UNIVERSIDADE FEDERAL DE SÃO PAULO

INSTITUTO DE CIÊNCIA E TECNOLOGIA DE SÃO JOSÉ DOS CAMPOS

Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores

Victor de Sá Nunes
RA: 92350
Julho/2016

Sumário

1. INTRODUÇÃO

Sistemas computacionais são amplamente empregados em diversas aplicações ao redor do mundo. Os computadores estão presentes em sistemas de segurança, sistemas médicos dentre outros. Um engenheiro de computação necessita ter habilidades para desenvolver sistemas que se apliquem a determinadas necessidades. Sendo assim, o objetivo deste trabalho é projetar e desenvolver um sistema computacional simplificado.

Este projeto faz parte de uma sequência de projetos presentes no curso Ciência e Tecnologia da Universidade Federal de São Paulo (UNIFESP) e é parte da disciplina Laboratório de Sistemas Computacionais: Arquitetura e Organização de Computadores oferecida na UNIFESP.

1.1. Objetivo geral

Esse projeto é e possui como objetivo desenvolver um sistema computacional composto por uma unidade de processamento, unidade de controle e interface de comunicação.

1.2. Objetivo específico

Neste relatório será apresentada a implementação do sistema computacional como produto final. O sistema é composto de uma unidade lógica e aritmética, banco de registradores, memória de dados, memória de instruções, extensor de bit, entrada e saída de dados e um contador de programas.

Para melhor desenvolvimento houve uma separação em 4 partes, denominadas pontos de checagem. O primeiro ponto consistiu em elaborar um conjunto de instruções e esboçar a arquitetura do processador juntamente com a memória e interface de comunicação. Já o segundo ponto consiste em implementar, utilizando a linguagem de descrição de hardware (HDL - Hardware Description Language) Verilog, a unidade de processamento, sendo este relatório pertencente a este ponto. O terceiro ponto consiste em implementar, também em Verilog, a unidade de controle. Por fim, a quarta e última parte consiste em apresentar o produto final completo, integrando as partes já desenvolvidas.

2. FUNDAMENTAÇÃO TEÓRICA

Para melhor entendimento do projeto é preciso compreender o que significa e como funciona algumas partes que serão implementadas futuramente bem como alguns conceitos importantes para a fundamentação do trabalho.

2.1. Unidade de processamento

A unidade de processamento (CPU - Central Processing Unit) é a entidade que irá realizar as instruções que serão lidas da memória de instruções. Essas instruções podem ser para somar dois números ou realizar uma operação lógica entre eles, por exemplo. Este é o cérebro do computador. Seu desempenho afeta diretamente o desempenho geral do computador. À seguir serão explicados um pouco mais algumas outras partes de um computador digital, entretanto, podemos visualizar sua organização na Figura 1.

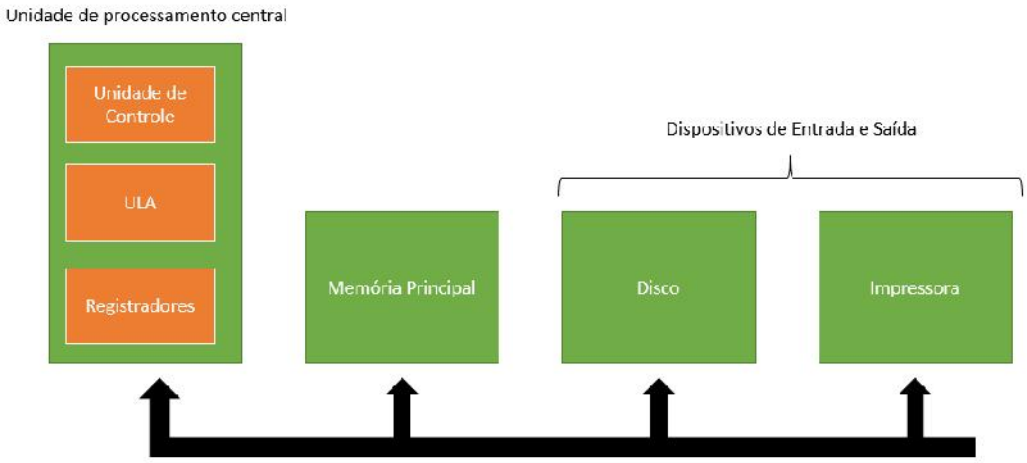

Figura 1. Organização de um computador digital

2.2. Unidade de controle

A unidade de controle (UC) é a entidade que irá gerenciar o fluxo de dados bem como o momento em que a transferência de dados irá ocorrer. Para isso, são necessários sinais que comandam uma microoperação (pode ser desde uma operação aritmética até o carregamento de um dado num registrador. Uma unidade de controle é uma máquina finita de estados (Finite State Machine - FSM) [3]. Sendo assim, quando uma instrução de 32 bits é informada é preciso informar quais componentes do sistema computacional serão utilizados assim como qual o comportamento que eles devem ter mediante a tal palavra.

2.3. Conjunto de instruções

Uma instrução indica qual a sequência de microoperações que o computador deve realizar. Ela é composta por uma sequência de bits que será decodificada. No projeto de um sistema computacional é necessário criar um conjunto de instruções que permita ao computador realizar uma série de tarefas. O ideal é que este conjunto seja capaz de executar qualquer algoritmo que seja passado para o sistema. Isto dá ao sistema a característica de portabilidade [1].

2.4. Memória

Memória é a parte do sistema que irá armazenar informações. Para melhor organização, ela é dividida em palavras. Cada palavra possui somente um endereço e essas palavras podem armazenar dados e instruções. O acesso à memória também varia conforme a arquitetura adotada. Ela pode ser feita de maneira sequencial, direta ou aleatório. No primeiro modo, os

dados são acessados segundo uma sequência linear específica [2]. Este projeto terá o acesso sequencial como uma maneira de obter dados da memória.

2.5. Modos de endereçamento

Uma instrução opera sobre operandos (elementos que serão usados. Por exemplo, dois números). Estes operandos podem estar em qualquer posição de memória ou em qualquer registrador. Sendo assim, é necessário passar o endereço destes operandos. E isto é passado junto da instrução. Pode haver casos em que seja necessário se passar o enderço da próxima instrução, ao invés do endereço do operando. Este é o caso de comandos de desvio como o if em C, ou Python, por exemplo. [3]. Os modos de endereçamento são vários, dentre eles o por registrador, acesso imediato e acesso direto. No primeiro modo, o registrador é acessado para que a informação contida nele seja recuperada. Assim, na instrução deve ser passado o número do registrado dentro do banco de registradores. No segundo modo, um valor é passado para se acessar a posicao na memória que corresponda aquele número. E no modo de acesso direto, o dado é escrito diretamente numa posição de memória.

2.6. CISC x RISC

Por um longo tempo a tendência na arquitetura dos computadores era de deixar a complexidade do processador maior, aumentando o conjunto de instruções, os modos de endereçamento, quantidade de registradores especializados dentre outros. Isso acontecia devido à vontade de se ter compiladores mais simples além e de melhoras no desempenho. De acordo com o conjunto de instruções, uma arquitetura é definidia como CISC (Complex Instruction Set Computer) ou RISC (Reduced Instruction Set Computer). Computadores RISC possuem um conjunto reduzido de instruções simples. Aqui há uma ênfase em referências a registradores, que por existerem em baixa quantidade, requerem um número menor de bits para serem endereçados. Já a arquitetura CISC contém mais instruções e de complexidade maior. Esta, por sua vez, enfatiza a referência à memória [2].

Algumas características da arquitetura RISC podem ser citadas, como feito por [2]:

- Uma instrução por ciclo de clock
- Operação de registrador para registrador
- Modos de endereçamento simples
- Formatos de instrução simples

2.7. MIPS

O MIPS (Microprocessor without interlocked pipeline stages) é um processador amplamente conhecido na literatura. Desenvolvido por Jhon Henessey em 1981, sua arquitetura é baseada em registradores, logo a CPU utiliza apenas registradores para realizar suas operaçoes lógicas e

aritmeticas. O MIPS trabalha com uma instrução por ciclo de clock e segue uma arquitetura RISC. Este projeto s erá baseado neste processador já desenvolvido [3].

3. DESENVOLVIMENTO DO TRABALHO

Foi definido um conjunto de instruções que contém 32 instruções. Para a elaboração do conjunto de instruções, foi escrito um algoritmo em C. À partir deste algoritmo ele foi reescrito em RTL (Register Transference Language - Linguagem de transferência de registradores, uma linguagem que permite explicitar os registradores e as operações entre eles). Com isso, foi possível visualizar quais instruções seriam necessárias para executar no mínimo aquele algoritmo. Algumas instruções extras foram colocadas a fim de facilitar o trabalho de um possível programador.

À seguir está a tabela que contém este conjunto bem como a operação realizada escrita em RTL.

Tabela 1. Conjunto de instruções

INSTRUÇÃO	OPERACAO	TIPO DE INSTRUÇÃO
ADD [RD, RS, RT]	RD <= RS + RT	ARITMÉTICA
ADDI [RD, RS, IM16]	RD <= RS + IM16	ARITMÉTICA
SUB [RD, RS, RT]	RD <= RS - RT	ARITMÉTICA
SUBI [RD, RS, IM16]	RD <= RS - IM16	ARITMÉTICA
AND [RD, RS, RT]	RD <= RS and RT	ARITMÉTICA
ANDI [RD, RS, IM16]	RD <= RS and IM16	ARITMÉTICA
OR [RD, RS, RT]	RD <= RS or RT	LÓGICA
ORI [RD, RS, IM16]	RD <= RS or IM16	LÓGICA
NOT [RD,RS]	RD <= not(RS)	LÓGICA
XOR [RD, RS, RT]	RD <= RS xor RT	LÓGICA
JUMPR [RD, RT]	PC <= RT	SALTO INCONDICIONAL
MOVE [RD, RS]	RD <= RS	TRANSFERÊNCIA
NOP	NENHUMA OPERACAO	CONTROLE
HALT	PARA PROCESSAMENTO	CONTROLE
SLT [RD, RS, RT]	RD <= 1 se RS < RT	SALTO CONDICIONAL
SLTI [RD, RS IM16]	RD <= 1 se RS < IM16	SALTO CONDICIONAL
BEQ [RS, RT, IM16]	se (RS == RT) go to IM16	SALTO CONDICIONAL
BNE [RS, RT, IM16]	se (RS != RT) go to IM16	SALTO CONDICIONAL
IN [RD]	RD <= Slide-Switches	E/S
OUT [RD, PORTA]	Saida_dados(PORTA) <= RD	E/S
LOAD [RD, ENDERECO]	RD <= RAM_dados[ENDERECO]	ACESSO A MEMÓRIA
LOADI [RD, IM22]	RD <=IM21	ACESSO A MEMÓRIA
STORE [RD, ENDERECO]	RAM_dados[ENDERECO] <= RD	ACESSO A MEMÓRIA
SLE[RD, RS, RT]	RD <= 1 se RS menor ou igual RT	SALTO CONDICIONAL
SLEI[RD, RS, IM16]	RD <= 1 se RS menor ou igual IM16	SALTO CONDICIONAL
SHE[RD, RS, RT]	RD <= 1 se RS >= RT	SALTO CONDICIONAL
SHEI[RD, RS, IM16]	RD <= 1 se RS >= IM16	SALTO CONDICIONAL
SHT[RD, RS, RT]	RD <= 1 se RS > RT	SALTO CONDICIONAL

SHTI[RD, RS, IM16]	RD <= 1 se RS > IM16	SALTO CONDICIONAL
SETI[RD, RS, IM16]	RD <= 1 se RS == IM16	SALTO CONDICIONAL
SET[RD, RS, RT]	RD <= 1 se RS == RT	SALTO CONDICIONAL

Após ter sido montado o conjunto foi necessário definir o formato das instruções. Nesta parte do projeto foi preciso definir a quantidade de bits para cada operando, sabendo que cada palavra conteria 32 bits. Como o conjunto de instruções possui 31 instruções são necessários 5 bits para dizer qual operação o processador deve realizar, entretanto, por motivos de precaução foram disponibilizados 6 bits para o opcode, caso esse conjunto necessite ser ampliado. Para os registradores também são necessários 5 bits pois o banco de registradores contém 32 registradores $(2^5 = 32)$. Os demais operandos possuem a quantidade restante de bits para serem endereçados. À seguir é explicado a distribuição de bits para cada formato de instrução criado.

Foram criados 6 tipos de formatos de instruções. Dessa forma as instruções criadas se encaixam em um desses tipos. Por exemplo, no formato 5 os operandos são: opcode, de 6 bits, registrador de destino, de 5 bits e um número imediato de 21 bits, totalizando 32 bits na palavra a ser passada.

Abaixo está a tabela com os formatos, operandos e a quantidade de bits para cada um, discriminada entre parênteses.

Tabela 2. Formato das instruções

FORMATO DAS INSTRUÇÕES
FORMATO 1: [OPCODE (31:26) | RD (25:21) | RS (20:16) | RT (15:11) | 0 (10:0)]
FORMATO 2: [OPCODE (31:26) | RD (25:21) | RS (20:16) | IM16 (15:0)]
FORMATO 3: [OPCODE (31:26) | RD (25:21) | IM21 (20:0)]

À seguir está a tabela com o significado de cada sigla presente na tabela.

Tabela 3. Significado das siglas contidas nos formatos das instruções

	Significado
OPCODE	Código da instrução
RD	Registrador destino - um valor será armazenado nele
RS	Registrador fonte - seu valor será utilizado na operação
RT	Registrador fonte - seu valor será utilizado na operação
0	Sequencia de zeros
ENDEREC	Endereço de memória
IM16	Imediato de 16 bits
IM21	Imediato de 21 bits

O formato 1 consiste de um campo para o opcode, outros 3 campos para 3 registradores. (dois para terem seus valores utilizados e outro para ter o resultado da instrução armazenado). No formato 2 é ligeiramente diferente do primeiro. Aqui, ao invés de um segundo registrador ser utilizado para obter o valor do operando, um valor imediato de 16 bits é passado. Já no terceiro formato, a instrução conterá um campo de opcode, outro de um registrador e outro de um valor de endereço de memória.

Além disso, os modos de endereçamento também precisaram ser definidos. Como o sistema

computacional está baseado no MIPS, os modos de endereçamento são: imediato, por registrador e direto. Isto pode ser verificado nos formatos das instruções. Nos formatos 1 e 6 o modo de endereçamento é por registrador. Nos formatos 2 e 5, por imediato. E por fim, no formato 3 o acesso é direto.

A unidade de controle (UC) pode ser pensada como uma máquina de estados finita. Cada estado será um dos opcodes. Sendo assim, ela será uma máquina com 32 estados. Conforme um estado é passado para ela, a UC determina qual caminho os bits devem seguir no caminho de dados, mostrado na Figura 2.

O caminho de dados do sistema computacional em questão pode ser visto na figura à seguir.

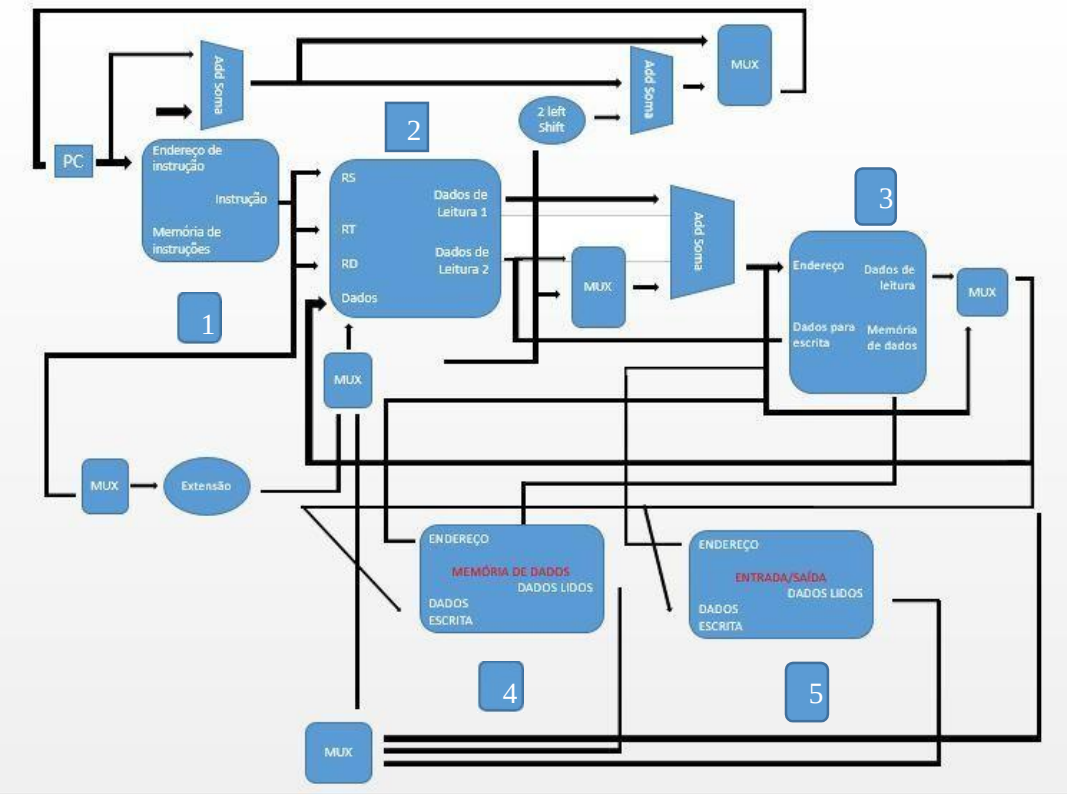

Figura 2. Caminho de dados do sistema projetado.

Como é possível verificar, na Figura 2 existem alguns marcadores numéricos. Eles estão apontando para algumas partes do caminho de dados. O “1” é a parte responsável pela busca da instrução na memória de instruções. Neste momento, a palavra é separa entre seus operandos. Em “2”, a palavra foi dividida entre seus operandos os dados para leitura serão processados na Unidade Lógica e Aritmética (ULA) descrita por Add Soma na figura. O resultado é então passado para “3”, onde esse endereço pode ser usado para salvar o resultado na memória de dados (“4”) como também para ser enviado para a interface de entrada e saída (“5”). Os MUX servem apenas para decidir qual caminho os bits devem tomar.

Este sistema computacional conterá uma interface de comunicação. Na Figura 1 ela é representada pela caixa escrita “ENTRADA/SAÍDA”. Como a implementação será feita no FPGA do modelo Altera DE2-115, os dados serão exibidos nos displays de sete segmentos. A entrada de dados poderá ser feita pelos pinos contidos na placa.

A memória do sistema projetado conterá um banco de registradores (BR). “ Um BR é uma coleção de registradores em que qualquer registrador pode ser lido ou escrito especificando o número do registrador no

banco.” [1]. Além disso, conterá também uma memória de dados, que armazenará os resultados das instruções; e uma memória de instruções, que guardará as instruções. Quando o contador de programa (indicado por PC na figura 1), for processado pela ULA, ele indicará qual o endereço na memória de instruções que está a próxima instrução.

Como já citado aqui foram implementados alguns componentes do sistema computacional. À seguir eles serão apresentados e mais detalhados.

3.1 – Unidade lógica e aritmética

A unidade lógica e aritmética (ALU) de um computador é a parte responsável por executar todas as operações de cunho matemático, sendo tanto operações aritméticas (soma, multiplicação, por exemplo) como operações lógicas, como uma operação AND. Logo, foi implementado um módulo que pudesse realizar as operações referentes à lógica e à aritmética. No conjunto de instruções, exibido na Tabela 1 na coluna “TIPOS”, é possível identificar quais operações serão realizadas na ALU. O Algoritmo 1 exibe o código em Verilog da ALU.

Algoritmo 1: Unidade de controle

```
module alu(num1, num2, operacao, resultado, sinal);
	input [31:0] num1, num2;
	input [4:0] operacao;
	output reg [31:0] resultado;
	output sinal;

	always @(num1 or num2) begin
		case(operacao)
			5'd0: resultado = num1 + num2; //soma
			5'd1: resultado = num1 - num2; //sub
			5'd2: resultado = num1 & num2; //and
			5'd3: resultado = num1 | num2; //or
			5'd4: resultado = ~num1;       //not
			5'd5: resultado = num1 ^ num2; //xor
			5'd6: resultado = num1 < num2 ? 1 : 0; //set less than
			5'd7: resultado = num1 <= num2 ? 1 : 0; //set less or equal
			5'd8: resultado = num1 >= num2 ? 1 : 0; //set higher or equal
			5'd9: resultado = num1 > num2 ? 1 : 0; //set higher than
			5'd10: resultado = num1 == num2 ? 1 : 0; //set equal to
			5'd11: resultado = num1 != num2 ? 1 : 0; //set not equal
			default: resultado = 32'dX;
		endcase
	end

	assign sinal = resultado[31]; //Guarda o sinal do resultado.

endmodule
```

O campo operação é um número de 4 bits que irá dizer para a ALU qual das operações que ela pode realizar deverá ser feita. Dentro do case é possível ver a identificação em binário de cada operação. O resultado da operação será armazenado numa variável de 32 bits para possa ser guardado na memória de dados.

A variável sinal guardará o sinal do resultado. Caso o número seja negativo, é preciso um

tratamento especial para exibí-lo no display de sete segmentos da placa. O sinal é obtido pelo bit mais significativo do resultado, já que o número está escrito na forma de complemento de 2.

3.2 – Memória de dados

A memória de dados foi implementada como um vetor em que cada posição tem tamanho 32 bits. Cada posição da memória guardará uma palavra, de 32 bits. Quanto maior o tamanho do vetor, mais dados a memória é capaz de armazenar, entretanto, para fins de testes, convém utilizar uma memória menor, já que o tempo para compilar o projeto cresce conforme aumenta- se seu tamanho. Na Figura 4 o código da memória de dados é exibido.

Algoritmo 2: Memória de dados

```
module ramDados(dado, endLinha, enable, clock, saida);
input[31:0] dado;
input[4:0] endLinha;
input enable, clock;
output [31:0] saida;

reg [31:0] ram_dados[31:0];

always @(posedge clock) begin
            if (enable) begin //Autoriza a escrita
                  ram_dados[endLinha] <= dado;
            end
      end

      //Leitura do dado eh constante
      assign saida = ram_dados[endLinha];
endmodule
```

O campo enable irá autorizar a escrita de algum dado na memória caso ele seja 1. Essa escrita, deverá ocorrer na subida do clock. A linha 7 contém a declaração da memória. Como já justificado, ela possui um tamanho menor do que o desejado.

A variável endLinha irá informar a posição no vetor ram_dados em que o dado será buscado ou escrito. A leitura de dados ocorre constantemente.

3.3 – Memória instruções

A memória de instruções se comporta semelhante à memória de dados com a diferença de que nela serão armazenadas as instruções para execução de um programa. As instruções que serão armazenadas são as que estão contidas na Tabela 1, que especifica o conjunto de instruções. O algoritmo 3 se refere ao código da memória de instruções em Verilog.

Algoritmo 3: Memória de instruções

```
module ramInstrucoes(endLinha, clock, saida);
      input [4:0] endLinha;
      input clock;
      output [31:0] saida;
      reg [31:0] memoriaInstrucoes [31:0]; //32 posicoes de 32 bits
      integer primeiroClock = 1;
```

```
	always @(posedge clock) begin

		if(primeiroClock == 1) begin
		//n primeiros termos de Fibonacci
		memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
		memoriaInstrucoes[1] <= {6'd21, 5'd0, 21'd0}; //IN R0
		memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd0}; //LOAD R1, 1 :primeiro
valor
		memoriaInstrucoes[3] <= {6'd24, 5'd2, 21'd1};  //LOAD R2, 0 :segundo
valor
		memoriaInstrucoes[4] <= {6'd24, 5'd3, 21'd0};  //LOAD R3, 0 :contador
		memoriaInstrucoes[5] <= {6'd0, 5'd4, 5'd2, 5'd1, 11'd0}; //ADD R4,
R2, R1
		memoriaInstrucoes[6] <= {6'd1, 5'd3, 5'd3, 16'd1}; //ADD I R3, 1
		memoriaInstrucoes[7] <= {6'd13, 5'd1, 5'd2, 16'd0}; //MOVE R1, R2
		memoriaInstrucoes[8] <= {6'd13, 5'd2, 5'd4, 16'd0}; //MOVE R2, R4
		memoriaInstrucoes[9] <= {6'd20, 5'd0, 5'd3, 16'd5}; //BNQ R0, R3, 4
		memoriaInstrucoes[10] <= {6'd25, 5'd4, 21'd0}; //STORE R4, 0
		memoriaInstrucoes[11] <= {6'd22, 5'd4, 21'd0}; //OUT R4
		memoriaInstrucoes[12] <= {6'd16, 26'd0};
		primeiroClock <= 0;
		end
	end

	assign saida = memoriaInstrucoes[endLinha];

endmodule
```

O código aqui é muito semelhante, a diferença é que não contém um habilitador para escrita. Sempre na subida do clock, ocorrerá a escrita da instrução em uma determinada posição do vetor. Como mostrado na figura 5, houve a escrita de 4 instruções na memória.

Novamente, por questões de tempo gasto na compilação, a memória de instruções foi limitada a 32 posições. Posteriormente, esse valor será aumentado, visando armazenar mais instruções.

3.4 – Banco de registradores

O banco de registradores consiste em uma memória também. Na Tabela 1, na coluna "OPERAÇÃO" é explicado o funcionamento de cada instrução. A grande maioria trabalha com registradores (denominados por RD, RS ou RT). Sendo que cada registrador irá possuir uma função podendo ser usado para leitura de dado ou para armazenamento. Portanto, estes comportamentos precisam ser tratados. O Algoritmo 6 está o código em Verilog do banco de registradores.

Algoritmo 4: Banco de registradores

```
module bancoDeRegistradores(regEscrita, regLeitura1, regLeitura2,
                            valorEscrita, valorLeitura1, valorLeitura2,
                            enable, entradaUC, clock, enter);

	input [4:0] regEscrita; //endereco de escrita RD
	input [4:0] regLeitura1; //endereco de leitura RS
	input [4:0] regLeitura2; //endereco de leitura RT
```

```
	input clock, entradaUC;
	input enable, enter;
	input [31:0] valorEscrita;
	output [31:0] valorLeitura1;
	output [31:0] valorLeitura2; //valor armazenado nos registradores que sera
repassado para a ALU

	reg [31:0] registradores[31:0]; //Banco de registradores. 32 registradores
de 32 bits cada

	assign valorLeitura1 = registradores[regLeitura1];
	assign valorLeitura2 = registradores[regLeitura2];

	always @(posedge clock) begin
		if(enable) //Habilita a escrita no registrador
			begin
				registradores[regEscrita] = valorEscrita;
			end
	end

endmodule
```

O banco de registradores possuirá 32 registradores de 32 bits cada. Para serem endereçados, é preciso um número de 5 bits. Esta é a função dos campos regEscrita, regLeitura1, regLeitura2. O regEscrita será o endereço em que uma informação será armazenada, escrita no banco de registradores. O campo enable habilita a escrita caso esteja em 1. E isto ocorrerá na subida do clock. Como os dados armazenados são de 32 bits, os valores a serem lidos e o valor a ser escrito necessita ter 32 bits também. Logo, os campos valorEscrita, valorLeitura1 e valorLeitura2 possuem essa função. Na linha 17 ocorre a escrita de um valor. Nas linhas 21 e 22 ocorrem a leitura de dois valores contidos no banco.

3.5 – Contador de programa (PC)

Quando uma instrução é lida é necessário buscar a próxima instrução do programa. Como explicado na seção 3.3, a memória de instruções foi projetada na forma de um vetor. Portanto, a próxima instrução a ser lida deve ser buscada na posição seguinte à da instrução atual. Tendo isto em vista o PC calcula e retorna o endereço da próxima instrução. À seguir está o código em Verilog do PC.

Algoritmo 5: Contador de programa

```
1  module PC(clock, endLinhaAtual, endLinhaNova, flag, reset, saida);
2      input clock, reset;
3      input [4:0] endLinhaAtual, endLinhaNova;
4      input [1:0] flag; //Vai indicar se eh jump, nop, ou incremento normal
5      output reg [4:0] saida; //Novo endereco de memoria
6
7      always @(posedge clock or posedge reset) begin
8          if(reset == 1) begin
9              saida = 5'd0;
10         end
11
12         else begin
```

```
13                case (flag)
14                      2'b00: saida <= endLinhaAtual + 5'b00001; //incremento
normal
15                      2'b01: saida <= endLinhaNova; //jump
16                      2'b10: saida <= saida;
17                      2'b11: saida <= saida; //nop
18                      default: saida <= saida + 5'd1;
19                endcase
20          end
21    end
22
23 endmodule
```

O novo endereço é calculado na linha 14. O campo saída é um número de 5 bits que será um endereço de memória. Isto se deve ao fato da palavra ter 32 bits, como explicado na seção

Além de calcular o novo endereço, o PC deve ser capaz de calcular o endereço de uma instrução que executa um salto, como a instrução JUMP, presente no conjunto de instruções. Existe também a possibilidade de uma instrução não realizar nenhuma operação. Essa é a instrução, NOP. Neste caso, o próximo endereço de memória é equivalente ao endereço atual. Isso é feito na linha 15. E para identificar qual o tipo de operação a ser realizada existe um campo flag que irá determinar isto. Caso o campo reset seja igual a 1, o contador de programa zera e volta para a primeira posição.

3.6 – Extensor de bit

As palavras que serão passadas para a memória de dados, banco de registradores ou memória de instruções possuem 32 bits de tamanho. Como alguns números a serem armazenados possuem menos do que 32 bits, como na instrução ADDI, em que um imediato de 16 bits é passado, é preciso completar o número com 0's ou 1's para se ter um número com 32 dígitos em binário. Este é o papel do extensor de bit. No algoritmo 6 está presente o código em Verilog do extensor de bit.

Algoritmo 6: Contador de programa

```
module muxExtensor(extendeImediato, imediato16, imediato21, saida);
      input [1:0] extendeImediato;
      input [15:0] imediato16;
      input [20:0] imediato21;
      output reg [31:0] saida;
      reg [31:0] im16_ext, im21_ext;
      //integer flagExtensor = 1;

      always @(*) begin
            im16_ext = imediato16; //Caso em que o numero passado eh de 16 bits
            //Checa se o numero eh negativo e completa com 1
            if(imediato16[15] == 1'b1)
                  im16_ext = im16_ext + 32'b11111111111111110000000000000000;
                  else im16_ext = im16_ext +
32'b00000000000000000000000000000000;
                  im21_ext = imediato21; //Caso em que o numero passado eh de 21
bits
                  //Checa se o numero eh negativo e completa com 1
```

```
                  if(imediato21[20] == 1'b1)
                        im21_ext = im21_ext +
32'b11111111111000000000000000000000;
                  else im21_ext = im21_ext + 32'd0;
            //Escolhe qual imediato sera tido como output
            case (extendeImediato)
                  2'd0: saida = im16_ext;
                  2'd1: saida = im21_ext;
                  2'd2: saida = saida;
            endcase
      end
endmodule
```

Ele funciona como um multiplexador que extende os bits e por meio do campo extendeImediato decide qual imediato, o de 16 bits ou o de 21 bits, será armazenado no campo saída. Os números negativos são representados em complemento de 1. Então, caso o número passado seja negativo, seu primeiro bit será 1. Então, os bits que faltam para completar 32 bits são preenchidos com 1.

3.7 – Unidade de controle

A unidade de controle é reponsável por atribuir valores às flags para os componentes do sistema computacional. Sendo assim, ela possui como flags os campos *enableEscritaMemoria*, *escreveRAM_DADOS*, *flagPC*, *zeraImediato*, *extendeImediato*, *operacaoALU e tipoInstrucao.*

À seguirse encontra o código em verilog da unidade de controle.

Algoritmo 7: Unidade de controle

```
module unidadeDeControle(clock, opcode,
                         tipoInstrucao, operacaoALU,
                         enableEscritaMemoria, extendeImediato,
                         zeraImediato, enter, entradaUC, flagPC,
                         escreveRAM_DADOS, reset,
                         escolheValor);

      input [5:0] opcode;
      input clock, reset, enter;

      //flags
      output reg enableEscritaMemoria; //habilitador de escrita no Banco de
Registradores
      output reg escreveRAM_DADOS; //habilitador de escrita na RAM
      output reg escolheValor; //
      output reg entradaUC;
      output reg [1:0] flagPC;
      output reg [1:0] extendeImediato; //0: extende 16, 1: extende21, 2: nao
extende
      output reg zeraImediato; //informa se algum imediato deve ser zerado
      output reg [4:0] operacaoALU; //codigo de operacao na ALU
      output reg [2:0] tipoInstrucao; //formato da instrucao
```

```
localparam inicia = 1'b1;
localparam execucao = 1'b1;

reg [2:0] estado_atual;
reg [2:0] proximo_estado;

always @(posedge clock) begin
      if(reset) estado_atual <= inicia;
      else estado_atual <= proximo_estado;
end

always @(*) begin
      proximo_estado = estado_atual;
      case(estado_atual)
            inicia: proximo_estado = execucao;
            execucao: proximo_estado = execucao;
      endcase
end

always @ (clock) begin
case(estado_atual)
      execucao: begin

      //zeraImediato = 1'b1;
            case(opcode)
                  //add
                  6'd0: begin
                        operacaoALU <= 5'd0;
                        enableEscritaMemoria <= 1'b1;
                        tipoInstrucao <= 3'd1;
                        flagPC <= 2'd0;
                        extendeImediato <= 2'd2;
                        zeraImediato <= 1'b1;
                        escreveRAM_DADOS <= 1'b0;
                        escolheValor <= 1'b1;
                        entradaUC <= 1'b0;

                  end
                  //add i
                  6'd1: begin
                        operacaoALU =5'd0;
                        tipoInstrucao =3'd2;
                        zeraImediato =1'b0;
                        enableEscritaMemoria = 1'b1;
                        flagPC =2'd0;
                        extendeImediato = 2'd0;
                        zeraImediato = 1'b0;
                        escreveRAM_DADOS = 1'b0;
                        escolheValor = 1'b1;
                        entradaUC = 1'b0;

                  end

                  //subtracao
                  6'd2: begin
                        operacaoALU <=5'd1;
                        enableEscritaMemoria <=1'b1;
                        tipoInstrucao <=3'd1;
```

```
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //sub i
    6'd3: begin
        operacaoALU <=5'd1;
        tipoInstrucao <=3'd2;
        zeraImediato <=1'b0;
        enableEscritaMemoria <=1'b1;
        flagPC <=2'd0;
        extendeImediato <= 2'd0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //and
    6'd4: begin
        operacaoALU <=5'd2;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //and i
    6'd5: begin
        operacaoALU <=5'd2;
        tipoInstrucao <=3'd2;
        zeraImediato <=1'b0;
        enableEscritaMemoria <=1'b1;
        flagPC <=2'd0;
        extendeImediato <= 2'd0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end
    //or
    6'd6: begin
        operacaoALU <=5'd3;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
```

```
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //or i
    6'd7: begin
        operacaoALU <=5'd3;
        tipoInstrucao <=3'd2;
        zeraImediato <=1'b0;
        enableEscritaMemoria <=1'b1;
        flagPC <=2'd0;
        extendeImediato <= 2'd0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //not
    6'd8: begin
        operacaoALU <=5'd4;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd2;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //xor
    6'd9: begin
        operacaoALU <=5'd5;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //jump r
    6'd11: begin
        operacaoALU <= 5'd20;
        enableEscritaMemoria <= 1'b0;
        tipoInstrucao <= 3'd1;
        flagPC <= 2'd1;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
```

```
        entradaUC <= 1'b0;

    end
                                    //move
    //Move funciona como um ADD RD, RS, 16'd0
    6'd13: begin
        operacaoALU <= 5'd0;
        enableEscritaMemoria <= 1'd1;
        tipoInstrucao <= 3'd2;
        flagPC <= 2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //nop
    6'd15: begin
        operacaoALU <= 5'd20;
        enableEscritaMemoria <= 1'b0;
        flagPC <= 2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //halt
    6'd16: begin
        operacaoALU <= 5'd20;
        enableEscritaMemoria <= 1'd0;
        tipoInstrucao <= 3'd3;
        flagPC <= 2'd3;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //set less than
    6'd17: begin
        operacaoALU <=5'd6;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end
```

```
//set less than i
6'd18: begin
      operacaoALU <=5'd6;
      tipoInstrucao <=3'd2;
      zeraImediato <=1'b0;
      enableEscritaMemoria <=1'b1;
      flagPC <=2'd0;
      extendeImediato <= 2'd0;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b1;
      entradaUC <= 1'b0;

end

//beq
6'd19: begin
      operacaoALU <= 5'd20;
      tipoInstrucao <= 3'd2;
      zeraImediato <= 1'b0;
      enableEscritaMemoria <= 1'b0;
      flagPC <= 2'd1;
      extendeImediato <=2'd2;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b1;
      entradaUC <= 1'b0;

end

//bne
6'd20: begin
      operacaoALU <= 5'd20;
      tipoInstrucao <= 3'd2;
      zeraImediato <= 1'b0;
      enableEscritaMemoria <= 1'b0;
      flagPC <= 2'd1;
      extendeImediato <=2'd2;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b1;
      entradaUC <= 1'b0;

end

//in
6'd21: begin
      operacaoALU <= 5'd20;
      tipoInstrucao <= 3'd3;
      zeraImediato <= 1'b0;
      enableEscritaMemoria <= 1'b1;
      extendeImediato <= 2'd1;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b1;
      entradaUC <= 1'b1;

      if(enter)
            flagPC = 2'd0;
      else if(!enter)
                        flagPC = 2'd3;
```

```
end

//output
6'd22: begin
      operacaoALU <= 5'd20;
      tipoInstrucao <= 3'd3;
      zeraImediato <= 1'b0;
      enableEscritaMemoria <= 1'b1;
      flagPC <= 2'd0;
      extendeImediato <= 2'd2;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b1;
      entradaUC <= 1'b0;
end

//load
6'd23: begin
      operacaoALU <=5'd20;
      enableEscritaMemoria <=1'b1;
      tipoInstrucao <=3'd3;
      flagPC <=2'd0;
      extendeImediato <= 2'd1;
      zeraImediato <= 1'b1;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b0;
      entradaUC <= 1'b0;
end

//load i
6'd24: begin
      operacaoALU <=5'd20;
      tipoInstrucao <=3'd3;
      enableEscritaMemoria <=1'b1;
      flagPC <=2'd0;
      extendeImediato <= 2'd1;
      zeraImediato <= 1'b0;
      escreveRAM_DADOS <= 1'b0;
      escolheValor <= 1'b0;
      entradaUC <= 1'b0;
end

//store
6'd25: begin
      operacaoALU <=5'd20;
      enableEscritaMemoria <=1'b1;
      tipoInstrucao <=3'd3;
      zeraImediato <=1'b0;
      flagPC <=2'd0;
      extendeImediato <= 2'd1;
      escreveRAM_DADOS <= 1'b1;
      entradaUC <= 1'b0;
end

//set less or equal
6'd26: begin
      operacaoALU <=5'd7;
      enableEscritaMemoria <=1'b1;
      tipoInstrucao <=3'd1;
```

```
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //set less or equal to i
    6'd27: begin
        operacaoALU <=5'd7;
        tipoInstrucao <=3'd2;
        zeraImediato <=1'b0;
        enableEscritaMemoria <=1'b1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //set higher or equal to
    6'd28: begin
        operacaoALU <=5'd8;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
        zeraImediato <= 1'b1;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //set higher or equal i
    6'd29: begin
        operacaoALU <=5'd8;
        tipoInstrucao <=3'd2;
        zeraImediato <=1'b0;
        enableEscritaMemoria <=1'b1;
        flagPC <=2'd0;
        extendeImediato <= 2'd0;
        escreveRAM_DADOS <= 1'b0;
        escolheValor <= 1'b1;
        entradaUC <= 1'b0;

    end

    //set higher than
    6'd30: begin
        operacaoALU <=5'd9;
        enableEscritaMemoria <=1'b1;
        tipoInstrucao <=3'd1;
        flagPC <=2'd0;
        extendeImediato <= 2'd2;
```

```
                        zeraImediato <= 1'b1;
                        escreveRAM_DADOS <= 1'b0;
                        escolheValor <= 1'b1;
                        entradaUC <= 1'b0;

                    end

                    //set higher than i
                    6'd31: begin
                        operacaoALU <=5'd9;
                        tipoInstrucao <=3'd2;
                        enableEscritaMemoria <=1'b1;
                        zeraImediato <=1'b0;
                        flagPC <=2'd0;
                        extendeImediato <= 2'd0;
                        escreveRAM_DADOS <= 1'b0;
                        escolheValor <= 1'b1;
                        entradaUC <= 1'b0;

                    end

                    //set equal to
                    6'd32: begin
                        operacaoALU <=5'd10;
                        enableEscritaMemoria <=1'b1;
                        tipoInstrucao <=3'd1;
                        flagPC <=2'd0;
                        extendeImediato <= 2'd2;
                        zeraImediato <= 1'b1;
                        escreveRAM_DADOS <= 1'b0;
                        escolheValor <= 1'b1;
                        entradaUC <= 1'b0;

                    end

                    //set equal to i
                    6'd33: begin
                        operacaoALU <=5'd10;
                        enableEscritaMemoria <=1'b1;
                        tipoInstrucao <=3'd2;
                        zeraImediato <=1'b0;
                        flagPC <=2'd0;
                        extendeImediato <= 2'd0;
                        escreveRAM_DADOS <= 1'b0;
                        escolheValor <= 1'b1;
                        entradaUC <= 1'b0;

                    end
                endcase
            end
        endcase
    end

endmodule
```

A Tabela 4 contém uma descrição da função de cada flag contida na unidade de controle.

Tabela 4. Descrição das funções das flags da unidade de controle.

Flag	Função
operacaoALU	Informa a ALU qual operação deve ser realizada
enableEscritaMemória	Habilita ou desabilita a escrita do valor no banco de registradores
tipoInstrucao	Diz qual o formato da instrução
zeraImediato	Define se deve ou não zerar o imediato presente nos formatos de instrução 2 e 3
flagPC	Define o comportamento do PC: se é incremento normal, jumo ou se ele deve permancer parado
extendeImediato	Define qual imediato deve ser extendido, o de 16 ou o de 21 bits. Diz também se nenhum dos dois imediatos deve ser extendido
escreveRAM_DADOS	Habilita ou não a escrita do dado na memória de dados
entradaUC	Informa se a instrução

3.7 – Entrada de dados

O módulo de entrada de dados terá a função de ler dados das chaves do FPGA. O número informado será de 10 bits, logo 10 chaves da placa serão necessárias para ocorrer a leitura. Além disso, uma chave de enter será necessário para informar que o dado já foi completamente inserido. À seguir está o código do módulo de entrada de dados.

Algoritmo 8: Entrada de dados

```
module entradaDeDadosSwitch (entradaSwitch,
                             dado,
                             enter,
                             escreveuBR,
                             entradaUC,
                             lendoSwitch);

//Entradas
      input [9:0]entradaSwitch;
      input enter; //Sinal do botao com debounce
      input escreveuBR;
      input entradaUC;

//Saidas
      output reg [31:0] dado;
      output reg lendoSwitch;

//Logica sequencial
      always @ (enter)
            begin
                  if (enter && entradaUC) begin
                              dado[9:0] <= entradaSwitch[9:0];
                              dado[31:9] <= entradaSwitch[9];
                              lendoSwitch <= 1'b1;
                  end

                  else if(lendoSwitch            == 1'b1 &&
                              escreveuBR               == 1'b0 &&
                              entradaUC == 1'b1) begin
```

```
                                lendoSwitch <= 1'b1;
                            end

                            else
                                  lendoSwitch <= 1'b0;
        end

endmodule
```

3.8 - Display de 7 segmentos

O display de 7 segmentos será utilizado para exibir os dados para o usuário. Cada segmento do display deve estar aceso ou apagado conforme o número. Devido ao funcionamento da placa FPGA, para acender um segmento este deve receber 0 e para apaga-lo, deve estar em 1. À seguir está o código em Verilog do display de 7 segmentos.

Algoritmo 9: Display de 7 segmentos

```
module display7seg (entrada, saida);
    input [3:0] entrada;
    output reg [6:0]saida;

    always @ (*) begin

                    case (entrada)
                    0:saida=7'b0000001;
                    1:saida=7'b1001111;
                    2:saida=7'b0010010;
                    3:saida=7'b0000110;
                    4:saida=7'b1001100;
                    5:saida=7'b0100100;
                    6:saida=7'b0100000;
                    7:saida=7'b0001111;
                    8:saida=7'b0000000;
                    9:saida=7'b0000100;
                    10:saida=7'b1111110; //sinal de menos
                    default:saida=7'b1111111; //display apagado
                    endcase
                end

endmodule
```

3.9 - Conversor para BCD

As operações podem gerar números com mais de um dígito. Quando isso acontece não é possível exibir o número nos displays de 7 segmentos. Para isso, é preciso separar um número em seus dígitos. O módulo de conversão de número binário para BCD faz isso. À seguir, está o código do conversor.

Algoritmo 10: Conversor de binário para BCD

```
module BCD (numBCD, saida_centena, saida_dezena, saida_sinal, saida_unidade,
```

```
                        entradaUnidade, entradaDezena,entradaCentena,
entradaSinal);
      input [31:0] numBCD;
      integer i;
      reg [9:0] numero;
      output reg [3:0] entradaUnidade, entradaDezena,entradaCentena,
entradaSinal;
      output [6:0] saida_centena, saida_dezena, saida_sinal, saida_unidade;

      display7seg display_unidade(entradaUnidade, saida_unidade);
      display7seg display_dezena(entradaDezena, saida_dezena);
      display7seg display_centena(entradaCentena, saida_centena);
      display7seg display_sinal(entradaSinal, saida_sinal);

      always @ (numBCD) begin
            entradaCentena = 4'd0;
            entradaDezena = 4'd0;
            entradaUnidade = 4'd0;

            if(numBCD == 10'b1111111111)begin //display apagado
                  entradaCentena = 4'd11;
                  entradaDezena = 4'd11;
                  entradaUnidade = 4'd11;
            end

            else begin
                  if (numBCD[31] == 1) begin //numBCDero negativo
                        numero = numBCD - 1;
                        numero = ~numero;
                        entradaSinal = 4'd10;
                  end
                  else if (numBCD[31] == 0) begin //numero positivo
                        entradaSinal = 4'd11;
                        numero = numBCD;
                  end

                  for (i=9; i>=0; i=i-1) begin
                        if (entradaCentena >= 5) begin
                              entradaCentena = entradaCentena + 3;
                              end
                        if (entradaDezena >= 5) begin
                              entradaDezena = entradaDezena +3;
                              end
                        if (entradaUnidade >= 5) begin
                              entradaUnidade = entradaUnidade +3;
                              end
                        entradaCentena = entradaCentena << 1;
                        entradaCentena[0] = entradaDezena[3];
                        entradaDezena = entradaDezena << 1;
                        entradaDezena[0] = entradaUnidade[3];
                        entradaUnidade = entradaUnidade << 1;
                        entradaUnidade[0] = numero[i];
                  end
            end

      end
endmodule
```

3.10 - Multiplexador para contador de programa

Cada instrução do conjunto de instruções comunica ao contador de programas se esse deve parar, incrementar ou ir para um endereço informado.

Algoritmo 11: muxFlagPC

```
module muxFlagPC(clock, opcode, enter, escreveuBR,
                                    flagPC, flagBranch, flagPC_final,
                                    entradaUC, lendoSwitch);

      input [5:0] opcode;
      input [1:0] flagPC;
      input clock, flagBranch, enter, escreveuBR, entradaUC, lendoSwitch;
      output reg[1:0] flagPC_final;

      always @(clock) begin
            case(opcode)
                  //jump r
                  6'd11: begin
                        flagPC_final <= flagPC;
                  end

                  //halt
                  6'd16: begin
                        flagPC_final <= 2'd2;
                  end
                  //beq
                  6'd19: begin
                        if(flagBranch) begin
                              flagPC_final <= flagPC; //realiza o jump
                        end
                        else
                              flagPC_final <= 2'd0; //incrementa posicao
                  end

                  //bnq
                  6'd20: begin
                        if(flagBranch) begin
                              flagPC_final <= flagPC; //realiza o jump
                        end
                        else
                              flagPC_final <= 2'd0; //incrementa posicao
                  end

                  //in
                  6'd21: begin
                              if(enter)
                                          flagPC_final <= 2'd0;
                              else
                                    flagPC_final <= 2'd2;
                  end

                  default: flagPC_final <= 2'd0;

            endcase
      end
```

```
endmodule
```

3.11 – Temporizador

O clock da placa FPGA tem uma frequência de clock de 50 Mhz. Caso este valor seja utilizado pode acontecer dos valores serem exibidos tão rapidamente sendo impossível de vê-los. Uma solução para isso é dividi a frequência do clock. Para isto foi implementado um temporizador que realize tal tarefa. O algoritmo 12 contém o código em Verilog do temporizador.

Algoritmo 12: Temporizador

```
module temporizador(clockin,
                                    clockout);

      input clockin;
      output wire clockout;

      parameter n = 10;
      reg [n:0] count;

      always @ (posedge clockin) begin
            count <= count + 1;
      end

      assign clockout = count[n];
endmodule
```

O valor do n pode variar. Caso a frequência desejada seja por volta de 1Hz, o valor de n deverá ser 26, pois log_{26}

3.12 – Sistema computacional

Após todos os módulos terem sido criados eles foram integrados em um módulo superior. O código deste módulo se encontra no Algoritmo 13.

Algoritmo 13: Sistema Computacional

```
module sistemaComputacional(clock50, reset, entradaSwitch, enter,
                            display_unidade, display_centena,
                            display_dezena, sinalBCD);

      temporizador temp(.clockin(clock50), .clockout(clock));

      PC pc1(.clock(clock),
                        .endLinhaAtual(endInstr),
                        .endLinhaNova(endLinhaNova),
                        .flag(flagPC),
                        .reset(reset),
                        .saida(endInstr));

      ramInstrucoes MI(.endLinha(endInstr),
                                        .clock(clock),
                                        .saida(palavra));

      unidadeDeControle UC(.clock(clock),
                                              .reset(reset),
                                              .opcode(opcode),
```

```
                                        .tipoInstrucao(tipoInstrucao),
                                        .operacaoALU(operacaoALU),

.enableEscritaMemoria(enableEscritaMemoria),

.extendeImediato(extendeImediato),
                                        .zeraImediato(zeraImediato),
                                        .flagPC(flagPC_aux),
                                        .entradaUC(entradaUC),

.escreveRAM_DADOS(escreveRAM_DADOS),
                                        .escolheValor(escolheValor));

muxExtensor mux_ext(.extendeImediato(extendeImediato),
                                .imediato16(imediato16),
                                      .imediato21(imediato21),
                                .saida(operando_ext));

muxFlagPC mux_PC(.clock(clock),
                                    .opcode(opcode),
                                    .enter(enter),
                                    .escreveuBR(escreveuBR),
                                    .flagPC(flagPC_aux),
                                    .flagBranch(flagBranch),
                                    .flagPC_final(flagPC));

bancoDeRegistradores BR(.regEscrita(RD),
                                                .regLeitura1(RS),
                                                .regLeitura2(RT),

.valorEscrita(valorEscrita),
                                                .valorLeitura1(num1),
                                                .valorLeitura2(num2),

.enable(enableEscritaMemoria),
                                                .clock(clock),
                                                .enter(enter),
                                                .entradaUC(entradaUC));

alu ULA(.num1(aux1),
                  .num2(aux2),
                  .operacao(operacaoALU),
                  .resultado(resultado),
                  .sinal(sinal));

ramDados MD(.dado(valorEscritaRAM),
                        .endLinha(endEscritaRAM),
                        .enable(escreveRAM_DADOS),
                        .clock(clock),
                        .saida(saidaRAM));

entradaDeDadosSwitch(.entradaSwitch(entradaSwitch),
                                        .dado(numEntrada),
                                        .enter(enter),
                                    .escreveuBR(escreveuBR),
                                    .entradaUC(entradaUC),
                                    .lendoSwitch(lendoSwitch));
```

```
	muxSaidaDeDados(.numBCD(numBCD),
                                    .imprime(imprime),
                                    .reset(reset),
                                    .numEntradaBCD(numEntradaBCD));

	BCD BCD1(.numBCD(numEntradaBCD),
                        .saida_centena(display_centena),
                        .saida_dezena(display_dezena),
                        .saida_unidade(display_unidade),
                        .entradaUnidade(entradaUnidade),
                        .entradaDezena(entradaDezena),
                        .entradaCentena(entradaCentena),
                        .entradaSinal(entradaSinal),
                        .saida_sinal(sinalBCD));

//Entradas
	input clock50, reset, enter;
	input [9:0] entradaSwitch;

//Saidas
	output wire [6:0] display_unidade;
	output wire [6:0] display_centena;
	output wire [6:0] display_dezena;
	output wire [6:0] sinalBCD;

//Fios
	wire enableEscritaMemoria;
	wire zeraImediato, escreveRAM_DADOS;
	wire escreveuBR, lendoSwitch, entradaUC;
	wire sinal;
	wire clock;
	wire [1:0] flagPC;
	wire [1:0] flagPC_aux;
	wire [1:0] extendeImediato;
	wire [2:0] tipoInstrucao;
	wire [4:0] endInstr;
	wire [4:0] operacaoALU;
	wire [4:0] endInstrPC;
	wire [31:0] palavra, numEntradaBCD;
	wire [31:0] operando_ext, resultado;
	wire [31:0] num1, num2, numEntrada;
	wire [31:0] saidaRAM;

//Regs
	reg flagBranch, imprime;
	reg  [4:0] RS, RD, RT, endEscritaRAM;
	reg  [4:0] endLinhaNova;
	reg  [4:0] endLinhaAtual;
	reg  [5:0] opcode;
	reg  [15:0] imediato16;
	reg  [20:0] imediato21;
	reg  [31:0] valorEscrita, valorEscritaRAM;
	reg  [31:0] aux1, aux2;
	reg  [31:0] numBCD;

//Separa os campos da instrucao
	always @(palavra) begin
		opcode <= palavra[31:26];
```

```
end

always @(opcode) begin
      endLinhaAtual <= endInstr;
      case (opcode)
            //ADD
            6'd0: begin
                  RD <= palavra[25:21];
                  RS <= palavra[20:16];
                  RT <= palavra[15:11];
                  aux1 <= num1;
                  aux2 <= num2;
                  valorEscrita <= resultado;
                  imprime <= 1'b0;
            end

            //ADD I
            6'd1: begin
                  RD <= palavra[25:21];
                  RS <= palavra[20:16];
                  imediato16 <= palavra[15:0];
                  aux1 <= num1;
                  aux2 <= operando_ext;
                  valorEscrita <= resultado;
                  imprime <= 1'b0;
            end

            //SUB
            6'd2: begin
                  RD <= palavra[25:21];
                  RS <= palavra[20:16];
                  RT <= palavra[15:11];
                  aux1 <= num1;
                  aux2 <= num2;
                  valorEscrita <= resultado;
                  imprime <= 1'b0;
            end

            //SUB I
            6'd3: begin
                  RD <= palavra[25:21];
                  RS <= palavra[20:16];
                  imediato16 <= palavra[15:0];
                  aux1 <= num1;
                  aux2 <= operando_ext;
                  valorEscrita <= resultado;
                  imprime <= 1'b0;
            end

            //and
                  6'd4: begin
                  RD <= palavra[25:21];
                  RS <= palavra[20:16];
                  RT <= palavra[15:11];
                  aux1 <= num1;
                  aux2 <= num2;
                  valorEscrita <= resultado;
                  imprime <= 1'b0;
```

```
        end

    //and i
        6'd5: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        imediato16 <= palavra[15:0];
        aux1 <= num1;
        aux2 <= operando_ext;
        valorEscrita <= resultado;
        imprime <= 1'b0;
        end

    //or
        6'd6: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        RT <= palavra[15:11];
        aux1 <= num1;
        aux2 <= num2;
        valorEscrita <= resultado;
        imprime <= 1'b0;
        end

    //or i
    6'd7: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        imediato16 <= palavra[15:0];
        aux1 <= num1;
        aux2 <= operando_ext;
        valorEscrita <= resultado;
        imprime <= 1'b0;
    end

    //not
    6'd8: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        aux1 <= num1;
        aux2 <= 32'd0;
        valorEscrita <= resultado;
        imprime <= 1'b0;
    end

    //xor
    6'd9: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        RT <= palavra[15:11];
        aux1 <= num1;
        aux2 <= num2;
        valorEscrita <= resultado;
        imprime <= 1'b0;
    end

    //jump r
    6'd11: begin
```

```
        RD <= palavra[25:21];
        endLinhaNova <= palavra[20:16];
        imprime <= 1'b0;
end

//move
//MOVE funciona como um ADD I RD, RS, 16'd0
6'd13: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        aux1 <= num1;
        valorEscrita <= aux1;
        imprime <= 1'b0;
end

//nop
6'd15: begin
        RD <= 5'd0;
        RT <= 5'd0;
        RS <= 5'd0;
        imediato16 <= 16'd0;
        imediato21 <= 21'd0;
        imprime <= 1'b0;
end

//halt
6'd16: begin
        RD <= 5'd0;
        RT <= 5'd0;
        RS <= 5'd0;
        imediato16 <= 16'd0;
        imediato21 <= 21'd0;
        aux1 <= 32'd0;
        aux2 <= 32'd0;
        valorEscrita <= 32'd0;
        imprime <= 1'b0;
end

//set less than
6'd17: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        RT <= palavra[15:11];
        aux1 <= num1;
        aux2 <= num2;
        valorEscrita <= resultado;
        imprime <= 1'b0; ;
end

//set less than i
6'd18: begin
        RD <= palavra[25:21];
        RS <= palavra[20:16];
        imediato16 <= palavra[15:0];
        aux1 <= num1;
        aux2 <= operando_ext;
        valorEscrita <= resultado;
        imprime <= 1'b0;
```

```
                end

                //beq
                6'd19: begin
                      RS <= palavra[25:21];
                      RT <= palavra[20:16];

                      aux1 <= num1;
                      aux2 <= num2;
                      imprime <= 1'b0;
                      if(aux1 == aux2) begin
                            endLinhaNova <= palavra[4:0]; //endereco da nova
posicao na memoria de instrucao
                            flagBranch <= 1'd1;
                      end

                      else flagBranch <= 1'd0;

                end

                //bne
                6'd20: begin
                      RS <= palavra[25:21];
                      RT <= palavra[20:16];
                      aux1 <= num1;
                      aux2 <= num2;
                      imprime <= 1'b0;
                      if(aux1 != aux2) begin
                            endLinhaNova <= palavra[4:0];
                            flagBranch <= 1'd1;
                      end

                      else flagBranch <= 1'd0;

                end

                //in
                6'd21: begin
                      RD <= palavra[25:21];
                      valorEscrita <= numEntrada;
                      imprime <= 1'b0;
                end

                //out
                6'd22: begin
                      RS <= palavra[25:21];
                      imprime <= 1'b1;
                      numBCD <= num1;

                end

                //load da memoria
                6'd23: begin
                      RD <= palavra[25:21];
                      endEscritaRAM <= palavra[4:0];
                      valorEscrita <= saidaRAM;
                      imprime <= 1'b0;
                end
```

```
//load i
6'd24: begin
      RD <= palavra[25:21];
      RS <= palavra[20:16];
      imediato21 <= palavra[20:0];
      valorEscrita <= operando_ext;
      imprime <= 1'b0;
end

//store
6'd25: begin
      RS <= palavra[25:21];
      endEscritaRAM <= palavra[4:0];
      valorEscritaRAM <= num1;
      imprime <= 1'b0;
end

//set less or equal
6'd26: begin
      RD <= palavra[25:21];
      RS <= palavra[20:16];
      RT <= palavra[15:11];
      aux1 <= num1;
      aux2 <= num2;
      valorEscrita <= resultado;
      imprime <= 1'b0;
end

//set less or equal to i
6'd27: begin
      RD <= palavra[25:21];
      RS <= palavra[20:16];
      imediato16 <= palavra[15:0];
      aux1 <= num1;
      aux2 <= operando_ext;
      valorEscrita <= resultado;
      imprime <= 1'b0;
end

//set higher or equal to
6'd28: begin
      RD <= palavra[25:21];
      RS <= palavra[20:16];
      RT <= palavra[15:11];
      aux1 <= num1;
      aux2 <= num2;
      valorEscrita <= resultado;
      imprime <= 1'b0;
end

//set higher or equal i
6'd29: begin
      RD <= palavra[25:21];
      RS <= palavra[20:16];
      imediato16 <= palavra[15:0];
      aux1 <= num1;
      aux2 <= operando_ext;
```

```
                        valorEscrita <= resultado;
                        imprime <= 1'b0;
                    end

                    //set higher than
                    6'd30: begin
                        RD <= palavra[25:21];
                        RS <= palavra[20:16];
                        RT <= palavra[15:11];
                        aux1 <= num1;
                        aux2 <= num2;
                        valorEscrita <= resultado;
                        imprime <= 1'b0;
                    end

                    //set higher than i
                    6'd31: begin
                        RD <= palavra[25:21];
                        RS <= palavra[20:16];
                        imediato16 <= palavra[15:0];
                        aux1 <= num1;
                        aux2 <= operando_ext;
                        valorEscrita <= resultado;
                        imprime <= 1'b0;
                    end

                    //set equal to
                    6'd32: begin
                        RD <= palavra[25:21];
                        RS <= palavra[20:16];
                        RT <= palavra[15:11];
                        aux1 <= num1;
                        aux2 <= num2;
                        valorEscrita <= resultado;
                        imprime <= 1'b0;
                    end

                    //set equal to i
                    6'd33: begin
                        RD <= palavra[25:21];
                        RS <= palavra[20:16];
                        imediato16 <= palavra[15:0];
                        aux1 <= num1;
                        aux2 <= operando_ext;
                        valorEscrita <= resultado;
                        imprime <= 1'b0;
                    end
                endcase
            end
        endmodule
```

Para cada instrução, todos os campos são separados para os módulos que estão conectados entre si.

4. RESULTADOS

Após as implementações dos módulos foram feitas simulações do sistema computacional completo. Essa seção irá mostrar os resultados obtidos e explicá-los.

4.1 – Simulação da unidade lógica e aritmética (ALU)

A ALU possui 11 operações em seu núcleo que são todas aquelas que dos tipos aritmética ou lógica. As Figuras 3 (a) e 3 (b) exibem as simulações das operações que a ALU pode realizar.

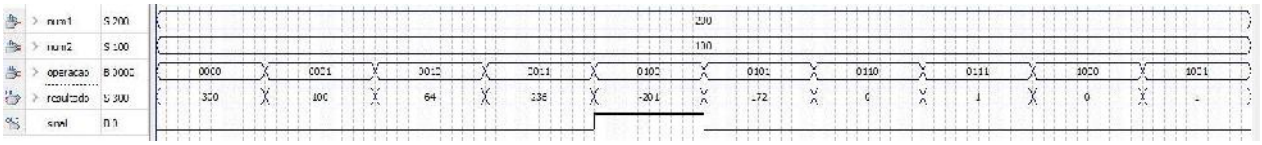

Figura 3(a). Simulação das operações da ALU.

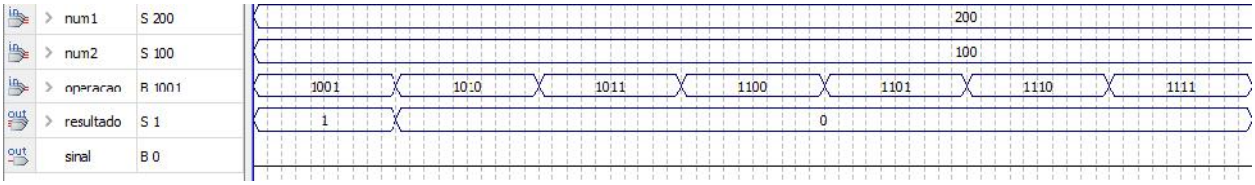

Figura 3(b). Simulação das operações da ALU.

Para esta simulação os dados de entrada foram dois números, 200 e 100 que correspondem aos campos num1 e num2 respectivamente. O resultado da operação pode ser verificado logo abaixo do campo operacao, que tem o código correspondente à operação realizada. Como exemplo, podemos citar a operação 0110 que é a set less than. Como 200 é maior do que 100, então o resultado será 0. Caso num1 fosse 100 e num2 fosse 200, o resultado seria 1. Caso o a operacao assuma um valor entre 1011 e 1111 o resultado será 0 pois não há operações definidas para esses códigos.

4.2 – Simulação da memória de dados

Para a memória de dados foi feita uma simulação de leitura e escrita. À seguir na Figura 4 a simulação é exibida.

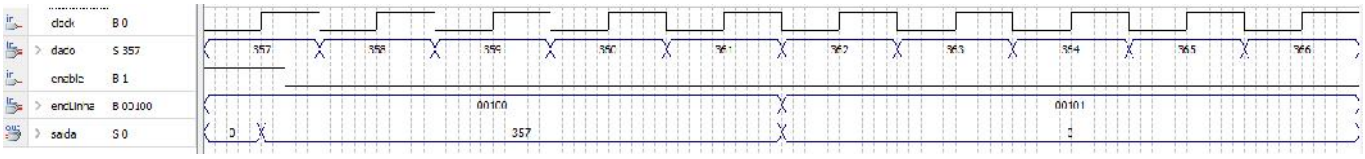

Figura 4. Simulação da memória de dados.

A escrita ocorre quando o clock sobe. Como o enable estava em 1 desde o começo, no momento em que o clock sobe a escrita ocorre. Isso pode ser confirmado com a mudança do valor do campo saída, que foi de 0 para 357. O valor do dado a ser escrito variou durante o tempo, entretanto como o enable esteve desabilitado, a saída permaneceu inalterada. Sendo assim, ao tentar ler um valor num endereço de memória que não teve nenhum dado escrito, o resultado será 0, como pode ser observado no valor do campo saída ao tentar ler o valor contido no endereço 00101 de memória.

4.3 – Simulação da memória de instruções

Da mesma forma que foi feito para a memória de dados, a memória de instruções também foi simulada. O comportamento desta memória pode ser pensado como a memória de dados. Entretanto o dado a ser armazenado é o código da instrução a ser realizada. A Figura 12 exibe a simulação.

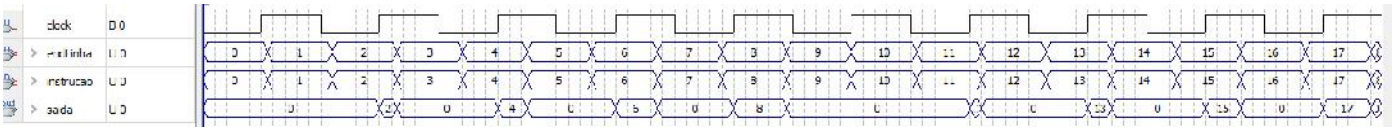

Figura 5. Simulação da memória de instruções.

Como aqui a escrita de uma instrução só ocorre na subida do clock o tempo de variação dos valores dos campos endLinha e instrução foram ligeiramente atrasados em relação ao do clock para que a simulção pudesse ficar mais fácil de visualizá-la. O valor do resultado é o número da instrução que foi escrita. Sendo assim, é de se esperar que seu valor altere somente depois que o clock assumiu valor 1. Isso pode ser observado na Figura 5 em que os valores do campos saída ora assume 0 ora assume valor diferente de 0, sendo este último após a subida do clock.

4.4 – Simulação do banco de registradores

Mais um tipo de memória do sistema computacional foi simulado. Desta vez fez o banco de registradores foi testado. Na figura 6 está a simulação do banco. Um número foi armazenado em um registrador e posteriormente foi lido.

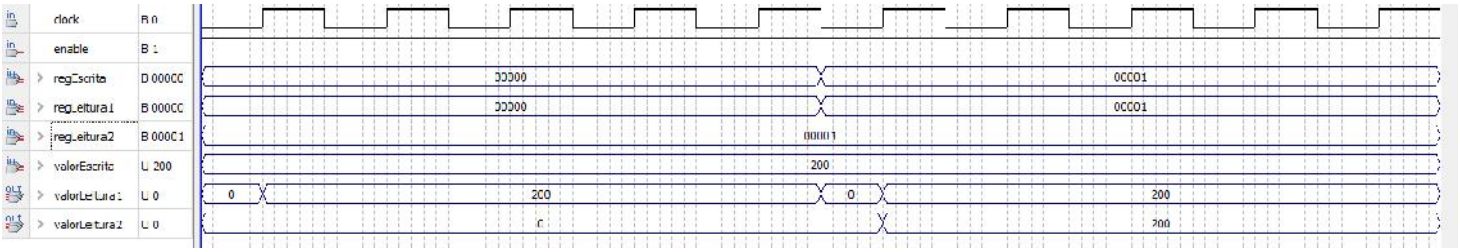

Figura 6. Simulação do banco de registradores.

O valor a ser armazenado é o campo valorEscrita, que assumiu valor 200 para esta simulação. O enable esteve em alto durante toda a simulação para que a escrita nos registradores pudesse acontecer. O campo regEscrita corresponde ao endereço do registrador que irá receber o valor contido em valorEscrita. O campo valorLeitura1 é o endereço do registrador a ser lido, a fim de ter o valor armazenado nele recuperado. Isto é análogo para os campos regLeitura2 e valorLeitura2.

O campo valorLeitura1 começa em zero pois o clock ainda não tinha subido. Após ele assumir o valor 1, o valor de saída para valorLeitura1 se torna 200. Já o valor de valorLeitura2 começa em zero pois o registrador ao qual está sendo feita a leitura ainda não teve nenhum dado escrito. Quando o clock sobe, ele muda o valor para 200, momento em que ocorreu a escrita no registrador referenciado por regLeitura2.

4.5 – Simulação do contador de programa (PC)

O contador de programa foi simulado para que seus valores pudessem ser analisados. Na Figura 7 esta simulação é exibida.

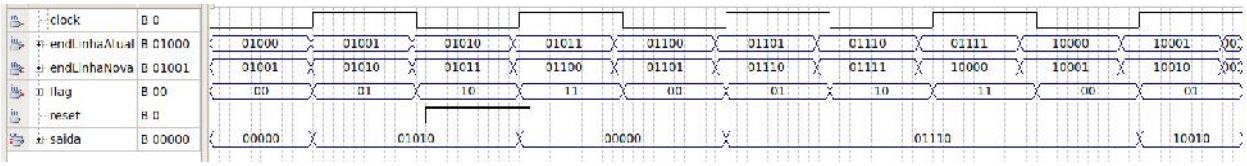

Figura 7. Simulação do contador de programa (PC).

Seu funcionamento é bem simples. Apenas 3 possibilidades de operação no contador de programa existem. Para o caso em que o campo flag assume valor 00, o valor do campo saída será a soma do valor do campo endLinhaAtual com o número 1. Esta operação é feita na subida do clock. Portanto, corresponde ao segundo valor assumido pelo campo saída. Caso flag seja 01, a operação é de jump. Ou seja, um endereço para a memória de instruções é passado.

4.6 – Simulação da unidade de controle

Algumas instruções foram escritas na memória de instruções para se poder testar o funcionamento da unidade de controle integrada aos outros módulos. Sendo assim, as instruções informadas para esta primeira simulação foram exibidas na Figura 5, que consistem em carregar

o número 20, em seguida o número 30 e por fim somá-los. À seguir, na Figura 8 está a simulação da unidade de controle.

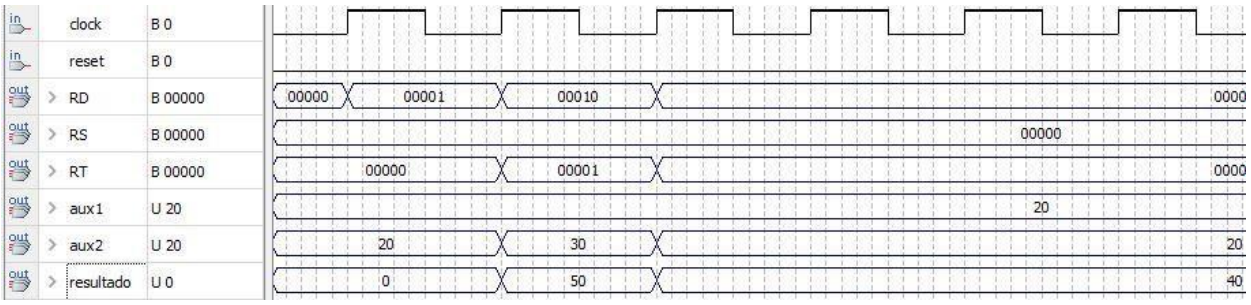

Figura 8 (a). Simulação da unidade de controle.

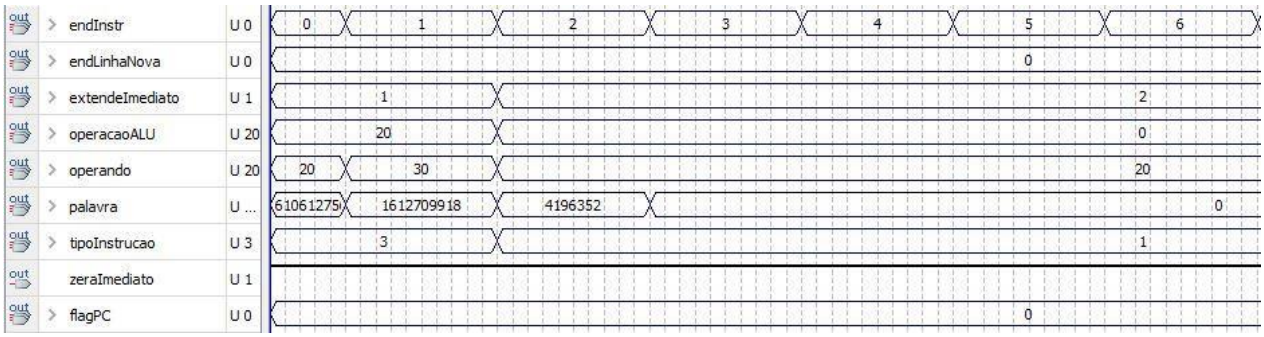

Figura 8 (b). Simulação da unidade de controle.

Primeiramente, o imediato 20 de 16 bits é gravado no registrador 0. Isto é feito na primeira borda de subida do clock. Em seguida a mesma operação é feita para o imediato 30, sendo este armazenado no registrador 1 e a instrução realizada na segunda subida do clock. Após isto, os dois valores contidos nos registradores 0 e 1 são somados na ALU e o valor é gravado no registrador 2, do banco de registradores. Depois, é realizada uma subtração entre o valor contido no registrador 2 com o imediato 50. Como a ALU independe do clock, o valor do resultado da soma está finalizado no segundo clock, e o da subtração no terceiro clock, visto que foi necessário um clock a mais para poder ler a nova instrução.

Os valores das flags podem ser observados na figura 8 (b). O campo *enableEscritaMemoria* permanece com valor 1 durante todo o processamento pois todas as instruções executadas realizam escritas em registradores. O campo *escreveRAM_DADOS* é habilitado para as instruções ADD e SUB, sendo assim, os resultados de seus processamentos será gravado na memória de dados. O campo *endInstr* não é uma flag, mas sim o endereço da nova instrução. Ele está relacionado com o campo *flagPC*. Como este é 0 durane todo o processamento, o valor de *endInstr* é incrementado em 1 a cada subida de clock. Para as duas primeiras instruções, o campo *operacaoALU* recebe o valor 20 que significa que nenhuma operação na ALU deverá ser realizada, haja visto que a instrução para este caso contida na memória de instruções são do tipo LOAD. Posteriormente, seu valor é alterado para 0, que corresponde ao ADD. O campo *operando* é utilizado para ter seu valor escrito no banco de

registradores. Sendo assim, primeiramente ele assume o valor 20 e depois 30, correspondendo às duas instruções LOAD. O campo *palavra* corresponde ao número da instrução contida na memória de instruções. Na simulação, ela foi representada em decimal para facilitar a visualização, já que possui 32 bits. O campo *tipoInstrucao* corresponde ao formato da instrução. Para as duas primeiras instruções o formato é o 3, já para as duas últimas, o formato é o 1 e 2, respectivamente. O campo *zeraImediato* é tido com 1 para instruções do tipo 1.

4.7 – Simulação do sistema computacional

Para testar o funcionamento do sistema computacional foram executados alguns algoritmos. O primeiro deles foi o algoritmo de Fibonacci em que o usuário deveria inserir por meio das chaves n-ésimo termo desejado. Na Figura X à seguir está o resultado em forma de onda da simulação.

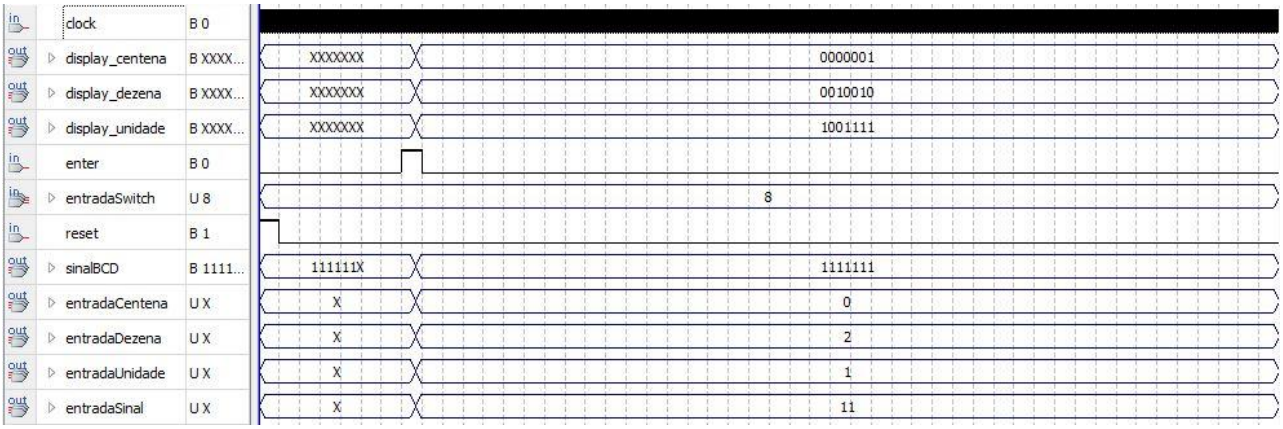

Figura 9. Simulação do cálculo do n-ésimo valor da sequência de Fibonacci.

Os valores de cada display foram colocados como output sendo eles *display_centena, display-_dezena, display_unidade* e *sinalBCD.* As correspondências em binário para cada display foram colocadas como outputs para análise. Seus nomes são *entradaCentena, entradaDezena, entradaUnidade* e *entradaSinal.* Na Figura X os campos *entradaCentena, entradaDezena, entradaUnidade* possuem como saída os números 0, 2, 1, respectivamente, o que corresponde ao 8° termo da sequência de Fibonacci, o número 021.

O campo *entradaSwitch* se refere ao valor informado através das chaves. O algoritmo de Fibonacci foi escrito na memória de instruções semelhante ao que está na Figura X. Na simulação o valor informado pelas chaves foi 8. À seguir está o algoritmo de Fibonacci implementado de acordo com as instruções contidas na conjunto de instruções.

```
26      memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
27      memoriaInstrucoes[1] <= {6'd21, 5'd0, 21'd0}; //IN R0
28      memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd1};  //LOAD R1, 1 :primeiro valor
29      memoriaInstrucoes[3] <= {6'd24, 5'd2, 21'd1};  //LOAD R2, 1 :segundo valor
30      memoriaInstrucoes[4] <= {6'd24, 5'd3, 21'd2};  //LOAD R3, 0 :contador
31      memoriaInstrucoes[5] <= {6'd0, 5'd4, 5'd2, 5'd1, 11'd0}; //ADD R4, R2, R1
32      memoriaInstrucoes[6] <= {6'd1, 5'd3, 5'd3, 16'd1}; //ADD I R3, 1
33      memoriaInstrucoes[7] <= {6'd13, 5'd1, 5'd2, 16'd0}; //MOVE R1, R2
34      memoriaInstrucoes[8] <= {6'd13, 5'd2, 5'd4, 16'd0}; //MOVE R2, R4
35      memoriaInstrucoes[9] <= {6'd20, 5'd0, 5'd3, 16'd5}; //BNQ R0, R3, 4
36      memoriaInstrucoes[10] <= {6'd25, 5'd4, 21'd0}; //STORE R4, 0
37      memoriaInstrucoes[11] <= {6'd22, 5'd4, 21'd0}; //OUT R4
38      memoriaInstrucoes[12] <= {6'd16, 26'd0};
```

Figura 10. Algoritmo de Fibonacci escrito pelo conjunto de instruções do sistema.

Neste algoritmo são utilizadas instruções de load imediato (linhas 28 e 29), soma de registradores (linha 31), soma de imediato (linha 32), instruções para mover valores entre registradores (linhas 33 e 34), instrução de salto condicional (linha 35), instruções para guardar valor na memória de dados (linha 36), instrução de saída de dados (linha 37) e por fim uma instrução de parada no processamento (linha 38). Na linha 26 existe uma instrução de *NOP* que serve para garantir que o contador de programas não irá ler lixo.

Como o programa testado não trabalha com todos os tipos de instruções, à seguir serão mostrados os resultados de simulações das instruções do conjunto de instruções

.

4.7.1 – Set less than

A instrução que compara se um valor contido num registrador é menor do que outro valor contido em outro registrador foi simulada. À seguir está a forma de onda para tal simulação.

```
72    memoriaInstrucoes[0] <= {6'd21, 5'd0, 21'd0}; //IN R0
73    memoriaInstrucoes[1] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
74    memoriaInstrucoes[2] <= {6'd17, 5'd2, 5'd1, 5'd0}; // SLT R2, R1, R0
75    memoriaInstrucoes[3] <= {6'd22,  5'd2,  21'd0}; //OUT R2
```

Figura 11. Instruções para a simulação contidas na memória de instruções.

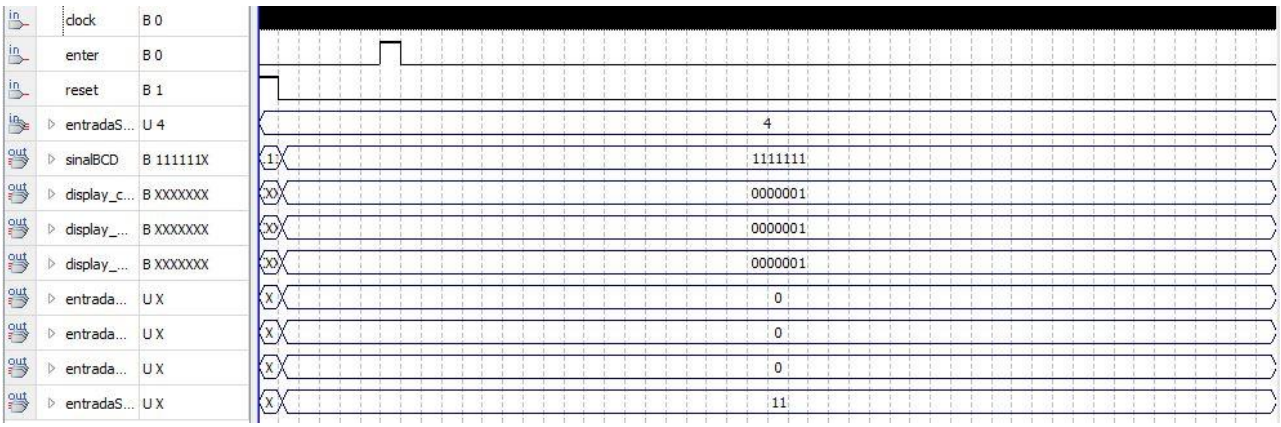

Figura 12. Simulação da instrução "SET LESS THAN".

4.7.2 – Set less than immediate

A instrução que compara se um valor contido num registrador é menor do que outro valor imediato foi simulada. À seguir está a forma de onda para tal simulação.

```
72    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
73    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd5}; //LOAD R0, 5
74    memoriaInstrucoes[2] <= {6'd18, 5'd2, 5'd0, 16'd8}; // SLT I R2, R0, 8
75    memoriaInstrucoes[3] <= {6'd22, 5'd2, 21'd0}; //OUT R2
76    memoriaInstrucoes[4] <= {6'd16, 26'd0};
```

Figura 13. Instruções para a simulação contidas na memória de instruções

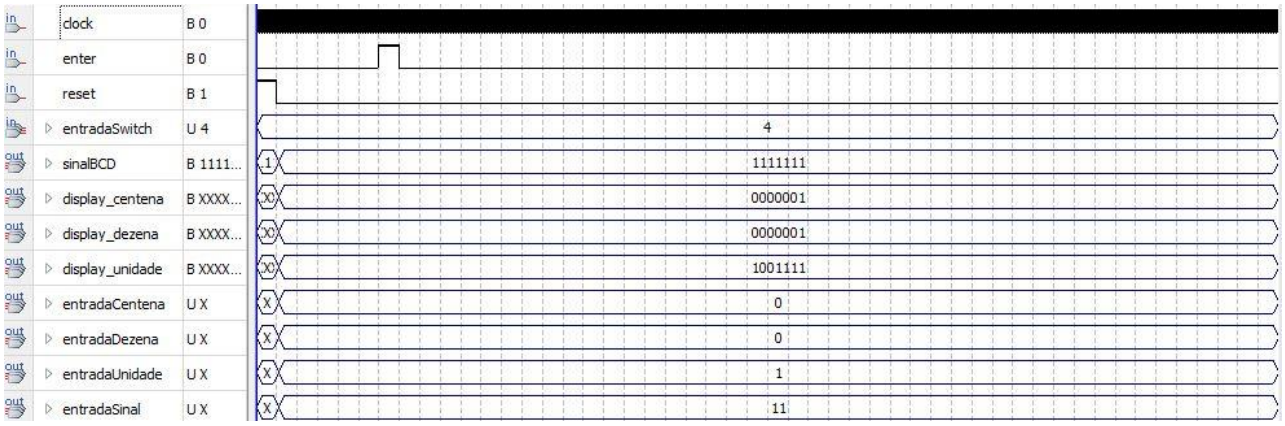

Figura 14. Simulação da instrução "SET LESS THAN IMMEDIATE".

4.7.3 – Set less or equal to

A instrução que compara se um valor contido num registrador é menor do que outro valor contido em outro registrador ou igual foi simulada. À seguir está a forma de onda para tal simulação.

```
79    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
80    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd8}; //LOAD R0, 8
81    memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
82    memoriaInstrucoes[3] <= {6'd26, 5'd2, 5'd1, 5'd0}; // SLE R2, R1, R0
83    memoriaInstrucoes[4] <= {6'd22, 5'd2, 21'd0}; //OUT R2
84    memoriaInstrucoes[5] <= {6'd16, 26'd0}; //HLT
```

Figura 15. Instruções para a simulação contidas na memória de instruções

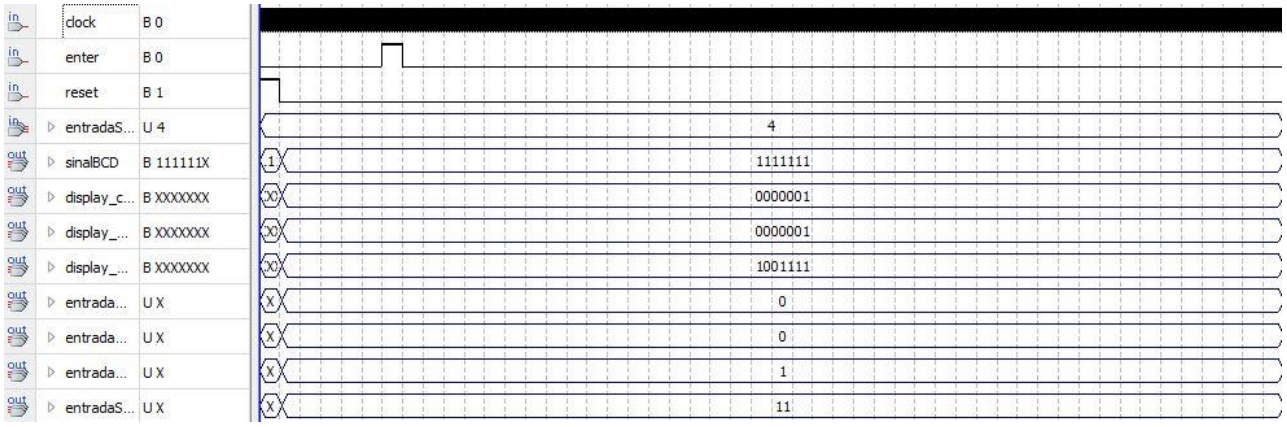

Figura 16. Simulação da instrução "SET LESS OR EQUAL TO".

4.7.4– Set less or equal to immediate

A instrução que compara se um valor contido num registrador é menor do que outro valor imediato ou igual foi simulada. À seguir está a forma de onda para tal simulação.

```
72    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
73    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd5}; //LOAD R0, 5
74    memoriaInstrucoes[2] <= {6'd26, 5'd2, 5'd0, 16'd8}; // SLE I R2, R0, 8
75    memoriaInstrucoes[3] <= {6'd22, 5'd2, 21'd0}; //OUT R2
76    memoriaInstrucoes[4] <= {6'd16, 26'd0}; //HLT
```

Figura 17. Instruções para a simulação contidas na memória de instruções

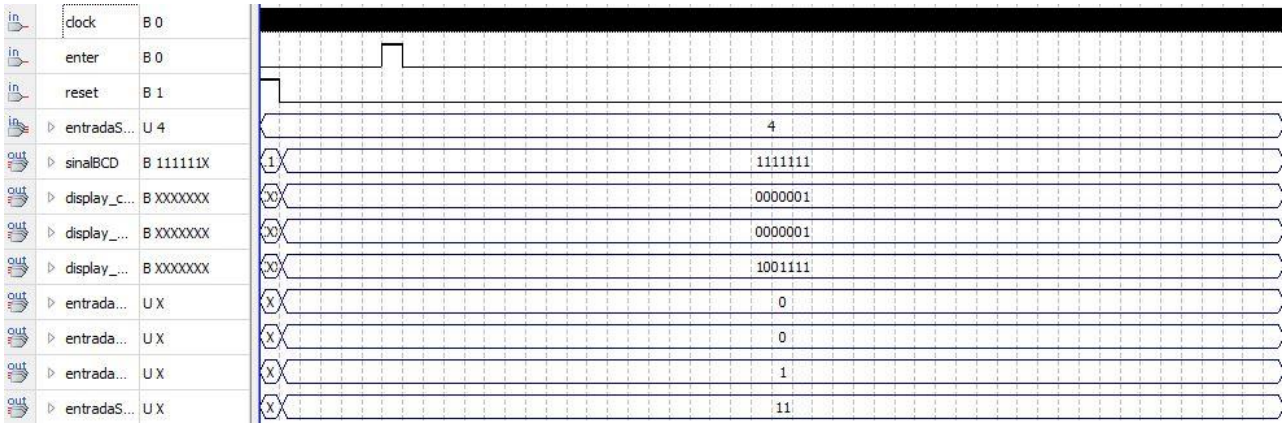

Figura 18. Simulação da instrução "SET LESS OR EQUAL TO IMMEDIATE".

4.7.5 – Set higher than

A instrução que compara se um valor contido num registrador é maior do que outro valor contido em outro registrador foi simulada. À seguir está a forma de onda para tal simulação.

```
79    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
80    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd8}; //LOAD R0, 8
81    memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
82    memoriaInstrucoes[3] <= {6'd28, 5'd2, 5'd1, 5'd0}; // SHT R2, R1, R0
83    memoriaInstrucoes[4] <= {6'd22, 5'd2, 21'd0}; //OUT R2
84    memoriaInstrucoes[5] <= {6'd16, 26'd0}; //HLT
```

Figura 19. Instruções para a simulação contidas na memória de instruções

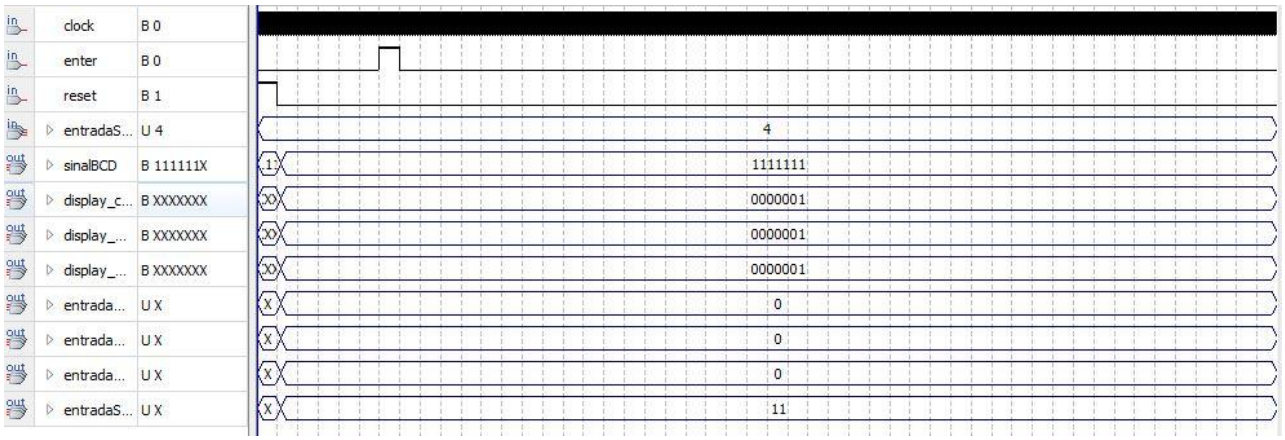

Figura 20. Simulação da instrução "SET HIGHER THAN".

4.7.6– Set higher than immediate

A instrução que compara se um valor contido num registrador é maior do que outro valor imediato foi simulada. À seguir está a forma de onda para tal simulação.

```
72    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
73    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd5}; //LOAD R0, 5
74    memoriaInstrucoes[2] <= {6'd29, 5'd2, 5'd0, 16'd8}; // SHT I R2, R0, 8
75    memoriaInstrucoes[3] <= {6'd22, 5'd2, 21'd0}; //OUT R2
76    memoriaInstrucoes[4] <= {6'd16, 26'd0}; //HLT
```

Figura 21. Instruções para a simulação contidas na memória de instruções

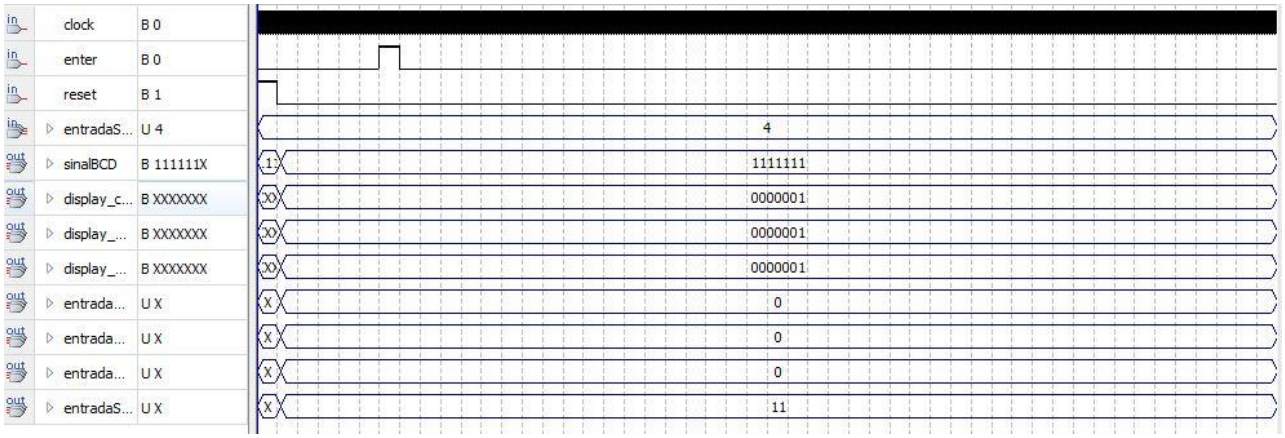

Figura 22. Simulação da instrução "SET HIGHER THAN IMMEDIATE".

4.7.7 – Set higher or equal to

A instrução que compara se um valor contido num registrador é maior do que outro valor contido em outro registrador ou igual foi simulada. À seguir está a forma de onda para tal simulação.

```
79    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
80    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd8}; //LOAD R0, 8
81    memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
82    memoriaInstrucoes[3] <= {6'd30, 5'd2, 5'd1, 5'd0}; // SHT R2, R1, R0
83    memoriaInstrucoes[4] <= {6'd22, 5'd2, 21'd0}; //OUT R2
84    memoriaInstrucoes[5] <= {6'd16, 26'd0}; //HLT
```

Figura 23. Instruções para a simulação contidas na memória de instruções

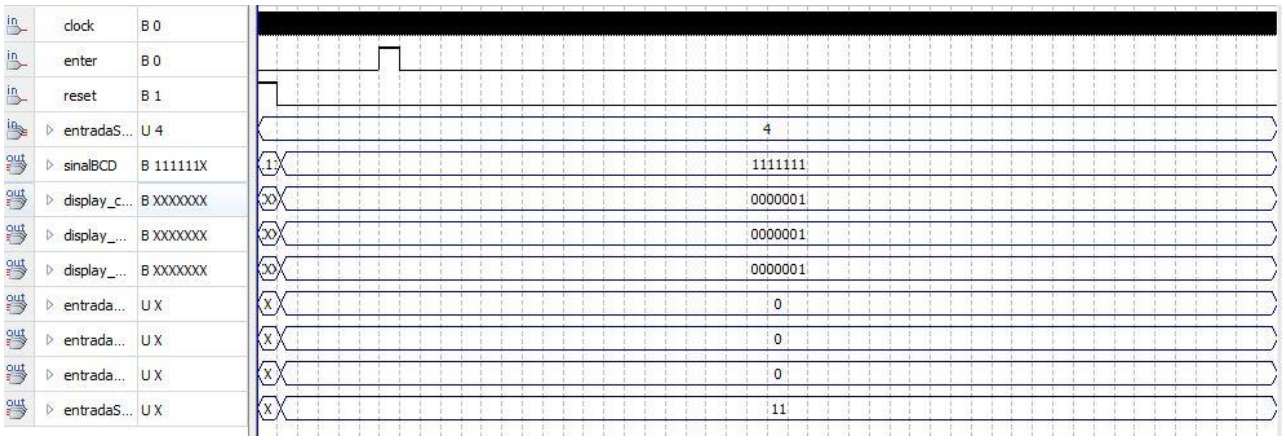

Figura 24. Simulação da instrução "SET HIGHER OR EQUAL TO".

4.7.8– Set higher or equal to immediate

A instrução que compara se um valor contido num registrador é maior do que outro valor imediato ou igual foi simulada. À seguir está a forma de onda para tal simulação.

```
72  memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
73  memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd5}; //LOAD R0, 5
74  memoriaInstrucoes[2] <= {6'd29, 5'd2, 5'd0, 16'd8}; // SHT I R2, R0, 8
75  memoriaInstrucoes[3] <= {6'd22, 5'd2, 21'd0}; //OUT R2
76  memoriaInstrucoes[4] <= {6'd16, 26'd0}; //HLT
```

Figura 25. Instruções para a simulação contidas na memória de instruções

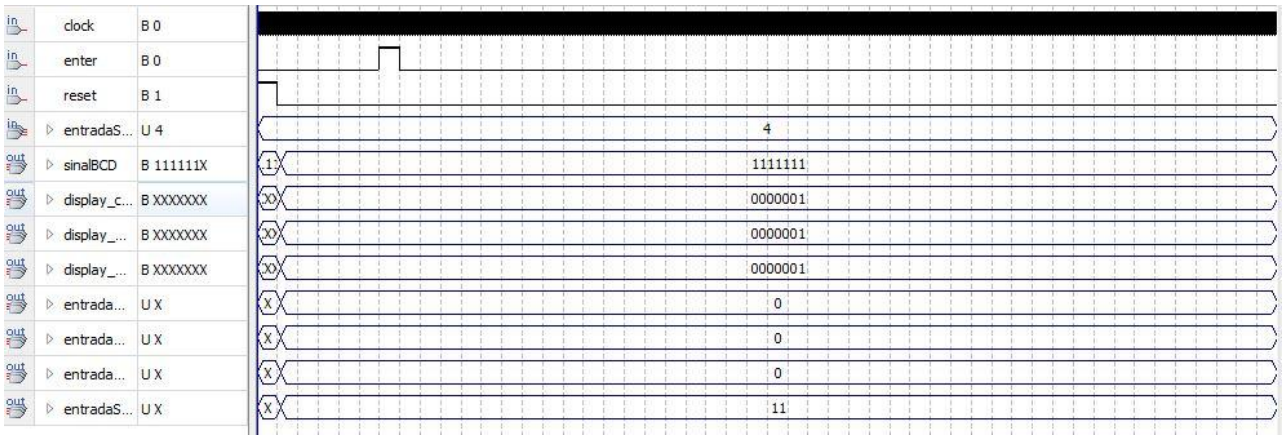

Figura 26. Simulação da instrução "SET HIGHER OR EQUAL TO IMMEDIATE".

4.7.9 – Set equal to

A instrução que compara se um valor contido num registrador é igual a outro valor contido em outro registrador foi simulada. À seguir está a forma de onda para tal simulação.

```
79  memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
80  memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd8}; //LOAD R0, 8
81  memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
82  memoriaInstrucoes[3] <= {6'd32, 5'd2, 5'd1, 5'd0}; // SET R2, R1, R0
83  memoriaInstrucoes[4] <= {6'd22, 5'd2, 21'd0}; //OUT R2
84  memoriaInstrucoes[5] <= {6'd16, 26'd0}; //HLT
```

Figura 27. Instruções para a simulação contidas na memória de instruções

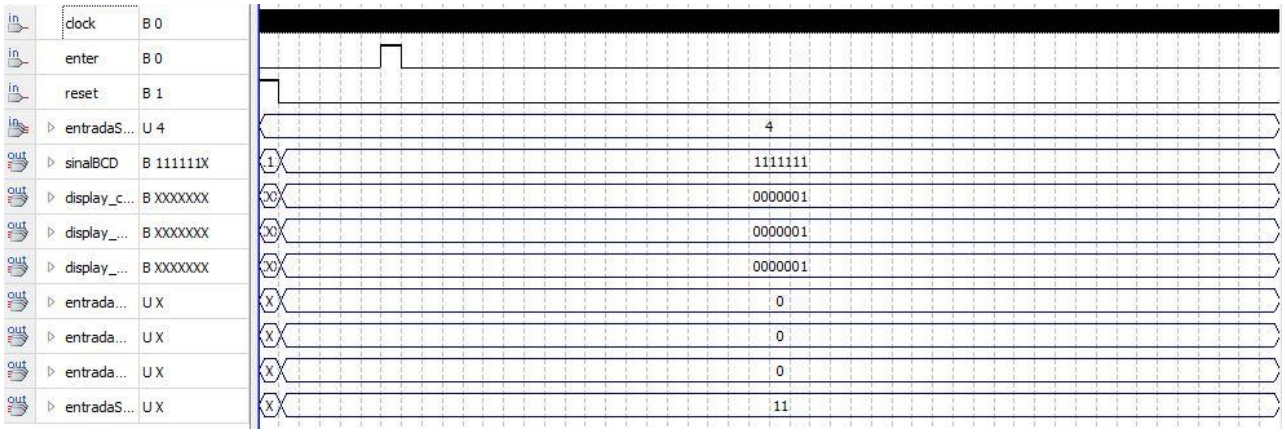

Figura 28. Simulação da instrução "SET EQUAL TO".

4.7.10– Set equal to immediate

A instrução que compara se um valor contido num registrador igual a um valor imediato foi simulada. À seguir está a forma de onda para tal simulação.

```
72    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
73    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd5}; //LOAD R0, 5
74    memoriaInstrucoes[2] <= {6'd33, 5'd2, 5'd0, 16'd8}; // SET I R2, R0, 8
75    memoriaInstrucoes[3] <= {6'd22, 5'd2, 21'd0}; //OUT R2
76    memoriaInstrucoes[4] <= {6'd16, 26'd0}; //HLT
```

Figura 29. Instruções para a simulação contidas na memória de instruções

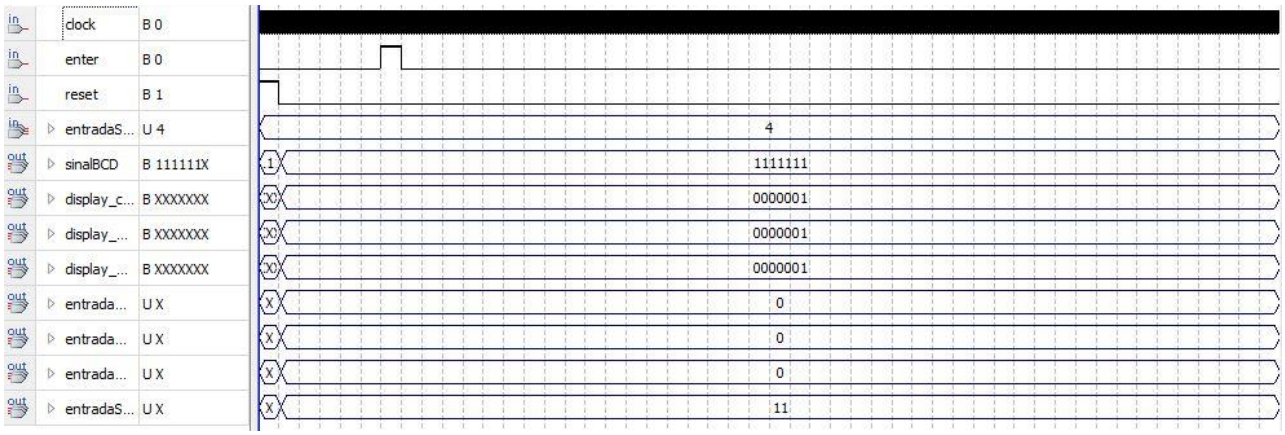

Figura 30. Simulação da instrução "SET EQUAL TO IMMEDIATE".

4.7.11– ADD I

A instrução que realiza a soma de um valor contido num registrador com um valor imediato foi testada. À seguir está a forma de onda para tal simulação.

```
79    memoriaInstrucoes[0] <= {6'd15, 26'd0}; //NOP
80    memoriaInstrucoes[1] <= {6'd24, 5'd0, 21'd8}; //LOAD R0, 8
81    memoriaInstrucoes[2] <= {6'd24, 5'd1, 21'd5}; //LOAD R1, 5
82    memoriaInstrucoes[3] <= {6'd32, 5'd2, 5'd1, 5'd0}; // SET R2, R1, R0
83    memoriaInstrucoes[4] <= {6'd22, 5'd2, 21'd0}; //OUT R2
84    memoriaInstrucoes[5] <= {6'd16, 26'd0}; //HLT
```

Figura 31. Instruções para a simulação contidas na memória de instruções

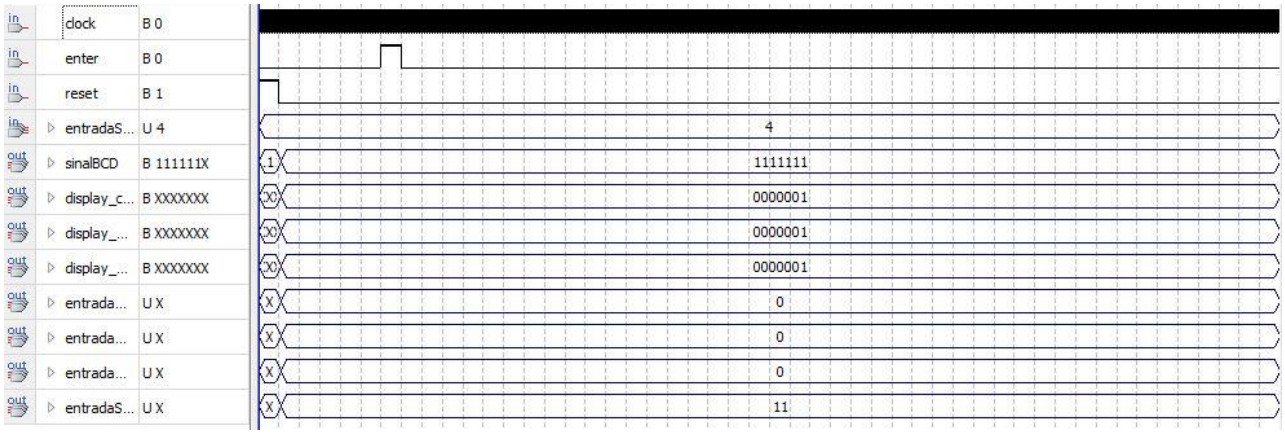

Figura 32. Simulação da instrução "ADD I".

5. CONCLUSÃO

A elaboração de um sistema computacional requer diversos passos que muitas vezes podem ficar obscuros, dificultando sua implementação. No começo da elaboração do projeto houve dificuldades em definir este conjunto devido às inúmeras possibilidades que existem para instruções. A montagem do conjunto também foi pensada de maneira a se seguir uma arquitetura RISC, tomando cuidados para não se tornar CISC.

Este trabalho teve como objetivo desenvolver e implementar um sistema computacional composto por uma centra de processamento, uma memória e uma interface de entrada e saída de dados.

Quanto à implementação dos módulos a parte mais difícil foi o gerenciamento e o tratamento dado a cada bit dos valores envolvidos. Cada módulo trabalharia de uma determinada maneira conforme as informações repassadas a ela. A exibição das informações será feita por meio dos displays de sete segmentos.

6. REFERÊNCIA BIBLIOGRÁFICA

[1] PATTERSON, J. L.; H. D. A.; Organização e projeto de computadores: A inferface hardware/software, São Paulo, 3. Ed., 2005.

[2] STALLINGS, W.; Arquitetura e Organização de Computadores: Projeto para o desempenho, São Paulo, Prentice Hall, 5. Ed., 2002.

[3] WEBER, R. F.; Fudamentos de Arquitetura de Computadores, Porto Alegre, Bookman, 3. Ed, 2004.