Allan Carlos Salvador Portes

Desenvolvimento de um Sistema Computacional Baseado na Arquitetura MIPS

Laboratório de sistemas computacionais: Arquitetura e organização de computadores

São José dos Campos - Brasil

Julho de 2018

Allan Carlos Salvador Portes

Desenvolvimento de um Sistema Computacional Baseado na Arquitetura MIPS

Laboratório de sistemas computacionais: Arquitetura e organização de computadores

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

Julho de 2018

Resumo

O relatório em questão apresenta detalhes a respeito da implementação e desenvolvimento de um sistema computacional com arquitetura baseada na MIPS monociclo desenvolvido em liguagem de *hardware* Verilog. Durante este relatório será apresentado detalhadamente informações à respeito do conjunto e formato de instruções, modos de endereçamento e arquitetura-base e todos os aspectos fundamentais para o desenvolvimento da arquitetura.

Palavras-chaves: MIPS, arquitetura e organização de computadores, Verilog, sistema computacional.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

Com o avanço tecnológico, a utilização de sistemas computacionais em nosso dia-a-dia se tornou cada vez mais presente e relevante. E entender o funcionamento desta tecnologia é fundamental. Um computador é uma máquina capaz de realizar variados tipos de tratamentos de informações ou processamento de dados, possuindo inúmeros atributos como armazenamento e processamento de dados, operações lógicas e aritméticas e tratamento de imagens gráficas.

Além disso, é composto por diversos componentes que possuem funções distintas e complexas, que quando são combinadas funcionam em perfeita harmonia. Um exemplo disso é a CPU (unidade central de processamento) que em termos didáticos poderia ser considerada o cérebro deste sistema.

Para garantir a realização correta de todas as suas funções, um computador necessita de um conjunto de instruções (código de máquina compreendido pela CPU, que atua como interface entre hardware e software), também conhecido como ISA (*Instruction Set Architecture*). Este conjunto é tratado pela Unidade de Processamento que garante todos os requisitos necessários para a execução de um programa, sejam elas operações lógicas e aritméticas, acesso à memória, entrada e saída de dados, entre outras. Outra unidade fundamental para o funcionamento do computador é a Unidade de Controle, que será mostrada mais adiante.

Este é apenas um resumo que mostra quão importante e complexo pode ser um computador, e nas próximas páginas iremos conhecer em detalhes o funcionamento e desenvolvimento de alguns destes componentes.

2 Objetivos

2.1 Geral

O objetivo geral deste projeto é o desenvolvimento de um sistema computacional que opere de forma similar a um processador, de forma que suas instruções sejam testadas e seu funcionamento seja comprovado através de simulação em elementos de *hardware*.

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 Sistema Computacional

Um sistema computacional consiste num conjunto de dispositivos eletrônicos (*hardware*) capazes de processar informações de acordo com um programa (*software*).

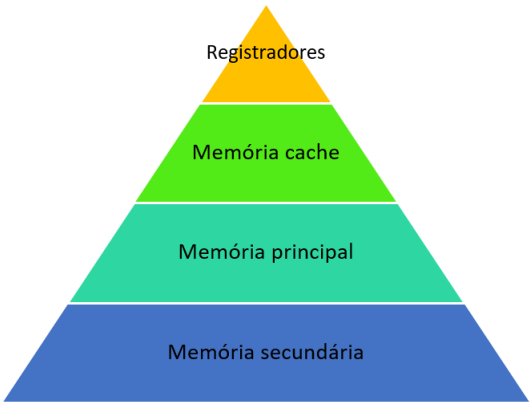
Um computador é composto de dispositivos de entrada e saída (como teclado, mouse e monitor), memória (que armazena todos os dados) e processador. Onde os dados que chegam através dos dispositivos de entrada são salvos na memória, para que após seu processamento sejam mostrados através do dispositivo de saída.

3.2 Hierarquia de memória

Dada a complexidade de um sistema computacional e suas operações realizadas é simples notar a necessidade da existência de diferentes níveis de armazenamento de dados, este conceito de criação é conhecido como hierarquia de memória.

Basicamente, a memória de um computador pode ser divida em quatro níveis principais, são eles: registradores, memória CACHE, memória principal e memória secundária. A Figura 1 representa a ordenação destes níveis hierárquicos, de forma que quanto mais no topo estiver, maior será o custo e velocidade de acesso e menor será sua capacidade de armazenamento .

Figura 1 – Hierarquia de Memória



Fonte: MaxiEduca (1)

De baixo para cima, a **memória secundária** é composta por dispositivos não voláteis como memórias externas de armazenamento em massa - por exemplo, disco rígido,

DVD, disquete, fita magnética, pen drive, entre outros. Sua principal característica é o armazenamento de dados que precisam se buscados antes de acessados, pois possuem baixa velocidade de acesso.

Subindo um nível, a **memória principal** possui uma alta velocidade de acesso, através da utilização da memória RAM (*Random Access Memory*). Que além de permitir leitura e escrita, mantém armazenado os programas operacionais básicos. A **memória CACHE** serve como ponte entre a troca de dados dos registradores e memória RAM. É uma memória de acesso rápido e otimiza os blocos de memória que são pertinentes ao processo em execução.

Finalizando, os **registradores** são os meios mais rápidos e computacionalmente caros de armazenamento de dados. Usualmente utilizados de forma temporária, registradores são as unidades de memória diretamente utilizadas pelas instruções atualmente sendo executadas pelo sistema.

3.3 Arquitetura Computacional

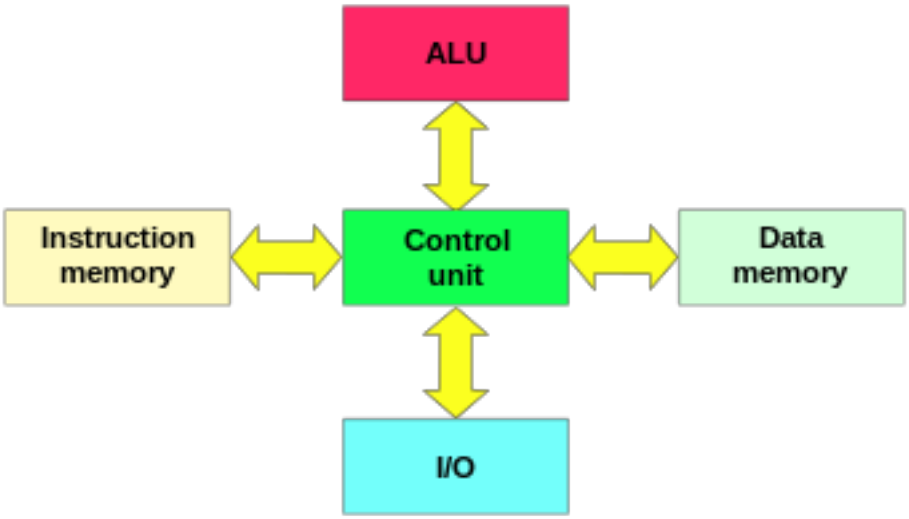
Para a construção de um processador é necessário definir quais instruções este deverá executar, definindo desta forma sua arquitetura. A arquitetura organiza a estrutura computacional de forma a otimizar seu funcionamento, a partir da combinação de diversas instruções e visando a execução de alguma função específica.

3.3.1 Arquitetura Havard

A Arquitetura de Harvard - representada na Figura 2 - é uma arquitetura de computador que se distingue das outras por possuir separadamente circuito para sinais e armazenamento para, dados e instruções, que são independentes em termos de barramento e ligação ao processador (2).

Esta arquitetura se baseia no conceito da Arquitetura de Von Neumann e é composta por unidade de controle, memória de instrução, memória de dados, unidade lógica e aritmética e módulos de entrada e saída. A diferença entre a arquitetura de Von Neumann e a Harvard é que a última separa o armazenamento e o comportamento das instruções do CPU e os dados, enquanto a anterior utiliza o mesmo espaço de memória para ambos (2).

Figura 2 – Arquitetura Harvard



Fonte: Diego Macedo (2)

3.3.2 Arquitetura RISC

A arquitetura RISC (*Reduced Instruction Set Computers*) possui um número reduzido e simplificado de instruções, podendo operar a velocidades maiores de clock.

Desta forma, este tipo de arquitetura possui uma ênfase maior em *software* e requer um trabalho maior do programador, uma vez que exige mais linhas de código. Suas principais características são:

1.Número reduzido de ciclos de clock por instrução;

2.Utiliza um formato fixo de instrução;

3.Conjunto reduzido de instruções.

Processadores baseados nesta arquitetura são mais simples e muito mais baratos, possuindo um menor número de circuitos internos, como por exemplo os processadores Alpha.

Apesar das diversas peculiaridades e diferenças destas arquiteturas, atualmente muitos modelos de processadores abrigam características de ambas. E as grandes fabricantes utilizam desta combinação visando melhorar o desempenho durante a execução das instruções.

3.4 MIPS

Vimos que a arquitetura de um processador é a forma como ele se comporta funcionalmente. Com o passar dos anos e do desenvolvimento de novas tecnologias, diversas arquiteturas foram criadas, dentre elas a arquitetura MIPS.

Desenvolvida nos anos 80 por pesquisadores da Universidade de Stanford, esta arquitetura segue o padrão de arquiteturas RISC. Utilizando de um número reduzido de registradores e um conjunto de instruções menor.

Diversos conjuntos de MIPS foram implementados utilizando diferentes números de registradores, contudo, os números principais são os de 32 bits (número que será utilizado neste projeto) e o de 64 bits.

Outra característica interessante é que a execução desta arquitetura é feita em cinco estágios, são estes: busca, decodificação, execução, acesso à memória e escrita de dados.

3.4.1 Tipos de Instruções

A arquitetura MIPS realiza diversos tipos de operações, de forma que seus tipos de instruções são divididos em três partes: R, I e J.

É no **tipo R** que todas as instruções de operação lógica e aritmética com dados dos registradores se encontram, como por exemplo soma e subtração. Ainda há subdivisões neste tipo de instrução chamados: opcode, RS, RT, RD, *shamt* e *funct*.

Tabela 1 – Instruções de formato R

Campo	*Opcode*	RS	RT	RD	*shamt*	*funct*
Tamanho (bits)	6	5	5	5	5	6
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5-0
Função	Funciona como um identificador de instruções	Representam os endereços dos bancos de registradores que os dados serão retirados	Representam os endereços dos bancos de registradores que os dados serão retirados	Endereço do registrador que receberá o resultado da operação	Quantidade de bits que serão deslocados (se for necessário)	Diferencia instruções na ULA, funciona como uma extensão do opcode

Fonte: Autor

O **tipo I**, possui instruções que realizam operações lógicas e aritméticas com dados de *offset* a partir do campo imediato, ou seja, executam acesso à memória e executam saltos condicionais. A tabela abaixo ilustra quais são os campos deste tipo de instrução.

Tabela 2 – Instruções de formato I

Campo	*Opcode*	RS	RT	*offset*
Tamanho (bits)	6	5	5	11
Bits	31 - 26	25 - 21	20 - 16	15 - 0
Função	Funciona como um identificador de instruções	Representam os endereços dos bancos de registradores que os dados serão retirados	Endereço do registrador que receberá o resultado da operação	O campo offset ou imediato, possui um valor codificado que será usado para possíveis saltos condicionais ou operações aritméticas

Fonte: Autor

Finalizando, o terceiro tipo de instrução é o **tipo J**, responsável pelas instruções de saltos. Por esta característica, possui uma divisão mais simples.

Tabela 3 – Instruções de formato J

Campo	*Opcode*	*adress*
Tamanho (bits)	6	26
Bits	31 - 26	25 - 0
Função	Funciona como um identificador de instruções	Endereço no qual o salto será destinado

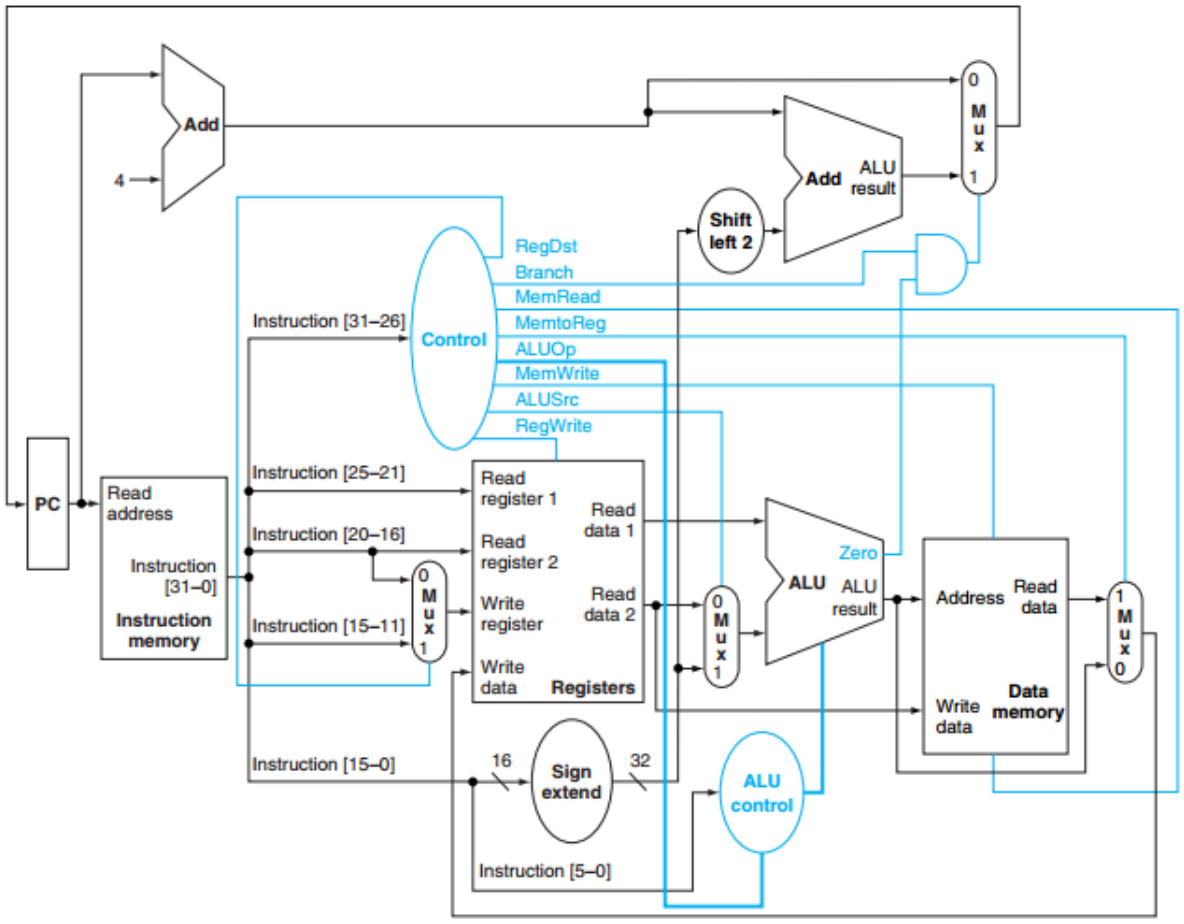
Fonte: Autor

3.4.2 Datapath

Como o nome sugere, o *Datapath* ou caminho de dados, é o caminho tomado pelos bits processados em uma instrução, representando o total de componentes presentes na arquitetura.

Quando estes componentes interligados recebem a ação de uma sinal de controle (*flag*), direcionam a operação através das diferentes partes do circuito com o intuito de obter um resultado específico. A Figura 3, mostra o caminho de dados da arquitetura MIPS.

Figura 3 – MIPS Monociclo



Fonte: Acervo Livro (3)

3.5 FPGA

O FPGA (*Field Programmable Gate Array*) é um circuito integrado projetado para ser programável. Este chip consiste em um arranjo de blocos lógicos configuráveis contidos em um único circuito. Cada célula contém capacidade computacional para implementar funções lógicas e realizar roteamento para comunicação entre elas. O primeiro FPGA disponível comercialmente foi desenvolvido pela empresa Xilinx Inc, em 1983 (4).

4 Desenvolvimento

Para este projeto, o modelo MIPS foi escolhido pois é baseado na arquitetura RISC, portanto é mais simplificado e fácil de implementar. Além disso, é um modelo didaticamente viável e possui uma vasta literatura para fins de estudo.

4.1 Conjunto de instruções

A arquitetura MIPS diversos tipos de instruções com características que variam de acordo com o tipo de implementação. Em um MIPS de 32 bits, pode-se trabalhar com 32 registradores, acessados por um intervalo de 5 bits.

Para este projeto, utilizou-se um conjunto semelhante e reduzido ao MIPS. Consistindo por instruções (Tabela 4) que visam abranger todas as operações básicas do processador, como operações lógicas e aritméticas, saltos e acesso à memória.

4.1.1 Modos de Endereçamento

Para este projeto, será utilizado quatro modos de endereçamentos utilizados na arquitetura MIPS.

Endereçamento imediato: considerado o modo mais simples de endereçamento, o endereçamento imediato possui o operando contido no campo de endereço, portanto, não necessita acesso à memória para buscar o operando.

Endereçamento por registrador: este modo referencia o respectivo registrador no campo de endereço e nele já contém o operando, sem fazer acesso à memória. É utilizado por instruções como sub, add, or, etc.

Endereçamento por deslocamento: no endereçamento por deslocamento ou de base, o registrador contém um endereço base e um campo de endereço é um deslocamento.

Endereçamento relativo ao PC: neste modo, a próxima instrução a ser executada é relativa ao endereço da instrução atual. Desta forma, seu endereço é calculado através da soma entre uma constante da instrução e o PC. Geralmente é utilizada por instruções de desvio condicional, como a beq.

Endereçamento pseudodireto: utilizado por instruções de salto incondicional, no endereçamento pseudodireto, o endereço é formado pela concatenação dos 26 bits da instrução com os bits mais altos do PC.

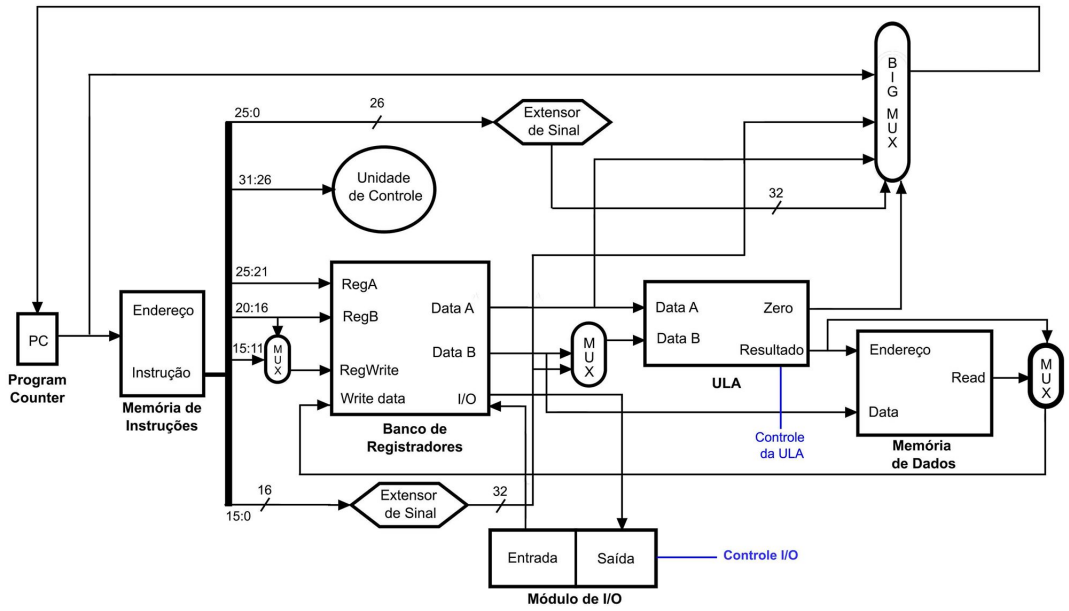
Tabela 4 – Conjunto de Instruções

Categoria	**Instrução**	**Tipo**	**tipo de instrução**
Aritmética	**adição**	**add**	**R**
Aritmética	**subtração**	**sub**	**R**
Aritmética	**adição com imediato**	**addi**	**R**
Aritmética	**subtração com imediato**	**subi**	**R**
Aritmética	**multiplicação**	**mult**	**R**
Lógicas	**NOT**	**not**	**R**
Lógicas	**AND**	**and**	**R**
Lógicas	**OR**	**or**	**R**
Lógicas	**XOR**	**xor**	**R**
Lógicas	**set less than**	**stl**	**R**
Lógicas	**deslocamento à esquerda**	**shfl**	**R**
Lógicas	**deslocamento à direita**	**shfr**	**R**
Memória	**load word**	**lw**	**I**
Memória	**load imediato**	**li**	**I**
Memória	**store worf**	**sw**	**I**
Saltos	**branch and equal**	**beq**	**I**
Saltos	**branch and not equal**	**bne**	**I**
Saltos	**branch and equal zero**	**beqz**	**I**
Saltos	**jump**	**jump**	**J**
Saltos	**jump to register**	**jumpr**	**J**
Outros	**NOP**	**nop**	**R**
Outros	**Input**	**in**	**Outros**
Outros	**Output**	**out**	**Outros**
Outros	**HLT**	**hlt**	**R**

Fonte: Autor

4.2 Arquitetura-base do Processador

Figura 4 – Arquitetura Base



Fonte: Autor

O caminho de dados deste projeto é semelhante ao da arquitetura MIPS monociclo, porém com um grau de complexidade menor. Nele, estão presentes alguns componentes importantes para o funcionamento do processador, que serão explicados detalhadamente.

4.2.1 Contador de Programa

O **Contador de Programa**, ou *Program Counter* (PC), é o primeiro elemento do caminho de dados. Nele é armazenado o endereço atual da instrução em execução através de um registrador, que sempre é atualizado a cada subida do *clock*.

Essa atualização de valor pode ser efetuada de três formas principais. Na primeira, o valor do registrador simplesmente é acrescido em uma unidade, indicando qual será a próxima instrução a ser executada.

A segunda maneira é voltada para quando ocorre algum tipo de desvio, como nas instruções do tipo *jump*. Neste caso, o valor atual do PC não é acrescido de uma unidade, mantendo-se o mesmo. No terceiro e último caso, quando o sinal de *reset* é igual a 1, o valor do PC é zerado. Outro aspecto, é que existe um sinal chamado "insign"como entrada no módulo, que é utilizado em algumas instruções com o objetivo de impedir a mudança no PC até que o usuário indique.

```
1 module Program_Counter (clock, adress, reset, hlt, pgcount, jump, jr, insign);
2
```

```
3         input clock, reset, hlt, jump, jr, insign;
4         input[9:0]adress;
5         output reg [9:0]pgcount;
6
7         wire [9:0]  newValue;
8         assign newValue = adress;
9
10
11        always @(posedge clock) begin
12
13                        if(reset) begin
14                                pgcount = 0;
15                        end
16
17                        else if(insign ==1) begin
18
19                        end
20
21                        else if((jump==1) || (jr==1))
22                                pgcount <= newValue;
23
24
25                        else begin
26                                pgcount <= adress + 1;
27                        end
28
29                end
30
31
32  endmodule
```

Listing 4.1 – Algoritmo 1 - Program Counter

4.2.2 Memória de Instruções

A **Memória de Instruções** (*Instruction Memory*) é responsável pelo armazenamento de todas as instruções realizadas pelo processador, em posições contíguas de memória.

Estas instruções são enviadas para outros componentes referenciados pelo endereço contido no PC, que após isso é atualizado na próxima subida de *clock* e busca a instrução armazenada no endereço seguinte.

A implementação deste módulo, consiste basicamente em registradores que guardam cada instrução do processador ou possivelmente algum programa específico, como o de Fibonacci que será utilizado futuramente.

Dado um determinado endereço, a saída do programa é a instrução equivalente a posição de memória deste vetor de registradores. E a execução deste módulo ocorre na descida do *clock*.

O Algoritmo 4.2, contém apenas 3 instruções no endereço da memória, para melhor visualização de sua implementação. Porém, na parte final deste relatório, será possível

visualizar a implementação completa.

```
1  module MemoryInstruction (address, InstructionOut, clock);
2
3          input [9:0] address;
4          input clock;
5          output [31:0] InstructionOut;
6          reg [31:0] mem [31:0];
7          integer flag = 1;
8
9    always @ (posedge clock) begin
10
11         if(flag == 1) begin
12
13                 mem[0] = {32'd0}; //nop
14                 mem[1] = {32'd20}; //add
15                 mem[2] = {32'd50}; //sub
16                 flag <= 1;
17         end
18
19   end
20
21   assign InstructionOut = mem[address];
22
23  endmodule
```

Listing 4.2 – Algoritmo 2 - Memória de Instruções

4.2.3 Banco de Registradores

O Banco de Registradores é o componente em que são armazenados dados temporários pertinentes à execução das instruções, sendo o tipo de memória situada no topo da hierarquia (5).

Neste componente, existem 32 registradores de propósito geral que são constantemente alterados com base nos endereços de entrada das instruções. E seu armazenamento é realizado com base em um sinal, que indica quando os dados provenientes do final do circuito devem ou não ser escritos nestes endereços. Neste módulo, as variáveis **Reg1** e **Reg2** representam os dois endereços dos registradores de entrada, já o ***WriteRegister*** é o endereço do registrador de escrita, que recebe a saída do multiplexador anterior ao Banco de Registradores, visto na Arquitetura-Base da Figura 4.

O ***WriteData*** designou-se para receber os dados escritos no registrador de escrita, provenientes do multiplexador que seleciona a saída da ULA ou da Memória de Instruções. As duas saídas **Data1** e **Data2**, representam os dados de leitura de registradores de entrada.

O Banco de Registradores foi implementado de forma síncrona ao *clock* em borda de subida a cada ciclo, para que o mesmo funcionel de acordo com um sinal de seleção determinado ***RegWrite*** no algoritmo. Quando este sinal é positivo, significa que o dado será escrito no registrador, caso contrário, nada seria escrito nos registradores.

Outro aspecto importante é que a entrada e saída do Módulo I/O ocorre através do Banco de Registradores, a partir de um sinal provindo da Unidade de Controle. Caso o sinal de escrita **inputControl** for ativado, o dado provindo das chaves do FPGA é salvo no registrador e no caso da saída, quando o sinal de *output* estiver ativo, o dado é salvo na variável **resultado**.

```
1  module RegBank (clock, Reg1, Reg2, RegWrite, WriteData, WriteRegister, Data1, Data2,
       Resultado, outputControl, RegOut1, RegOut2, inputControl, DataIn);
2
3          input [4:0] Reg1, Reg2, WriteRegister;
4          input [31:0] WriteData;
5          input RegWrite, clock, outputControl, inputControl;
6          output [31:0] Data1, Data2;
7          reg [31:0] Register [31:0];
8          output [4:0] RegOut1, RegOut2;
9          input [31:0] DataIn;
10
11     output reg [31:0] Resultado;
12
13         always @(posedge clock)
14                 begin
15                         Register[0] = 32'b0;
16
17                         if(RegWrite)
18                                 Register [WriteRegister] = WriteData;
19
20                         if(outputControl) begin
21                                 Resultado = Register[Reg1];
22
23                         end
24                         else begin
25                                 Resultado = 8'b0;
26                         end
27
28                         if(inputControl) begin
29                                 Register [WriteRegister] = DataIn;
30                         end
31
32
33
34
35                 end
36
37                 assign Data1 = Register[Reg1];
38                 assign Data2 = Register[Reg2];
39                 assign RegOut1 = Reg1;
40                 assign RegOut2 = Reg2;
41
42 endmodule
```

Listing 4.3 – Algoritmo 3 - Banco de Registradores

4.2.4 Unidade Lógica e Aritmética

O componente responsável por realizar cálculos e deslocamentos dos dados de entrada é a **Unidade Lógica e Aritmética** (ULA) ou ALU (*Arithmetic Logical Unity*).

Tabela 5 – - Operações realizadas pela ULA

Opcode	**Tipo**	**Operação Realizada**
00000	Soma	result = data1 + data2
00001	Subtração	result = data1 - data2
00010	Soma com imediato	result = data1 + 1
00011	Subtração com imediato	result = data1 -1
00100	*Set less than*	result = data1 <data2 ? 1: 0;
00101	Multiplicação	result = data1 * data2
00110	*Set Greater than*	restult = data1 >data2 ? 1: 0
00111	*Shift Left*	restult = data1 <<shamt
01000	*Shift Right*	result = data1 >>shamt
01001	*Not*	result = ∼data1
01010	*And*	result = data1 & data2
01011	*Or*	result = data1 | data2
01100	*Xor*	result = data1 ^data2
01101	*Branch on Equal*	result = data1 == data2 ? 0 : 1
01110	*Set Less or Equal Than*	result = data1 <= data2 ? 0 : 1
01111	*Set Greater or Equal Than*	result = data1 >= data2 ? 0: 1
1000	*Branch on Not Equal*	result = data1 != data2 ? 0:1
1001	*Branch on Equal Zero*	result = data1 == 0 ? 0 : 1

Tabela 6 – Fonte: Autor

Pode ser utilizada para diversos propósitos, como verificar igualdade entre dados para determinando saltos condicionais, calcular um endereço de escrita ou ainda realizar operações lógicas e aritméticas. Todas as operações realizadas pela Unidade Lógica e aritmética estão referenciadas na Tabela 5.

Seu funcionamento não depende do *clock*, como outros componentes, mas de variações de seus sinais de entrada. Devido ao fato de realizar diversas operações, a ULA conta com um único sinal de controle, gerado com base na leitura do opcode da instrução, diferenciando suas possíveis formas de execução.

Este componente conta com um *opcode* que indentifica qual é a instrução a ser executada. Além disso, como este código já é suficiente para todas as possíveis operações, não foi necessário utilizar o *funct* de determinados tipos de instruções. Utilizou-se também uma variavel para determinar se o resultado da operação foi zero, isto será utilizado por instruções de salto condicional, como *branch on equal*.

```
1  module ULA (op, data1, data2, result, signal_zero, signal_neg, shamt);
2
3
4          input [31:0]data1;
5          input [31:0]data2;
6          input [4:0]shamt;
7          input [4:0]op;
8
9          output signal_zero;
```

```
10          output signal_neg;
11          output reg [31:0]result;
12
13
14          always @(data1 or data2 or op or shamt) begin
15                  case(op[4:0])
16                          5'b00000: result = data1 + data2; //add
17                     5'b00001: result = data1 - data2; //sub
18                     5'b00010: result = data1 + 1;         //addi
19                     5'b00011: result = data1 - 1;  //subi
20                     5'b00100: result = data1 < data2 ? 1 : 0; //slt
21                     5'b00101: result = data1[15:0] * data2[15:0]; //mult
22                     5'b00110: result = data1 > data2 ? 1 : 0; // Set greater than (Branch)
23                     5'b00111: result = data1 << shamt;//shfl
24                     5'b01000: result = data1 >> shamt; //shfr
25                     5'b01001: result = ~data1; //not
26                     5'b01010: result = data1 & data2; //and
27                     5'b01011: result = data1 | data2; //or
28                     5'b01100: result = data1 ^ data2; //xor
29                          5'b01101 : result = data1 == data2 ? 1: 0; // Set equal than (
                                Branch)
30                          5'b01110 : result = data1 <= data2 ? 1 : 0; // Set less or equal
                                than (Branch)
31                          5'b01111 : result = data1 >= data2 ? 1 : 0; // Set greater or
                                equal than (Branch)
32                          5'b10000 : result = data1 != data2 ? 1 : 0; // Set different
33                          5'b10001 : result = data1 == 0 ? 1 : 0; //Branch and equal zero -
                                 beqz
34                          default : result = 0;
35                  endcase
36          end
37
38          assign signal_zero = (result == 0);
39          assign signal_neg = (($signed(result) < 0));
40
41
42  endmodule
```

Listing 4.4 – Algoritmo 4 - ULA

4.2.5 Memória de Dados

A **Memória de Dados** tem o objetivo de armazenar eventuais informações durante a execução de uma instrução.

Sua escrita ocorre a partir do direcionamento de uma *flag*, semelhante ao Banco de Registradores e suas únicas entradas são o endereço de acesso e o dado que será salvo neste respectivo endereço. Contudo, apenas as instruções load e store realizam acesso direto aos seus endereços e os dados contidos nele.

Portanto, sua implementação pode ser considerada mais simplificada em relação a outros módulos da arquitetura. Sua entrada principal consiste em um dado e seu respectivo endereço. Um sinal de controle determina se este dado será gravado na memória (no caso

da instrução *store word*) ou se irá retornar o valor referente ao endereço de entrada (caso a instrução seja *load word*).

Este módulo atua na borda de descida do *clock*, para evitar que a memória tente ler um dado de um endereço ainda não calculado do Banco de Registradores. Além disso, a Memória de Dados envia sinais na saída em todo ciclo de *clock*, porém estes valores sçao utilizados apenas quando selecionados pelo seu respectivo multiplexador.

```
1  module DataMemory(clock, address, flag, DataOut, data);
2
3  input [31:0] data;
4  input [31:0] address;
5  input clock , flag;
6  output [31:0] DataOut;
7  reg [31:0] Out[9:0];
8
9  always @ (negedge clock) begin
10
11 if (flag)
12         Out[address] = data;
13 end
14 assign DataOut = Out[address];
15
16 endmodule
```

Listing 4.5 – Algoritmo 5 - Memória de Dados

4.2.6 Unidade de Controle

Já a **Unidade de Controle** (UC) é o módulo responsável por realizar a troca de todas as *flags* com base na instrução em execução. Este módulo possui um papel de extrema importância no funcionamento do processador, uma vez que diferencia o resultado das instruções a partir dos seus sinais de controle.

Podemos comparar este componente ao cérebro do processador, orquestrando seu funcionamento e ativando cada *flag* de cada instrução. Ou seja, a UC que é um circuito combinacional - é dito combinacional pois a saída depende única e exclusivamente das combinações das variáveis de entrada recebidas em um dado momento (6) - recebe e interpreta o opcode de cada instrução e a partir desta interpretação define a disposição adequada dos passos de execução do caminho de dados.

Devido a extensão de linhas, a implementação completa desta unidade pode ser encontrada na sessão de apêndices.

4.2.7 MUX

Os **multiplexadores** (MUX), funcionam como um seletor do que qual informação enviará para a saída. Visto que, o mesmo pode possuir dois ou mais dados de entradas. Como a implementação é igual para todos os MUX's utilizados no processador (com

exceção do Big Mux), apenas um dos MUX utilizados estão presentes abaixo no Algoritmo 4.6

```
module MuxBank1(input1, input2, out, select); //Mux de entrada do Banco de Registradores

  input select;
  input [4:0] input1, input2;
  output reg [4:0] out;

  always @ (*) begin

        case(select)
                      1'b0 : out = input1;
                1'b1 : out = input2;
        endcase

  end

endmodule
```

Listing 4.6 – Algoritmo 6 - MUX

4.2.7.1 Big MUX

Um elemento importante no desenvolvimento da arquitetura é o chamado **Big MUX**, que possui o objetivo de determinar qual será o endereço correto do Contador de Programa ao término de cada instrução. Seu funcionamento é baseado em sinais seletores que selecionam qual entrada será a saída de retorno do PC.

Este componente foi pensado para contemplar corretamente o funcionamento das instruções *branch on equal* e *branch on not equal*. Que são executadas a partir de um sinal 0 ou 1.

Para tratar intruções de *branch on equal*, *branch on not equal* e *brach on equal zero*, além do sinal de controle (zero) utilizou-se *flags* para determinar qual das entradas fornecidas seriam usadas como saída de volta ao PC, isso porque existe uma diferenciação do valor deste sinal em sua comparação.

Neste componente, também efetuou-se um tratamento para as intruções de *jump* e *jump to register* e por padrão, o valor de saída era o endereço provindo do PC.

```
module BigMux(zero, beq, bneq, beqz, selectbm, outputbm, outputpc, sum, signal, regdata);

  input zero, beq, bneq, beqz;
  input [1:0]  selectbm;
  input [31:0]  outputpc, sum, signal, regdata;
  output reg [31:0] outputbm;

always @ (*) begin

        /*if(beq  == 1 && zero == 1)
      outputbm = sum;
        /*else if(jump)
                outputbm = signal;//jump
```

```
14          else if(jr)
15                  outputbm = regdata;
16          else
17           outputbm <= outputpc + 32'd1;
18           */
19
20                  case(selectbm[1:0])
21                          2'b01: //branch
22                                  begin
23                                          if(bneq == 1 && zero == 0)
24                                                  outputbm = outputpc;
25
26                                          else if(bneq == 1 && zero == 1)
27                                                  outputbm = sum-1;   //pula
28
29                                          else if(beq == 1 && zero == 1)
30                                                  outputbm = sum-1;
                                                              //pula
31
32                                          else if(beq == 1 && zero == 0)
33                                                  outputbm = outputpc;
34
35                                          else if(beqz == 1 && zero == 1)
36                                                  outputbm = sum-1;       //pula
37
38                                          else if(beqz == 1 && zero == 0)
39                                                  outputbm = outputpc;
40                                          else
41                                                  outputbm = outputpc;
42
43                                  end
44
45                          2'b10: outputbm = signal;//jump
46
47                          2'b11 : outputbm = regdata;//jump register
48
49                          default: outputbm = outputpc; //2'b00
50
51                  endcase
52  end
53
54  endmodule
```

Listing 4.7 – Algoritmo 7 - BIGMUX

4.2.8 Extensor

Outro componente auxiliar do processador, é o extensor de *bit* (*Sign Extend*). Que possui a função de receber uma entrada de um determinado tamanho em bits, concatená-la com 0's ou 1's até atingir um tamanho específico e mandar esta nova informação para outro componente. Por exemplo, um extensor pode transformar um código de 16 *bits* em um de 32 *bits*, sem alterar a informação original.

Para que esta operação fosse efetuada sem erros, levou-se em consideração a que os números binários são tratados em complemento de 2 na linguagem *Verilog*. Ou seja, caso

um número apresentasse 0 como o *bit* mais significativo, bastou-se adicionar a quantidade restante de *bits* para completar 32 com mais 0's. Já se o bit mais significativo fosse 1, o número binário apresentava sinal negativo, e desta forma, seus *bits* restantes eram completados com 1's.

```
1  module BitExtender (DataIn, DataOut); //Extensor de bits de 16 para 32
2
3          input [15:0] DataIn;
4          output reg [31:0] DataOut;
5
6
7          always @ (*)
8                  begin
9                          if(DataIn[15])
10                                 DataOut = {16'b1111111111111111, DataIn};
11                         else
12                                 DataOut = {16'b0, DataIn};
13                 end
14
15
16 endmodule
```

Listing 4.8 – Algoritmo 8 - Extensor de bit

4.2.9 Entrada e Saída

Em relação a entrada e saída de dados do processador, será utilizado um módulo de entrada e saída com o objetivo de efetuar transferências de dados entre o processador e os periféricos de I/O. Existem três possíveis formas de realizar esta ação, por interrupção, por programação e por DMA.

Tomando conhecimento das principais formas de implementação, para este projeto o I/O por programação será utilizado. Visto que ele é o que melhor se adequa às necessidades do processador e ao grau de complexidade.

4.2.9.1 I/O por programação

Ocorre a partir do resultado das instruções de I/O que estão presentes no programa de um computador. Cada transferência de dados é iniciado por uma instrução no programa.Este tipo pode admitir dois tipos de transferência:

Transferência incondicional: a transferência é realizada independentemente do estado da interface ou periférico.

Transferência condicional: a operação de E/S só é realizada se o dispositivo estiver pronto para tal (7). Em geral, o programa deste tipo de transferência possui um laço de espera que efetua constantes testes do estado do dispositivo até que o mesmo indique que a operação pode ser realizada.

4.2.9.2 I/O por interrupção

Neste modo, a transferência é acionada através de uma interrupção, ou seja, a operação De E/S é requerido pelo dispositivo externo através de um pedido de interrupção. Normalmente, este pedido é solicitado quando o dispositivo ou interface está pronto para realizar a transferência.

4.2.9.3 I/O por DMA

O último tipo é o por DMA (*Direct Memory Access*), que pode ser definido como uma técnica de transferência de dados onde os periféricos se comunicam diretamente entre si, utilizando os barramentos de memória e removendo a intervenção da CPU (8). Basicamente, o controlador DMA assume os barramentos para gerenciar diretamente a transferência entre os dispositivos de E/S e a unidade de memória.

Apesar deste módulo fazer parte da Unidade de Processamento, sua implementação será finalização no último ponto de checagem do projeto. Isso porque o seu real funcionamento acontecerá através da utilização da placa FPGA. Contudo, o código abaixo já contemplo sua primeira parte, que será finalizada adiante.

Os algoritmos deste módulos se encontram na sessão de Apêndices.

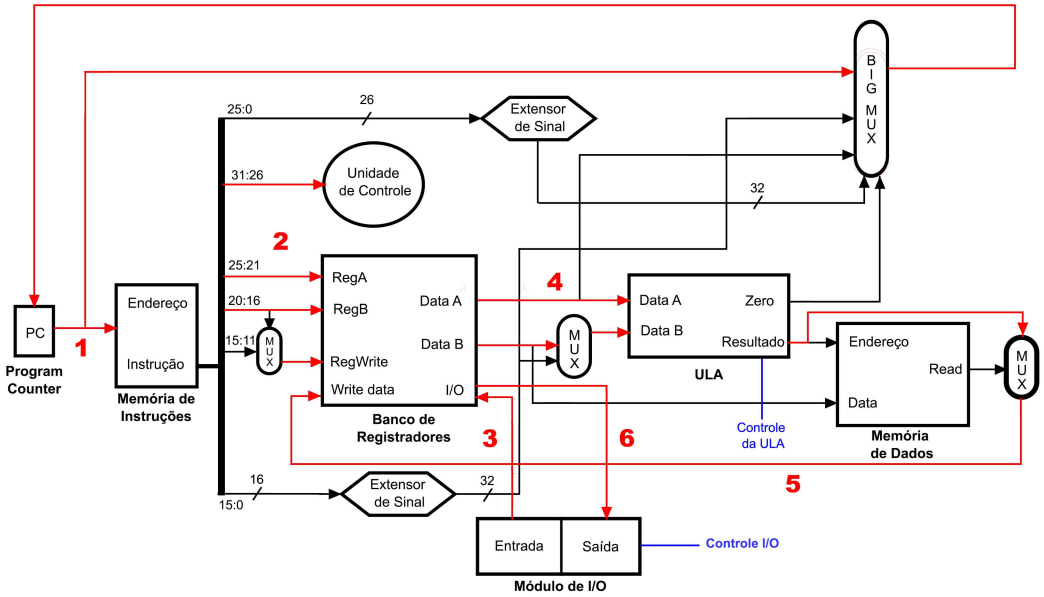
4.3 Exemplos

Para validar o desenvolvimento e funcionamento da arquitetura base proposta, foram efetuados testes no caminho de dados a partir da simulação de algumas instruções e seus diferentes tipos.

Esta etapa é importante pois servirá de guia para o funcionamento do processador ao longo de sua implementação.

4.3.1 Instruções do tipo R

Figura 5 – Caminho de dados para instruções do tipo R



Fonte: Autor

Inicialmente o primeiro teste é para instruções do tipo R, como soma e subtração. Vale ressaltar que todo o funcionamento dos passos citados são orquestrados a partir da unidade de controle que recebe o opcode e determina quais serão os passos adotados no *datapath*.

Passo 1: Inicialmente o endereço da instrução que está disponível no PC é encaminhado para a Memória de Instruções. Prosseguindo, o endereço atual do PC é incrementado e ao ser selecionado como saída pelo Big MUX, este novo endereço atualizado é retornado ao módulo do *Program Counter*. Vale ressaltar que este novo código referencia a próxima instrução que será executada.

Passo 2: Ao mesmo tempo, o endereço da primeira etapa é interpretado pela Memória de Instruções que seleciona qual é a instrução referente a este código e envia estas informações ao Banco de Registradores e a Unidade de Controle.

Passo 3: Em seguida, o valor inserido através do módulo de entrada é salvo no Banco de Registradores.

Passo 4: Instruções do tipo R, possuem três registradores. Dois registradores de origem (RegA e RegB), que recebem os dados que servirão como argumento para a ULA efetuar determinada operação. E um registrador (RegWrite) de destino, que recebe o resultado da operação realizada.

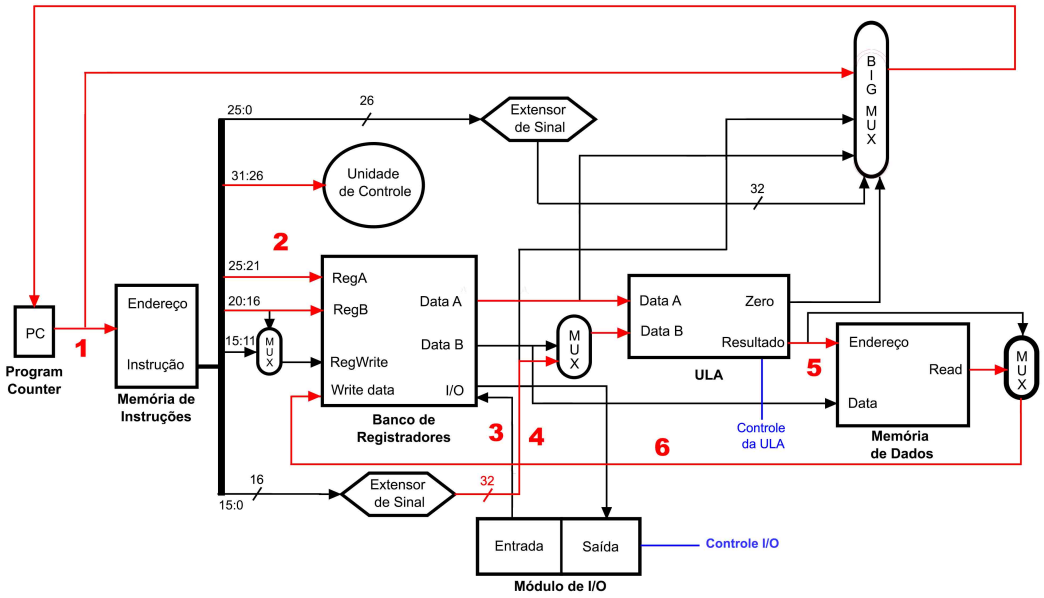
Passo 5: O dois valores dos registradores de origem são transferidos para a ULA

através dos campos Data A e Data B.

Passo 6: A partir do opcode da instrução, a ULA irá determinar qual operação realizar com os dados e o respectivo resultado é encaminhado para Memória de Instruções (mais especificamente no RegWrite), após ser selecionado como saída do MUX. Além disso, este resultado é mostrado no *display* do FPGA através do módulo de saída.

4.3.2 Instrução *Load Word*

Figura 6 – Caminho de dados para instrução lw



Fonte: Autor

Outra instrução pertinente é a lw, que possui a função de de movimentação de dados da memória para registrador(9).

Os passos 1 e 2 são os mesmos citados na sessão 4.1

Passo 3: A instrução *lw* utiliza apenas dois registradores (RegA e RegB), o registrador de endereço base é transferido como dado de saída do Banco de Registradores (Data A).

Passo 4: Paralelamente a isso, um imediato de 16bits é estendido pelo módulo de Extensor de Sinal e através do MUX, este sinal estendido serve como entrada para ULA (mais especificamente na entrada Data B).

Passo 5: A partir do opcode da instrução, a ULA irá determinar somar as informações provindas das entradas, ou seja, do sinal estendido e do registrador de endereço base.

Passo 6: O resultado desta operação é transferido para a Memória de Dados que identifica qual é o dado referente ao endereço de entrada. A partir deste endereço, o dado referente volta para o banco de registradores e é salvo.

5 Resultados e Discussões

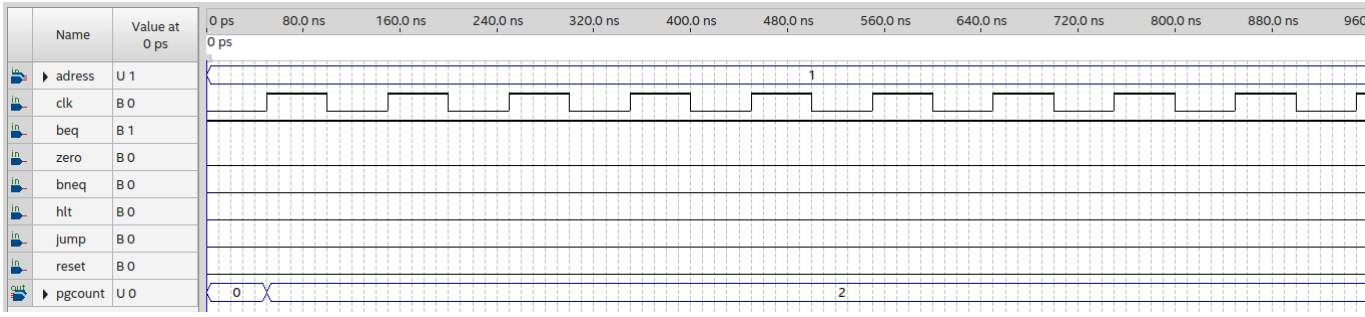
Para verificar o funcionamento e implementação nos modos apresentados anteriormente, efetuou-se simulações em duas etapas: primeiramente de cada módulo separadamente e em seguida com suas partes integradas. Em ambas as opções, seus resultados foram observados a partir de *waveforms* gerados pelo *Quartus*

5.1 Módulos separados

5.1.1 Contador de Programa

O primeiro módulo implementado foi o PC, seu funcionamento padrão é receber um endereço (adress) e somar uma unidade que será retornada em **pgcount**. As Figura 7 abaixo, apresenta o caso padrão de quando o PC é somado.

Figura 7 – *Waveform 1:* Contador de Programas

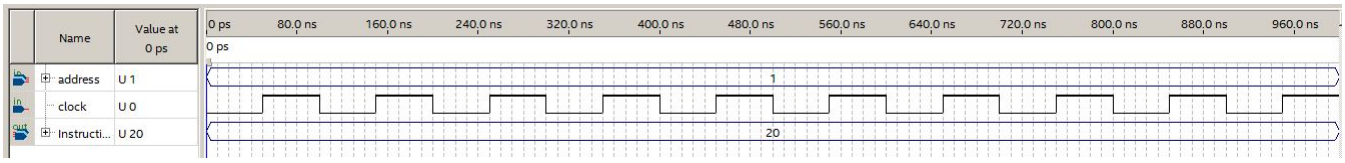


Fonte: Autor

5.1.2 Memória de Instruções

Para testar a Memória de Instruções, foram escolhidos dois endereços, o primeiro equivalente ao valor decimal de retorno 20 (referente a instrução *add*) e o segundo, ao valor 50 (referenta a instrução *sub*). Ao selecionar qual seria o endereço de entrada, o valor é atualizado pelo PC e encaminhado para a memória de instruções que recebe este dado ea cada subida de *clock* retorna como a saída a instrução respectiva.

Figura 8 – *Waveform 1:* Memória de Instruções



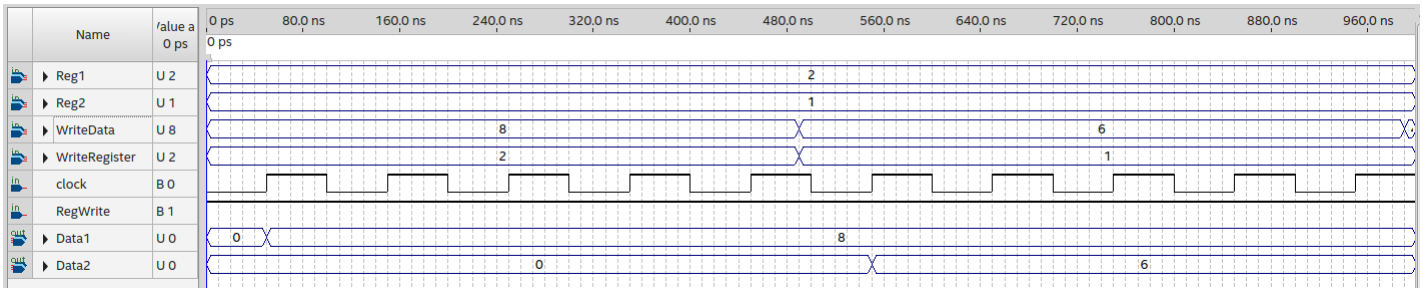
Fonte: Autor

5.1.3 Banco de Registradores

Para testar o banco de registradores, considerando que o sinal de controle **RegWrite** está ativo, significa que o dado de entrada da váriavel **WriteData** será salvo no vetor **Register**, na posição de memória dada pelo **WriteRegister**. No exemplo abaixo, foi salvo o valor 8 no vetor Register[2] e o valor 6 na posição Register[1].

Para o caso da leitura dos dados, as variavéis **Reg1** e **Reg2** indicavam qual endereço do dado do vetor Register, **Data1** e **Data2** iriam receber respectivamente. Para quesitos de simplificação, Reg1 e Reg2 receberam o mesmo valor do endereço dado pelo WriteRegiste na escrita dos dados, para que dessa forma fosse possível conferir se os dados escritos, também poderiam ser lidos corretamente.

Figura 9 – *Waveform 1:* Banco de Registradores



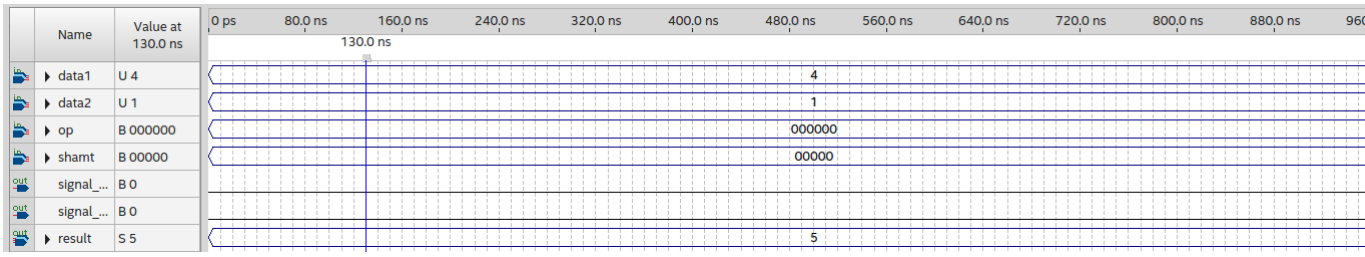
Fonte: Autor

5.1.4 Unidade Lógica e Aritmética

Para testar a Unidade Lógica e Aritmética, considerou-se três possíveis tipos de operações. Onde cada instrução era selecionada a partir de seu respectivo *opcode*.

A primeira, é referente a operação de soma. Ao selecionar esta operação a partir do *opcode* 00000, os valores das variáveis **Data1** e **Data2** eram somados e o resultado apresentou-se correto. Além disso, como o resultado da operação não é nulo e nem possui sinal negativo, suas respectivas *flags* não foram setadas.

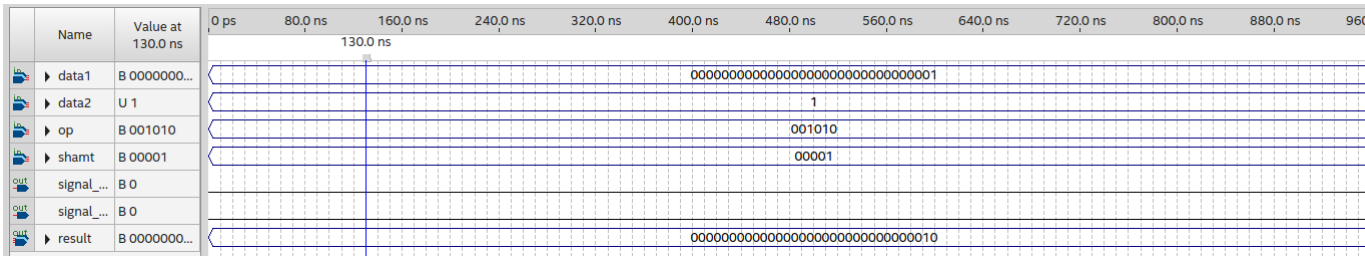
Figura 10 – *Waveform 1:* Unidade Lógica e Aritmética



Fonte: Autor

A segunda operação refere-se a instrução *shift left* (shfl) que efetua um deslocamento de bits para a esquerda. A quantidade de bits a ser deslocada é dada pelo campo **shamt**. Portanto, o resultado apresentado foi do valor binário de **Data1** com um bit deslocado para a esquerda.

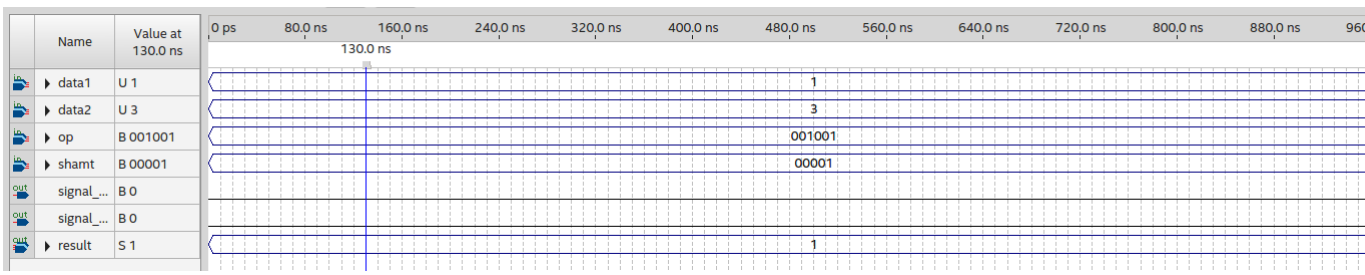
Figura 11 – *Waveform 2:* Unidade Lógica e Aritmética



Fonte: Autor

Finalizando, a terceira operação é a comparação *set on less than* (stl), que verifica se o valor **Data1** é menor que o de **Data2**. Como no exemplo da Figura 12, esta afirmação é verdadeira, o valor retornado é igual a 1.

Figura 12 – *Waveform 3:* Unidade Lógica e Aritmética

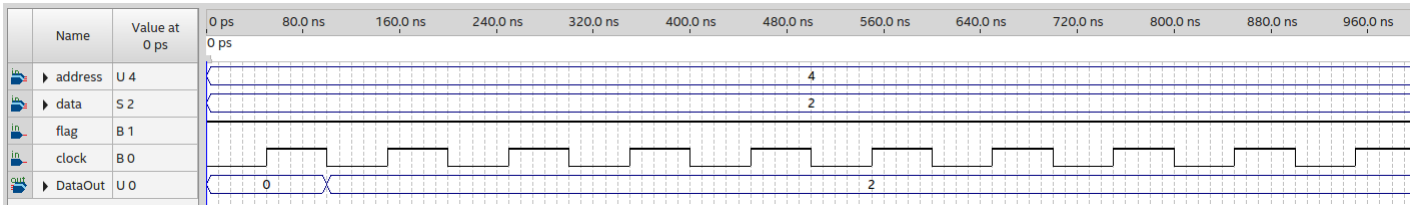


Fonte: Autor

5.1.5 Memória de Dados

Para testar a Memória de Dados, após a descida do *clock* o valor presente em **data** foi salvo na posição dada por **adress**. Além disso, **DataOut** indica qual dado foi salvo na memória.

Figura 13 – *Waveform 1:* Memória de Dados



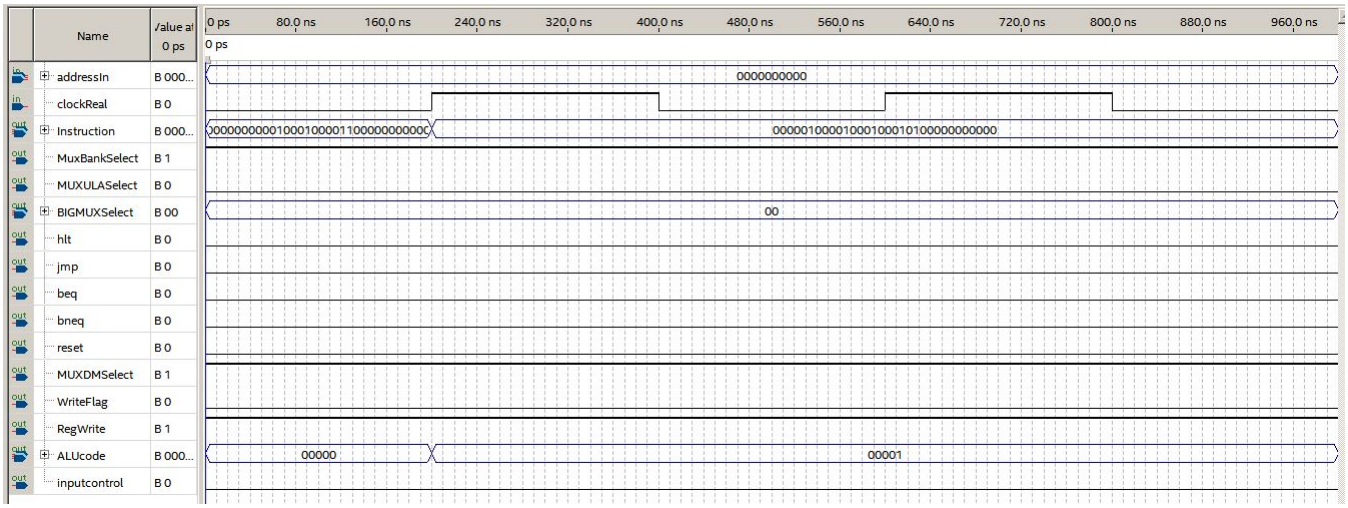
Fonte: Autor

5.1.6 Unidade de Controle

No caso da Unidade de Controle, sua validação consiste em verificar se os sinais de controle de saída estão condizentes com o respectivo valor da instrução.

Na Figura 14, o teste realizado foi para as instruções **add** e **sub**. Ao inserir manualmente 0 em **addressIn**, este valor era incrementado no PC e os sinais de controle retornados eram referentes a instrução de subtração.

Figura 14 – *Waveform 1:* Unidade de Controle



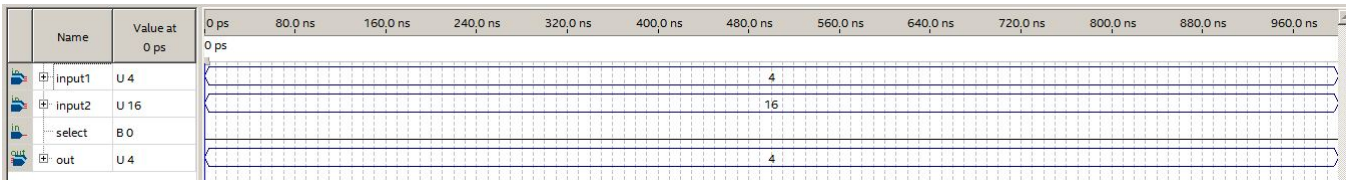
Fonte: Autor

5.1.7 MUX e BigMux

Inicialmente, o funcionamento de todos os multiplexadores do processador é o mesmo. Consistindo em um sinal seletor que determina qual será sua saída dada duas

determinadas entradas. A Figura 15 apresenta um caso de teste que é semelhante para todos os outros MUX presentes na arquitetura.

Figura 15 – *Waveform 1:* Mux de Entrada da ULA

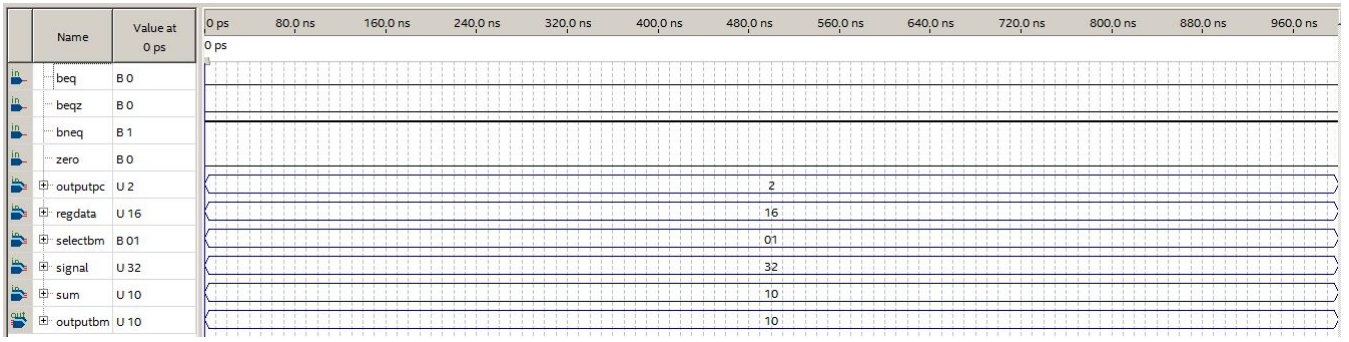


Fonte: Autor

O funcionamento do módulo BigMux é semelhante a todos os multiplexadores citados, sua única diferença é o tratamento das instruções do tipo *branch* e *jump*. Vale lembrar que este módulo é responsável por indicar qual será a entrada do PC.

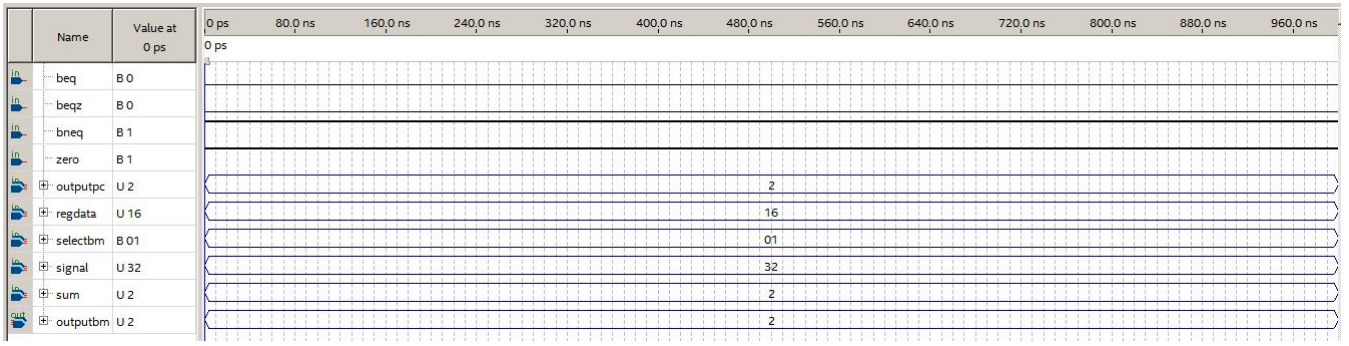
As Figuras 16 e 17 apresentam respectivamente os casos da instrução *banch on not equal*. Um sinal pronvindo da **selectbm** indica quando os casos de *branch* deverão ser considerados. Nos exemplos abaixo, quando **bneq** e **zero** são igual a 1, o sinal é o endereço de saída provindo do próprio PC ou se será o valor de saída da ULA.

Figura 16 – *Waveform 2: BigMux (bneq=1 e zero=0)*



Fonte: Autor

Figura 17 – *Waveform 3: BigMux (bneq=1 e zero=1)*

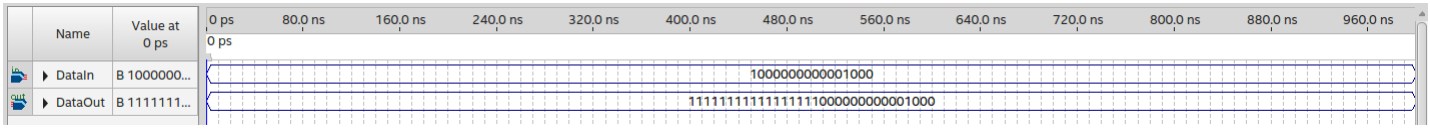


Fonte: Autor

5.1.8 Extensor

O ultimo módulo é o Extensor de *Bits*, que no exemplo abaixo representa uma entrada de 16 *bits* com *bit* mais significativo igual a 1.

Figura 18 – *Waveform 1: BigMux (jump register com saído dada por regdata)*



Fonte: Autor

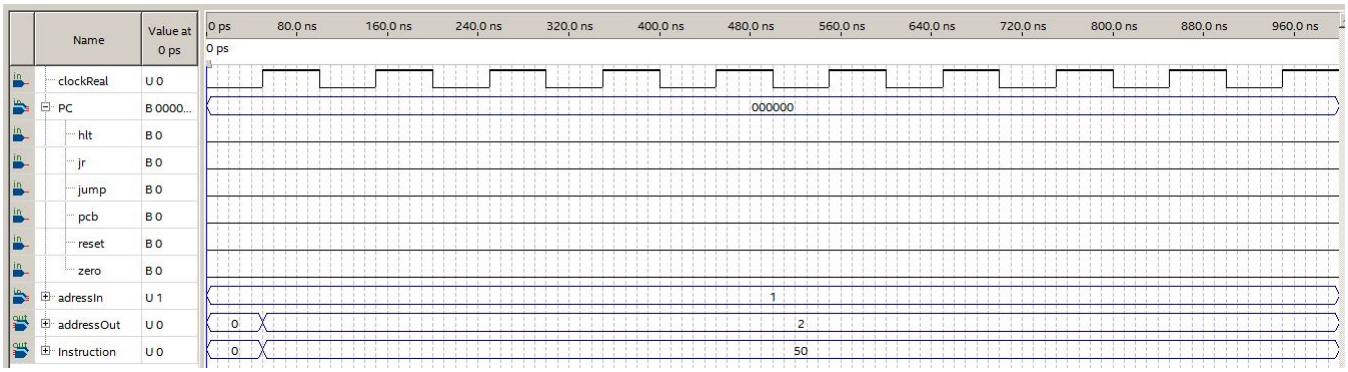
5.2 Módulos unificados

Após a implementação e testes de cada módulo da arquitetura-base, juntou-se todas as partes, para que as mesmas funcionassem de forma unificada. A implementação completa deste módulo encontra-se na sessão de Apêndices. Apesar da implementação em *Verilog*, é possível comparar esta etapa em conectar fios em cada módulo, para que a entrada de um fosse a saída do outro e que tudo estivesse interconectado. Para facilitar o processo de implementação, cada módulo foi testado na ordem apresentada pela arquitetura base, começando pelo PC e indo até o *BigMux*. E o número de componentes conectados crescia módulo a módulo.

5.2.1 *Program Counter* e Memória de Instruções

Começando pela integração do PC com a Memória de Instruções, ao inserir um valor de entrada em **adressIn**, este dado era somado no PC e retornado para memória de instruções através do **adressOut**. Com este valor (que no exemplo da Figura 19 é igual a 2) a Memória de Intruções encontra a intrução equivalente à posição do vetor dada por **adressOut** e retorna o valor equivalente.

Figura 19 – *PC/Instruction Memory*

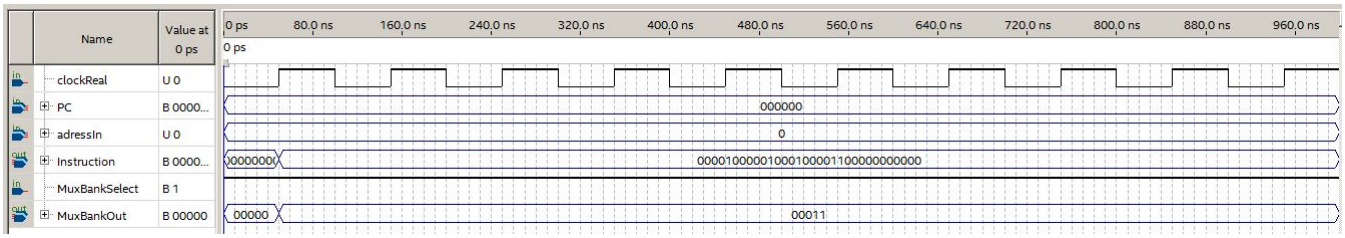


Fonte: Autor

5.2.2 Módulos anteriores e MUX Pré Banco de Registradores

Prosseguindo, parte desta instrução retornada é encaminhada para um MUX que está posicionado antes do Registradores e dependendo do estado do sinal **MuxBankSelect**, determinado dado será a saída do multiplexador. No exemplo da Figura 20, caso **MuxBankSelect** fosse igual a zero, o valor retornado em **MuxBankOut** seria os *bits* 15 a 11 do **Instruction**, já para o caso igual a 1, o valor retornado seria os *bits* 16 a 20.

Figura 20 – *PC/Instruction Memory/Mux pré Banco de Registradores*

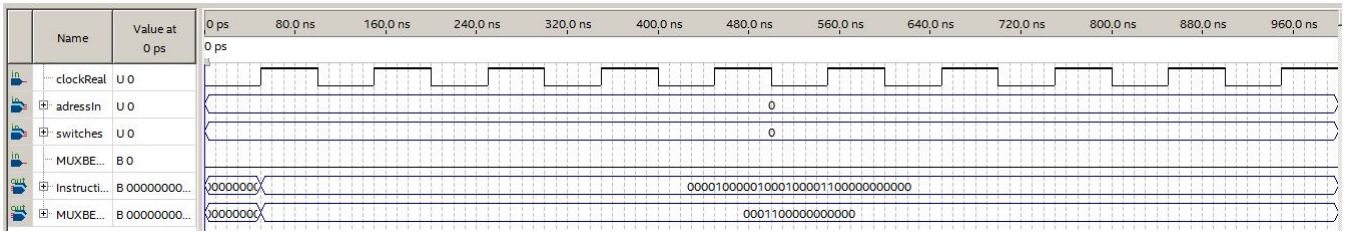


Fonte: Autor

5.2.3 Módulos anteriores e MUX Pré Extensor de *Bits*

O mesmo caso ocorre com o multiplexador posicionado antes do Extensor de *Bits*. Caso o sinal de controle **MUXBESelect** fosse igual à zero ou um, o valor de saída era um determinado valor de *bits* de **Instruction**, equivalente aos *bits* 0 a 15.

Figura 21 – *Módulos anteriores/Mux pré Bit Extender*

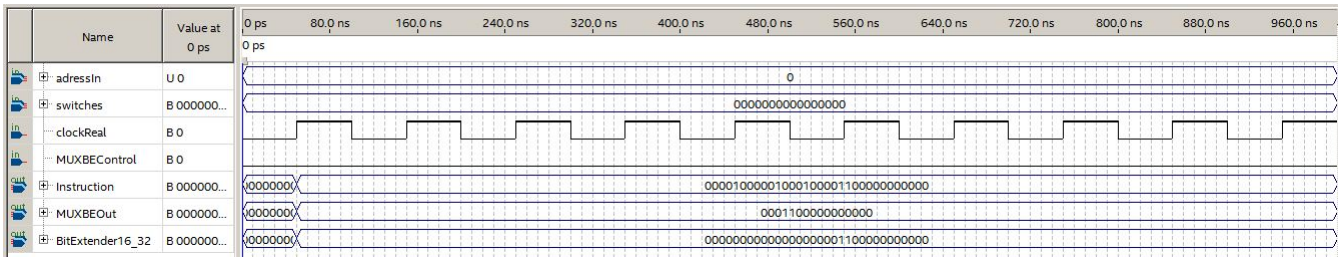


Fonte: Autor

5.2.4 Módulos anteriores e Extensor de Bits

Prosseguindo, o valor de saída dado por **MUXBEOut** era encaminhado como entrada do Extensor de *Bits*, que concatenava estes 15 *bits* e retornava como saída o valor de entrada acrescentado de quinze bits igual a zero.

Figura 22 – *Módulos anteriores/Mux pré Bit Extender/Bit Extender*



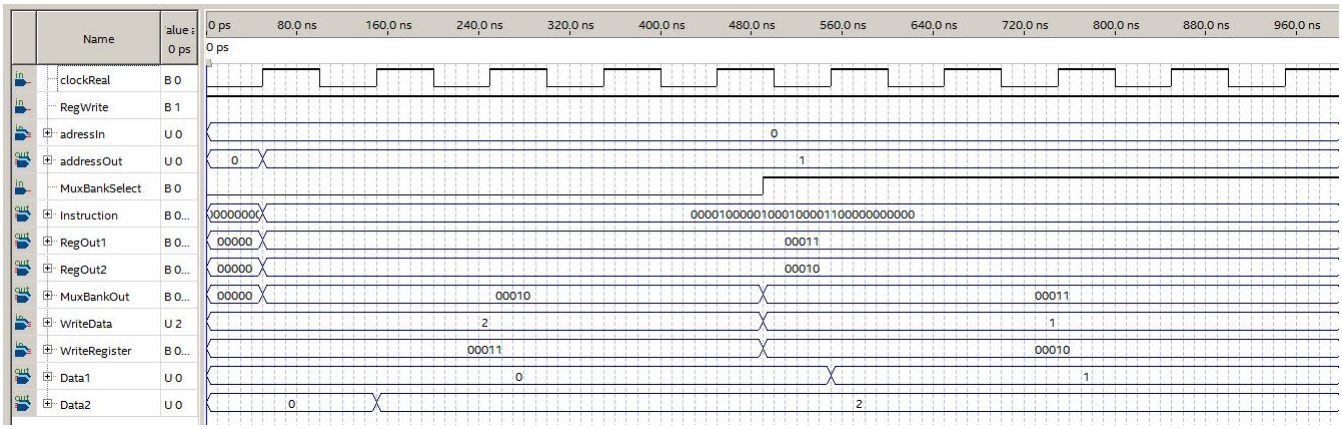
Fonte: Autor

5.2.5 Módulos anteriores e Banco de Registradores

Os testes do Banco de Registradores é semelhante à sessão 5.1.3, a Figura 23 exemplifica a escrita de dados no registrador. Como o sinal **RegWrite** está ativo todo dado de entrada (dado pela variável **WriteData**) será salvo na posição do registrador dada por **MuxBankOut**.

As outras váriaveis da Figura 23 são utilizadas apenas para quando desejamos resgatar algum dado do registrador. É possível resgatar dois dados (através das variáveis **Data1** e **Data2**) por vez através do Banco de Registradores. A posição do registrador à qual o dado a ser retornado se encontra, é referenciado através de **RegOut1** e **RegOut2**.

Figura 23 – *Módulos anteriores/Banco de Registradores*



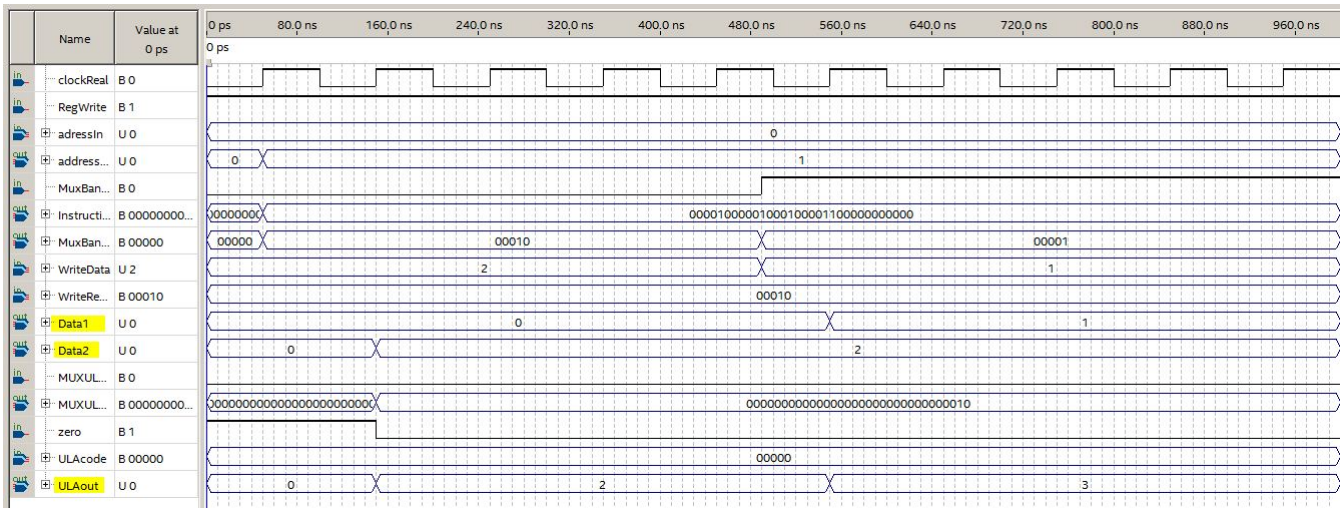
Fonte: Autor

5.2.6 Módulos anteriores e ULA

Apesar da quantidade de informações inseridas na Figura 24, as variáveis relevantes para o teste de adição estão destacadas em amarelo. O valor retornado pela **ULAout** é referente à soma dos valores de **Data 1** e **Data 2**.

Note que está operação só ocorre devido ao código de entrada **ULAcode**. Que é interpretado pela ULA e efetua a operação equivalente.

Figura 24 – *Módulos anteriores/ULA (instrução add)*

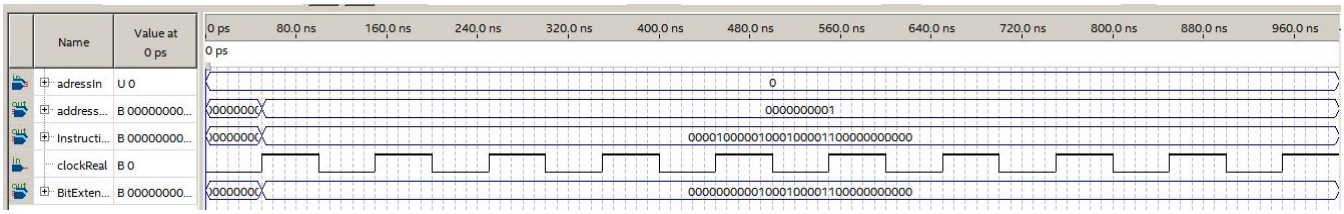


Fonte: Autor

5.2.7 Módulos anteriores e Extensor de Bits

O Extensor de Bits da Figura 25, funciona da mesma forma apresentada na sessão 5.2.7. A unica diferença é que o valor de entrada possui 26 *bits* (dado pelos *bits* 0 a 25 da variável **Instruction**) e é extendido para 32.

Figura 25 – *Módulos anteriores/ULA/Bit Extender (26 para 32 bits)*

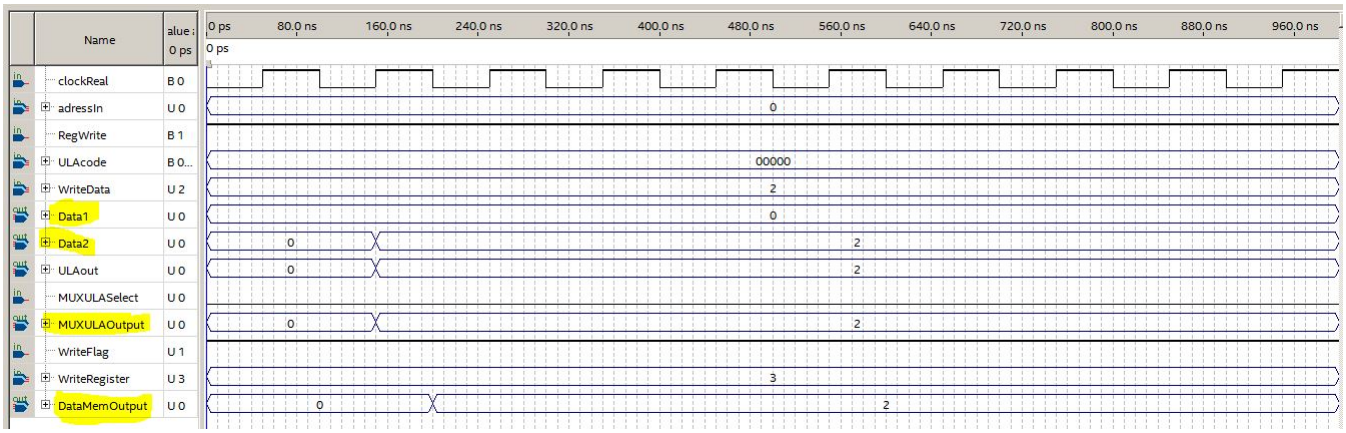


Fonte: Autor

5.2.8 Módulos anteriores e Memória de Dados

No caso da memória de dados,o módulo recebe como entrada o valor da ULA que servirá como endereço do vetor onde será salvo o dado recebido do Banco de Registradores (**Data 2**). Além disso, um sinal de controle chamado **WriteFlag** irá determinar quando o dado deverá ou não ser salvo. No exemplo da Figura 26, o valor 2 provindo da variável **Data 2** é salvo na posição 2 (**MUXULAOuput**) do vetor. Este mesmo valor é retornado através da **DataMemOutput**.

Figura 26 – *Módulos anteriores/Memória de Dados*



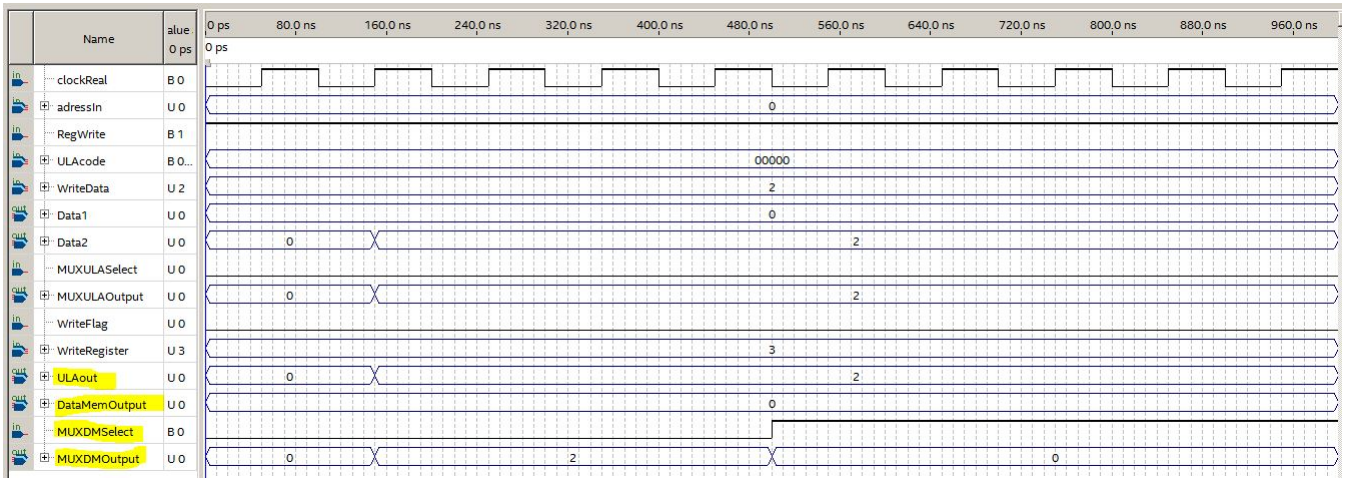
Fonte: Autor

5.2.9 Módulos anteriores e Mux Pós Memória de Dados

O multiplexador pós Memória de Dados é responsável com selecionar se o dado encaminhado para o Banco de Registradores será a saída da ULA ou da Memória de Dados.

No exemplo da Figura 27 ao variar o sinal de controle **MUXDMSelect**, o dado de saída em **MUXDMOutput** será 2 (equivalente a saída da ULA) ou será 0 (equivalenta a saída da Memória de Dados).

Figura 27 – *Módulos anteriores/Memória de Dados/Mux pós Memória de Dados*



Fonte: Autor

5.3 Placa FPGA

Com todos os testes dos módulos efetuados através da *waveform*, os últimos passos foram compilar o projeto na placa FPGA e mapear todos os componentes utilizados como display, leds e botões. E também validar o funcionamento das instruções através de algoritmos implementados na Memória de Instruções.

5.3.1 Algoritmo de Fibonnacci

A Sequência de Fibonacci é a sequência numérica proposta pelo matemático Leonardo Pisa, conhecido como Fibonacci (10). Esta sequência é definida mediante a soma dos dois numerais antecedentes a um elemento Fn. Por exemplo, a sequência de Fibonacci de Fn = 5 é igual a 3 + 2. Logo, a sequência é dada por **1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...**

O objetivo do algoritmo proposto é dado um valor de entrada provindo dos *switches* do FPGA, calcular e retornar o valor da respectiva sequência de Fibonacci. Além disso, este algoritmo foi escolhido pois através dele é possível utilizar todos os grupos de instruções propostas, como lógicas, aritméticas, instruções de *branch* e *jump*, além de outras instruções como *halt* e I/O.

Figura 28 – *Implementação do programa proposto na Memória de Instruções*

```
//FIBONACCI
//1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89
mem[0]  = 32'b010100_00000_00000_00000_00000_000000; // nop
mem[1]  = 32'b001101_00000_00001_00000_00000_000001; // li r[1] = Im(1)
mem[2]  = 32'b001101_00000_00010_00000_00000_000001; // li r[2] = Im(1)
mem[3]  = 32'b001101_00000_00100_00000_00000_000000; // li r[4] = Im(0)
mem[4]  = 32'b010101_00000_00011_00000_00000_000000; // in reg[3] <= Valor
mem[5]  = 32'b010110_00011_00000_00000_00000_000000; // out reg[3]
mem[6]  = 32'b000011_00011_00011_00000_00000_000001; // subi r[3] = r[3] - Im(1)
mem[7]  = 32'b001001_00000_00011_01010_00000_000000; // slt r[10] = r[0] < r[3]
mem[8]  = 32'b001111_01010_00000_00000_00000_001101; // beq r[10] == r[0], PC = 13
mem[9]  = 32'b000000_00001_00100_00010_00000_000000; // add r[2] = r[1] + r[4]
mem[10] = 32'b000000_00000_00001_00100_00000_000000; // add (move) r[4] = r[0] + r[1]
mem[11] = 32'b000000_00000_00010_00001_00000_000000; // add (move) r[1] = r[0] + r[2]
mem[12] = 32'b010010_00000_00000_00000_00000_000110; // jump PC = 6
mem[13] = 32'b001110_00000_00010_00000_00000_000011;  // sw MEM[r[0] + Im(3)] = r[2]
mem[14] = 32'b001100_00000_01000_00000_00000_000011; // lw r[8] = MEM[r[0] + Im(3)]
mem[15]  = 32'b000101_01000_10000_00000_00000_000000; //not r[16] = ~r[8]
mem[16]  = 32'b000101_10000_10000_00000_00000_000000; //not r[16] = ~r[16]
mem[17]  = 32'b010110_10000_00000_00000_00000_000000; // out reg[16] //SAIDA = 0
mem[18] = 32'b010111_00000_00000_00000_00000_000000;    //hlt
```

Fonte: Autor

A Figura 28 apresenta a implementação do problema proposto, presente na Memória de Instruções. A cada vez que o valor do PC é atualizado a posição do vetor **mem** é executada e cada posição representa uma instrução do programa.

Inicialmente, após a instrução de **nop** ser executada, ou seja, nenhuma ação é realizada pelo processador. Três valores imediatos são carregados nos registradores 1, 2 e 4, através da instrução **li**.

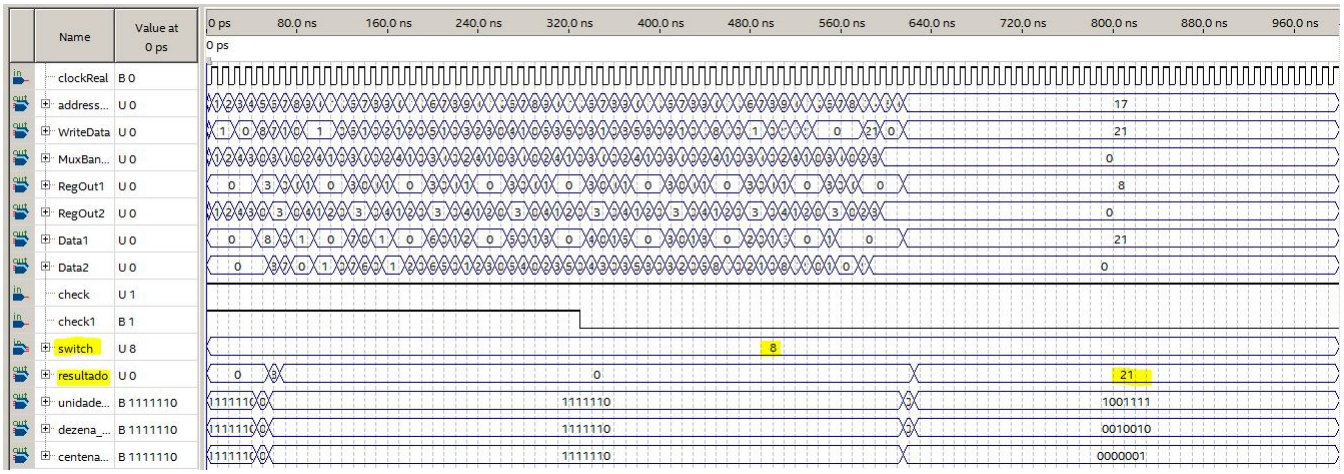
Prosseguindo, o valor a ser calculado pelo algoritmo é requisitado através da instrução **in** e é salvo no registrador 3. Vale ressaltar que este valor é o mesmo inserido através das chaves do FPGA, além disso um botão é utilizado como sinal de interrupção é utilizado para que o valor do PC só seja atualizado após a inserção dos dados. O mesmo ocorre para instrução **out**, que no caso da linha 5 do programa, apenas mostra no *display* o valor inserido.

Em seguida, o valor inserido anteriormente é decrementado de uma unidade pela instrução **subi**. E verificou-se através da instrução **slt** se o valor presente no registrador 3 é menor que zero. Caso esta afirmação fosse verdadeira, o valor salvo em r[10] seria 1, e 0 caso contrário.

A partir disso, uma comparação é feita entre o valor inserido em r[10] com o valor de r[0] que por padrão é 0. Caso a afirmação seja falsa, a próxima instrução é executada. Ou seja, o registrador 2 recebe o conteúdo presente em r[1]+r[4] e os registradores r[4] e r[1] recebem o valor de r[1] e r[2] respectivamente. Após isso, uma instrução de **jump** retorna o valor do PC para a instrução número 6 e este laço de execução entre as instruções 6 e 12 é executado até que na instrução de *branch on equal* o conteúdo de r[3] seja igual a 0.

Se isto acontecer, o PC é atualizado para 13 e significa que o resultado da sequência está contido em r[2]. Prosseguindo, o valor do resultado é salvo na posição 0 da memória através da instrução **sw** e o registrador r[8] recebe este conteúdo salvo através da instrução **lw**. Finalizando o algorimo, com o intuito de testar instruções lógicas o contido em r[8] é negado duas vezes e salvo em r[16]. Por último, uma instrução de **out** retorna o valor calculado e a instrução de **halt** finaliza a execução.

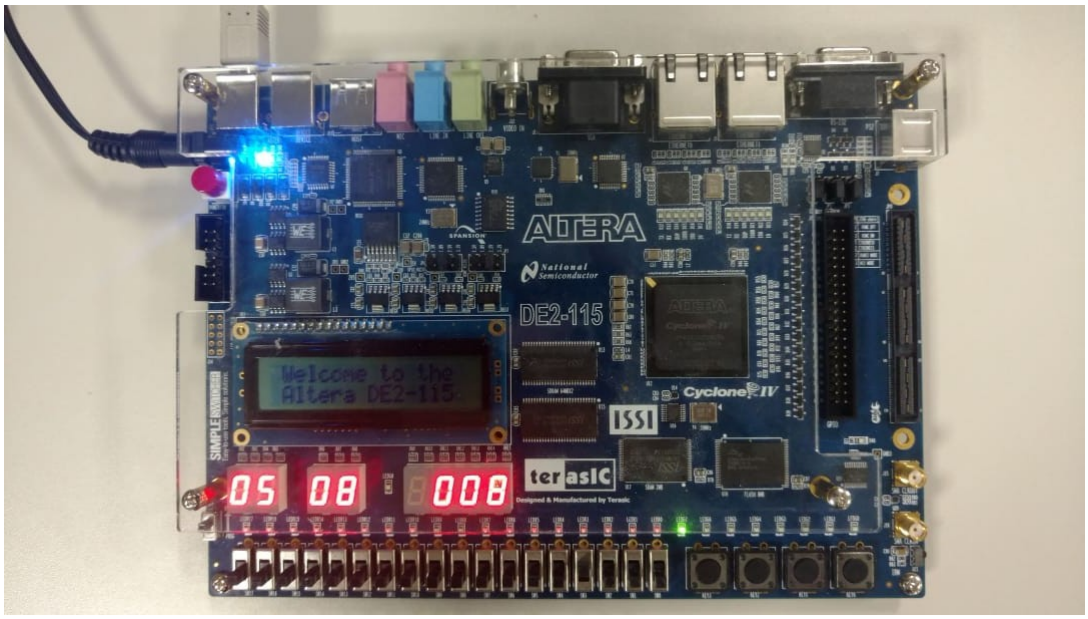
Figura 29 – *Waveform* da execução do algoritmo



Fonte: Autor

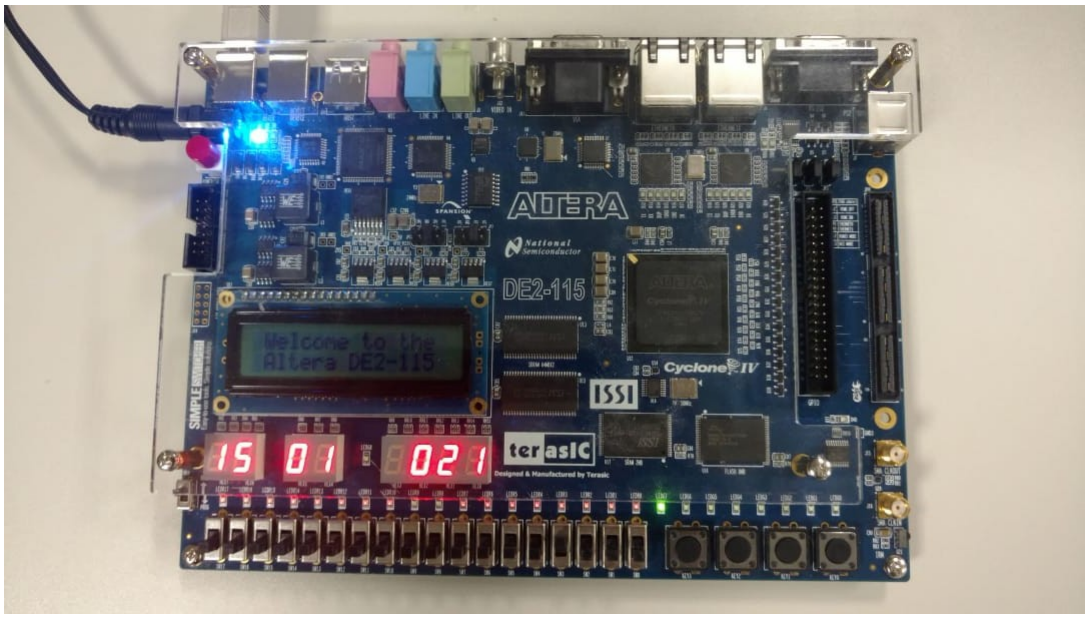
A Figura 29 apresenta a excução do algorimo para a entrada igual a 8 (inseridos através do **switch**). O valor é calculado e retornado para o *display* através do **resultado**. Note que o valor de 21 é exibido através do código BCD nas variáveis **unidade**, **dezena** e **centena**.

Figura 30 – Entrada de dados no FPGA



Fonte: Autor

Figura 31 – Resultado do algoritmo no FPGA



Fonte: Autor

As Figuras 30 e 31 apresentam a inserção de dados no FPGA e em resulta o

resultado apresentado pelo algoritmo. Observe que há três grupos de *displays*. O primeiro deles da esquerda para direita, indica a saída do PC, ou seja, qual instrução esta em execução. O segundo, apresenta os dois valores de saída do Banco de Registradores e o terceiro, os resultados da operação.

6 Considerações Finais

Analisando as etapas anteriores deste relatório, é possível perceber que desenvolver um sistema computacional pode ser algo complexo. E para esta arquitetura não foi diferente.

Durante o desenvolvimento deste relatório, a principal dificuldade encontrada foi interligar todos os módulos, principalmente e relação aos testes, visto que conforme o número de módulos já conectados aumentavam, maior era o número de váriveis necessárias para verificar. Requisitando extrema atenção em cada ponto das formas de onda geradas.

Outro ponto de extrema dificuldade, foi em relação ao mapeamento das chaves utilizadas no FPGA. Visto que devido a um erro nesta etapa, uma das instruções contempladas pela arquitetura apresentou erro de funcionamento. Apesar disso, o desenvolvimento deste projeto demonstrou-se proveitoso e permitiu consolidar todo conhecimento teórico apresentado na disciplina de Arquitetura e Organização de Computadores.

Com este projeto devidamente implementado, os próximos passos é utilizar o processador como módulo integrante do sistema computacional desenvolvido ao longo do curso de Engenharia de Computação da UNIFESP - Universidade Federal de São Paulo. Concluindo, gostaria de agradecer o apoio de todos os colegas de classe, professores e monitoria que contriuiram de alguma forma para o desenvolvimento deste projeto.

Referências

1 MAXIEDUCA. *Descubra a importância da memória cache para o computador*. 2017. Disponível em: <http://blog.maxieduca.com.br/memoria-cache-computador/>. Acesso em: 13/04/2018. Citado na página 11.

2 MACEDO, D. *Arquitetura: Von Neumann Vs Harvard*. 2012. Disponível em: <http://www.diegomacedo.com.br/arquitetura-von-neumann-vs-harvard/>. Acesso em: 13/04/2018. Citado 2 vezes nas páginas 12 e 13.

3 PATTERSON, D. A.; HENNESY, J. L. *Computer Organization and Design*. 5th edition. ed. Waltham/MA, EUA: Morgan Kaufmann, 2007. Citado na página 16.

4 OLIVEIRA, C. *Sequencia de Fibonacci*. 2011. Disponível em: <https://www.todamateria.com.br/sequencia-de-fibonacci/>. Acesso em: 09/07/2018. Citado na página 16.

5 COMPUTER Hardware/Software Architecture. [S.l.]: Morgan Kaufmann, 1998. Citado na página 21.

6 UFMT. *Circuito Combinacional*. 2016. Disponível em: <http://araguaia2.ufmt.br/professor/disciplina_arquivo/90/201608301202.pdf>. Acesso em: 13/04/2018. Citado na página 25.

7 MARTINO, P. J. M. D. *Interface de Entrada e Saída*. 2004. Disponível em: <http://www.dca.fee.unicamp.br/courses/EA078/1s2004/arquivos/turma_ab/cap7.pdf>. Acesso em: 10/04/2018. Citado na página 28.

8 GEEKS, G. for. *I/O Interface (Interrupt and DMA Mode)*. 2016. Disponível em: <https://www.geeksforgeeks.org/io-interface-interrupt-dma-mode/>. Acesso em: 11/04/2018. Citado na página 29.

9 CENTODUCATTE, P. C. *Conjunto de Instruções MIPS*. 2002. Disponível em: <http://www.ic.unicamp.br/~pannain/mc542/aulas/ch3_arq.pdf>. Acesso em: 08/04/2018. Citado na página 31.

10 MATéRIA, T. *Dispositivos Lógicos Programáveis*. 2011. Disponível em: <http://www.feg.unesp.br/Home/PaginasPessoais/ProfMarceloWendling/logica-programavel.pdf>. Acesso em: 09/07/2018. Citado na página 43.

Apêndices

APÊNDICE A – Algoritmo de integração dos módulos

```
1  module Processador (
2
3                                                          centena_out, dezena_out,
                                                               unidade_out, switch,
4                                                          clock, check, check1, reset,
                                                               dezena_pcOUT, unidade_pcOUT,
                                                               insinal, beq, zero,
5                                                          unidade_regOUT, unidade_regOUT2
6
7                                                      );
8
9
10         wire clockReal;
11    input wire reset;
12         wire PCBranch, jr;
13    wire [31:0]addressIn;
14    wire [31:0]addressOut;
15
16    wire [31:0] Instruction;
17
18   wire MuxBankSelect;
19   wire [4:0] MuxBankOut;
20
21   wire MUXBEControl;
22   wire [15:0] MUXBEOut;
23
24
25    wire [31:0] BitExtender16_32;
26
27
28    wire [31:0] BitExtender26_32;
29
30
31    wire [31:0] WriteData;
32    wire [31:0] Data2;
33    wire [31:0]Data1;
34    wire [4:0] RegOut1, RegOut2;
35         wire [31:0] RegDataWrite;
36
37   wire [31:0] MUXULAOutput;
38   wire MUXULASelect;
39
40
41         wire  jmp;
42         wire  WriteFlag;
43         wire  RegWrite;
44         wire [4:0] ALUcode;
45         wire inputcontrol;
46    wire halt;
47
48
```

```
49     wire [31:0] ULAout;
50     output wire zero;
51
52          wire [31:0] DataMemOutput;
53
54          wire [31:0] MUXDMOutput;
55          wire MUXDMSelect;
56
57           wire bneq;
58           wire beqz;
59           wire [1:0]  BIGMUXSelect;
60           output wire beq;
61
62
63           wire inputControl;
64     input wire [4:0] switch;
65           wire [15:0] MUXINOutput;
66          input check1;
67          input check;
68
69
70    wire negative;
71
72
73          wire [3:0] centena, dezena, unidade;
74      output wire [6:0] centena_out, dezena_out, unidade_out;
75           wire [31:0] resultado;
76
77
78     wire [3:0] centena_pc, dezena_pc, unidade_pc;
79          output  wire [6:0] dezena_pcOUT, unidade_pcOUT;
80          wire [6:0] centena_pcOUT;
81
82
83                  wire [3:0] centena_reg, dezena_reg, unidade_reg;
84       wire [6:0] centena_regOUT, dezena_regOUT;
85            output wire [6:0] unidade_regOUT;
86
87           wire [3:0] centena_reg2, dezena_reg2, unidade_reg2;
88       wire [6:0] centena_regOUT2, dezena_regOUT2;
89           output wire [6:0] unidade_regOUT2;
90
91           wire pin_button;
92           input wire clock;
93           output wire insinal;
94           wire outputControl;
95
96
97
98
99  Temporizador Temp (
100                                                          .clk_auto(clock),
101                                                          .clk(clockReal)
102                                                  );
103
104
105 Program_Counter PC (
106                 .clock(clockReal),
107                 .adress(addressIn),
108                 .reset(reset),
```

```
109                 .hlt(halt),
110                 .pgcount(addressOut),
111                 .jump(jmp),
112                 .jr(jr),
113                 .insign(insinal)
114 
115                 );
116 
117 assign insinal = ((outputControl==1 && check1==0) || (check ==0 && inputControl==1)) ? 1
        : (halt ==1) ? 1 : 0;
118 
119 MemoryInstruction InstrMem (
120                                                     .address(addressOut[9:0]),
121                                                     .InstructionOut(Instruction),
122                                                     .clock(clockReal));
123 
124 MuxBank1 MuxBR (
125                         .input1(Instruction[20:16]),
126                         .input2(Instruction[15:11]),
127                         .out(MuxBankOut),
128                         .select(MuxBankSelect));
129 
130 
131 BitExtender BitExtender32(
132                                 .DataIn(Instruction[15:0]),
133                                 .DataOut(BitExtender16_32)
134                                 );
135 
136 
137 RegBank RegBank (.clock(clockReal),
138                                             .Reg1(Instruction[25:21]),
139                                             .Reg2(Instruction[20:16]),
140                                             .RegWrite(RegWrite),
141                                             .WriteData(WriteData),
142                                             .WriteRegister(MuxBankOut),
143                                             .Data1(Data1),
144                                             .Data2(Data2),
145                                             .Resultado(resultado),
146                                             .outputControl(outputControl),
147                                             .RegOut1(RegOut1),
148                                             .RegOut2(RegOut2),
149                                             .inputControl(inputControl),
150                                             .DataIn({28'd0, switch})
151 
152                                             );
153 
154 
155 
156 
157 ControlUnit UC (
158                                         .clock(clockReal),
159                                         .opcode(Instruction[31:26]),
160                                         .MuxBankSelect(MuxBankSelect),
161                                         .MUXULASelect(MUXULASelect),
162                                         .BIGMUXSelect(BIGMUXSelect),
163                                         .hlt(halt),
164                 .jmp(jmp),
165                                         .beq(beq),
166                                         .bneq(bneq),
167                                         .MUXDMSelect(MUXDMSelect),
```

```
168                                         .WriteFlag(WriteFlag),
169                                         .RegWrite(RegWrite),
170                                         .ULAcode(ALUcode),
171                                         .inputControl(inputControl),
172                                         .outputControl(outputControl),
173                                         .jr(jr),
174                                         .beqz(beqz)
175                                         );
176
177
178
179
180 BitExtender26 BitExtender26 (
181                                                 .DataIn(Instruction[25:0]),
182                                                 .DataOut(BitExtender26_32)
183                                                 );
184
185
186 MuxULA MUXULA(.input1(Data2),
187                                 .input2(BitExtender16_32),
188                                 .out(MUXULAOutput),
189                                 .select(MUXULASelect)
190                                 );
191
192
193 ULA ULA (.op(ALUcode),
194                         .data1(Data1),
195                         .data2(MUXULAOutput),
196                         .result(ULAout),
197                         .signal_zero(zero),
198                         .shamt(Instruction[10:6])
199                         );
200
201
202
203 DataMemory DataMemory(
204                                         .clock(clockReal),
205                                         .address(ULAout),
206                                      .flag(WriteFlag),
207                                         .DataOut(DataMemOutput),
208                                         .data(Data2)
209                                         );
210
211
212 MuxDataMem MUXDM(
213                                 .input1(DataMemOutput),
214                                 .input2(ULAout),
215                                 .out(WriteData),
216                                 .select(MUXDMSelect)
217                                 );
218
219
220 BigMux BIGMUX(.zero(zero),
221                                   .beq(beq) ,
222                                   .bneq(bneq),
223                                   .beqz(beqz),
224                                   .selectbm(BIGMUXSelect),
225                                   .outputbm(addressIn),
226                                   .outputpc(addressOut),
227                                   .sum(BitExtender16_32),
```

```
228                                  .signal(BitExtender26_32),
229                                  .regdata(Data1)
230
231                                 );
232
233
234
235 Conversor_BCD PC_IOModule(
236                                                                  .bin(addressOut[7:0]),
237                                                                  .centena(centena_pc),
238                                                                  .dezena(dezena_pc),
239                                                                  .unidade(unidade_pc));
240
241 InOut_Module InOut_PC (
242                                                          .centena(centena_pc),
243                                                          .dezena(dezena_pc),
244                                                          .unidade(unidade_pc),
245                                                          .out(1'b1),
246                                                          .centena_out(centena_pcOUT),
247                                                          .dezena_out(dezena_pcOUT),
248                                                          .unidade_out(unidade_pcOUT));
249
250
251
252
253 Conversor_BCD reg_address_value(
254                                        .bin(Data1),
255                                                .centena(centena_reg),
256                                  .dezena(dezena_reg),
257                                  .unidade(unidade_reg));
258
259 InOut_Module InOut_Reg (
260                                 .centena(centena_reg),
261                             .dezena(dezena_reg),
262                             .unidade(unidade_reg),
263                             .out(1'b1),
264                             .dezena_out(dezena_regOUT),
265                             .unidade_out(unidade_regOUT));
266
267
268 Conversor_BCD reg_address_value2(
269                                        .bin(MUXULAOutput),
270                                                .centena(centena_reg2),
271                                  .dezena(dezena_reg2),
272                                  .unidade(unidade_reg2));
273
274 InOut_Module InOut_Reg2 (
275                                 .centena(centena_reg2),
276                             .dezena(dezena_reg2),
277                             .unidade(unidade_reg2),
278                             .out(1'b1),
279                             .dezena_out(dezena_regOUT2),
280                             .unidade_out(unidade_regOUT2));
281
282
283 Conversor_BCD Result_IOModule (
284                                                                  .bin(resultado),
285                                                                  .centena(centena),
286                                                                  .dezena(dezena),
287                                                                  .unidade(unidade),
```

```
288                                                                 .neg(negative)
289                                                                 );
290
291 InOut_Module InOut_Result (
292                                                 .centena(centena),
293                                                 .dezena(dezena),
294                                                 .unidade(unidade),
295                                                 .out(outputControl),
296                                                 .centena_out(centena_out),
297                                                 .dezena_out(dezena_out),
298                                                 .unidade_out(unidade_out));
299
300
301 endmodule
```

Listing A.1 – Algoritmo de integração dos Módulos

APÊNDICE B – Unidade de Controle

```
1  module ControlUnit(clock, opcode, MuxBankSelect, MUXULASelect, BIGMUXSelect, hlt,
2                     jmp, beq, bneq, MUXDMSelect, WriteFlag, RegWrite, ULAcode,
                          inputControl, outputControl, jr, beqz);
3
4          input clock;
5          input [5:0] opcode;
6          output reg MuxBankSelect;
7          output reg MUXULASelect;
8          output reg [1:0] BIGMUXSelect;
9          output reg hlt, jmp, beq, bneq, jr, beqz;
10         output reg MUXDMSelect;
11         output reg WriteFlag;
12         output reg RegWrite;
13         output reg[4:0] ULAcode;
14         output reg inputControl;
15         output reg outputControl;
16
17         always @ (*) begin
18         case(opcode[5:0])
19
20         6'b000000:
21                 begin
22
23                                 MuxBankSelect = 1'b1;
24                                 MUXULASelect = 1'b0;
25                                 BIGMUXSelect = 2'b00;
26                                 hlt = 1'b0;
27                                 jmp = 1'b0;
28                                 beq = 1'b0;
29                                 bneq = 1'b0;
30                                 outputControl = 1'b0;
31                                 MUXDMSelect = 1'b1;
32                                 WriteFlag = 1'b0;
33                                 RegWrite = 1'b1;
34                                 ULAcode = 5'b00000;
35                                 inputControl = 1'b0;
36                                 jr = 1'b0;
37                                 beqz = 1'b0;
38                         end
39
40                         6'b000001: begin
41                                 MuxBankSelect = 1'b1;
42                                 MUXULASelect = 1'b0 ;
43                                 BIGMUXSelect = 2'b00;
44                                 hlt = 1'b0;
45                                 jmp = 1'b0;
46                                 beq = 1'b0;
47                                 bneq = 1'b0;
48                                 outputControl = 1'b0;
49                                 MUXDMSelect = 1'b1;
50                                 WriteFlag = 1'b0;
51                                 RegWrite = 1'b1;
52                                 ULAcode = 5'b00001;
53                                 inputControl = 1'b0;
```

```
54                              jr = 1'b0;
55                              beqz = 1'b0;
56
57                      end
58
59                      6'b000010: begin
60                              MuxBankSelect = 1'b0;
61                              MUXULASelect = 1'b1;
62                              BIGMUXSelect = 2'b00;
63                              hlt = 1'b0;
64                              jmp = 1'b0;
65                              beq = 1'b0;
66                              bneq = 1'b0;
67                              outputControl = 1'b0;
68                              MUXDMSelect = 1'b1;
69                              WriteFlag = 1'b0;
70                              RegWrite = 1'b1;
71                              ULAcode = 5'b00000;
72                              inputControl = 1'b0;
73                              jr = 1'b0;
74                              beqz = 1'b0;
75                              end
76
77                      6'b000011: begin
78                              MuxBankSelect = 1'b0;
79                              MUXULASelect = 1'b1;
80                              BIGMUXSelect = 2'b00;
81                              hlt = 1'b0;
82                              jmp = 1'b0;
83                              beq = 1'b0;
84                              bneq = 1'b0;
85                              outputControl = 1'b0;
86                              MUXDMSelect = 1'b1;
87                              WriteFlag = 1'b0;
88                              RegWrite = 1'b1;
89                              ULAcode = 5'b00001;
90                              inputControl = 1'b0;
91                              jr = 1'b0;
92                              beqz = 1'b0;
93                              end
94
95                      6'b000100: begin
96                              MuxBankSelect = 1'b1;
97                              MUXULASelect = 1'b0;
98                              BIGMUXSelect = 2'b00;
99                              hlt = 1'b0;
100                             jmp = 1'b0;
101                             beq = 1'b0;
102                             bneq = 1'b0;
103                             outputControl = 1'b0;
104                             MUXDMSelect = 1'b1;
105                             WriteFlag = 1'b0;
106                             RegWrite = 1'b1;
107                             ULAcode = 5'b00101;
108                             inputControl = 1'b0;
109                             jr = 1'b0;
110                             beqz = 1'b0;
111
112                             end
113
```

```
114                         6'b000101: begin
115                                 MuxBankSelect = 1'b1;
116                                 MUXULASelect = 1'b0;
117                                 BIGMUXSelect = 2'b00;
118                                 hlt = 1'b0;
119                                 jmp = 1'b0;
120                                 beq = 1'b0;
121                                 bneq = 1'b0;
122                                 outputControl = 1'b0;
123                                 MUXDMSelect = 1'b1;
124                                 WriteFlag = 1'b0;
125                                 RegWrite = 1'b1;
126                                 ULAcode = 5'b01001;
127                                 inputControl = 1'b0;
128                                 jr = 1'b0;
129                                 beqz = 1'b0;
130
131                                 end
132
133                         6'b000110: begin
134                                 MuxBankSelect = 1'b1;
135                                 MUXULASelect = 1'b0;
136                                 BIGMUXSelect = 2'b00;
137                                 hlt = 1'b0;
138                                 jmp = 1'b0;
139                                 beq = 1'b0;
140                                 bneq = 1'b0;
141                                 outputControl = 1'b0;
142                                 MUXDMSelect = 1'b1;
143                                 WriteFlag = 1'b0;
144                                 RegWrite = 1'b1;
145                                 ULAcode = 5'b01010;
146                                 inputControl = 1'b0;
147                                 jr = 1'b0;
148                                 beqz = 1'b0;
149
150                                 end
151
152                         6'b000111: begin
153                                 MuxBankSelect = 1'b1;
154                                 MUXULASelect = 1'b0;
155                                 BIGMUXSelect = 2'b00;
156                                 hlt = 1'b0;
157                                 jmp = 1'b0;
158                                 beq = 1'b0;
159                                 bneq = 1'b0;
160                                 outputControl = 1'b0;
161                                 MUXDMSelect = 1'b1;
162                                 WriteFlag = 1'b0;
163                                 RegWrite = 1'b1;
164                                 ULAcode = 5'b01011;
165                                 inputControl = 1'b0;
166                                 jr = 1'b0;
167                                 beqz = 1'b0;
168
169                                 end
170
171                         6'b001000: begin
172                                 MuxBankSelect = 1'b1;
173                                 MUXULASelect = 1'b0;
```

```
174                             BIGMUXSelect = 2'b00;
175                             hlt = 1'b0;
176                             jmp = 1'b0;
177                             beq = 1'b0;
178                             bneq = 1'b0;
179                             outputControl = 1'b0;
180                             MUXDMSelect = 1'b1;
181                             WriteFlag = 1'b0;
182                             RegWrite = 1'b1;
183                             ULAcode = 5'b01100;
184                             inputControl = 1'b0;
185                             jr = 1'b0;
186                             beqz = 1'b0;
187
188                             end
189
190                     6'b001001: begin
191                             MuxBankSelect = 1'b1;
192                             MUXULASelect = 1'b0;
193                             BIGMUXSelect = 2'b00;
194                             hlt = 1'b0;
195                             jmp = 1'b0;
196                             beq = 1'b0;
197                             bneq = 1'b0;
198                             outputControl = 1'b0;
199                             MUXDMSelect = 1'b1;
200                             WriteFlag = 1'b0;
201                             RegWrite = 1'b1;
202                             ULAcode = 5'b00100;
203                             inputControl = 1'b0;
204                             jr = 1'b0;
205                             beqz = 1'b0;
206
207                             end
208
209                     6'b001010: begin
210                             MuxBankSelect = 1'b1;
211                             MUXULASelect = 1'b0;
212                             BIGMUXSelect = 2'b00;
213                             hlt = 1'b0;
214                             jmp = 1'b0;
215                             beq = 1'b0;
216                             bneq = 1'b0;
217                             outputControl = 1'b0;
218                             MUXDMSelect = 1'b1;
219                             WriteFlag = 1'b0;
220                             RegWrite = 1'b1;
221                             ULAcode = 5'b00111;
222                             inputControl = 1'b0;
223                             jr = 1'b0;
224                             beqz = 1'b0;
225
226                             end
227
228                     6'b001011: begin
229                             MuxBankSelect = 1'b1;
230                             MUXULASelect = 1'b0;
231                             BIGMUXSelect = 2'b00;
232                             hlt = 1'b0;
233                             jmp = 1'b0;
```

```
234                             beq = 1'b0;
235                             bneq = 1'b0;
236                             outputControl = 1'b0;
237                             MUXDMSelect = 1'b1;
238                             WriteFlag = 1'b0;
239                             RegWrite = 1'b1;
240                             ULAcode = 5'b01000;
241                             inputControl = 1'b0;
242                             jr = 1'b0;
243                             beqz = 1'b0;
244
245                             end
246
247                     6'b001100: begin
248                             MuxBankSelect = 1'b0;
249                             MUXULASelect = 1'b1;
250                             BIGMUXSelect = 2'b00;
251                             hlt = 1'b0;
252                             jmp = 1'b0;
253                             beq = 1'b0;
254                             bneq = 1'b0;
255                             outputControl = 1'b0;
256                             MUXDMSelect = 1'b0;
257                             WriteFlag = 1'b0;
258                             RegWrite = 1'b1;
259                             ULAcode = 5'b00000;
260                             inputControl = 1'b0;
261                             jr = 1'b0;
262                             beqz = 1'b0;
263
264                     end
265
266                     6'b001101: begin
267                             MuxBankSelect = 1'b0;
268                             MUXULASelect = 1'b1;
269                             BIGMUXSelect = 2'b00;
270                             hlt = 1'b0;
271                             jmp = 1'b0;
272                             beq = 1'b0;
273                             bneq = 1'b0;
274                             outputControl = 1'b0;
275                             MUXDMSelect = 1'b1;
276                             WriteFlag = 1'b0;
277                             RegWrite = 1'b1;
278                             ULAcode = 5'b00000;
279                             inputControl = 1'b0;
280                             jr = 1'b0;
281                             beqz = 1'b0;
282
283                     end
284
285
286                     6'b001110: begin
287                             MuxBankSelect = 1'b0;
288                             MUXULASelect = 1'b1;
289                             BIGMUXSelect = 2'b00;
290                             hlt = 1'b0;
291                             jmp = 1'b0;
292                             beq = 1'b0;
293                             bneq = 1'b0;
```

```
294                             outputControl = 1'b0;
295                             MUXDMSelect = 1'b0;
296                             WriteFlag = 1'b1;
297                             RegWrite = 1'b0;
298                             ULAcode = 5'b00000;
299                             inputControl = 1'b0;
300                             jr = 1'b0;
301                             beqz = 1'b0;
302
303                     end
304
305                     6'b001111: begin
306                             MuxBankSelect = 1'b0;
307                             MUXULASelect = 1'b0;
308                             BIGMUXSelect = 2'b01;
309                             hlt = 1'b0;
310                             jmp = 1'b0;
311                             beq = 1'b1;
312                             bneq = 1'b0;
313                             MUXDMSelect = 1'b0;
314                             WriteFlag = 1'b0;
315                             RegWrite = 1'b0;
316                             ULAcode = 5'b01101;
317                             inputControl = 1'b0;
318                             outputControl = 1'b0;
319                             jr = 1'b0;
320                             beqz = 1'b0;
321
322                             end
323
324                     6'b010000: begin
325                             MuxBankSelect = 1'b0;
326                             MUXULASelect = 1'b0;
327                             BIGMUXSelect = 2'b01;
328                             hlt = 1'b0;
329                             jmp = 1'b0;
330                             beq = 1'b0;
331                             bneq = 1'b1;
332                             outputControl = 1'b0;
333                             MUXDMSelect = 1'b0;
334                             WriteFlag = 1'b0;
335                             RegWrite = 1'b0;
336                             ULAcode = 5'b10000;
337                             inputControl = 1'b0;
338                             jr = 1'b0;
339                             beqz = 1'b0;
340
341                             end
342
343                     6'b010001: begin
344                             MuxBankSelect = 1'b0;
345                             MUXULASelect = 1'b0;
346                             BIGMUXSelect = 2'b01;
347                             hlt = 1'b0;
348                             jmp = 1'b0;
349                             beq = 1'b0;
350                             bneq = 1'b0;
351                             outputControl = 1'b0;
352                             MUXDMSelect = 1'b0;
353                             WriteFlag = 1'b0;
```

```
354                             RegWrite = 1'b0;
355                             ULAcode = 5'b10001;
356                             inputControl = 1'b0;
357                             jr = 1'b0;
358                             beqz = 1'b1;
359
360                     end
361
362                     6'b010010: begin
363                             MuxBankSelect = 1'b0;
364                             MUXULASelect = 1'b0;
365                             BIGMUXSelect = 2'b10;
366                             hlt = 1'b0;
367                             jmp = 1'b1;
368                             beq = 1'b0;
369                             bneq = 1'b0;
370                             outputControl = 1'b0;
371                             MUXDMSelect = 1'b0;
372                             WriteFlag = 1'b0;
373                             RegWrite = 1'b0;
374                             ULAcode = 5'b00010;
375                             inputControl = 1'b0;
376                             jr = 1'b0;
377                             beqz = 1'b0;
378
379                             end
380
381                     6'b010011: begin
382                                     MuxBankSelect = 1'b0;
383                                     MUXULASelect = 1'b0;
384                                     BIGMUXSelect = 2'b11;
385                                     hlt = 1'b0;
386                                     jmp = 1'b1;
387                                     beq = 1'b0;
388                                     bneq = 1'b0;
389                                     outputControl = 1'b0;
390                                     MUXDMSelect = 1'b0;
391                                     WriteFlag = 1'b0;
392                                     RegWrite = 1'b0;
393                                     ULAcode = 5'b00000;
394                                     inputControl = 1'b0;
395                                     jr = 1'b1;
396                                beqz = 1'b0;
397                             end
398
399                     6'b010100: begin
400                             MuxBankSelect = 1'b1;
401                             MUXULASelect = 1'b0;
402                             BIGMUXSelect = 2'b00;
403                             hlt = 1'b0;
404                             jmp = 1'b0;
405                             beq = 1'b0;
406                             bneq = 1'b0;
407                             outputControl = 1'b0;
408                             MUXDMSelect = 1'b1;
409                             WriteFlag = 1'b0;
410                             RegWrite = 1'b1;
411                             ULAcode = 5'b00000;
412                             inputControl = 1'b0;
413                             jr = 1'b0;
```

```
414                     beqz = 1'b0;
415                     end
416
417             6'b010101: begin
418
419                                     MuxBankSelect = 1'b0;
420                                     MUXULASelect = 1'b1;
421                                     BIGMUXSelect = 2'b00;
422                                     outputControl = 1'b0;
423                                     hlt = 1'b0;
424                                     jmp = 1'b0;
425                                     beq = 1'b0;
426                                     bneq = 1'b0;
427                                     MUXDMSelect = 1'b1;
428                                     WriteFlag = 1'b0;
429                                     RegWrite = 1'b1;
430                                     ULAcode = 5'b00000;
431                                     inputControl = 1'b1;
432                                     jr = 1'b0;
433                             beqz = 1'b0;
434
435                     end
436             6'b010110: begin
437
438                                     MuxBankSelect = 1'b1;
439                                     MUXULASelect = 1'b0;
440                                     BIGMUXSelect = 2'b00;
441                                     hlt = 1'b0;
442                                     jmp = 1'b0;
443                                     beq = 1'b0;
444                                     bneq = 1'b0;
445                                     outputControl = 1'b1;
446                                     MUXDMSelect = 1'b1;
447                                     WriteFlag = 1'b0;
448                                     RegWrite = 1'b0;
449                                     ULAcode = 5'b00000;
450                                     inputControl = 1'b0;
451                                     jr = 1'b0;
452                             beqz = 1'b0;
453                     end
454
455             6'b010111: begin
456                     MuxBankSelect = 1'b0;
457                     MUXULASelect = 1'b0;
458                     BIGMUXSelect = 2'b00;
459                     hlt = 1'b1;
460                     jmp = 1'b0;
461                     beq = 1'b0;
462                     bneq = 1'b0;
463                     MUXDMSelect = 1'b0;
464                     WriteFlag = 1'b0;
465                     RegWrite = 1'b0;
466                     ULAcode = 5'b00000;
467                     inputControl = 1'b0;
468                     outputControl = 1'b0;
469                     jr = 1'b0;
470                     beqz = 1'b0;
471             end
472
473             default: begin
```

```
474                                     MuxBankSelect = 1'b0;
475                                     MUXULASelect = 1'b0;
476                                     BIGMUXSelect = 2'b00;
477                                     hlt = 1'b0;
478                                     jmp = 1'b0;
479                                     beq = 1'b0;
480                                     bneq = 1'b0;
481                                     MUXDMSelect = 1'b0;
482                                     WriteFlag = 1'b0;
483                                     RegWrite = 1'b0;
484                                     ULAcode = 5'b00000;
485                                     inputControl = 1'b0;
486                                     outputControl = 1'b0;
487                                     jr = 1'b0;
488                                     beqz = 1'b0;
489                             end
490                     endcase
491             end
492
493     endmodule
```

Listing B.1 – Unidade de Controle

APÊNDICE C – Módulo de Entrada e Saída

```
1  module InOut_Module (centena, dezena, unidade, out, centena_out, dezena_out, unidade_out)
       ;
2
3          input out;
4          input [3:0] centena, dezena, unidade;
5          output [6:0] centena_out, dezena_out, unidade_out;
6
7          Display cent(.in(centena), .out(centena_out), .halt(out));
8          Display dez(.in(dezena), .out(dezena_out), .halt(out));
9          Display uni(.in(unidade), .out(unidade_out), .halt(out));
10
11 endmodule
```

Listing C.1 – Módulo de Saída

```
1  module Conversor_BCD(bin, centena, dezena, unidade, neg);
2
3          integer i;
4          input [7:0] bin;
5          reg [7:0] binaux ;
6          output reg neg;
7          output reg [3:0] dezena, unidade, centena;
8
9          always@(bin) begin
10
11                 centena = 4'D0;
12                 dezena = 4'D0;
13                 unidade = 4'D0;
14
15                 if(bin[7]) begin
16                         binaux = ~bin + 8'd1;
17                         neg = 1;
18                 end
19                 else begin
20                         binaux = bin;
21                         neg = 0;
22                 end
23
24
25                 for(i = 7; i>=0; i=i-1) begin
26
27                         if(centena >= 5)
28                                 centena = centena + 4'd3;
29                         if (dezena >= 5)
30                                 dezena = dezena + 4'd3;
31                         if (unidade >= 5)
32                                 unidade = unidade + 4'd3;
33
34                         centena = centena << 1;
35                         centena[0] = dezena[3];
36                         dezena = dezena << 1;
37                         dezena[0] = unidade[3];
38                         unidade = unidade << 1;
39                         unidade[0] = binaux[i];
```

```
40                  end
41          end
42
43  endmodule
```

Listing C.2 – Conversor BCD

```
1   module Display(in, out, halt);
2
3           input halt;
4           input [3:0] in;
5           output reg [6:0] out;
6
7           always@(*) begin
8                   if(halt) begin
9                           case (in)
10                                  4'b0000: out = 7'b0000001;
11                                  4'b0001: out = 7'b1001111;
12                                  4'b0010: out = 7'b0010010;
13                                  4'b0011: out = 7'b0000110;
14                                  4'b0100: out = 7'b1001100;
15                                  4'b0101: out = 7'b0100100;
16                                  4'b0110: out = 7'b0100000;
17                                  4'b0111: out = 7'b0001111;
18                                  4'b1000: out = 7'b0000000;
19                                  4'b1001: out = 7'b0001100;
20                                  default: out = 7'b1111111;
21                          endcase
22                  end
23                  else
24                          out = 7'b1111110;
25          end
26
27  endmodule
```

Listing C.3 – Display de 7 segmentos