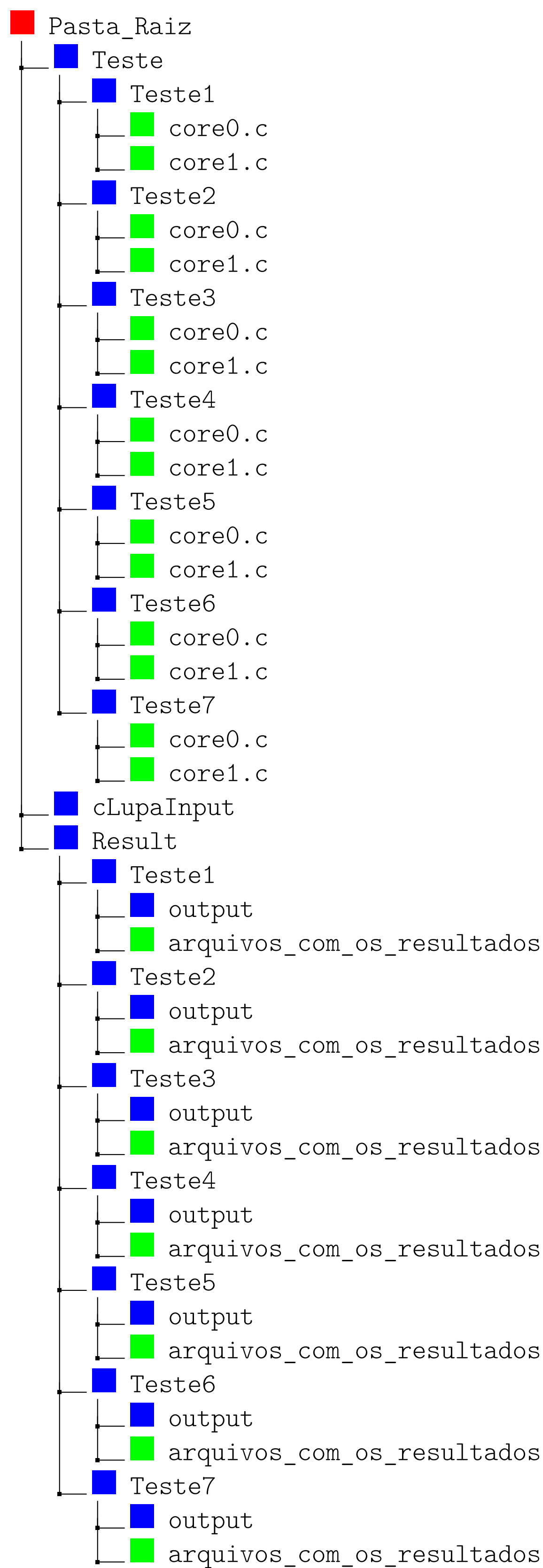
Programa 7.1: Script de teste para testar as simulações.

```
testeFolder=$1
while [ $testeFolder -le $2 ]
do
 echo "Teste $testeFolder"
 cp ./Teste/Teste$testeFolder/*.c ./cLupaInput/ &&
 cd cLupaInput &&
 rm -rf prog1.bin prog.bin data.bin data1.bin 2>&1 &&
 rm -rf ../prog1.bin ../prog.bin ../data.bin ../data1.bin 2>&1 &&
 ../bin/compile.sh core1.c &&
 cp data.bin data1.bin && cp prog.bin prog1.bin &&
 ../bin/compile.sh core0.c &&
 cp *.bin ../ &&
 cd .. &&

 n=$3

 # continue until $n equals 256
 mkdir -p "./result/teste$testeFolder/" &&
 mkdir -p "./result/teste$testeFolder/output" &&
 if [ $testeFolder -ge 5 ] && [ $testeFolder -lt 7 ]
 then
  filename10000="./result/teste$testeFolder/result_t10000_b$n"
  output10000="./result/teste$testeFolder/output/output_t10000_b$n"
  ./bin/run.sh -v pipe.sav -u u -t 10000 -n 1>"$output10000" 2>"$filename10000"
```

```
25  else
26   while [ $n -le $4 ] || [ $n == 7 ]
27   do
28    echo "BCD $n"
29
30    filename300="./result/teste$testeFolder/result_t300_b$n"
31    filename500="./result/teste$testeFolder/result_t500_b$n"
32
33    output300="./result/teste$testeFolder/output/output_t300_b$n"
34    output500="./result/teste$testeFolder/output/output_t500_b$n"
35
36    cp ./Teste/testes/$n/*.vhd ./vhdl/ &&
37
38    ./bin/run.sh -v pipe.sav -u u -t 300 -n 1>"$output300" 2>"$filename300" &&
39    ./bin/run.sh -v pipe.sav -u u -t 500 -n 1>"$output500" 2>"$filename500"
40
41    n=$(( n*2 )) # increments $n
42   done
43  fi
44
45  testeFolder=$(( testeFolder+1 ))
46 done
```

O script funciona da seguinte forma: o *loop* iniciado na linha 2 refere-se a simulação a ser executada, o parâmetro $1 é a simulação inicial, o paramêtro $2 a simulacao final, então este *loop* irá executar todas as simulações entre $1 e $2, as linhas 4 à 13 são referentes a compilação da simulação, cada simulação compilada gera 4 arquivos, sendo estes *data.bin*, *data1.bin*, *prog.bin* e *prog1.bin*, esses arquivos serão usados pelo *testbench* do *CMips* para executar o programa de cada *core*. As linhas 18 e 19 são as instruções para a criação de uma pasta com os resultados das simulações, cada simulação recebe uma pasta com seus resultados. o *if* da linha 20 é em consequência que as simulações 5 e 6 são simulações usadas para obtenção de outros dados não condizentes com as 5 simulações propostas. O *while* implementado na linha 26 refere-se a execução de uma simulação para um tamanho de *Buffer*, para isso ele se utiliza de dois paramêtros $4 e $5 que referem-se ao tamanho inicial do BCD e o tamanho final do BCD, cada teste de tamanho ira gerar dois tipos de arquivos, um contendo os resultados para um determinado tempo e outro contendo saidas do tipo *stdout* da simulação.

7.1.2 Diretórios



Os diretórios apresentados são divididos em três partes, a primeira refere-se a pasta *Teste*, esta pasta contém dentro dela as simulações que o script *teste.sh* irá executar dentro de cada

simulação se encontra dois arquivos .c, estes são as simulações que cada core ira executar. A segunda parte refere-se a compilação, os arquivos .c descritos anteriormente são movidos para dentro da pasta *cLupaInput*, lá dentro ocorre a compilação desse código e após isso a sua execução. A terceira parte é referente ao resultado da execução do .c, cada simulação possuí uma pasta que fica dentro do diretório *results* e cada uma dessas pastas possui vários arquivos que são os resultados desta simulação para um determinado tamanho de *Buffer* e também contem uma pasta chamada *output* que contém as saídas *stdout* de cada simulação.

BIBLIOGRAFIA

[1] Edmar Andre Bellorini. Clupa: Um ampliador digital de documentos impressos sobre uma plataforma multicore. http://www.inf.ufpr.br/roberto/dissEdmar.pdf, 2015. [Online; Acessado 22/06/2016].

[2] Mips architectures. https://imgtec.com/mips/architectures/. [Online; Acessado 14/11/2016].

[3] David A Patterson John L henessy. Computer archicture a quantitative approach fifth edition. Morgan Kaufmann, 2011.

[4] Arquitetura mips. https://pt.wikipedia.org/wiki/Arquitetura_MIPS. [Online; Acessado 14/11/2016].

[5] Análise comparativa de uso dos conjuntos de instruções dos microprocessadores de 32 bits mips, powerpc e sparc. http://www.inf.ufpr.br/roberto/dissFabiany.pdf, note = "[Online; Acessado 14/11/2016]".

[6] Dominic Sweetman. See mips® run second edition. Elsevier, 2007.

[7] Nintendo 64 technical specifications. https://en.wikipedia.org/wiki/Nintendo_64_technical_specifications. [Online; Acessado 14/11/2016].

[8] Playstation technical specifications. https://en.wikipedia.org/wiki/PlayStation_technical_specifications. [Online; Acessado 14/11/2016].

[9] Mips goes to pluto. https://imgtec.com/blog/mips-goes-to-pluto/. [Online; Acessado 14/11/2016].

[10] Roberto Andre Hexsel. cmips – a synthesizable vhdl model for the classical five stage pipeline. https://gitlab.c3sl.ufpr.br/roberto/cmips/blob/master/cMIPS/docs/cMIPS.pdf, 2016. [Online; Acessado 23/11/2016].