Universidade Federal de São Paulo - UNIFESP

Laboratório de sistemas computacionais:
Arquitetura e Organização de Computadores
Relatório Final

José Henrique Fortes Leite - RA: 86.844

Julho/2016

Resumo

Neste relatório será apresentado todos os componentes do sistema computacional projetado e implementado, todos os módulos implementados e todos os testes realizados para comprovar o seu funcionamento.

Lista de Ilustrações

Lista de Tabelas

Sumário

1. Introdução

Os primeiros computadores, antes mesmo da década de 50, não podiam armazenar dados. Alguns deles, como o ENIAC (Figura 1), tinham em seu projeto inicial o plano de armazenar Softwares, porém, para agilizar o lançamento, essa idéia foi deixada para trás.

Com isso, era necessário reposicionar os cabos cada vez que uma tarefa diferente fosse ser realizada, e assim era feito o processamento.

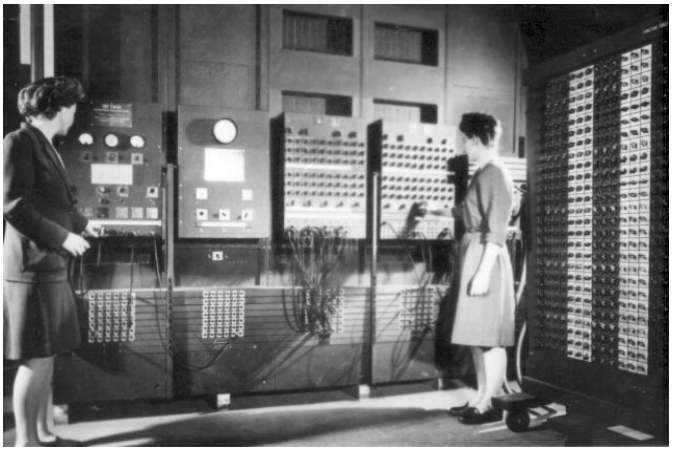

Figura 1. ENIAC - Referência [1]

John Von Neumann, em 1945, publicou a idéia de uma unidade central de processamento. Um projeto chamado EDVAC foi realizado com essa característica e finalizado em 1949, na qual deu origem aos primeiros modelos de processamento da forma que é realizada hoje, capazes de armazenar programas.

No início da década de 70, surgiram as CPU's desenvolvidas em circuitos integrados, utilizando a arquitetura x86, que serve de base pra boa parte dos processadores atuais.

O processador encontra-se por trás de todo sistema computacional, desde os mais antigos aos mais atuais. É ele o responsável por efetuar todos os cálculos que os softwares demandam.

.

2. Objetivos

2.1. Objetivo Geral

O objetivo geral desde trabalho é desenvolver um sistema computacional completo e funcional, em uma FPGA utilizando a linguagem Verilog como linguagem de programação de hardware.

2.1. Objetivo Específico

O objetivo específico é completar o sistema computacional como um todo, e demonstrar seu correto funcionamento através da FPGA com alguns códigos escritos que efetue os testes.

3. Fundamentação Teórica

3.1. Arquitetura de Computadores

Um computador possui diversas características, existem diversos parâmetros e formas de se comparar um computador. As diversas características, tais como a CPU, a capacidade de memória, dentre outros, se caracteriza como a arquitetura do computador. Uma arquitetura determina características pré definidas, que relacionam diretamente ao desempenho e a qual fim o computador será orientado.

Uma arquitetura define também coisas relacionadas a arquitetura da CPU do computador, tais como seu conjunto de instruções, seus modos de endereçamento, entre outros. Essa abordagem é comumente chamada de Arquitetura de processadores.

3.1.1. RISC x CISC

Uma arquitetura definida como RISC (Reduced Instruction Set Computer) é um tipo de arquitetura de processadores que tem como definição um conjunto simples e pequeno de instruções que levam geralmente o mesmo tempo para serem processadas. Uma grande parte dos processadores modernos, como o MIPS, utiliza dessa arquitetura.

Já a arquitetura CISC (Complex Instruction Set Computer) representa um tipo de arquitetura de processadores cuja principal característica é executar centenas de instruções

complexas e sendo extremamente versátil. Muita das instruções da arquitetura CISC são redundantes.

Os processadores atuais trabalham em uma mistura entre RISC e CISC.

3.2. MIPS

A arquitetura MIPS (Microprocessor without interlocked pipeline stages) é uma arquitetura de microprocessadores RISC desenvolvida pela MIPS Computer Systems. Utiliza uma arquitetura baseada em registrador, ou seja, a CPU utiliza de registradores para armazenar suas informações relacionadas às suas operações.

Ao longo dos anos foram realizadas diversas atualizações e diversos novas revisões foram sendo lançadas. Cada revisão é um superconjunto melhorado da versão anterior.

3.3. FPGA

É um tipo de circuito integrado, projetado especialmente para ser reconfigurada após a fabricação, diferentemente dos chips que acompanham a maioria dos eletrônicos, que vêm pré-programado de fábrica e não é possível que se mude a configuração.

O FPGA surgiu como uma nova categoria de Hardware, onde é possível reconfigura-lo através do software do próprio fabricante, onde as suas funcionalidades são definidas pelos próprios usuários.

Através do FPGA, é possível descarregar o código gerado por linguagens de descrição de hardware, pois elas são constituídas por milhares de transistores, que fazem basicamente o mesmo que vários circuitos integrados, com uma diferença de ser reprogramável.

3.4. Unidade de Controle

A unidade de controle é um “bloco” responsável por gerenciar todos os sinais que controlam as operações, tanto internamente quanto externamente ao CPU. Executa três ações básicas: Busca, Decodificação e Execução. Dependendo do tipo de microprocessador, a Unidade de Controle pode ser fixa ou programável.

3.5. Verilog

Verilog é uma linguagem de descrição de hardware utilizada para modelar sistemas eletrônicos. É uma poderosa ferramenta que suporta projetos analógicos, digitais e híbridos com diversos níveis de abstração. Utilizando algumas placas especiais, como a FPGA, é possível descarregar o código gerado nesta linguagem.

4. Desenvolvimento

4.1. Características Gerais

Foi projetado um processador com características semelhantes ao MIPS. A arquitetura projetada segue os padrões RISC, mesmo com algumas instruções sofrendo redundância.

O Banco de registradores foi projetado para conter 32 registradores de propósito geral, no qual demandam 5 bits de Opcode para serem especificados.

A memória de propósito geral foi projetada para conter 8kb e 8192 espaços, portanto, são necessários 13 bits para identificação.

4.2. Instruções

O conjunto de instruções foi projetado para ser eficiente, leve e versátil, com ampla possibilidade do programador realizar diversos tipos de tarefas com as instruções escritas, mesmo que não haja, de fato, alguma instrução específica para aquela tarefa a ser realizada.

As instruções foram divididas em alguns formatos específicos, e cada instrução possui um único formato. Foi decidido utilizar 6 bits de Opcode, o que permite até 63 instruções diferentes.

4.2.1. Tipo-R

As instruções do Tipo-R foram projetadas especificamente para atividades envolvendo Registradores, portanto, não é possível a entrada de nenhum Imediato utilizando este tipo de instrução. Há espaço para até três registradores, especificados na Tabela1 como Reg0, Reg1 e Reg2. Neste tipo de instrução, 11 bits não são aproveitados e recebem “dont care”. Seu formato geral pode ser visualizado através da Tabela 1.

Tabela 1. Formato da instrução do Tipo-R.

Opcode	Reg0	Reg1	Reg2	---
6 bits (31-26)	5 bits (25-21)	5 bits (20-16)	5 bits (15-11)	11 bits (10-0)

4.2.2. Tipo-I

É o tipo de instrução mais geral, há espaço para até dois registradores, especificados como Reg0 e Reg1, e um imediato de até 16 bits. É o tipo de instrução para trabalhar-se com

registradores e imediatos juntos, normalmente para enviar ou receber algum valor em algum registrador. Seu formato geral pode ser visualizado através da Tabela 2.

Tabela 2. Formato da instrução do Tipo-I.

Opcode	Reg0	Reg1	Im
6 bits (31-26)	5 bits (25-21)	5 bits (20-16)	16 bits (15-0)

4.2.3. Tipo-J

Este é o único tipo de instrução que não se trabalha com nenhum registrador, possui 26 bits para ser utilizado com um Imediato, portanto, permite trabalhar com um grande número. Normalmente utilizado para instruções de "jump", seu limite máximo é de "67108862" positivo ou negativo, convertendo diretamente para o decimal. Seu formato geral pode ser visualizado na Tabela 3.

Tabela 3. Formato da instrução do Tipo-J.

Opcode	End
6 bits (31-26)	26 bits (25-0)

Todas as instruções foram divididas de acordo com seu tipo e formato, e estas podem ser visualizadas através da Tabela 4.

Tabela 4. Tipos de cada instrução

Formato	Instrução
Tipo-R	ADD, SUB, AND, OR, NOT, XOR
Tipo-I	ADDI, SUBI, ANDI, ORI, SLL, SRL, BEQ, BNE, BGT, BLT, LW, STORE, MOV, IN, OUT
Tipo-J	JMP, NOP, HLT

4.2.4. Instruções Aritméticas

São as instruções que realizam algum tipo de operação aritmética, soma ou subtração. As operações podem ser realizadas entre registrador e registrador ou registrador e imediato. Todas as instruções desta categoria podem ser visualizadas através da Tabela 5.

Tabela 5. Instruções Aritméticas

Opcode	Nome	Sintaxe	Significado	Descrição	Tipo
000000	**ADD**	add $1, $2, $3	$3 = $1 + $2	Adiciona dois registradores em um terceiro	R
000001	**ADDI**	addi $1, A	$1 = $1 + A	Adiciona um imediato a um registrador	I
000010	**SUB**	sub $1, $2, $3	$3 = $1 - $2	Remove de um registrador outro registrador e salva em um terceiro registrador	R
000011	**SUBI**	subi $1, A	$1 = $1 - A	Remove um imediato de um registrador e salva em um terceiro registrador	I

4.2.5. Instruções de Lógica

Este tipo de instruções realiza as operações lógica com registradores, e armazenam seu resultado também em registradores, como pode ser visualizado através da Tabela 6.

Tabela 6. Instruções de Lógica

Opcode	Nome	Sintaxe	Significado	Tipo
001000	**AND**	and $1, $2, $3	$1 = $2 AND $3	R
001001	**ANDI**	andi $1, $2 and A	$1 = $2 AND A	I
001010	**OR**	or $1, $2, $3	$1 = $2 OR $3	R
001011	**ORI**	ori $1, $2 and A	$1 = $2 OR A	I
001100	**NOT**	not $1, $2	$1 = NOT $2	R
001101	**XOR**	xor $1, $2, $3	$1 = $2 XOR $3	R
010000	**SLL**	sll $1,$2,CONST	$1 = $2 << CONST	I
010001	**SRL**	srl $1,$2,CONST	$1 = $2 >> CONST	I

4.2.6. Instruções de Branch

São instruções que “saltam” entre os endereços de instruções de um programa, seja com alguma condição ou não. Todas as comparações são realizadas entre registradores e através desta classe de instruções, que pode ser visualizada através da Tabela 7.

Tabela 7. Instruções de Branch

Opcode	Nome	Sintaxe	Significado	Tipo
011000	**BEQ**	beq	if (R1 == R2) PC = LABEL	I
011001	**BNE**	bne	if (R1 != R2) PC = LABEL	I
011010	**BGT**	bgt	if (R1 > R2) PC = LABEL	I
011011	**BLT**	blt	if (R1 < R2) PC = LABEL	I
011100	**JMP**	jmp	PC = LABEL	J

4.2.7. Instruções de Dados

É o tipo de instruções que permite mover dados entre a memória de propósito geral e os registradores, além de mover endereços entre os registradores.

Tabela 8. Instruções de Dados

Opcode	Nome	Sintaxe	Significado	Tipo
100000	**LW**	lw $1, CONST($2)	$1 = MEMORY[$2 + CONST]	I
100001	**STORE**	store $1, CONST($2)	MEMORY[$2] = $1	I
100010	**MOV**	mov [R1, R2]	R1 = R2	I

4.2.8. Instruções de controle

Neste tipo de instruções, é possível controlar o que acontece com o processo.

Tabela 9. Instruções de Controle

Opcode	Nome	Descrição	Tipo
101000	**NOP**	Nenhuma Operação	J
101001	**HLT**	Parar Processamento	J
101010	**RESET**	Reseta o program counter para o endereço inicial	J

4.2.9. Instruções I/O

São as instruções que fazem a ligação entre o usuário e o processador.

Tabela 10. Instruções I/O

Opcode	Nome	Significado	Tipo
111000	**IN**	Permite a inserção de um número em binário.	I
111001	**OUT**	Altera o valor em um dos displays de 7 segmentos para o valor de um registrador.	I
111010	**OUT3**	Altera o valor nos três displays de 7 segmentos para valores de 3 registradores.	R

4.3. Modos de Endereçamento

Os modos de endereçamento definem como a instrução fará sua referência na memória. Os modos de endereçamento são específicos de cada instrução de acordo com o que ela necessita.

4.3.1. Endereçamento Imediato

O operando neste modo de endereçamento é o próprio imediato, não há referências na memória para buscar o dado, por isso, este modo é muito rápido, porém, o tamanho do operando é limitado pelo tamanho do campo da instrução.

4.3.2. Endereçamento Direto

A instrução contém o endereço do operando, é necessário buscar na memória seu valor para efetuar a operação especificada pela instrução, não é tão rápida quanto o Endereçamento Imediato, porém, a limitação está no tamanho da memória, o que é muito superior ao tamanho do campo de instrução.

4.3.3. Endereçamento por Registrador

O operando encontra-se no registrador, a sua maior limitação é o número de registradores, que é bem limitado. Como não há acesso a memória, este modo de endereçamento é rápido e eficiente.

4.4. Datapath

O Datapath é o caminho que os dados percorrem dentro do processador. Foi projetado um Datapath monociclo, com alguns módulos síncronos e outros assíncronos. Cada um dos blocos de execução possuem a sua devida finalidade específica. O datapath completo pode ser visualizado através do Anexo A.

4.4.1. Caminho de instruções

Para efetuar a leitura das instruções, são necessários alguns módulos trabalhando em conjunto.

4.4.1.1. Program Counter

É o responsável por guardar o endereço da próxima instrução. Foi implementado um Registrador de 32 bits que armazena este endereço. A cada ciclo de clock, o Program Counter envia o endereço que ele armazenou e recebe um novo endereço para armazenamento. Há uma flag de "reset" que, caso esteja em alta, reseta o Program Counter para a primeira instrução "0". Sua implementação em Verilog pode ser visualizada através da Figura 2.

```
module programCounter(
end_entrada,
end_saida,
clk,
reset);

   input [31:0] end_entrada;
   input clk;
   input reset;

   output reg[31:0] end_saida;

   always @ (posedge clk) begin
      if (reset) end_saida <= 32'b00000000000000000000000000000000;
      else end_saida <= end_entrada;
   end

endmodule
```

Figura 2. Implementação do Program Counter em Verilog.

4.4.1.2. Somador de Instruções

É um módulo simples, responsável apenas para somar o endereço do clock atual para um novo endereço, ou seja, apenas faz a soma de “+1” no endereço binário da instrução atual. Sua implementação pode ser visualizada através da Figura 3.

```
module somadorInstrucoes(
end_atual,
end_saida
);
   input [31:0] end_atual;
   output reg [31:0] end_saida;
   always @ (end_atual or end_saida) begin
      end_saida <= end_atual+1;
   end
endmodule
```

Figura 3. Somador de instruções em Verilog.

4.4.1.3. Multiplexadores

Há dois multiplexadores responsáveis por controlar qual será o endereço de instrução lido pelo Program Counter. O primeiro deles controla se o endereço do Branch virá da unidade lógica e aritmética (para as instruções branch) ou do Extensor de Bits (para instrução jump). Recebe sempre os dois endereços e uma flag de controle, sua implementação pode ser visualizada através da Figura 4.

```
module muxBranch(
controle,
endereco_extensor,
endereco_ula,
endereco_saida
);

input controle;
input [31:0] endereco_extensor;
input [31:0] endereco_ula;

output reg [31:0] endereco_saida;

always @ (controle or endereco_extensor or endereco_ula) begin
   if (controle == 1'b0)
      endereco_saida = endereco_extensor;
   else
      endereco_saida = endereco_ula;
end

endmodule
```

Figura 4. Multiplexador de endereço de branch a ser tomado em Verilog.

Há também um outro multiplexador um pouco mais específico, que controla se o endereço a ser recebido pelo Program Counter virá do Branch, do somador ou será o endereço antigo, sem alterações (para instruções Halt ou Input). Nele, é recebido, além de uma flag de controle, uma flag da Unidade Lógica e Aritmética informando se o branch foi ou não tomado, além de todos os demais endereços. Sua implementação pode ser visualizada através da Figura 5.

```
module muxInstrucao(
controle,
branch,
endereco_branch,
endereco_instrucao,
endereco_antigo,
endereco_saida
);
   input branch;
   input [1:0] controle;
   input [31:0] endereco_branch, endereco_instrucao, endereco_antigo;
   output reg [31:0] endereco_saida;
   always @ (controle or branch or endereco_branch or endereco_instrucao) begin
      if(controle == 2'b11)
         begin
            if (branch == 1'b1)
               endereco_saida = endereco_branch;
            else
               endereco_saida = endereco_instrucao;
         end
      else if (controle == 2'b10)
         endereco_saida = endereco_antigo;
      else
         endereco_saida = endereco_instrucao;
   end
endmodule
```

Figura 5. Multiplexador de Instruções.

4.4.1.4. Memória de instruções

Este é o módulo onde encontra-se todo o código do programa na linguagem de máquina. Todas as instruções são armazenadas na memória de instruções. Este módulo possui uma matriz onde cada linha possui 32 bits para armazenar cada instrução, como entrada, recebe o endereço que saiu do Program Counter, e envia como saída a instrução para os demais blocos do processador. Sua implementação pode ser visualizada, juntamente com um programa escrito para cálculo da sequência de Fibonacci através da Figura 6.

```
// INPUTS
input clk;
input [31:0]end_mem;
// OUTPUTS
output [31:0] saida_instrucao;
// VARIAVEIS INTERNAS
integer clockInicio = 0;
reg[31:0] mem_instrucoes [0:31];
always @ (posedge clk) begin
    if (clockInicio == 0) begin
    // FIBONACCI
    mem_instrucoes[0] = 32'b10100000000000000000000000000000;  // instrucao vazia
    mem_instrucoes[1] = 32'b00000100001000000000000000000001;  // ADDI 1 NO 00001
    mem_instrucoes[2] = 32'b11100000010xxxxxxxxxxxxxxxxxxxx;  // input no reg 00010
    mem_instrucoes[3] = 32'b11101000010000100001000xxxxxxxxxx;  // OUTPUT REG 00010 - armazena resultado que o usuario quer
    mem_instrucoes[4] = 32'b01100000000000010000000000010000;  // BEQ 00000 00010
    mem_instrucoes[5] = 32'b01100000000100010000000000010010;  // BEQ 00001 00010
    mem_instrucoes[6] = 32'b00000100100000000000000000000000;  // SETA INICIALMENTE O VALOR DO ANTIGO 2 para 0 - 00100
    mem_instrucoes[7] = 32'b00000100101000000000000000000001;  // SETA INICIALMENTE O VALOR DO ANTIGO 1 para 1 - 00101
    mem_instrucoes[8] = 32'b00000101000000000000000000000010;  // SETA INICIALMENTE O CONTADOR PARA 2 - 01000
    // INICIA O LAÇO DO FIBO PARA NUMEROS DIFERENTES DE 0 E 1
    mem_instrucoes[9] = 32'b00000000100001010011xxxxxxxxxx;  // REGISTRADOR DE FIBO ATUAL 00111 = 00100 + 00101
    mem_instrucoes[10] = 32'b11101000100001010011xxxxxxxxxx; // IMPRIME FIBO ATUAL 00111
    mem_instrucoes[11] = 32'b00000101000000000000000000000001; //ACRESCENTA +1 NO CONTADOR - 01000
    mem_instrucoes[12] = 32'b01100000010010000000000000010011; //VERIFICA SE CONTADOR E IGUAL AO 00010, SE SIM PULA PRO HALT
    mem_instrucoes[13] = 32'b10001000100001xxxxxxxxxxxxxxxx; //COLOCA NO ANTIGO 2 - 00100 - O VALOR DO ANTIGO 1
    mem_instrucoes[14] = 32'b10001000101001xxxxxxxxxxxxxxxx; //COLOCA NO ANTIGO 1 - 00101 - O VALOR DO ATUAL
    mem_instrucoes[15] = 32'b01110000000000000000000001001; //JUMP PRO INICIO DO LAÇO
    mem_instrucoes[16] = 32'b11101000000000000000xxxxxxxxxx; //IMPRIME ZERO
    mem_instrucoes[17] = 32'b01110000000000000000000010011; //JUMP PRO HALT
    mem_instrucoes[18] = 32'b11101000001000100001xxxxxxxxxx; //IMRPIME UM
    mem_instrucoes[19] = 32'b101001xxxxxxxxxxxxxxxxxxxxxxxx; //HALT
    clockInicio <= 1'b1;
    end
end
assign saida_instrucao = mem_instrucoes[end_mem];
endmodule
```

Figura 6. Implementação da Memória de Instruções em Verilog.

4.4.2. Registradores

Os registradores são armazenados em um Banco de Registradores, e precisam de Multiplexadores para funcionar além de seu Banco.

4.4.2.1. Banco de Registradores

O Banco de Registradores funciona praticamente todo de forma assíncrona, a única parte síncrona está na hora de efetuar a escrita em algum registrador, isso porque é necessário esperar que o dado fique pronto antes de ser escrito.

Possui uma Matriz de 32x32 que armazena todos os registradores com seus respectivos dados. Recebe três endereços de registradores: reg1,reg2 e reg3, mas só é possível escrever no reg3, portanto, instruções de escrita devem enviar o endereço para o reg3. Há um multiplexador que controla justamente qual endereço entrará no reg3. Há uma flag que, quando em alta, permite a escrita no registrador. A implementação do Banco de Registradores pode ser visualizada através da Figura 7.

```
// VARIAVEIS INTERNAS
parameter tamanho_palavra = 32;
parameter tam_memoria = 32;
integer i;

// INPUTS
input clk;
input [4:0] end_reg_1;
input [tamanho_palavra-1:0] dado_escrita;
input [4:0] end_reg_2;
input [4:0] end_reg_3;
input escrita;
// OUTPUTS
output [tamanho_palavra-1:0] saida_dado_1;
output [tamanho_palavra-1:0] saida_dado_2;
output [tamanho_palavra-1:0] saida_dado_3;
reg[tamanho_palavra-1:0] mem [0:tam_memoria-1];
always @ (posedge clk) begin
   if (escrita) begin
      mem[end_reg_3] = dado_escrita;
   end
end
assign saida_dado_1 = mem[end_reg_1];
assign saida_dado_2 = mem[end_reg_2];
assign saida_dado_3 = mem[end_reg_3];
endmodule
```

Figura 7. Implementação do Banco de Registradores.

4.4.2.2. Multiplexadores

Há dois multiplexadores acoplados próximos ao Banco de Registradores. O "Muxreg3" identifica qual endereço de registrador entrará no reg3. Para isso, recebe uma flag da Unidade de Controle e os endereços disponíveis nos espaços "15-11" e "25-21" das instruções. São os endereços identificados como Reg0 ou Reg2. Sua implementação em Verilog pode ser visualizada através da Figura 8.

```
module muxDadoReg3(
dado_1,
dado_2,
controle,
saida,
);

input [4:0] dado_1, dado_2;
input controle;

output reg [4:0] saida;

always @(dado_1 or dado_2 or controle) begin
   if(controle)
      saida = dado_2;
   else
      saida = dado_1;
end

endmodule
```

Figura 8. Multiplexador de definição de endereço do Registrador 3.

Há um outro multiplexador que define qual dado entrará para a escrita, este recebe dados da Unidade Lógica Aritmética, da Memória de Propósito Geral ou do Input. Caso o dado a ser escrito seja oriundo do Banco de Registradores, este passa pela Unidade Lógica Aritmética sem realizar nenhuma instrução. Há uma flag da Unidade de Controle para setar qual dos dados será enviado para o Banco de Registradores. Sua implementação pode ser visualizada através da Figura 9.

```
module muxRegistradores(
dado_ula,
dado_memoria,
controle,
saida,
dado_input
);

   input [31:0] dado_ula;
   input [31:0] dado_memoria;
   input [31:0] dado_input;
   input [1:0] controle;

   output reg [31:0] saida;
   always @ (*) begin
      if(controle == 2'b00)
         saida = dado_ula;
      else if (controle == 2'b01)
         saida = dado_memoria;
      else if (controle == 2'b11)
         saida = dado_input;
   end


endmodule
```

Figura 9. Implementação do multiplexador para dados de escrita nos registradores.

4.4.3. Extensor de Bits

O Extensor de Bits foi implementado devido a necessidade de transformar a extensão do sinal de 16 para 32 bits, quando assim é necessário. O código do Extensor de Bits pode ser visualizado na Figura 10.

Por se tratar de um módulo simples, não foi necessário adicionar nenhum sinal oriundo da Unidade de Controle para o funcionamento do mesmo.

```
module extensorDeBits(
entrada,
saida
);

input [15:0] entrada;
output reg [31:0] saida;

   always @ (entrada) begin
      if(entrada[15] == 1'b1) saida = entrada + 32'b11111111111111110000000000000000;
      else saida = entrada + 32'b00000000000000000000000000000000;
   end

endmodule
```

Figura 10. Extensor de Bits.

4.4.4. Divisor de Frequência

O divisor de frequência reduz a velocidade do clock na FPGA para ser possível a visualização de todos os passos do processamento.

A velocidade de Clock utilizada foi de 50MHz, o divisor divide essa frequência em "10000000" vezes. Sua implementação pode ser visualizada através da Figura 11.

```
module freqdivider(clk,reduzclock);
input clk;
output reduzclock;
reg [25:0]contador;

assign reduzclock = (contador == 10000000);
always @(posedge clk) begin
         contador <= contador+1'b1;
end

endmodule
```

Figura 11. Divisor de frequência em Verilog.

4.4.5. Unidade Lógica Aritmética

Esta unidade é responsável por efetuar todas as operações exceto a de somar endereços para a próxima instrução, que possui o seu somador próprio.

A Unidade Lógica Aritmética, ou ULA, recebe como entrada dois dados para realizar as operações, que podem vir tanto do Banco de Registradores quanto do Extensor de Bits, possui também uma entrada vinda do controle para identificar qual tipo de operação será realizada.

Nesta unidade é realizada operações de Aritmética, Lógica, Deslocamento e Branch. Seu código em Verilog pode ser visualizado através da Figura 12.

```
output reg [31:0] result;
output reg branch;

always @ (alu_op or dado_1 or dado_2) begin
        case (alu_op[5:0])
            // Aritmetica
            6'b000000: result = dado_1 + dado_2; // soma com 2 regs
            6'b000001: result = dado_1 + dado_2; // soma de imediato com um reg
            6'b000010: result = dado_1 - dado_2; // subtrai com 2 regs
            6'b000011: result = dado_1 - dado_2; // subtrai um imediato com um reg
            //logica
            6'b001000: result = dado_1 & dado_2; // AND
            6'b001001: result = dado_1 & dado_2; // ANDI
            6'b001010: result = dado_1 | dado_2; // OR
            6'b001011: result = dado_1 | dado_2; // ORI
            6'b001100: result = ~dado_1; // NOT
            6'b001101: result = dado_1 ^ dado_2; // XOR
            //deslocamento
            6'b010000: result = dado_1 << dado_2; // SLL
            6'b010001: result = dado_1 >> dado_2; // SRL
            6'b100010: result = dado_2;//MOV
            //branch
            6'b011000: if (dado_1 == dado_2) branch <= 1'b1;
            else branch <= 1'b0;
            6'b011001: if (dado_1 != dado_2) branch <= 1'b1;
            else branch <= 1'b0;
            6'b011010: if (dado_1 > dado_2) branch <= 1'b1;
            else branch <= 1'b0;
            6'b011011: if (dado_1 < dado_2) branch <= 1'b1;
            else branch <= 1'b0;
            6'b011100: branch <= 1'b1;
            6'b011101: branch <= 1'b1;
        endcase
    end
endmodule
```

Figura 12. Implementação da Unidade Lógica Aritmética em Verilog.

4.4.5.1. Multiplexador

Há um multiplexador para a Unidade Lógica Aritmética que define de onde virá o segundo operando. O primeiro operando sempre vêm do Banco de Registradores, o segundo pode vir tanto do Banco de Registradores quanto do Extensor de Bits. Sua implementação pode ser visualizada através da Figura 13.

```
module muxUla (
regOrExtensor,
saidaReg,
saidaExtensor,
saidaDado
);

input [31:0] saidaReg;
input [31:0] saidaExtensor;
input regOrExtensor;

output reg [31:0] saidaDado;

always @ (saidaReg or saidaExtensor) begin
   if(regOrExtensor == 0)
   saidaDado = saidaReg;
   else
   saidaDado = saidaExtensor;
end
endmodule
```

Figura 13. Multiplexador da Unidade Lógica Aritmética.

4.4.6. Memória de Propósito Geral

A Memória de Propósito Geral possui seu funcionamento semelhante ao Banco de Registradores, com a diferença de ser muito maior, e consequentemente mais lenta. Funciona de forma assíncrona, sendo a escrita como a única parte síncrona de seu funcionamento. Recebe um endereço para leitura ou escrita e envia o valor contido no endereço recebido. Sua implementação pode ser visualizada através da Figura 14.

```
module bancoDeMemoria(
clk,
endereco,
dado_escrita,
dado_saida,
escrita
);

input clk;
input escrita;
input [15:0] endereco;
input [31:0] dado_escrita;
output [31:0] dado_saida;

reg [31:0] mem_ram[100:0];

always @ (posedge clk) begin
   if(escrita) begin
      mem_ram[endereco] = dado_escrita;
   end
end
assign dado_saida = mem_ram[endereco];
endmodule
```

Figura 14. Implementação da Memória de Propósito Geral em Verilog.

4.4.7. Entrada de Dados

A entrada de dados consiste em sete chaves switch que representam os bits de um número em binário. O usuário alterna o valor de cada bit em Alto ou Baixo posicionando as chaves para cima ou para baixo, respectivamente. Os leds acima de cada chave identificam o valor da chave, quando ligados, em alto, e quando desligados, em baixo. Quando a chave de confirmação, a direita, está em alta, o dado é adicionado no processador. Todas estas chaves podem ser visualizadas através da Figura 15.

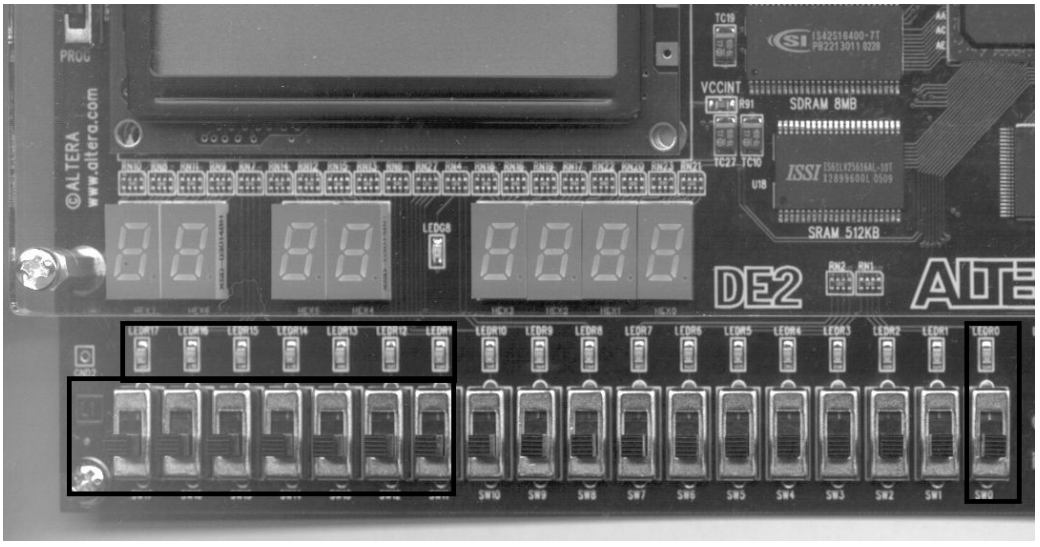

Figura 15. Disposição da entrada de dados na placa.

Sua implementação foi realizada utilizando os inputs descritos acima. O dado descrito através das chaves é convertido de 7 para 32 bits no próprio módulo de entrada. Como saída, além do dado, há uma flag que envia para a unidade de controle que o dado foi adicionado. Sua implementação pode ser visualizada através da Figura 16.

```
module entradaDeDados(
chave,
dado_reg,
botao,
flag,
);

input [7:0] chave;
input botao;

output reg [31:0] dado_reg;
output reg flag;

always @ (botao) begin
      if (chave[7] == 0) begin
         dado_reg = chave + 32'd0;
      end
      else begin
         dado_reg = chave + 32'b11111111111111111111111100000000;
      end
      flag = botao;
end
endmodule
```

Figura 16. Entrada de dados implementada em Verilog.

4.4.8. Saída de dados

A Saída de dados utiliza de oito displays de sete segmentos que podem ser visualizados na Figura 17. Cada um dos displays unidos mostram um único dado. Sendo assim, podemos mostrar dados de dois ou quatro dígitos e até três números diferentes por vez.

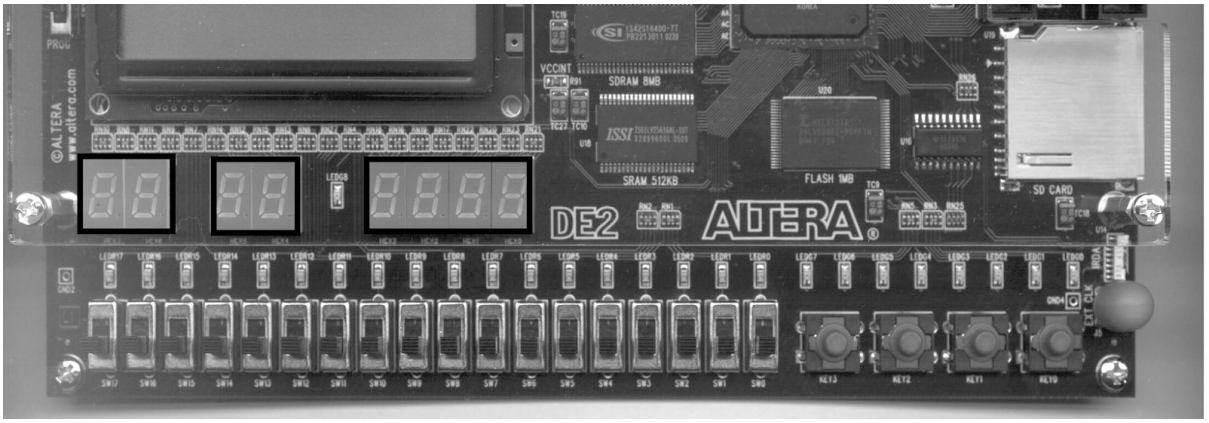

Figura 17. Displays de saída.

É realizada através de diversos outros módulos acoplados. Inicialmente, há o módulo que faz a comunicação entre o processador e os Displays, que funciona de forma síncrona e recebe uma flag de controle da Unidade de Controle, que verifica se a saída se dá utilizando apenas um Display ou os três displays. Sua implementação pode ser visualizada através da Figura 18.

```
always @ (posedge clk) begin
   if (controle == 2'b01) begin
      saida_1 = dado_reg_1;
      saida_2 = dado_reg_2;
      saida_result = dado_reg_result;
   end
   else if (controle == 2'b10) begin
      saida_result = dado_reg_result;
   end
end
```

Figura 18. Saída de dados

4.4.8.1. Conversor Binário Decimal

Para que o display de sete segmentos mostre o número corretamente, é necessário convertê-lo de binário para decimal. Há dois tipos de conversores, para dois ou quatro displays, a única diferença entre eles é o acréscimo da centena e do milhar nos quatro displays, onde não há nos dois displays. Sua implementação pode ser visualizada através da Figura 19.

```
always @ (numero) begin
    milhar = 4'b0000;
    centena = 4'b0000;
    dezena = 4'b0000;
    unidade = 4'b0000;
    if(numero[15] == 0) begin
        for(i = 15; i >= 0; i = i-1) begin
            if(milhar >= 5) milhar = milhar + 3;
            if(centena >= 5) centena = centena + 3;
            if(dezena >= 5) dezena = dezena + 3;
            if(unidade >= 5) unidade = unidade + 3;
            milhar = milhar << 1;
            milhar[0] = centena[3];
            centena = centena << 1;
            centena[0] = dezena[3];
            dezena = dezena << 1;
            dezena[0] = unidade[3];
            unidade = unidade << 1;
            unidade[0] = numero[i];
        end
    end
    else begin
        aux = ~(numero) + 16'b0000000000000001;
        for(i = 15; i >= 0; i = i-1) begin
            if(milhar >= 5) milhar = milhar + 3;
            if(centena >= 5) centena = centena + 3;
            if(dezena >= 5) dezena = dezena + 3;
            if(unidade >= 5) unidade = unidade + 3;
            milhar = milhar << 1;
            milhar[0] = centena[3];
            centena = centena << 1;
            centena[0] = dezena[3];
            dezena = dezena << 1;
            dezena[0] = unidade[3];
            unidade = unidade << 1;
            unidade[0] = aux[i];
        end
    end
end
```

Figura 19. Conversor implementado em Verilog.

4.4.8.2. Conversor Decimal Display

Os números em binário são convertidos para decimal e, em seguida, são enviados para o conversor que seta os valores do Display de sete segmentos como alto ou baixo de acordo com o número que ele recebeu. Os números são enviados separadamente para a unidade, dezena, centena e milhar. Sua implementação para a "unidade" pode ser visualizada através da Figura 20. Todos os demais campos seguem o mesmo formato.

```
always @ (dezena or unidade or centena or milhar) begin
   case (dezena)
         4'b0000 : D = 7'b0000001;
         4'b0001 : D = 7'b1001111;
         4'b0010 : D = 7'b0010010;
         4'b0011 : D = 7'b0000110;
         4'b0100 : D = 7'b1001100;
         4'b0101 : D = 7'b0100100;
         4'b0110 : D = 7'b0100000;
         4'b0111 : D = 7'b0001111;
         4'b1000 : D = 7'b0000000;
         4'b1001 : D = 7'b0000100;
         default : D = 7'b1111111;
   endcase
```

Figura 20. Implementação do conversor Decimal Display em Verilog.

4.4.9. Unidade de Controle

A unidade de controle, como o próprio nome ja diz, controla todos os módulos de dentro do processador. Como o processador foi projetado para realizar suas tarefas em um único ciclo de clock, é possível que a Unidade de Controle seja muito mais simples, uma vez que pode setar os valores das flags de acordo com o Opcode em cada ciclo de clock.

4.4.9.1. Flags

Foi implementado diversas flags que atuam como controladoras de todos os blocos do processador, seus nomes e significados podem ser visualizados na Tabela 11.

Tabela 11. Flags da unidade de controle.

Nome	Função
pc_reset	Reseta o PC para o endereço inicial quando está em alto.
mux_branch_instrucao	Controla se o endereço da próxima instrução mudará ou não, e se mudar, controla se o endereço virá do somador ou do branch.
mux_branch_ula_extensor	Controla se o endereço do branch virá da ULA ou do Extensor de Bits.
memoria_registradores_escrita	Quando está em alto, o Banco de Registradores permite escrita.
mux_registradores_dado_escrita	Controla qual dado irá para a escrita no banco de registradores: Input, Memória ou ULA.
mux_ula_regOrExtensor	Controla se o dado_2 que entra na ULA é oriundo do Banco de Registradores ou do Extensor de Bits.
ula_op	Controla a operação da ULA.
escrita_banco_memoria	Quando está em alto, a Memória de propósito geral permite escrita.
saida_dados	Controla se a saída de dados está ligada e quais dados estão sendo mostrados.
mux_reg3	Controla de onde sai o endereço do registrador que será escrito.
flag_entrada	Input que é ativada quando a entrada é realizada.

4.4.9.2. Sinais de acordo com o Opcode

Cada instrução possui flags específicas que indicam qual “caminho” o processador seguirá, que pode ser vista através da Tabela 2.

Tabela 12. Ativação das flags de acordo com a instrução.

Instrução	Opcode	pc_r	mbi	mbue	mre	mrde	muroe	uop	ebm	sd	mr3
ADD	000000	0	00	x	1	00	0	000000	0	00	0
ADDI	000001	0	00	x	1	00	1	000001	0	00	0
SUB	000010	0	00	x	1	00	0	000010	0	00	0
SUBI	000011	0	00	x	1	00	1	000011	0	00	0
AND	001000	0	00	x	1	00	0	001000	0	00	0
ANDI	001001	0	00	x	1	00	1	001001	0	00	0
OR	001010	0	00	x	1	00	0	001010	0	00	0
ORI	001011	0	00	x	1	00	1	001011	0	00	0
NOT	001100	0	00	x	1	00	0	001100	0	00	0
XOR	001101	0	00	x	1	00	0	001101	0	00	0
SLL	010000	0	00	x	1	00	1	010000	0	00	0
SRL	010001	0	00	x	1	00	1	010001	0	00	0
BEQ	011000	0	11	1	0	xx	0	011000	0	00	0
BNE	011001	0	11	1	0	xx	0	011001	0	00	0
BGT	011010	0	11	1	0	xx	0	011010	0	00	0
BLT	011011	0	11	1	0	xx	0	011011	0	00	0
JMP	011100	0	11	0	0	xx	x	011100	0	00	0
JMPR	011101	0	11	0	0	xx	x	011101	0	00	0
LW	100000	0	00	x	1	01	1	100000	1	00	0
STORE	100001	0	00	x	0	xx	x	xxxxxx	1	00	0
MOV	100010	0	00	x	1	00	0	100010	0	00	0
NOP	101000	0	00	x	0	xx	x	xxxxxx	0	00	0
RESET	101010	1	00	x	0	xx	x	xxxxxx	0	00	0
IN	111000	0	10	x	1	11	x	xxxxxx	0	00	0
OUT	111001	0	00	x	0	xx	x	xxxxxx	0	10	0
OUT3	111010	0	00	x	0	xx	x	xxxxxx	0	01	1

4.4.9.3. Implementação em Verilog

Após a projeção sobre a ativação das flags, foi dado início a implementação em Verilog da Unidade de Controle. Inicialmente, foi declarado as flags de acordo com os nomes listados na Tabela 11, as entradas e saídas podem ser visualizadas através da Figura 21.

```
module unidadeControle(
clk,
opcode,
pc_reset,
mux_branch_instrucao,
mux_branch_ula_extensor,
memoria_registradores_escrita,
mux_registradores_dado_escrita,
mux_ula_regOrExtensor,
ula_op,
escrita_banco_memoria,
saida_dados,
mux_reg3,
flag_entrada,
);

    input [5:0] opcode;
    input clk, flag_entrada;

    //FLAGS
    output reg pc_reset, mux_branch_ula_extensor,
    memoria_registradores_escrita, mux_ula_regOrExtensor,
    escrita_banco_memoria, mux_reg3;
    output reg [1:0] saida_dados, mux_branch_instrucao, mux_registradores_dado_escrita;
    output reg [5:0] ula_op;
```

Figura 21. Entradas e Saídas da Unidade de Controle em Verilog.

Foram utilizadas variáveis locais para o controle e organização da Unidade de Controle, que podem ser visualizadas através da Figura 22.

O reg Pausa é utilizado quando a instrução de Input está ativada e é necessário esperar o usuário pressionar o botão de Input para dar continuidade ao processamento.

As demais variáveis locais são utilizadas apenas para armazenar o opcode de cada instrução de forma a organizar e facilitar a implementação.

```
reg PAUSA = 1'b0;

//INSTRUCOES ARITMETICAS
localparam [5:0] ADD = 6'b000000, ADDI = 6'b000001, SUB = 6'b000010, SUBI = 6'b000011;

//INSTRUÇÕES DE LÓGICA
localparam [5:0] AND = 6'b001000, ANDI = 6'b001001, OR = 6'b001010, ORI = 6'b001011, NOT = 6'b001100, XOR = 6'b001101;

//INSTRUÇOES DE DESLOCAMENTO
localparam [5:0] SLL = 6'b010000, SRL = 6'b010001;

//INSTRUÇOES DE BRANCH
localparam [5:0] BEQ = 6'b011000, BNE = 6'b011001, BGT = 6'b011010, BLT = 6'b011011, JMP = 6'b011100, JMPR = 6'b011101;

//INSTRUÇOES DE DADOS
localparam [5:0] LW = 6'b100000, STORE = 6'b100001, MOV = 6'b100010;

//INSTRUÇÕES DE CONTROLE
localparam [5:0] NOP = 6'b101000, RESET = 6'b101010;

//INSTRUÇÕES I/O
localparam [5:0] OUT3 = 6'b111010, IN = 6'b111000, OUT = 6'b111001;
```

Figura 22. Variáveis locais da Unidade de Controle em Verilog.

Toda instrução é decodificada após um comando always, que funciona em cada negedge clock. A escolha do negedge fez-se necessário para que o processador funcione com um único ciclo de clock.

Sempre que entra no always, é verificado se a flag PAUSA está em alto, se estiver, é verificado se a entrada foi pressionada, em caso de positivo, a flag PAUSA é setada para baixo e o processador retorna a próxima instrução, caso negativo, o processador mantém a flag ligada e aguarda o próximo negedge clock para realizar a verificação novamente. Essa implementação pode ser visualizada através da Figura 23.

```
always @(negedge clk) begin
    if(PAUSA == 1'b1) begin
        PAUSA = ~flag_entrada;
        if (flag_entrada == 1'b1)
            mux_branch_instrucao = 2'b00;
    end
    else begin
        case (opcode[5:0])
            BLT: begin
```

Figura 23. Implementação da decodificação da Unidade de Controle.

Todas as instruções apenas setam suas flags correspondentes na Unidade de Controle. Por exemplo, o add, que seta o "percurso" de seu dado saindo de dois registradores, passando pela ULA, somando e enviando seu Opcode por meio da flag da ULA e alterando a flag de escrita nos registradores como alta, como pode ser visualizado na Figura 24.

```
ADD: begin
   pc_reset = 1'b0;
   mux_branch_instrucao = 2'b00;
   mux_branch_ula_extensor = 1'bx;
   memoria_registradores_escrita = 1'b1;
   mux_registradores_dado_escrita = 2'b00;
   mux_ula_regOrExtensor = 1'b0;
   ula_op = opcode;
   escrita_banco_memoria = 1'b0;
   saida_dados = 2'b00;
   mux_reg3 = 1'b0;
end
```

Figura 24. Decodificação em flags da instrução ADD.

A instrução Input é uma instrução especial, que para todo o processamento até que o dado esteja disponível para armazenar no registrador correspondente e prosseguir o processamento, por isso, ela seta a flag local PAUSE como alta, além de distribuir as flags correspondentes conforme necessário e conforme mostra a Tabela 2. A implementação da instrução Input pode ser visualizada na Figura 25.

```
IN: begin
    pc_reset = 1'b0;
    mux_branch_instrucao = 2'b10;
    mux_branch_ula_extensor = 1'bx;
    memoria_registradores_escrita = 1'b1;
    mux_registradores_dado_escrita = 2'b11;
    mux_ula_regOrExtensor = 1'bx;
    ula_op = opcode;
    escrita_banco_memoria = 1'b0;
    saida_dados = 2'b00;
    mux_reg3 = 1'b0;
    PAUSA = 1'b1;
end
```

Figura 25. Instrução Input decodificada em Verilog.

5. Resultados

Foi realizado algumas simulações para comprovar a funcionalidade da unidade de controle e dos demais módulos. Em seguida, foi testado alguns algorítmos na FPGA que utilizassem algumas instruções em looping e uma certa interação com o usuário.

5.1. Testes em Forma de Onda

Nestes testes, não foi utilizado a FPGA, apenas a forma de onda demonstrada através do Software.

5.1.1. Instruções Aritméticas

Inicialmente, foi utilizado um programa simples que utiliza instruções aritméticas. Foi adicionado 1 no registrador 00001 através da instrução ADDI, em sequência foi somado o valor dele duas vezes através da instrução ADD. Sua implementação na memória de instruções pode ser visualizada na Figura 26.

```
always @ (posedge clk) begin
    if (clockInicio == 0) begin
        //PROGRAMA ESCRITO:

        mem_instrucoes[0] = 32'b10100000000000000000000000000000;//INSTR VAZIA
        mem_instrucoes[1] = 32'b00000100001000000000000000000001;//adiciona 1 no reg 00001 - ADDI
        mem_instrucoes[2] = 32'b00000100001000000000000000000000;//adiciona 0 no reg 00001 - ADDI
        mem_instrucoes[3] = 32'b00000000001000010000100001xxxxxxxxxx;//adiciona o valor de 00001+00001
        mem_instrucoes[4] = 32'b00000100001000000000000000000000;//adiciona 0 no reg 00001 - ADDI
        clockInicio <= 1'b1;
    end
end
```

Figura 26. Programa de testes de instruções aritméticas.

É possível visualizar através da Figura 27 a forma de onda desta simulação que representam o resultado esperado. as somas que utilizam imediato com valor “0” são utilizadas para ficar visualizar o valor que se encontra no registrador logo após a operação.

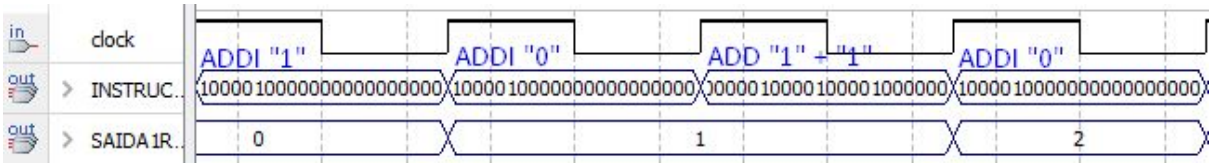

Figura 27. Forma de onda dos testes de instruções aritméticas.

5.1.2. Branch

Para testar instruções de branch, foi escrito um programa que soma “1” no registrador “00001” e no registrador “00010”, em sequência foi utilizada a instrução BEQ para verificar se os registradores eram iguais, caso o branch não fosse tomado, era somado “1” no registrador, em sequência, foi impresso o valor do registrador, como pode ser visualizado na Figura 28.

```
mem_instrucoes[0] = 32'b10100000000000000000000000000000;//INSTR VAZIA
mem_instrucoes[1] = 32'b00000100001000000000000000000001;//adiciona 1 no reg 00001 - ADDI
mem_instrucoes[2] = 32'b00000100010000000000000000000001;//adiciona 1 no reg 00010 - ADDI
mem_instrucoes[3] = 32'b01100000001000100000000000000101;//BEQ 00001 00010 IR PARA 000101
mem_instrucoes[4] = 32'b00000100001000000000000000000001;//adiciona 1 no reg 00001 - ADDI
mem_instrucoes[5] = 32'b00000100001000000000000000000000;//adiciona 0 no reg 00001 - ADDI
clockInicio <= 1'b1;
```

Figura 28. Programa implementado para teste na instrução Branch Equal.

Como pode ser visualizado na Figura 29, o resultado da instrução em formas de onda foi o esperado, uma vez que a instrução BEQ “pulou” a instrução ADDI “1” e o valor final contido no registrador foi “1”, adicionado na 2ª linha.

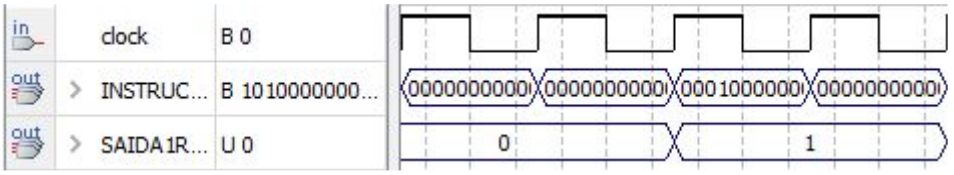

Figura 29. Forma de onda após a execução do algorítmo Branch.

5.2. Testes na FPGA

Nestes testes, foi escrito um programa completo, com input e output, e testado na FPGA seu resultado.

5.2.1. Multiplicador

Foi escrito um código para efetuar a multiplicação entre dois números, mostrando o passo a passo através dos Displays.

Inicialmente, o usuário deve entrar com os números a serem multiplicados. Após a chave switch de confirmação ser acionada, o número é mostrado nos displays de sete segmentos para confirmação visual. Na Figura 30 podemos visualizar a entrada do número "4".

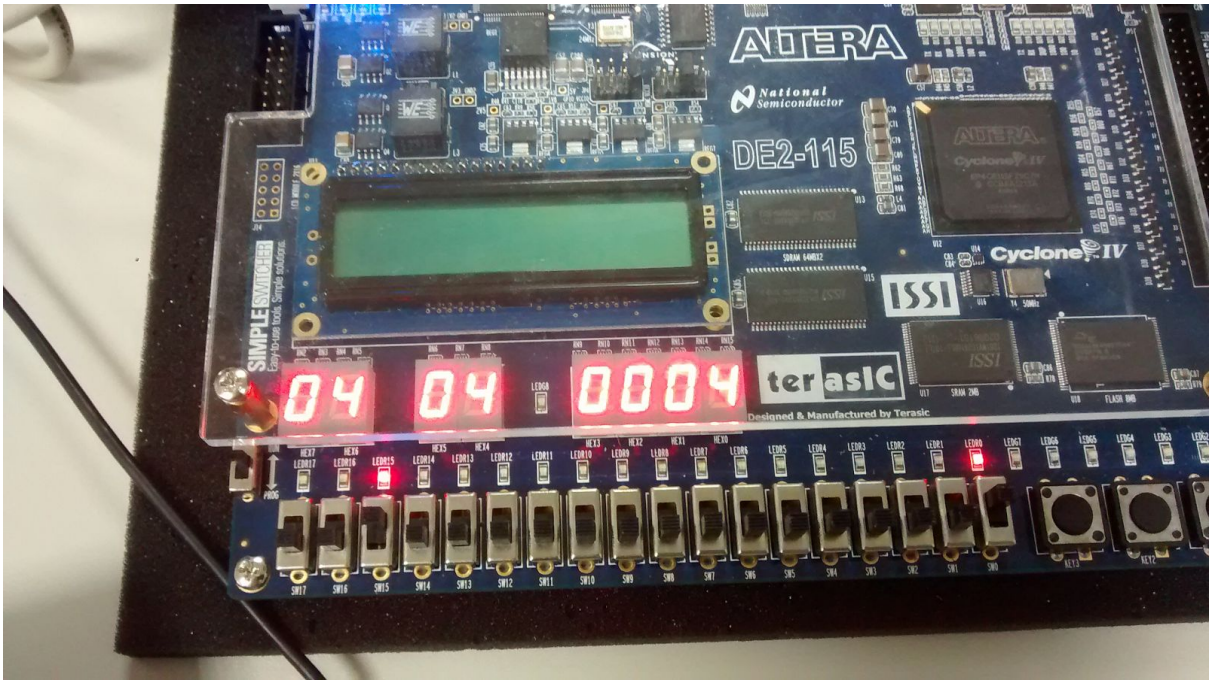

Figura 30. Input do número 4.

Após o usuário digitar os dois números, inicia-se o processo de multiplicação, é mostrado no display de sete segmentos o passo a passo de cada multiplicação, o início do processo pode ser visualizado através da Figura 31, onde o usuário entrou com o número "8" e, logo após, com o número "4".

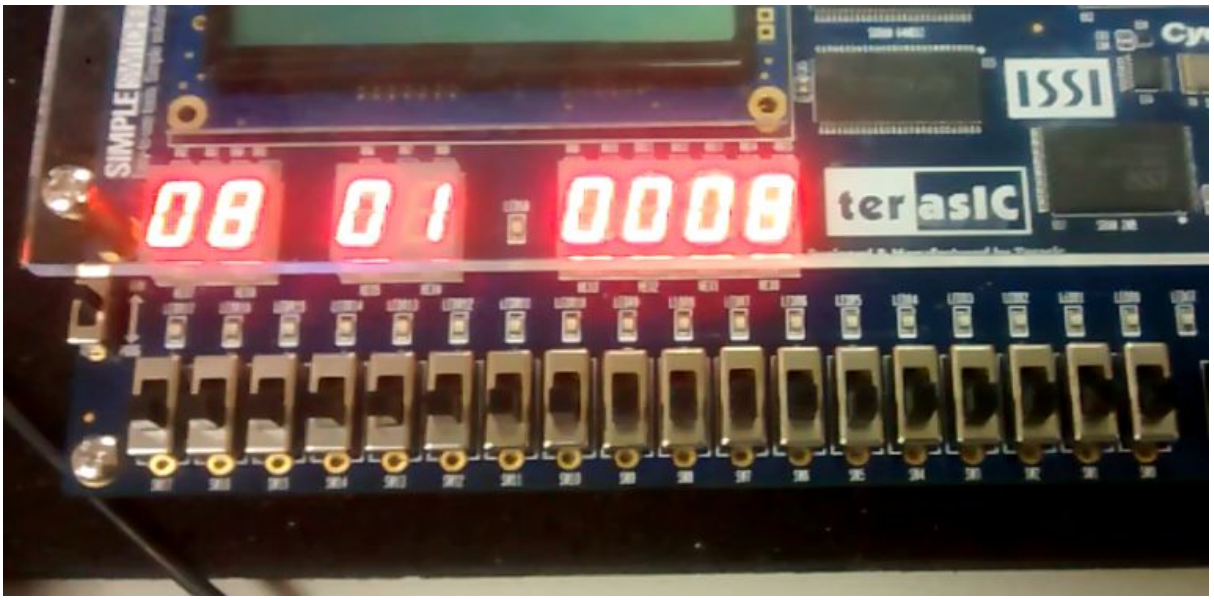

Figura 31. Início da multiplicação.

Após o passo a passo mostrar a multiplicação de “8x1”, “8x2” e “8x3”, o visor mostrou o resultado que o usuário esperava, “8x4”, que pode ser visualizado através da Figura 32.

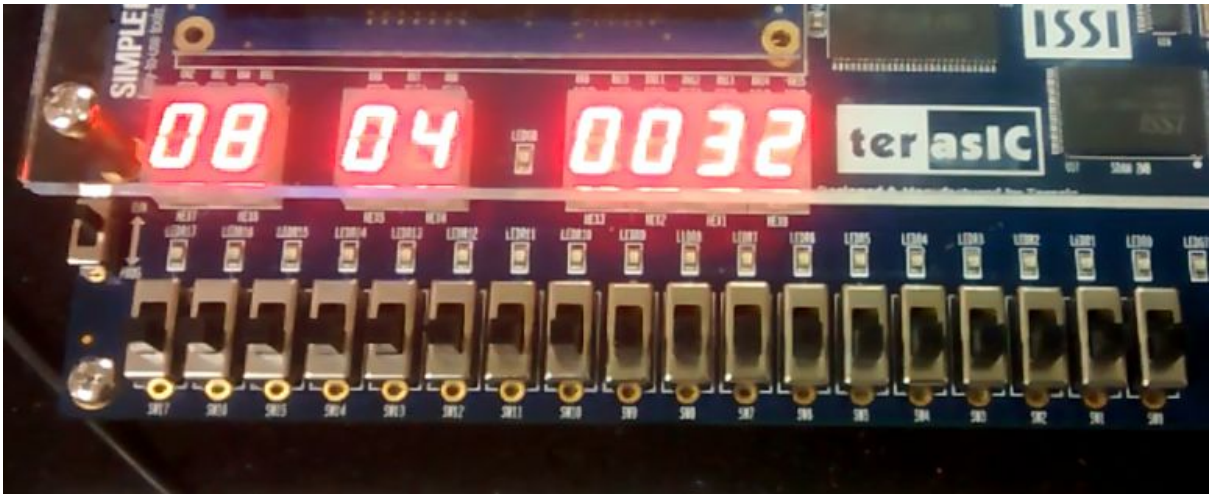

Figura 32. Fim da multiplicação, com o resultado esperado sendo mostrado no display.

Como pode-se observar, a FPGA mostrou corretamente o resultado esperado para a multiplicação. O código deste programa pode ser visualizado na Memória de Instruções, implementado em Verilog, através da Figura 33.

```
//MULTIPLICADOR

  mem_instrucoes[0] = 32'b10100000000000000000000000000000; // instrucao vazia
  mem_instrucoes[1] = 32'b11100000001xxxxxxxxxxxxxxxxxxxx; // input no reg 00001
  mem_instrucoes[2] = 32'b11101000001000010000 1xxxxxxxxxx; //OUTPUT REG 00001
  mem_instrucoes[3] = 32'b11100000010xxxxxxxxxxxxxxxxxxxx; // input no reg 00010
  mem_instrucoes[4] = 32'b11101000010000100010xxxxxxxxxx; //OUTPUT REG 00010
  mem_instrucoes[5] = 32'b00000100100000000000000000000000; // add 0 no reg 00100
  mem_instrucoes[6] = 32'b00000100011000000000000000000001; // add 1 no contador 00011
  mem_instrucoes[7] = 32'b00000000001001000010 0xxxxxxxxxx; // add valor dos dois antigos no atual
  mem_instrucoes[8] = 32'b10000000100000000000000000000001; // SALVA NA MEMORIA O VALOR DO REG 0010
  mem_instrucoes[9] = 32'b10000100100000000000000000000001; // CARREGA DA MEMORIA 01 NO REG 00100
  mem_instrucoes[10] = 32'b11101000001000110010 0xxxxxxxxxx; //OUTPUT REG 00100
  mem_instrucoes[11] = 32'b01100100011000100000000000000110; // BNE para verificar se ja acabou
  //RESETANDO AS INSTRUÇÕES PARA REPETIR O LAÇO
  mem_instrucoes[12] = 32'b00000100010000100000100xxxxxxxxxx; //SUB
  mem_instrucoes[13] = 32'b00000100001100011000 11xxxxxxxxxx; //SUB
  mem_instrucoes[14] = 32'b11101000011000110001 1xxxxxxxxxx; //OUTPUT REG 00100
  mem_instrucoes[15] = 32'b01110000000000000000000000000001; // JMP 1
```

Figura 33. Implementação do multiplicador em Verilog.

5.2.2. Sequência de Fibonacci

A Sequência de Fibonacci trás sempre como valor atual a soma entre os dois últimos números, quando o valor atual é maior que “1”. Se o valor atual for “0” ou “1”, o valor da sequência de Fibonacci passa a ser ele mesmo.

Foi implementado um algorítmo em linguagem de máquina que efetua o cálculo da sequência de Fibonacci. Sua implementação pode ser visualizada através da Figura 34.

```
// FIBONACCI

mem_instrucoes[0] = 32'b10100000000000000000000000000000;  // instrucao vazia
mem_instrucoes[1] = 32'b00000100001000000000000000000001;  // ADDI 1 NO 00001
mem_instrucoes[2] = 32'b11100000010xxxxxxxxxxxxxxxxxxxx;  // input no reg 00010
mem_instrucoes[3] = 32'b11101000010000100010xxxxxxxxxx;  // OUTPUT REG 00010 - armazena resultado que o usuario quer
mem_instrucoes[4] = 32'b01100000000000100000000000010000;  // BEQ 00000 00010
mem_instrucoes[5] = 32'b01100000001000100000000000010010;  // BEQ 00001 00010
mem_instrucoes[6] = 32'b00000100100000000000000000000000;  // SETA INICIALMENTE O VALOR DO ANTIGO 2 para 0 - 00100
mem_instrucoes[7] = 32'b00000100101000000000000000000001;  // SETA INICIALMENTE O VALOR DO ANTIGO 1 para 1 - 00101
mem_instrucoes[8] = 32'b00000101000000000000000000000010;  // SETA INICIALMENTE O CONTADOR PARA 2 - 01000
// INICIA O LAÇO DO FIBO PARA NUMEROS DIFERENTES DE 0 E 1
mem_instrucoes[9] = 32'b00000000100001010011 1xxxxxxxxxx;  // REGISTRADOR DE FIBO ATUAL 00111 = 00100 + 00101
mem_instrucoes[10] = 32'b11101000100001010011 1xxxxxxxxxx; // IMPRIME FIBO ATUAL 00111
mem_instrucoes[11] = 32'b00000101000000000000000000000001; //ACRESCENTA +1 NO CONTADOR - 01000
mem_instrucoes[12] = 32'b01100000010010000000000000010011; //VERIFICA SE CONTADOR E IGUAL AO 00010, SE SIM PULA PRO HALT
mem_instrucoes[13] = 32'b10001000100001 01xxxxxxxxxxxxxxx; //COLOCA NO ANTIGO 2 - 00100 - O VALOR DO ANTIGO 1
mem_instrucoes[14] = 32'b10001000101001 11xxxxxxxxxxxxxxx; //COLOCA NO ANTIGO 1 - 00101 - O VALOR DO ATUAL
mem_instrucoes[15] = 32'b01110000000000000000000000001001; //JUMP PRO INICIO DO LAÇO
mem_instrucoes[16] = 32'b11101000000000000000 0xxxxxxxxxx; //IMPRIME ZERO
mem_instrucoes[17] = 32'b01110000000000000000000000010011; //JUMP PRO HALT
mem_instrucoes[18] = 32'b11101000001000010000 1xxxxxxxxxx; //IMRPIME UM
mem_instrucoes[19] = 32'b101001xxxxxxxxxxxxxxxxxxxxxxxxxx; //HALT
```

Figura 34. Sequência de Fibonacci.

Para este teste, o usuário entra com a posição da sequência desejado, e é efetuado os cálculos um a um até aquela posição. É mostrado nos displays de sete segmentos o valor dos três últimos números da sequência, sendo o último o atual. O processo para quando a instrução atinge a posição desejada.

É possível visualizar através da Figura 35 o usuário entrando com o número “8”, posição desejada do Fibonacci.

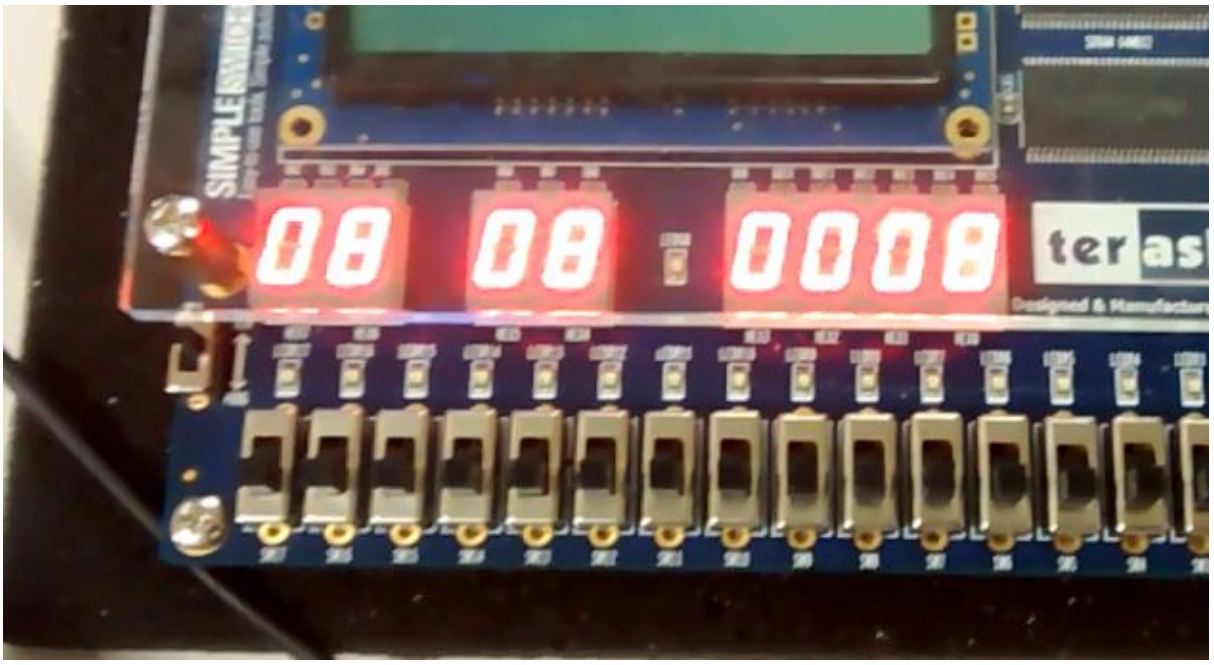

Figura 35. Entrada do número desejado da sequência de Fibonacci.

O processo inicia-se mostrando os valores à partir da posição “3” da sequência, sendo assim, é mostrado nos displays o valor da posição “1”, “2” e “3”, como pode-se observar através da Figura 36.

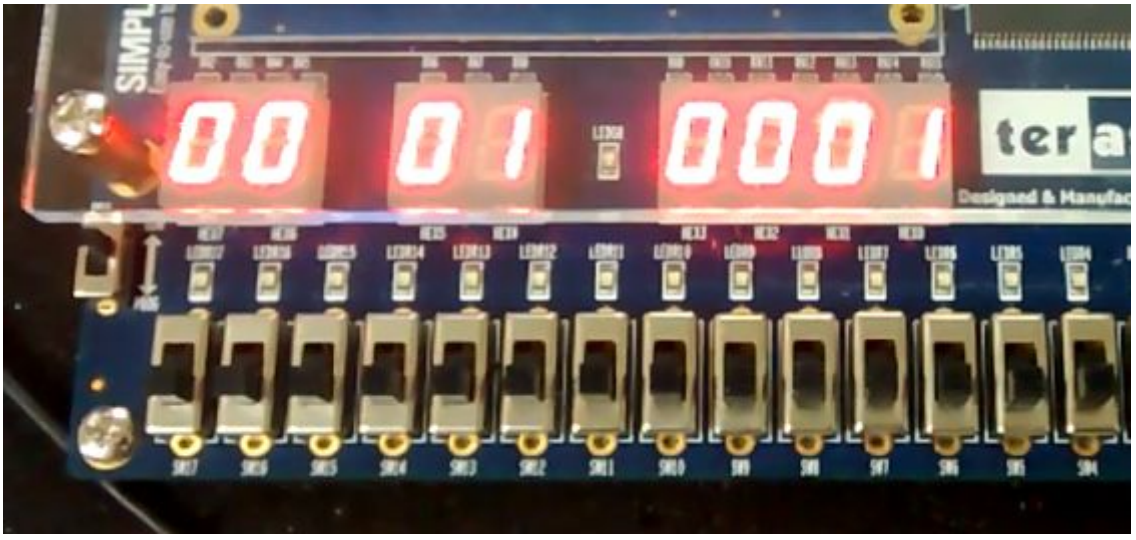

Figura 36. Sequência de Fibonacci na posição “3”.

O processo continua e é calculado o valor para as próximas posições, o processo chega no fim quando os displays mostram os valores “5”, “8” e “13”, os 6º, 7º e 8º valores da sequência, como pode ser visualizado através da Figura 37.

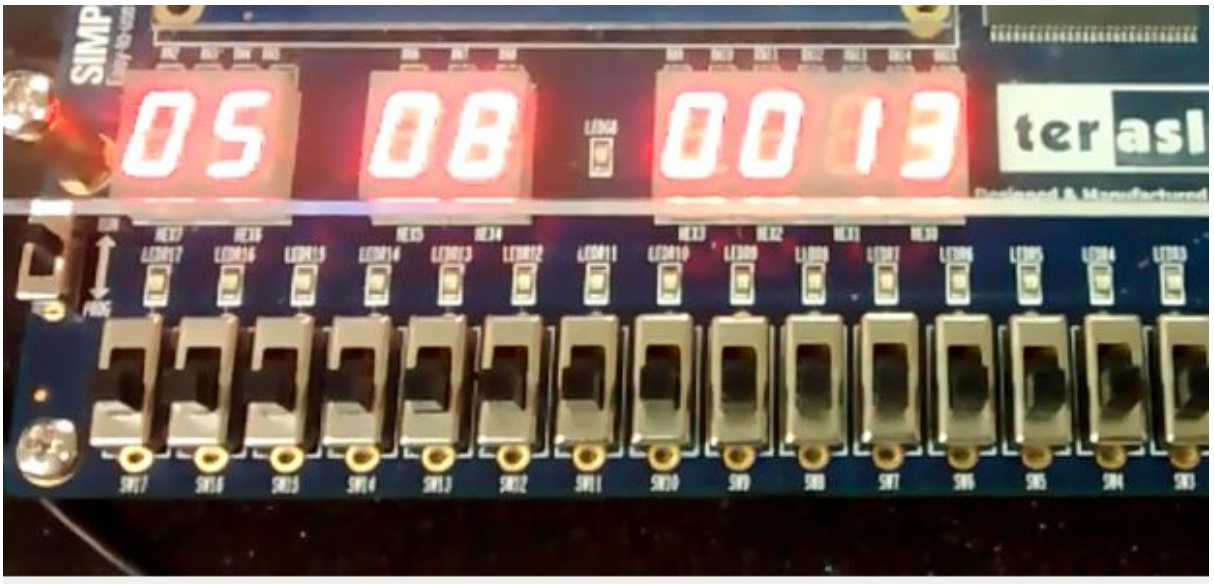

Figura 37. Valores da sequência de Fibonacci para “6”, “7” e “8”.

6. Conclusão

Após os testes e resultados, é possível concluir que o processador funcionou corretamente. Todas as suas instruções foram testadas e funcionam perfeitamente. Os algorítmos implementados também demonstraram seu correto funcionamento na FPGA.

7. Referências

1. http://www.columbia.edu/cu/computinghistory/eniac.html - Acesso em 01/04/2016
2. https://www.cise.ufl.edu/~mssz/CompOrg/CDA-proc.html - Acesso em 28/03/2016
3. Stallings, William. Computer organization and architecture: designing for performance. 9.ed.
4. TANENBAUM, Andrew S. Organização estruturada de computadores. 5ª ed. São Paulo: Pearson Prentice Hall, 2007.
5. HENNESSY, J.L. and PATTERSON, D.A. Arquiteturade Computadores: Uma Abordagem Quantitativa. 3ª ed. Rio de Janeiro: Campus, 2003.
6. Flynn, Michael J. (1995). *Computer architecture: pipelined and parallel processor design* [S.l.: s.n.] pp.

Anexo A - Datapath