

ID:

**Desenvolvimento da Unidade de
Processamento: Processador *bbtron*
ENHANCED[®] .**

São José dos Campos - Brasil 🔥

Abril de 2017

ID:

Desenvolvimento da Unidade de Processamento: Processador *bbtron ENHANCED*® .

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

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

São José dos Campos - Brasil 🔥

Abril de 2017

Resumo

Este documento apresenta os dados obtidos através da elaboração de uma Unidade Central de Processamento(UCP) para a disciplina de Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores denominada *bbtron ENHANCED®*. A UCP foi desenvolvida com a utilização da linguagem de descrição de *Hardware Verilog* e a simulação dos resultados no *Software Intel® Quartus Prime*. Com a unidade central de processamento implementada, por fim, foi realizado o teste de seu funcionamento através da placa *Altera's Cyclone® IV FPGA* para certificação do funcionamento total do *bbtron ENHANCED®*.

Palavras-chaves: UCP. UNIFESP. Arquitetura de Organização de Computadores. Engenharia da Computação. Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores.

Listas de ilustrações

Figura 1 – Esquemático <i>bbtron ENHANCED®</i>	22
Figura 2 – Instrução <i>add</i>	27
Figura 3 – Instrução <i>sub</i>	27
Figura 4 – Instrução <i>and</i>	28
Figura 5 – Instrução <i>or</i>	28
Figura 6 – Instrução <i>xor</i>	28
Figura 7 – Instrução <i>slt</i>	29
Figura 8 – Instrução <i>mul</i>	29
Figura 9 – Instrução <i>div</i>	29
Figura 10 – Instrução <i>mod</i>	29
Figura 11 – Instrução <i>beq</i>	30
Figura 12 – Instrução <i>bne</i>	30
Figura 13 – Instrução <i>addi</i>	30
Figura 14 – Instrução <i>subi</i>	31
Figura 15 – Instrução <i>inc</i>	31
Figura 16 – Instrução <i>dec</i>	31
Figura 17 – Instrução <i>lw</i>	31
Figura 18 – Instrução <i>sw</i>	32
Figura 19 – Instrução <i>not</i>	32
Figura 20 – Instrução <i>lwi</i>	32
Figura 21 – Instrução <i>in</i>	32
Figura 22 – Instrução <i>out</i>	33
Figura 23 – Instrução <i>j</i>	33
Figura 24 – Instrução <i>hlt</i>	34

Lista de tabelas

Tabela 1 – Instruções do <i>bbtron ENHANCED[®]</i>	19
Tabela 2 – Formato de Instruções do tipo R	20
Tabela 3 – Formato de Instruções do tipo I	20
Tabela 4 – Formato de Instruções do tipo K	20
Tabela 5 – Formato de Instruções do tipo J	21
Tabela 6 – Sinais de Controle do <i>bbtron ENHANCED[®]</i>	26

Sumário

1	INTRODUÇÃO	9
2	OBJETIVOS	11
2.1	Geral	11
2.2	Específico	11
3	FUNDAMENTAÇÃO TEÓRICA	13
3.1	Processador	13
3.2	Conjunto de Instruções	13
3.3	RISC	13
3.4	CISC	14
3.5	Modos de Endereçamento	14
3.6	Unidade de Processamento	14
3.6.1	Contador de Programa	14
3.6.2	Memória de Instrução	15
3.6.3	Banco de Registradores	15
3.6.4	Unidade Lógica e Aritmética	15
3.6.5	Memória de Dados	16
3.6.6	Multiplexador	16
3.6.7	Extensor de Sinal	16
3.7	Linguagem de descrição de <i>Hardware</i>	16
3.7.1	Verilog	17
3.8	Intel® Quartus Prime	17
4	DESENVOLVIMENTO	19
4.1	Conjunto de Instruções	19
4.1.1	Instruções do tipo R	19
4.1.2	Instruções do tipo I	20
4.1.3	Instruções do tipo K	20
4.1.4	Instruções do tipo J	20
4.2	Modos de Endereçamento	21
4.2.1	Enderecamento Direto	21
4.2.2	Enderecamento por Registrador	21
4.2.3	Enderecamento por Imediato	21
4.3	Caminho de Dados	21
4.4	Unidade Central de Processamento	22

4.4.1	Contador de Programa	22
4.4.2	Memória de Instrução	23
4.4.3	Banco de Registradores	23
4.4.4	Unidade Lógica e Aritmética	23
4.4.5	Memória de Dados	24
4.4.6	Extensor de Sinal	24
4.4.7	Multiplexadores e Operações lógicas	24
4.4.8	Módulo de Entrada	25
4.4.9	Módulo de Saída	25
4.5	Unidade de Controle	25
4.6	Interligação dos módulos	25
5	RESULTADOS OBTIDOS E DISCUSSÕES	27
5.1	Teste da Instruções do tipo R	27
5.1.1	Instrução <i>add</i>	27
5.1.2	Instrução <i>sub</i>	27
5.1.3	Instrução <i>and</i>	27
5.1.4	Instrução <i>or</i>	28
5.1.5	Instrução <i>xor</i>	28
5.1.6	Instrução <i>slt</i>	28
5.1.7	Instrução <i>mul</i>	29
5.1.8	Instrução <i>div</i>	29
5.1.9	Instrução <i>mod</i>	29
5.2	Teste Instrução do tipo I	30
5.2.1	Instrução <i>beq</i>	30
5.2.2	Instrução <i>bne</i>	30
5.2.3	Instrução <i>addi</i>	30
5.2.4	Instrução <i>subi</i>	30
5.2.5	Instrução <i>inc</i>	31
5.2.6	Instrução <i>dec</i>	31
5.2.7	Instrução <i>lw</i>	31
5.2.8	Instrução <i>sw</i>	31
5.2.9	Instrução <i>not</i>	32
5.2.10	Instrução <i>lwi</i>	32
5.2.11	Instrução <i>in</i>	32
5.3	Teste Instrução do tipo K	33
5.3.1	Instrução <i>out</i>	33
5.4	Teste Instrução do tipo J	33
5.4.1	Instrução <i>j</i>	33
5.4.2	Instrução <i>hlt</i>	33

5.5	Teste na <i>Altera's Cyclone® IV FPGA</i>	34
5.5.1	Teste do Algoritmo A	34
5.5.2	Teste do Algoritmo B	39
6	CONCLUSÃO	41
7	CONSIDERAÇÕES FINAIS	43
 REFERÊNCIAS		45

 APÊNDICES		47
APÊNDICE A – ALU.V	49	
APÊNDICE B – AND.V	51	
APÊNDICE C – REGISTERBENCH.V	53	
APÊNDICE D – CONTROLUNITY.V	55	
APÊNDICE E – DATAMEMORY.V	63	
APÊNDICE F – INSTRUCTIONMEMORY.V	65	
APÊNDICE G – MUXALUSCR.V	67	
APÊNDICE H – MUXIN SIGNAL.V	69	
APÊNDICE I – MUXJUMP.V	71	
APÊNDICE J – MUXMEMTOREG.V	73	
APÊNDICE K – MUXPCSCR.V	75	
APÊNDICE L – MUXREGDEST.V	77	
APÊNDICE M – PROGRAMCOUNTER.V	79	
APÊNDICE N – SIGNEXTENDERJ.V	81	
APÊNDICE O – SIGNEXTENDERR.V	83	
APÊNDICE P – BBTRONENHANCEDCPU.V	85	

APÊNDICE Q – BINTOBCD2.V	89
APÊNDICE R – OUTMODULE.V	91
APÊNDICE S – DISPLAY.V	93
 ANEXOS	 95
ANEXO A – ANEXO 1	97
ANEXO B – ANEXO 2	99

1 Introdução

Com o desenvolvimento de novas tecnologia nossa vida cotidiana se vê cada vez mais dependente do aparelho celular que é nada mais e nada menos que um computador de bolso. O computador que antes era composto por uma sala com inúmeros gabinetes agora esta em todos os lugares, celular, aparelho de som, *tablet* etc. Tudo isso só é possível graças ao avanços tecnológicos e diretamente ao processador das unidades computacionais. Na unidade curricular “Arquitetura e Organização de Computadores” aprendemos como a unidade computacional funciona e agora no laboratório iremos certificar o funcionamento na prática. Com base nos fundamentos adquiridos em “Arquitetura e Organização de Computadores” e na unidade curricular "Circuitos Digitais", iremos agora realizar o desenvolvimento de Unidade Central de Processamento. Os circuitos computacionais se encontram cada vez mais complexos para desenvolvimento, logo, foram desenvolvidas linguagens de descrição de Hardware que nos permitem desenvolver os sistemas computacionais de forma rápida e prática com um fim didático. Temos como exemplo da linguagem que será usada para o desenvolvimento da unidade de processamento, o Verilog. Com o Verilog, podemos transformar todos os circuitos aprendidos na Unidade Curricular "Circuitos Digitais", em linhas de código, minimizando assim, o tempo de desenvolvimento. Por fim, com todos os módulos presente na Unidade Central de Processamento elaborados é realizado a conexão entre eles e também o teste de funcionamento na placa *Altera's Cyclone® IV FPGA* para assim concluir o projeto.

2 Objetivos

2.1 Geral

Esse relatório tem como objetivo geral relatar o desenvolvimento de uma unidade de processamento computacional em lógica programável.

2.2 Específico

O projeto tem como objetivo específico o desenvolvimento de cada uma das unidades presentes na unidade de processamento utilizando a linguagem de descrição de *Hardware Verilog*. Além da construção, deve também possuir os testes das unidades interligadas. Por fim, concluir o desenvolvimento da unidade de processamento e demonstrar o funcionamento na placa *Altera's Cyclone® IV FPGA* para assim concluir o projeto.

3 Fundamentação Teórica

3.1 Processador

O microprocessador, geralmente chamado apenas de processador, é um circuito integrado que realiza as funções de cálculo e tomada de decisão de um computador. Todos os computadores e equipamentos eletrônicos baseiam-se nele para executar suas funções, podemos dizer que o processador é o cérebro do computador por realizar todas estas funções. Um microprocessador incorpora as funções de uma unidade central de computação (UCP) em um único circuito integrado, ou no máximo alguns circuitos integrados. É um dispositivo multifuncional programável que aceita dados digitais como entrada, processa de acordo com as instruções armazenadas em sua memória, e fornece resultados como saída. Microprocessadores operam com números e símbolos representados no sistema binário. (1)

3.2 Conjunto de Instruções

Conjunto de instruções (tradução de instruction set) são as operações que um processador, microprocessador, microcontrolador, CPU ou outros periféricos programáveis suporta, fornece ou disponibiliza para o programador, ou seja, é a representação em mnemônicos do código de máquina, com a finalidade de facilitar o acesso ao componente. Cada componente possui o seu próprio conjunto de instruções, que é fornecido pelo fabricante, que também costuma fornecer ou disponibilizar um montador assembly, que transforma o conjunto de instruções em código de máquina para ser utilizado pelo componente. No caso dos processadores, quando o conjunto de instruções for reduzido leva-o a ter o nome de RISC e se forem complexas o nome de CISC. (2)

3.3 RISC

A arquitetura RISC é constituída por um pequeno conjunto de instruções simples que são executadas diretamente pelo hardware, onde não há a intervenção de um interpretador (microcódigo), o que significa que as instruções são executadas em apenas uma microinstrução (de uma única forma e seguindo um mesmo padrão). As máquinas RISC só se tornaram viáveis devido aos avanços de software otimizado para essa arquitetura, através da utilização de compiladores otimizados e que compensem a simplicidade dessa arquitetura. (3)

3.4 CISC

CISC é uma linha de arquitetura de processadores capaz de executar centenas de instruções complexas diferentes sendo, assim, extremamente versátil. 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 vantagem da arquitetura CISC é a diminuição do trabalho do compilador ao traduzir uma sentença de alto nível para linguagem de máquina. (4)

3.5 Modos de Endereçamento

Para que a unidade de processamento realize cálculos e operações com dados, é necessário que haja uma organização no armazenamento de dados para que o processador possa encontrar o dado especificado na memória ou nos registradores. Com esse intuito, foram desenvolvidos os modos de endereçamento, que são responsáveis identificação e organização dos dados. O campo ou os campos de endereço num formato de instrução típico são relativamente pequenos. Nós gostaríamos de poder referenciar um grande intervalo de posições da memória principal ou, em alguns sistemas, da memória virtual. Para alcançar esse objetivo, uma grande variedade de técnicas de endereçamento foi desenvolvida. Todas envolvem algum tipo de troca entre intervalo de endereços e/ou flexibilidade de endereçamento por um lado e o número de referências de memória dentro da instrução e/ou a complexidade de cálculo de endereços por outro. (??)

3.6 Unidade de Processamento

A unidade central de processamento ou CPU (*Central Processing Unit*), também conhecido como processador, é a parte de um sistema computacional, que realiza as instruções de um programa de computador, para executar a aritmética básica, lógica, e a entrada e saída de dados. A CPU tem papel parecido ao cérebro no computador. O termo vem sendo usado desde o início de 1960. A forma, desenho e implementação mudaram drasticamente desde os primeiros exemplos, porém o seu funcionamento fundamental permanece o mesmo. (5)

3.6.1 Contador de Programa

Contador de programa é um registrador de uma Unidade Central de Processamento que indica qual é a posição atual na sequência de execução de um processo. Dependendo dos detalhes da arquitetura, ele armazena o endereço da instrução sendo executada ou o endereço da próxima instrução. O contador de programa é automaticamente incrementado para cada ciclo de instrução de forma que as instruções são normalmente executadas

sequencialmente a partir da memória, sendo que o contador de programa deve ser colocado a zero no inicio da execução do mesmo. (6)

3.6.2 Memória de Instrução

Memória de instrução é uma memória presente na unidade processamento que possui a função de armazenar as instruções que vão ser executadas pela unidade de processamento. A instrução é inicialmente acessada na memória e transferida para o interior da CPU, mais especificamente num registrador especial da unidade de controle chamado de RI ("Registrador de Instrução"). Uma vez no RI, a instrução é interpretada por um circuito decodificador. (7)

3.6.3 Banco de Registradores

O registrador (português brasileiro) de uma CPU (unidade central de processamento) é a memoria RAM que armazena n bits. Os registradores estão no topo da hierarquia de memória, sendo assim, são o meio mais rápido e caro de se armazenar um dado. Lembrando que os registradores são circuitos digitais capazes de armazenar e deslocar informações binárias, e são tipicamente usados como um dispositivo de armazenamento temporário. São utilizados na execução de programas de computadores, disponibilizando um local para armazenar dados. Na maioria dos computadores modernos, quando da execução das instruções de um programa, os dados são movidos da memória principal para os registradores. Então, as instruções que utilizam estes dados são executadas pelo processador e, finalmente, os dados são movidos de volta para a memória principal. É uma tecnologia com custo elevado. (8)

3.6.4 Unidade Lógica e Aritmética

A unidade lógica e aritmética (ULA) ou em inglês *Arithmetic Logic Unit* (ALU) é um circuito digital que realiza operações lógicas e aritméticas. A ULA é uma peça fundamental da unidade central de processamento (CPU), e até dos mais simples microprocessadores. É na verdade, uma "grande calculadora eletrônica" do tipo desenvolvido durante a II Guerra Mundial, e sua tecnologia já estava disponível quando os primeiros computadores modernos foram construídos. A ULA executa as principais operações lógicas e aritméticas do computador. Ela soma, subtrai, divide, determina se um número é positivo ou negativo ou se é zero. Além de executar funções aritméticas, uma ULA deve ser capaz de determinar se uma quantidade é menor ou maior que outra e quando quantidades são iguais. A ULA pode executar funções lógicas com letras e com números. (9)

3.6.5 Memória de Dados

Memória de dados é a memória responsável por armazenar os dados que vão ser manipulados pela CPU. Possui uma acesso mais lento do que o banco de registradores, mas de capacidade bem maior. Armazena grande conjunto de dados que os registradores não suportam. São mais usadas que os registradores pois seu custo-benefício é superior devido o alto valor dos registradores.

3.6.6 Multiplexador

Um multiplexador, multiplexer, mux ou multiplex é um dispositivo que seleciona as informações de duas ou mais fontes de dados num único canal. São utilizados em situações onde o custo de implementação de canais separados para cada fonte de dados é maior que o custo e a inconveniência de utilizar as funções de multiplexação/demultiplexação. Numa analogia física, consideremos o comportamento de viajantes que atravessam uma ponte com largura pequena, para atravessarem, os veículos executarão curvas para que todos passem em fila pela ponte. Ao atingir o fim da ponte eles separaram-se em rotas distintas rumo a seus destinos. (10)

3.6.7 Extensor de Sinal

Extensor de sinal é um módulo da unidade de processamento que é responsável pelo aumento o numero de bits de uma informação recebida pelo módulo. Temos como exemplo um conjunto de 5 bits representando um número inteiro que desejamos transformar no mesmo valor, mas representado com 8 bits. Para que isso aconteça, devemos colocar zeros a esquerda e assim aumentar o tamanho em bits mas mantendo um novo tamanho de informação.

3.7 Linguagem de descrição de *Hardware*

Em eletrônica, uma linguagem de descrição de hardware ou LDH é qualquer linguagem de uma classe de linguagens de computador, linguagem de especificação ou linguagem de modelagem para uma descrição formal e design de circuitos eletrônicos, e mais comumente, a lógica digital. Pode descrever o funcionamento do circuito, a sua concepção e organização, e ainda testá-lo para verificar seu funcionamento por meio de simulação. LDHs são padrões de expressões baseados em texto, da estrutura espacial, temporal e comportamental dos sistemas eletrônicos. Como outras linguagens de programação, LDHs incluem anotações explícitas para expressar a simultaneidade bem como sintaxe e semântica próprias. (11)

3.7.1 Verilog

Verilog, cuja padronização atual é a IEEE 1364-2005, é uma linguagem de descrição de hardware (*Hardware Description Language - HDL*) usada para modelar sistemas eletrônicos ao nível de circuito. Essa ferramenta suporta o projeto, verificação e implementação de projetos analógicos, digitais e híbridos em vários níveis de abstração. Um dos principais atributos da modelagem de circuitos por linguagem descritiva frente à modelagem por captura de esquemático, é que desta maneira o projeto se torna independente da plataforma de desenvolvimento (IDE) na qual se está trabalhando. Além disso, adotando-se as boas práticas na descrição dos circuitos, o compilador é inclusive capaz de contornar a ausência de determinado recurso na tecnologia onde o circuito será sintetizado, conferindo uma portabilidade desse modelo para qualquer dispositivo (*target*) onde pode ser sintetizado. (12)

3.8 Intel® Quartus Prime

É o *software* responsável pela compilação dos códigos em Verilog e o teste do mesmo. Agora adquirido pela Intel, mas antigamente da Altera Corporation que é uma empresa fabricante de dispositivos lógicos programáveis. Faz parte do grupo de ações de tecnologia NASDAQ-100 e do S&P 500. Seus principais produtos são as FPGAs Stratix, Arria e Cyclone e o software de criação Quartus II. O hardware da Altera é programado em VHDL, Verilog, ou ainda em uma linguagem própria AHDL. (13)

4 Desenvolvimento

4.1 Conjunto de Instruções

Com base nas necessidades expostas pelo professor, foi desenvolvido um conjunto de instrução que deve ser capaz de executar os algoritmos especificado no início do projeto. Com esse intuito, o conjunto de instrução foi elaborado como na tabela 1 e dividido em 4 categorias que serão citadas a seguir.

Tabela 1 – Instruções do *bbtron ENHANCED®*

Instrução	OPcode	Sigla	Tipo	Operação
<i>Add</i>	000000	add	R	$R[DR] \leftarrow R[SA] + R[SB]$
<i>Subtract</i>	000001	sub	R	$R[DR] \leftarrow R[SA] - R[SB]$
<i>AND</i>	000010	and	R	$R[DR] \leftarrow R[SA] \wedge R[SB]$
<i>OR</i>	000011	or	R	$R[DR] \leftarrow R[SA] \vee R[SB]$
<i>XOR</i>	000100	xor	R	$R[DR] \leftarrow R[SA] \oplus R[SB]$
<i>Set on less than</i>	000101	slt	R	Se $R[SA] < R[SB]$, então $R[DR] = 1$
<i>Multiply</i>	000110	mul	R	$R[DR] \leftarrow R[SA] * R[SB]$
<i>Devide</i>	000111	div	R	$R[DR] \leftarrow R[SA] / R[SB]$
<i>Rest</i>	001000	mod	R	$R[DR] \leftarrow R[SA] \% R[SB]$
<i>Branch on Equal</i>	001001	beq	I	Se $R[SA] = R[SB]$ então $PC \leftarrow IM$. Se não, $PC \leftarrow PC + 1$
<i>Branch on Not Equal</i>	001010	bne	I	Se $R[SA] \neq R[SB]$ então $PC \leftarrow IM$. Se não, $PC \leftarrow PC + 1$
<i>Add Immediate</i>	001011	addi	I	$R[DR] \leftarrow R[SA] + IM$
<i>Sub Immediate</i>	001100	subi	I	$R[DR] \leftarrow R[SA] - IM$
<i>Incremmment</i>	001101	inc	I	$R[DR] \leftarrow R[SA] + 1$
<i>Decremment</i>	001110	dec	I	$R[DR] \leftarrow R[SA] - 1$
<i>Load Word</i>	001111	lw	I	$R[DR] \leftarrow M[IM + R[0]]$
<i>Store Word</i>	010000	sw	I	$M[IM + R[0]] \leftarrow R[DR]$
<i>Not</i>	010001	not	I	$R[DR] \leftarrow !R[SA]$
<i>Load Word Immediate</i>	010100	lwi	I	$R[DR] \leftarrow IM + R[0]$
<i>In</i>	010101	in	I	$R[DR] \leftarrow Switches + R[0]$
<i>Out</i>	010110	out	K	$Displays \leftarrow R[DR]$
<i>Jump</i>	010111	j	J	$PC \leftarrow IM$
<i>Stop Operation</i>	011000	hlt	J	<i>No operation</i>

Fonte: O autor

4.1.1 Instruções do tipo R

As instruções do tipo R são formadas por 3 registradores e são responsáveis pela maioria das operações aritméticas. A instrução possui o *Opcode* da instrução, 2 registradores de operação ($R[SA]$ e $R[SB]$) e 1 registrador de destino ($R[DR]$), tendo assim um espaço vazio (*Blank*) na instrução de 11 bits como podemos ver na tabela 2.

Tabela 2 – Formato de Instruções do tipo R

Tamanho (bits)	6	5	5	5	11
Campo	Opcode	R[SA]	R[SB]	R[DR]	Blank
Bits	31 - 26	25 - 21	20 - 16	15 - 11	10 - 0

Fonte: O autor

4.1.2 Instruções do tipo I

As instruções do tipo I são formadas por 2 registradores e são responsáveis pelo deslocamento de bits, somas de imediatos, negação etc. A instrução possui o *Opcode* da instrução, 1 registrador de operação (R[SA]), 1 registrador de destino (R[DR]) e uma área de 16 bits que é utilizada para armazenar o imediato, endereços ou o número de bits a serem deslocados, como podemos ver na tabela 3.

Tabela 3 – Formato de Instruções do tipo I

Tamanho (bits)	6	5	5	16
Campo	Opcode	R[SA]	R[DR]	Imediato/Endereço/Shamt
Bits	31 - 26	25 - 21	20 - 16	15 - 0

Fonte: O autor

4.1.3 Instruções do tipo K

As instruções do tipo K são formadas por 1 registrador e é responsável pela exibição dos dados no *display* de sete segmentos. A instrução possui o Opcode da instrução, 1 registrador (R[DR]) de destino e uma área de 21 bits que é utilizada para armazenar o imediato como podemos ver na tabela 4.

Tabela 4 – Formato de Instruções do tipo K

Tamanho (bits)	6	5	21
Campo	Opcode	R[DR]	Imediato/Endereço/Shamt
Bits	31 - 26	25 - 21	20 - 0

Fonte: O autor

4.1.4 Instruções do tipo J

As instruções do tipo K são formadas por nenhum registrador e são responsáveis pelo saltos e pela parada de operação da UCP. A instrução possui o Opcode da instrução e uma área de 26 bits que é utilizada para armazenar o imediato ou apenas continuar vazia, como podemos ver na tabela 5.

Tabela 5 – Formato de Instruções do tipo J

Tamanho (bits)	6	25
Campo	Opcode	Imediato/Endereço/Shamt
Bits	31 - 26	25 - 0

Fonte: O autor

4.2 Modos de Endereçamento

Como observado nos Conjunto de Instruções, usaremos 3 tipos de endereçamento conforme a necessidade. Dentre eles temos: Endereçamento Direto, Endereçamento por Registrador, Endereçamento por Imediato.

4.2.1 Endereçamento Direto

A unidade de processamento buscando as informações direto na memória. Temos como exemplo as instruções de *Load* e *Store*.

4.2.2 Endereçamento por Registrador

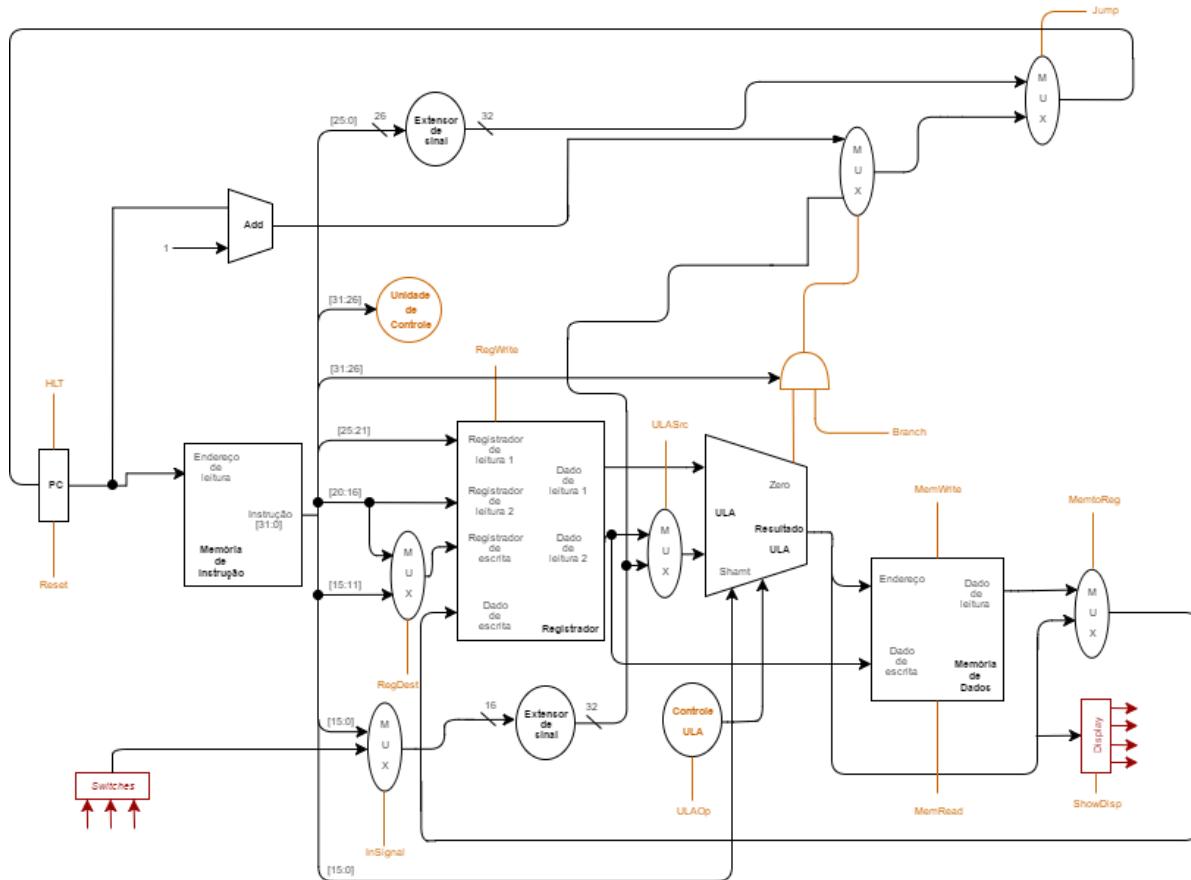
A unidade de processamento buscando e realizando operações com as informações direto do registrador. Temos como exemplo as instruções de operação aritmética *Add*, *Subtract*.

4.2.3 Endereçamento por Imediato

A unidade de processamento buscando e realizando operações com as informações da instrução, a partir do Imediato. Temos como exemplo as instruções de operação aritmética *Add Immediate*, *Subtract Immediate*.

4.3 Caminho de Dados

Com as instruções e o modo de endereçamento especificado, foi desenvolvido o esquemático do processador, mostrado na [Figura 1](#).

Figura 1 – Esquemático *bbtron ENHANCED®*

Fonte: O Autor

4.4 Unidade Central de Processamento

Com o caminho de dados elaborado, é necessário agora realizar o desenvolvimento de cada uma das unidades do esquemático na linguagem Verilog. Ao decorrer dessa seção, iremos citar os sinais de controle que serão listados da próxima seção.

4.4.1 Contador de Programa

O contador de programa é nada mais, nada menos que um registrador que é responsável por guardar o endereço da instrução a ser executada. Por padrão, o contador de programa guarda seu valor e é incrementado a cada iteração, indo assim, para a próxima instrução. Algumas instruções como o *Jump* e os *Branches* vão direto pra instruções que não estão em ordem na memória de instrução, necessitando assim um tratamento especial. As instruções *Jump*, *Branch Equals* e *Branch Not Equals* são acionadas através dos sinais de controle *Jump* e *Branch*, realizando assim, a alteração do contador de programa. Caso o sinal de controle *Jump* estiver acionado, o contador de programa irá direto para a instrução indicada. Quando o sinal de controle *Branch* e o *output* da ULA denominado *Zero* são

acionados, o valor informado na instrução é enviado para o contador de programa. O sinal de controle *HLT* coloca o sistema em um estado de espera, sem modificar o PC. O papel do sinal de controle *Reset* é atribuir zero ao valor do contador de programa. O módulo possui como parâmetros de entrada *inAddy* (16 bits), *hlt* (1 bit), *clock* (1 bit) e *reset* (1 bit). Como parâmetro de saída, apenas o *outAddy* (16 bits). O código em Verilog do contador de programa pode ser encontrado no [Apêndice M](#).

4.4.2 Memória de Instrução

Como especificado na fundamentação teórica, a memória de instrução é o módulo da unidade de processamento onde são armazenados as instruções a serem executadas. É composta por uma 1024 registradores de 32 bits cada um. Sincronizada com o *clock*, fornecendo ao *output* a instrução a ser executada. O módulo possui como parâmetros de entrada o *clock* (1 bit) e o *addy* (10 bits). Como parâmetro de saída apenas o *RAMOutput* (32 bits). O código em Verilog da memória de instrução pode ser encontrado no [Apêndice F](#).

4.4.3 Banco de Registradores

Como na arquitetura MIPS, o banco de registradores possui 32 registradores de 32 bits cada um. O banco de registradores possui *inputs* que são 3 endereços de registradores de 5 bits cada (*readAddy1*, *readAddy2* e *writeAddy*) que referenciam os registradores que iram ser utilizados. Dois endereços de leitura e um endereço para escrita. O banco de registradores pode receber também um dado de 32 bits *writeData* que pode ser a informação que vai ser escrita no banco. Sem se deferir as outras unidades da unidade de processamento, o banco de registradores também possui sinais de controle como *input* para controlar seu funcionamento. No caso temos o sinal de controle *writeReg* que é responsável por dizer se os dados de escrita devem ser escritos no registrador informado no *writeAddy*. Por fim, o banco de registrador recebe como entrada também, o *clock*. Como parâmetro de saída os dados *data1*, *data2* e *data3* originados respectivamente dos registradores de endereço *readAddy1*, *readAddy2* e *writeAddy*. O código do banco de registradores pode ser encontrado no [Apêndice C](#).

4.4.4 Unidade Lógica e Aritmética

A unidade lógica e aritmética é responsável por realizar as operações com os operandos *data1* e *data2*, como soma e subtração. Com os dados inseridos na ALU é necessário agora selecionar a operação que deve ser executada por ele, essa tarefa fica como responsabilidade da unidade de controle que será discutida na próxima seção, enviando um sinal de controle *aluOp* que irá informar a ALU a operação lógica ou aritmética a ser executada. A ULA pode receber também, o *input shamt* que dita o número de bits que deve ser deslocado durante a instrução *Shift*. No módulo da ULA, uma *flag* pode ser

acionada no componente conforme o resultado das operações, a *zero* quando resultado da operação é igual a zero. Por fim, temos o *output aluOut* que consiste no resultado da operação. O código da unidade lógica aritmética pode ser encontrado no [Apêndice A](#).

4.4.5 Memória de Dados

A memória de dados é formada por 1024 registradores de 32 bits cada. Temos como *inputs* o endereço de memória que irá ser consultado denominado *memoryAddy* (10 bits), o dado a ser escrito denominado *writeData* (32 bits), o *clock* e os dois sinais de controle *writeEnable* (1 bit) e *readEnable* (1 bit) que são responsáveis por dizer à memória de dados se na instrução indicada, iremos realizar uma operação de escrita ou leitura na memória, respectivamente. dado a ser informado, endereço de escrita ou endereço de leitura. A memória de dados possui também o *output dataRAMOutput* que é responsável por informar os dados lidos da memória. O processo de leitura e escrita acontece da seguinte forma: uma variável com o endereço do registrador é separada para armazenar o endereço de leitura na memória. Caso o sinal de controle *writeEnable* que é originado pela unidade de controle esteja ativado, escreve-se o dado de *input* no endereço especificado. Caso o sinal de controle *readEnable* que é originado pela unidade de controle esteja ativado, o dado escrito no endereço de memória *memoryAddy* é enviado a partir do *dataRAMOutput*. O código da memória de dados pode ser encontrado no [Apêndice E](#).

4.4.6 Extensor de Sinal

O *bbtron ENHANCED®* possui dois extensores de sinal que tem como objetivo passar um dado que 16 ou 26 bits para 32 bits. Os módulos possuem como parâmetro de entrada o *inputA* (16 bits) para o *signExtenderR* e o *inputB* (26 bits) para o *signExtenderJ*. Como saída, *extenderOutputA* (32 bits) e *extenderOutputB* (32 bits) respectivamente. O código dos extensores de memória pode ser encontrado no [Apêndice O](#) e no [Apêndice N](#).

4.4.7 Multiplexadores e Operações lógicas

Como podemos observar na [Figura 1](#), possuímos inúmeros componentes denominados "MUX" que são nada mais, nada menos, que os multiplexadores. Com base no valor do sinal de controle recebido como *input* no módulo do multiplexador, um sinal de entrada vai ser passado como saída. Os códigos dos multiplexadores de memória podem ser encontrados no [Apêndice G](#), [Apêndice H](#), [Apêndice I](#), [Apêndice J](#), [Apêndice K](#) e [Apêndice L](#). Além dos multiplexadores, possuímos uma porta lógica "AND" que faz a verificação se dever existir ou não a execução do *branch*. O código da porta lógica pode ser encontrado no [Apêndice B](#).

4.4.8 Módulo de Entrada

Com base nas necessidades estabelecidas no início do projeto, foi necessário a implementação de um módulo que envia dados para o registrador ou memória de dados a partir da necessidade do usuário. O solução encontrada foi o envio dos sinais originados dos 16 *switches* da *Altera's Cyclone® IV FPGA*. Com 16 *switches* temos a representação de 16 bits. Os sinais dos *switches* são enviados da seguinte maneira: quando a instrução *in* é acionada enviamos para o multiplexador encontrado no [Apêndice H](#). O sinal de controle *inSignal* é então o responsável por selecionar os dados originados dos *switches* ao invés dos bits [15:0] originados pela instrução. O sinal dos *switches* é enviado como imediato das instruções que utilizam o mesmo.

4.4.9 Módulo de Saída

O desafio agora, foi implementar o módulo de saída, mostrando a partir dos *displays* de sete segmentos disponíveis na *Altera's Cyclone® IV FPGA* o valor do registrador indicado na instrução. Na instrução *out* enviamos o endereço do registrador que possui o dado que deve ser mostrado no *display*. Passamos então o dado do banco de registradores para a ULA que em seguida passa o valor para o módulo de saída. Como o valor em binário utilizamos do código de conversão de binário para *BCD* (disponível no [Apêndice Q](#)) para podermos ter as casas da unidade, dezena, centena e o sinal de negativo (que é ativado quando número enviado é negativo). Por fim, enviamos os dados originados para os displays a partir de um *case* que faz o tratamento de qual *led* do display deve ser acesso. Os códigos utilizados na elaboração do módulo de saída estão disponíveis no [Apêndice Q](#) (BinToBCD), [Apêndice R](#) (outModule) e [Apêndice S](#) (Display).

4.5 Unidade de Controle

Como citado na fundamentação teórica, a unidade de controle é responsável por coordenar as ações da UCP. Para a realização de tal feito, foram elaborados os sinais de controle demonstrados na [Tabela 6](#). O código da unidade de controle pode ser encontrado no [Apêndice D](#).

4.6 Interligação dos módulos

Por fim, foi executada a interligação dos módulos no arquivo presente no [Apêndice P](#).

Tabela 6 – Sinais de Controle do *bbtron ENHANCED®*

Sinal de Controle	Ação
writeReg	Indica se a UCP deve escrever no banco de registradores.
regDest	Indica se o endereço de escrita do registrador é o segundo ou o terceiro na instrução.
memtoReg	Indica se devemos escrever o dado na memória no banco de registradores ou não.
Jump	Indica se temos uma instrução de jump.
inSignal	Indica de recebemos o imediato vindo da instrução ou do módulo de entrada(<i>switches</i>).
aluScr	Indica se um dos dados de entrada da ULA é originado do imediato ou do banco de registradores.
writeEnable	Indica se devemos escrever na memória.
readEnable	Indica se devemos ler da memória.
Branch	Indica se devemos fazer o branch.
aluOp	Indica a ULA qual operação ela deve executar.
hlt	Indica ao contador de programa se o processamento deve ser interrompido.
reset	Indica ao contador de programa se ele deve voltar pro endereço de instrução 0.
showDisplay	Indica se devemos mostrar o dado do registrador escolhido no módulo de saída.

Fonte: O autor

5 Resultados Obtidos e Discussões

Após o desenvolvimento de cada uma das unidades em Verilog, as mesmas são interligadas e um teste para cada tipo de instrução é realizado.

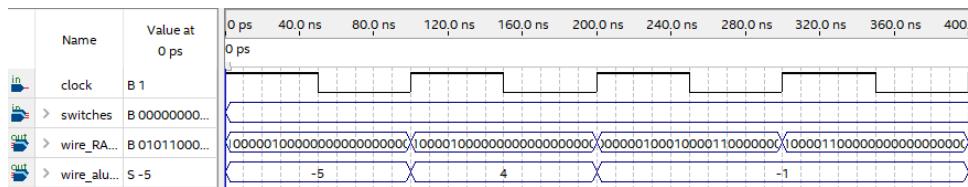
5.1 Teste da Instruções do tipo R

Iremos nessa seção realizar o teste de todas as *Waveforms* geradas a partir das instruções do tipo R no *software* Intel® Quartus Prime.

5.1.1 Instrução *add*

Na [Figura 2](#) podemos certificar o funcionamento da instrução *add*.

Figura 2 – Instrução *add*

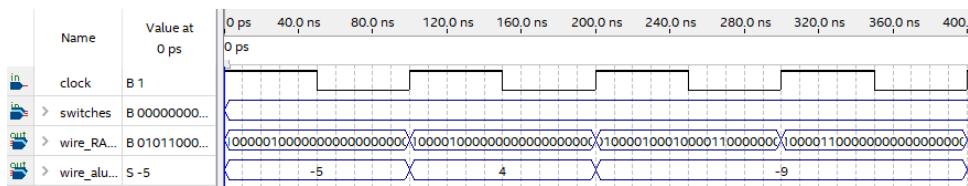


Fonte: O Autor

5.1.2 Instrução *sub*

Na [Figura 3](#) podemos certificar o funcionamento da instrução *sub*.

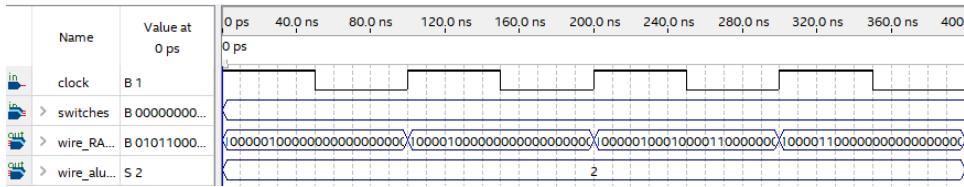
Figura 3 – Instrução *sub*



Fonte: O Autor

5.1.3 Instrução *and*

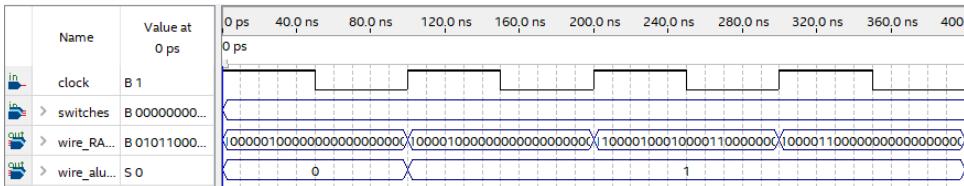
Na [Figura 4](#) podemos certificar o funcionamento da instrução *and*.

Figura 4 – Instrução *and*

Fonte: O Autor

5.1.4 Instrução *or*

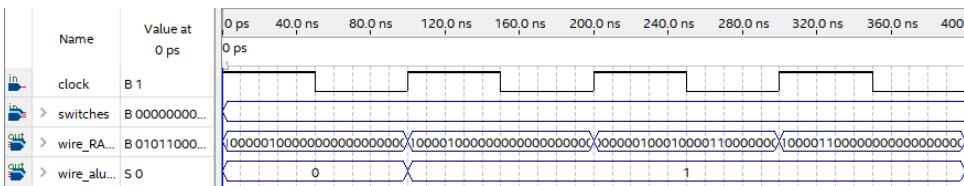
Na Figura 5 podemos certificar o funcionamento da instrução *or*.

Figura 5 – Instrução *or*

Fonte: O Autor

5.1.5 Instrução *xor*

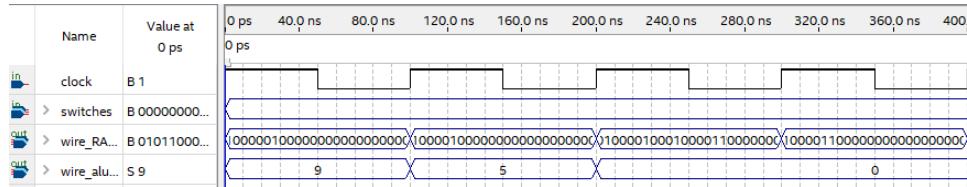
Na Figura 6 podemos certificar o funcionamento da instrução *xor*.

Figura 6 – Instrução *xor*

Fonte: O Autor

5.1.6 Instrução *slt*

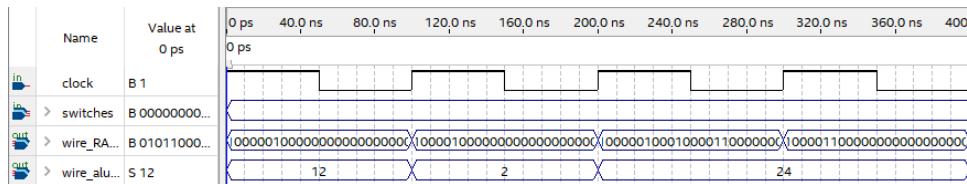
Na Figura 7 podemos certificar o funcionamento da instrução *slt*.

Figura 7 – Instrução *slt*

Fonte: O Autor

5.1.7 Instrução *mul*

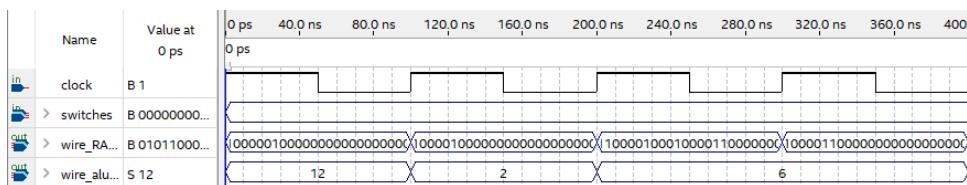
Na [Figura 8](#) podemos certificar o funcionamento da instrução *mul*.

Figura 8 – Instrução *mul*

Fonte: O Autor

5.1.8 Instrução *div*

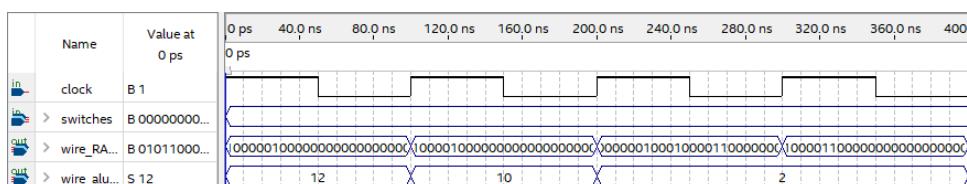
Na [Figura 9](#) podemos certificar o funcionamento da instrução *div*.

Figura 9 – Instrução *div*

Fonte: O Autor

5.1.9 Instrução *mod*

Na [Figura 10](#) podemos certificar o funcionamento da instrução *mod*.

Figura 10 – Instrução *mod*

Fonte: O Autor

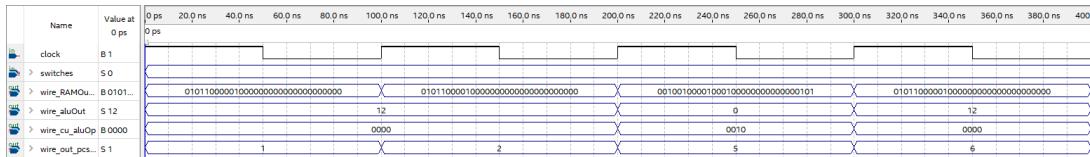
5.2 Teste Instrução do tipo I

Iremos nessa seção realizar o teste de todas as *Waveforms* geradas a partir das instruções do tipo I no *software* Intel® Quartus Prime.

5.2.1 Instrução *beq*

Na [Figura 11](#) podemos certificar o funcionamento da instrução *beq*.

Figura 11 – Instrução *beq*

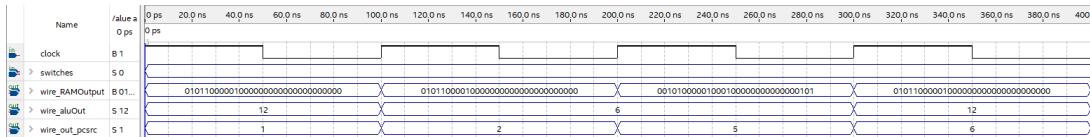


Fonte: O Autor

5.2.2 Instrução *bne*

Na [Figura 12](#) podemos certificar o funcionamento da instrução *bne*.

Figura 12 – Instrução *bne*

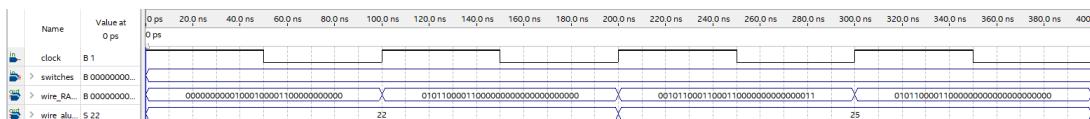


Fonte: O Autor

5.2.3 Instrução *addi*

Na [Figura 13](#) podemos certificar o funcionamento da instrução *addi*.

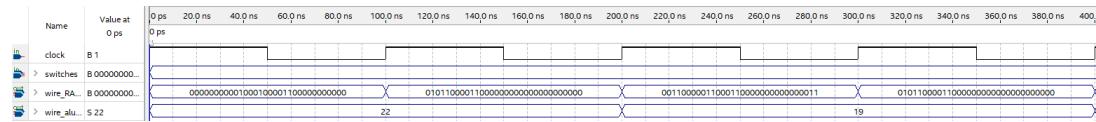
Figura 13 – Instrução *addi*



Fonte: O Autor

5.2.4 Instrução *subi*

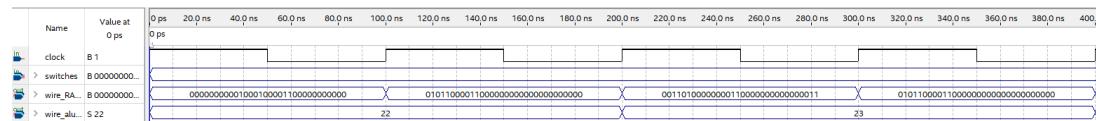
Na [Figura 14](#) podemos certificar o funcionamento da instrução *subi*.

Figura 14 – Instrução *subi*

Fonte: O Autor

5.2.5 Instrução *inc*

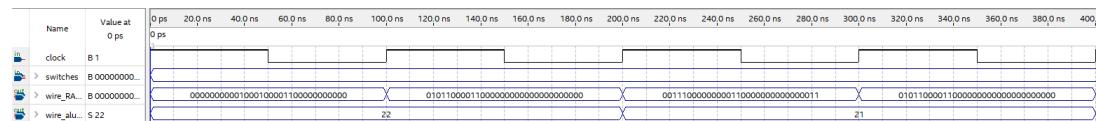
Na Figura 15 podemos certificar o funcionamento da instrução *inc*.

Figura 15 – Instrução *inc*

Fonte: O Autor

5.2.6 Instrução *dec*

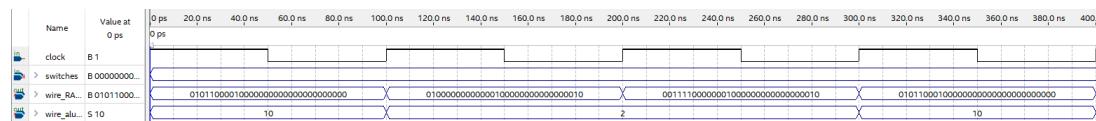
Na Figura 16 podemos certificar o funcionamento da instrução *dec*.

Figura 16 – Instrução *dec*

Fonte: O Autor

5.2.7 Instrução *lw*

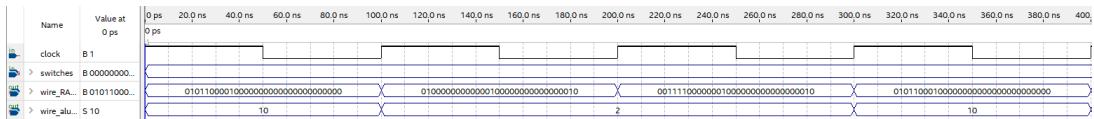
Na Figura 17 podemos certificar o funcionamento da instrução *lw*.

Figura 17 – Instrução *lw*

Fonte: O Autor

5.2.8 Instrução *sw*

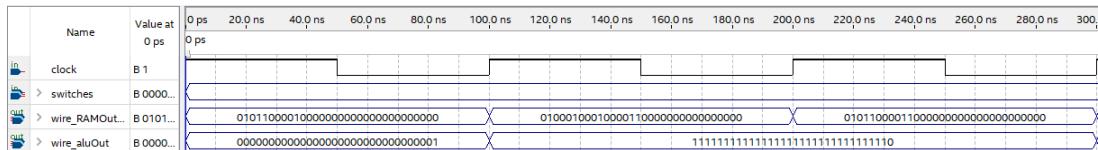
Na Figura 18 podemos certificar o funcionamento da instrução *sw*.

Figura 18 – Instrução *sw*

Fonte: O Autor

5.2.9 Instrução *not*

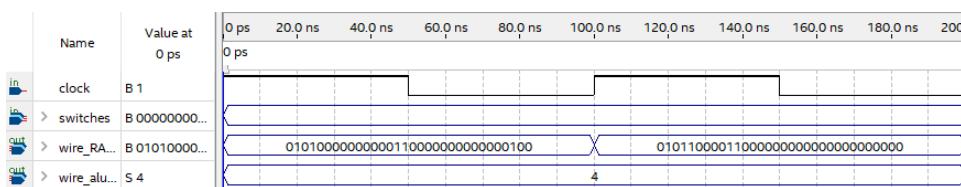
Na Figura 19 podemos certificar o funcionamento da instrução *not*.

Figura 19 – Instrução *not*

Fonte: O Autor

5.2.10 Instrução *lwi*

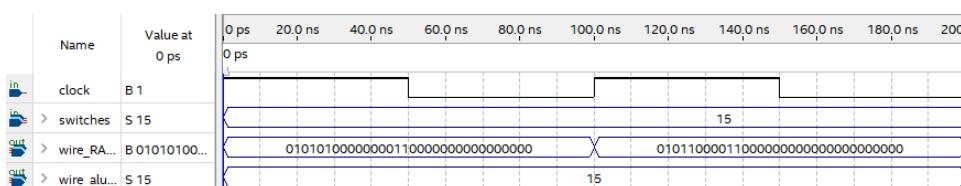
Na Figura 20 podemos certificar o funcionamento da instrução *lwi*.

Figura 20 – Instrução *lwi*

Fonte: O Autor

5.2.11 Instrução *in*

Na Figura 21 podemos certificar o funcionamento da instrução *in*.

Figura 21 – Instrução *in*

Fonte: O Autor

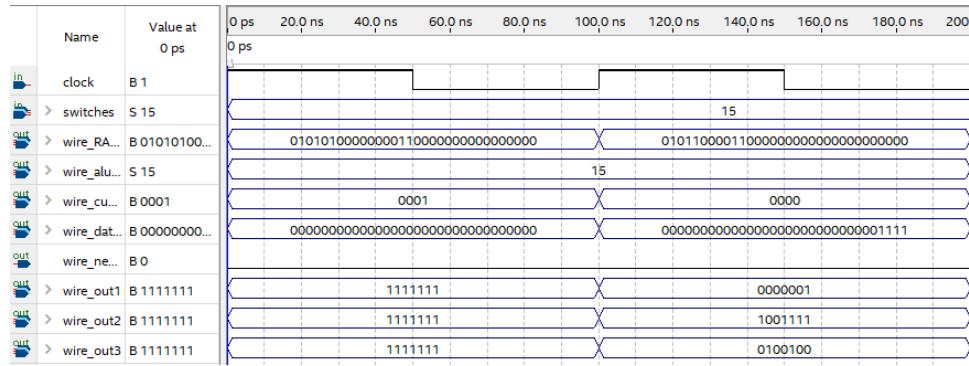
5.3 Teste Instrução do tipo K

Iremos nessa seção realizar o teste de todas as *Waveforms* geradas a partir das instruções do tipo K no *software* Intel® Quartus Prime.

5.3.1 Instrução *out*

Na [Figura 22](#) podemos certificar o funcionamento da instrução *out*.

Figura 22 – Instrução *out*



Fonte: O Autor

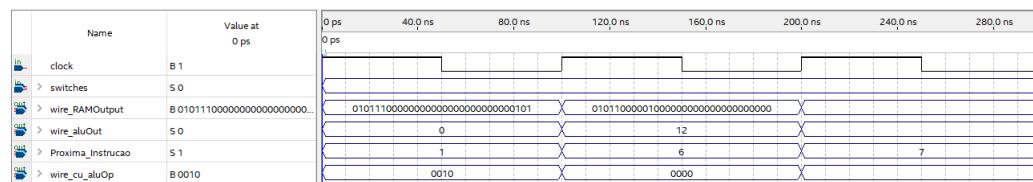
5.4 Teste Instrução do tipo J

Iremos nessa seção realizar o teste de todas as *Waveforms* geradas a partir das instruções do tipo J no *software* Intel® Quartus Prime.

5.4.1 Instrução *j*

Na [Figura 23](#) podemos certificar o funcionamento da instrução *j*.

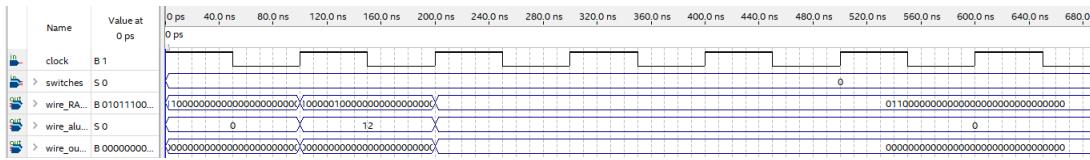
Figura 23 – Instrução *j*



Fonte: O Autor

5.4.2 Instrução *hlt*

Na [Figura 24](#) podemos certificar o funcionamento da instrução *hlt*.

Figura 24 – Instrução *hlt*

Fonte: O Autor

5.5 Teste na *Altera's Cyclone® IV FPGA*

Para certificação do funcionamento dos módulos de entrada e saída, foram executados dois algoritmos.

5.5.1 Teste do Algoritmo A

Código do arquivo algortimoA.v

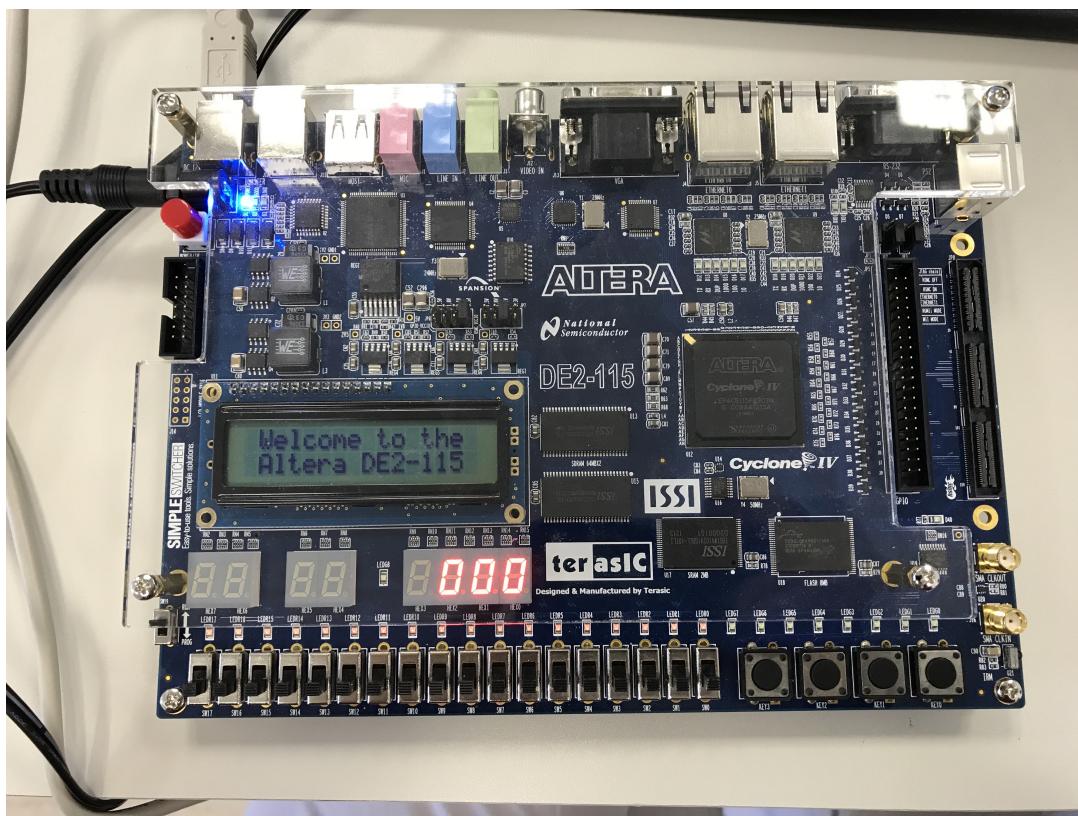
```

1  instructionRAM[0] = 32'b010110_00000_00000000000000000000; //out R[0]
2  instructionRAM[1] = 32'b010101_00000_00010_0000000000000000; //in R[2] <- X
3  instructionRAM[2] = 32'b010110_00010_00000000000000000000; //out R[2]
4  instructionRAM[3] = 32'b010101_00000_00001_0000000000000000; //in R[1] <- 0
5  instructionRAM[4] = 32'b010110_00001_00000000000000000000; //out R[1]
6  instructionRAM[5] = 32'b010101_00000_00011_0000000000000000; //in R[3] <- 2
7  instructionRAM[6] = 32'b010110_00011_00000000000000000000; //out R[3]
8  instructionRAM[7] = 32'b010101_00000_00110_0000000000000000; //in R[6] <- 3
9  instructionRAM[8] = 32'b010110_00110_00000000000000000000; //out R[6]

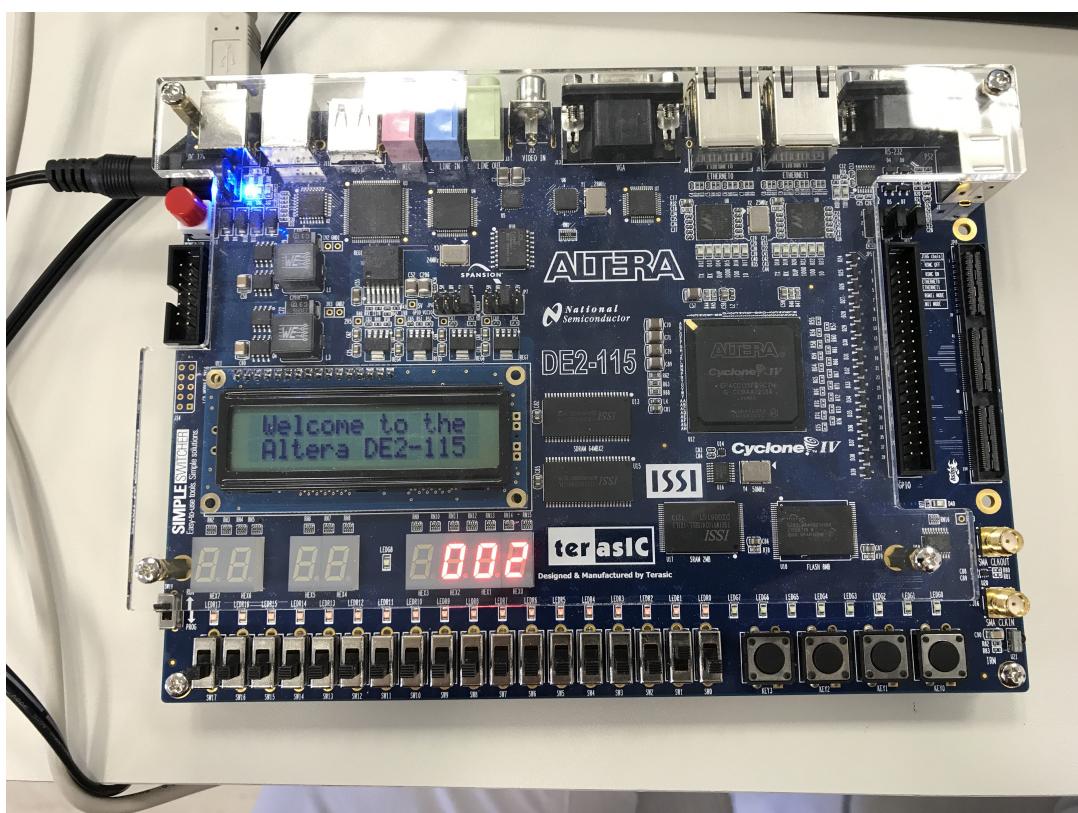
10
11 instructionRAM[9] = 32'b010000_00000_00010_0000000000000001; //sw do R[2] no MEM[1]
12
13 instructionRAM[10] = 32'b001001_00001_00110_0000000000010000; // beq R[1] e R[6]
14 instructionRAM[11] = 32'b001101_00000_00010_0000000000000000; //inc R[2]
15 instructionRAM[12] = 32'b000110_00011_00010_00010_000000000000; //mul R[2] = R[2] * R[3]
16 instructionRAM[13] = 32'b010110_00010_00000000000000000000; //out R[2]
17 instructionRAM[14] = 32'b001101_00000_00001_0000000000000000; //inc R[1]
18 instructionRAM[15] = 32'b010111_0000000000000000000000000000001010; //j INS[10]
19 instructionRAM[16] = 32'b001111_00000_00100_0000000000000000; //lw do MEM[1] no R[4]
20
21 instructionRAM[17] = 32'b000001_00100_00010_00101_000000000000; //sub R[5] = R[4] - R[2]
22 instructionRAM[18] = 32'b010110_00101_00000000000000000000; //out R[5]
23 instructionRAM[19] = 32'b011000_00101_00000000000000000000; // hlt

```

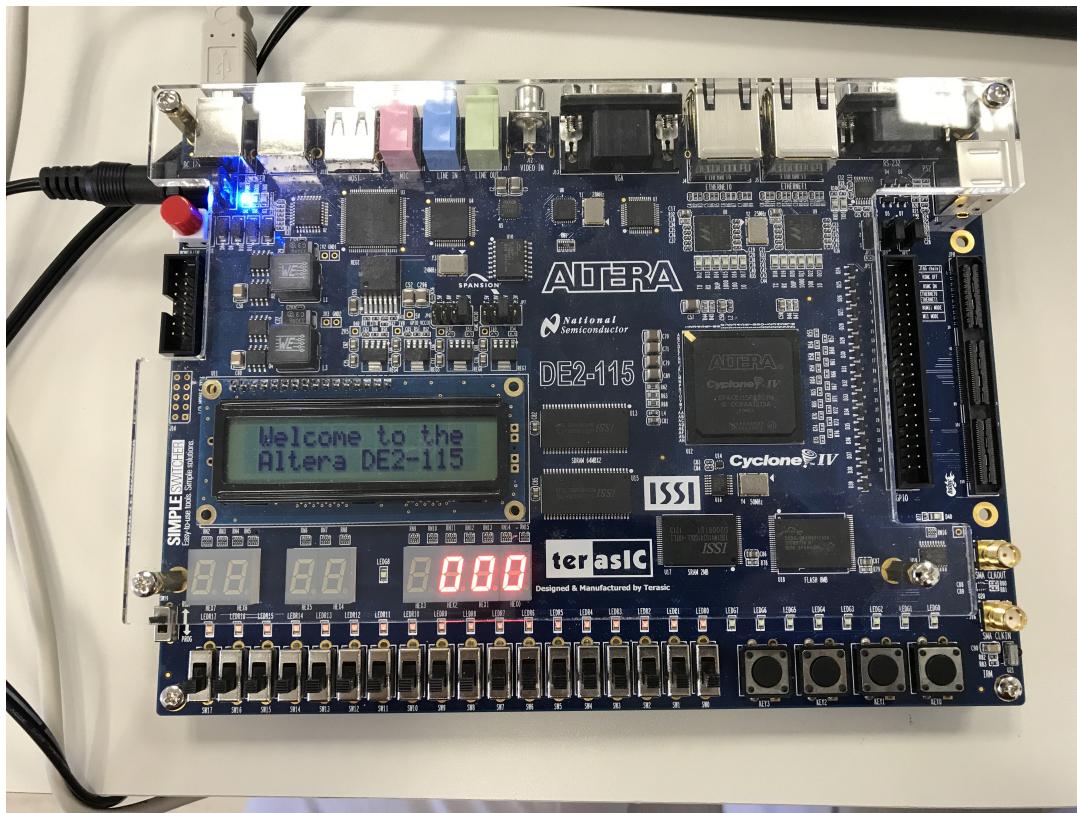
Com o algoritmo salvo na memória de instrução, obtivemos como resposta os seguintes valores no display de sete segmentos:



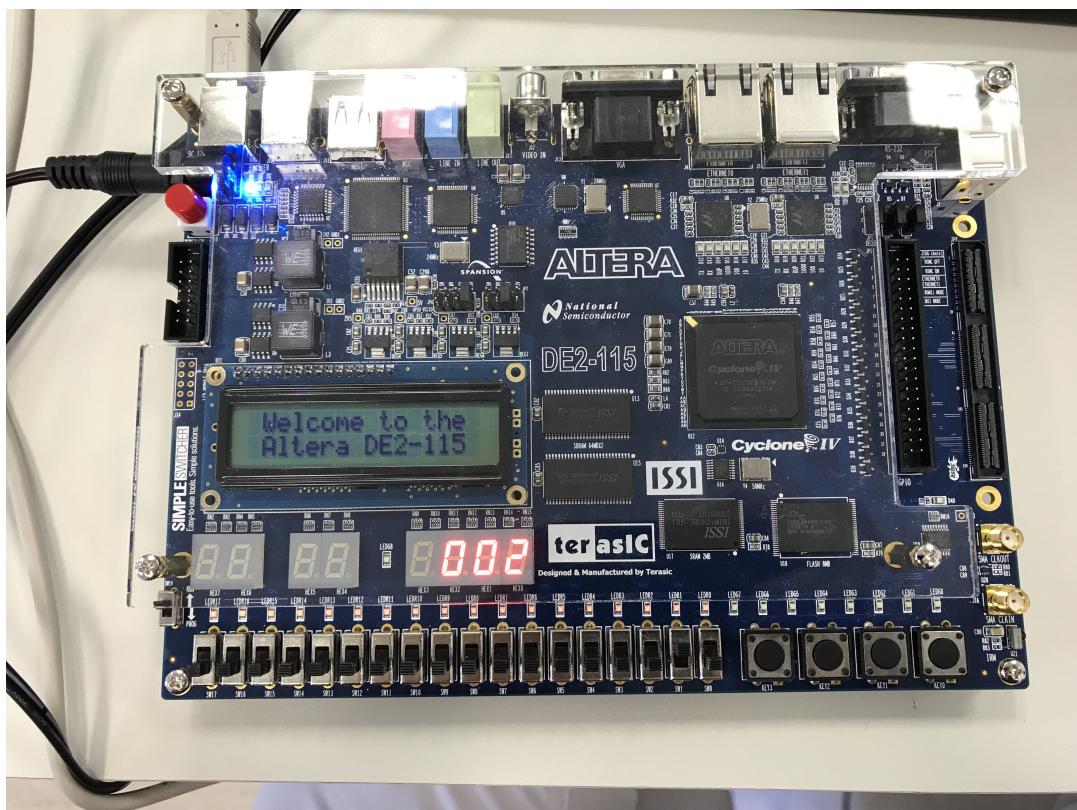
Fonte: O Autor



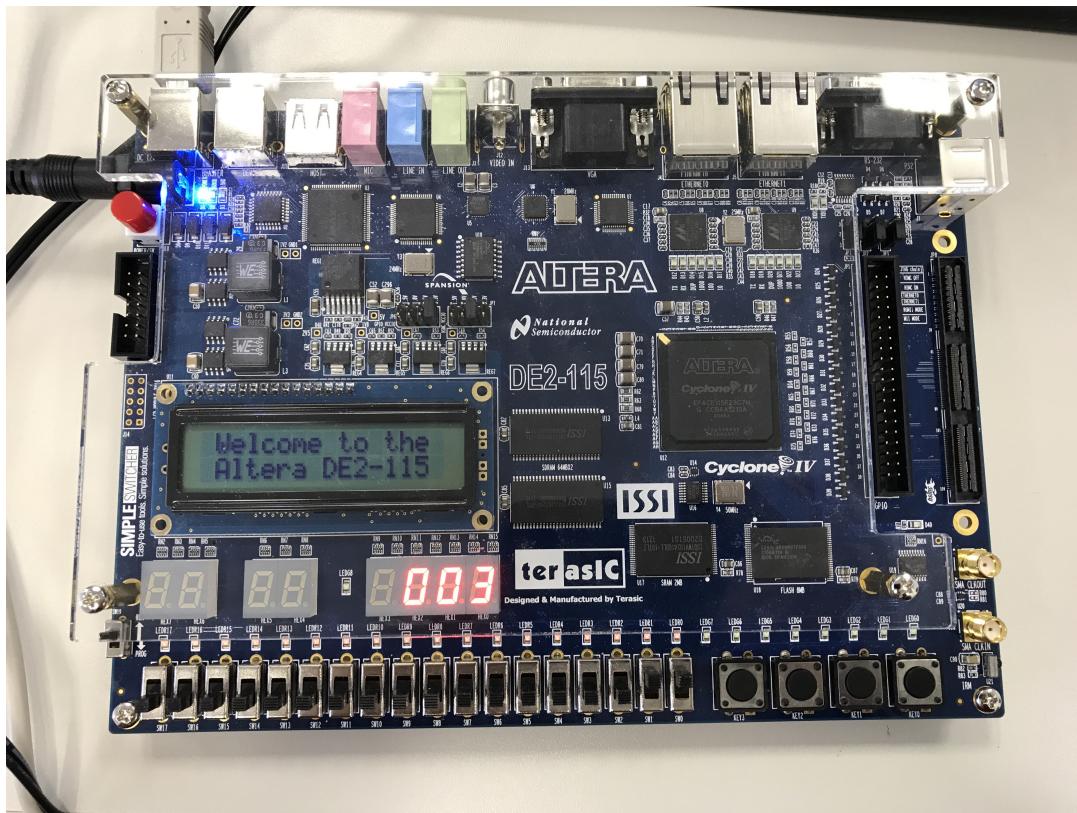
Fonte: O Autor



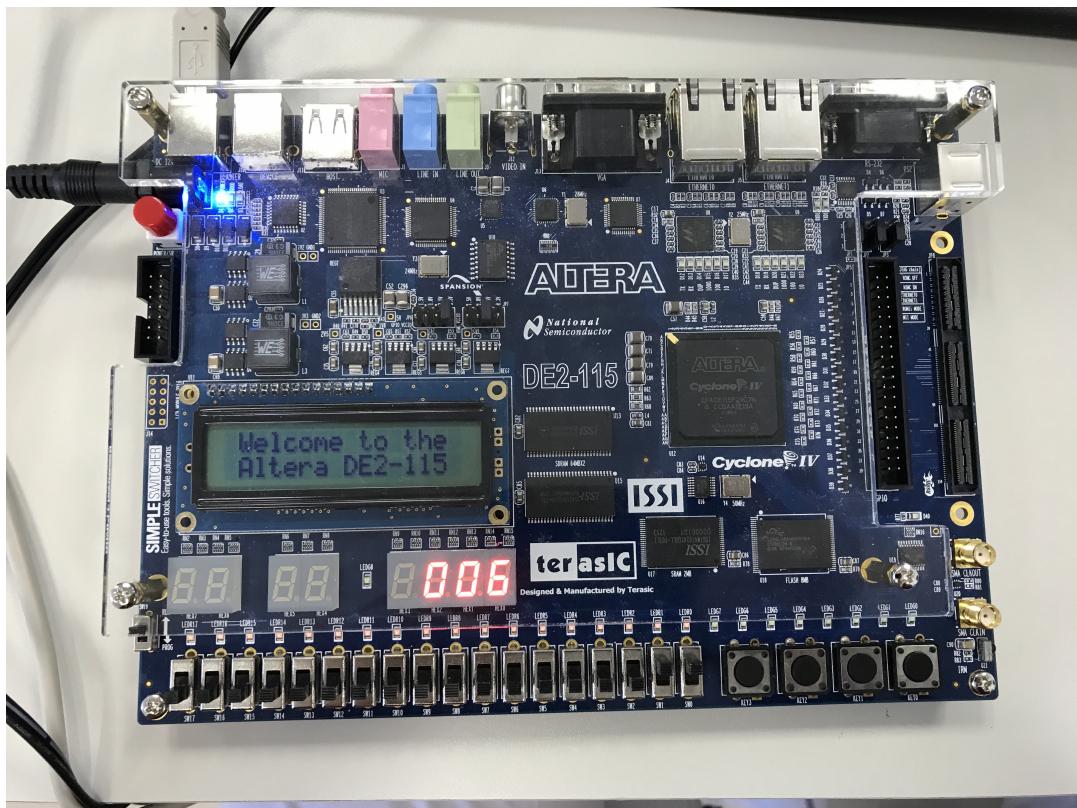
Fonte: O Autor



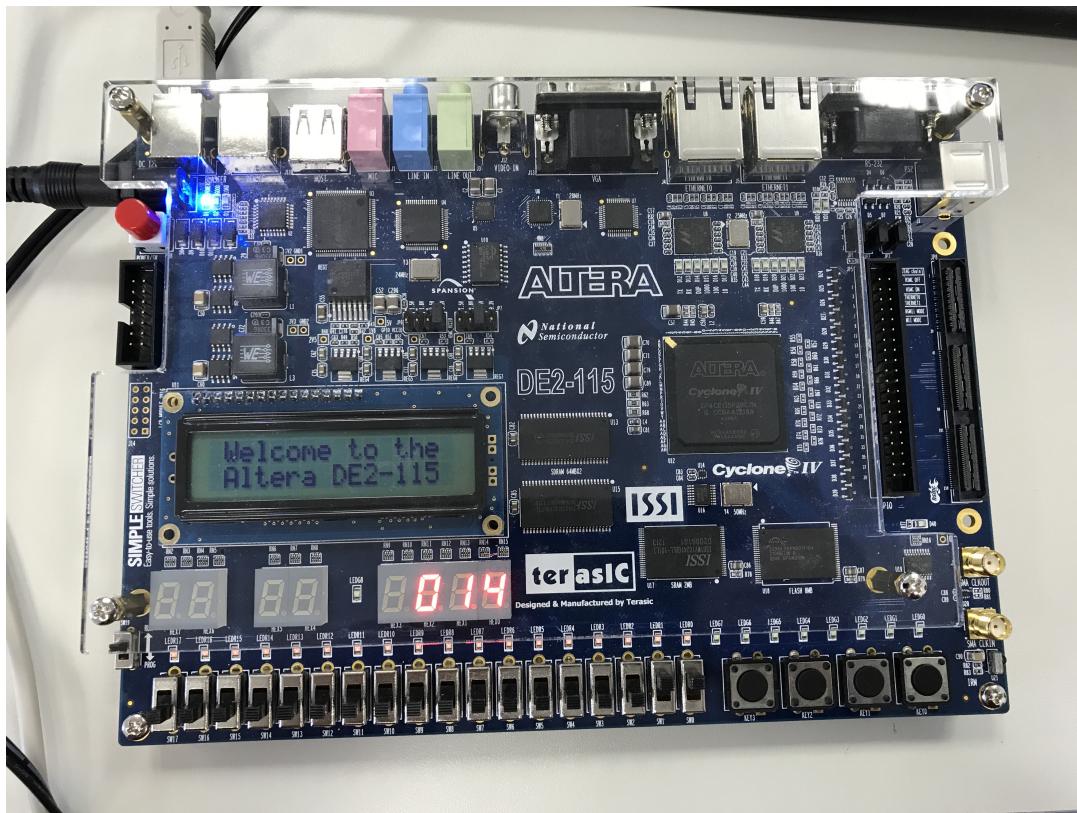
Fonte: O Autor



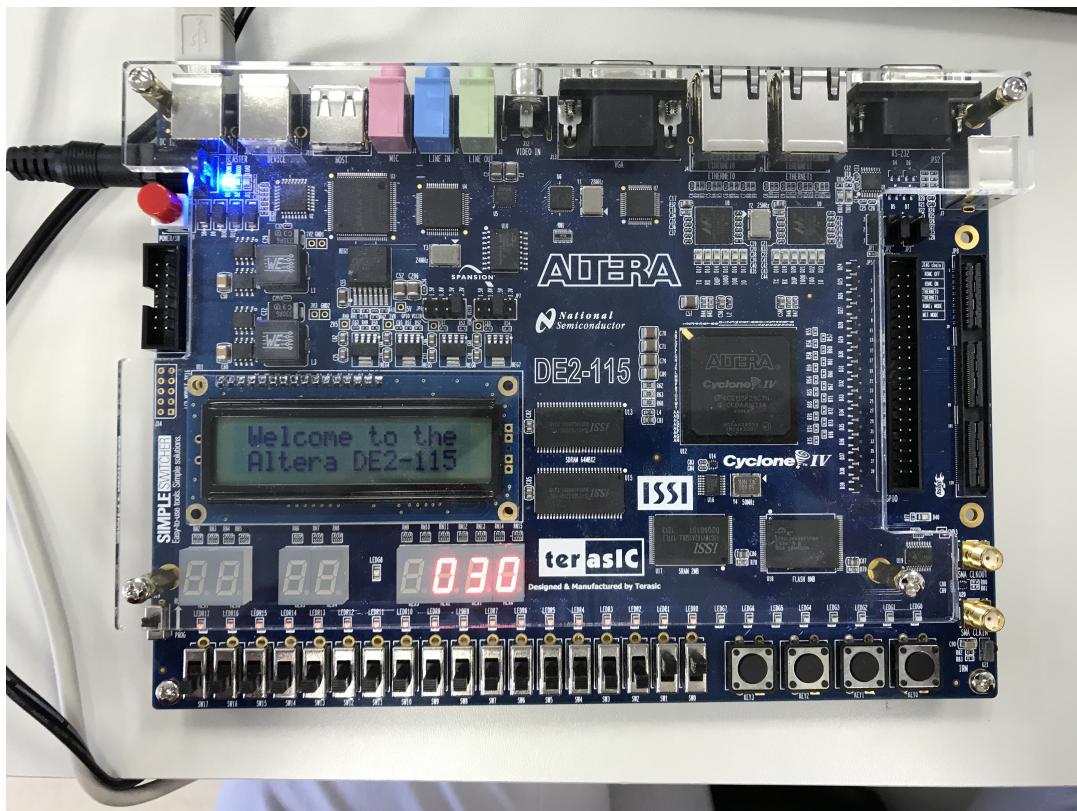
Fonte: O Autor



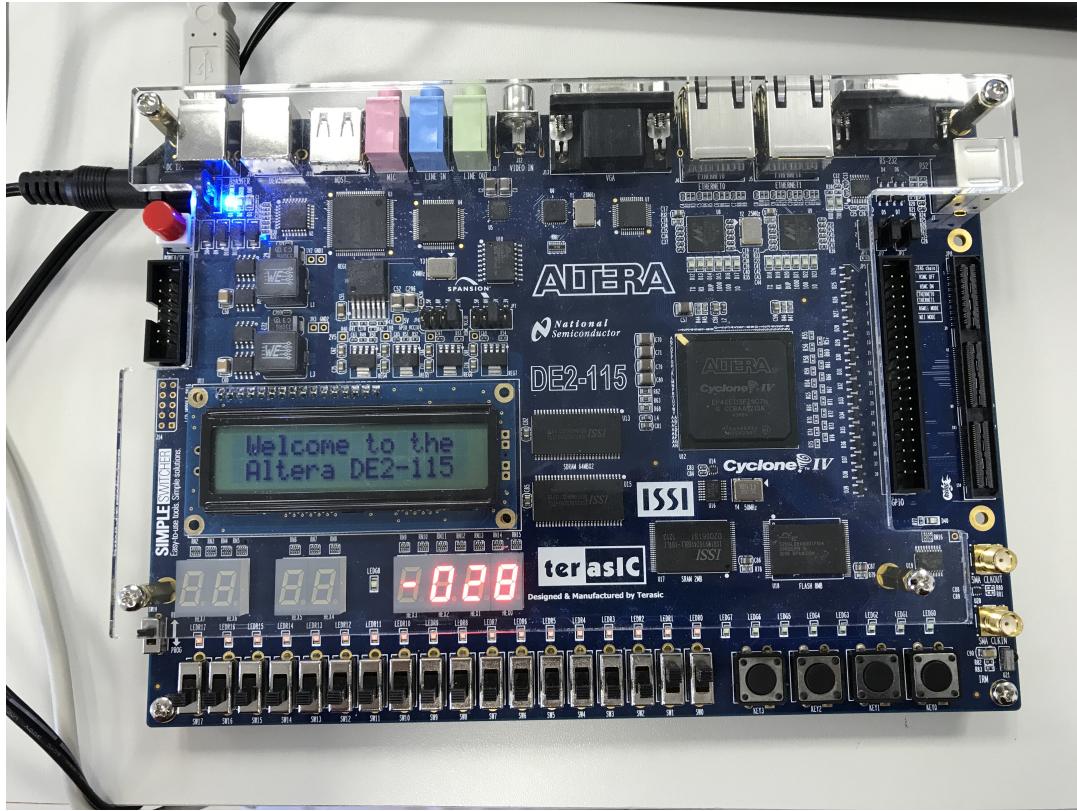
Fonte: O Autor



Fonte: O Autor



Fonte: O Autor



Fonte: O Autor

5.5.2 Teste do Algoritmo B

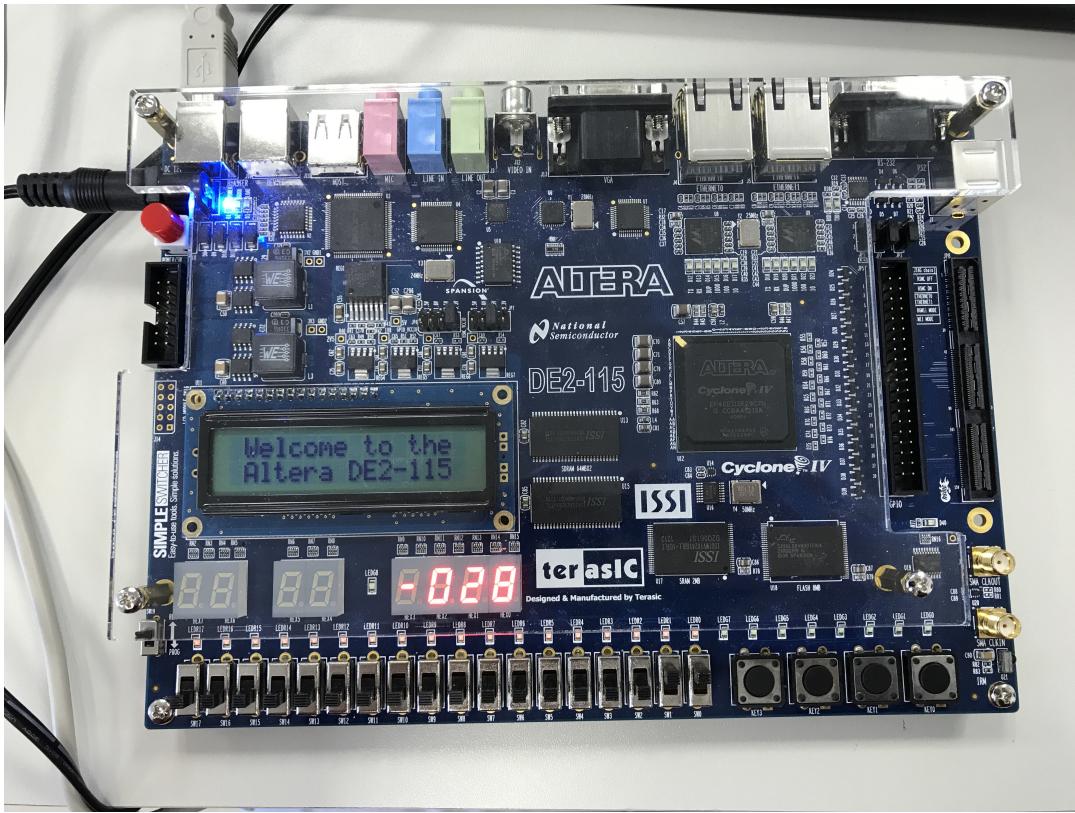
Código do arquivo algoritmoB.v

```

1 instructionRAM[0] = 32'b010110_00000_00000000000000000000; //out R[0]
2 instructionRAM[1] = 32'b010101_00000_00010_0000000000000000; //in R[2] <- X
3 instructionRAM[2] = 32'b010110_00010_00000000000000000000; //out R[2]
4 instructionRAM[3] = 32'b010101_00000_00001_0000000000000000; //in R[1] <- 0
5 instructionRAM[4] = 32'b010110_00001_00000000000000000000; //out R[1]
6 instructionRAM[5] = 32'b010101_00000_00011_0000000000000000; //in R[3] <- 2
7 instructionRAM[6] = 32'b010110_00011_00000000000000000000; //out R[3]
8 instructionRAM[7] = 32'b010101_00000_00110_0000000000000000; //in R[6] <- 3
9 instructionRAM[8] = 32'b010110_000110_00000000000000000000; //out R[6]
10
11 instructionRAM[9] = 32'b010000_00000_00010_0000000000000001; //sw do R[2] no MEM[1]
12
13 instructionRAM[10] = 32'b001001_00001_00110_0000000000010000; // beq R[1] e R[6]
14 instructionRAM[11] = 32'b001101_00000_00010_0000000000000000; //inc R[2]
15 instructionRAM[12] = 32'b000110_00011_00010_00010_000000000000; //mul R[2] = R[2] * R[3]
16 instructionRAM[13] = 32'b010110_00010_00000000000000000000; //out R[2]
17 instructionRAM[14] = 32'b001101_00000_00001_0000000000000000; //inc R[1]
18 instructionRAM[15] = 32'b010111_000000000000000000001010; //j INS[10]
19 instructionRAM[16] = 32'b001111_00000_00100_0000000000000001; //lw do MEM[1] no R[4]
20
21 instructionRAM[17] = 32'b000001_00100_00010_00101_000000000000; //sub R[5] = R[4] - R[2]
22 instructionRAM[18] = 32'b010110_00101_00000000000000000000; //out R[5]
23 instructionRAM[19] = 32'b011000_00101_00000000000000000000; // hlt

```

Com o algoritmo salvo na memória de instrução, obtivemos como resposta os seguintes valor no display de sete segmentos:



Fonte: O Autor

6 Conclusão

O objetivo especificado no início do projeto foi o desenvolvimento de uma unidade de processamento computacional em lógica programável. A primeira ação executada foi o desenvolvimento do conjunto de instruções e a partir dele, o desenvolvimento do caminho de dados. Com as instruções especificadas, o trabalho foi direcionado ao desenvolvimento dos blocos da UCP a partir da linguagem de descrição de *hardware* Verilog. Com metade do projeto completado, a atenção nesse momento foi dado ao desenvolvimento da unidade de controle, controlando de forma automática os movimentos da UCP. Por fim, mas não menos importante, foi realizado os testes nas *waveforms* e na placa *Altera's Cyclone® IV FPGA*. Após a correção dos *bugs*, o projeto foi apresentado ao professor, encerrando então, todo o trabalho.

7 Considerações Finais

O objetivo proposto no inicio do relatório foi relatar o desenvolvimento de uma unidade de processamento computacional em lógica programável, e o mesmo foi concluído com sucesso. Dentre as dificuldades encontradas no desenvolvimento do projeto podemos citar as constantes falhas no *software Quartus* que o levava a um *crash* e assim encerrando o mesmo. Outra dificuldade encontrada foi a de testar as instruções sem uma unidade de controle, tendo que assim inserir todos os sinais de controle manualmente. Por fim, a correção dos *bugs* devido a sinais de controles "setados" de maneira incorreta e instruções montadas na memória de instrução de maneira incorreta.

Referências

- 1 PROCESSADOR. Disponível em: <<https://pt.wikipedia.org/wiki/Microprocessador>>. Citado na página 13.
- 2 CONJUNTO de Instruções. Disponível em: <https://pt.wikipedia.org/wiki/Conjunto_de_instru%C3%A7%C3%A3o>. Citado na página 13.
- 3 RISC. Disponível em: <<http://www.sistemasembarcados.org/2015/11/15/processadores-arquitetura-risc-e-cisc/>>. Citado na página 13.
- 4 CISC. Disponível em: <<https://pt.wikipedia.org/wiki/CISC>>. Citado na página 14.
- 5 UNIDADE de Processamento. Disponível em: <https://pt.wikipedia.org/wiki/Unidade_central_de_processamento>. Citado na página 14.
- 6 CONTADOR de Programa. Disponível em: <https://pt.wikipedia.org/wiki/Contador_de_programa>. Citado na página 15.
- 7 MEMÓRIA de Instrução. Disponível em: <<http://www.di.ufpb.br/raimundo/ArqDI/Arq6.htm>>. Citado na página 15.
- 8 REGISTRADOR. Disponível em: <[https://pt.wikipedia.org/wiki/Registrador_\(inform%C3%A1tica\)](https://pt.wikipedia.org/wiki/Registrador_(inform%C3%A1tica))>. Citado na página 15.
- 9 ULA. Disponível em: <https://pt.wikipedia.org/wiki/Unidade_l%C3%B3gica_e_aritm%C3%A9tica>. Citado na página 15.
- 10 MULTIPLEXADOR. Disponível em: <<https://pt.wikipedia.org/wiki/Multiplexador>>. Citado na página 16.
- 11 LINGUAGEM de Descrição de Hardware. Disponível em: <https://pt.wikipedia.org/wiki/Linguagem_de_descri%C3%A7%C3%A3o_de_hardware>. Citado na página 16.
- 12 VERILOG. Disponível em: <<https://pt.wikipedia.org/wiki/Verilog>>. Citado na página 17.
- 13 QUARTUS. Disponível em: <<https://pt.wikipedia.org/wiki/Altera>>. Citado na página 17.

Apêndices

APÊNDICE A – ALU.v

Código do arquivo ALU.v

```

1  module ALU (data1, data2, cu_aluOp, zero, /*negative,*/ shamt, aluOut);
2    input [31:0] data1, data2; //Dados para opera o da ALU
3    input [3:0] cu_aluOp; //Sinal vindo da unidade de controle para sele o da
4      opera o
5    input [4:0] shamt; //Sinal para descolocamento.
6
7    output reg [31:0] aluOut; //Dados de sa da da ALU;
8    //output negative;
9    output zero;
10
11   always @ (cu_aluOp or data1 or data2) begin
12     case(cu_aluOp[3:0])
13       4'b0000: aluOut = data1; //Mantem
14       4'b0001: aluOut = data1 + data2; //Soma
15       4'b0010: aluOut = data1 - data2; // Subtrai
16       4'b0011: aluOut = data2 + 1; //Soma 1
17       4'b0100: aluOut = data2 - 1; //Subtrai 1
18       4'b0101: aluOut = data1 & data2; //And
19       4'b0110: aluOut = data1 | data2; //Or
20       4'b0111: aluOut = data1 ^ data2; //Xor
21       4'b1000: aluOut = ~ data1; //Not
22       4'b1001: aluOut = data1 << shamt; //Desloca para esquerda
23       4'b1010: aluOut = data1 >> shamt; //Desloca para direita
24       4'b1011: aluOut = data1 < data2 ? 1 : 0; //Set on less than
25       4'b1100: aluOut = data1 * data2; //Multiplica
26       4'b1101: aluOut = data1 / data2; //Divide
27       4'b1110: aluOut = data1 % data2; //Resto
28         4'b1111: aluOut = data2 + 0; //Mantem quando valor vem do segundo "slot"
          de registradores
29     endcase
30   end
31
32   assign zero = (aluOut == 0); //Se o aluOut for zero, flag "zero" ativada.
33   //assign negative = ($signed(aluOut) < 0); //Se o aluOut for negativo, flag "negative"
          ativada.
34 endmodule // Unidade l gica e aritim tica.

```


APÊNDICE B – and.v

Código do arquivo and.v

```
1 module and (zero, cu_Branch, pcScr);
2   input zero;
3   input cu_Branch;
4   output reg PCScr;
5
6   if((zero && cu_Branch) == 1)
7     pcScr = 1;
8 endmodule // and
```


APÊNDICE C – registerBench.v

Código do arquivo registerBench.v

```

1 module registerBench (readAddy1, readAddy2, writeAddy, writeData, data1, data2, data3,
2   clock, cu_writeReg);
3   input [4:0] readAddy1; //Endere o de Leitura 1
4   input [4:0] readAddy2; //Endere o de Leitura 2
5   input [4:0] writeAddy; //Endere o de Escrita - cu_writeReg virou cu_regWrite
6
7   input [31:0] writeData; //Dados a ser escrito
8   input clock, cu_writeReg; //Clock e Sinal de controle para escrita no Registrador
9
10  output [31:0] data1; //Dados de saída 1
11  output [31:0] data2; //Dados de saída 2
12  output [31:0] data3; //Dados de saída 3
13
14  reg [31:0] regBench [31:0];
15
16  always @ (posedge clock) begin
17    if(cu_writeReg)begin
18      regBench[writeAddy] = writeData;
19      end
20      regBench[0] = 32'b0;
21    end
22
23  assign data1 = (regBench[readAddy1]);
24  assign data2 = (regBench[readAddy2]);
25  assign data3 = (regBench[writeAddy]);
26
27
28 endmodule // Banco de registradores

```


APÊNDICE D – controlUnity.v

Código do arquivo controlUnity.v

```

1  module controlUnity (opcode, cu_writeReg, cu_regDest, cu_memtoReg, cu_Jump, cu_inSignal,
2    cu_aluScr, cu_writeEnable, cu_readEnable, cu_Branch, cu_aluOp, cu_hlt, cu_reset,
3    cu_showDisplay);
4  input [5:0] opcode;
5  output reg cu_writeReg, cu_regDest, cu_memtoReg, cu_Jump, cu_inSignal, cu_aluScr,
6    cu_writeEnable, cu_readEnable, cu_Branch, cu_hlt, cu_reset, cu_showDisplay;
7  output reg [3:0] cu_aluOp;
8
9
10 always @ (opcode) begin
11   case(opcode)
12     6'b000000: begin //Add_OK
13       cu_writeReg = 1'b1;
14       cu_regDest = 1'b0;
15       cu_memtoReg = 1'b0;
16       cu_Jump = 1'b0;
17       cu_inSignal = 1'b0;
18       cu_aluScr = 1'b0;
19       cu_writeEnable = 1'b0;
20       cu_readEnable = 1'b0;
21       cu_Branch = 1'b0;
22       cu_aluOp = 4'b0001;
23       cu_hlt = 1'b0;
24       cu_reset = 1'b0;
25       cu_showDisplay = 1'b0;
26     end
27
28     6'b000001: begin //Subtract_OK
29       cu_writeReg = 1'b1;
30       cu_regDest = 1'b0;
31       cu_memtoReg = 1'b0;
32       cu_Jump = 1'b0;
33       cu_inSignal = 1'b0;
34       cu_aluScr = 1'b0;
35       cu_writeEnable = 1'b0;
36       cu_readEnable = 1'b0;
37       cu_Branch = 1'b0;
38         cu_aluOp = 4'b0010;
39       cu_hlt = 1'b0;
40       cu_reset = 1'b0;
41       cu_showDisplay = 1'b0;
42     end
43
44     6'b000010: begin //And_OK
45       cu_writeReg = 1'b1;
46       cu_regDest = 1'b0;
47       cu_memtoReg = 1'b0;
48       cu_Jump = 1'b0;
49       cu_inSignal = 1'b0;
      cu_aluScr = 1'b0;
      cu_writeEnable = 1'b0;
      cu_readEnable = 1'b0;
      cu_Branch = 1'b0;
    end
  end
endmodule

```

```
50     cu_aluOp = 4'b0101;
51     cu_hlt = 1'b0;
52     cu_reset = 1'b0;
53         cu_showDisplay = 1'b0;
54     end
55
56 6'b000011: begin //0r_OK
57     cu_writeReg = 1'b1;
58     cu_regDest = 1'b0;
59     cu_memtoReg = 1'b0;
60     cu_Jump = 1'b0;
61     cu_inSignal = 1'b0;
62     cu_aluScr = 1'b0;
63     cu_writeEnable = 1'b0;
64     cu_readEnable = 1'b0;
65     cu_Branch = 1'b0;
66     cu_aluOp = 4'b0110;
67     cu_hlt = 1'b0;
68     cu_reset = 1'b0;
69         cu_showDisplay = 1'b0;
70     end
71
72 6'b000100: begin //Xor_OK
73     cu_writeReg = 1'b1;
74     cu_regDest = 1'b0;
75     cu_memtoReg = 1'b0;
76     cu_Jump = 1'b0;
77     cu_inSignal = 1'b0;
78     cu_aluScr = 1'b0;
79     cu_writeEnable = 1'b0;
80     cu_readEnable = 1'b0;
81     cu_Branch = 1'b0;
82     cu_aluOp = 4'b0111;
83     cu_hlt = 1'b0;
84     cu_reset = 1'b0;
85         cu_showDisplay = 1'b0;
86     end
87
88 6'b000101: begin //Set on less than_OK
89     cu_writeReg = 1'b1;
90     cu_regDest = 1'b0;
91     cu_memtoReg = 1'b0;
92     cu_Jump = 1'b0;
93     cu_inSignal = 1'b0;
94     cu_aluScr = 1'b0;
95     cu_writeEnable = 1'b0;
96     cu_readEnable = 1'b0;
97     cu_Branch = 1'b0;
98     cu_aluOp = 4'b1011;
99     cu_hlt = 1'b0;
100    cu_reset = 1'b0;
101        cu_showDisplay = 1'b0;
102    end
103
104 6'b000110: begin //Multiply_OK
105     cu_writeReg = 1'b1;
106     cu_regDest = 1'b0;
107     cu_memtoReg = 1'b0;
108     cu_Jump = 1'b0;
109     cu_inSignal = 1'b0;
```

```

110     cu_aluScr = 1'b0;
111     cu_writeEnable = 1'b0;
112     cu_readEnable = 1'b0;
113     cu_Branch = 1'b0;
114     cu_aluOp = 4'b1100;
115     cu_hlt = 1'b0;
116     cu_reset = 1'b0;
117         cu_showDisplay = 1'b0;
118     end
119
120 6'b000111: begin //Divide_OK
121     cu_writeReg = 1'b1;
122     cu_regDest = 1'b0;
123     cu_memtoReg = 1'b0;
124     cu_Jump = 1'b0;
125     cu_inSignal = 1'b0;
126     cu_aluScr = 1'b0;
127     cu_writeEnable = 1'b0;
128     cu_readEnable = 1'b0;
129     cu_Branch = 1'b0;
130     cu_aluOp = 4'b1101;
131     cu_hlt = 1'b0;
132     cu_reset = 1'b0;
133         cu_showDisplay = 1'b0;
134     end
135
136 6'b001000: begin //Rest_OK
137     cu_writeReg = 1'b1;
138     cu_regDest = 1'b0;
139     cu_memtoReg = 1'b0;
140     cu_Jump = 1'b0;
141     cu_inSignal = 1'b0;
142     cu_aluScr = 1'b0;
143     cu_writeEnable = 1'b0;
144     cu_readEnable = 1'b0;
145     cu_Branch = 1'b0;
146     cu_aluOp = 4'b1110;
147     cu_hlt = 1'b0;
148     cu_reset = 1'b0;
149         cu_showDisplay = 1'b0;
150     end
151
152 6'b001001: begin //Branch on Equal
153     cu_writeReg = 1'b0;
154     cu_regDest = 1'b1;
155     cu_memtoReg = 1'b0;
156     cu_Jump = 1'b0;
157     cu_inSignal = 1'b0;
158     cu_aluScr = 1'b0;
159     cu_writeEnable = 1'b0;
160     cu_readEnable = 1'b0;
161     cu_Branch = 1'b1;
162     cu_aluOp = 4'b0010;
163     cu_hlt = 1'b0;
164     cu_reset = 1'b0;
165         cu_showDisplay = 1'b0;
166     end
167
168 6'b001010: begin //Branch on not Equal
169     cu_writeReg = 1'b0;

```

```
170     cu_regDest = 1'b1;
171     cu_memtoReg = 1'bx;
172     cu_Jump = 1'b0;
173     cu_inSignal = 1'b0;
174     cu_aluScr = 1'b0;
175     cu_writeEnable = 1'b0;
176     cu_readEnable = 1'bx;
177     cu_Branch = 1'b1;
178     cu_aluOp = 4'b0010;
179     cu_hlt = 1'b0;
180     cu_reset = 1'b0;
181         cu_showDisplay = 1'b0;
182     end
183
184 6'b001011: begin //Add Immediate
185     cu_writeReg = 1'b1;
186     cu_regDest = 1'b1;
187     cu_memtoReg = 1'b0;
188     cu_Jump = 1'b0;
189     cu_inSignal = 1'b0;
190     cu_aluScr = 1'b1;
191     cu_writeEnable = 1'b0;
192     cu_readEnable = 1'b0;
193     cu_Branch = 1'b0;
194     cu_aluOp = 4'b0001;
195     cu_hlt = 1'b0;
196     cu_reset = 1'b0;
197         cu_showDisplay = 1'b0;
198     end
199
200 6'b001100: begin //Sub Immediate
201     cu_writeReg = 1'b1;
202     cu_regDest = 1'b1;
203     cu_memtoReg = 1'b0;
204     cu_Jump = 1'b0;
205     cu_inSignal = 1'b0;
206     cu_aluScr = 1'b1;
207     cu_writeEnable = 1'b0;
208     cu_readEnable = 1'bx;
209     cu_Branch = 1'b0;
210     cu_aluOp = 4'b0010;
211     cu_hlt = 1'b0;
212     cu_reset = 1'b0;
213         cu_showDisplay = 1'b0;
214     end
215
216 6'b001101: begin //Increment
217     cu_writeReg = 1'b1;
218     cu_regDest = 1'b1;
219     cu_memtoReg = 1'b0;
220     cu_Jump = 1'b0;
221     cu_inSignal = 1'b0;
222     cu_aluScr = 1'b0;
223     cu_writeEnable = 1'b0;
224     cu_readEnable = 1'b0;
225     cu_Branch = 1'b0;
226     cu_aluOp = 4'b0011;
227     cu_hlt = 1'b0;
228     cu_reset = 1'b0;
229         cu_showDisplay = 1'b0;
```

```

230      end
231
232      6'b001110: begin //Decrement
233          cu_writeReg = 1'b1;
234          cu_regDest = 1'b1;
235          cu_memtoReg = 1'b0;
236          cu_Jump = 1'b0;
237          cu_inSignal = 1'b0;
238          cu_aluScr = 1'b0;
239          cu_writeEnable = 1'b0;
240          cu_readEnable = 1'b0;
241          cu_Branch = 1'b0;
242          cu_aluOp = 4'b0100;
243          cu_hlt = 1'b0;
244          cu_reset = 1'b0;
245              cu_showDisplay = 1'b0;
246      end
247
248      6'b001111: begin // Load Word_OK
249          cu_writeReg = 1'b1;
250          cu_regDest = 1'b1;
251          cu_memtoReg = 1'b1;
252          cu_Jump = 1'b0;
253          cu_inSignal = 1'b0;
254          cu_aluScr = 1'b1;
255          cu_writeEnable = 1'b0;
256          cu_readEnable = 1'b1;
257          cu_Branch = 1'b0;
258          cu_aluOp = 4'b0001;
259          cu_hlt = 1'b0;
260          cu_reset = 1'b0;
261              cu_showDisplay = 1'b0;
262      end
263
264      6'b010000: begin //Store Word_OK
265          cu_writeReg = 1'b0;
266          cu_regDest = 1'b1;
267          cu_memtoReg = 1'b0;
268          cu_Jump = 1'b0;
269          cu_inSignal = 1'b0;
270          cu_aluScr = 1'b1;
271          cu_writeEnable = 1'b1;
272          cu_readEnable = 1'b0;
273          cu_Branch = 1'b0;
274          cu_aluOp = 4'b0001;
275          cu_hlt = 1'b0;
276          cu_reset = 1'b0;
277              cu_showDisplay = 1'b0;
278      end
279
280      6'b010001: begin //Not
281          cu_writeReg = 1'b1;
282          cu_regDest = 1'b1;
283          cu_memtoReg = 1'b0;
284          cu_Jump = 1'b0;
285          cu_inSignal = 1'b0;
286          cu_aluScr = 1'b0;
287          cu_writeEnable = 1'b0;
288          cu_readEnable = 1'b0;
289          cu_Branch = 1'b0;

```

```

290     cu_aluOp = 4'b1000;
291     cu_hlt = 1'b0;
292     cu_reset = 1'b0;
293         cu_showDisplay = 1'b0;
294     end
295
296 6'b010010: begin //Shift Left Logical_NOK
297     cu_writeReg = 1'b1;
298     cu_regDest = 1'b1;
299     cu_memtoReg = 1'b0;
300     cu_Jump = 1'b0;
301     cu_inSignal = 1'b0;
302     cu_aluScr = 1'b1;
303     cu_writeEnable = 1'b0;
304     cu_readEnable = 1'b0;
305     cu_Branch = 1'b0;
306     cu_aluOp = 4'b1001;
307     cu_hlt = 1'b0;
308     cu_reset = 1'b0;
309         cu_showDisplay = 1'b0;
310     end
311
312 6'b010011: begin //Shift Right Logical_NOK
313     cu_writeReg = 1'b1;
314     cu_regDest = 1'b1;
315     cu_memtoReg = 1'b0;
316     cu_Jump = 1'b0;
317     cu_inSignal = 1'b0;
318     cu_aluScr = 1'b1;
319     cu_writeEnable = 1'b0;
320     cu_readEnable = 1'b0;
321     cu_Branch = 1'b0;
322     cu_aluOp = 4'b1010;
323     cu_hlt = 1'b0;
324     cu_reset = 1'b0;
325         cu_showDisplay = 1'b0;
326     end
327
328 6'b010100: begin //Load Word Immediate_OK
329     cu_writeReg = 1'b1;
330     cu_regDest = 1'b1;
331     cu_memtoReg = 1'b0;
332     cu_Jump = 1'b0;
333     cu_inSignal = 1'b0;
334     cu_aluScr = 1'b1;
335     cu_writeEnable = 1'b0;
336     cu_readEnable = 1'b0;
337     cu_Branch = 1'b0;
338     cu_aluOp = 4'b0001;
339     cu_hlt = 1'b0;
340     cu_reset = 1'b0;
341         cu_showDisplay = 1'b0;
342     end
343
344 6'b010101: begin //In_OK
345     cu_writeReg = 1'b1;
346     cu_regDest = 1'b1;
347     cu_memtoReg = 1'b0;
348     cu_Jump = 1'b0;
349     cu_inSignal = 1'b1;

```

```

350     cu_aluScr = 1'b1;
351     cu_writeEnable = 1'b0;
352     cu_readEnable = 1'b0;
353     cu_Branch = 1'b0;
354     cu_aluOp = 4'b0001;
355     cu_hlt = 1'b0;
356     cu_reset = 1'b0;
357         cu_showDisplay = 1'b0;
358     end
359
360 6'b010110: begin //Out_OK
361     cu_writeReg = 1'b0;
362     cu_regDest = 1'b0;
363     cu_memtoReg = 1'b0;
364     cu_Jump = 1'b0;
365     cu_inSignal = 1'b0;
366     cu_aluScr = 1'b0;
367     cu_writeEnable = 1'b0;
368     cu_readEnable = 1'b0;
369     cu_Branch = 1'b0;
370     cu_aluOp = 4'b0000;
371     cu_hlt = 1'b0;
372     cu_reset = 1'b0;
373         cu_showDisplay = 1'b1;
374     end
375
376 6'b010111: begin //Jump
377     cu_writeReg = 1'bx;
378     cu_regDest = 1'bx;
379     cu_memtoReg = 1'bx;
380     cu_Jump = 1'b1;
381     cu_inSignal = 1'b0;
382     cu_aluScr = 1'bx;
383     cu_writeEnable = 1'bx;
384     cu_readEnable = 1'bx;
385     cu_Branch = 1'b0;
386     cu_aluOp = 4'bxxxx;
387     cu_hlt = 1'b0;
388     cu_reset = 1'b0;
389         cu_showDisplay = 1'b0;
390     end
391
392 6'b011000: begin //No Operation
393     cu_writeReg = 1'bx;
394     cu_regDest = 1'bx;
395     cu_memtoReg = 1'bx;
396     cu_Jump = 1'bx;
397     cu_inSignal = 1'b0;
398     cu_aluScr = 1'bx;
399     cu_writeEnable = 1'bx;
400     cu_readEnable = 1'bx;
401     cu_Branch = 1'bx;
402     cu_aluOp = 4'bxxxx;
403     cu_hlt = 1'b1;
404     cu_reset = 1'b0;
405         cu_showDisplay = 1'b0;
406     end
407
408 6'b011001: begin //Reset
409     cu_writeReg = 1'bx;

```

```
410 cu_regDest = 1'bx;
411 cu_memtoReg = 1'bx;
412 cu_Jump = 1'bx;
413 cu_inSignal = 1'b0;
414 cu_aluScr = 1'bx;
415 cu_writeEnable = 1'bx;
416 cu_readEnable = 1'bx;
417 cu_Branch = 1'bx;
418 cu_aluOp = 4'bxxxx;
419 cu_hlt = 1'b0;
420 cu_reset = 1'b1;
421         cu_showDisplay = 1'b0;
422 end
423
424 endcase
425 end
426 endmodule // controlUnityopcode ,
```

APÊNDICE E – dataMemory.v

Código do arquivo dataMemory.v

```

1  module dataMemory (memoryAddy , writeData , cu_writeEnable , clock , dataRAMOutput ,
2    cu_readEnable );
3
4  input [9:0] memoryAddy ;
5  input [31:0] writeData;
6  input cu_writeEnable , clock , cu_readEnable ;
7
8  output [31:0] dataRAMOutput;
9
10
11 reg [31:0] RAM[/*1023*/30:0];
12 reg [9:0] regAddy;
13
14
15 always @ (negedge clock) begin
16   if (cu_writeEnable == 1) begin
17     RAM[memoryAddy] = writeData;
18     regAddy = memoryAddy;
19   end
20   if(cu_readEnable == 1) begin //Adicionado duas flags de controle para leitura e
21     regAddy = memoryAddy;
22   end
23 end
24
25 assign dataRAMOutput = RAM[regAddy];
26
27
28 endmodule // Mem ria de dados.

```


APÊNDICE F – instructionMemory.v

Código do arquivo instructionMemory.v

```

1  module instructionMemory (addy, clock, RAMOuput, reset);
2    input [9:0] addy;
3    input clock;
4    input reset;
5
6    output [31:0] RAMOuput;
7
8    integer firstClock = 0;
9
10   reg [31:0] instructionRAM [30:0];
11
12  always @ (addy or reset) begin
13    if (reset) begin
14      // Algoritmo a ser executado pelo processador
15
16      instructionRAM[0] = 32'b010110_00000_00000000000000000000; //out R[0]
17      instructionRAM[1] = 32'b010101_00000_00010_0000000000000000; //in R[2] <-
18      X
19      instructionRAM[2] = 32'b010110_00010_00000000000000000000; //out R
20      [2]
21      instructionRAM[3] = 32'b010101_00000_00001_0000000000000000; //in R[1] <-
22      0
23      instructionRAM[4] = 32'b010110_00001_00000000000000000000; //out R[1]
24      instructionRAM[5] = 32'b010101_00000_00011_0000000000000000; //in R[3] <-
25      2
26      instructionRAM[6] = 32'b010110_00011_00000000000000000000; //out R[3]
27      instructionRAM[7] = 32'b010101_00000_00110_0000000000000000; //in R[6] <-
28      5
29      instructionRAM[8] = 32'b010110_00110_00000000000000000000; //out R[6]
30
31      instructionRAM[9] = 32'b010000_00000_00010_0000000000000001; //sw do R[2]
32      no MEM[1]
33
34      instructionRAM[10] = 32'b001001_00001_00110_00000000001000; // beq R[1]
35      e R[6]
36      instructionRAM[11] = 32'b001101_00000_00010_0000000000000000; //inc R[2]
37      instructionRAM[12] = 32'b000110_00011_00010_00010_000000000000; //mul R[2]
38      = R[2] * R[3]
39      instructionRAM[13] = 32'b010110_00010_00000000000000000000; //out R[2]
40      instructionRAM[14] = 32'b001101_00000_00001_0000000000000000; //inc R[1]
41      instructionRAM[15] = 32'b010111_000000000000000000001010; //j INS[10]
42      instructionRAM[16] = 32'b001111_00000_00100_0000000000000001; //lw do MEM
43      [1] no R[4]
44
45      instructionRAM[17] = 32'b000001_00100_00010_00101_000000000000; //sub R[5]
46      = R[4] - R[2]
47      instructionRAM[18] = 32'b010110_00101_00000000000000000000; //out R[5]
48      instructionRAM[19] = 32'b011000_00101_00000000000000000000; // hlt
49
50    end
51  end
52

```

```
43 assign RAMOutput = instructionRAM[addr];  
44  
45 endmodule // Memória de instruções
```

APÊNDICE G – muxALUScr.v

Código do arquivo muxALUScr.v

```
1 module muxALUScr (data2, extenderOutputA, out, cu_aluScr);
2   input cu_aluScr;
3   input [31:0] data2;
4   input [31:0] extenderOutputA;
5
6   output reg [31:0] out;
7
8   always @ ( * ) begin
9     if(cu_aluScr)
10       out = extenderOutputA;
11     else
12       out = data2;
13   end
14 endmodule // muxALUScr
```


APÊNDICE H – muxInSignal.v

Código do arquivo muxInSignal.v

```
1 module muxInSignal (immediate, switches, out, cu_inSignal);
2   input cu_inSignal;
3   input [15:0] immediate;
4   input [15:0] switches;
5
6   output reg [15:0] out;
7
8   always @ ( * ) begin
9     if(cu_inSignal)
10       out = switches;
11     else
12       out = immediate;
13   end
14 endmodule // muxInSignal
```


APÊNDICE I – muxJump.v

Código do arquivo muxJump.v

```
1 module muxJump (muxPCScr, signExtenderJ, out, cu_Jump);
2   input cu_Jump;
3   input [31:0] muxPCScr;
4   input [31:0] signExtenderJ;
5   output reg [31:0] out;
6
7   always @ ( * ) begin
8     if(cu_Jump)
9       out = signExtenderJ;
10    else
11      out = muxPCScr;
12  end
13 endmodule // muxJumpcu_jump
```


APÊNDICE J – muxMemtoReg.v

Código do arquivo muxMemtoReg.v

```
1 module muxMemtoReg (aluOut, dataRAMOutput, out, cu_memtoReg);
2   input cu_memtoReg;
3   input [31:0] aluOut;
4   input [31:0] dataRAMOutput;
5
6   output reg [31:0] out;
7
8   always @ ( * ) begin
9     if(cu_memtoReg)
10       out = dataRAMOutput;
11     else
12       out = aluOut;
13   end
14 endmodule // muxMemtoReg
```


APÊNDICE K – muxPCScr.v

Código do arquivo muxPCScr.v

```
1 module muxPCScr (extenderOutputA, outAddy, out, pcScr);
2   input pcScr;
3   input [15:0] outAddy;
4   input [31:0] extenderOutputA;
5
6   output reg [31:0] out;
7
8   always @ ( * ) begin
9     if(pcScr)
10       out = extenderOutputA;
11     else
12       out = outAddy + 1;
13   end
14
15 endmodule // muxPCScr
```


APÊNDICE L – muxRegDest.v

Código do arquivo muxRegDest.v

```
1 module muxRegDest (readAddy2, writeAddy, out, cu_regDest);
2   input cu_regDest;
3   input [4:0] readAddy2;
4   input [4:0] writeAddy;
5
6   output reg [4:0] out;
7
8   always @ ( * ) begin
9     if(cu_regDest)
10       out = readAddy2;
11     else
12       out = writeAddy;
13   end
14 endmodule // muxRegDest
```


APÊNDICE M – programCounter.v

Código do arquivo programCounter.v

```
1 module programCounter (inAddy , outAddy , hlt , clock , reset);
2   input clock , reset;
3   input [15:0] inAddy;
4   input hlt;
5   reg [15:0] novo;
6   output reg [15:0] outAddy;
7
8   always @ ( * ) begin
9     novo = inAddy;
10    end
11
12  always @ (posedge clock) begin
13    if(reset)begin
14      outAddy = 0;
15    end // N o faz nada.
16    else if(hlt) begin
17      // N o faz nada
18      end
19    else
20      outAddy = novo;
21    end
22
23 endmodule // programCounter
```


APÊNDICE N – signExtenderJ.v

Código do arquivo signExtenderJ.v

```

1 module signExtenderJ (inputB, extenderOutputB);
2   input [25:0] inputB; // Extensor de entradas de 21 bits.
3   output reg [31:0] extenderOutputB;
4
5   always @ (*) begin // Multiplexador usado para avaliar que tipo de extensor o "signExtender" deve fazer.
6     // Caso a seleção seja "01" ele completa uma entrada de 25 bits. Instrução j.
7     extenderOutputB = inputB;
8     if(inputB[25]) begin
9       extenderOutputB = {{6{1'b1}}, inputB};
10      end
11    else begin
12      extenderOutputB = {{6{1'b0}}, inputB};
13      end
14    end
15 endmodule // signExtenderJ

```


APÊNDICE O – signExtenderR.v

Código do arquivo signExtenderR.v

```

1 module signExtenderR (inputA, extenderOutputA);
2   input [15:0] inputA; //Extensor de entradas de 16 bits.
3   output reg [31:0] extenderOutputA;
4
5   always @ (*) begin //Multiplexador usado para avaliar que tipo de extensor o "signExtender" deve fazer.
6     //Caso a seleção seja "00" ele completa uma entrada de 16 bits. Instrução r.
7     extenderOutputA = inputA;
8     if(inputA[15]) begin
9       extenderOutputA = {{16{1'b1}}, inputA};
10      end
11    else begin
12      extenderOutputA = {{16{1'b0}}, inputA};
13      end
14    end
15
16 endmodule // Extensor de bits.

```


APÊNDICE P – bbtronenhancedCPU.v

Código do arquivo bbtronenhancedCPU.v

```

1 module bbtronenhancedCPU (clock, auto_clock, switches, wire_out1, wire_out2, wire_out3,
2   wire_negative, reset);
3   //Entradas: Clock e Switches
4   input clock;
5   input [15:0] switches;
6   input auto_clock;
7   input reset;
8
9   //Wires para a Control Unity.
10  wire wire_cu_writeReg;
11  wire wire_cu_regDest;
12  wire wire_cu_memtoReg;
13  wire wire_cu_Jump;
14  wire wire_cu_inSignal;
15  wire wire_cu_aluScr;
16  wire wire_cu_writeEnable;
17  wire wire_cu_readEnable;
18  wire wire_cu_Branch;
19  wire [3:0] wire_cu_aluOp;
20  wire wire_cu_hlt;
21  wire wire_cu_reset;
22  wire wire_cu_showDisplay;
23
24  //Wires para "Program Counter".
25  wire [15:0] wire_outAddy;
26
27  //wires para "Instruction Memory".
28  wire [31:0] wire_RAMOutput;
29
30  //Wires para "Signal Extender R".
31  wire [31:0] wire_extenderOutputA;
32
33  //Wires para "Signal Extender J".
34  wire [31:0] wire_extenderOutputB;
35
36  //Wires para "Register Bench".
37  wire [31:0] wire_data1;
38  wire [31:0] wire_data2;
39  wire [31:0] wire_data3;
40
41  //Wires para "Data Memory".
42  wire [31:0] wire_dataRAMOutput;
43
44  //Wires para "ALU".
45  wire [31:0] wire_aluOut;
46  wire wire_negative;
47  wire wire_zero;
48
49  //Wires para "Out Module"
50  output wire [6:0] wire_out1;
51  output wire [6:0] wire_out2;
52  output wire [6:0] wire_out3;

```

```

52     output wire wire_negative;
53
54 //Wires da "DeBouncer"
55 wire wire_DB_Out;
56
57 //Wires para "Register Destination MUX".
58 wire [4:0] wire_out_regdest;
59
60 //Wires para "Memory to Register MUX".
61 wire [31:0] wire_out_memtoreg;
62
63 //Wires para "Jump MUX".
64 wire [31:0] wire_out_jump;
65
66 //Wires para "In Signal MUX".
67 wire [15:0] wire_out_insignal;
68
69 //Wires para "ALU Source MUX".
70 wire [31:0] wire_out_aluscr;
71
72 //Wires para "And MUX".
73 wire wire_out_andmux;
74
75 //Wires para "PC Source MUX".
76 wire [31:0] wire_out_pcsrc;
77
78 //Instancia dos módulos.
79
80 //Program Counter_OK_VERIFICAR SE ESTA CORRETO
81 programCounter inst_programCounter(.inAddy(wire_out_jump [15:0]), .outAddy(wire_outAddy
    [15:0]), .hlt(wire_cu_hlt), .clock(wire_DB_Out), .reset(wire_cu_reset));
82
83 //Instruction Memory_OK_TESTAR SE ESTA CORRETO
84 instructionMemory inst_instructionMemory(.addy(wire_outAddy [15:0]), .clock(wire_DB_Out
    ), .RAMOutput(wire_RAMOutput [31:0]));
85
86 //Signal Extender R_OK
87 signExtenderR inst_signExtenderR(.inputA(wire_out_insignal [15:0]), .extenderOutputA(
    wire_extenderOutputA [31:0]));
88
89 //Signal Extender J_OK
90 signExtenderJ inst_signExtenderJ(.inputB(wire_RAMOutput [25:0]), .extenderOutputB(
    wire_extenderOutputB [31:0]));
91
92 //Register Bench_OK
93 registerBench inst_registerBench(.readAddy1(wire_RAMOutput [25:21]), .readAddy2(
    wire_RAMOutput [20:16]), .writeAddy(wire_out_regdest [4:0]), .writeData(
    wire_out_memtoreg [31:0]), .data1(wire_data1 [31:0]), .data2(wire_data2 [31:0]), .
    data3(wire_data3 [31:0]), .clock(wire_DB_Out), .cu_writeReg(wire_cu_writeReg));
94
95 //Control Unity_OK
96 controlUnity inst_controlUnity(.opcode(wire_RAMOutput [31:26]), .cu_writeReg(
    wire_cu_writeReg), .cu_RegDest(wire_cu_RegDest), .cu_memtoReg(wire_cu_memtoReg), .
    cu_Jump(wire_cu_Jump), .cu_inSignal(wire_cu_inSignal), .cu_aluScr(wire_cu_aluScr),
    .cu_writeEnable(wire_cu_writeEnable), .cu_readEnable(wire_cu_readEnable), .
    cu_Branch(wire_cu_Branch), .cu_aluOp(wire_cu_aluOp [3:0]), .cu_hlt(wire_cu_hlt), .
    cu_reset(wire_cu_reset), .cu_showDisplay(wire_cu_show_Display));
97
98 //Data Memory_OK

```

```

99   dataMemory inst_dataMemory(.memoryAddy(wire_aluOut [31:0]), .writeData(wire_data2
100    [31:0]), .cu_writeEnable(wire_cu_writeEnable), .clock(wire_DB_Out), .dataRAMOutput(
101      wire_dataRAMOutput [31:0]), .cu_readEnable(wire_cu_readEnable));
102
103 //ALU_FALTA NEGATIVE
104 ALU inst_ALU(.data1(wire_data1 [31:0]), .data2(wire_out_aluscr [31:0]), .cu_aluOp(
105    wire_cu_aluOp [3:0]), .zero(wire_zero), .shamt(wire_RAMOutput [10:6]), .aluOut(
106      wire_aluOut [31:0]));
107
108 //Out Module
109 outModule inst_outModule(.in(wire_aluOut [31:0]), .out1(wire_out1 [6:0]), .out2(
110    wire_out2 [6:0]), .out3(wire_out3 [6:0]), .negative(wire_negative), .cu_showDisplay(
111      wire_cu_show_Display));
112
113 //Debouncer
114 deBouncer inst_deBouncer(.clk(auto_clock), .n_reset(1), .button_in(~clock), .DB_out(
115      wire_DB_Out));
116
117 //MUXs
118 //Register Destination MUX_OK
119 muxRegDest inst_muxRegDest(.readAddy2(wire_RAMOutput [20:16]), .writeAddy(
120    wire_RAMOutput [15:11]), .out(wire_out_regdest [4:0]), .cu_regDest(wire_cu_regDest)
121    );
122
123 //Memory to Register MUX_OK
124 muxMemtoReg inst_muxMemtoReg(.aluOut(wire_aluOut [31:0]), .dataRAMOutput(
125    wire_dataRAMOutput [31:0]), .out(wire_out_memtoreg [31:0]), .cu_memtoReg(
126      wire_cu_memtoReg));
127
128 //Jump MUX_OK
129 muxJump inst_muxJump(.muxPCScr(wire_out_pcsrc [31:0]), .signExtenderJ(
130    wire_extenderOutputB [31:0]), .out(wire_out_jump[31:0]), .cu_Jump(wire_cu_Jump));
131
132 //In Signal MUX_OK
133 muxInSignal inst_muxInSignal(.immediate(wire_RAMOutput [15:0]), .switches(switches), .
134     out(wire_out_insignal [15:0]), .cu_inSignal(wire_cu_inSignal));
135
136 //ALU Source MUX_OK
137 muxALUScr inst_muxALUScr(.data2(wire_data2 [31:0]), .extenderOutputA(
138    wire_extenderOutputA [31:0]), .out(wire_out_aluscr [31:0]), .cu_aluScr(
139      wire_cu_aluScr));
140
141 //And MUX_OK
142 andmux inst_andmux(.zero(wire_zero), .cu_Branch(wire_cu_Branch), .pcScr(wire_out_andmux),
143    ), .opcode(wire_RAMOutput [31:26]));
144
145 //PC Source MUX_OK
146 muxPCScr inst_muxPCScr(.extenderOutputA(wire_extenderOutputA [31:0]), .outAddy(
147    wire_outAddy [15:0]), .out(wire_out_pcsrc [31:0]), .pcScr(wire_out_andmux));
148
149
150 endmodule // bbtron_enhanced

```


APÊNDICE Q – BinToBCD2.v

Código do arquivo BinToBCD2.v

```

1 //Conversor de Binario para BCD (Display de 7 segmentos)
2 module BinToBCD2(binario, dezena, unidade);
3   input [6:0] binario;
4   output reg [3:0] dezena, unidade;
5   reg [3:0] centena;
6   integer i;
7
8   always@(binario) begin
9     centena = 4'D0;
10    dezena = 4'D0;
11    unidade = 4'D0;
12
13    for(i = 6; i>=0; i=i-1) begin
14      if(centena >= 5)
15        centena = centena + 3;
16      if (dezena >= 5)
17        dezena = dezena + 3;
18      if (unidade >= 5)
19        unidade = unidade + 3;
20
21      centena = centena << 1;
22      centena[0] = dezena[3];
23
24      dezena = dezena << 1;
25      dezena[0] = unidade[3];
26
27      unidade = unidade << 1;
28      unidade[0] = binario[i];
29    end
30  end
31 endmodule

```


APÊNDICE R – outModule.v

Código do arquivo outModule.v

```

1 module outModule(in, out1, out2, out3, negative, cu_showDisplay);
2   input [32:0] in;
3   input cu_showDisplay;
4   wire [3:0] centena, dezena, unidade;
5
6   output wire negative;
7   output wire [6:0] out1, out2, out3;
8
9
10  binToBCD(.in(in [7:0]), .centena(centena), .dezena(dezena), .unidade(
11    unidade));
12
13  assign negative = (in[7] && cu_showDisplay) ? 0 : 1;
14
15  display d1(.Entrada(centena), .Saida(out1), .cu_showDisplay(
16    cu_showDisplay));
17  display d2(.Entrada(dezena), .Saida(out2), .cu_showDisplay(cu_showDisplay
18    ));
19  display d3(.Entrada(unidade), .Saida(out3), .cu_showDisplay(
19    cu_showDisplay));
20
21 endmodule

```


APÊNDICE S – display.v

Código do arquivo display.v

```

1 module display (Entrada, Saida, cu_showDisplay) ;
2     input [3:0] Entrada;
3     output reg [6:0] Saida;
4     input cu_showDisplay;
5
6     always@( * )
7         begin
8             if(cu_showDisplay)begin
9                 case (Entrada)
10                     4'b0000 : Saida = 7'b0000001 ;
11                     4'b0001 : Saida = 7'b1001111 ;
12                     4'b0010 : Saida = 7'b0010010 ;
13                     4'b0011 : Saida = 7'b0000110 ;
14                     4'b0100 : Saida = 7'b1001100 ;
15                     4'b0101 : Saida = 7'b0100100 ;
16                     4'b0110 : Saida = 7'b0100000 ;
17                     4'b0111 : Saida = 7'b0001111 ;
18                     4'b1000 : Saida = 7'b0000000 ;
19                     4'b1001 : Saida = 7'b00001100 ;
20                     default : Saida = 7'b1111111 ;
21             endcase
22         end
23         else
24             Saida = 7'b1111111 ;
25     end
26 endmodule

```


Anexos

ANEXO A – Anexo 1

ANEXO B – Anexo 2