

ID: .....

**Laboratório de Arquitetura de Computadores**  
**Desenvolvimento e Funcionamento da Unidade**  
**de Controle Baseada na Arquitetura Base e**  
**Sua União Com a Unidade de Processamento**

São José dos Campos - Brasil

Abril de 2017



ID: .....

**Laboratório de Arquitetura de Computadores**  
**Desenvolvimento e Funcionamento da Unidade de**  
**Controle Baseada na Arquitetura Base e Sua União Com**  
**a Unidade de Processamento**

Relatório apresentado à Universidade Federal de São Paulo como parte dos requisitos para aprovação na disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Docente: Prof. Dr. Tiago de Oliveira  
Universidade Federal de São Paulo - UNIFESP  
Instituto de Ciência e Tecnologia - Campus São José dos Campos

São José dos Campos - Brasil  
Abril de 2017

# Resumo

Este projeto tratou do desenvolvimento da Unidade de Controle e de sua união com a Unidade de Processamento de forma funcional, baseada na Arquitetura base definida no projeto anterior. O principal objetivo deste projeto baseou-se em colocar em prática o conjunto de instruções, os modos de endereçamento e o esquemático definidos anteriormente utilizando da linguagem de descrição de *Hardware Verilog*. Por conta disto, esperou-se um funcionamento correto das instruções pré programadas na Memória de Instruções, algo alcançado, e comprovado através de *Waveforms*. Além disto, o funcionamento correto da Unidade de Controle foi comprovado. Cada item citado no desenvolvimento foi testado separadamente e, posteriormente, em conjunto, comprovando o funcionamento correto de todos os componentes.

**Palavras-chaves:** *Verilog*. *Hardware*. conjunto de instruções. Arquitetura. MIPS.

# Listas de Algoritmos

4.1	Sinais de <i>overflow</i> da <i>ALU</i> . . . . .	24
4.2	Divisão por zero . . . . .	24
4.3	Unidade Lógica e Aritmética . . . . .	24
4.4	Implementação do <i>Data Memory</i> . . . . .	26
4.5	Implementação da <i>Instruction Memory</i> . . . . .	27
4.6	Implementação do <i>Program Counter</i> . . . . .	30
4.7	Implementação do <i>Register Bank</i> . . . . .	31
4.8	Implementação do Extensor de 16 para 32 bits . . . . .	33
4.9	Implementação do módulo de saída . . . . .	34
4.10	Implementação do conversor de binário para <i>BCD</i> . . . . .	35
4.11	Todos os <i>Displays</i> . . . . .	37
4.12	Implementação do Temporizador . . . . .	38
4.13	Implementação do <i>Debouncer</i> . . . . .	38
4.14	Implementação da Unidade de Controle . . . . .	41
4.15	Módulo Final . . . . .	53
5.1	Sequência de Fibonacci na Linguagem C . . . . .	59
5.2	<i>Insertion Sort</i> . . . . .	60

# Listas de ilustrações

Figura 1 – Heiraquia de Memórias . . . . .	13
Figura 2 – Organização da Memória <i>cache</i> . . . . .	14
Figura 3 – Tipos de Instruções . . . . .	15
Figura 4 – Caminho de dados <i>MIPS</i> . . . . .	16
Figura 5 – <i>Waveform 1</i> . . . . .	64
Figura 6 – <i>Waveform 2</i> . . . . .	64
Figura 7 – <i>Waveform 3</i> . . . . .	65
Figura 8 – <i>Waveform 4</i> . . . . .	65
Figura 9 – Caminho de dados . . . . .	73
Figura 10 – Caminho de dados modificado . . . . .	73
Figura 11 – <i>Mux 1</i> . . . . .	74
Figura 12 – <i>Mux 2</i> . . . . .	74
Figura 13 – <i>Mux 3</i> . . . . .	74
Figura 14 – <i>Mux 4</i> . . . . .	75
Figura 15 – <i>Mux 5</i> . . . . .	75
Figura 16 – <i>Mux 6</i> . . . . .	75
Figura 17 – Conversor para <i>Display</i> de 7 segmentos . . . . .	76
Figura 18 – Primeiro termo de Fibonacci . . . . .	77
Figura 19 – Segundo termo de Fibonacci . . . . .	77
Figura 20 – Terceiro termo de Fibonacci . . . . .	78
Figura 21 – Quarto termo de Fibonacci . . . . .	78
Figura 22 – Quinto termo de Fibonacci . . . . .	79
Figura 23 – Sexto termo de Fibonacci . . . . .	79
Figura 24 – Caminho de dados <i>MIPS</i> com <i>pipelining</i> . . . . .	83

# Lista de tabelas

Tabela 1 – Tipos de Instrução . . . . .	19
Tabela 2 – Conjunto de Instruções . . . . .	21
Tabela 3 – <i>OPcodes</i> da <i>ALU</i> . . . . .	23
Tabela 4 – Tabela de sinais de controle . . . . .	40
Tabela 5 – Tipos de Instrução . . . . .	52

# Sumário

<b>1</b>	<b>INTRODUÇÃO</b>	<b>9</b>
<b>2</b>	<b>OBJETIVOS</b>	<b>11</b>
2.1	Geral	11
2.2	Específico	11
<b>3</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>13</b>
3.1	Tipos de Memória	13
3.2	Arquitetura e Organização	14
3.3	A Arquitetura <i>MIPS</i>	15
3.3.1	Conjunto de Instruções, Seus Tipos e Modos de Endereçamento	15
3.3.2	Caminho de dados <i>MIPS</i>	16
3.4	Unidade de Controle	17
3.4.1	<i>Hardwired</i>	17
3.4.2	Microprogramadas	17
3.5	Linguagem de descrição de <i>Hardware Verilog</i>	17
<b>4</b>	<b>DESENVOLVIMENTO</b>	<b>19</b>
4.1	Conjunto de Instruções	19
4.2	Tipos de Endereçamento	22
4.3	Módulos de operação da Unidade de Processamento	22
4.3.1	<i>Arithmetic and Logic Unit (ALU)</i>	22
4.3.2	<i>Data Memory</i>	25
4.3.3	<i>Instruction Memory</i>	26
4.3.4	<i>Program Counter</i>	29
4.3.5	<i>Register Bank</i>	31
4.3.6	Multiplexadores	32
4.3.7	Extensores de Sinal	33
4.3.8	Entrada e Saída	34
4.3.8.1	Instruções <i>In</i> e <i>Out</i>	34
4.3.8.2	<i>Displays</i> de 7 segmentos	35
4.3.8.3	Tratamento do <i>clock</i> e dos botões de entrada da FPGA	38
4.4	A Unidade de Controle	40
4.5	Módulo Final	53
<b>5</b>	<b>RESULTADOS OBTIDOS E DISCUSSÕES</b>	<b>59</b>

6	<b>CONSIDERAÇÕES FINAIS . . . . .</b>	<b>67</b>
	<b>REFERÊNCIAS . . . . .</b>	<b>69</b>
	<b>APÊNDICES</b>	<b>71</b>
	<b>APÊNDICE A – APÊNDICE 1 . . . . .</b>	<b>73</b>
	<b>APÊNDICE B – APÊNDICE 2 . . . . .</b>	<b>77</b>
	<b>ANEXOS</b>	<b>81</b>
	<b>ANEXO A – ANEXO 1 . . . . .</b>	<b>83</b>



# 1 Introdução

O presente relatório caracteriza-se como o terceiro da série, feito com a finalidade de desenvolver a Unidade de Processamento e a Unidade de Controle do processador através da linguagem de descrição de *Hardware Verilog*, levando em conta a Arquitetura, Conjunto de Instruções e Modos de Endereçamentos definidos no relatório anterior.

Um processador, também conhecido como Central Processing Unit ou CPU, consiste de um circuito integrado que é responsável por comandar todo o computador, utilizando dos demais componentes para realizar diversas funções, como operações aritméticas, lógicas, transferência de dados e entrada e saída. O processador é comumente conhecido como o “cérebro” do computador exatamente por estar a cargo de todas as operações essenciais.

Um processador só é capaz de realizar suas tarefas com base em conjunto de instruções que define seu funcionamento básico, desde operações simples como uma soma, até operações lógicas mais complexas. Um conjunto de instruções não passa de uma linguagem baseada em bits, sendo esse formato necessário para que a CPU possa interpretar os dados e executar as instruções. Além do conjunto de instruções, um processador precisa de uma arquitetura muito bem definida e executada em *Hardware* para seu funcionamento preciso e correto.

As arquiteturas dos processadores foram mudando muito com os anos em que seu desenvolvimento foi crescendo, mas os componentes mais importantes ainda continuam na mesma linha de características, sendo assim o processador pode ser dividido em partes principais, como sua Unidade de Processamento, responsável pela realização das diversas operações lógicas, aritméticas, transferência de dados e saltos, sempre se comunicando com o restante do processador e do sistema e sua Unidade de Controle, responsável pelo controle do caminho de dados, gerando os sinais de controle corretos de acordo com a instrução especificada. Sem a Unidade de Controle não é possível o funcionamento correto de um processador.

No desenvolvimento deste trabalho, abordou-se o escopo da implementação da Unidade de Controle e sua união com a Unidade de Processamento, visando colocar em prática a Arquitetura definida no primeiro projeto utilizando a linguagem de descrição de *Hardware Verilog*, colocando em prática os conhecimentos desta linguagem, os conhecimentos em arquitetura de computadores e de desenvolvimento de sistemas digitais, com a utilização do *software Quartus* e da placa FPGA.



## 2 Objetivos

### 2.1 Geral

Com o desenvolvimento deste projeto, buscou-se o desenvolvimento de um sistema digital funcional que se assemelha-se a um processador simples capaz de realizar todas as operações básicas necessárias para que este pudesse realizar um algoritmo simples de lógica de programação. Com isso, buscou-se também adquirir conhecimento sobre a implementação de um sistema digital mais complexo e que coloca-se os conhecimentos em arquitetura e organização de computadores em prática

### 2.2 Específico

Neste projeto, buscou-se a implementação e desenvolvimento da Unidade de Controle baseada na Arquitetura, Conjunto de Instruções e Modos de Endereçamento definidos nos projetos anteriores, utilizando *Verilog*. Além disso, buscou-se a interligação desta unidade com a Unidade de Processamento já desenvolvida, para assim completar o processador. Por fim, após as etapas anteriores estarem completas, o teste dos algoritmos pré-programados foi essencial, algo realizado na placa FPGA com a ajuda de seus componentes, necessário para comprovar os resultados esperados.



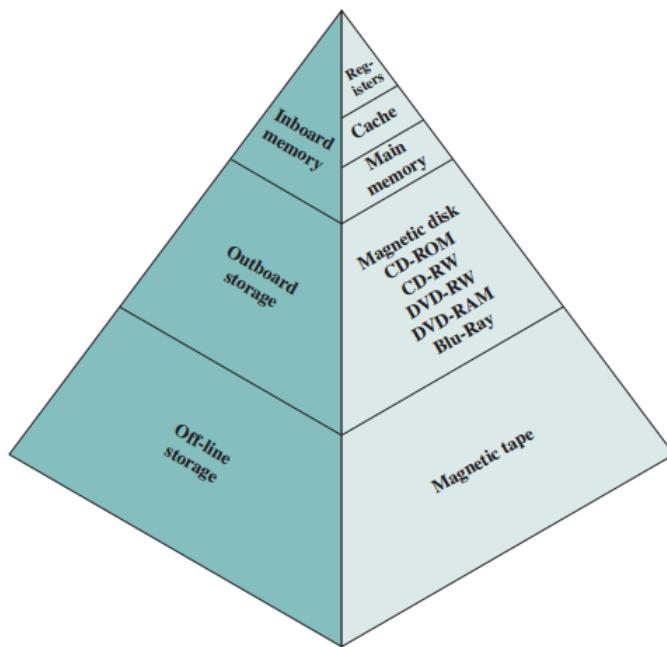
# 3 Fundamentação Teórica

## 3.1 Tipos de Memória

As memórias presentes em sistemas computacionais são de extrema importância para o seu funcionamento e estas são divididas em diversas classes baseadas em seu custo por *bit*, armazenamento e velocidade.

Esses tipos de memória são divididos em uma hierarquia, uma pirâmide que agrupa todos os tipos de memória e as divide conforme suas características [1], como visto na [Figura 1](#). Cada tipo possui uma característica específica, como as memórias *cache*, são memórias mais rápidas, porém são mais caras por *bit* e com menor capacidade [1].

Figura 1 – Heiraquia de Memórias



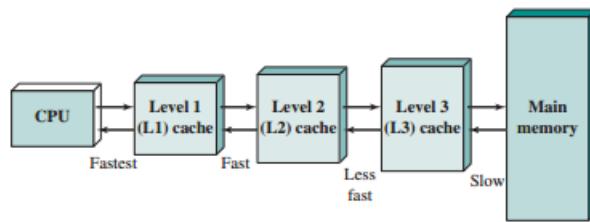
Fonte: Computer Organization and Architecture [1]

Como observado na [Figura 1](#), conforme tem-se memórias maiores, o tempo de acesso aumenta e o custo por *bit* diminui. O contrário acontece com memórias mais rápidas que apresentam capacidade menor e custo por *bit* maior.

Memórias do tipo *cache* e registradores, são as mais utilizadas quanto trata-se de processadores. Isso deve-se ao fato de que são os tipos de memória com menor latência e, por isso, menor tempo de acesso e maior velocidade [1] requerida pelo processador, porém possuem capacidade bem menor do que as memórias principais.

A memória *cache* geralmente é distribuída em níveis diferentes para a comunicação com o processador, sendo que os níveis mais distante do processador apresentam menor velocidade e maior capacidade e comunicação direta com a memória principal, e os níveis mais próximos são menores, porém mais rápidos e possuem apenas comunicação entre eles e o processador [1], como visto na [Figura 2](#).

Figura 2 – Organização da Memória *cache*



Fonte: Computer Organization and Architecture [1]

Esses fatores são importantes no desenvolvimento de um sistema computacional já que os projetistas precisam levar o custo, tamanho e velocidade das memórias para cada tipo de necessidade do computador [2].

## 3.2 Arquitetura e Organização

Arquitetura e organização são conceitos chave no desenvolvimento de sistemas computacionais, em especial um processador. Arquitetura é um termo utilizado para englobar um conjunto de regras que define a implementação, funcionalidade, conjunto de instruções e até a organização de um sistema computacional [2]. Os objetivos de design de uma arquitetura sempre visam o maior desempenho, funcionalidade, menor consumo de energia e compatibilidade entre sistemas. Técnicas que envolvem o desenvolvimento da arquitetura são o paralelismo e o *pipelining*. Existem alguns exemplos de arquiteturas utilizadas conforme os anos, mas as mais evidentes são as arquiteturas *RISC* e *CISC*.

A organização de um computador está interligada com a arquitetura, dependendo desta para seu funcionamento. Enquanto a arquitetura lida com os conceitos relacionados a como serão tratados os métodos de endereçamento, como as instruções serão projetadas, de que forma os componentes devem funcionar e conceitos nestes aspectos, a organização trata do ponto mais prático, lidando com a interligação dos componentes, como cada parte do sistema deve se comunicar, sempre seguindo as especificações da arquitetura.

### 3.3 A Arquitetura MIPS

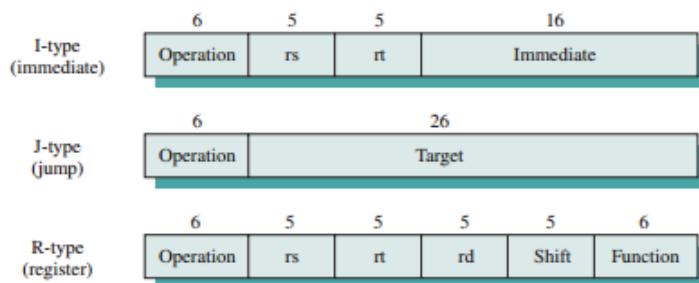
Uma das variantes da Arquitetura *RISC* foi a Arquitetura *MIPS* (*Microprocessor Without Interlocked Pipeline Stages*), desenvolvida pela *MIPS Technology Inc.*[1], com propósitos comerciais e acadêmicos.

Este tipo de Arquitetura apresenta características únicas que possibilitam um desenvolvimento de sua organização de forma simples e também possibilita a implantação de *pipelining* mais facilmente [1]. É uma arquitetura totalmente baseada no uso de registradores para as operações internas do processador, contantando com 32 registradores de propósito geral de 32 bits cada, caracterizando uma maior velocidade, e todas as operações de movimentação de dados baseiam-se em *load/store* [2].

#### 3.3.1 Conjunto de Instruções, Seus Tipos e Modos de Endereçamento

Para que sua organização fosse simplificada e utilizasse somente de registradores em operações internas, fez-se com que o seu conjunto de instruções fosse padronizado com instruções de apenas 32 bits divididas em 3 tipos distintos [3] conforme a Figura 3: R, I e J.

Figura 3 – Tipos de Instruções



Fonte: Computer Organization and Architecture [1]

Cada tipo possui campos diferentes em seu formato e atendem a diferentes instruções. O tipo R é utilizado por instruções aritméticas que dependem do conteúdo de 2 registradores para suas operações, assim como instruções lógicas de comparação e deslocamento. O tipo I é utilizado para instruções aritméticas e lógicas que dependem do valor imediato que acompanha a instrução e para instruções de *branch*. Por fim, o tipo J é utilizado para instruções de salto [4].

Por fim, as instruções presentes na Arquitetura *MIPS* possuem modos de endereçamento diferenciados e necessários para cada tipo de instrução e operação, se dividindo em: Endereçamento Imediato (o operando é a constante dentro do campo da instrução), Endereçamento por Registrador (o operando é o registrador), Endereçamento Indireto ou por

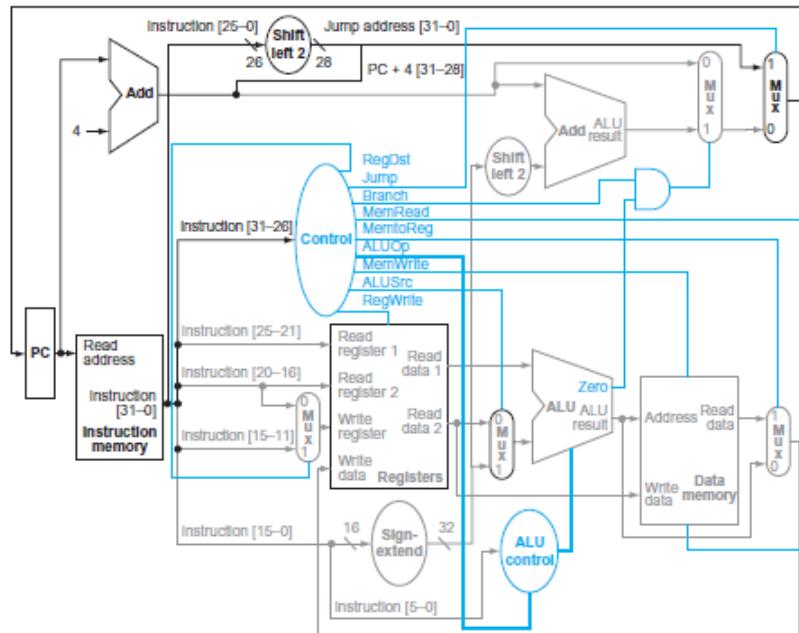
Deslocamento (o operando é um local da memória apontado pela soma de um registrador com um imediato), Endereçamento Relativo ao *Program Counter* (o endereço de *branch* é a soma do endereço atual mais a constante presente na instrução) e o Endereçamento Pseudo direto (o endereço de pulo está contido no imediato da instrução de tipo J) [2].

### 3.3.2 Caminho de dados MIPS

Um caminho de dados, ou *datapath*, é a representação dos componentes do processador conectados entre si, gerando uma estrutura completa que pode executar as instruções definidas pela arquitetura [2].

O caminho de dados da Arquitetura *MIPS* para uma implementação monociclo é representado na figura [Figura 4](#), onde todos os componentes estão ligados de forma que qualquer instrução pode ser executada em um ciclo de *clock*. Este tipo de caminho de dados pode ser aprimorado para que o *pipelining* e o multi-ciclo possam ser implementado. A implementação em *pipelining* pode ser vista no Anexo 1, na [Figura 24](#).

Figura 4 – Caminho de dados *MIPS*



Fonte: Computer Organization and Design [2]

Cada componente possui um papel diferente no funcionamento deste caminho de dados: *Program counter* passa para o próximo endereço de instrução, *Instruction memory* busca a instrução, *Register bank* possui os registradores para as operações, *ALU (Arithmetic and Logic Unit)* realiza as operações lógicas e aritméticas, *Data memory* lê os dados com base em endereço e a *Control unit* gera os sinais de controle para que as instruções sigam o caminho correto [2].

## 3.4 Unidade de Controle

Uma Unidade de Controle de um processador é responsável por controlar todas as operações de um processador e executar corretamente todas as instruções presentes no conjunto de instruções do processador[2]. Para que ela possa realizar isto, esta unidade gera sinais de controle que ditam o fluxo de execução e o *timing* do processador, para que assim funcione corretamente.

Sua implementação pode variar em termos de dificuldade, mas é uma das partes mais complicadas em termos de lógica digital dentro do processador, envolvendo circuitos complexos necessários para a decodificação correta das instruções e para gerar os sinais de controle[1].

Existem dois tipos principais de implementação desta unidade: *hardwired* e micro-programada.

### 3.4.1 Hardwired

Unidades de Controle que são implementadas desta maneira apresentam maior velocidade, porém são menos flexíveis. São implementadas com circuitos combinacionais e utilizam de arquiteturas fixas, já que qualquer mudança implica na alteração dos fios que formam a unidade em si, por isso possuem pouca flexibilidade[5]. Assim, são preferíveis em arquiteturas RISC, por possuírem um conjunto de instruções mais simplificado.

### 3.4.2 Microprogramadas

São unidades em que seus sinais de controle são ditados por microinstruções situadas em uma memória de controle[2], uma memória a parte destinada apenas para a Unidade de Controle. A vantagem é que são menos custosas de implementar e são mais flexíveis, sendo necessária apenas uma mudança nas microinstruções.

## 3.5 Linguagem de descrição de *Hardware Verilog*

Linguagens de programação de alto nível existem desde que os projetistas necessitaram de algo formal e mais abstrato para que programas mais complexos fossem concebidos, surgindo assim linguagens como *Pascal*, *Cobol*, *C*, *BASIC*, entre outras. Porém, os projetistas de *hardware* também sentiram necessidade de uma linguagem que descrevesse todo seu trabalho com os sistemas computacionais, surgindo assim as *Hardware Description Languages (HDLs)*, sendo uma destas o *Verilog* [6].

As *HDLs* possuem um grande diferencial de possuírem características de linguagens procedurais e ao mesmo tempo ser uma linguagem de execução paralela. Muitos circuitos

projetados em linguagem de descrição de *hardware* possuem blocos que executam em paralelo [6], como os blocos *always* em *Verilog*. Esta é uma característica muito importante deste tipo de linguagem e que trás um ganho de desempenho e uma melhora na lógica de programação.

Estas são linguagens que facilitaram em muito o desenvolvimento de sistemas digitais, já que o trabalho extra dos projetistas de ligar portas lógicas manualmente foi deixado de lado para o uso de linguagens que facilitavam essa tradução da lógica digital para a prática [6].

# 4 Desenvolvimento

Para este projeto, utilizou-se da Arquitetura base definida no projeto anterior (PC1), implementando a Unidade de Controle com a presença da Unidade de Processamento, utilizando *Verilog* e o software *Quartus*. Todas as instruções foram levadas em consideração no desenvolvimento deste projeto assim como o esquemático produzido.

## 4.1 Conjunto de Instruções

Este conjunto contou com 32 instruções que abrangem o grupo aritmético, lógico, transferência de dados e entrada e saída. Como toda instrução possui o tamanho fixo de 32 bits, cada tipo de instrução apresentou modificações nos campos da instrução para atender as necessidades na hora da execução.

Tabela 1 – Tipos de Instrução

	Campos					
Tipo 1	OPcode	R3	R1	R2	shift	-
Tipo 2	OPcode	R3	R1	Imediato/Endereço		
Tipo 3	OPcode	Endereço				

Fonte: Autoria Própria

Essas modificações podem ser vistas na [Tabela 5](#), onde instruções do tipo 1 possuíram os campos subdivididos em *OPcode*, R3, R1, R2, *shamt* e o último campo vazio, propícias para instruções lógicas, aritméticas e de salto utilizando registrador. *OPcode* representou os bits necessários para a identificação da instrução pela unidade de controle para que esta tomasse as devidas providências no tratamento das instruções, ou seja, este campo encarregou-se de ditar o caminho de dados de determinada instrução. Os campos R3, R1 e R2 foram necessários para operações aritméticas e lógicas, ou seja, R3 correspondeu ao registrador de destino das operações, R1 e R2 corresponderam aos registradores de operação, ou seja, foram responsáveis por armazenar os valores dos operandos em instruções de soma por exemplo. O campo *shamt*, abreviação de *shift amount*, responsabilizou-se por armazenar a quantidade de bits que deveriam ser deslocados em uma instrução de deslocamento. O campo vazio deveu-se ao fato de que, como este modelo foi baseado no conjunto do MIPS, o campo *funct* do conjunto MIPS não foi necessário já que não planejou-se o componente de controle da unidade lógica e aritmética neste processador.

Designou-se o tipo 2 da [Tabela 5](#) para instruções lógicas e aritméticas que utilizavam de valores imediatos, para instruções do tipo *branch* e para transferência de dados. Os campos R2 e *shamt* foram retirados para a inclusão do campo Imediato/Endereço, necessários para operações com valores imediatos e para saltos condicionais, chamados de *branches*. Quando uma instrução não precisou do valor de um segundo registrador como operando, o valor imediato pôde ser usado para o cálculo, assim como utilizou-se esse campo como endereço para carregamento de dados.

Designou-se o tipo 3 da [Tabela 5](#) para instruções de pulo, de parada e de ciclo vazio. Esse tipo apresentou apenas 2 campos, *OPcode* e Endereço. Ocultaram-se os outros campos já que não eram necessários para estas instruções, pois uma instrução de pulo precisou somente de um endereço, uma instrução de parada de processamento utilizou somente o *OPcode* e uma instrução vazia também utilizou somente o *OPcode*.

Em todas as instruções, os campos *OPcode* e de registradores possuíram as mesmas finalidades descritas anteriormente.

Depois de determinados os tipos de instruções, a escolha do conjunto de instruções foi o próximo passo. Utilizando dos 3 formatos da [Tabela 5](#), escolheu-se um conjunto com 32 instruções que abrangem os campos aritméticos, lógicos, transferência de dados, pulos, saltos condicionais e entrada e saída. Cada instrução recebeu um *OPcode* específico que variou de 0 a 31 em binário. Todas essas instruções podem ser vistas na [Tabela 2](#).

Cada instrução apresentou uma funcionalidade específica, e todas elas representaram um conjunto de instruções essenciais e de propósito geral, ou seja, mesmo não abrangendo operações complexas, essas mostraram-se necessárias para que chegue-se à operações complexas combinando-as.

O campo Instruções da [Tabela 2](#) definiu o nome das instruções do conjunto em abreviações. O campo Função da [Tabela 2](#) definiu o que cada instrução realizou. O campo Expressão definiu o que cada instrução realizou em termos matemáticos e lógicos.

As instruções *Add*, *Sub*, *Mul*, *Div*, *Slt*, *Shr*, *Shl*, *Jr*, *Xor*, *And*, *Or* e *Not*, enquadraram-se no tipo 1 da [Tabela 5](#), ou seja, são instruções lógicas, aritméticas e de salto utilizando registrador.

As instruções *Addi*, *Subi*, *Li*, *Lw*, *Sw*, *Xori*, *andi*, *Ori*, *Beq*, *Bneq*, *Bnez*, *Bgt*, *Blt*, *Bgtz* e *Bltz* enquadraram-se no tipo 2 da [Tabela 5](#), ou seja, são instruções que utilizaram-se do valor imediato para contas, operações lógicas, transferência de dados e saltos condicionais ou *branches*.

Tabela 2 – Conjunto de Instruções

OP Code	Instruções	Função	Expressão
000000	Nop	Ciclo vazio	-
000001	Halt	Para o processador	-
000010	Add	Soma	$R3 = R1 + R2$
000011	Addi	Soma imediato	$R3 = R1 + Imd$
000100	Sub	Subtração	$R3 = R1 - R2$
000101	Subi	Subtração imediato	$R3 = R1 - Imd$
000110	Mul	Multiplicação	$R3 = R1 * R2$
000111	Div	Divisão	$R3 = R1 / R2$
001000	Slt	1 se menor, 0 se maior	$R1 < 0 ? 1 : 0$
001001	Shr	Desloca bits para a direita	$R3 = R1 >> shamt$
001010	Shl	Desloca bits para a esquerda	$R3 = R1 << shamt$
001011	Lw	Carrega palavra	$R3 = \text{Mem}(R1 + \text{adress})$
001100	Li	Carrega imediato	$R3 = \text{Mem}(Imediato)$
001101	Sw	Guarda palavra	$\text{Mem} = R1$
001110	Jump	Pula de endereço	$\text{Pc} = \text{NovoPc}$
001111	Jr	Pula para registrador	$\text{Pc} = \text{Adress de Reg}$
010000	Xor	Xor bit a bit	$R3 = R1 \wedge R2$
010001	And	And bit a bit	$R3 = R1 \& R2$
010010	Or	Or bit a bit	$R3 = R1 \parallel R2$
010011	Not	Nega o bit	$R3 = \sim R1$
010100	Xori	Xor imediato	$R3 = R1 \wedge Imd$
010101	Andi	And imediato	$R3 = R1 \& Imd$
010110	Ori	Or imediato	$R3 = R1 \parallel Imd$
010111	Beq	Branch se igual	Se $R1 == R2$ , $\text{PC} = \text{novoPC}$
011000	Bneq	Branch se diferente	Se $R1 != R2$ , $\text{PC} = \text{novoPC}$
011001	Beqz	Branch se igual a zero	Se $R1 == 0$ , $\text{PC} = \text{novoPC}$
011010	Bnez	Branch se diferente de zero	Se $R1 != 0$ , $\text{PC} = \text{novoPC}$
011011	Bgt	Branch se maior	Se $R1 > R3$ , $\text{PC} = \text{novoPC}$
011100	Blt	Branch se menor	Se $R1 < R3$ , $\text{PC} = \text{novoPC}$
011101	Bgtz	Branch se maior que zero	Se $R1 > 0$ , $\text{PC} = \text{novoPC}$
011110	Bltz	Branch se menor que zero	Se $R1 < 0$ , $\text{PC} = \text{novoPC}$
011111	In	Sinal de entrada	-
100000	Out	Sinal de saída	-

Fonte: Autoria Própria

As instruções *Jump*, *Nop* e *Halt* enquadram-se no tipo 3 da Tabela 5, ou seja, são instruções de salto ou de parada. A instrução *Halt* para todo o processamento e a instrução *Nop* configura um ciclo vazio.

As instruções *In* e *Out* não receberam um tipo definido, já que foram instruções responsáveis pela interação com o usuário, recebendo um valor de entrada ou mandando um sinal de saída conforme o necessário.

## 4.2 Tipos de Endereçamento

Após a definição de um conjunto de instruções, necessitou-se da declaração de tipos de endereçamento realizados pelo processador. Essa categoria subdividiu-se em 3: endereçamento imediato, por registrador e indireto por registrador.

O endereçamento imediato referenciou-se à instruções que utilizaram do campo imediato, ou seja, instruções do tipo 2 da [Tabela 5](#). Neste tipo de endereçamento, o operando localizou-se como um valor imediato referenciado diretamente na instrução.

O endereçamento por registrador definiu-se quando um operando era referenciado por um registrador, ou seja, o registrador continha o operando, como visto nas instruções do tipo 1 da [Tabela 5](#).

O endereçamento por registrador definiu-se quando um registrador continha um endereço que apontava para uma posição de memória e nesta posição situava-se o valor necessário para a operação.

## 4.3 Módulos de operação da Unidade de Processamento

Esta Arquitetura apresentou componentes similares aos da Arquitetura *MIPS*, sendo assim, foram desenvolvidos os componentes presentes no esquemático da [Figura 10](#) em *Verilog* para que executassem todas as instruções descritas no conjunto de instruções, presente na [Tabela 2](#). Como a Unidade de Controle foi o foco deste projeto, os sinais de controle não precisaram mais serem modificados manualmente, já que este é o principal objetivo deste módulo, explicado posteriormente.

### 4.3.1 *Arithmetic and Logic Unit (ALU)*

A *Arithmetic and Logic Unit*, ou Unidade Lógica e Aritmética, foi o componente encarregado de executar todas as operações aritméticas e lógicas, ou seja, foi o componente responsável por executar as quatro operações aritméticas base, deslocamento de *bits*, comparações lógicas *bit a bit* e comparações numéricas. A *ALU* não dependeu do *clock*, uma vez que ela deveu executar operações baseando-se apenas na mudança do *OPcode* e dos operandos.

Para cada tipo de operação, necessitou-se de um *OPcode* específico, gerado pela Unidade de Controle com base no *OPcode* da instrução fornecida pela *Instruction Memory*. Representaram-se esses *OPcodes* da *ALU* na [Tabela 3](#).

Tabela 3 – *OPcodes* da *ALU*

<i>ALU OPcode</i>	<i>Operação</i>
00000	Soma
00001	Subtração
00010	Multiplicação
00011	Divisão
00100	Nega bit
00101	And bit a bit
00110	Or bit a bit
00111	Xor bit a bit
01000	Desloca Esquerda
01001	Desloca Direita
01010	Menor que (1 se sim, 0 se não)
01011	Maior que (1 se sim, 0 se não)
01100	Igual (1 se sim, 0 se não)
01101	Menor ou igual que (1 se sim, 0 se não)
01110	Maior ou igual que (1 se sim, 0 se não)
01111	Diferente (1 se sim, 0 se não)

Fonte: Autoria Própria

O *OPcode* da *ALU* teve seu tamanho determinado para 5 *bits* pois, caso fossem necessárias mais operações no futuro, não seria necessária a mudança do tamanho do *OPcode*, já que todos os 4 *bits* menos significativos já foram preenchidos.

Como este componente encarregou-se de diversas operações aritméticas, tratou-se a possibilidade de *overflow* de maneira que este caso fosse comunicado ao usuário através de dois sinais que saem da *ALU*: O sinal que indica *overflow* na soma (*overflowadd*) e o sinal que indica *overflow* na subtração (*overflowsub*), como consta na implementação em *Verilog* contida no [Algoritmo 4.1](#).

Esta implementação segue a idéia de que o *software Quartus* utiliza da representação dos números binários em complemento de 2. Sendo assim, se uma soma entre dois números binários positivos é realizada e o *overflow* ocorre, o resultado desta operação apresenta o sinal invertido, ou seja, negativo. A mesma coisa ocorre com a adição de dois números negativos, com o resultado apresentando-se positivo. Com uma subtração de números de sinais diferentes, observam-se dois casos semelhantes: Caso o primeiro operando seja positivo e o segundo negativo, o sinal do resultado é negativo; caso o primeiro operando seja negativo e o segundo positivo, o sinal do resultado é positivo.

Com esta definição em mãos, possibilitou-se o desenvolvimento de um sistema de predição de *overflow* em somas e subtrações a partir da checagem do *bit* mais significativo dos operandos, ou seja, seu *bit* de sinal, e da checagem do *bit* mais significativo do resultado, gerando um sinal alto de *overflow* caso um ocorra, conforme o [Algoritmo 4.1](#).

Algoritmo 4.1 – Sinais de *overflow* da *ALU*

```

1      assign value_add = op1 + op2;
2      assign value_sub = op1 - op2;
3
4      assign overflow_add = (op1[31] == op2[31] && op1[31] != value_add[31]) ? 1 : 0;
5      assign overflow_sub = (op1[31] != op2[31] && op1[31] != value_sub[31]) ? 1 : 0;

```

Ainda existiam dois casos não tratados de conflitos no *ALU*, um deles de *overflow* na multiplicação e o outro de divisão por 0. Para tratar o caso da multiplicação, limitou-se o tamanho dos operandos para a multiplicação de 32 para 16 *bits*, reduzindo a possibilidade de *overflow*. Para o caso da divisão por 0, restringiu-se o uso do segundo operando da divisão com o valor zero, já que toda vez que o usuário tentar utilizar o valor 0 como denominador, a *ALU* automaticamente muda para 1, mantendo o número original e evitando uma condição de erro, como visto na [Algoritmo 4.2](#). Assim, o segundo operando torna-se a variável *notzero*, assumindo o valor 1 caso o *op2* seja 0 ou assumindo o próprio valor do *op2* caso o valor deste não seja 0.

Algoritmo 4.2 – Divisão por zero

```

1      assign not_zero = (op2 == 0) ? 1 : op2;

```

Por fim, como a *ALU* responsabilizou-se por operações de comparação, os saltos condicionais também dependeram dela, mais precisamente do sinal *zero*, *output* da *ALU*. Este sinal *zero* assume os valores 1 ou 0: caso o resultado da operação da *ALU* seja 0, o sinal *zero* assume o valor 1, caso contrário, o sinal *zero* assume o valor 0. Isso foi necessário já que, se o sinal está em alta e o sinal de controle de um *branch* também está em alta, o salto condicional deve ser realizado, já que um *branch* depende de uma comparação realizada na *ALU*. Esta atribuição de valor ocorreu na última linha de código da *ALU*, visto no [Algoritmo 4.3](#).

Algoritmo 4.3 – Unidade Lógica e Aritmética

```

1 module ALU (OPcode, op1, op2, result, shamt, zero, overflow_add,
2   overflow_sub, clock);
3
4     input [31:0] op1, op2;
5     input clock;
6     input [4:0] OPcode;
7     input [4:0] shamt;
8     output reg [31:0] result;
9     reg [15:0] a, b;
10    wire [31:0] not_zero;
11    output zero, overflow_add, overflow_sub;
12    wire [31:0] value_add, value_sub;

```

```

12
13     assign value_add = op1 + op2;
14     assign value_sub = op1 - op2;
15
16     assign overflow_add = (op1[31] == op2[31] && op1[31] != value_add[31]) ? 1 : 0;
17     assign overflow_sub = (op1[31] != op2[31] && op1[31] != value_sub[31]) ? 1 : 0;
18
19     assign not_zero = (op2 == 0) ? 1 : op2;
20
21     always@(op1 or op2 or OPcode)
22         begin
23             a <= {op1[15:0]};
24             b <= {op2[15:0]};
25             case (OPcode[4:0])
26                 5'b00000: result = op1 + op2;
27                 5'b00001: result = op1 - op2;
28                 5'b00010: result = a * b;
29                 5'b00011: result = op1 / not_zero; o
30                 5'b00100: result = ~op1;
31                 5'b00101: result = op1 & op2;
32                 5'b00110: result = op1 | op2;
33                 5'b00111: result = op1 ^ op2;
34                 5'b01000: result = op1 << shamt;
35                 5'b01001: result = op1 >> shamt;
36                 5'b01010: result = op1 < op2 ? 1 : 0;
37                 5'b01011: result = op1 > op2 ? 1 : 0;
38                 5'b01100: result = op1 == op2 ? 1: 0;
39                 5'b01101: result = op1 <= op2 ? 1 : 0;
40                 5'b01110: result = op1 >= op2 ? 1 : 0;
41                 5'b01111: result = op1 != op2 ? 1 : 0;
42             default: result = 0;
43         endcase
44     end
45
46
47     assign zero = (result == 0);
48
49 endmodule

```

### 4.3.2 Data Memory

A Memória de Dados, ou *Data Memory*, encarregou-se de ser o componente responsável por armazenar dados de 32 bits, como uma memória secundária. Este componente contou com 32 espaços de memória com 32 bits de tamanho cada um, ou seja, uma memória

com espaço para 1024 *bits* de armazenamento de dados. Cada posição desta memória pôde ser acessada através de uma posição escolhida, como num vetor em linguagem C. O *Data Memory* dependeu do *clock* do sistema. Este componente foi representado no [Algoritmo 4.4](#).

A *Data Memory* apresentou um sinal de controle responsável por controlar a escrita ou não de dados, o sinal de controle *MemWrite*, visto no [Algoritmo 4.4](#). Caso este sinal apresentasse o valor 1, os dados seriam escritos no *Data Memory*, ou seja, uma instrução de *Store Word*; caso apresentasse o valor 0, nada seria escrito na memória, ou seja, uma instrução de *Load Word*, que precisa somente ler o conteúdo da memória. Este componente, no entanto, fornece o sinal de saída de dados constantemente independente de uma instrução *Load Word*. Isto não é um problema, já que há um multiplexador responsável por utilizar essa saída somente em uma instrução *Load Word*.

Algoritmo 4.4 – Implementação do *Data Memory*

```

1 module DataMemory (clock, WriteData, MemWrite, DataMemoryOut, adress);
2
3     input [31:0] WriteData, adress;
4     input clock, MemWrite;
5     output [31:0] DataMemoryOut;
6
7     reg [31:0] memory [31:0];
8
9     always @ (posedge clock)
10        begin
11            if (MemWrite)
12                memory [adress] = WriteData;
13        end
14
15        assign DataMemoryOut = memory [adress];
16
17 endmodule

```

### 4.3.3 *Instruction Memory*

A Memória de Instruções, ou *Instruction Memory*, responsabilizou-se por armazenar, em sequência, uma lista de todas as instruções a serem executadas pelo processador, ou seja, um pequeno programa convertido em linguagem binária. Assim, a cada pulso de *clock*, uma nova instrução era mandada como *output* dependendo do endereço fornecido pelo *Program Counter*, sempre na borda positiva do *clock*. Sua implementação encontra-se no [Algoritmo 4.5](#).

A memória presente neste módulo segue o mesmo princípio da memória do *Data Memory*, porém com espaço para 55 instruções de 32 *bits* cada. Cada posição do vetor

representa um endereço fornecido pelo *Program Counter*, começando em 0 e aumentando sequencialmente de 1 em 1, caso não ocorra algum salto.

Como desenvolveu-se a *Instruction Memory* para ser síncrona com a borda positiva do *clock*, não foi necessário escrever as instruções a cada ciclo de *clock*, por isso foi criada uma *flag*, vista no [Algoritmo 4.5](#), com valor inicial 0, responsável por controlar a escrita de dados na *Instruction Memory*, ou seja, no primeiro *clock* as instruções são escritas na memória e depois esta *flag* recebe o valor 1, parando de escrever continuamente a cada ciclo de *clock*.

Algoritmo 4.5 – Implementação da *Instruction Memory*

```

1 module InstructionMemory (adress , InstructionOut , clock);
2
3     input [9:0] adress;
4     input clock;
5     output [31:0] InstructionOut;
6     reg [31:0] mem [55:0];
7     integer flag = 0;
8
9     always @ (posedge clock)
10    begin
11        if (flag == 0)
12            begin
13
14                /*
15                 //Fibonacci
16                 mem[0] = 32'b00000000000000000000000000000000;
17                 mem[1] = 32'b00110000000011000000000000000000;
18                 mem[2] = 32'b00110000000011000000000000000001;
19                 mem[3] = 32'b01111000000001000000000000000000;
20                 mem[4] = 32'b00010100001000010000000000000001;
21                 mem[5] = 32'b00110000000001000000000000000001;
22                 mem[6] = 32'b00110000000001100000000000000000;
23                 mem[7] = 32'b011011000110000100000000000010011;
24                 mem[8] = 32'b011011000110000100000000000001100;
25                 mem[9] = 32'b000010000110000000101000000000000;
26                 mem[10] = 32'b00110100000001010000000000000001;
27                 mem[11] = 32'b0011100000000000000000000000000010000;
28                 mem[12] = 32'b000010001100011100101000000000000;
29                 mem[13] = 32'b0000100011100000001100000000000000;
30                 mem[14] = 32'b000010001010000000111000000000000;
31                 mem[15] = 32'b00110100000001010000000000000001;
32                 mem[16] = 32'b1000000000000000000000000000000001;
33                 mem[17] = 32'b000011000110001100000000000000001;
34                 mem[18] = 32'b00111000000000000000000000000000111;
35                 mem[19] = 32'b001011000000101100000000000000001;
36                 mem[20] = 32'b00010101010110000000000000000010;

```



O Algoritmo 4.5 já apresenta instruções pré programadas, utilizadas nos testes das instruções no processador, vistos no final deste relatório.

#### 4.3.4 Program Counter

O Contador de Programa, ou *Program Counter*, responsabilizou-se por dar continuidade ao processamento de instruções em uma sequência linear e crescente ou pulando endereços, dependendo da instrução. Este componente também dependeu do sinal de *clock*.

gerando um endereço de saída diferente a cada pulso, sempre na borda positiva de *clock*. Sua implementação pode ser vista no [Algoritmo 4.6](#).

O *Program Counter* dependeu de alguns sinais de controle e do *clock*, sendo estes sinais de controle advindos da Unidade de Controle ou da entrada do usuário, responsáveis pela sinalização de um *branch* (*PCSrc* e *zero*), *jump* (*Jmp*), *jump* com registrador (*Jr*), parada de execução (*halt*) e o *reset* do processador (*reset*). Caso todos estes sinais se apresentassem baixos, o fluxo normal do programa seria adotado, ou seja, o *Program Counter* adiciona mais 1 ao endereço anterior, dando continuidade ao fluxo e buscando a próxima instrução de forma sequencial. Caso os sinal de controle de *branch* se apresentassem altos, o *Program Counter* pulava para o endereço fornecido pelo imediato da instrução de *branch*; Caso o sinal de *jump* se apresentasse em alta, o *Program Counter* pulava para o valor imediato advindo da instrução de *Jump*; Caso o sinal de *jump* com registrador se apresentasse em alta, o *Program Counter* pulava para o valor advindo do registrador da instrução de *Jump Register*; Caso o sinal de *halt* se apresentasse em alta, o *Program Counter* parava a execução até que este sinal se encontrasse em baixa; Caso o sinal de *reset* se apresentasse em alta, o *Program Counter* começaria a contagem de endereços novamente a partir do 0. Lembrando que o sinal de entrada *mJr* vem do ultimo multiplexador antes do *Program Counter*, ou seja, o ultimo seletor de endereço, visto na [Figura 9](#) no Apêndice 1.

#### Algoritmo 4.6 – Implementação do *Program Counter*

```

1 module ProgramCounter (clock, reset, halt, adressIn, adressOut, PCSrc,
2   zero, Jmp, Jr, mJr);
3
4   input clock, reset, halt, PCSrc, zero, Jmp, Jr;
5   input [31:0] adressIn, mJr;
6   reg [31:0] adressOut;
7   wire [31:0] pcInc, pcIncB; // pcIncB eh o endereco de branch
8   output [31:0] adressOut;
9
10  assign pcInc = adressOut + 1;
11  assign pcIncB = mJr;
12
13  always @(posedge clock)
14    begin
15      if (reset)
16          adressOut <= 0;
17      else if (halt)
18          begin
19          end
20      else if ((PCSrc && zero) || Jmp || Jr)
21          begin
22              adressOut <= pcIncB;
```

```

22           end
23     else
24       addressOut <= pcInc;
25   end
26
27
28 endmodule

```

### 4.3.5 Register Bank

O Banco de Registradores, ou *Register Bank*, encarregou-se de armazenar os 32 registradores de propósito geral utilizados para todas as operações. Implementou-se a organização destes registradores da mesma maneira que a *Data Memory*, ou seja, 32 posições de um vetor com 32 bits cada uma, com cada posição deste vetor correspondendo a um registrador de propósito geral. Por escolha, o primeiro registrador, encontrado na posição 0 e chamado de *registrador zero*, sempre armazenou o valor 0, independente se o usuário tenta modificar este campo, já que a atribuição do valor 0 encontrou-se dentro do bloco *always* ativado pela borda positiva de *clock*, como visto no [Algoritmo 4.7](#).

Este módulo apresentou 4 entradas e 2 saídas, vistas no [Algoritmo 4.7](#). A primeira entrada designou-se para o endereço do primeiro registrador proveniente do campo [25:21] da instrução, a segunda entrada designou-se para o endereço do segundo registrador proveniente do campo [20:16], a terceira entrada designou-se para ser o endereço do registrador de escrita, recebendo a saída do multiplexador anterior ao Banco de Registradores, visto na [Figura 9](#), onde sua saída foi definida para ser o endereço do registrador [20:16] em instruções que utilizam de imediato por exemplo, ou para ser o endereço do registrador [15:11] em instruções lógicas e aritméticas e, por fim, a quarta entrada designou-se para receber os dados a serem escritos no registrador de escrita, provenientes do multiplexador que seleciona a saída da *ALU* ou do *Data Memory*, visto na [Figura 9](#). As duas saídas do *Register Bank* designaram-se para serem os dados de leitura do primeiro e do segundo registrador.

Projetou-se o *Register Bank* para que este fosse síncrono com o *clock*, sempre trabalhando na borda positiva a cada ciclo e para funcionar de acordo com um sinal de seleção designado como *RegWrite*, visto na implementação do *Register Bank* no [Algoritmo 4.7](#). Este sinal de seleção responsabilizou-se por controlar a escrita de dados no *Register Bank*, dependendo de seu sinal. Caso o sinal de *RegWrite* estivesse em alta, o dado seria escrito no endereço do registrador de escrita, fornecido como a terceira entrada ou, caso o sinal de *RegWrite* estivesse em baixa, nada seria escrito no *Register Bank*.

```

1 module RegisterBank (Reg1, Reg2, WriteRegister, WriteData, RegWrite,
2   ReadData1, ReadData2, clock);
3
4   input [4:0] Reg1, Reg2, WriteRegister;
5   input [31:0] WriteData;
6   input RegWrite, clock;
7   output [31:0] ReadData1, ReadData2;
8   reg [31:0] Registers [31:0];
9
10  always @(posedge clock)
11    begin
12      Registers [0] = 32'b0;
13      if (RegWrite)
14        Registers [WriteRegister] = WriteData;
15    end
16
17    assign ReadData1 = Registers [Reg1];
18    assign ReadData2 = Registers [Reg2];
19
endmodule

```

#### 4.3.6 Multiplexadores

Os multiplexadores utilizados para o desenvolvimento deste projeto foram de importância na determinação do caminho seguido pelas instruções pelo caminho de dados, visto no [Figura 9](#). Cada multiplexador presente neste caminho de dados foi implementado em *Verilog* assim como todos os outros módulos e suas implementações podem ser encontradas no Apêndice 1.

O multiplexador, visto no [Figura 11](#), que encontrou-se entre a *Instruction Memory* e o *Register Bank*, responsabilizou-se por selecionar o endereço [20:16] caso seu sinal de controle *RDst* fosse 0 e, caso estivesse em 1, selecionaria o endereço [15:11].

O multiplexador, visto no [Figura 12](#), que encontrou-se entre o *Register Bank* e a *ALU*, responsabilizou-se por selecionar a saída do segundo registrador caso seu sinal de controle *ASrc* fosse 0 e, caso estivesse em 1, selecionaria o valor imediato advindo da instrução.

O multiplexador, visto no [Figura 13](#), que encontrou-se depois do *Data Memory*, responsabilizou-se por selecionar a saída da *ALU* caso seu sinal de controle fosse *MemToReg* 0 e, caso estivesse em 1, selecionaria a saída do *Data Memory*.

Os multiplexadores restantes, presentes no [Figura 14](#), [Figura 15](#) e [Figura 16](#), formaram um efeito cascata, onde a saída de um resultava na entrada do outro, já que estes multiplexadores definiram qual endereço seria encaminhado para o *Program Counter*.

O primeiro multiplexador recebeu o endereço atual e o endereço de *branch* e, caso os sinais de controle *PCSrc* e *Zero*, vistos no [Figura 14](#), estivessem em alta, o endereço de *branch* seria selecionado. No caso do segundo multiplexador, que recebe o endereço de *jump* e o endereço do multiplexador anterior, caso o sinal de controle *Jmp*, visto no [Figura 15](#), estivesse em alta, o valor de *jump* seria selecionado. Por fim, o ultimo multiplexador, que recebe a saída do multiplexador anterior e o valor contido no primeiro registrador, caso seu sinal de controle *JumpReg*, visto no [Figura 16](#), estivesse em alta, o valor contido no registrador seria selecionado para o salto de endereço.

#### 4.3.7 Extensores de Sinal

Estes componentes responsabilizaram-se pela padronização dos campos de entradas dos módulos para 32 *bits*, já que algumas entradas caracterizaram-se por sendo de 16 *bits* (imediato advindo da instrução) ou de 26 *bits* (imediato que indica endereço de *jump*). Assim, a conversão para 32 *bits* foi necessária já que este processador trabalha exclusivamente com dados de 32 *bits*.

Para que um dado de 16 ou 26 *bits* pudesse ser estendido com a finalidade de apresentar 32 *bits*, necessitou-se levar em conta a forma como os números binários são tratados em *Verilog*, ou seja, em complemento de 2. Isto significa que, caso um número binário apresentasse seu *bit* mais significativo sendo 0, bastava adicionar a quantidade restante de *bits* para completar 32 *bits* com mais 0's. Caso o *bit* mais significativo fosse 1, o número binário apresentava sinal negativo então, para que seu sinal e valor fossem carregados sem alteração na hora da extensão, necessitou-se completar os *bits* restantes com 1's. Pode-se ver esta implementação no [Algoritmo 4.8](#).

Algoritmo 4.8 – Implementação do Extensor de 16 para 32 *bits*

```

1 module Extender_16_to_32 (ExIn , ExOut);
2
3     input [15:0] ExIn;
4     output reg [31:0] ExOut;
5
6
7     always @ (*)
8         begin
9             if (ExIn[15])
10                 ExOut = {16'b1111111111111111, ExIn};
11             else
12                 ExOut = {16'b0, ExIn};
13         end
14
15 endmodule

```

### 4.3.8 Entrada e Saída

#### 4.3.8.1 Instruções *In* e *Out*

A entrada e saída do sistema foram tratadas de forma separada das outras instruções, já que para a realização de uma instrução *In* utilizou-se uma interrupção para que o usuário pudesse entrar com os valores nas chaves da placa FPGA e, para uma instrução *Out*, necessitou-se a criação de um módulo separado para que salvasse um valor que seria mostrado continuamente nos *Displays*, como um comando *printf* na linguagem C.

Para a realização de uma instrução *In*, precisou-se da intervenção da Unidade de Controle, que realizaria o trabalho de causar uma interrupção no sistema, através da ativação do sinal *Halt*, capaz de parar a contagem de endereço do *Program Counter* até que este fosse desativado. Para isso, uma *flag* ativada para o usuário indica quando a interrupção deve parar pois todos os dados já foram selecionados nas alavancas das placas. Para que estes dados pudessem ser lidos e salvos pelo sistema, adicionou-se um multiplexador anterior ao extensor de 16 para 32 bits, visto no *datapath* modificado da Figura 10, selecionando a entrada do imediato da instrução ou, no caso de uma instrução *In*, selecionando o valor das alavancas como imediato. Assim, a instrução *In* realizaria uma soma com o valor das alavancas mais o valor do registrador zero, salvando o resultado no registrador de destino, completando a instrução.

Para a realização de uma instrução *Out*, criou-se um módulo de saída para que o valor a ser mostrado nos *Displays* pudesse estar ali de forma contínua, visto na Figura 10, sem que fosse reescrito a cada ciclo de *clock*. Assim, este módulo possui uma memória de 1024 bits para armazenar valores e, para que um valor possa ser armazenado, é necessária uma instrução *Store Word*, salvando o valor no *Data Memory* e no módulo de saída simultaneamente no mesmo endereço em ambos, já que possuem a mesma quantidade de memória nas mesmas posições. Assim, quando uma instrução *Out* é realizada, o módulo de saída manda constantemente o valor ali salvo como *output*, mantendo os *Displays* ativos através dos ciclos de *clock*, já que a Unidade de Controle muda o valor de *outenable* para 1 quando uma instrução *Out* é realizada. Com uma instrução de saída, a Unidade de Controle também tem o papel de parar o sistema para que o usuário veja a saída, já que com o uso do *clock* automático, não seria possível ver a saída. Além disso, quando o sinal de *reset* é ativado, o valor de saída deste módulo será o valor zero, para assim voltar os *displays* para o estado original, com o valor zero em todos. A implementação deste módulo é vista no Algoritmo 4.9.

Algoritmo 4.9 – Implementação do módulo de saída

```

1 module Output_Module (clock, adress, writedata, dataout, out, MemWrite,
2   rst);
3   input clock, out, MemWrite, rst;

```

```

4      input [4:0] adress;
5      input [31:0] writedata;
6      output reg [31:0] dataout;
7      reg [31:0] mem [31:0];
8
9      always @(posedge clock)
10     begin
11         mem[0] = 32'b0;
12         if (MemWrite)
13             mem[adress] = writedata;
14         else if (out)
15             dataout = mem[adress];
16         else if (rst)
17             dataout = mem[0];
18     end
19
20 endmodule

```

#### 4.3.8.2 Displays de 7 segmentos

Por fim, para que os resultados pudessem ser mostrados nos *Displays* da placa FPGA, necessitou-se a criação de um módulo que transformasse um número binário de 4 *bits*, com valor decimal até 9, em uma saída de 7 *bits* interpretada pelo *Display* da placa. Assim, utilizou-se do módulo conversos para *Display* de 7 segmentos feito na unidade curricular passada, visto no [Figura 17](#), Apêndice 1.

Porém, como todo dado emitido pelo processador foi de 32 *bits*, não foi possível a representação direta utilizando o *Display*, por isso utilizou-se de um conversor de binário para *BCD* para que cada casa do número pudesse ser mostrada em cada *Display* disponível. Por exemplo, caso queira-se mostrar o número 1530, o primeiro *Display* deve mostrar o número 0, o segundo o número 3, o terceiro o número 5 e o quarto o número 1. Para isso, utilizou-se da implementação em *Verilog* de um conversor binário para *BCD* de fonte não própria, advinda do site [7], onde o código pode ser visto no [Algoritmo 4.10](#).

Algoritmo 4.10 – Implementação do conversor de binário para *BCD*

```

1 module Binary_to_BCD(binary, neg, first, second, third, fourth, fifth,
2                         sixth, seventh, eighth);
3
4     input [31:0] binary;
5     reg [31:0] binary_out;
6     output reg [3:0] first, second, third, fourth, fifth, sixth,
7                     seventh, eighth;
8     output reg neg;
9
10    integer i;

```

```
9      always@(binary)
10     begin
11
12         if(binary[7])
13             begin
14                 binary_out = ~binary + 1;
15                 neg = 1;
16             end
17         else
18             begin
19                 binary_out = binary;
20                 neg = 0;
21             end
22
23         first = 4'd0;
24         second = 4'd0;
25         third = 4'd0;
26         fourth = 4'd0;
27         fifth = 4'd0;
28         sixth = 4'd0;
29         seventh = 4'd0;
30         eighth = 4'd0;
31
32         for(i=31; i>=0; i=i-1)
33             begin
34                 if(eighth >= 5)
35                     eighth = eighth + 4'd3;
36                 if(seventh >= 5)
37                     seventh = seventh + 4'd3;
38                 if(sixth >= 5)
39                     sixth = sixth + 4'd3;
40                 if(fifth >= 5)
41                     fifth = fifth + 4'd3;
42                 if(fourth >= 5)
43                     fourth = fourth + 4'd3;
44                 if(third >= 5)
45                     third = third + 4'd3;
46                 if(second >= 5)
47                     second = second + 4'd3;
48                 if(first >= 5)
49                     first = first + 4'd3;
50
51                 eighth = eighth << 1;
52                 eighth[0] = seventh[3];
53                 seventh = seventh << 1;
54                 seventh[0] = sixth[3];
55                 sixth = sixth << 1;
```

```

56         sixth[0] = fifth[3];
57         fifth = fifth << 1;
58         fifth[0] = fourth[3];
59         fourth = fourth << 1;
60         fourth[0] = third[3];
61         third = third << 1;
62         third[0] = second[3];
63         second = second << 1;
64         second[0] = first[3];
65         first = first << 1;
66         first[0] = binary_out[i];
67     end
68 end
69
70 endmodule

```

Este tipo de conversor não levou em conta números negativos, portanto necessitou-se realizar algumas modificações no código, mais precisamente depois do primeiro *begin* do bloco *always*, adicionando algumas de linhas de código responsáveis por identificar um número negativo e transformá-lo de complemento de 2 para sua forma binária original e, para isso, também necessitou-se de uma pequena alteração na ultima linha de código, já presentes neste código.

Assim, o módulo contendo todos os *Displays* com suas representações decimais, foi representado no [Algoritmo 4.11](#).

Algoritmo 4.11 – Todos os *Displays*

```

1 module Displays_Final (binary, um, cem, mil, neg, milhao, bilhao, trilhao
, quadrilhao, neg, out);
2
3     input [31:0] binary;
4     output neg;
5     input out;
6     output [6:0] um, cem, mil, milhao, bilhao, trilhao, quadrilhao;
7     wire [3:0] binUm, binCem, binMil, binMilhao, binBilhao,
8     binTrilhao, binQuadrilhao;
9
10    Binary_to_BCD BTB (binary[31:0], neg, binUm[3:0], binCem[3:0],
11    binMil[3:0], binMilhao[3:0], binBilhao[3:0], binTrilhao[3:0],
12    binQuadrilhao[3:0]);
13
14    Display U1 (binUm[3:0], um[6:0], out);
15    Display U2 (binCem[3:0], cem[6:0], out);
16    Display U3 (binMil[3:0], mil[6:0], out);
17    Display U4 (binMilhao[3:0], milhao[6:0], out);
18    Display U5 (binBilhao[3:0], bilhao[6:0], out);

```

```

16     Display U6 (binTrilhao[3:0], trilhao[6:0], out);
17     Display U7 (binQuadrilhao[3:0], quadrilhao[6:0], out);
18
19 endmodule

```

#### 4.3.8.3 Tratamento do *clock* e dos botões de entrada da FPGA

Para que o *clock* automático pudesse ser utilizado, um temporizador foi utilizado para que sua frequência não fosse tão alta e causasse problemas, diminuindo de 50Mhz para uma frequência rápida, mas observável e controlável. Assim, a implementação vista no [Algoritmo 4.12](#) divide o *clock* automático para que este possa ser visível na placa e ao mesmo tempo possa ser rápido o suficiente sem causar problemas em pular instruções de *input* e *output* seguidas.

Algoritmo 4.12 – Implementação do Temporizador

```

1 module Temporizador(clk_auto, clk);
2     input clk_auto;
3     output reg clk;
4     reg[24:0] cont;
5
6     always@(posedge clk_auto)
7         begin
8             if(cont==5_000_000)
9                 begin
10                 cont <= 0;
11                 clk = ~clk;
12             end
13         else
14             cont <= cont + 1; //1'b1
15         end
16
17 endmodule

```

Além deste módulo necessário, adicionou-se também um *Debouncer* para o uso dos botões da placa FPGA, já que estes apresentam trepidações que devem ser removidas para seu funcionamento correto. A implementação deste módulo pode ser vista no [Algoritmo 4.13](#).

Algoritmo 4.13 – Implementação do *Debouncer*

```

1 module DeBounce_v
2     (
3         input                      clk, n_reset, button_in,
4         output reg        DB_out
5     );
6     parameter N = 11 ;

```

```
7      reg [N-1 : 0] q_reg;
8      reg [N-1 : 0] q_next;
9      reg DFF1, DFF2;
10     wire q_add;
11     wire q_reset;
12
13     assign q_reset = (DFF1 ^ DFF2);
14     assign q_add = ~(q_reg[N-1]);
15
16     always @ ( q_reset, q_add, q_reg)
17         begin
18             case( {q_reset , q_add})
19                 2'b00 :
20                     q_next <= q_reg;
21                 2'b01 :
22                     q_next <= q_reg + 1;
23                 default :
24                     q_next <= { N {1'b0} };
25             endcase
26         end
27
28     always @ ( posedge clk )
29         begin
30             if(n_reset == 1'b0)
31                 begin
32                     DFF1 <= 1'b0;
33                     DFF2 <= 1'b0;
34                     q_reg <= { N {1'b0} };
35                 end
36             else
37                 begin
38                     DFF1 <= button_in;
39                     DFF2 <= DFF1;
40                     q_reg <= q_next;
41                 end
42         end
43
44     always @ ( posedge clk )
45         begin
46             if(q_reg[N-1] == 1'b1)
47                 DB_out <= DFF2;
48             else
49                 DB_out <= DB_out;
50         end
51
52     endmodule
```

Vale lembrar que o algoritmo *Debounce* não é de autoria própria, sendo este advindo do site *eewiki*[8].

## 4.4 A Unidade de Controle

A Unidade de Controle foi o componente encarregado de fazer com que as instruções seguissem o caminho de dados de forma correta para seu funcionamento. Para isso, utilizou-se de sinais de controle específicos de cada módulo responsáveis pelo funcionamento correto dependendo da instrução. Estes sinais podem ser vistos na [Tabela 4](#).

Tabela 4 – Tabela de sinais de controle

Sinais de Controle	Módulo	Função
Rw	<i>Register Bank</i>	Escrita de Registrador
Mw	<i>Data Memory/Output Module</i>	Escrita na Memória de dados e <i>Output</i>
RDst	<i>Mux Instrução</i>	Escolhe os registradores mandados
Asrc	<i>Mux ALU</i>	Imediato ou registrador 2
Mtg	<i>Mux Data Memory</i>	Resultado da <i>ALU</i> ou <i>Data Memory</i>
PSrc	<i>Mux Branch</i>	Receber endereço de <i>branch</i> ou não
Jmp	<i>Mux Jump</i>	Receber endereço de <i>jump</i> ou não
Jr	<i>Mux Jump Register</i>	Receber Endereço de <i>Jump Reg</i> ou não
ALUop	<i>ALU</i>	<i>OPcode</i> da <i>ALU</i>
Out	<i>Output Module</i>	Mostrar o conteúdo do módulo no <i>displays</i>
Mo	<i>Mux Switches</i>	Imediato da instrução ou <i>switches</i>
Halt	<i>Program Counter</i>	Para ou não a execução do processador

Fonte: Autoria própria

Cada sinal pode apresentar o valor 0 ou 1, exceto o *ALUop*, que pode apresentar outros valores de vistos na [Tabela 3](#). Tendo em vista isso, a Unidade de Controle foi implementada visando seguir estes sinais. Este módulo foi desenvolvido com um bloco *always* e, dentro deste, um *switch case* para cada instrução possível. Em quase todas as instruções os procedimentos foram iguais, com a mudança dos sinais de acordo com a necessidade, mas as instruções *In* e *Out* apresentaram uma particularidade relacionada à interrupção mencionada anteriormente. Quando o sinal chamado de *flag* apresenta valor 1, o processador para as operações e quando seu valor é 0, o processador retoma as operações, parando com a interrupção necessária para a entrada de dados ou para a observação dos dados de saída.

Desenvolveu-se o bloco *always* para que este fosse sensível à mudança do *OPcode* e da *flag* de entrada e saída, para que assim mudasse os sinais de controle apenas com a mudança do *OPcode* e da *flag*.

A partir disto, percebeu-se que a Unidade de Controle caracterizou-se por ser do

tipo *Hardwired*, ou seja, um tipo de Unidade de Controle que apresenta pouca flexibilidade mas grande velocidade, ideal para um processador do tipo *MIPS*, no caso o processador em questão. A implementação deste módulo pode ser vista na [Algoritmo 4.14](#).

Algoritmo 4.14 – Implementação da Unidade de Controle

```

1 module ControladorHumano (OPcode, RW, MW, RDst, ASrc, MTG, PSrc, Jmp, Jr
, ALUop, halt, flag, in, out, MO);
2
3     input [5:0] OPcode;
4     input flag;
5     output reg RW, MW, RDst, ASrc, MTG, PSrc, Jmp, Jr, halt, out, MO
;
6     output reg [4:0] ALUop;
7
8     always @ (OPcode or flag)
9         begin
10             case (OPcode)
11                 6'b000000: // nop
12                     begin
13                         RW = 1'b0;
14                         MW = 1'b0;
15                         RDst = 1'b0;
16                         ASrc = 1'b0;
17                         MTG = 1'b0;
18                         PSrc = 1'b0;
19                         Jmp = 1'b0;
20                         Jr = 1'b0;
21                         ALUop = 5'bxxxxx;
22                         out = 1'b0;
23                         MO = 1'b0;
24                         halt = 1'b0;
25                     end
26                 6'b000010: // add
27                     begin
28                         RW = 1'b1;
29                         MW = 1'b0;
30                         RDst = 1'b1;
31                         ASrc = 1'b0;
32                         MTG = 1'b0;
33                         PSrc = 1'b0;
34                         Jmp = 1'b0;
35                         Jr = 1'b0;
36                         ALUop = 5'b00000;
37                         out = 1'b0;
38                         MO = 1'b0;
39                         halt = 1'b0;
40                     end

```

```
41          6'b000011: // addi
42              begin
43                  RW = 1'b1;
44                  MW = 1'b0;
45                  RDst = 1'b0;
46                  ASrc = 1'b1;
47                  MTG = 1'b0;
48                  PSrc = 1'b0;
49                  Jmp = 1'b0;
50                  Jr = 1'b0;
51                  ALUop = 5'b00000;
52                  out = 1'b0;
53                  MO = 1'b0;
54                  halt = 1'b0;
55          end
56          6'b000100: // sub
57              begin
58                  RW = 1'b1;
59                  MW = 1'b0;
60                  RDst = 1'b1;
61                  ASrc = 1'b0;
62                  MTG = 1'b0;
63                  PSrc = 1'b0;
64                  Jmp = 1'b0;
65                  Jr = 1'b0;
66                  ALUop = 5'b00001;
67                  out = 1'b0;
68                  MO = 1'b0;
69                  halt = 1'b0;
70          end
71          6'b000101: // subi
72              begin
73                  RW = 1'b1;
74                  MW = 1'b0;
75                  RDst = 1'b0;
76                  ASrc = 1'b1;
77                  MTG = 1'b0;
78                  PSrc = 1'b0;
79                  Jmp = 1'b0;
80                  Jr = 1'b0;
81                  ALUop = 5'b00001;
82                  out = 1'b0;
83                  MO = 1'b0;
84                  halt = 1'b0;
85          end
86          6'b000110: // mult
87              begin
```

```

88          RW = 1'b1;
89          MW = 1'b0;
90          RDst = 1'b1;
91          ASrc = 1'b0;
92          MTG = 1'b0;
93          PSrc = 1'b0;
94          Jmp = 1'b0;
95          Jr = 1'b0;
96          ALUop = 5'b000010;
97          out = 1'b0;
98          MO = 1'b0;
99          halt = 1'b0;
100         end
101         6'b000111: // div
102         begin
103             RW = 1'b1;
104             MW = 1'b0;
105             RDst = 1'b1;
106             ASrc = 1'b0;
107             MTG = 1'b0;
108             PSrc = 1'b0;
109             Jmp = 1'b0;
110             Jr = 1'b0;
111             ALUop = 5'b000011;
112             out = 1'b0;
113             MO = 1'b0;
114             halt = 1'b0;
115         end
116         6'b001000: // slt
117         begin
118             RW = 1'b1;
119             MW = 1'b0;
120             RDst = 1'b1;
121             ASrc = 1'b0;
122             MTG = 1'b0;
123             PSrc = 1'b0;
124             Jmp = 1'b0;
125             Jr = 1'b0;
126             ALUop = 5'b01010;
127             out = 1'b0;
128             MO = 1'b0;
129             halt = 1'b0;
130         end
131         6'b001001: // shr
132         begin
133             RW = 1'b1;
134             MW = 1'b0;

```

```

135                               RDst = 1'b0;
136                               ASrc = 1'bx;
137                               MTG = 1'b0;
138                               PSrc = 1'b0;
139                               Jmp = 1'b0;
140                               Jr = 1'b0;
141                               ALUop = 5'b01001;
142                               out = 1'b0;
143                               MO = 1'b0;
144                               halt = 1'b0;
145                               end
146           6'b0001010: // shl
147           begin
148               RW = 1'b1;
149               MW = 1'b0;
150               RDst = 1'b0;
151               ASrc = 1'bx;
152               MTG = 1'b0;
153               PSrc = 1'b0;
154               Jmp = 1'b0;
155               Jr = 1'b0;
156               ALUop = 5'b01000;
157               out = 1'b0;
158               MO = 1'b0;
159               halt = 1'b0;
160           end
161           6'b0001011: // lw
162           begin
163               RW = 1'b1;
164               MW = 1'b0;
165               RDst = 1'b0;
166               ASrc = 1'b1;
167               MTG = 1'b1;
168               PSrc = 1'b0;
169               Jmp = 1'b0;
170               Jr = 1'b0;
171               ALUop = 5'b00000;
172               out = 1'b0;
173               MO = 1'b0;
174               halt = 1'b0;
175           end
176           6'b0001100: // li
177           begin
178               RW = 1'b1;
179               MW = 1'b0;
180               RDst = 1'b0;
181               ASrc = 1'b1;

```

```

182          MTG = 1'b0;
183          PSrc = 1'b0;
184          Jmp = 1'b0;
185          Jr = 1'b0;
186          ALUop = 5'b00000;
187          out = 1'b0;
188          MO = 1'b0;
189          halt = 1'b0;
190      end
191 6'b0001101: // sw
192  begin
193          RW = 1'b0;
194          MW = 1'b1;
195          RDst = 1'b0;
196          ASrc = 1'b1;
197          MTG = 1'bx;
198          PSrc = 1'b0;
199          Jmp = 1'b0;
200          Jr = 1'b0;
201          ALUop = 5'b00000;
202          out = 1'b0;
203          MO = 1'b0;
204          halt = 1'b0;
205      end
206 6'b0001110: // jump
207  begin
208          RW = 1'b0;
209          MW = 1'b0;
210          RDst = 1'b0;
211          ASrc = 1'b1;
212          MTG = 1'bx;
213          PSrc = 1'b0;
214          Jmp = 1'b1;
215          Jr = 1'b0;
216          ALUop = 5'bxxxxxx;
217          out = 1'b0;
218          MO = 1'b0;
219          halt = 1'b0;
220      end
221 6'b0001111: // jump register
222  begin
223          RW = 1'b0;
224          MW = 1'b0;
225          RDst = 1'b0;
226          ASrc = 1'b0;
227          MTG = 1'b0;
228          PSrc = 1'b0;

```

```

229          Jmp = 1'b0;
230          Jr = 1'b1;
231          ALUop = 5'bxxxxx;
232          out = 1'b0;
233          MO = 1'b0;
234          halt = 1'b0;
235      end
236 6'b010000: // xor
237      begin
238          RW = 1'b1;
239          MW = 1'b0;
240          RDst = 1'b1;
241          ASrc = 1'b0;
242          MTG = 1'b0;
243          PSrc = 1'b0;
244          Jmp = 1'b0;
245          Jr = 1'b0;
246          ALUop = 5'b00111;
247          out = 1'b0;
248          MO = 1'b0;
249          halt = 1'b0;
250      end
251 6'b010001: // and
252      begin
253          RW = 1'b1;
254          MW = 1'b0;
255          RDst = 1'b1;
256          ASrc = 1'b0;
257          MTG = 1'b0;
258          PSrc = 1'b0;
259          Jmp = 1'b0;
260          Jr = 1'b0;
261          ALUop = 5'b00101;
262          out = 1'b0;
263          MO = 1'b0;
264          halt = 1'b0;
265      end
266 6'b010010: // or
267      begin
268          RW = 1'b1;
269          MW = 1'b0;
270          RDst = 1'b1;
271          ASrc = 1'b0;
272          MTG = 1'b0;
273          PSrc = 1'b0;
274          Jmp = 1'b0;
275          Jr = 1'b0;

```

```

276          ALUop = 5'b00110;
277          out = 1'b0;
278          MO = 1'b0;
279          halt = 1'b0;
280      end
281  6'b010011: // not
282      begin
283          RW = 1'b1;
284          MW = 1'b0;
285          RDst = 1'b0;
286          ASrc = 1'b0;
287          MTG = 1'b0;
288          PSrc = 1'b0;
289          Jmp = 1'b0;
290          Jr = 1'b0;
291          ALUop = 5'b00100;
292          out = 1'b0;
293          MO = 1'b0;
294          halt = 1'b0;
295      end
296  6'b010100: // xor
297      begin
298          RW = 1'b1;
299          MW = 1'b0;
300          RDst = 1'b0;
301          ASrc = 1'b1;
302          MTG = 1'b0;
303          PSrc = 1'b0;
304          Jmp = 1'b0;
305          Jr = 1'b0;
306          ALUop = 5'b00111;
307          out = 1'b0;
308          MO = 1'b0;
309          halt = 1'b0;
310      end
311  6'b010101: // andi
312      begin
313          RW = 1'b1;
314          MW = 1'b0;
315          RDst = 1'b0;
316          ASrc = 1'b1;
317          MTG = 1'b0;
318          PSrc = 1'b0;
319          Jmp = 1'b0;
320          Jr = 1'b0;
321          ALUop = 5'b00101;
322          out = 1'b0;

```

```

323                               MO = 1'b0;
324                               halt = 1'b0;
325                               end
326   6'b010110: // ori
327       begin
328           RW = 1'b1;
329           MW = 1'b0;
330           RDst = 1'b0;
331           ASrc = 1'b1;
332           MTG = 1'b0;
333           PSrc = 1'b0;
334           Jmp = 1'b0;
335           Jr = 1'b0;
336           ALUop = 5'b00110;
337           out = 1'b0;
338           MO = 1'b0;
339           halt = 1'b0;
340       end
341   6'b010111: // beq
342       begin
343           RW = 1'b0;
344           MW = 1'b0;
345           RDst = 1'b0;
346           ASrc = 1'b0;
347           MTG = 1'b0;
348           PSrc = 1'b1;
349           Jmp = 1'b0;
350           Jr = 1'b0;
351           ALUop = 5'b01111;
352           out = 1'b0;
353           MO = 1'b0;
354           halt = 1'b0;
355       end
356   6'b011000: // bneq
357       begin
358           RW = 1'b0;
359           MW = 1'b0;
360           RDst = 1'b0;
361           ASrc = 1'b0;
362           MTG = 1'b0;
363           PSrc = 1'b1;
364           Jmp = 1'b0;
365           Jr = 1'b0;
366           ALUop = 5'b01100;
367           out = 1'b0;
368           MO = 1'b0;
369           halt = 1'b0;

```

```

370           end
371           6'b011001: // beqz
372             begin
373               RW = 1'b0;
374               MW = 1'b0;
375               RDst = 1'b0;
376               ASrc = 1'b0;
377               MTG = 1'b0;
378               PSrc = 1'b1;
379               Jmp = 1'b0;
380               Jr = 1'b0;
381               ALUop = 5'b01111;
382               out = 1'b0;
383               MO = 1'b0;
384               halt = 1'b0;
385             end
386           6'b011010: // bneqz
387             begin
388               RW = 1'b0;
389               MW = 1'b0;
390               RDst = 1'b0;
391               ASrc = 1'b0;
392               MTG = 1'b0;
393               PSrc = 1'b1;
394               Jmp = 1'b0;
395               Jr = 1'b0;
396               ALUop = 5'b01100;
397               out = 1'b0;
398               MO = 1'b0;
399               halt = 1'b0;
400             end
401           6'b011011: // bgt
402             begin
403               RW = 1'b0;
404               MW = 1'b0;
405               RDst = 1'b0;
406               ASrc = 1'b0;
407               MTG = 1'b0;
408               PSrc = 1'b1;
409               Jmp = 1'b0;
410               Jr = 1'b0;
411               ALUop = 5'b01010;
412               out = 1'b0;
413               MO = 1'b0;
414               halt = 1'b0;
415             end
416           6'b011100: // blt

```

```

417      begin
418          RW = 1'b0;
419          MW = 1'b0;
420          RDst = 1'b0;
421          ASrc = 1'b0;
422          MTG = 1'b0;
423          PSrc = 1'b1;
424          Jmp = 1'b0;
425          Jr = 1'b0;
426          ALUop = 5'b01011;
427          out = 1'b0;
428          MO = 1'b0;
429          halt = 1'b0;
430      end
431      6'b011101: // bgtz
432      begin
433          RW = 1'b0;
434          MW = 1'b0;
435          RDst = 1'b0;
436          ASrc = 1'b0;
437          MTG = 1'b0;
438          PSrc = 1'b1;
439          Jmp = 1'b0;
440          Jr = 1'b0;
441          ALUop = 5'b01010;
442          out = 1'b0;
443          MO = 1'b0;
444          halt = 1'b0;
445      end
446      6'b011110: // bltz
447      begin
448          RW = 1'b0;
449          MW = 1'b0;
450          RDst = 1'b0;
451          ASrc = 1'b0;
452          MTG = 1'b0;
453          PSrc = 1'b1;
454          Jmp = 1'b0;
455          Jr = 1'b0;
456          ALUop = 5'b01011;
457          out = 1'b0;
458          MO = 1'b0;
459          halt = 1'b0;
460      end
461      6'b011111: // in
462      begin
463          RW = 1'b1;

```

```

464           MW = 1'b0;
465           RDst = 1'b0;
466           ASrc = 1'b1;
467           MTG = 1'b0;
468           PSrc = 1'b0;
469           Jmp = 1'b0;
470           Jr = 1'b0;
471           ALUop = 5'b00000;
472           out = 1'b0;
473           MO = 1'b1;
474           if (flag)
475               halt = 1'b1;
476           else
477               halt = 1'b0;
478       end
479   6'b100000: //out
480       begin
481           RW = 1'b0;
482           MW = 1'b0;
483           RDst = 1'b0;
484           ASrc = 1'b1;
485           MTG = 1'b0;
486           PSrc = 1'b0;
487           Jmp = 1'b0;
488           Jr = 1'b0;
489           ALUop = 5'b00000;
490           out = 1'b1;
491           MO = 1'b0;
492           if (flag)
493               halt = 1'b1;
494           else
495               halt = 1'b0;
496       end
497   default:
498       begin
499           RW = 1'b0;
500           MW = 1'b0;
501           RDst = 1'b0;
502           ASrc = 1'b0;
503           MTG = 1'b0;
504           PSrc = 1'b0;
505           Jmp = 1'b0;
506           Jr = 1'b0;
507           ALUop = 5'b00000;
508           out = 1'b0;
509           MO = 1'b0;
510           halt = 1'b0;

```

```

511           end
512       endcase
513   end
514 endmodule

```

Tabela 5 – Tipos de Instrução

	Campos					
Tipo 1	OPcode	R3	R1	R2	shift	-
Tipo 2	OPcode	R3	R1	Imediato/Endereço		
Tipo 3	OPcode	Endereço				

Fonte: Autoria Própria

Cada instrução modificou de certa maneira os sinais de controle. A instrução *Nop* mudou todos os sinais de controle para o valor 0, já que esta representa um ciclo vazio que não deve realizar nenhuma operação. Em todas as instruções do tipo 1 (Tabela 5), o sinal *RW* apresentou-se em alta, já que são instruções que precisam gravar dados nos registradores. Como esse tipo de instrução engloba tanto operações com registradores e operações com de deslocamento de *bits*, a única diferença apresentada está no sinal *RDst* e *ASrc*. Em instruções de *shift* o sinal *RDst* deve estar em baixa e *ASrc* não importa neste ocasião, já que o valor de deslocamento alimenta a *ALU* diretamente. Em operações de soma por exemplo, o sinal *RDst* deve estar em alta pois existem 3 registradores de operação, assim o multiplexador deve selecionar o terceiro como registrador de imediato.

Operações com imediatos caracterizaram-se por serem do tipo 2 (Tabela 5), o multiplexador anterior a *ALU* deve selecionar o valor imediato, por isso *ASrc* está em alta, e *RDst* deve estar em baixa pois não existem três registradores para operações com imediato, assim ele deve selecionar o segundo registrador passado como o de destino. Em operações sem imediato o contrário acontece. Como o tipo 2 também engloba instruções de saltos condicionais, apenas o sinal *PSrc* deve estar em alta, já que este sinal determina que o endereço provido pela instrução de salto condicional deve ser utilizado pelo multiplexador de endereços e passado para o *Program Counter* e os demais sinais devem estar em baixa, já que nenhuma outra operação deve ser realizada. Como um salto condicional depende do resultado de uma comparação feita pela *ALU*, o *ALUop* muda para cada tipo de *branch*, já que cada salto condicional depende de uma comparação diferente.

Além destas, as instruções de movimentação de dados também estão presentes no tipo 2. Estas são instruções que utilizaram de sinais diferentes das anteriores. No caso de uma instrução *Load Word*, o sinal *RW* está em alta pois o valor carregado da memória deve ser armazenado em algum registrador e os sinais *ASrc* e *MTG* devem estar em alta pois esta instrução necessita do valor de imediato para cálculo de endereço, por isso o primeiro

sinal em alta, e precisa que o valor advindo da memória de instruções seja gravado no Banco de Registradores, assim o último sinal deve estar em alta para que o multiplexador que localiza-se depois deste módulo selecione este valor.

Por fim, o tipo 3 ([Tabela 5](#)) foi designado para as instruções *Jump* e *Halt*. Para o *Jump*, o sinal *Jmp* está em alta para que o Contador de Programa receba o valor de salto. Para a instrução *Halt*, os sinais não importam pois sua única função é parar o processador, assim apenas o sinal *halt* fica em alta.

## 4.5 Módulo Final

Com ambas as Unidade de Controle e Unidade de Processamento feitas, bastou realizar a interligação entre todos os componentes em um único módulo. Essa implementação pode ser vista no [Algoritmo 4.15](#).

Algoritmo 4.15 – Módulo Final

```

1 module FX      (botaoclk ,
2                      clk_automatico ,
3                      switches ,
4                      out ,
5                      um ,
6                      cem ,
7                      rst ,
8                      mil ,
9                      neg ,
10                     milhao ,
11                     bilhao ,
12                     trilhao ,
13                     quadrilhao ,
14                     hlt ,
15                     oflow_add ,
16                     oflow_sub);
17
18     wire clk, botaosaida;
19     input wire [15:0] switches;
20     input clk_automatico, botaoclk;
21
22     output [6:0] um, cem, mil, milhao, bilhao, trilhao, quadrilhao;
23     output neg;
24
25     output wire out;
26     input rst;
27     output wire hlt;
28     wire [31:0] Instruction;
29     wire [9:0] adIn;
```



```

75                                         .RegWrite(rw),
76                                         .ReadData1(data1),
77                                         .ReadData2(data2),
78                                         .clock(clk));
79
80     ALU_OP (.OPcode(ALUop),
81               .op1(data1),
82               .op2(mAlu),
83               .result(ALUres),
84               .shamt(Instruction
85               [10:6]),
86               .zero(zero),
87               .overflow_add(oflow_add
88               ),
89               .overflow_sub(oflow_sub
90               ),
91               .clock(clk));
92
93     DataMemory_DM (.clock(clk),
94                     .WriteData(data2),
95                     .MemWrite(mw),
96                     .DataMemoryOut(rdData),
97                     .adress(ALUres));
98
99     MuxInstruction MI (.Reg1(Instruction[20:16]),
100                      .Reg2(Instruction
101                      [15:11]),
102                      .RegOut(mInstr),
103                      .RegDst(Rdst));
104
105     MuxAlu MA (.Imediate(ex16),
106                  .Reg(data2),
107                  .MuxOut(mAlu),
108                  .ALUSrc(ASrc));
109
110     MuxMem MM (.ReadData(rdData),
111                  .ALUresult(ALUres),
112                  .MemToReg(mtg),
113                  .MuxOut(mMem));
114
115     MuxPC MPC (.PCIInc(adOut),
116                  .JumpBranch(ex16),
117                  .PCSrc(PSrc),
118                  .Zero(zero),
119                  .MuxOut(mPC));
120
121     MuxJump MJMP (.MPC(mPC),
122                    .JumpTarget(jmp));

```

```

118                               .JumpAddress(
119           ex26),
120                               .Jmp(Jmp),
121           ;
122           MuxJr  MJR
123                               (.MJ(mJump),
124                               .Reg(data1),
125                               .JumpReg(Jr),
126                               .MuxOut(mJr));
127           MuxIm  MIM
128                               (.Immediate(Instruction[15:0]),
129                               .Switches(switches),
130                               .MuxOut(mMim),
131                               .MO(MO));
132           Output_Module OM
133                               (.clock(clk),
134                               .adress(ALUres),
135                               .writedata(data2),
136                               .dataout(do),
137                               .out(out),
138                               .MemWrite(mw),
139                               .rst(rst));
140           Displays_Final DF
141                               (.binary(do),
142                               .um(um),
143                               .cem(cem),
144                               .mil(mil),
145                               .milhao(milhao1),
146                               .bilhao(bilhao1),
147                               .trilhao(trilhao1),
148                               .quadrilhao(quadrilhaoo1
149           ),
150                               .neg(neg),
151                               .out(out));
152           Extender_16_to_32 EXT1
153                               (.ExIn(mMim),
154                               .ExOut(ex16));
155           Extender_26_to_32 EXT2
156                               (.ExtenderIn(Instruction[25:0]),
157                               .ExtenderOut(
158           ex26));
159           ControladorHumano CH
160                               (.OPcode(Instruction[31:26]),
161                               .RW(rw),
162                               .MW(mw),
163                               .RDst(Rdst),
164

```

```

161
162
163
164
165
166
167
168
),
169
170
171
172
173     DeBounce_v  DBCE      (.clk(clk),
174
175
176
177
178     Temporizador TMP    (.clk_auto(clk_automatico),
179
180
181 endmodule

```



## 5 Resultados Obtidos e Discussões

Como a Unidade de Controle foi o foco deste projeto, para a conclusão dos testes, realizou-se a união de todos os componentes de maneira que os testes pudessem ser realizados de forma correta.

Como visto no [Algoritmo 4.5](#), 3 algoritmos distintos foram criados para os testes. Primeiramente, o algoritmo de Fibonacci foi testado na placa FPGA com *clock* manual e seu resultado foi comprovado, como visto na [Figura 18](#), [Figura 19](#), [Figura 20](#), [Figura 21](#), [Figura 22](#) e [Figura 23](#), presentes no Apêndice 2. Para a realização deste algoritmo, converteu-se um código em C para a linguagem binária deste processador, utilizando das instruções de *Input*, *Output*, *Add*, *Sub*, *Branches*, *Lw* e *Sw*. Como o algoritmo necessita de um laço responsável por realizar os cálculos da sequência de Fibonacci e mostrar cada termo, necessitou-se do uso de saltos condicionais e saltos diretos, ou seja, *Branches* e *Jumps*. Assim, com uma instrução de *Input*, o usuário escolhe quantos termos serão apresentados. O algoritmo correspondente em C pode ser visto no [Algoritmo 5.1](#).

Algoritmo 5.1 – Sequência de Fibonacci na Linguagem C

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int n, first = 0, second = 1, next, c;
6
7     printf("Entre com o numero de termos\n");
8     scanf("%d", &n);
9
10    printf("Os primeiros %d termos da sequencia de Fibonacci sao:\n", n);
11
12    for ( c = 0 ; c < n ; c++ )
13    {
14        if ( c <= 1 )
15            next = c;
16        else
17        {
18            next = first + second;
19            first = second;
20            second = next;
21        }
22        printf("%d\n", proximo);
23    }
24
25    return 0;

```

26

Para realizar uma operação *scanf*, utilizou-se da instrução *Input*, recebendo a quantidade de termos da sequência a serem calculados e mostrados. Para que os termos aparecessem nos *displays* da placa, utilizou-se da instrução de *Output*. Para os comandos *for* e *if/else* das linhas 12, 14 e 16 do [Algoritmo 5.1](#), foram utilizadas as instruções, respectivamente, *Blt* (*Branch less than*) e *Slt* (*Set less than*). Implicitamente, para manter o laço repetição, quando um escopo fosse alcançado, ou seja, uma chave como a da linha 23, uma instrução *Jump* era utilizada para retornar para a linha 12 novamente, atualizando e verificando a condição de laço.

O segundo algoritmo testado foi o *Insertion Sort*. Novamente, o algoritmo correspondente em C, visto no [Algoritmo 5.2](#) foi traduzido para a linguagem binária do processador. Para este, utilizou-se o *clock* automático da FPGA, controlado pelo temporizador, já que sua complexidade é alta, levando muito tempo para ser testado com o *clock* manual. Assim como no algoritmo anterior, necessitou-se a criação de laços de repetição, utilizando assim de saltos e saltos condicionais. Para realizar-se a representação de vetores como na linguagem C, salvou-se os valores a serem ordenados sequencialmente na memória de dados, como em um vetor. Para que as "posições" fossem acessadas, destinou-se dois registradores para dois iteradores i e j, que percorrem todo o vetor, no caso realizando operações de *load* e *store* a partir de seus valores para acessar os valores contidos em cada posição da memória de dados. O resultado foi comprovado na placa FPGA e todos os números da sequência 1, 12, 15, 6, 10, 37 foram mostrados nos *displays* de forma crescente.

#### Algoritmo 5.2 – *Insertion Sort*

```

1 #include <stdio.h>
2
3 int main ()
4 {
5
6     int vet[6], n, i, j, aux;
7
8     scanf ("%d", &n);
9     for (i=0; i<n; i++)
10        scanf ("%d", &vet[i]);
11
12    for (i=1; i<n; i++)
13    {
14        j = i-1;
15        aux = vet[i];
16        while (j>=0 && (aux < vet[j]))
17        {
18            vet[j+1] = vet[j];
19            j--;

```

```

20         }
21         vet[j+1] = aux;
22     }
23
24     printf ("\n");
25     for (i=0; i<n; i++)
26         printf ("%d", vet[i]);
27     printf ("\n");
28
29     return 0;
30 }
```

O último algoritmo foi criado para que todas as instruções pudessem ser testas. Assim, ele não apresenta nenhuma funcionalidade lógica, como a sequência de Fibonacci, apenas apresenta todas as instruções testas em sequência, assim não possui um equivalente na Linguagem C. Seus resultados podem ser comprovados na *waveforms* da [Figura 5](#), [Figura 6](#), [Figura 7](#) e [Figura 8](#). A implementação deste pode ser vista no [Algoritmo 4.5](#), sendo este o último bloco de instruções.

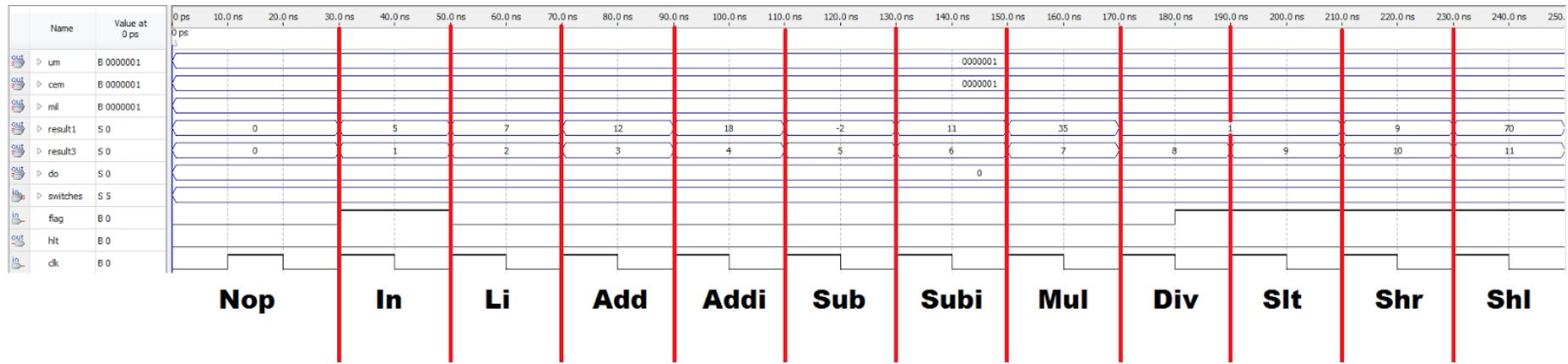
A sequência de instruções executada neste algoritmo segue o seguinte padrão:

- Instrução *Nop*;
- *Input* de dados. No caso do teste, entrou-se com o valor 5 no registrador 1;
- Carrega imediato 7 no registrador 2;
- Soma registrador 2 com 1 e guarda o valor no 3 ( $7 + 5 = 12$ );
- Soma o valor do registrador 1 com o imediato 13 e guarda o resultado em 4 ( $5 + 13 = 18$ );
- Subtrai o valor do registrado 2 do 1 e guarda o resultado no 5 ( $5 - 7 = -2$ );
- Subtrai o imediato 7 do registrador 4 e salva o resultado no 6 ( $18 - 7 = 11$ );
- Multiplica o valor do registrador 1 pelo 2 e salva o resultado no 8 ( $7 * 5 = 35$ );
- Divide o valor do registrador 8 pelo 4 e salva o resultado inteiro no 9 ( $35 / 18 = 1$ );
- Verifica se o valor do registrador 1 é menor que o registrador 2 e salva o resultado no 7 ( $\text{reg}[1] < \text{reg}[2] ? 1 : 0$ );
- Desloca em 1 *bit* para a direita o binário contido no registrador 4 e salva o resultado no registrador 11 ( $\text{reg}[4] \gg 1$ );
- Desloca em um *bit* para a esquerda o binário contido no registrador 8 e salva o resultado do registrador 12 ( $\text{reg}[8] \ll 1$ );

- Salva o valor do registrador 8 na posição de memória 1; (valor 35)
- Carrega o valor contido na posição de memória 1 para o registrador 11;
- *Output* do valor contido na posição 1 do módulo de saída; (valor 35)
- Salva o valor do registrador 3 na posição de memória 2; (valor 12);
- *Output* do valor contido na posição 2 do módulo de saída; (valor 12)
- *Jump* para o endereço 30;
- *Xor bit a bit* do valor do registrador 1 com o 2 e salva o resultado no 13;
- *And bit a bit* do valor do registrador 1 com o 2 e salva o resultado no 14;
- *Or bit a bit* do valor do registrador 1 com o 3 e salva o resultado no 15;
- Nega o valor binário do registrador 1 e salva no 16;
- *Xor bit a bit* do valor do registrador 1 com o valor imediato 13 e salva o resultado no registrador 17;
- *And bit a bit* do valor do registrador 1 com o valor imediato 13 e salva o resultado no registrador 18;
- *Or bit a bit* do valor do registrador 1 com o valor imediato 13 e salva o resultado no registrador 19;
- *Input* do valor 5 no registrador 20;
- *Nop*;
- *Branch not equal to zero* do registrador 1. Caso verdade, pula para o endereço 49;
- *Branch less than* do registrador 1 com o 2 ( $\text{reg}[1] < \text{reg}[2]$ ). Caso verdade, pula para o endereço 47;
- *Branch greater than zero* do registrador 1. Caso verdade pula para o endereço 51;
- *Branch greater than* do registrador 2 com o 1 ( $\text{reg}[2] > \text{reg}[1]$ ). Caso verdade pula para o endereço 44;
- *Branch less than zero* do registrador 5. Caso verdade pula para o endereço 55;
- *Output* do valor do registrador 1;

Todos os resultados, como citados anteriormente, podem ser vistos na *waveforms* da Figura 5, Figura 6, Figura 7 e Figura 8. O valor representado por *result 3* nos *waveforms* mostrou o endereço de saída do *Program Counter*; O valor representado por *result 1* mostrou o resultado advindo da *ALU* e o valor advindo do *Data Memory*; O valor representado por *do* mostrou o valor a ser mostrado nos *displays* caso uma instrução de *Output* fosse executada. Percebe-se que o valor de *do* se mantém ativo até a próxima instrução de saída, característica desejada na hora da implementação e comprovada com os testes.

Figura 5 – Waveform 1



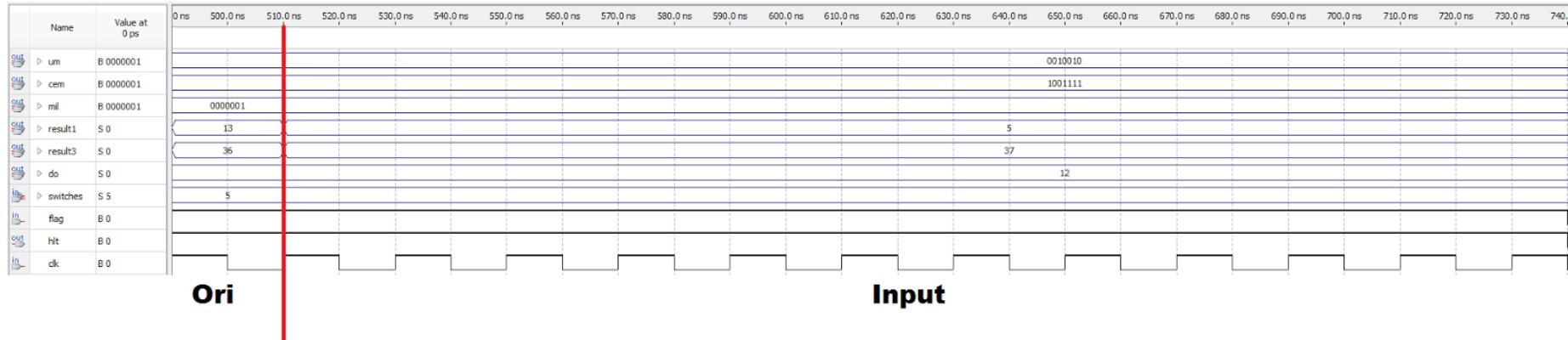
Fonte: Autoria Própria

Figura 6 – Waveform 2



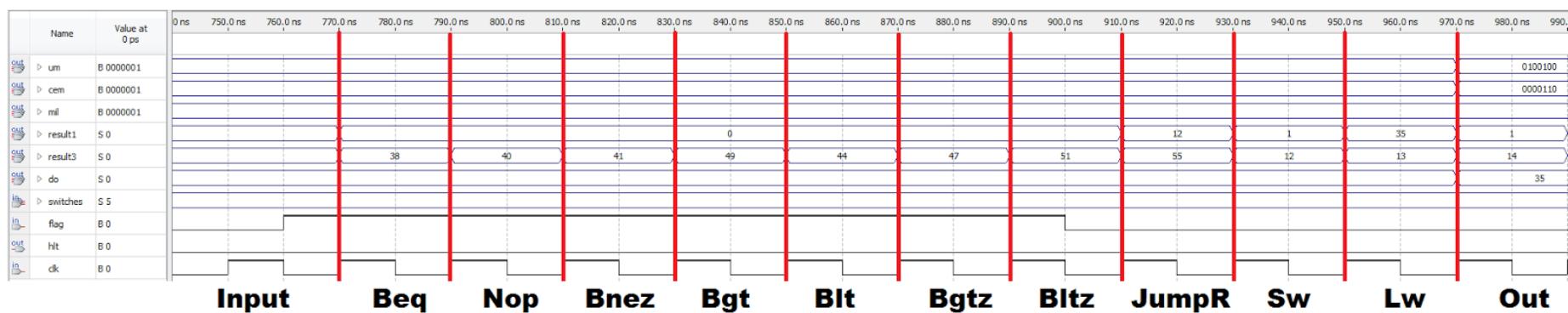
Fonte: Autoria Própria

Figura 7 – Waveform 3



Fonte: Autoria Própria

Figura 8 – Waveform 4



Fonte: Autoria Própria



## 6 Considerações Finais

Este projeto mostrou como a montagem e funcionamento de um processador, mesmo que simples, pode ser difícil e desafiadora, mas ao mesmo tempo, mostrou como a parte teórica de Arquitetura e Organização de Computadores interliga-se com a parte prática, aprimorando os conhecimentos sobre sistema digitais e processadores, além do aprofundamento nos conhecimentos sobre *MIPS* e seu funcionamento.

Nesta parte do projeto, implementou-se a Unidade de Controle do processador, podendo assim fazer a união entre Unidade de Processamento e Unidade de Controle, terminando assim a série de projetos deste laboratório. Para o futuro, busca-se aprimorar o processador com mais características que ajudem o usuário e até mudar a arquitetura de mono para multiciclo.

As maiores dificuldades deste projeto foram com relação à interligação dos componentes como um todo e a interpretação das instruções para que os resultados fossem obtidos de forma coerente. Com relação à montagem dos componentes individualmente, não houveram grandes dificuldades. Outro problema foi no uso da placa, que apresentou alguns problemas no uso do *clock* automático, mas logo foram resolvidos.



# Referências

- 1 STALLINGS, W. *Computer Organization and Architecture*. 9th edition. ed. Upper Saddle River/New Jersey, EUA: Pearson, 2013. Citado 4 vezes nas páginas 13, 14, 15 e 17.
- 2 PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado 5 vezes nas páginas 14, 15, 16, 17 e 83.
- 3 TECHNOLOGIES, I. *MIPS Instruction Set Quick Reference*. 2008. <<https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00565-2B-MIPS32-QRC-01.01.pdf>>. [Online, acessado 4/05/2017]. Citado na página 15.
- 4 TECHNOLOGIES, I. *Introduction to the MIPS32 Architecture*. 2014. <<https://imagination-technologies-cloudfront-assets.s3.amazonaws.com/documentation/MD00082-2B-MIPS32INT-AFP-06.01.pdf>>. [Online, acessado 4/05/2017]. Citado na página 15.
- 5 ENGLANDER, I. The architecture of computer hardware. *System Software, and Networking: An Information Technology Approach 4ED (P)*, 2013. Citado na página 17.
- 6 PALNIKTAR, S. *Verilog HDL, A guide to Digital Design and Synthesis*. 1th edition. ed. [S.l.]: SunSoft Press, 1996. Citado 2 vezes nas páginas 17 e 18.
- 7 MCDONAL, N. *BCD Converter Verilog*. <<http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html>>. [Online, acessado 2/05/2017]. Citado na página 35.
- 8 STOREY, T. *Debounce Logic Circuit*. 2013. <<https://eewiki.net/pages/viewpage.action?pageId=13599139>>. [Online, acessado 29/06/2017]. Citado na página 40.

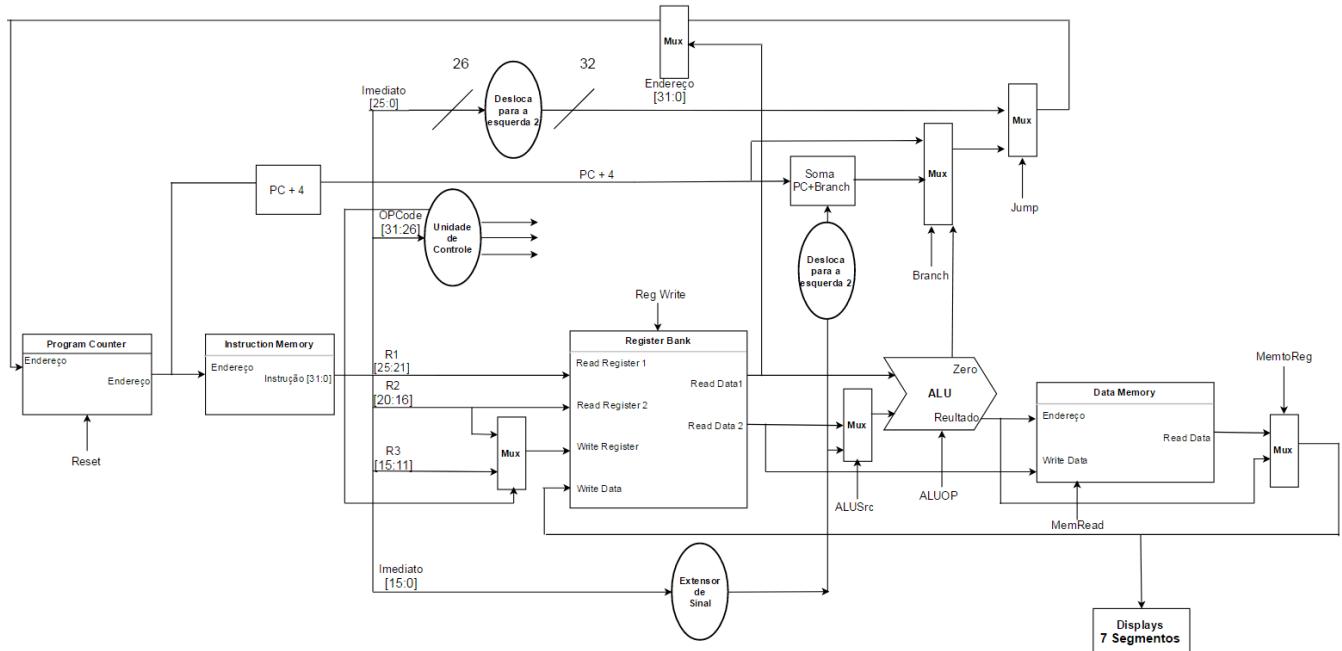


# Apêndices



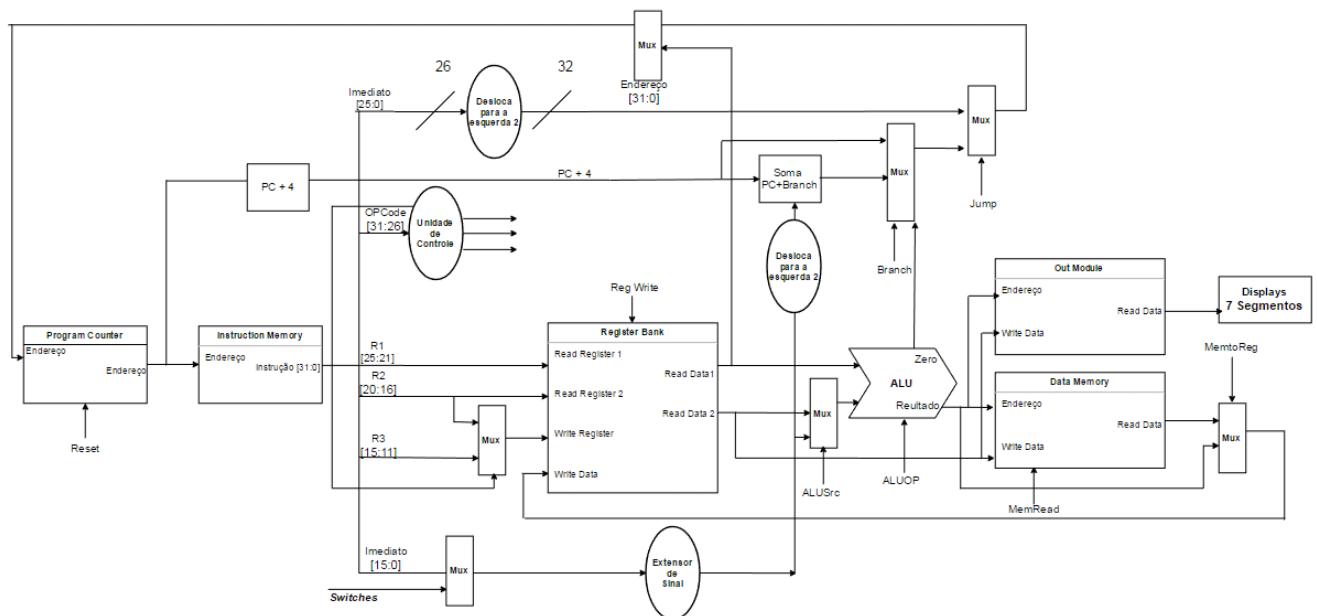
# APÊNDICE A – Apêndice 1

Figura 9 – Caminho de dados



Fonte: Autoria Própria

Figura 10 – Caminho de dados modificado



Fonte: Autoria Própria

Figura 11 – Mux 1

```
module MuxInstruction (Reg1, Reg2, RegOut, RegDst);
    input [31:0] Reg1, Reg2;
    input RegDst;
    output reg [31:0] RegOut;

    always @ (*)
        begin
            if (RegDst)
                RegOut = Reg2;
            else
                RegOut = Reg1;
        end

    endmodule
```

Fonte: Autoria Própria

Figura 12 – Mux 2

```
module MuxAlu (Imediate, Reg, MuxOut, ALUSrc);
    input [31:0] Imediate, Reg;
    output reg [31:0] MuxOut;
    input ALUSrc;

    always @ (*)
        begin
            if (ALUSrc)
                MuxOut = Imediate;
            else
                MuxOut = Reg;
        end

    endmodule
```

Fonte: Autoria Própria

Figura 13 – Mux 3

```
module MuxMem (ReadData, ALUresult, MemToReg, MuxOut);
    input [31:0] ReadData, ALUresult;
    input MemToReg;
    output reg [31:0] MuxOut;

    always @ (*)
        begin
            if (MemToReg)
                MuxOut = ReadData;
            else
                MuxOut = ALUresult;
        end

    endmodule
```

Fonte: Autoria Própria

Figura 14 – *Mux 4*

```

module MuxPC (PCInc, JumpBranch, PCSrc, Zero, MuxOut);

    input [31:0] PCInc, JumpBranch;
    input PCSrc, Zero;
    output reg [31:0] MuxOut;

    always @ (*)
        begin
            if (PCSrc & Zero)
                MuxOut = JumpBranch;
            else
                MuxOut = PCInc;
        end

endmodule

```

Fonte: Autoria Própria

Figura 15 – *Mux 5*

```

module MuxJump (MPC, JumpAdress, Jmp, MuxOut);

    input [31:0] MPC, JumpAdress;
    input Jmp;
    output reg [31:0] MuxOut;

    always@(*)
        begin
            if (Jmp)
                MuxOut = JumpAdress;
            else
                MuxOut = MPC;
        end

endmodule

```

Fonte: Autoria Própria

Figura 16 – *Mux 6*

```

module MuxJr (MJ, Reg, JumpReg, MuxOut);

    input [31:0] MJ, JumpReg;
    input Reg;
    output reg [31:0] MuxOut;

    always@(*)
        begin
            if (JumpReg)
                MuxOut = Reg;
            else
                MuxOut = MJ;
        end

endmodule

```

Fonte: Autoria Própria

Figura 17 – Conversor para *Display* de 7 segmentos

```
module Display (decimal_input, display_output);

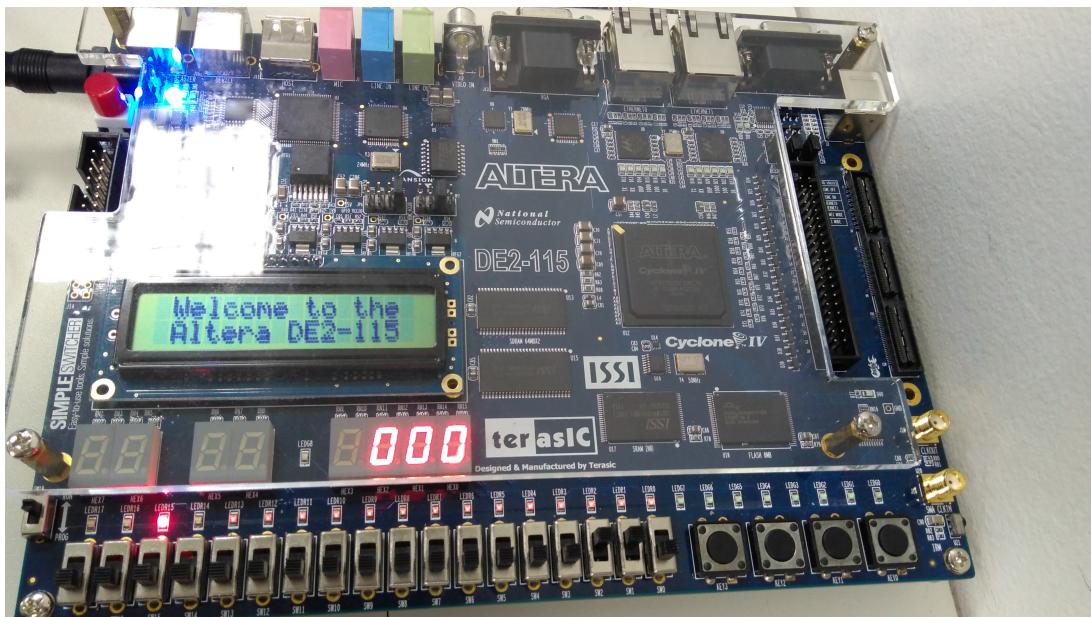
    input [3:0] decimal_input;
    output reg [6:0] display_output;

    always @ (decimal_input)
        begin
            case (decimal_input[3:0])
                'd0: display_output = 7'b0000001;
                'd1: display_output = 7'b1001111;
                'd2: display_output = 7'b0010010;
                'd3: display_output = 7'b0000110;
                'd4: display_output = 7'b0101100;
                'd5: display_output = 7'b0100100;
                'd6: display_output = 7'b0100000;
                'd7: display_output = 7'b0001111;
                'd8: display_output = 7'b0000000;
                'd9: display_output = 7'b0000100;
            endcase
        end
endmodule
```

Fonte: Autoria Própria

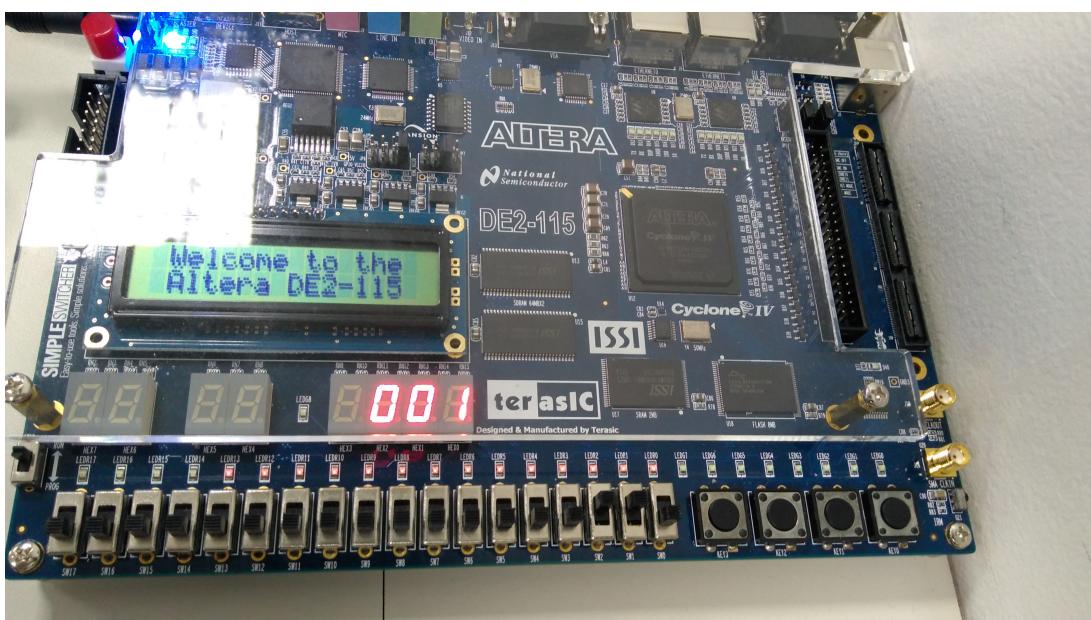
## APÊNDICE B – Apêndice 2

Figura 18 – Primeiro termo de Fibonacci



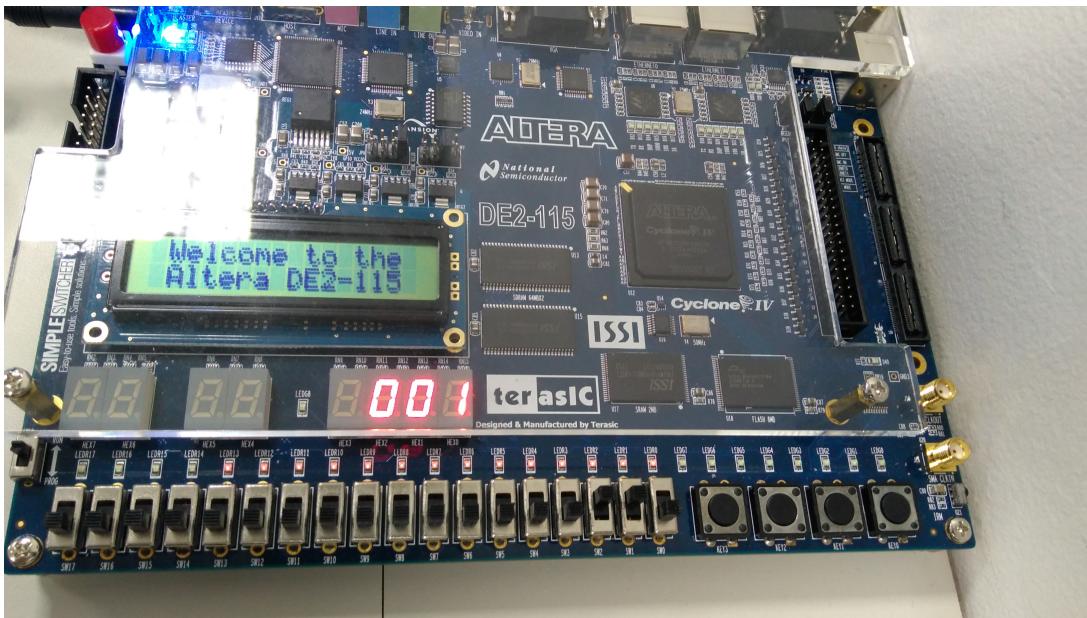
Fonte: Autoria Própria

Figura 19 – Segundo termo de Fibonacci



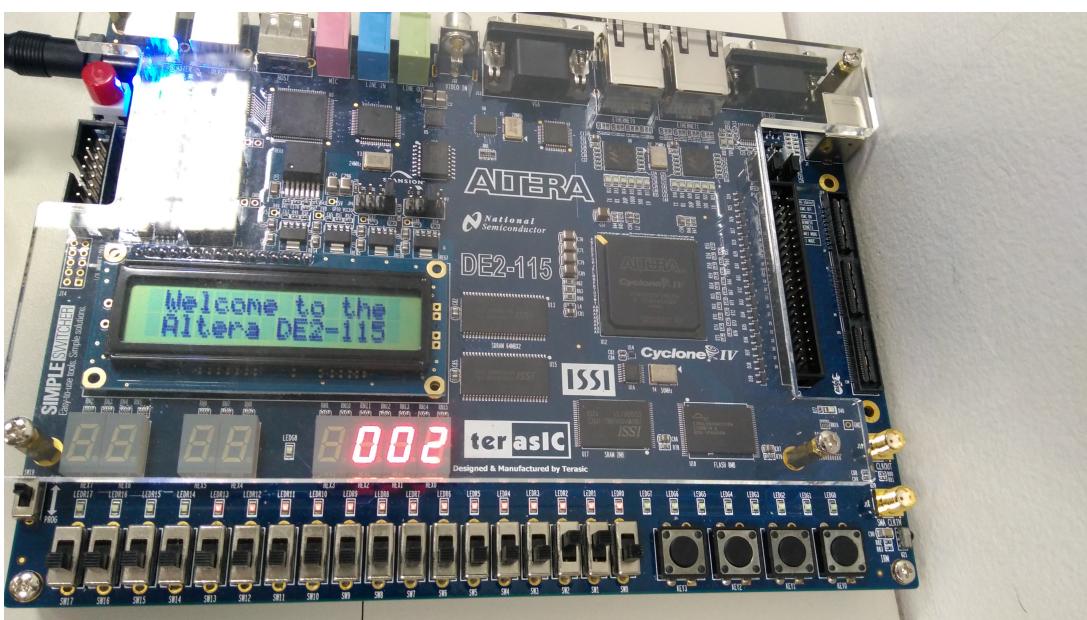
Fonte: Autoria Própria

Figura 20 – Terceiro termo de Fibonacci



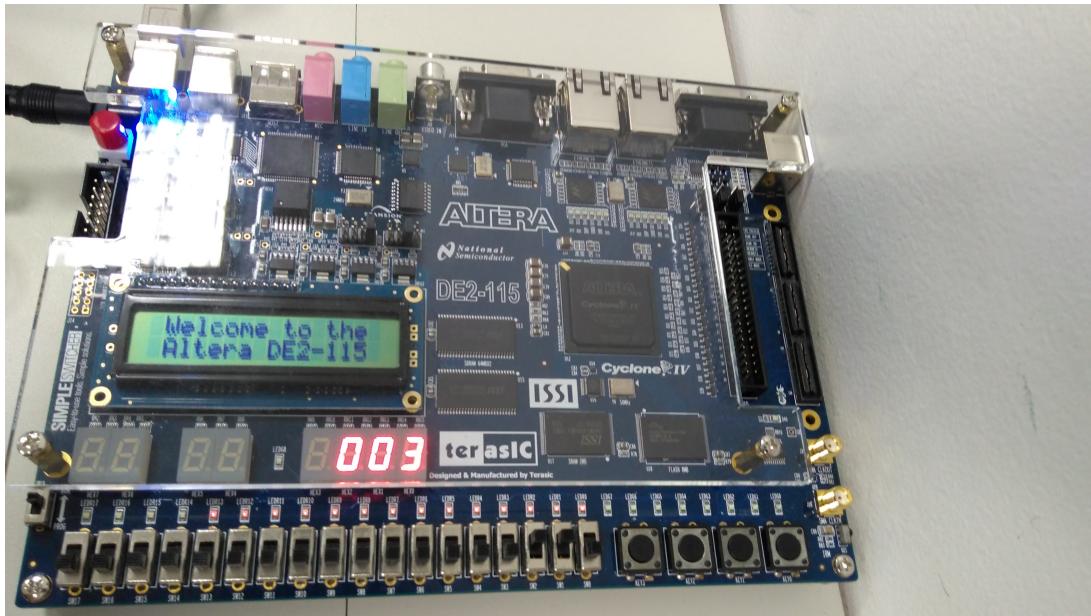
Fonte: Autoria Própria

Figura 21 – Quarto termo de Fibonacci



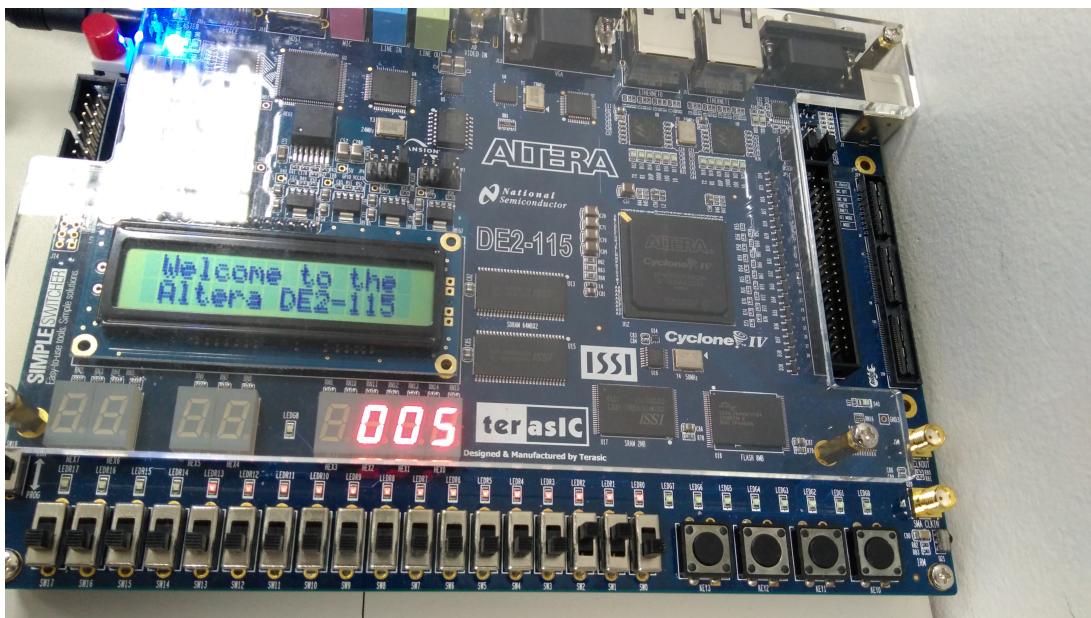
Fonte: Autoria Própria

Figura 22 – Quinto termo de Fibonacci



Fonte: Autoria Própria

Figura 23 – Sexto termo de Fibonacci



Fonte: Autoria Própria

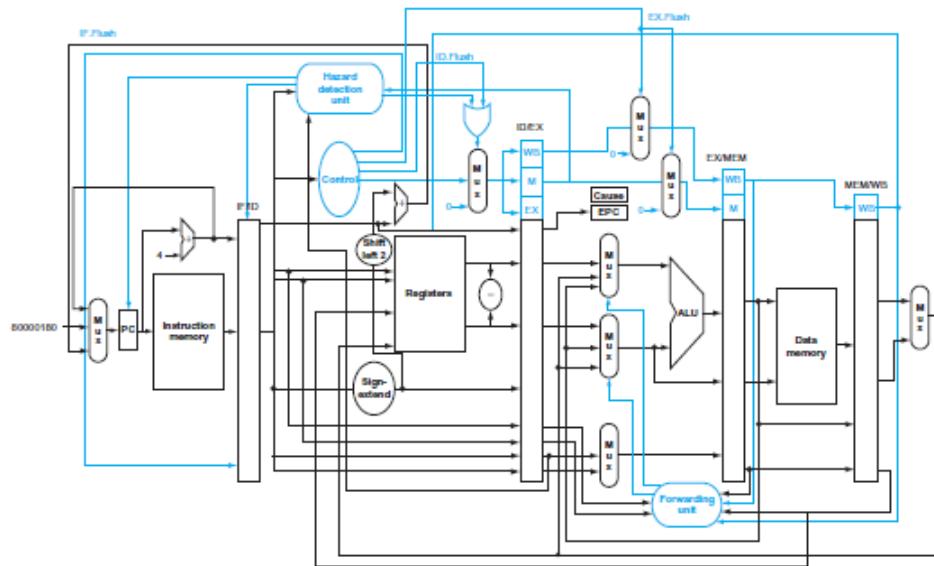


## Anexos



# ANEXO A – Anexo 1

Figura 24 – Caminho de dados MIPS com *pipelining*



Fonte: Computer Organization and Architecture [2]