

Universidade Federal de São Paulo  
Instituto de Ciência e Tecnologia  
Laboratório de Arquitetura e Organização de Computadores

## **Desenvolvimento de um Sistema Computacional**

Aluno: Davi Melo Morales  
Professor: Dr. Tiago de Oliveira

Julho  
2016

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
1.1	Contextualização e Motivação . . . . .	3
1.2	Objetivos . . . . .	3
1.3	Ferramentas Utilizadas . . . . .	4
<b>2</b>	<b>Fundamentação Teórica</b>	<b>4</b>
2.1	CISC e RISC . . . . .	4
2.1.1	CISC . . . . .	5
2.1.2	RISC . . . . .	5
2.2	Tipos de Instruções . . . . .	6
2.2.1	Instruções de Transferência de Dados . . . . .	6
2.2.2	Instruções de Manipulação de Dados . . . . .	6
2.2.3	Instruções de Controle de Programa . . . . .	8
2.2.4	Instruções de Interrupção . . . . .	8
2.3	MIPS . . . . .	9
2.3.1	A Arquitetura de Conjunto de Instruções . . . . .	9
2.3.2	O Caminho de Dados . . . . .	11
2.3.3	<i>Program Counter</i> . . . . .	11
2.3.4	Memória . . . . .	11
2.3.5	Extensor de <i>bits</i> . . . . .	12
2.3.6	Banco de Registradores . . . . .	12
2.3.7	Unidade Lógica e Aritmética . . . . .	13
2.4	A Unidade de Controle . . . . .	14
2.5	A Máquina de Estados . . . . .	14
2.6	A Máquina de Estados Aplicada à Unidade de Controle . . . . .	15
2.7	Verilog HDL e FPGA . . . . .	16
<b>3</b>	<b>Desenvolvimento</b>	<b>17</b>
3.1	Conjunto de Instruções . . . . .	17
3.1.1	Formatos das Instruções . . . . .	18
3.1.2	Modos de Endereçamento . . . . .	20
3.2	Esboço da Arquitetura do Processador . . . . .	20
3.3	Implementação dos Componentes . . . . .	20
3.3.1	Banco de Registradores . . . . .	21
3.3.2	Unidade Lógica e Aritmética . . . . .	22
3.3.3	Extensor de <i>bits</i> . . . . .	23
3.3.4	Memória de Dados . . . . .	24
3.3.5	Memória de Instruções . . . . .	25
3.3.6	Program Counter . . . . .	26

3.3.7	Memória de Saída . . . . .	27
3.3.8	Multiplexadores e FlipFlops . . . . .	28
3.4	A Implementação da Unidade de Controle . . . . .	28
3.4.1	Instruções Lógicas e Aritméticas . . . . .	29
3.4.2	Instruções com Operações sobre Registradores . . . . .	34
3.4.3	Instruções com Acesso à Memória de Dados . . . . .	37
3.4.4	Instruções de Controle de Fluxo de Informações . . . . .	40
3.4.5	Demais Instruções . . . . .	42
3.5	Entrada e Saída . . . . .	45
3.5.1	Controlador de Saída . . . . .	46
3.5.2	Unidade de Saída . . . . .	47
3.5.3	Componentes Adicionais . . . . .	51
3.6	Implementação da CPU . . . . .	51
<b>4</b>	<b>Resultados</b>	<b>54</b>
4.1	Conjunto de Instruções . . . . .	55
4.1.1	Modos de Endereçamento . . . . .	55
4.2	Esboço da Arquitetura do Processador . . . . .	55
4.3	Testes em Forma de Onda . . . . .	56
4.3.1	Instruções A . . . . .	56
4.4	Instruções B . . . . .	57
4.5	Instruções C . . . . .	59
4.6	Testes no kit FPGA . . . . .	60
4.6.1	Sub-algoritmo A . . . . .	61
4.6.2	Sub-algoritmo B . . . . .	68
4.6.3	Sub-algoritmo C . . . . .	72
<b>5</b>	<b>Conclusão</b>	<b>75</b>
<b>6</b>	<b>Anexos</b>	<b>78</b>
6.1	Códigos complementares . . . . .	78
6.2	Esquemático Completo . . . . .	101

# 1 Introdução

## 1.1 Contextualização e Motivação

Define-se como computador, segundo [2], um dispositivo eletrônico utilizado para armazenar, organizar e encontrar palavras, números e imagens, para realizar cálculos e para controlar outras máquinas.

A especificação de um computador, de acordo com [1], consiste da descrição de sua apresentação para o programador em seu nível mais baixo, sua *arquitetura de conjunto de instruções*. A partir de seu conjunto de instruções (ISA, *Instruction Set Architecture*), formula-se a arquitetura do computador.

Tais instruções, segundo [15], devem ser tratadas pela *Unidade de Processamento*, onde operações de diversos tipos se realizam: lógicas e aritméticas, de entrada e saída e operações de memória. Assim, a *Unidade de Processamento* trabalha como um cérebro para o sistema computacional. Ela consiste de um conjunto de componentes; cada um deles é responsável por realizar as operações previamente citadas. Um deles, em especial se mostra essencial para a realização de todas elas: a *Unidade de Controle*.

Ao se considerar a forma como sistemas computacionais modernos trabalham, a presença de um componente que coordene o sistema como um todo se torna imperativa para uma realização eficiente de todas as instruções demandadas. A *Unidade de Controle* (UC) tem esse papel: ela direciona as operações do processador ao informar à memória, à unidade lógica e aritmética e aos componentes de entrada e saída sobre como eles devem se comportar mediante uma instrução [10].

Portanto, o projeto do sistema computacional apresentado nesse trabalho segue a seguinte ordem: *Arquitetura de Conjunto de Instruções*, *Unidade de Processamento* (componentes) e Unidade de Controle.

O restante da Seção 1 discute os objetivos desse projeto e as ferramentas utilizadas para a implementação da presente arquitetura. A Seção 2 apresenta a importância de todos os passos na elaboração de uma CPU, assim como os conceitos necessários para a compreensão desse seu funcionamento. Na Seção 3, é visto como o projeto foi implementado; na Seção 4, os resultados obtidos a partir do sistema são apresentados. Finalmente, na Seção 5, se discute sobre a realização desse projeto, as dificuldades encontradas e expectativas para trabalhos futuros.

## 1.2 Objetivos

Elaborar uma *Arquitetura de Conjunto de Instruções* e, a partir dela, projetar, implementar e simular um sistema computacional funcional.

## 1.3 Ferramentas Utilizadas

As ferramentas a serem utilizadas no desenvolvimento do projeto consistem na linguagem Verilog HDL, no software *Altera Quartus II®* e no kit FPGA *ALTERA DE2-115 Cyclone IV®*.

Esse kit é de propósito educacional e possui LEDs, alavancas, botões, um *display* de 7 segmentos e um *display* LCD, além de outros componentes de entrada e saída. A Figura 1 apresenta em detalhes sua configuração:

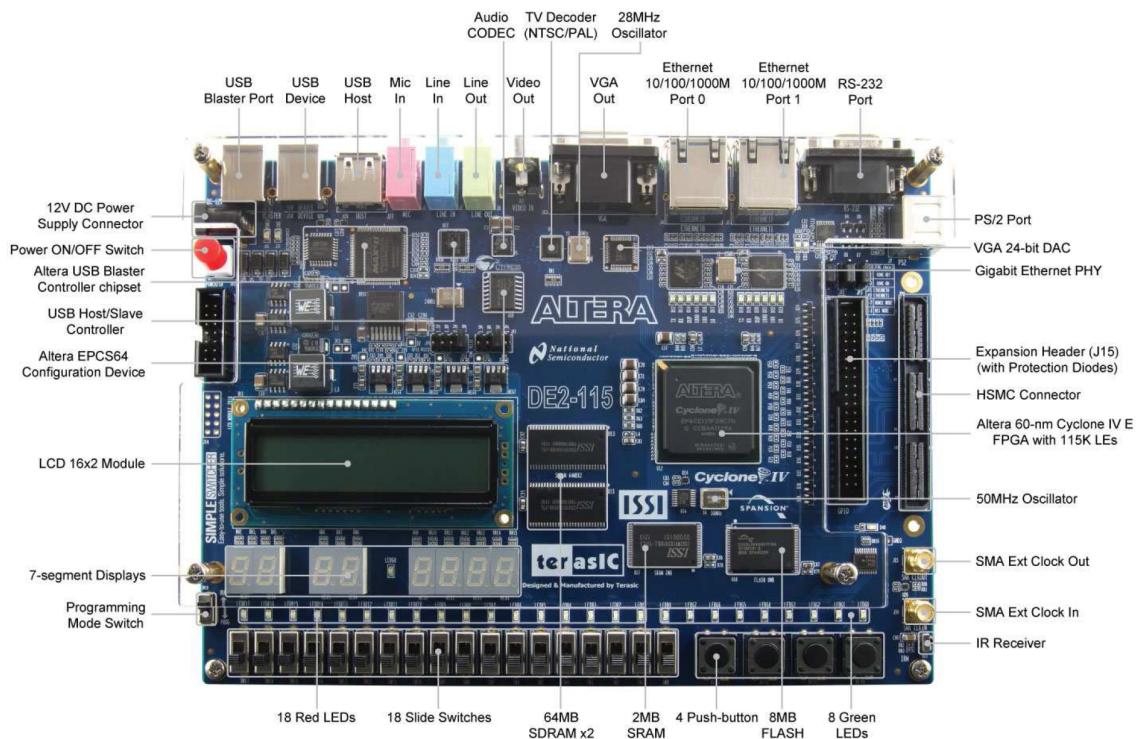


Figura 1: Kit FPGA *ALTERA DE2-115 Cyclone IV®* [14].

## 2 Fundamentação Teórica

### 2.1 CISC e RISC

Computadores apresentam conjuntos de instruções que permitem a realização das tarefas computacionais. Os conjuntos de instruções podem apresentar diversos tipos de variações, como no nome simbólico dado às instruções.

Existem dois tipos principais distintos de arquitetura de conjunto de instruções que diferem razoavelmente na relação entre hardware e software, a CISC (Complex instruction set computers) e a RISC (Reduced instruction set computers) [1].

### 2.1.1 CISC

O objetivo principal da arquitetura CISC é completar uma tarefa no menor número de linhas de assembly possível. Isso é alcançado através da construção de um hardware de processamento capaz de compreender e executar uma série de operações.

Essa arquitetura trabalha com instruções complexas, as quais operam diretamente nos bancos de memória e não requerem que o programador informe diretamente funções de load ou store.

Uma das maiores vantagens obtidas é a diminuição no trabalho do compilador ao traduzir uma sentença de alto nível para assembly. Por conta do tamanho de código reduzido, menos memória é necessária para guardar instruções [3].

Em suma, uma arquitetura puramente CISC possuirá as seguintes propriedades:

1. O acesso à memória é disponível à maior parte dos tipos de instrução;
2. Modos de endereçamento são substanciais em número;
3. Formatos de instrução apresentam tamanhos variados;
4. Instruções realizam tanto operações simples como complexas [1].

### 2.1.2 RISC

Apresentando um conjunto de instruções reduzido, a arquitetura RISC enfatiza instruções simples e flexibilidade que, combinados, garantem uma execução mais rápida [1].

Ela exige do programador mais linhas de código, o que requer mais memória e mais trabalho do compilador. As instruções reduzidas, porém, requerem menos espaço de transistores em hardware, deixando mais espaço para registradores de propósito geral [3].

Uma arquitetura RISC segue, portanto, as seguintes propriedades:

1. Acessos à memória se restringem a funções de load e store, ao passo que instruções de manipulação de dados ocorrem de registrador para registrador;

2. Modos de endereçamento são limitados em número;
3. Formatos de instrução possuem o mesmo tamanho;
4. Instruções realizam operações simples.

## 2.2 Tipos de Instruções

### 2.2.1 Instruções de Transferência de Dados

Instruções de transferência de dados movimentam os dados de um lugar a outro dentro do computador sem realizar alterações. Podem ocorrer entre a memória principal e os registradores do processador, entre registradores e dispositivos de entrada e saída, e entre os próprios registradores do processador [1].

A Tabela 1 apresenta exemplos relevantes de instruções dessa natureza:

Tabela 1: Instruções de transferência de dados

Nome	Descrição
Load	Dados são lidos da memória e carregados em registradores para a realização de operações
Store	Dados são passados de registradores para a memória
Load imediato	Semelhante à instrução <i>Load</i> ; a referência à posição na memória, porém, é armazenada em um imediato
Entrada	Permite ao usuário inserir informações no sistema através dos periféricos de entrada
Saída	Permite <i>feedback</i> do sistema às ações do usuário através dos periféricos de saída

### 2.2.2 Instruções de Manipulação de Dados

Instruções de manipulação de dados são aquelas que realizam as operações nos dados. Dividem-se, normalmente, em três tipos distintos:

1. Instruções aritméticas: referem-se às operações aritméticas básicas realizadas por um computador; uma vez que o número de *bits* nos registradores é finito, os resultados das operações apresentam precisão limitada. Exemplos:

Tabela 2: Exemplos de instruções aritméticas.

Nome	Descrição
Adição	Adiciona operandos em registradores
Subtração	Subtrai operandos em registradores
Adição Imediato	Adiciona operando em registrador a imediato
Subtração	Subtrai operando em registrador a imediato
Incrementa	Adiciona 1 ao valor em registrador ou palavra de memória
Decrementa	Subtrai 1 do valor em registrador ou palavra de memória
Negação	Muda o sinal aritmético do conteúdo de um registrador e aloca o resultado em um outro registrador

2. Instruções lógicas e de manipulação de *bits*: realizam operações binárias em palavras guardadas nos registradores, sendo úteis para se manipular *bits* ou grupos de *bits* que representem informação codificada em binário. Instruções lógicas consideram cada bit do operando separadamente e o tratam como uma variável binária, e aplicam-se para alterar valores de *bits*, limpar grupos de *bits* ou inserir novos valores em *bits* sobre operandos armazenados em registradores ou memória. Alguns exemplos relevantes se encontram na Tabela 4:

Tabela 3: Exemplos de instruções lógicas.

Nome	Descrição
And	Limpa <i>bits</i> de um operando seletivamente aplicando And ao operando com uma palavra que possui 0s nas posições a serem limpadas e 1s nas posições a permanecerem.
Or	Seta <i>bits</i> de um operando seletivamente ao aplicar Or ao operando com uma palavra com 1s nas posições que devem ser setadas como 1.
Xor	Completa seletivamente bits de um operando, de modo que uma variável binária é complementada quando Xor com 1 mas não muda quando Xor com 0

3. Instruções de deslocamento: permitem o deslocamento de *bits* do operando para a esquerda ou para a direita; podem se apresentar de formas variadas [1]. Exemplos:

Tabela 4: Instruções aritméticas.

Nome	Descrição
Desloca à Esquerda	Realiza deslocamento lógico à esquerda dos <i>bits</i> de um operando.
Desloca à Direita	Realiza deslocamento lógico à direita dos <i>bits</i> de um operando.

### 2.2.3 Instruções de Controle de Programa

Ao serem processadas pelo controle, as instruções de um programa são lidas a partir de localizações consecutivas na memória e executadas uma a uma, de maneira que o *Program Counter* é incrementado para acompanhar a ordem das instruções. Uma instrução de controle de programa, porém, traz alterações ao valor de endereço do PC, alterando assim o fluxo de controle.

A mudança no PC a partir da execução de uma instrução de controle de programa causa uma interrupção na sequência de execução das instruções, o que permite controle sobre o fluxo da execução de um programa e a capacidade de ramificação para diferentes segmentos do programa, dependendo de resultados anteriores [1].

A Tabela 5 contém exemplos relevantes de instruções de controle de programa:

Tabela 5: Instruções de controle de programa

Nome	Descrição
Branch em zero	Realiza ramificação para a localização especificada caso o conteúdo do registrador fonte seja igual a zero [4].
Branch em negativo	Realiza ramificação para a localização especificada caso o conteúdo do registrador fonte seja negativo.
Jump	Salta incondicionalmente para a instrução especificada [5].
Set on less than	Se o valor contido em um operando é menor que o do outro, a saída é definida como 1; caso contrário, 0 [6].

### 2.2.4 Instruções de Interrupção

São instruções utilizadas quando se precisa sair da execução corriqueira do programa, tais como instruções que param operações ou que informam a ausência delas.

## 2.3 MIPS

MIPS é uma arquitetura RISC simples, altamente escalável e bastante utilizada para fins didáticos [9]. Ela evoluiu da pesquisa em organização eficiente de processadores e integração de VLSI na Universidade de *Stanford*.

Uma vez que o processador desenvolvido nesse projeto baseia-se na arquitetura MIPS, torna-se importante mencionar os aspectos fundamentais do funcionamento dessa arquitetura, muitos dos quais estendem-se a processadores de maneira geral.

A seguir se explicam aspectos referentes à sua arquitetura de conjunto de instruções, ao funcionamento do caminho de dados e aos principais componentes que constituem a arquitetura, sendo eles: o *program counter*, a memória de instruções, o banco de registradores, o extensor de *bits*, a unidade lógica e aritmética (ULA) e a memória de dados.

### 2.3.1 A Arquitetura de Conjunto de Instruções

- Execução de cinco estágios: busca, decodificação, execução, acesso à memória e escrita de resultado;
- Conjunto de instruções regular (32-bit);
- Instruções lógicas e aritméticas de três operandos;
- Ausência de instruções complexas;
- Apenas instruções de *Load* e *Store* acessam a memória.

Um dos principais recursos da arquitetura MIPS é o conjunto de registradores regulares, o qual consiste em um registrador reservado como *program counter* (PC) e em um banco de 32 registradores de propósito geral - r0-r31 de 32-bits. Todos esses registradores podem ser utilizados como registradores alvo e fontes de dados para todas as instruções que os requisitarem. Por ser internamente definido como igual a 0, r0 é uma exceção a essa propriedade [7].

A MIPS apresenta também uma Unidade Lógica e Aritmética (ULA), a qual consiste em um circuito combinacional que realiza um conjunto de instruções aritméticas básicas e microoperações lógicas [1].

Sua memória principal divide-se em memória de instruções - onde gravam-se as instruções a serem executadas pelo processador - e em memória de dados - onde gravam-se os dados a serem utilizados nas operações do processador.

Seu conjunto de instruções se divide em três grupos principais de instruções, sendo que cada uma apresenta codificação própria:

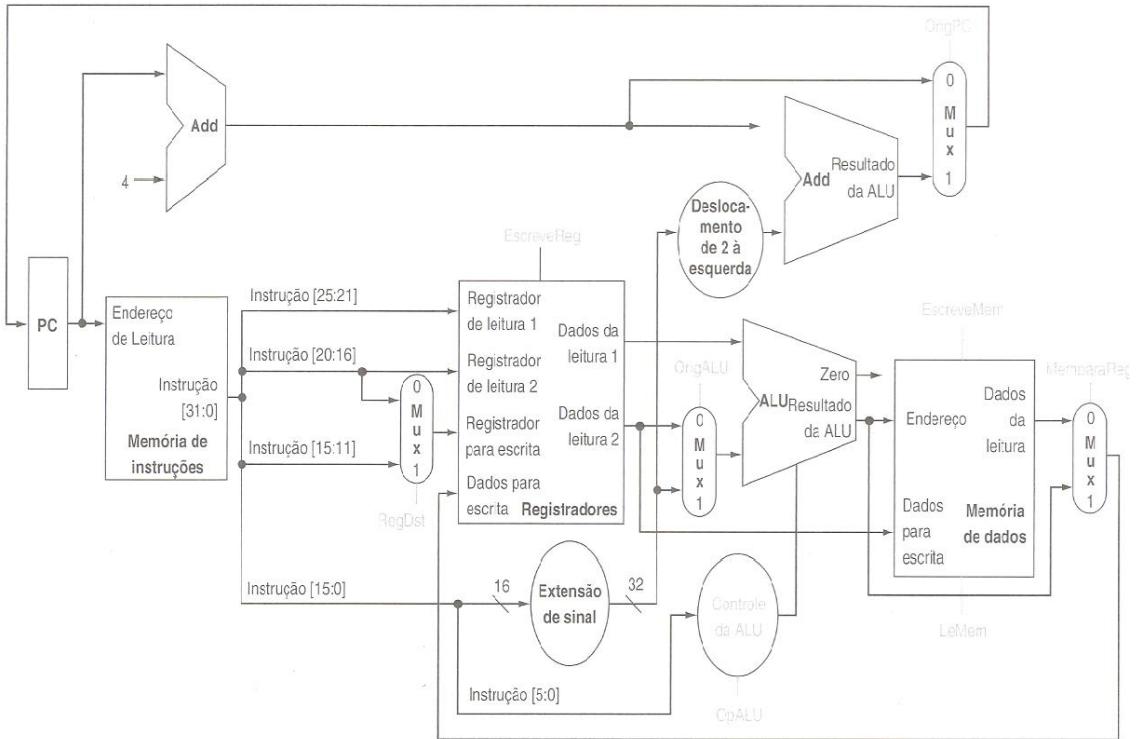


Figura 2: Esboço da arquitetura MIPS e seus componentes [10].

- Tipo I (Imediato): apresenta em seu campo inicial o *opcode*, parcela dos bits referente ao comando da instrução; *rs* é o registrador fonte, *rt* o registrador de destino e um último campo de *offset* que, a depender da instrução, pode ser interpretado como um *unsigned integer* de alcance 0 até 65535 ou um *sign-extended integer* de alcance -32768 até 32767.

31 - 26	25 - 21	20 - 16	15 - 0
opcode	rs	rt	offset

- Tipo J (*Jump*): além do *opcode*, esse tipo de instrução reserva um *offset* de 26 bits que pode ser usado como um *offset* de extensão de sinal para ramificações (*branches*) relativas ao PC; em outros casos, os 5 bits menos significativos são utilizados para selecionar um dos registradores de propósito geral [7].

31 - 26	25 - 0
opcode	endereço da instrução

- Tipo R (Registrador): inclui todas as operações lógicas e aritméticas mais comuns, assim como as instruções de *load* e *store*. Nesse caso, os registradores fonte são *rs* e *rt*; *rd* é o registrador de destino, *sa* é a quantia de deslocamento e *função* é utilizada para *opcode*=0 para distinguir operações na ULA [11].

31 - 26	25 - 21	20 - 16	15 - 11	10 - 6	5 - 0
opcode	rs	rt	rd	sa	função

### 2.3.2 O Caminho de Dados

O caminho de dados contém a lógica digital que implementa as diversas microoperações realizadas por um sistema computacional. Tal lógica consiste de barramentos, multiplexadores, decodificadores e unidades de processamento [1].

Considerando-se uma abordagem monocíclica, as instruções são realizadas em série e uma a cada ciclo de *clock*. Sua realização depende da interação dos diversos componentes que formam o processador. A Figura 2 apresenta esses componentes e como eles se conectam em uma arquitetura MIPS.

### 2.3.3 *Program Counter*

O primeiro componente a ser mencionado é o *Program Counter*, ou *contador de programa*. Ele consiste em um registrador que contém o endereço da instrução atual do programa em execução. Associa-se, normalmente, a um somador que o incrementa para o endereço da instrução seguinte, quatro *bytes* depois [10].

### 2.3.4 Memória

Em se tratando de sistemas digitais, a memória se define como uma coleção de compartimentos onde são guardadas informações em binário. Ela pode ser utilizada em partes diversas do computador, provendo tanto armazenamento permanente quanto temporário, e pode interagir tanto com dispositivos de entrada e saída quanto com elementos que processam informação.

Dois tipos fundamentais de memórias se utilizam em um sistema computacional, a memória de acesso aleatório (RAM) e a memória apenas de leitura

(ROM). Uma diferença fundamental entre as duas é o fato de que a memória RAM permite a leitura - processo de transferência de informação para fora da memória - e escrita - armazenamento de informações na memória -, ao passo que a memória ROM permite apenas a leitura.

A memória de acesso aleatório tem como característica a possibilidade de acesso a qualquer um de seus compartimentos em um tempo constante, o que a diferencia fundamentalmente da memória serial e a torna eficiente em termos de tempo de acesso [1].

Na arquitetura MIPS, todas as instruções de máquina se codificam em palavras de 32 *bits*, assim como as operações com inteiros costumam ser realizadas com inteiros de 32 *bits*. Sua memória principal se organiza através de endereços de tamanho semelhante [7].

Sua memória principal se divide entre uma reservada para instruções e outra para dados:

- Memória de Instruções: parcela da memória principal reservada para o armazenamento das instruções a serem executadas pelo processador;
- Memória de Dados: componente da arquitetura utilizado a fim de se armazenar e retirar informações ao longo da execução de um programa.

### 2.3.5 Extensor de *bits*

Definem-se originalmente nessa arquitetura três tipos principais de dados: as já mencionadas palavras de 32 *bits*, as meias-palavras de 16 *bits* e os *bytes* de 8 *bits*. As transferências de dados entre a memória e o processador, porém, se realizam sempre através de palavras de 32 *bits*; um componente torna-se necessário para permitir e extrair o subconjunto correspondente de dados quando se executam instruções de armazenamento que façam uso de *bytes* e meias-palavras [7].

Nesse contexto, apresenta-se o *extensor de bits*. Ele se define como o componente responsável pela extensão de *bits* para que as operações possam ser realizadas com palavras de 32 *bits*.

### 2.3.6 Banco de Registradores

Um registrador é definido, segundo [1], como um conjunto de *flip-flops* que realizam tarefas de processamento de dados. Uma vez que cada *flip-flop* é capaz de armazenar um *bit* de informação, um registrador de  $n$  *bits*, composto por  $n$  *flip-flops*, armazena  $n$  *bits* de informação. A Figura 3 exemplifica um registrador simples formado por *flip-flops* tipo D.

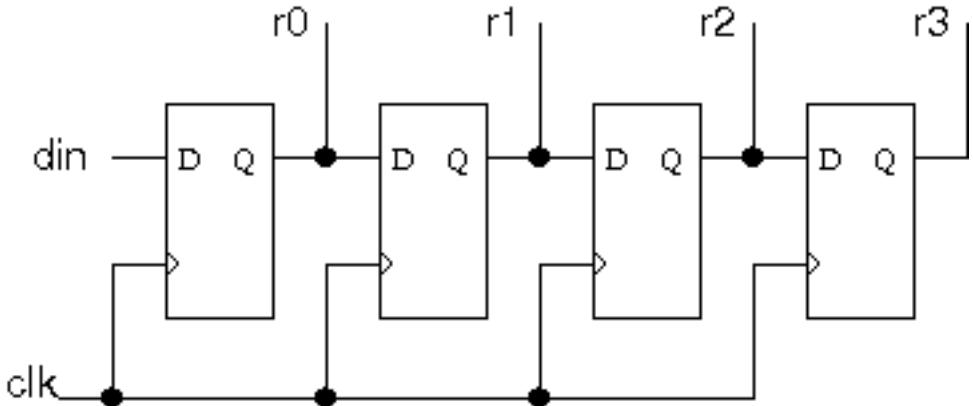


Figura 3: Exemplo de um registrador de 4-bits [16].

Uma característica marcante da arquitetura MIPS é seu conjunto de registradores, o qual se compõe pelo já mencionado *program counter*, e por um *banco de registradores* que contém 32 registradores de propósito geral - identificados de 0 a 31 -, todos com tamanho fixo de 32 bits. Com exceção de  $r_0$ , todos os registradores de propósito geral podem ser usados como registradores alvo e fontes de dados para quaisquer operações lógicas, aritméticas, de acesso à memória e de fluxo de controle [7].

### 2.3.7 Unidade Lógica e Aritmética

A Unidade Lógica e Aritmética consiste em um circuito combinacional que realiza uma série de microoperações lógicas e aritméticas. Ela apresenta linhas de seleção, as quais determinam qual operação há de se realizar. Uma ULA deve ser projetada levando-se em consideração suas seções fundamentais: a seção aritmética e a seção lógica.

Um somador paralelo apresenta-se como o componente básico em um circuito aritmético. Ele é composto por uma série de circuitos de somadores completos interligados em cascata. Desta forma, através do controle da entrada de dados para o somador paralelo, há como se obter diversos tipos de operações aritméticas, tais como a de incremento e a de subtração. Através, ainda, da manipulação dessas operações, há como se obter operações de complexidade maior, tais como a multiplicação e a divisão.

As operações lógicas mais comumente usadas são: AND, OR, XOR e NOT; através delas, todas as outras operações lógicas se derivam. A seção lógica se responsabiliza pelas microoperações lógicas, as quais manipulam os bits de operandos ao tratar cada bit em um registrador como uma variável binária, permitindo operações bit a bit.

Um outro tipo de operação comumente realizada nesse componente é o deslocamento. Ele transforma os dados através do deslocamento para a direita ou para a esquerda [1].

## 2.4 A Unidade de Controle

Mais especificamente, a unidade de controle administra os sinais de controle do processador. Ela direciona todo o fluxo de entrada e saída, recebe código para instruções a partir de microprogramas e manipula outras unidades e modelos ao provê-los sinais de controle e de temporização. Ela pode ser implementada de duas formas:

- Fixa: o projeto se baseia em uma arquitetura fixa; assim, qualquer modificação no conjunto de instruções exige alterações de circuito. Essa abordagem é preferencial para arquiteturas RISC, onde há um número reduzido de instruções básicas;
- Microprogramada: quando microprogramas são armazenados em uma memória de controle específica, o que os torna mais fáceis de se trocar [8].

Sua implementação consiste em uma máquina de estados que opera a partir de imputações que especificam as instruções e atributos referentes a elas e, através de sinais, determina cada estado do sistema para que se realizem as operações desejadas.

## 2.5 A Máquina de Estados

Máquinas de estados são modelos computacionais úteis tanto para *hardware* quanto para certos tipos de software. Algumas de suas principais aplicações consistem em:

- Formas simples de combinação de padrões;
- Modelos para circuitos lógicos [17].

Suas aplicações no dia a dia se observam em quaisquer dispositivos que realizem uma sequência de ações esperada a partir de uma sequência de eventos, tais como: aparelhos microondas - que permitem a seleção de suas ações pelo usuário -, parquímetros eletrônicos - que entregam bilhetes a partir dos valores das moedas inseridas - entre outros.

Ela tem a possibilidade de estar em um estado por vez considerando-se um número finito de estados. Ela consiste, necessariamente, de um estado

*inicial*. Esse estado é alterado a partir de algum evento - ou condição - que ocorra ao sistema, o qual se denomina *transição*.

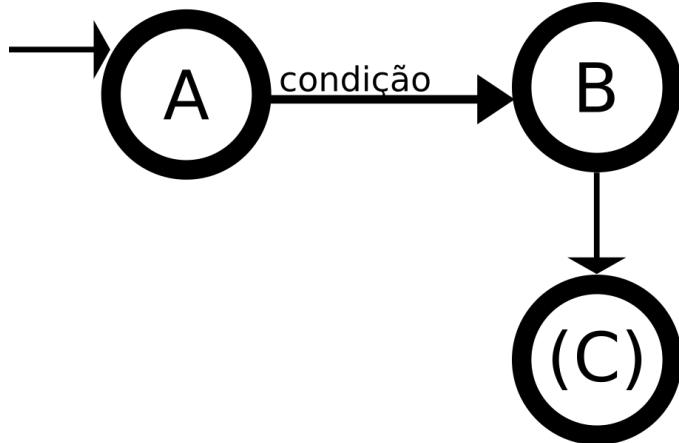


Figura 4: Representação gráfica de uma Máquina de Estados Finitos.

A Figura 4 apresenta a representação gráfica do modelo. Nela, cada círculo com uma letra representa um estado. A seta no estado A indica que ele é o *estado inicial*. As demais setas referem-se às transições, as quais ocorrem caso se satisfaçam as condições necessárias.

Transições podem ocorrer entre quaisquer estados - inclusive entre um estado e ele mesmo - e podem ser bidirecionais.

O estado C está representado como um estado final, no qual o sistema finaliza sua tarefa.

## 2.6 A Máquina de Estados Aplicada à Unidade de Controle

Os componentes que fazem parte de um processador são controlados a partir de determinados sinais, ou *flags*. Assim sendo, para cada operação a ser realizada pelo processador, torna-se necessária uma conformação específica desses sinais.

Portanto, ao se modelar a Unidade de Controle como uma Máquina de Estados, considera-se cada estado como uma conformação específica dos sinais a serem informados aos componentes a partir da Unidade de Controle.

As possíveis transições a serem realizadas por essa máquina são definidas pela conformação das instruções carregadas no sistema e o gatilho de transição entre elas é o *clock*, conforme ilustra a Figura 5.

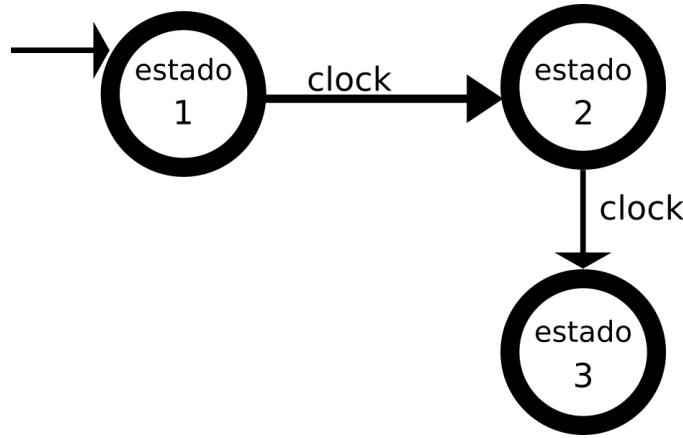


Figura 5: Representação gráfica de uma Máquina de Estados Finitos aplicada à Unidade de Controle.

## 2.7 Verilog HDL e FPGA

A implementação de um projeto computacional dessa natureza necessita de ferramentas versáteis e didáticas para sua realização. Sendo assim, apresentam-se o Verilog e o FPGA:

- Verilog é uma linguagem de descrição de *hardware* utilizada no projeto e documentação de sistemas eletrônicos, permitindo ao projetista criações em diversos níveis de abstração [12].
- FPGA, segundo [13], é um circuito integrado dotado de um grande número de unidades lógicas idênticas. Essas unidades lógicas consistem em componentes que se configuram de maneira independente e são interconectados a partir de uma matriz de trilhas condutoras e *switches* programáveis.

A configuração do FPGA se dá a partir de um arquivo binário, o qual especifica as funções das unidades lógicas e fecha, seletivamente, os *switches* da matriz de interconexão. Assim sendo, o *array* de unidades lógicas e a matriz de interconexão formam a estrutura básica do FPGA para a especificação de circuitos complexos [13].

Através de ferramentas de software, as especificações projetadas através do Verilog são traduzidas - compiladas - para binário e executadas no FPGA, de modo que é possível simular o projeto através dessas ferramentas em conjunto.

## 3 Desenvolvimento

Esse projeto consiste na implementação de um sistema computacional através da elaboração de um conjunto de instruções com esboço da representação de sua arquitetura, da implementação dos componentes da CPU e da implementação da Unidade de Controle.

A arquitetura projetada é RISC e se baseia em uma variação monocíclica da arquitetura MIPS, apresentando modificações inspiradas em conceitos presentes no livro *Logic and Computer Design Fundamentals* [1].

### 3.1 Conjunto de Instruções

Projetou-se um conjunto de instruções específico, o qual serve como fundamento para o desenvolvimento da presente arquitetura. Cada instrução se divide em seções que especificam os recursos a serem utilizados na operação. Tais recursos são explicados a seguir:

- DR: endereço de 5 bits que se refere ao registrador de destino da operação dentre os 32 presentes no banco de registradores;
- SA, SB: endereços de 5 bits que especificam os registradores de origem;
- IM: refere-se a um imediato de tamanho variável que é passado de maneira direta;
- Alavancas: refere-se às chaves de entrada presentes no kit FPGA;
- *Displays*: se refere aos oito *displays* de sete segmentos presentes no kit FPGA;
- PC: refere-se ao *Program Counter*;
- Shamt: quantidade de *bits* a serem deslocados;
- FZ e FN: são dois *flipflops* que guardam sinais checados para a realização de certas operações.

O *opcode* informa à unidade de controle a operação a ser realizada. Para isso, existe a necessidade do mapeamento das instruções em código binário. A Tabela 6 lista o conjunto de instruções executáveis pela arquitetura proposta e apresenta, para cada uma delas, seu mapeamento, mnemônico e a operação que realiza:

Tabela 6: Mapeamento e especificações referentes ao conjunto de instruções

Instrução	Opcode	Mnemônico	Operação
Adição	000000	ADD	$R[DR] \leftarrow R[SA] + R[SB]$
Adição Imediato	000001	ADDI	$R[DR] \leftarrow R[SA] + IM$
Subtração	000010	SUB	$R[DR] \leftarrow R[SA] - R[SB]$
Subtração Imediato	000011	SUBI	$R[DR] \leftarrow R[SA] - IM + 1$
Multiplicação	000100	MUL	$R[DR] \leftarrow R[SA] * R[SB]$
Divisão	000101	DIV	$R[DR] \leftarrow R[SA]/R[SB]$
Incrementa	000110	INC	$R[DR] \leftarrow R[SA] + 1$
Decrementa	000111	DEC	$R[DR] \leftarrow R[SA] - 1$
And	001000	AND	$R[DR] \leftarrow R[SA] \wedge R[SB]$
Or	001001	OR	$R[DR] \leftarrow R[SA] \vee R[SB]$
Resto	001010	MOD	$R[DR] \leftarrow R[SA]\%R[SB]$
Xor	001100	XOR	$R[DR] \leftarrow R[SA] \oplus R[SB]$
Not	001101	NOT	$R[DR] \leftarrow \overline{R[SA]}$
Desloca Esquerda	010000	SHL	$R[DR] \leftarrow sl(shamt)R[SA]$
Desloca Direita	010001	SHR	$R[DR] \leftarrow sr(shamt)R[SA]$
Pré-branch	011111	PBC	se $R[SA] = 0, FZ = 1;$ se $R[SA] < 0, FN = 1;$
Branch em Zero	010011	BOZ	se $FZ = 1$ , então $PC \leftarrow PC + 1 + IM$ se $FZ = 0$ , então $PC \leftarrow PC + 1$
Branch em Negativo	010100	BON	se $FN = 1$ , então $PC \leftarrow PC + IM$ se $FN = 0$ , então $PC \leftarrow PC + 1$
Jump	010101	JMP	$PC \leftarrow IM$
Set on Less Than	010111	SLT	se $R[SA] < R[SB]$ , então $R[DR] = 1$
Load	011000	LD	$R[DR] \leftarrow M[IM]$
Store	011001	ST	$M[IM] \leftarrow R[SA]$
Load Imediato	011010	LDI	$R[DR] \leftarrow IM$
Nop	011011	NOP	Sem Operação
HLT	011100	HLT	Parar Operação
Entrada	011101	IN	$R[DR] \leftarrow alavancas$
Saída	011110	OUT	$Displays \leftarrow R[SA]$

### 3.1.1 Formatos das Instruções

Projetaram-se quatro formatos distintos para as instruções com o objetivo de transmitir as informações necessárias de maneira simples e eficiente. Por se tratar de uma arquitetura RISC, todas as instruções apresentam o mesmo tamanho, 32 bits:

- Formato de três registradores: engloba a maioria das instruções lógicas e aritméticas - tais como ADD, SUB, AND, OR, XOR, SLT -, sendo que dois registradores fonte - SA e SB - fornecem os operandos e o resultado é gravado no registrador destino.

31 - 26	25 - 21	20 - 16	15 - 11	10 - 0
opcode	SA	SB	DR	—

- Formato de dois registradores: apresenta as instruções aritméticas ADDI e SUBI, de modo que os operandos estão contidos no registrador fonte e no imediato. As funções INC e DEC também são descritas por esse formato; nelas o último campo da instrução torna-se desnecessário. As instruções NOT, SL e SR também aparecem nesse formato, de modo que o registrador fonte é negado, deslocado à esquerda ou à direita, respectivamente; o campo final da instrução, no caso das duas últimas, indica quantos *bits* o operando deve ser deslocado. Todas gravam informações no registrador destino (DR).

31 - 26	25 - 21	20 - 16	15 - 0
opcode	SA	DR	imediato/endereço/shamt

- Formato de um registrador fonte: contém as instruções PBC, LDI, IN e OUT. No caso de PBC, o valor contido no registrador fonte é comparado a uma referência e, a depender desse resultado, FZ ou FN são *setados*; essa instrução apresenta a exceção de que o endereço do registrador é informado das posições [20..16] da instrução. Em LDI, o valor contido no imediato é carregado para registrador destino. Para a instrução IN, os valores informados através dos dispositivos de entrada (alavancas) tomarão o espaço do campo final da instrução e esse valor será carregado para o registrador destino. Para a função de saída, o valor contido no registrador é encaminhado para a saída de dados.

31 - 26	25 - 21	20 - 0
opcode	SA/DR	imediato/entrada/saída

- Formato sem registradores: utilizado pelas instruções JMP, BOZ, BON, NOP e HLT. A instrução JMP confere o valor do imediato ao *PC*. BOZ e BON somam o valor informado no imediato ao PC seguinte. NOP e HLT não necessitam desse valor, de modo que informam apenas a não realização ou parada de operação.

31 - 26	26 - 0
opcode	imediato/-

### 3.1.2 Modos de Endereçamento

Como pode-se observar nas estruturas das instruções, os seguintes modos de endereçamento foram adotados:

- Imediato: rápido, porém de alcance limitado, esse modo de endereçamento pode ser visto em instruções como SUBI e ADDI, nas quais o valor informado no imediato torna-se um operando.
- Direto: com equilíbrio entre eficiência e alcance, o modo direto está presente, também, nessa arquitetura. As instruções LD e ST apresentam esse modo, uma vez que buscam valores na memória utilizando endereços guardados em instruções.
- Por registrador: utilizado para acessar operandos guardados em registradores, esse modo é utilizado por muitas instruções, tais como: ADD, SUB, OR, XOR entre outras.

## 3.2 Esboço da Arquitetura do Processador

À semelhança da arquitetura MIPS, a presente arquitetura contém memória - neste caso subdividida entre memória de dados e memória de instruções -, um banco de registradores, uma unidade lógica e aritmética (ULA) e o PC.

A memória de instruções tem por função receber a instrução do PC e distribuí-la para seus diversos propósitos. O banco de registradores funciona como uma memória de curto prazo e é dotado de 32 registradores de 32 bits cada. A ULA tem por finalidade realizar as operações lógicas e aritméticas, enquanto a memória de dados armazena e carrega dados. Uma explicação mais detalhada a respeito das funções desses componentes se encontra na seção 2.3.

Para o caso de haverem operações entre um dado proveniente de um registrador e outro proveniente de um imediato - seja ele de 16 ou 21 bits -, há um extensor que emparelha a quantidade de bits do imediato em relação ao registrador.

Os dispositivos de entrada e saída também foram levados em consideração na elaboração dessa arquitetura, de maneira que dados tanto deverão poder ser inseridos no sistema através dos periféricos de entrada, quanto haverá retorno para o usuário através do dispositivo de saída.

## 3.3 Implementação dos Componentes

Os componentes principais do sistema foram implementados separadamente, cada um em um projeto próprio. Após a implementação e testes dos

componentes, eles foram integrados em um projeto principal. Nesta subseção descreve-se a implementação de cada componente, assim como os sinais necessários para o seu controle; eles são citados na ordem em que foram implementados.

### 3.3.1 Banco de Registradores

À semelhança da arquitetura MIPS, banco de registradores se constitui de 32 registradores de propósito geral, todos com capacidade de 32 *bits* de armazenamento.

Esse componente tem por entrada 3 endereços de 5 *bits*, que referenciam os registradores a serem utilizados; em se tratando das instruções do sistema, o primeiro endereço diz respeito ao registrador a receber dados de escrita, ao passo que os dois endereços restantes são referentes a registradores a serem lidos.

Ele pode receber, também, um valor de 32 *bits*, o qual é a informação a ser escrita em um registrador.

O banco de registradores possui apenas um sinal de controle, nesse caso a *flag writeRegister*. Como se observa no Código 1, essa *flag* tem por objetivo determinar se os dados de escrita - *writeData* - devem ser escritos no registrador determinado pelo endereço de escrita - *write address*.

Os dados presentes nos registradores referenciados são continuamente atribuídos, de maneira que há uma saída para cada um deles no componente.

Código 1: Banco de Registradores

```
module registerFile( writeAddress , readAddress1 ,
                     readAddress2 , cclock , writeRegister , writeData ,
                     dataA , dataB , dataC );
    input [4:0] readAddress1 , readAddress2 ,
               writeAddress ;
    input clock , writeRegister ;
    input [31:0] writeData ;
    output [31:0] dataA , dataB , dataC ;

    reg [31:0] RF [31:0];

    always @ (posedge clock) begin
        if( writeRegister == 1)
            RF[ writeAddress ] = writeData ;
    end
```

```

assign dataA = RF[ readAddress1 ];
assign dataB = RF[ readAddress2 ];
assign dataC = RF[ writeAddress ];

```

**endmodule**

### 3.3.2 Unidade Lógica e Aritmética

A unidade lógica e aritmética foi implementada de modo que uma variável chamada *aluSelection*, dentro de um *case*, controla qual operação há de ser realizada pelo componente. No caso, dois operandos de 32 *bits* entram na ULA, são transformados de acordo com as operações e, então, são exportados em *aluOut*; *shamt* indica o deslocamento para as operações de deslocamento de *bits*.

Duas *flags* podem ser acionadas nesse componente a depender do resultado das operações: a *zero* - acionada quando o resultado da operação é igual a zero-, e a *negative*, acionada caso o resultado da operação seja negativo.

Código 2: Unidade Lógica e Aritmética

```

module ALU( aluSelection , dataA , dataB2 , aluOut ,
            negative , zero , shamt );
input [3:0] aluSelection ;
input [4:0] shamt ;
input [31:0] dataA , dataB2 ;
output reg [31:0] aluOut ;
output negative ;
output zero ;

always @ ( aluSelection or dataA or dataB2 ) begin
    case( aluSelection [3:0])
        4'b0000: aluOut = dataA ;
        4'b0001: aluOut = dataA + dataB2 ;
        4'b0010: aluOut = dataA - dataB2 ;
        4'b0011: aluOut = dataA + 1;
        4'b0100: aluOut = dataA - 1;
        4'b0101: aluOut = dataA & dataB2 ;
        4'b0110: aluOut = dataA | dataB2 ;
        4'b0111: aluOut = dataA ^ dataB2 ;
        4'b1000: aluOut = ~dataA ;
        4'b1001: aluOut = dataA << shamt ;
        4'b1010: aluOut = dataA >> shamt ;

```

```

4'b1011: aluOut = dataA < dataB2 ? 1 : 0;
4'b1100: aluOut = dataA * dataB2;
4'b1101: aluOut = dataA / dataB2;
4'b1110: aluOut = dataA % dataB2;

endcase
end

assign zero = (aluOut==0);
assign negative = ($signed(aluOut)<0);

endmodule

```

### 3.3.3 Extensor de *bits*

Esse componente foi implementado da seguinte maneira: o valor a ser estendido pode vir a partir de três entradas diferentes, *inputA* e *inputB* e *inputC*; *extenderSelection* define, em um *case*, qual das duas há de ser estendida e atribuída à saída, *extenderOutput*.

A variável *inputA* possui 16 *bits*, enquanto a variável *inputB* possui 22; ambas são utilizadas em instruções diferentes que necessitam de operações com imediatos. A entrada *inputC* possui 18 *bits* e recebe dados diretamente do dispositivo de entrada.

Código 3: Extensor de Bits

```

module Extender( extenderSelection , inputA , inputB ,
                  inputC , extenderOutput );

input [15:0] inputA ;
input [20:0] inputB ;
input [17:0] inputC ;
input [1:0] extenderSelection ;
output reg [31:0] extenderOutput ;

always @ ( * ) begin
    case (extenderSelection)
        2'b00: begin
            extenderOutput = inputA ;
            if(inputA[15]==1'b1) begin
                extenderOutput = {16'h0000 , inputA } ;
                end

```

```

        end
2'b01: begin
    extenderOutput = inputB;
    if(inputB[20]==1'b1) begin
        extenderOutput = {11'h0000, inputB};
    end
end
2'b10: begin
    extenderOutput = inputC;
    if(inputC[17]==1'b1) begin
        extenderOutput = {14'h0000, inputC};
    end
end
default: begin
    extenderOutput = 32'b1;
end
endcase
end
endmodule

```

### 3.3.4 Memória de Dados

A memória de dados foi projetada para trabalhar com registradores de 32 *bits*. Ela consiste em um vetor com um desses registradores em cada uma de suas 1024 posições.

O dado a ser escrito é informado a partir de *dataC*. O endereço de escrita ou leitura é informado a partir de *address* e a saída do componente se dá através de *dataRAMOutput*, onde se informam os dados lidos da memória.

Os processos de leitura e escrita acontecem da seguinte forma: uma variável chamada *addressRegister*, de 9 *bits*, é reservada para armazenar o endereço de leitura registrado.

No caso do acionamento da flag *writeEnable*, escreve-se *dataC* no registrador cujo endereço foi informado; a *addressRegister* atribui-se tal endereço, o que permite que haja saída quando se atribui a *dataRAMOutput* o valor contido na posição de memória referente a *addressRegister*.

Código 4: Memória de Dados

```

module dataRAM(dataC, address, writeEnable,
               clock, dataRAMOutput);

input [31:0] dataC;

```

```

input [9:0] address;
input writeEnable , clock ;
output [31:0] dataRAMOutput;

reg [31:0] RAM[1023:0];
reg [9:0] addressRegister;

always @ (posedge clock)
begin
    if ( writeEnable) begin
        RAM[ address ] = dataC;
    end
    addressRegister = address;
end

assign dataRAMOutput = RAM[ addressRegister ];

endmodule

```

### 3.3.5 Memória de Instruções

Corresponde à parcela da memória que guarda as instruções a serem executadas.

Apresenta uma memória de 1024 posições, com registradores de 32 *bits*. Sincronizadamente ao *clock*, atribuem-se à saída - iRAMOutput - as instruções informadas dentro do próprio componente.

Código 5: Memória de Instruções

```

module simpleInstructionsRam( clock , address , iRAMOutput );
    input [9:0] address;
    input clock ;
    output [31:0] iRAMOutput;
    integer firstClock = 0;
    reg [31:0] instructionsRAM[1023:0];

    always @ ( posedge clock ) begin
        if (firstClock==0) begin
            instructionsRAM [0] = 32'b01101100000000000000000000000000;
            ...
            firstClock <= 0;
        end
    end

```

```

    end
end

assign iRAMOutput = instructionsRAM [ address ] ;
endmodule // simpleInstructionsRAM

```

### 3.3.6 Program Counter

Simplificadamente, o *program counter* é um registrador reservado para guardar o endereço da instrução corrente a ser executada.

Nesta implementação, o módulo referente ao *program counter*, PC, abrange todas as funções principais que se relacionam a esse componente.

Ele trabalha de forma síncrona ao *clock*. Por padrão, o *program counter* guarda seu valor e é incrementado a cada iteração.

As instruções de *jump* e *branch* são acionadas através das *flags zero*, *bzero*, *negative*, *bnegative* e *jump* e realizam alterações no valor do PC. Caso a *flag jump* estiver acionada, o valor seguinte do PC corresponderá ao que se informa através da instrução. No caso do acionamento de *zero* e *bzero* ou *negative* e *bnegative*, o valor informado na instrução deve ser somado ao valor incrementado de PC.

*ResetCPU* tem o poder atribuir zero ao valor de *PC*, levando o sistema a um estado inicial; *HLT* leva o sistema a um estado de espera onde não se atribui valores novos a PC.

Código 6: *Program Counter*

```

module PC(clock , address , zero , negative , bzero ,
bnegative , jump , programCounter , HLT, resetCPU);

input clock , zero , negative , jump , bzero ,
bnegative , HLT, resetCPU;
input [9:0] address;
reg [9:0] programCounter , newPc , muxA;
wire [9:0] pcInc , branchAdd , jumpAdd;
wire select;
output [9:0] programCounter;

assign select = (bzero & zero) | (bnegative & negative);
assign pcInc = programCounter + 1;

assign branchAdd = pcInc + address;//(branch << 2);

```

```

always @ ( branchAdd or pcInc or select ) begin
    if(select == 1)
        muxA = branchAdd; //branchAdd;
    else
        muxA = pcInc ;
end

assign jumpAdd = ( address );

always @ ( jump or jumpAdd or muxA ) begin
    if(jump)
        newPc = jumpAdd;
    else
        newPc = muxA;
end

always @ ( posedge clock ) begin
    if(resetCPU) begin
        programCounter <= 0;
    end
    else if(HLT) begin
    end
    else begin
        programCounter <= newPc;
    end
end
endmodule

```

### 3.3.7 Memória de Saída

Esse componente foi criado a fim de permitir a saída de dados do sistema. Ele apresenta uma memória de 1024 posições de 32 *bits*. No caso de a *flag* IO\_RAMwrite estar acionada, o dado é escrito, sincronizadamente, nessa memória.

É atribuído à saída - IO\_RAMOutput -, então, o valor presente no registrador informado.

Código 7: Memória de Saída

```
module dataOutput (dataC, address, IO_RAMwrite,
clock, IO_RAMOutput);
```

```

input IO_RAMwrite , clock ;
input [31:0] dataC;
input [9:0] address ;
output [31:0] IO_RAMOutput;

reg [31:0] ramOut [20:0];
always @ ( posedge clock ) begin
    if (IO_RAMwrite) begin
        ramOut [address] = dataC;
    end
end

assign IO_RAMOutput = ramOut [address] ;
endmodule // dataOutput

```

### 3.3.8 Multiplexadores e FlipFlops

Multiplexadores são componentes amplamente utilizados em circuitos lógicos e têm como função selecionar entre dois ou mais valores imputados. No presente exemplo, as *flags* são responsáveis por fazerem essa seleção dentro dos multiplexadores.

*Flipflops* são dispositivos de armazenamento binários. Eles apresentam dois estados - alto (1) e baixo (0) - e possuem a propriedade de permanecerem indefinidamente em um estado até que sejam orientados por um sinal de entrada a mudarem para o outro estado [18]. Os sinais de controle responsáveis pelo comportamento dos *flipflops* do sistema determinam a mudança de estado deles de acordo com as necessidades das instruções.

A Subseção 3.4 explica de maneira mais aprofundada os sinais de controle referentes a cada *flipflop* e multiplexador do sistema e sua importância para a execução adequada das instruções.

## 3.4 A Implementação da Unidade de Controle

Tendo em vista os sinais de controle mencionados na seção anterior para o projeto corrente, elaborou-se uma Unidade de Controle que permite a execução das instruções desenvolvidas para a arquitetura.

Ao se conhecerem os sinais de controle necessários para a coordenação dos componentes citados, determinaram-se os estados da unidade de controle referentes a cada instrução do conjunto; ou seja, qual deveria ser a saída da unidade de controle para cada instrução a ser realizada pelo sistema.

O reconhecimento da instrução desejada se dá através do *opcode*, informado nos 6 primeiros *bits* da instrução.

### 3.4.1 Instruções Lógicas e Aritméticas

A presente arquitetura de conjunto de instruções possui uma série de instruções que realizam operações lógicas e aritméticas e que devem ser tratadas de modo específico.

De maneira geral, essas instruções se comportam de forma semelhante quanto aos componentes que devem ser acionados para a realização das operações, divergindo apenas no código informado à ULA (aluSelection) para a seleção da operação a ser realizada e, em alguns casos, em sinais inerentes ao uso de imediato ou apenas registradores.

As instruções lógicas e aritméticas da presente arquitetura se dispõem conforme é descrito na Tabela 7 e exposto no Código 8.

Tabela 7: Instruções lógicas e aritméticas da arquitetura.

Instrução	opcode	aluSelection
Soma	000000	0001
Soma Imediato	000001	0001
Subtração	000010	0010
Subtração Imediato	000011	0010
Incrementa	000110	0011
Decrementa	000111	0100
And	001000	0101
Or	001001	0110
Xor	001100	0111
Not	001101	1000
Desloca Esquerda	010000	1001
Desloca Direita	010001	1010
Set on Less Than	010111	1011
Multiplicação	000100	1100
Divisão	000101	1101
Resto	001010	1110

Código 8: Instruções Lógicas e Aritméticas

```
6'b000000: begin //add
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0001;
```

```

extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
HLT = 0;
bzero = 1'b0;
bnegative = 1'b0;
end
6'b000001: begin//addi
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0001;
extenderSelection = 2'b00;
immediateSelection = 1'b1;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000010: begin//sub
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0010;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;

```

```

mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000011: begin//subi
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0010;
extenderSelection = 2'b00;
immediateSelection = 1'b1;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000100: begin//mul
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1100;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000101: begin//div

```

```

writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1101;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001010: begin//mod
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1110;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001000: begin//and
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0101;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;

```

```

writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001001: begin//or
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0110;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001100: begin//xor
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0111;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;

```

```

HLT = 0;
end

```

### 3.4.2 Instruções com Operações sobre Registradores

As instruções *incrementa*, *decrementa*, *not*, *desloca esquerda* e *desloca direita* realizam a operação sobre um registrador lido apenas; as demais instruções - exceto *soma imediato* e *subtração imediato* - realizam operações sobre dois registradores.

O estado referente à Unidade de Controle para instruções desse tipo se apresenta na Tabela 8; os estados referentes a cada uma dessas instruções se apresenta no Código 9.

Tabela 8: Configuração de saída da unidade de controle para instruções com operações sobre registradores.

Instrução	writeData Selection	write Register	alu Selection	extender Selection	immediate Selection	tripleMux Selection
opcode	1	1	alu Selection	xx	0	10
lastMux Sel	write Enable	IO_RAM write	enable	main Address	jump	bzero, bneg, HLT
0	0	0	1	0	0	0

O acompanhamento do caminho de dados nessas instruções torna possível que se justifique tal conformação para os sinais de controle.

Assim sendo, primeiramente os dados referentes aos endereços dos registradores são passados para o banco de registradores.

Para as instruções que operam sobre um registrador, da saída do banco de registradores aproveitam-se apenas os dados provenientes da saída *DataB*; portanto, o sinal *immediateSelection*, nesse caso é irrelevante.

Para as que operam sobre dois registradores, esse sinal seleciona no multiplexador *immediateSelectorMux*, como saída, o dado lido no segundo registrador de leitura para ser passado para a ULA.

O *aluSelection* determina qual das operações será realizada na unidade lógica e aritmética, enquanto o sinal *enable* tem o papel de determinar o acionamento de algum dos *flipflops* caso o resultado da operação seja negativo ou igual a zero.

O sinal *extenderSelection* é irrelevante, uma vez que nenhum valor precisa ser estendido. *LastMuxSel* em '0' garante que o valor proveniente da ULA seja selecionado no último multiplexador duplo; *tripleMuxSelection* em '10' seleciona o presente valor no multiplexador tripli para a escrita no registrador a ser escrito. *writeDataSelection* em '1' informa que o valor proveniente do

multiplexador triplo será escrito, enquanto *writeRegister* avisa ao banco de registradores que haverá escrita em um registrador.

Não há endereço a ser informado, saltos ou ramificações, tampouco escrita em memória de dados ou de saída em instruções lógicas e aritméticas. Desta forma, os sinais: *mainAddress*, *jump*, *branch*, *writeEnable* e *IO-RAMwrite* permanecem desligados em todas elas.

Código 9: Instruções com Operações sobre Registradores

```

6'b000110: begin//inc
    writeDataSelection = 1'b1;
    writeRegister = 1'b1;
    aluSelection = 4'b0011;
    extenderSelection = 2'bxx;
    immediateSelection = 1'b0;
    tripleMuxSelection = 2'b10;
    lastMuxSel = 1'b0;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b1;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end
6'b000111: begin//dec
    writeDataSelection = 1'b1;
    writeRegister = 1'b1;
    aluSelection = 4'b0100;
    extenderSelection = 2'bxx;
    immediateSelection = 1'b0;
    tripleMuxSelection = 2'b10;
    lastMuxSel = 1'b0;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b1;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;

```

```

end
6'b001101: begin//not
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1000;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b010000: begin//shift left
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1001;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b010001: begin//shift right
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1010;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;

```

```

tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

```

### 3.4.3 Instruções com Acesso à Memória de Dados

Abrange as instruções de carregamento, carregamento de imediato e de armazenamento - *load*, *load immediate* e *store*, respectivamente. Essas instruções se distinguem quanto à configuração da unidade de controle, mas foram agrupadas em virtude da semelhança entre os componentes acionados por elas.

A instrução *load* se comporta da seguinte forma: um valor informado na instrução é utilizado para definir qual posição da memória será lida a fim de que seus dados sejam carregados a um determinado registrador. Em *load immediate*, um valor já informado na própria instrução - imediato - é carregado a um registrador.

As Tabelas 9 e 10 ilustram o estado da unidade de controle para cada uma dessas instruções.

Tabela 9: Configuração de saída da unidade de controle para a instrução *load*.

Instrução	writeData Selection	write Register	alu Selection	extender Selection	immediate Selection	tripleMux Selection
011000	1	1	xxxx	xx	x	01
lastMux Sel	write Enable	IO_RAM write	enable	main Address	jump	bzero, bneg, HLT
1	0	0	0	0	0	0

Em *load*, o imediato de 16 bits informado na instrução é responsável por determinar o endereço de memória onde se buscam os dados desejados e, portanto, é selecionado por *extenderSelection*. Assim, *lastMuxSel* escolhe o valor proveniente do extensor de bits para que se informe o endereço à memória dedados através de *immediateSelection*.

*WriteEnable* é desligada, uma vez que dados são lidos da memória e não escritos. *TripleMuxSelection* é acionado em '01', o que o leva a selecionar

Tabela 10: Configuração de saída da unidade de controle para a instrução *load immediate*.

Instrução	writeData Selection	write Register	alu Selection	extender Selection	immediate Selection	tripleMux Selection
011010	0	1	xxxx	01	x	xx
lastMux Sel	write Enable	IO_RAM write	enable	main Address	jump	bzero, bneg, HLT
x	0	0	0	0	0	0

- em *tripleMux* - os dados provenientes da memória de dados para serem escritos no registrador.

*writeDataSelection* determina que os dados provenientes do multiplexador tripli sejam escritos na memória, enquanto *writeRegister* informa ao banco de registradores a necessidade de que se escreva em um registrador; os demais sinais não são relevantes para a execução dessa instrução e, portanto, permanecem desligados.

Para *load immediate*, apenas três sinais apresentam relevância direta; *writeRegister* e *extenderSelection* são acionados, enquanto *writeDataSelection* deve permanecer apagado.

Nesse caso, *extenderSelection* determina um valor mais extenso a ser carregado - imediato de 21 bits -, enquanto *writeDataSelection* escolhe o valor recém estendido para ser escrito no registrador; *writeRegister*, novamente, informa ao banco de registradores que haverá escrita.

A instrução *store* armazena um valor presente em um registrador na posição da memória de dados informada através de um imediato. Ela configura a unidade de controle conforme é descrito na Tabela 11

Tabela 11: Configuração de saída da unidade de controle para a instrução *store*.

Instrução	writeData Selection	write Register	alu Selection	extender Selection	immediate Selection	tripleMux Selection
011001	x	0	xxxx	01	x	xx
lastMux Sel	write Enable	IO_RAM write	enable	main Address	jump	bzero, bneg, HLT
1	1	0	0	0	0	0

Para tal instrução, a escrita de dados não é realizada em registradores pertencentes ao banco, portanto os sinais *writeRegister*, *writeDataSelection* e *tripleMuxSelection* não possuem relevância.

*ExtenderSelection* determina o uso do imediato de maior valor para a informação do endereço de memória, enquanto *lastMuxSel* faz com que esse valor seja passado como *lastMuxOutput*. Tal valor informa o endereço de escrita desejado.

*WriteEnable* informa à memória de dados que haverá escrita, e os dados a serem escritos provêm da saída *dataA* do banco de registradores.

O Código 10 apresenta as conformações mencionadas na forma como se implementou.

Código 10: Instruções de Carregamento e Armazenamento

```

6'b011000: begin//ld
    writeDataSelection = 1'b1;
    writeRegister = 1'b1;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'bxx;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'b01;
    lastMuxSel = 1'b1;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end

6'b011001: begin//st
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'b01;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'bxx;
    lastMuxSel = 1'b1;
    writeEnable = 1'b1;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end

6'b011010: begin//ldi

```

```

writeDataSelection = 1'b0;
writeRegister = 1'b1;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

```

### 3.4.4 Instruções de Controle de Fluxo de Informações

Se enquadram nessa categoria as instruções *pré-branch*, *branch on zero*, *branch on negative* e *jump*. A instrução *jump* funciona de maneira que, caso acionada, o PC ”salte” para o valor informado nessa insrução; as instruções de *branch* apresentam uma condição determinada que, caso satisfeita, soma ao valor do PC o valor contido na instrução.

Para a função *jump*, apenas o endereço e o sinal *jump* são relevantes; ao informar ao PC a respeito do ”salto”, a unidade de controle o provê, também, do endereço para onde deve ”saltar” e, dentro do PC, a instrução é executada quando esse endereço é atribuído ao PC.

As ramificações na presente arquitetura ocorrem através da ação conjunta das instruções *pré-branch* e *branch on zero* ou *branch on negative*. Sendo assim, *pré-branch* verifica a condição do registrador desejado, ao realizar a operação da ULA ’0000’, a qual confere ao dado de saída nada menos que o dado de entrada. Dessa forma, acionam-se as *flags* necessárias e, enfim, *branch on zero* e *branch on negative* podem completar a ramificação.

Assim sendo, para *pré-branch*, é importante que *aluSelection* esteja em ’0000’ e que *enable* esteja acionada, a fim de que se armazene a condição verificada nos devidos *flipflops*.

Para *branch on zero* e *branch on negative*, importa que o endereço informado à UC esteja no campo *mainAddress*, e que se acione a *flag* referente à devida operação: *bzero* para *branch on zero* e *bnegative* para *branch on negative*. O Código 11 apresenta a implementação de todas essas operações.

Código 11: Instruções de Controle de Fluxo de Informações

```
6'b011111: begin//pre-branch
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'b0000;
    extenderSelection = 2'bxx;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'bxx;
    lastMuxSel = 1'bx;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b1;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end

6'b010011: begin//branch on zero
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'b01;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'bxx;
    lastMuxSel = 1'bx;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = immediate[9:0];
    jump = 1'b0;
    bzero = 1'b1;
    bnegative = 1'b0;
    HLT = 0;
end

6'b010100: begin//branch on negative
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'b01;
```

```

immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = immediate[9:0];
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b1;
HLT = 0;
end
6'b010101: begin//jmp
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = immediate[9:0];
jump = 1'b1;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

```

### 3.4.5 Demais Instruções

Além das instruções já citadas, existem as instruções: *sem operação, entrada, saída* e *HLT*.

Em *sem operação*, nenhum sinal é acionado, ao passo que em *HLT*, o sinal *HLT* é acionado e enviado a PC, de modo que as operações são paradas.

Na instrução de saída de dados, não há escrita em registradores ou na memória de dados. Escreve-se, porém, na memória de saída o valor armazenado no registrador informado na instrução; portanto, *IO\_RAMwrite* é acionada. A saída da memória de escrita apresenta-se, portanto, como a saída do sistema.

Para a instrução de entrada, escreve-se o valor passado pelas chaves a um registrador, portanto *writeRegister* é acionada. A outra *flag* de importância é *extenderSelection*, a qual se coloca em estado '10', a fim de que os dados vindos dos dispositivos de entrada sejam recebidos e estendidos antes de serem escritos. O Código 12 apresenta a implementação dessas operações.

Código 12: Instruções de Fluxo de IO e de Controle de Operações

```

6'b011100: begin//hlt
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'bxx;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'bxx;
    lastMuxSel = 1'bx;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 1;
end
6'b011011: begin//nop
    writeDataSelection = 1'bx;
    writeRegister = 1'b0;
    aluSelection = 4'bxxxx;
    extenderSelection = 2'bxx;
    immediateSelection = 1'bx;
    tripleMuxSelection = 2'bxx;
    lastMuxSel = 1'bx;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end

```

```

6'b011101: begin//in
writeDataSelection = 1'b0;
writeRegister = 1'b1;
aluSelection = 4'bxxxx;
extenderSelection = 2'b10;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b011110: begin//pre-out
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'b1;
writeEnable = 1'b0;
IO_RAMwrite = 1'b1;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b100000: begin//out
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;

```

```

lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b1;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

```

### 3.5 Entrada e Saída

A interface entre o sistema computacional e o usuário foi projetada sobre os recursos oferecidos pelo kit FPGA *ALTERA DE2-115 Cyclone IV®*.

Para a entrada de dados, utilizam-se as 18 alavancas presentes no kit, assim como o botão mais à esquerda do painel. Para saída, utilizam-se todos os 8 *displays* de 7 segmentos, assim como 3 LEDS para o retorno de informações.

Na Figura 6 encontram-se indicados os dispositivos utilizados:

- Em vermelho: os *displays* de 7 segmentos;
- em amarelo: um LED específico utilizado para indicar que o número apresentado é negativo;
- em azul: as alavancas de entrada;
- em roxo: um LED utilizado para indicar que uma operação de saída está em andamento;
- em verde: um LED utilizado para indicar que uma operação de entrada está em andamento;
- em branco: o botão utilizado para auxiliar no controle do andamento das operações.

Observação: quando o sistema se encontra em parada de operações (*halt*), todos os LEDs mencionados acendem.

A entrada do sistema é inserida de forma direta a partir das alavancas. A saída, porém, é tratada antes de ser apresentada nos *displays*.

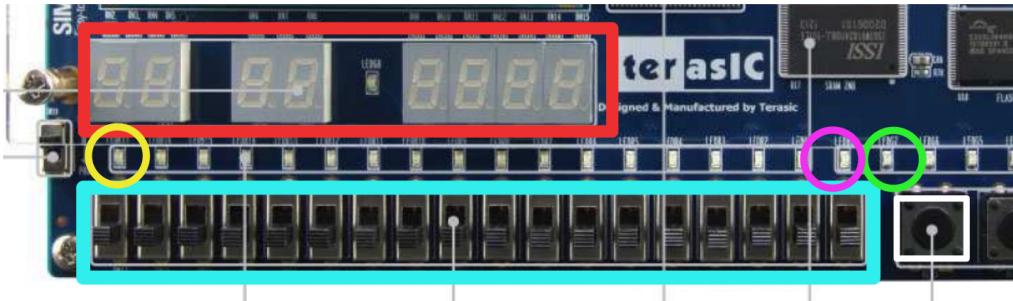


Figura 6: Indicação dos dispositivos utilizados do kit FPGA.

### 3.5.1 Controlador de Saída

Em operações de saída, os *displays* fornecem ao usuário a resposta do sistema às operações realizadas internamente; em operações de entrada, eles fornecem um retorno dos valores que são imputados através das alavancas.

O controlador de saída, portanto, a partir da operação que está sendo realizada, decide a fonte que utilizará para prover as informações a serem apresentadas nos *displays*.

Ele também fornece retorno ao usuário quanto a operação corrente através dos LEDs mencionados na subseção anterior. O Código 13 fornece a implementação desse componente.

Código 13: Controlador de Saída

```
module outputController (operation , switches , IO_RAMOutput ,
inLED , outLED , negLED , binary );

input [5:0] operation ;
input [17:0] switches ;
input [31:0] IO_RAMOutput;
output reg [31:0] binary ;
output reg inLED , outLED , negLED ;

reg temp;

always @ ( * ) begin
temp = IO_RAMOutput [31];
case (operation)
6'b011101: begin
binary = {14'h0000 , switches };//in
inLED = 1;
```

```

outLED = 0;
negLED = 0;
end
6'b100000: begin
inLED = 0;
outLED = 1;
if(temp==0) begin
    binary = IO_RAMOutput; //out
    negLED = 0;
end
if(temp==1) begin
    binary = -IO_RAMOutput; //out
    negLED = 1;
end
end
6'b011100: begin//hlt
binary = 32'b0;
inLED = 1;
outLED = 1;
negLED = 1;
end
default: begin
binary = 32'b0;
inLED = 0;
outLED = 0;
negLED = 0;
end
endcase
end

endmodule // outputController

```

### 3.5.2 Unidade de Saída

Para que um valor em binário seja apresentado nos *displays*, precisa-se que esse valor seja decomposto de modo que as dezenas se aloquem de forma adequada, de acordo com a notação decimal.

Assim, realizou-se uma conversão especial onde, para os *displays* de 7 segmentos, são apresentados valores de 0 a 9, de acordo com o algoritmo disponível em [19], que são passados para cada *display*. O Código 14 apresenta os módulos implementados para a realização adequada da saída do sistema.

Código 14: Componentes de Saída

```
module Output (clock , binary , ones , tens ,
hundreds , thousands , millions ,
billions , trillions , gazillions );

input clock;
input [31:0] binary;
output [6:0] ones , tens , hundreds , thousands , millions ,
billions , trillions , gazillions ;
wire [3:0] binOnes , binTens , binHundreds , binThousands ,
binMillions , binBillions , binTrillions , binGazillions ;

BCD converter(binary , binOnes , binTens , binHundreds , binThousands ,
binMillions , binBillions , binTrillions , binGazillions );

sevenSegmentsDisplay displayA(clock , binOnes ,
ones [0] , ones [1] , ones [2] ,
ones [3] , ones [4] , ones [5] , ones [6]);
sevenSegmentsDisplay displayB(clock , binTens ,
tens [0] , tens [1] , tens [2] ,
tens [3] , tens [4] , tens [5] , tens [6]);
sevenSegmentsDisplay displayC(clock , binHundreds ,
hundreds [0] , hundreds [1] , hundreds [2] ,
hundreds [3] , hundreds [4] , hundreds [5] , hundreds [6]);
sevenSegmentsDisplay displayD(clock , binThousands ,
thousands [0] , thousands [1] , thousands [2] ,
thousands [3] , thousands [4] , thousands [5] , thousands [6]);
sevenSegmentsDisplay displayE(clock , binMillions ,
millions [0] , millions [1] , millions [2] ,
millions [3] , millions [4] , millions [5] , millions [6]);
sevenSegmentsDisplay displayF(clock , binBillions ,
billions [0] , billions [1] , billions [2] ,
billions [3] , billions [4] , billions [5] , billions [6]);
sevenSegmentsDisplay displayG(clock , binTrillions ,
trillions [0] , trillions [1] , trillions [2] ,
trillions [3] , trillions [4] , trillions [5] , trillions [6]);
sevenSegmentsDisplay displayH(clock , binGazillions ,
gazillions [0] , gazillions [1] , gazillions [2] ,
gazillions [3] , gazillions [4] , gazillions [5] ,
gazillions [6]);
```

```

endmodule // Output

module BCD (binary , ones , tens , hundreds ,
thousands , millions ,
billions , trillions , gazillions );

input [31:0] binary;
output reg [3:0] ones , tens , hundreds ,
thousands , millions ,
billions , trillions , gazillions ;
integer i;

always @ ( binary ) begin

gazillions = 4'b0;
trillions = 4'b0;
billions = 4'b0;
millions = 4'b0;
thousands = 4'b0;
hundreds = 4'b0;
tens = 4'b0;
ones = 4'b0;

for (i=31; i>=0; i=i-1)
begin
  if(gazillions >= 5)
    gazillions = gazillions + 3;
  if(trillions >= 5)
    trillions = trillions + 3;
  if(billions >= 5)
    billions = billions + 3;
  if(millions >= 5)
    millions = millions + 3;
  if(thousands >= 5)
    thousands = thousands + 3;
  if(hundreds >= 5)
    hundreds = hundreds + 3;
  if (tens >= 5)
    tens = tens + 3;
  if(ones >=5)

```

```

ones = ones + 3;

gazillions = gazillions << 1;
gazillions[0] = trillions[3];
trillions = trillions << 1;
trillions[0] = billions[3];
billions = billions << 1;
billions[0] = millions[3];
millions = millions << 1;
millions[0] = thousands[3];
thousands = thousands << 1;
thousands[0] = hundreds[3];
hundreds = hundreds << 1;
hundreds[0] = tens[3];
tens = tens << 1;
tens[0] = ones[3];
ones = ones << 1;
ones[0] = binary[i];
end
end

endmodule // BCD

module sevenSegmentsDisplay(clock , inputNumber ,
A, B, C, D, E, F, G);

input clock;
input [3:0] inputNumber;
output A, B, C, D, E, F, G;
reg [6:0] result;

always @ (posedge clock) begin
case (inputNumber)
4'b0000: result = 7'b1111110;
4'b0001: result = 7'b0110000;
4'b0010: result = 7'b1101101;
4'b0011: result = 7'b1111001;
4'b0100: result = 7'b0110011;
4'b0101: result = 7'b1011011;
4'b0110: result = 7'b1011111;
4'b0111: result = 7'b1110000;

```

```

4'b1000: result = 7'b111111;
4'b1001: result = 7'b1111011;
default: result = 7'b0000000;
endcase
end

assign {A, B, C, D, E, F, G} = ~result;

endmodule // sevenSegmentsDisplay

```

### 3.5.3 Componentes Adicionais

Há ainda outros componentes auxiliares para a realização da entrada e saída. Há um divisor de frequência para o *clock*, assim como um *debounce* - código fonte disponível em [20] - para viabilizar o uso do botão.

O sistema opera de maneira que para a maior parte das operações, o *clock* do sistema é adotado. Para operações de entrada, saída e HLT, porém, o componente *clock Multiplexer* confere ao botão de controle do usuário o *clock*.

As implementações desses componentes podem ser encontradas na seção 6.

## 3.6 Implementação da CPU

Conforme a implementação descrita até aqui, a CPU foi desenvolvida em forma de esquemático, a fim de facilitar a compreensão do sistema como um todo, conforme as Figuras 7, 8 e 9.

Desta forma, ela consiste na união dos componentes com o propósito de que eles trabalhem em conjunto para a execução das instruções desejadas.

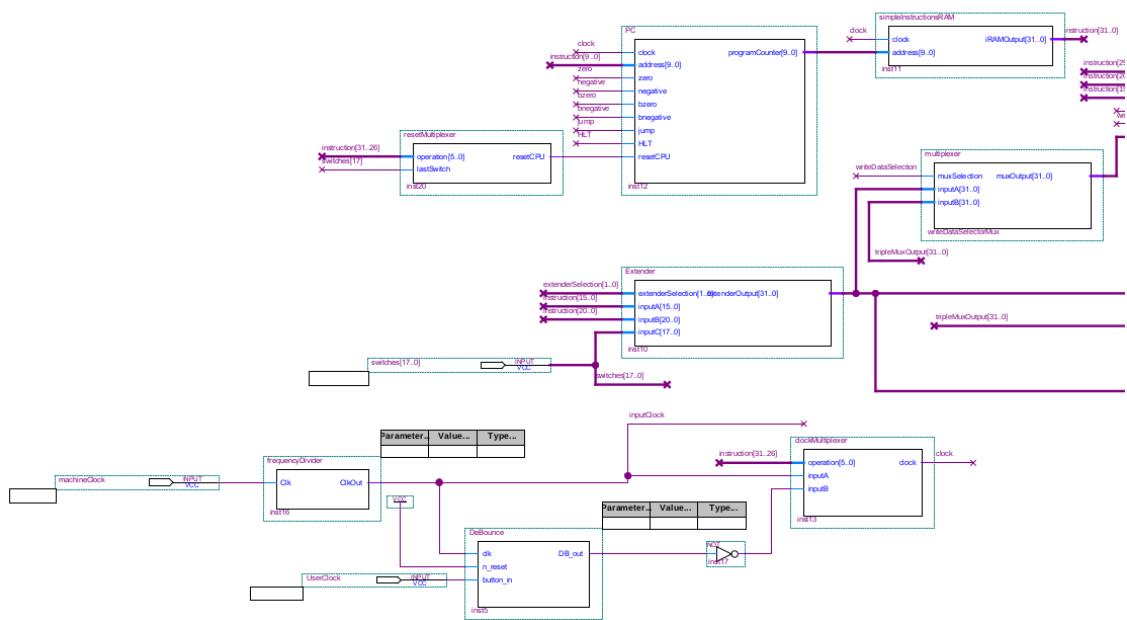


Figura 7: CPU; principais componentes: memória de instruções, PC, extensor de bits, seletor de dados de escrita.

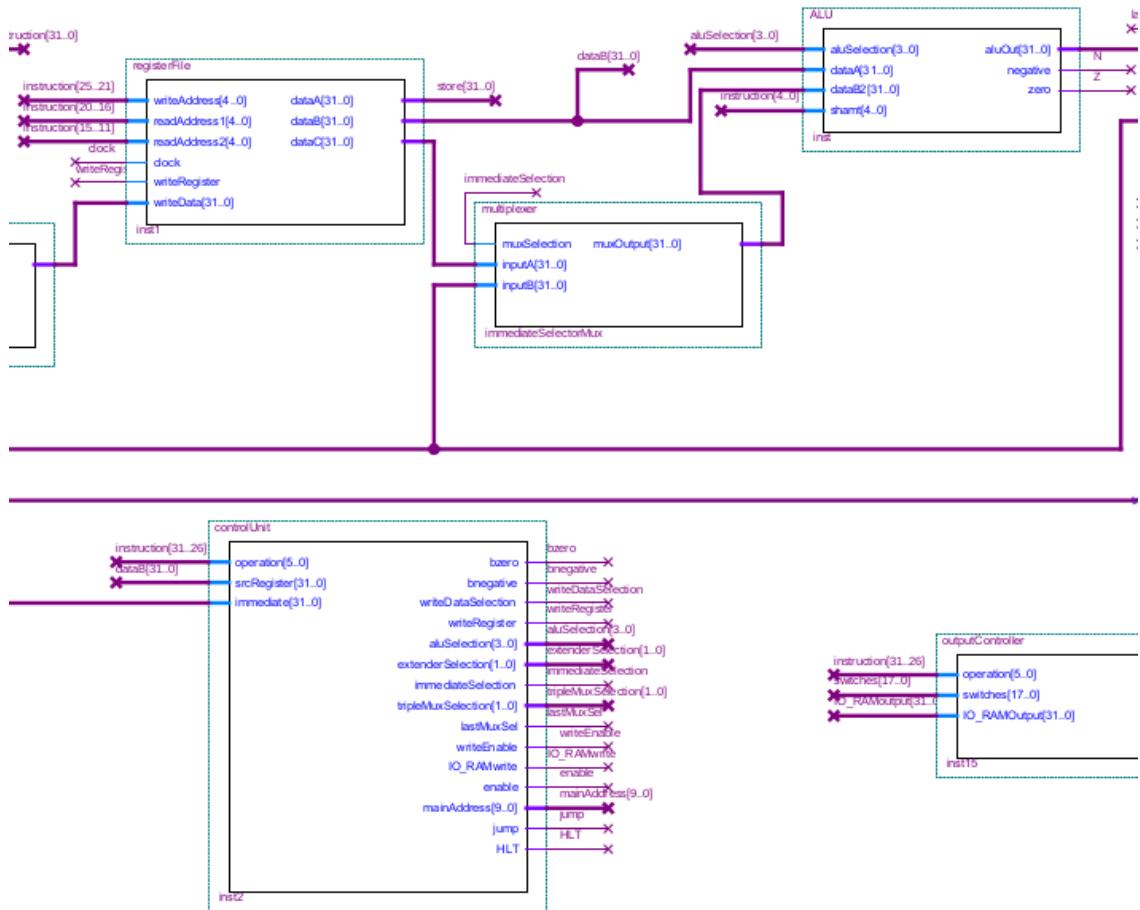


Figura 8: CPU; principais componentes: banco de registradores, seletor de imediato, unidade lógica e aritmética, unidade de controle.

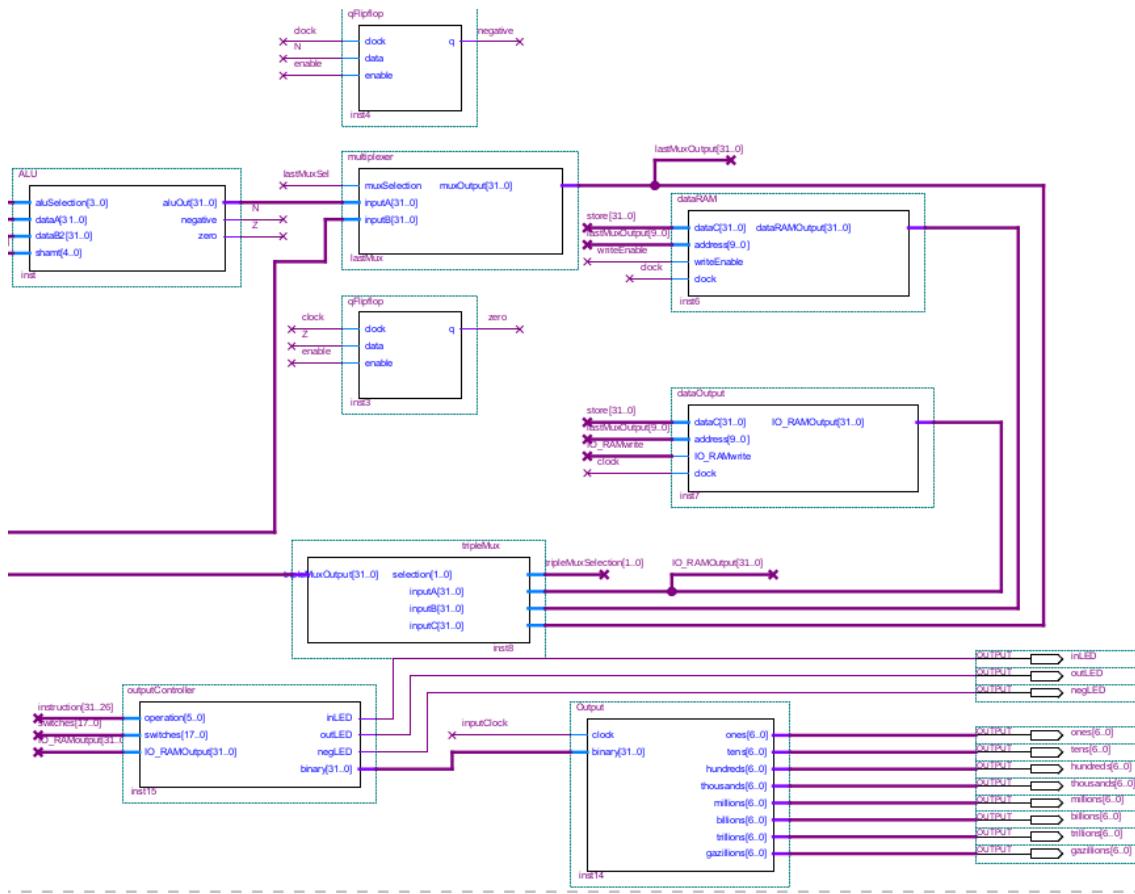


Figura 9: CPU; principais componentes: memória de dados, memória de saída, componentes de saída.

## 4 Resultados

A escolha de um conjunto de instruções RISC se deu por conta da maior facilidade em sua compreensão e do fato de que ele se adequa a projetos de arquitetura simplificada.

Por seu caráter didático - tendo sido previamente estudada na disciplina *Arquitetura e Organização de Computadores* - e pela grande quantidade de material disponível a seu respeito, a arquitetura MIPS foi escolhida como base para a realização do projeto, com uma abordagem monocíclica.

## 4.1 Conjunto de Instruções

As instruções foram adotadas com o objetivo de providenciarem ao usuário as ferramentas necessárias para a resolução de problemas computacionais. Todas as operações aritméticas básicas, assim como instruções lógicas, podem derivar da manipulação das instruções fornecidas.

As instruções que realizam operações aritméticas com imediatos foram adicionadas para que operações mais simples possam ser realizadas de maneira mais rápida, utilizando a menor quantidade de recursos.

Instruções de acesso à memória foram implementadas de forma simples, porém permitindo o uso desse recurso e ampliando as capacidades de armazenamento do sistema.

As instruções de controle foram escolhidas por permitirem a criação de desvios condicionais e acesso direto a partes diferentes do código, recursos fundamentais para a programação. As instruções de entrada e saída adequam-se aos dispositivos disponíveis para a realização do projeto.

### 4.1.1 Modos de Endereçamento

Os modos imediato e direto foram adotados por permitirem operações que utilizam os recursos de maneira eficiente em situações onde seja possível um alcance limitado.

Escolheu-se o modo por registrador por ele ser ideal para determinados tipos de operações, como as operações lógicas e aritméticas, que precisam ser realizadas relativamente rápido mas necessitam, também, de operandos de tamanho razoável.

## 4.2 Esboço da Arquitetura do Processador

O projeto da arquitetura foi realizado com o objetivo de se adequar às instruções definidas. Portanto, assim como no caso do conjunto de instruções, a arquitetura se baseou no MIPS.

Os dispositivos de entrada e saída foram adaptados ao projeto dessa arquitetura, assim como a presença de imediatos de tamanhos distintos e modificações realizadas em virtude de funções como BOZ e BON, onde seu resultado funciona como seletor, de maneira que o valor do imediato é somado ou não ao endereço seguinte para o PC. A Figura 10 o esboço inicial simplificado para a arquitetura elaborada.

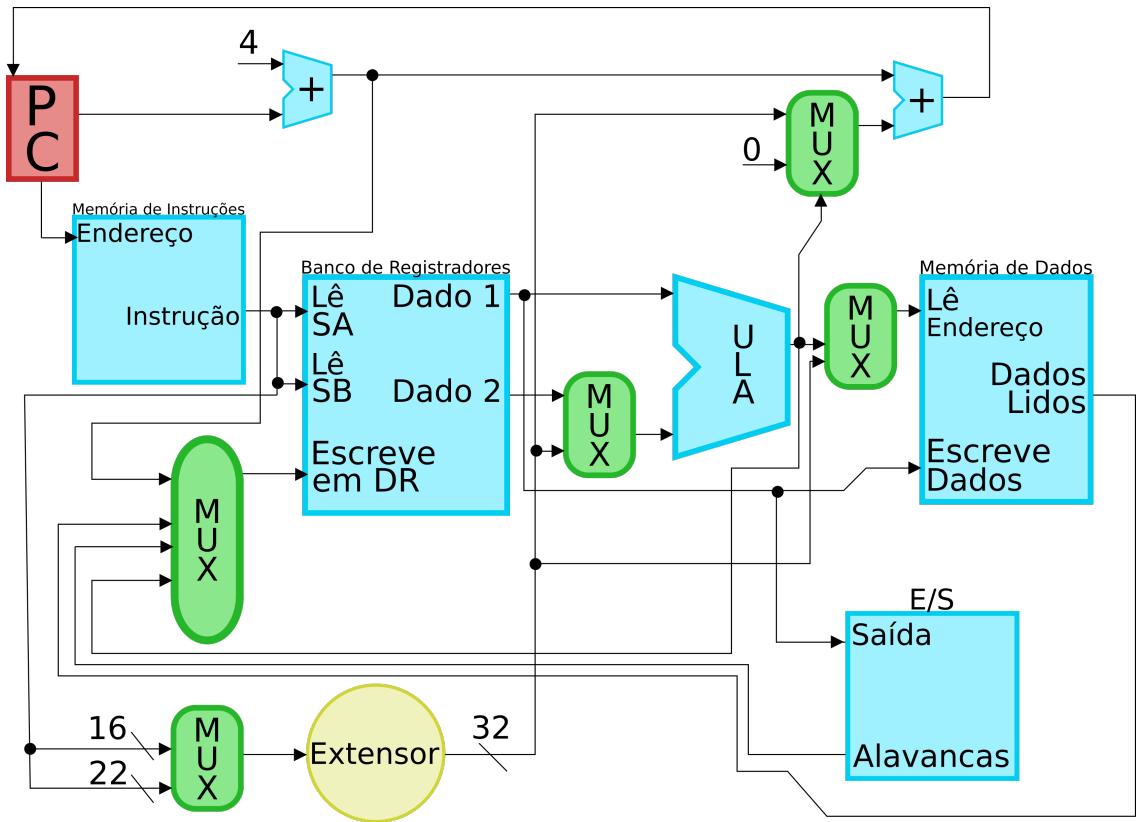


Figura 10: Esquemático do sistema computacional projetado.

### 4.3 Testes em Forma de Onda

Os primeiros testes do sistema se realizaram sobre o *software Altera Quartus II®*, em forma de onda. A unidade de controle, ao ser implementada, foi inserida diretamente sobre o processador já desenvolvido a fim de controlar seus componentes. Esses testes foram realizados antes da introdução dos componentes de entrada e saída; os operandos foram, portanto, inseridos através da função ldi, onde se carrega um imediato a um registrador.

Algumas séries de instruções foram elaboradas para a verificação do funcionamento do sistema; elas são descritas e comentadas a seguir.

#### 4.3.1 Instruções A

Tais instruções se desenrolaram na seguinte ordem:

1. nop: inicialização do sistema

2. carrega-se 6 ao registrador 4
3. carrega-se 4 ao registrador 5
4. a soma dos valores dos registradores 4 e 5 são carregadas ao registrador 1
5. o resultado de  $\text{reg}[4]\text{reg}[5]$  é carregado ao  $\text{reg}[2]$
6. o incremento do  $\text{reg}[5]$  é carregado ao  $\text{reg}[3]$
7. o decremento do  $\text{reg}[4]$  é carregado ao  $\text{reg}[3]$
8.  $\text{reg}[4] \& \text{reg}[5] \rightarrow \text{reg}[7]$
9.  $\text{reg}[4] \parallel \text{reg}[5] \rightarrow \text{reg}[8]$
10.  $\text{reg}[4] \text{ xor } \text{reg}[5] \rightarrow \text{reg}[9]$
11.  $\text{reg}[4]$  negado é carregado a  $\text{reg}[10]$
12. instruções seguintes: saída dos registradores 1, 2, 3, 6, 7, 8, 9 e 10

Conforme a Figura 11, todas as saídas obtiveram os valores esperados; portanto as instruções de carregamento de imediato, muitas das instruções lógicas e aritméticas e instruções de saída foram realizadas com sucesso pela CPU.

Isso é verificável de maneira que o campo *output I* se refere à instrução que está sendo realizada, enquanto *output B* apresenta a saída do sistema.

#### 4.4 Instruções B

1. nop: inicialização do sistema
2. carrega-se 6 ao registrador 4
3. carrega-se 4 ao registrador 5
4. desloca 3 à esquerda do  $\text{reg}[4] \rightarrow \text{reg}[1]$
5. desloca 2 à direita do  $\text{reg}[4] \rightarrow \text{reg}[2]$
6. set on less than  $\text{reg}[4] < \text{reg}[5] \rightarrow \text{reg}[3]$
7. set on less than  $\text{reg}[5] < \text{reg}[4] \rightarrow \text{reg}[6]$



Figura 11: Testes em forma de onda para *Instruções A*.

8.  $\text{reg}[4] \times \text{reg}[5] \rightarrow \text{reg}[7]$
9.  $\text{reg}[7] \div \text{reg}[5] \rightarrow \text{reg}[8]$
10.  $\text{reg}[7] \bmod \text{reg}[5] \rightarrow \text{reg}[9]$
11. guarda  $\text{reg}[5]$  em  $M[7]$
12. carrega  $M[7]$  a  $\text{reg}[10]$
13. instruções seguintes: saída dos registradores 1, 2, 3, 6, 7, 8, 9 e 10

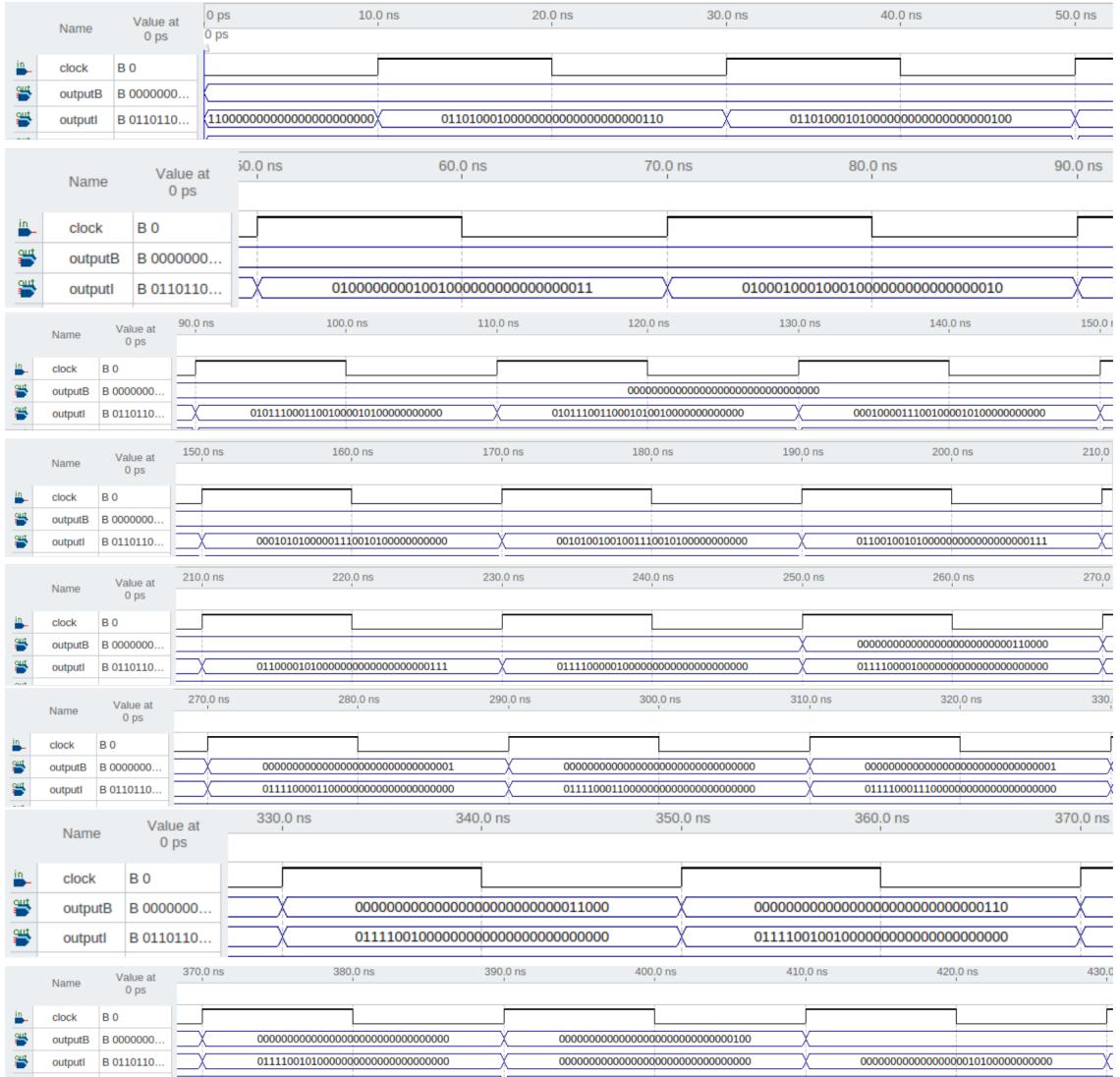


Figura 12: Testes em forma de onda para *Instruções B*.

Nesse grupo de instruções, foram abordadas as demais instruções aritméticas, outras instruções abordadas anteriormente e as instruções de carregamento e armazenamento. Todas as saídas apresentaram os valores esperados, portanto a CPU funcionou de maneira satisfatória. Os resultados estão presentes na Figura 12.

## 4.5 Instruções C

1. nop: inicialização do sistema

2. carrega-se 0 ao registrador 5
3. carrega-se 6 ao registrador 4
4. subtrai-se 6 do valor referente ao registrador 4 e o carrega ao reg[5]
5. ramificação: caso o resultado anterior seja zero, soma-se 1 ao valor seguinte do PC
6. Sem operação
7. saída do registrador 4

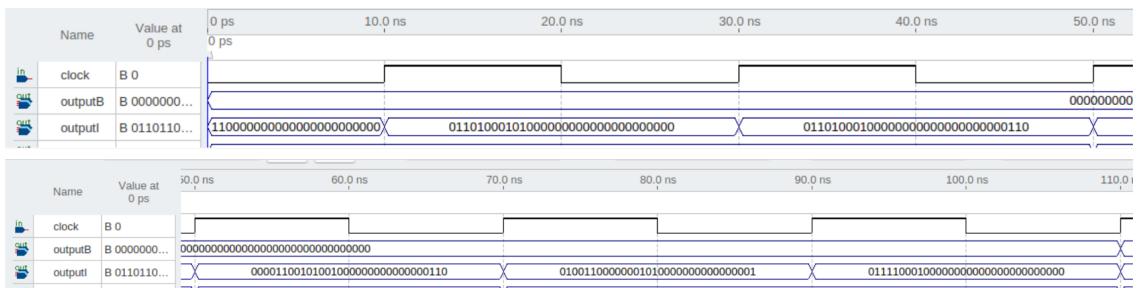


Figura 13: Testes em forma de onda para *Instruções C*.

Aqui se testa a instrução *branch on zero*, a qual funciona sob a seguinte condição: se o valor da operação anterior é igual a zero, altera-se o PC.

Na maior parte das implementações, ela é acompanhada da instrução *pré-branch*; isso, porém, não é imperativo. Os resultados estão presentes na Figura 13.

Esse grupo de instruções serve para mostrar que o controle sobre o valor do PC acontece de forma satisfatória na presente implementação. Testes mais aprofundados sobre instruções desse tipo - tais como *Branch on Negative* e *Jump* - se apresentam na subseção 4.6.

## 4.6 Testes no kit FPGA

Após a integração do sistema computacional com os dispositivos de entrada e saída, elaborou-se um algoritmo que testa todas as instruções fornecidas pelo sistema.

Esse algoritmo se divide em três sub-algoritmos:

- sub-algoritmo A: apresenta o funcionamento de instruções de acesso à memória, entrada, saída, e instruções lógicas e aritméticas;

- sub-algoritmo B: é um algoritmo funcional que realiza a multiplicação de dois números inteiros, fazendo uso de instruções de controle, como *jump* e *branch on zero*;
- sub-algoritmo C: a partir de duas entradas, ele imprime a menor, apresentando a instrução *branch on zero*, e mostra também a saída do *set on less than* para os valores inseridos.

Ao inicializar o sistema, o usuário tem como escolher qual algoritmo deseja seguir; isso é implementado via *software* através de ramificações.

Dessa forma, se o usuário insere um valor na primeira entrada, ele escolhe o sub-algoritmo A. Caso insira zero na primeira e outro valor na segunda, escolhe o sub-algoritmo B. Se o usuário não insere valor algum na primeira e segunda entrada, escolhe o sub-algoritmo C.

#### 4.6.1 Sub-algoritmo A

Dada a descrição anterior, o sub-algoritmo A realiza as seguintes operações:

1. *nop*: inicialização do sistema
2. entrada ao reg 0 e reg 1
3.  $reg0 + reg1 \rightarrow reg2$
4.  $reg0 + 2 \rightarrow reg3$
5.  $reg1 - reg0 \rightarrow reg4$
6.  $reg1 - 2 \rightarrow reg5$
7.  $reg0 \times reg1 \rightarrow reg6$
8.  $reg1 \div reg0 \rightarrow reg7$
9.  $reg0 + 1 \rightarrow reg8$
10.  $reg0 - 1 \rightarrow reg9$
11.  $reg0 \text{ and } reg1 \rightarrow reg10$
12.  $reg0 \text{ or } reg1 \rightarrow reg11$
13.  $reg0 \text{ restore } reg1 \rightarrow reg12$
14.  $reg0 \text{ xor } reg1 \rightarrow reg13$

15.  $\text{not } reg1 \rightarrow reg14$
16.  $reg0 \text{ sl2} \rightarrow reg15$
17.  $reg1 \text{ sl1} \rightarrow reg16$
18.  $\text{store } reg0 \rightarrow mem7$
19.  $\text{load } mem7 \rightarrow reg17$
20.  $\text{output } reg2 \text{ a } reg17$

Podem-se conferir as operações realizadas através do Código 15.

Código 15: Sub-algoritmo A

```

instructionsRAM[0] = 32'b01101100000000000000000000000000;
instructionsRAM[1] = 32'b01110100001000000000000000000000;
instructionsRAM[2] = 32'b01111000000000100000000000000000;
instructionsRAM[3] = 32'b010011000000000000000000100101;
instructionsRAM[4] = 32'b01110100000000000000000000000000;
instructionsRAM[5] = 32'b01110100001000000000000000000000;
instructionsRAM[6] = 32'b00000000100001000000000000000000;
instructionsRAM[7] = 32'b00001000110000000000000000000010;
instructionsRAM[8] = 32'b00001000100000100000000000000000;
instructionsRAM[9] = 32'b00001100101000100000000000000010;
instructionsRAM[10] = 32'b00010000110000010000000000000000;
instructionsRAM[11] = 32'b00010100110000100000000000000000;
instructionsRAM[12] = 32'b00011001000000000000000000000000;
instructionsRAM[13] = 32'b00011101001000000000000000000000;
instructionsRAM[14] = 32'b00100001010000000001000000000000;
instructionsRAM[15] = 32'b00100101011000010000000000000000;
instructionsRAM[16] = 32'b00101001100000100000000000000000;
instructionsRAM[17] = 32'b00110001101000010000000000000000;
instructionsRAM[18] = 32'b00110101100000100000000000000000;
instructionsRAM[19] = 32'b01000001111000000000000000000010;
instructionsRAM[20] = 32'b01000110000000010000000000000001;
instructionsRAM[21] = 32'b01100100000000000000000000000011;
instructionsRAM[22] = 32'b011000100010000000000000000000111;
instructionsRAM[23] = 32'b100000000100000000000000000000000;
instructionsRAM[24] = 32'b1000000001100000000000000000000000;
instructionsRAM[25] = 32'b10000000010000000000000000000000000;
instructionsRAM[26] = 32'b10000000010100000000000000000000000000000;
instructionsRAM[27] = 32'b100000000110000000000000000000000000000000;

```

```

instructionsRAM[28] = 32'b1000000011100000000000000000000000000000 ;
instructionsRAM[29] = 32'b100000100000000000000000000000000000000 ;
instructionsRAM[30] = 32'b100000100100000000000000000000000000000 ;
instructionsRAM[31] = 32'b100000101000000000000000000000000000000 ;
instructionsRAM[32] = 32'b100000101100000000000000000000000000000 ;
instructionsRAM[33] = 32'b100000110000000000000000000000000000000 ;
instructionsRAM[34] = 32'b100000110100000000000000000000000000000 ;
instructionsRAM[35] = 32'b100000111000000000000000000000000000000 ;
instructionsRAM[36] = 32'b100000111100000000000000000000000000000 ;
instructionsRAM[37] = 32'b100000100000000000000000000000000000000 ;
instructionsRAM[38] = 32'b1000001000100000000000000000000000000000 ;
instructionsRAM[39] = 32'b10000011110000000000000000000000000000000 ;
instructionsRAM[40] = 32'b0111000000000000000000000000000000000000000000000000 ;

```

Como se pode observar através das figuras 14, 15, 16, 17 e 18, o sistema apresentou os resultados esperados para as operações realizadas.

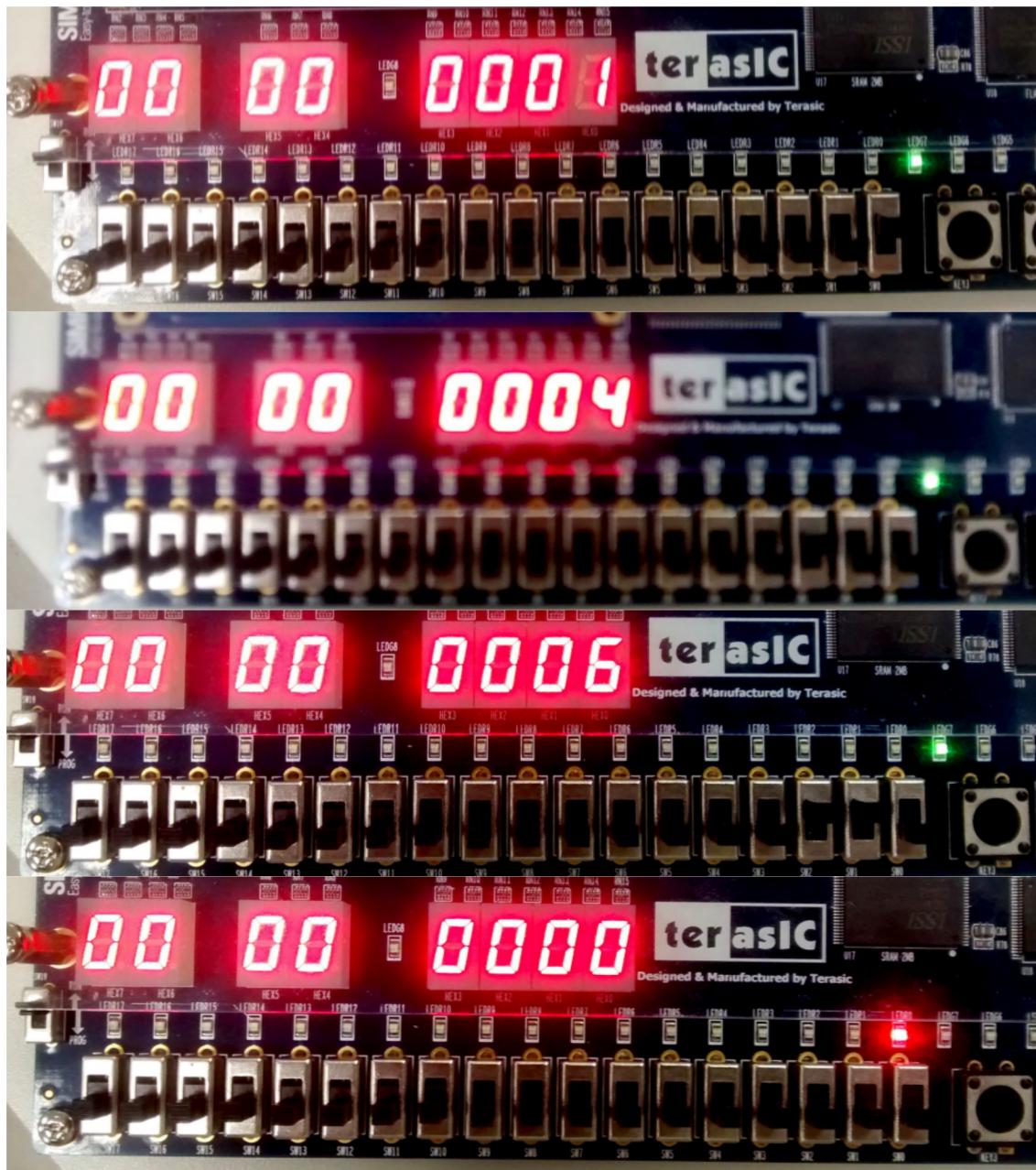


Figura 14: Resultados para o sub-algoritmo A, parte 1.

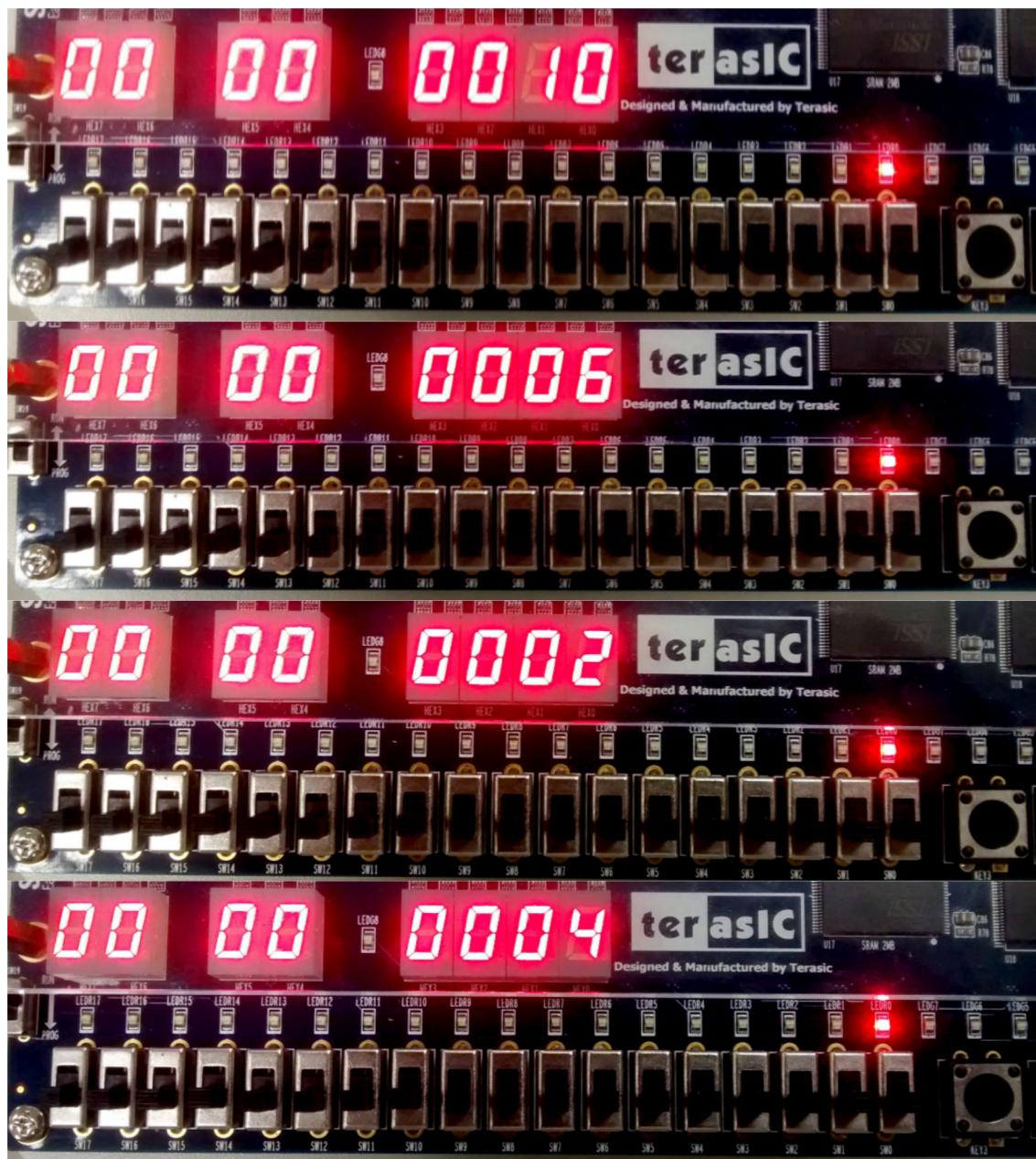


Figura 15: Resultados para o sub-algoritmo A, parte 2.

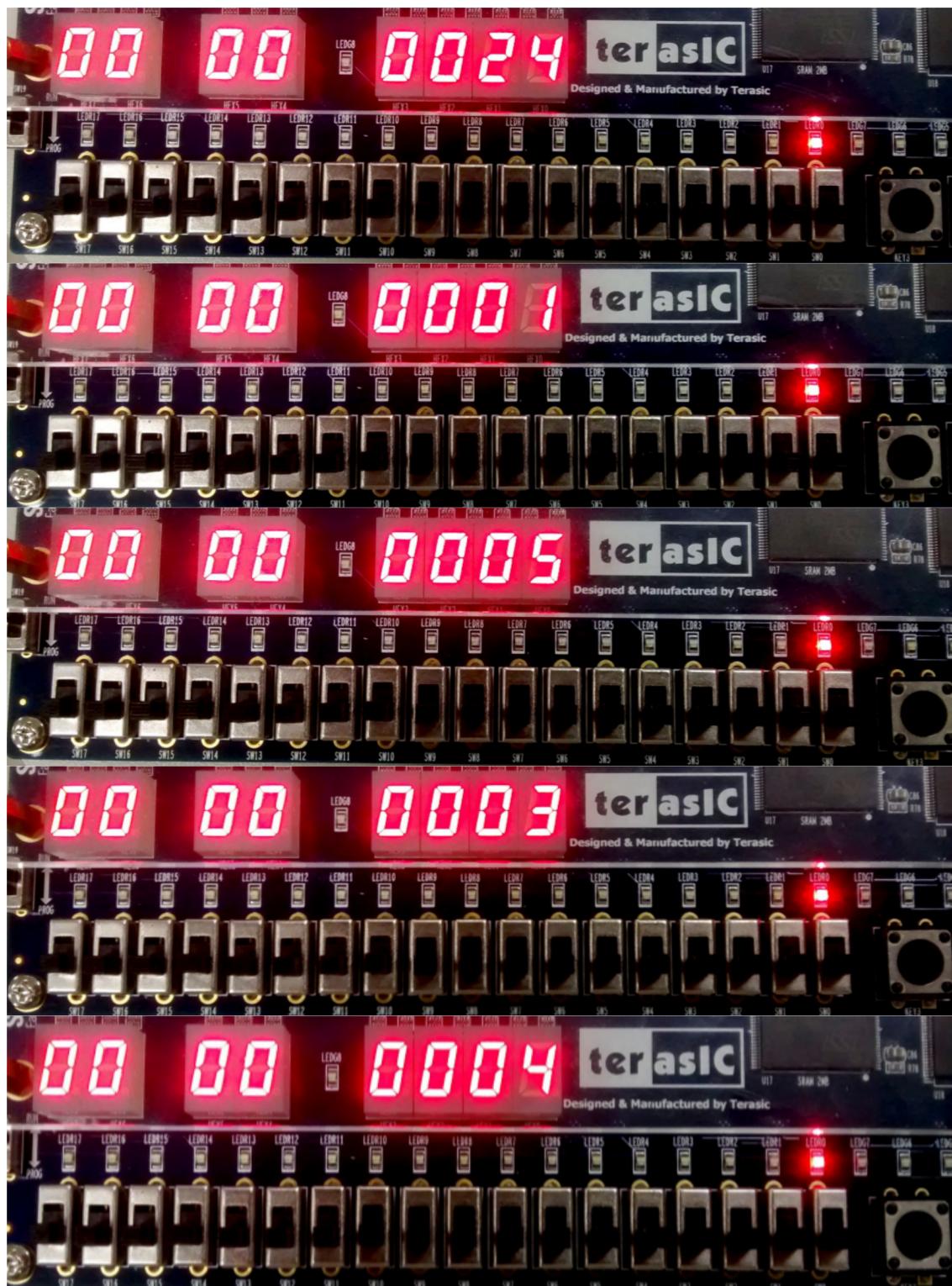


Figura 16: Resultados para o sub-algoritmo A, parte 3.  
66

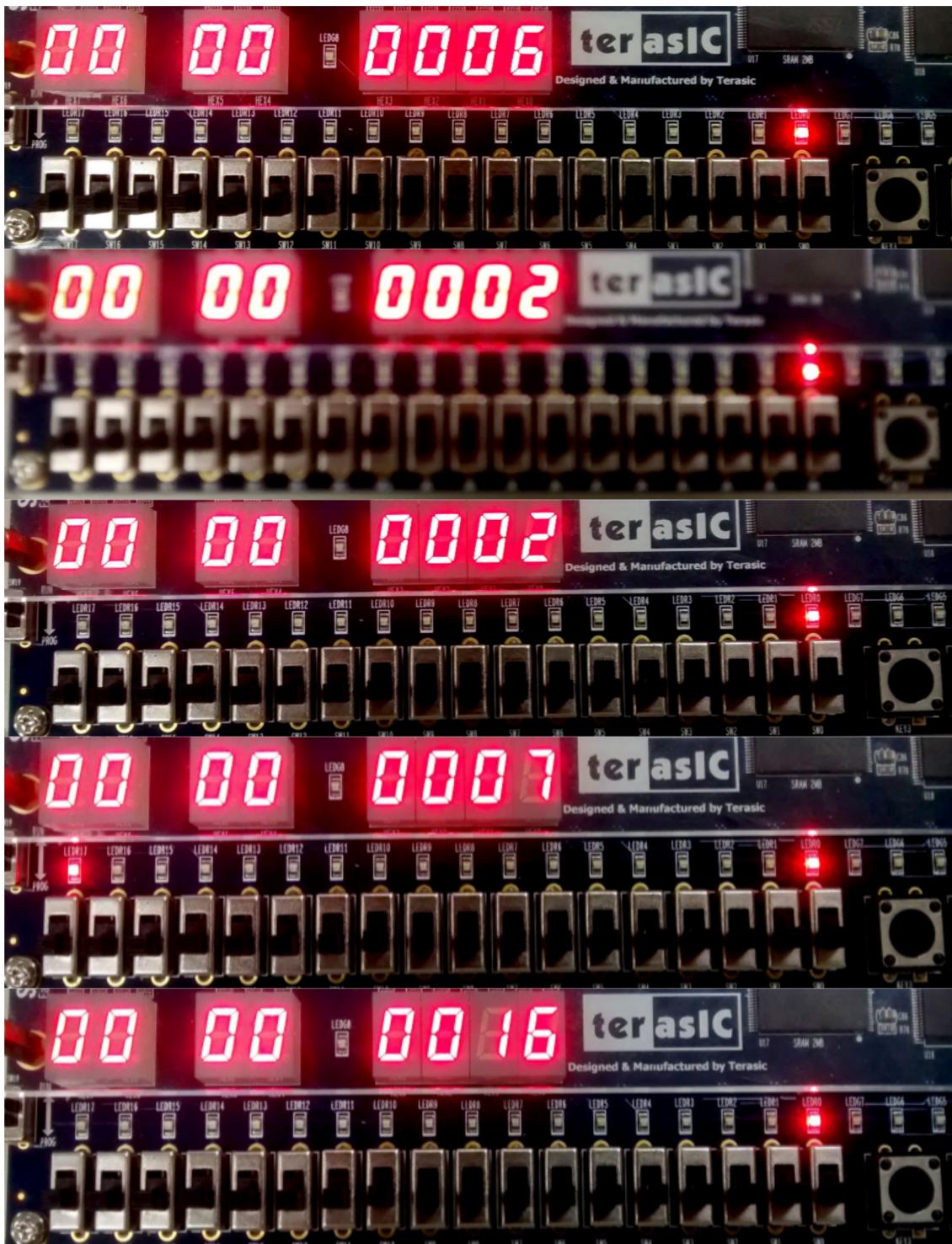


Figura 17: Resultados para o sub-algoritmo A, parte 4.

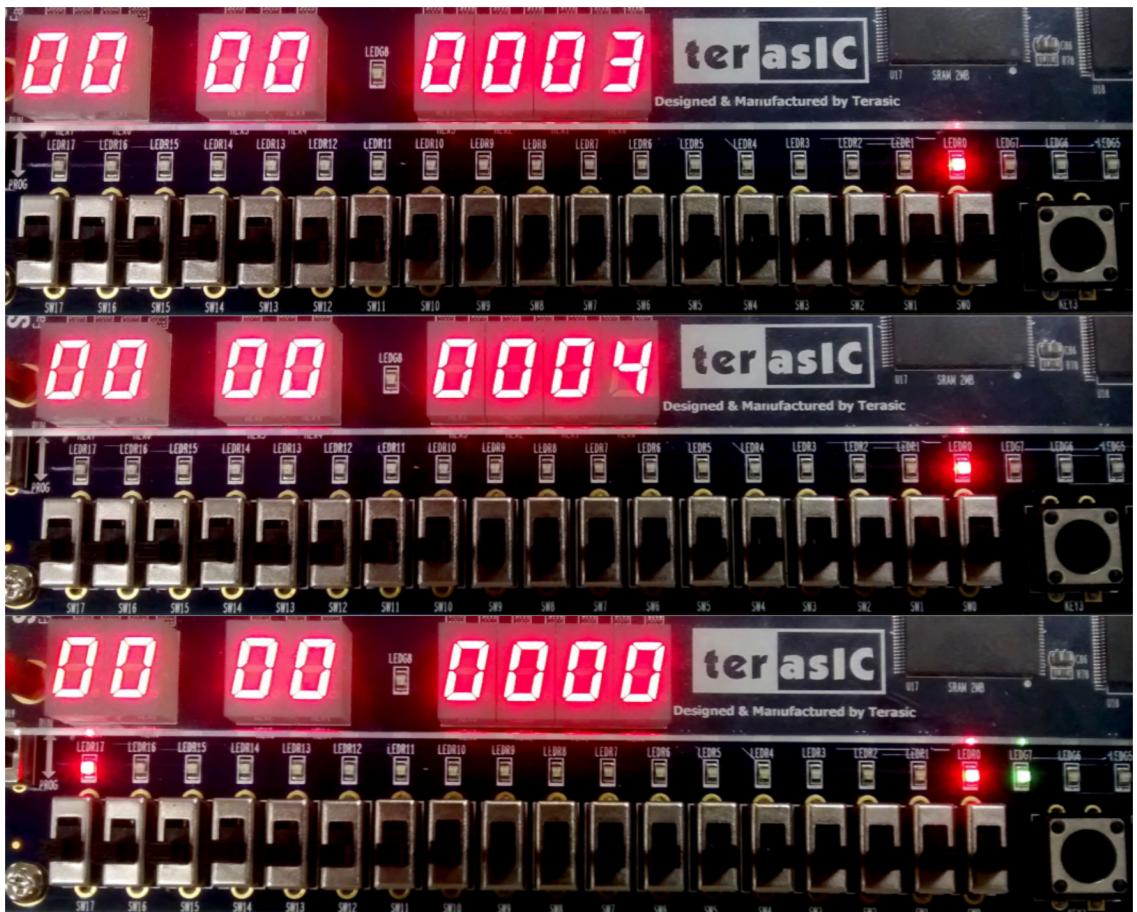


Figura 18: Resultados para o sub-algoritmo A, parte 5.

#### 4.6.2 Sub-algoritmo B

O sub-algoritmo B realiza a multiplicação da seguinte maneira:

1. *nop*: inicialização do sistema
2. entrada ao reg 0 e reg 1
3.  $ldi0 \rightarrow reg5$
4.  $se reg1 == 0 \rightarrow PC = PC + 1 + 5$
5.  $reg0 + reg5 \rightarrow reg5$
6.  $reg1 - 1 \rightarrow reg1$

7. output *reg5*
8. *jump* para o condicional
9. output *reg5*

As instruções realizadas se encontram em detalhe no Código 16.

Código 16: Sub-algoritmo B

```
instructionsRAM[41] = 32'b01110100001000000000000000000000;
instructionsRAM[42] = 32'b01111000000000100000000000000000;
instructionsRAM[43] = 32'b010011000000000000000000000000001100;
instructionsRAM[44] = 32'b01110100000000000000000000000000000000;
instructionsRAM[45] = 32'b0111010000100000000000000000000000000000;
instructionsRAM[46] = 32'b011010001010000000000000000000000000000;
instructionsRAM[47] = 32'b011110000000001000000000000000000000000;
instructionsRAM[48] = 32'b01001100000000000000000000000000101;
instructionsRAM[49] = 32'b00000000101001010000000000000000000000;
instructionsRAM[50] = 32'b00011000010000100000000000000000;
instructionsRAM[51] = 32'b1000000010100000000000000000000000000;
instructionsRAM[52] = 32'b01010100000000000000000000000000101111;
instructionsRAM[53] = 32'b100000001010000000000000000000000000000;
instructionsRAM[54] = 32'b10000011110000000000000000000000000000;
instructionsRAM[55] = 32'b011100000000000000000000000000000000000000;
```

As Figuras 19 e 20 apresentam as etapas da multiplicação dos valores inseridos, a qual se deu de forma correta, mostrando que o sistema computacional é capaz de realizar a execução de algoritmos.

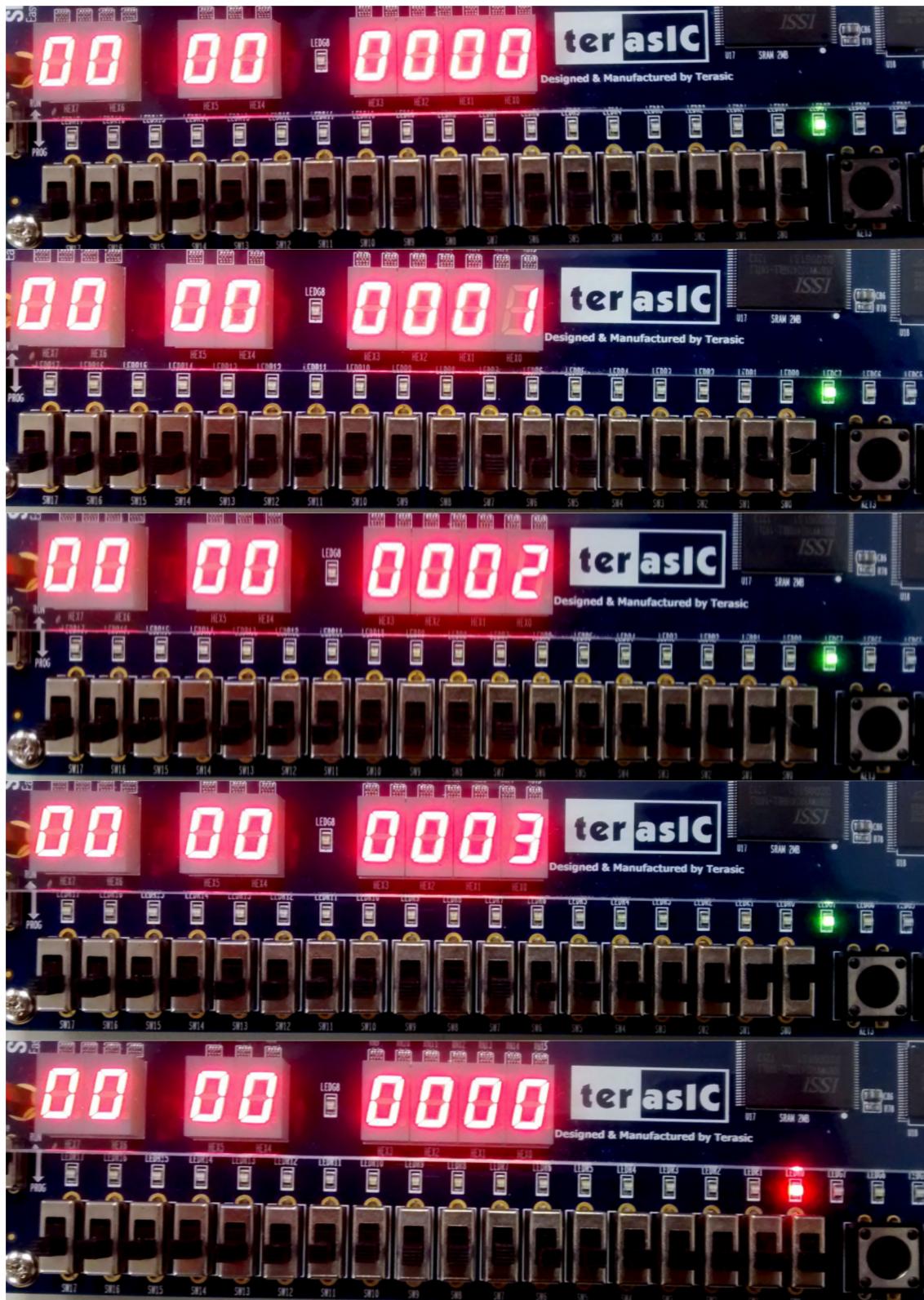


Figura 19: Resultados para o sub-algoritmo B, parte 1.

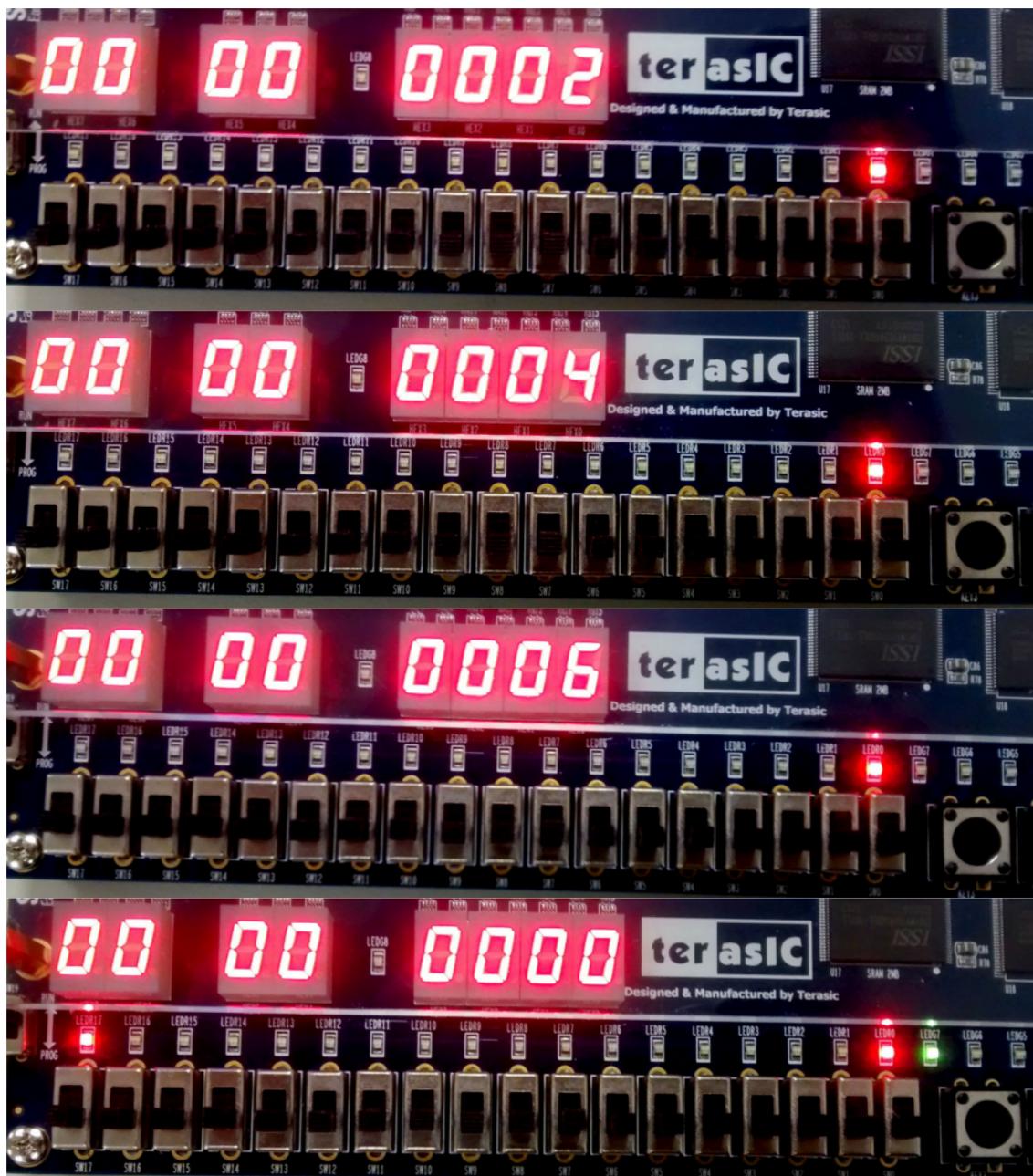


Figura 20: Resultados para o sub-algoritmo B, parte 2.

#### 4.6.3 Sub-algoritmo C

O sub-algoritmo C opera conforme se enumera a seguir:

1. *nop*: inicialização do sistema
2. entrada ao reg 0 e reg 1
3.  $reg1 - reg0 \rightarrow reg4$
4.  $se\ reg4 < 0 \rightarrow PC = PC + 1 + 2$
5. output *reg0*
6. *jump para slt*
7. output *reg1*
8. *slt reg1 < reg0 → reg5*
9. *slt reg0 < reg1 → reg4*
10. output *reg4 e 5*

As instruções realizadas se encontram em detalhe no Código 17.

Código 17: Sub-algoritmo B

```
instructionsRAM[56] = 32'b01110100000000000000000000000000;
instructionsRAM[57] = 32'b01110100001000000000000000000000;
instructionsRAM[58] = 32'b00001000100000100000000000000000;
instructionsRAM[59] = 32'b01111000000010000000000000000000;
instructionsRAM[60] = 32'b01010000000000000000000000000010;
instructionsRAM[61] = 32'b10000000000000000000000000000000;
instructionsRAM[62] = 32'b010101000000000000000000000000001000000;
instructionsRAM[63] = 32'b100000000010000000000000000000000;
instructionsRAM[64] = 32'b01011001010000000010000000000000;
instructionsRAM[65] = 32'b01011001000000100000000000000000;
instructionsRAM[66] = 32'b100000001000000000000000000000000;
instructionsRAM[67] = 32'b100000001010000000000000000000000;
instructionsRAM[68] = 32'b1000001111000000000000000000000000;
instructionsRAM[69] = 32'b01110000000000000000000000000000;
```

Como se pode observar através das figuras 21 e 22, o sistema apresentou os resultados esperados para as operações realizadas, uma vez que imprimiu o menor valor passado (2), afirmou *slt 2 < 4* e negou *slt 4 < 2*.



Figura 21: Resultados para o sub-algoritmo C, parte 1.

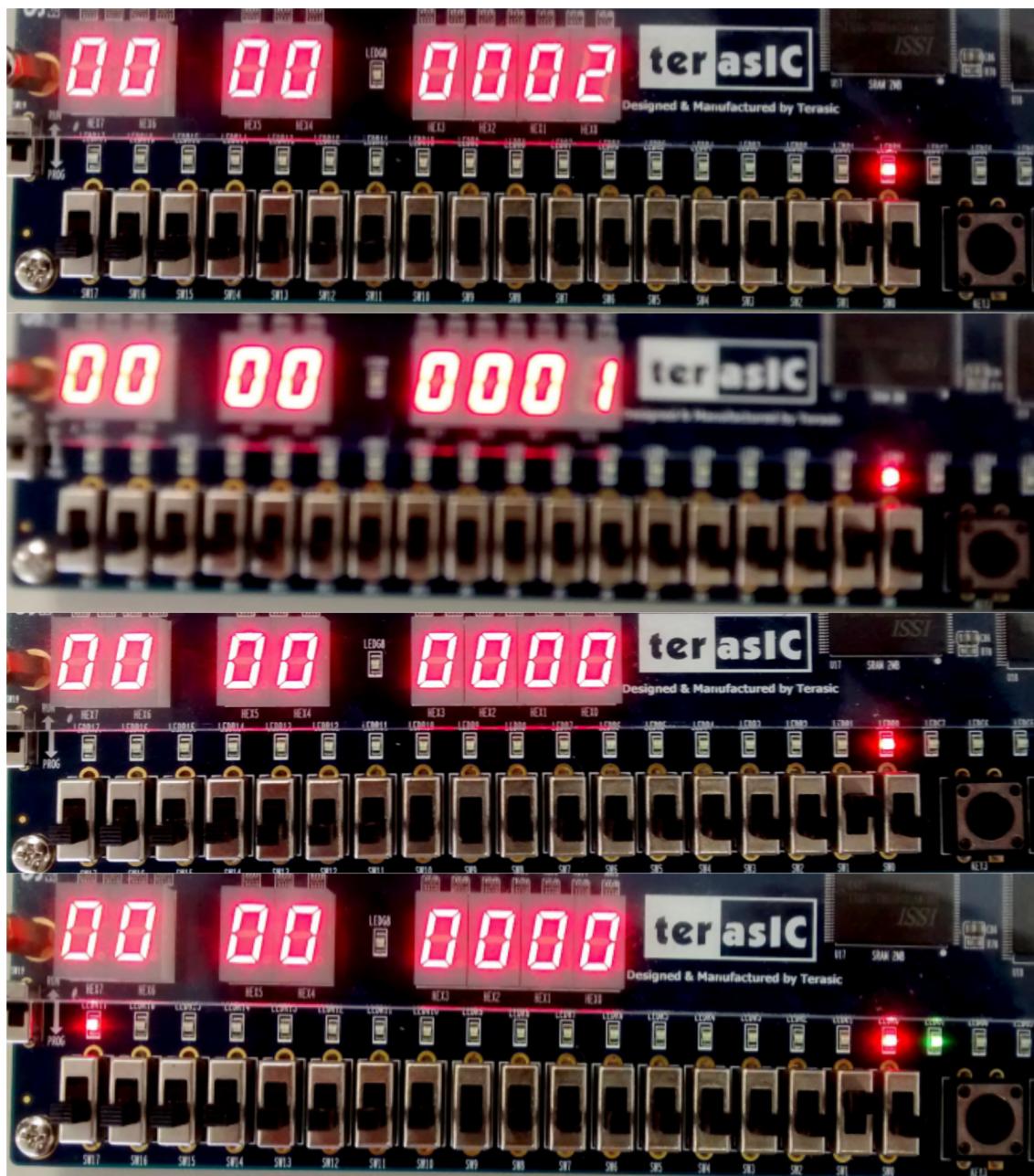


Figura 22: Resultados para o sub-algoritmo C, parte 2.

## 5 Conclusão

O desenvolvimento desse sistema computacional permitiu um aprendizado mais profundo a respeito de como funciona uma CPU. Proporcionou, também, o desenvolvimento de critérios de decisão quanto à elaboração de projetos a depender de seus fins.

Os seguintes desafios foram enfrentados ao longo do processo de desenvolvimento: inexperiência na elaboração de projetos dessa natureza e magnitude; escolha apropriada de arquitetura a se desenvolver; integração dos componentes da CPU e da Unidade de Controle; integração do sistema com o dispositivo físico.

Por seu caráter didático, o sistema desenvolvido ainda conta com diversas limitações; portanto, ainda há bastante espaço para elaborações futuras sobre ele, como a adição do uso do *Display* de LCD e um uso mais amplo do acesso à memória.

## Referências

- [1] Mano Morris, C Kime. 2008. *Logic and Computer Design Fundamentals*, PEARSON, Fourth Edition editora, edição, paginas
- [2] Cambridge Dictionary. *Computer Meaning*. Disponível em: <<http://dictionary.cambridge.org/pt/dicionario/ingles/computer>>. Acesso em: 3 de abril de 2016.
- [3] Stanford University. *Risc Cisc*. Disponível em: <<http://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/riscscisc/>>. Acesso em: 19 de março de 2016.
- [4] Microsoft Developer Network. *Branch if Equal to Zero*. Disponível em: <[https://msdn.microsoft.com/en-us/library/aa259006\(v=vs.60\).aspx](https://msdn.microsoft.com/en-us/library/aa259006(v=vs.60).aspx)>. Acesso em: 3 de abril de 2016.
- [5] Ian Moor. 2009. *Branch and Jump Instructions*. Disponível em:<<https://www.doc.ic.ac.uk/lab/secondyear/spim/node16.html>>. Acesso em: 3 de abril de 2016.
- [6] University of Idaho. 1998. *MIPS Instruction Reference*. Disponível em:<<http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>>. Acesso em: 3 de abril de 2016.
- [7] Universidade de Hamburgo. *MIPS*. Disponível em:<<https://tams.informatik.uni-hamburg.de/applets/hades/webdemos/mips.html>>. Acesso em: 3 de abril de 2016.
- [8] TECHNOPEDIA. Control Unit. Disponível em: <<https://www.techopedia.com/definition/2855/control-unit-cu>>. Acesso em 5 de Julho de 2016.
- [9] Davis University of California. *MIPS Architecture Overview*. Disponível em:<<http://web.cs.ucdavis.edu/peisert/teaching/ecs142-sp09/rt.html>>. Acesso em: 2 de abril de 2016.
- [10] PATTERSON, J. L. H. D. A. Organização e Projeto de Computadores, A interface harware/software. 3rd. ed. [S.l.: s.n.], 2005.
- [11] New York University. 1999-2000. *Class Notes for Computer Architecture*. Disponível em:<<http://cs.nyu.edu/courses/fall99/V22.0436-001/class-notes.html>>. Acesso em: 4 de abril de 2016.

- [12] VERILOG. *Verilog Resources*. Disponível em:<<http://www.verilog.com/>>. Acesso em: 4 de abril de 2016.
- [13] Universidade Estadual de Campinas. *Introdução FPGA Fluxo de Projeto*. Disponível em:< [http://www.decom.fee.unicamp.br/cardoso/ie344b/Introducao\\_FPGA\\_Fluxo\\_de\\_Projeto.pdf](http://www.decom.fee.unicamp.br/cardoso/ie344b/Introducao_FPGA_Fluxo_de_Projeto.pdf)>. Acesso em: 4 de abril de 2016.
- [14] University of Michigan. *de2-115*. Disponível em:<<https://web.eecs.umich.edu/~pmchen/engr100/lab1/de2-115.jpg>>. Acesso em: 4 de abril de 2016.
- [15] Study. Central Processing Unit (CPU): Parts, Definition and Function. Disponível em: <<http://study.com/academy/lesson/central-processing-unit-cpu-parts-definition-function.html>>. Acesso em 9 de Maio de 2016.
- [16] Ilustração disponível em: <[http://www.inf.ed.ac.uk/teaching/courses/inf2c/labs/inf2c\\_cs\\_l3](http://www.inf.ed.ac.uk/teaching/courses/inf2c/labs/inf2c_cs_l3)>. Acesso em 9 de Maio de 2016.
- [17] Universidade Técnica Chalmers. Finite-State Machines. Disponível em:<<http://www.cse.chalmers.se/~coquand/AUTOMATA/-book.pdf>>. Acesso em 13 de Julho de 2016.
- [18] DAENOTES. Flip Flop. Disponível em:<<http://www.daenotes.com/electronics/digital-electronics/flip-flops-types-applications-wokingtextgreater>>. Acesso em 13 de Julho de 2016.
- [19] Utah Engineering. *BCD Tutorial*. Disponível em:<<http://www.eng.utah.edu/~nmcdonal/Tutorials/BCDTutorial/BCDConversion.html>>. Acesso em: 4 de julho de 2016.
- [20] EEWIKI. *Debounce Code*. Disponível em:<[http://www.eewiki.net/display/LOGIC-/Debounce+Logic+Circuit+\(with+Verilog+example\)](http://www.eewiki.net/display/LOGIC-/Debounce+Logic+Circuit+(with+Verilog+example))>. Acesso em: 4 de julho de 2016.

## 6 Anexos

### 6.1 Códigos complementares

Código 18: Unidade de Controle

```
module controlUnit(operation , srcRegister , immediate ,
                   bzero , bnegative , writeDataSelection , writeRegister ,
                   aluSelection , extenderSelection , immediateSelection ,
                   tripleMuxSelection , lastMuxSel , writeEnable ,
                   IO_RAMwrite , enable , mainAddress , jump , HLT); //, branch);

  input [5:0] operation;
  input [31:0] srcRegister , immediate;
  // input zero , negative;

  output reg [9:0] mainAddress; //instruction [9..0];
  output reg [3:0] aluSelection;
  output reg [1:0] extenderSelection , tripleMuxSelection ;
  output reg jump , writeDataSelection , immediateSelection ,
  lastMuxSel , writeEnable , IO_RAMwrite , enable , writeRegister ,
  bzero , bnegative , HLT; //, branch;

  always @ ( operation ) begin
    case (operation)
      6'b000000: begin //add
        writeDataSelection = 1'b1;
        writeRegister = 1'b1;
        aluSelection = 4'b0001;
        extenderSelection = 2'bxx;
        immediateSelection = 1'b0;
        tripleMuxSelection = 2'b10;
        lastMuxSel = 1'b0;
        writeEnable = 1'b0;
        IO_RAMwrite = 1'b0;
        enable = 1'b1;
        mainAddress = 10'b0;
        jump = 1'b0;
        HLT = 0;
        bzero = 1'b0;
        bnegative = 1'b0;
      end
    endcase
  end
```

```

end
6'b000001: begin//addi
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0001;
extenderSelection = 2'b00;
immediateSelection = 1'b1;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000010: begin//sub
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0010;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000011: begin//subi
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0010;
extenderSelection = 2'b00;
immediateSelection = 1'b1;

```

```

tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000100: begin//mul
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1100;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000101: begin//div
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1101;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

```

```

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000110: begin//inc
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0011;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b000111: begin//dec
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0100;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

```

```

6'b001010: begin//mod
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1110;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b001000: begin//and
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0101;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end

6'b001001: begin//or
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0110;
extenderSelection = 2'bxx;

```

```

immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001100: begin//xor
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b0111;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b001101: begin//not
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1000;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;

```

```

enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;

bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b010000: begin//shift left
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1001;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b010001: begin//shift right
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1010;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;

```

```

end
6'b011111: begin//pre-branch
  writeDataSelection = 1'bx;
  writeRegister = 1'b0;
  aluSelection = 4'b0000;
  extenderSelection = 2'bxx;//2'b01;
  immediateSelection = 1'bx;
  tripleMuxSelection = 2'bxx;
  lastMuxSel = 1'bx;
  writeEnable = 1'b0;
  IO_RAMwrite = 1'b0;
  enable = 1'b1;//importante para testes
  mainAddress = 10'b0;//immediate[9:0];
  jump = 1'b0;
  bzero = 1'b0;
  bnegative = 1'b0;
  HLT = 0;
end
6'b010011: begin//branch on zero
  writeDataSelection = 1'bx;
  writeRegister = 1'b0;
  aluSelection = 4'bxxxx;
  extenderSelection = 2'b01;//01
  immediateSelection = 1'bx;
  tripleMuxSelection = 2'bxx;
  lastMuxSel = 1'bx;
  writeEnable = 1'b0;
  IO_RAMwrite = 1'b0;
  enable = 1'b0;//importante para testes
  mainAddress = immediate[9:0];
  jump = 1'b0;

  bzero = 1'b1;
  bnegative = 1'b0;
  HLT = 0;
end
6'b010100: begin//branch on negative
  writeDataSelection = 1'bx;
  writeRegister = 1'b0;
  aluSelection = 4'bxxxx;
  extenderSelection = 2'b01;//01

```

```

immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0; //importante para testes
mainAddress = immediate[9:0];
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b1;
HLT = 0;
end
6'b010101: begin//jmp
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01; //xx
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0; //vai que
mainAddress = immediate[9:0];
jump = 1'b1;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b010111: begin//slt
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'b1011;
extenderSelection = 2'bxx;
immediateSelection = 1'b0;
tripleMuxSelection = 2'b10;
lastMuxSel = 1'b0;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b1;
mainAddress = 10'b0;

```

```

jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b011000: begin//ld
writeDataSelection = 1'b1;
writeRegister = 1'b1;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'b01;
lastMuxSel = 1'b1;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b011001: begin//st
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'b1;
writeEnable = 1'b1;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b011010: begin//ldi
writeDataSelection = 1'b0;

```

```

writeRegister = 1'b1;
aluSelection = 4'bxxxx;
extenderSelection = 2'b01;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b011100: begin//hlt
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 1;
end
6'b011011: begin//nop
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;

```

```

IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b011101: begin//in
writeDataSelection = 1'b0;
writeRegister = 1'b1;
aluSelection = 4'bxxxx;
extenderSelection = 2'b10;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b0;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;
end
6'b100000: begin//out
writeDataSelection = 1'bx;
writeRegister = 1'b0;
aluSelection = 4'bxxxx;
extenderSelection = 2'bxx;
immediateSelection = 1'bx;
tripleMuxSelection = 2'bxx;
lastMuxSel = 1'bx;
writeEnable = 1'b0;
IO_RAMwrite = 1'b1;
enable = 1'b0;
mainAddress = 10'b0;
jump = 1'b0;
bzero = 1'b0;
bnegative = 1'b0;
HLT = 0;

```

```

end
default: begin
    writeDataSelection = 1'b0;
    writeRegister = 1'b0;
    aluSelection = 4'b0000;
    extenderSelection = 2'b00;
    immediateSelection = 1'b0;
    tripleMuxSelection = 2'b00;
    lastMuxSel = 1'b0;
    writeEnable = 1'b0;
    IO_RAMwrite = 1'b0;
    enable = 1'b0;
    mainAddress = 10'b0;
    jump = 1'b0;
    bzero = 1'b0;
    bnegative = 1'b0;
    HLT = 0;
end
endcase
end

endmodule //controlUnit

```

Código 19: Debounce  
*// DeBounce.v.v*

```

////////////////// Button Debounceer //////////////////
//***** *****
// FileName: DeBounce.v.v
// FPGA: MachXO2 7000HE
// IDE: Diamond 2.0.1
//
// HDL IS PROVIDED "AS IS." DIGI-KEY EXPRESSLY DISCLAIMS ANY
// WARRANTY OF ANY KIND, WHETHER EXPRESS OR IMPLIED, INCLUDING BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
// PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL DIGI-KEY
// BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL
// DAMAGES, LOST PROFITS OR LOST DATA, HARM TO YOUR EQUIPMENT, COST OF
// PROCUREMENT OF SUBSTITUTE GOODS, TECHNOLOGY OR SERVICES, ANY CLAIMS
// BY THIRD PARTIES (INCLUDING BUT NOT LIMITED TO ANY DEFENSE THEREOF)

```

```

// ANY CLAIMS FOR INDEMNITY OR CONTRIBUTION, OR OTHER SIMILAR COSTS.
// DIGI-KEY ALSO DISCLAIMS ANY LIABILITY FOR PATENT OR COPYRIGHT
// INFRINGEMENT.
//
// Version History
// Version 1.0 04/11/2013 Tony Storey
// Initial Public Release
// Small Footprint Button Debouncer

'timescale 1 ns / 100 ps
module DeBounce
(
    input                     clk , n_reset , button_in ,
    output reg      DB_out
);
// ////////////////////////////////////////////////////////////////// internal constants // //
parameter N = 11 ;
// ////////////////////////////////////////////////////////////////// internal variables // //
reg [N-1 : 0] q_reg ;
reg [N-1 : 0] q_next ;
reg DFF1, DFF2;
wire q_add ;
wire q_reset ;
// //////////////////////////////////////////////////////////////////
// ////////////////////////////////////////////////////////////////// contentious assignment for counter control
assign q_reset = (DFF1 ^ DFF2);           /
assign q_add = ~(q_reg[N-1]);
// //////////////////////////////////////////////////////////////////
// ////////////////////////////////////////////////////////////////// combo counter to manage q-next
always @ ( q_reset , q_add , q_reg )
begin
    case( { q_reset , q_add })
        2'b00 :                               q_next <= q_reg ;
        2'b01 :                               q_next <= q_reg + 1;
        default :                            q_next <= { N {1'b0} }
    endcase
end

```

```

//// Flip flop inputs and q_reg update
always @ ( posedge clk )
begin
    if (n_reset == 1'b0)
        begin
            DFF1 <= 1'b0;
            DFF2 <= 1'b0;
            q_reg <= { N {1'b0} };
        end
    else
        begin
            DFF1 <= button_in;
            DFF2 <= DFF1;
            q_reg <= q_next;
        end
    end

//// counter control
always @ ( posedge clk )
begin
    if (q_reg [N-1] == 1'b1)
        DB_out <= DFF2;
    else
        DB_out <= DB_out;
    end
endmodule

```

Código 20: Divisor de Frequência

```

//module temporizador( Clk , Rst , ClkOut );
module frequencyDivider( Clk , ClkOut );

    //input Clk , Rst ;
    input Clk;
    output reg ClkOut;

    parameter DivVal = 2500;
    reg [12:0] DivCnt;
    // parameter DivVal = 1250;
    //reg [24:0] DivCnt;
    reg ClkInt;

```

```

always @(posedge Clk) begin
/*      if( Rst == 1 ) begin
    DivCnt <= 0;
    ClkOut <= 0;
    ClkInt <= 0;
end
else*/ begin
    if( DivCnt == (DivVal-1) ) begin
        ClkOut <= ~ClkInt ;
        ClkInt <= ~ClkInt ;
        DivCnt <= 0;
end
else begin
        ClkOut <= ClkInt ;
        ClkInt <= ClkInt ;
        DivCnt <= DivCnt + 1;
end
end
end
endmodule

```

Código 21: Clock Multiplexer

```

module clockMultiplexer (operation , inputA , inputB , clock );

input inputA , inputB ;
input [5:0] operation ;
output reg clock ;

always @ ( * ) begin
    case(operation)
        6'b011101: clock = inputB ;//in
//        6'b011110: clock = inputB; //out
        6'b100000: clock = inputB ;//out
        6'b011100: clock = inputB ;//hlt
    default: clock = inputA ;
    endcase
end

endmodule // outputMultiplexer

```

### Código 22: CPU

```
// Copyright (C) 1991–2015 Altera Corporation. All rights reserved.  
// Your use of Altera Corporation's design tools, logic functions  
// and other software and tools, and its AMPP partner logic  
// functions, and any output files from any of the foregoing  
// (including device programming or simulation files), and any  
// associated documentation or information are expressly subject  
// to the terms and conditions of the Altera Program License  
// Subscription Agreement, the Altera Quartus Prime License Agreement,  
// the Altera MegaCore Function License Agreement, or other  
// applicable license agreement, including, without limitation,  
// that your use is for the sole purpose of programming logic  
// devices manufactured by Altera and sold by Altera or its  
// authorized distributors. Please refer to the applicable  
// agreement for further details.  
  
// PROGRAM "Quartus Prime"  
// VERSION "Version 15.1.0 Build 185 10/21/2015 SJ Lite E  
// CREATED "Tue Jul 5 21:14:37 2016"  
  
module CPU(  
    machineClock ,  
    UserClock ,  
    switches ,  
    inLED ,  
    outLED ,  
    negLED ,  
    billions ,  
    gazillions ,  
    hundreds ,  
    millions ,  
    offLEDS ,  
    ones ,  
    tens ,  
    thousands ,  
    trillions  
);  
  
input wire machineClock ;
```

```

input wire UserClock;
input wire [17:0] switches;
output wire inLED;
output wire outLED;
output wire negLED;
output wire [6:0] billions;
output wire [6:0] gazillions;
output wire [6:0] hundreds;
output wire [6:0] millions;
output wire [22:0] offLEDS;
output wire [6:0] ones;
output wire [6:0] tens;
output wire [6:0] thousands;
output wire [6:0] trillions;

wire [3:0] aluSelection;
wire bnegative;
wire bzero;
wire clock;
wire [31:0] dataB;
wire enable;
wire [1:0] extenderSelection;
wire HLT;
wire immediateSelection;
wire inputClock;
wire [31:0] instruction;
wire [31:0] IO_RAMOutput;
wire IO_RAMwrite;
wire jump;
wire [31:0] lastMuxOutput;
wire lastMuxSel;
wire [9:0] mainAddress;
wire N;
wire negative;
wire [31:0] store;
wire [31:0] tripleMuxOutput;
wire [1:0] tripleMuxSelection;
wire writeDataSelection;
wire writeEnable;
wire writeRegister;
wire Z;

```

```

wire      zero;
wire      [31:0] SYNTHESIZED_WIRE_0;
wire      [31:0] SYNTHESIZED_WIRE_15;
wire      [31:0] SYNTHESIZED_WIRE_2;
wire      [31:0] SYNTHESIZED_WIRE_3;
wire      [9:0]  SYNTHESIZED_WIRE_4;
wire      SYNTHESIZED_WIRE_5;
wire      SYNTHESIZED_WIRE_6;
wire      SYNTHESIZED_WIRE_7;
wire      [31:0] SYNTHESIZED_WIRE_8;
wire      SYNTHESIZED_WIRE_10;
wire      [31:0] SYNTHESIZED_WIRE_11;
wire      [31:0] SYNTHESIZED_WIRE_12;

assign    offLEDS = 23'b0000000000000000000000000000;
assign    SYNTHESIZED_WIRE_10 = 1;

```

```

multiplexer   b2v_immediateSelectorMux(
    . muxSelection(immediateSelection),
    . inputA(SYNTHESIZED_WIRE_0),
    . inputB(SYNTHESIZED_WIRE_15),
    . muxOutput(SYNTHESIZED_WIRE_2));

```

```

ALU        b2v_inst(
    . aluSelection(aluSelection),
    . dataA(dataB),
    . dataB2(SYNTHESIZED_WIRE_2),
    . shamt(instruction[4:0]),
    . negative(N),
    . zero(Z),
    . aluOut(SYNTHESIZED_WIRE_12));

```

```

registerFile   b2v_inst1(
    . clock(clock),
    . writeRegister(writeRegister),
    . readAddress1(instruction[20:16]),

```

```

.readAddress2(instruction[15:11]),
.writeAddress(instruction[25:21]),
.writeData(SYNTHESIZED_WIRE_3),
.dataA(store),
.dataB(dataB),
.dataC(SYNTHESIZED_WIRE_0));

```

```

Extender      b2v_inst10(
    .extenderSelection(extenderSelection),
    .inputA(instruction[15:0]),
    .inputB(instruction[20:0]),
    .inputC(switches),
    .extenderOutput(SYNTHESIZED_WIRE_15));

```

```

simpleInstructionsRAM   b2v_inst11(
    .clock(clock),
    .address(SYNTHESIZED_WIRE_4),
    .iRAMOutput(instruction));

```

```

PC      b2v_inst12(
    .clock(clock),
    .zero(zero),
    .negative(negative),
    .bzero(bzero),
    .bnegative(bnegative),
    .jump(jump),
    .HLT(HLT),
    .resetCPU(SYNTHESIZED_WIRE_5),
    .address(instruction[9:0]),
    .programCounter(SYNTHESIZED_WIRE_4));

```

```

clockMultiplexer      b2v_inst13(
    .inputA(inputClock),
    .inputB(SYNTHESIZED_WIRE_6),
    .operation(instruction[31:26]),
    .clock(clock));

```

```

outputController      b2v_inst15(
    .IO_RAMOutput(IO_RAMOutput),
    .operation(instruction[31:26]),
    .switches(switches),
    .inLED(inLED),
    .outLED(outLED),
    .negLED(negLED),
    .binary(SYNTHESIZED_WIRE_8));

frequencyDivider     b2v_inst16(
    .Clk(machineClock),
    .ClkOut(inputClock));
defparam           b2v_inst16.DivVal = 2500;

assign   SYNTHESIZED_WIRE_6 = ~SYNTHESIZED_WIRE_7;

finalOutput          b2v_inst18(
    .clock(inputClock),
    .binary(SYNTHESIZED_WIRE_8),
    .billions(billions),
    .gazillions(gazillions),
    .hundreds(hundreds),
    .millions(millions),
    .ones(ones),
    .tens(tens),
    .thousands(thousands),
    .trillions(trillions));

controlUnit          b2v_inst2(
    .immediate(SYNTHESIZED_WIRE_15),
    .operation(instruction[31:26]),
    .srcRegister(dataB),
    .bzero(bzero),
    .bnegative(bnegative),
    .writeDataSelection(writeDataSelection),
    .writeRegister(writeRegister),
    .immediateSelection(immediateSelection),

```

```

        . lastMuxSel(lastMuxSel),
        . writeEnable(writeEnable),
        . IO_RAMwrite(IO_RAMwrite),
        . enable(enable),
        . jump(jump),
        . HLT(HLT),
        . aluSelection(aluSelection),
        . extenderSelection(extenderSelection),

        . tripleMuxSelection(tripleMuxSelection));

resetMultiplexer      b2v_inst20(
    . lastSwitch(switches[17]),
    . operation(instruction[31:26]),
    . resetCPU(SYNTHESIZED_WIRE_5));

qFlipflop            b2v_inst3(
    . clock(clock),
    . data(Z),
    . enable(enable),
    . q(zero));

qFlipflop            b2v_inst4(
    . clock(clock),
    . data(N),
    . enable(enable),
    . q(negative));

DeBounce              b2v_inst5(
    . clk(inputClock),
    . n_reset(SYNTHESIZED_WIRE_10),
    . button_in(UserClock),
    . DB_out(SYNTHESIZED_WIRE_7));
defparam           b2v_inst5.N = 11;

```

```

dataRAM b2v_inst6(
    . writeEnable(writeEnable),
    . clock(clock),
    . address(lastMuxOutput[9:0]),
    . dataC(store),
    . dataRAMOutput(SYNTHESIZED_WIRE_11));

dataOutput      b2v_inst7(
    . IO_RAMwrite(IO_RAMwrite),
    . clock(clock),
    . address(lastMuxOutput[9:0]),
    . dataC(store),
    . IO_RAMOutput(IO_RAMOutput));

tripleMux      b2v_inst8(
    . inputA(IO_RAMOutput),
    . inputB(SYNTHESIZED_WIRE_11),
    . inputC(lastMuxOutput),
    . selection(tripleMuxSelection),
    . tripleMuxOutput(tripleMuxOutput));

multiplexer    b2v_lastMux(
    . muxSelection(lastMuxSel),
    . inputA(SYNTHESIZED_WIRE_12),
    . inputB(SYNTHESIZED_WIRE_15),
    . muxOutput(lastMuxOutput));

multiplexer    b2v_writeDataSelectorMux(
    . muxSelection(writeDataSelection),
    . inputA(SYNTHESIZED_WIRE_15),
    . inputB(tripleMuxOutput),
    . muxOutput(SYNTHESIZED_WIRE_3));

endmodule

```

## 6.2 Esquemático Completo

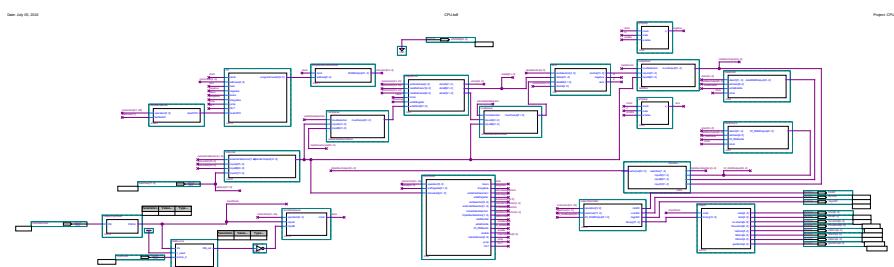


Figura 23: CPU desenvolvida.