Projeto Final
ZeptoProcessador-V

Grupo C5
Gabriel Felix de Lima, 21/1010332
Gabriel Xisto Barros, 21/1055503
Marcos Alexandre da Silva Neres, 21/1055334

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CIC0231 - Laboratório de Circuitos Lógicos
September 30, 2022

gabfelixlima@outlook.com, gxistobarros2@gmail.com, marcos.alex2004@gmail.com

Abstract. *This report addresses the implementation of a simple 16-bit processor with an ISA similar to RISC-V, implementing only a set of 11 instructions.*
The processor consists of 7 modules. After writing their hardware descriptions in SystemVerilog, putting them together, testing their waveforms in Quartus-II and synthesizing them in a DE2 FPGA development kit, we got the expected result: a simple programmable 16-bit processor.
The project passed all the tests given through programs created to test its capabilities: multiplication, division, obtaining the greatest common divisor, among other similar tests.

Resumo. *Este trabalho aborda a implementação de um simples processador de 16 bits com uma ISA similar à do RISC-V, implementando somente um conjunto de 11 instruções.*
O processador é composto por 7 módulos. Após escrever sua descrição em SystemVerilog, juntá-los, testar suas formas de onda no software Quartus-II e sintetizá-los em um kit de desenvolvimento FPGA DE2, obtivemos o resultado esperado: um simples processador de 16 bits programável.
O projeto passou todos os testes dados através de programas criados para testar suas capacidades: multiplicação, divisão, obtenção do máximo divisor comum, entre outros teste similares.

1. Introdução

Processadores são um dos componentes mais importantes de um computador. São frequentemente descritos como o seu "cérebro". Seu trabalho é carregar e executar instruções de sua memória, realizar cálculos, comparações e outras operações lógicas. O presente trabalho detalha a implementação do "ZeptoProcessador-V"em Verilog de acordo com a especificação fornecida em aula. Ele possui 11 instruções, 16 registradores, memória ROM, utiliza palavras de 16 bits e é programável, sendo possível carregar programas simples nele.

1.1. Objetivos

É necessário que sejam implementados 7 módulos: o registrador PC *(Program Counter)*, responsável por armazenar o endereço da instrução a ser executada, a memória ROM para

carregar programas, o banco de registradores, a unidade lógico-aritmética, o comparador, o bloco de controle, e por fim o módulo de monitoramento e controle.

O projeto objetiva a implementação correta e condizente com a especificação do processador em questão, através da implementação e conexão de todos os seus componentes supracitados. Isto é, visamos aplicar todo o conhecimento adquirido em aula através da criação um processador de 16 bits simples programável, implementado em Verilog e executado em um kit de desenvolvimento FPGA DE2.

2. Metodologia

Utilizamos o Quartus-II e a linguagem de descrição de hardware SystemVerilog para a realização de todo o processador. Durante testes do funcionamento da máquina, empregamos o kit de desenvolvimento FPGA DE2. A interação com o usuário é feita por meio de uma memória ROM na qual se carrega o programa a ser executado e *switches* de reset e alternação de clock manual/automático. A saída utiliza os próprios *displays* da placa para mostrar os resultados ou o endereço da memória da instrução atual. O projeto foi constituído em diversos módulos, juntados para formar o processador.

2.1. ULA e comparador

Iniciamos o projeto através da construção do comparador e da ULA (Unidade Lógica-Aritmética), por se tratarem de circuitos combinacionais simples, independentes da sincronicidade do processador.

Figura 1. Esquema do Comparador, retirado do roteiro

Inicialmente, o comparador foi feito somente recebendo as entradas A e B de 16 bits e a partir das comparações nativas de Verilog em comandos *assign*, da forma descrita abaixo:

```
module comparador (
input [15:0] A,
input [15:0] B,
output [3:0 S);
assign S[3] = (A == B);
assign S[2] = (A != B);
assign S[1] = (A >=) B);
assign S[0] = (A < B);
endmodule
```

Entretanto, durante a interligação dos componentes do processador notamos que a comparação não seria bem sucedida quando incluídos números negativos em complemento de dois, portanto utilizamos o comando *always* para modificar a comparação de

acordo com o bit mais significativo das entradas - que identifica o sinal. Quando ambos os bits mais significativos são 0 ou 1, a comparação ocorre normalmente, mas caso A[15] = 0 e B[15] = 1, a saída S se apresenta como 0110, e se A[15] = 1 e B[15] = 0, a saída se apresenta como 0101.

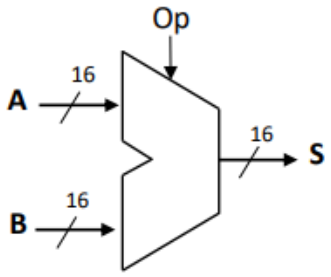

Figura 2. Esquema da Unidade Lógica-Aritmética, retirado do roteiro

A ULA se apresenta facilmente descrita em Verilog, uma vez que a linguagem de descrição já possui todos os comandos (adição, subtração, OR, AND e XOR) presentes nativamente e o sinal do número não acarreta em falhas. Portanto, apresentando a entrada de escolha Op de 3 bits e as duas entradas A e B de 16 bits, utilizamos um comando *always* para realizar uma operação de acordo com a entrada Op.

É importante notar que, embora apenas apenas os dígitos 000 a 100 representassem uma operação da ULA, ainda precisamos descrever seu comportamento nos intervalos 101-111, de forma que ela não se comporte como um circuito sequencial.

2.2. Memórias

2.2.1. Banco de Registradores

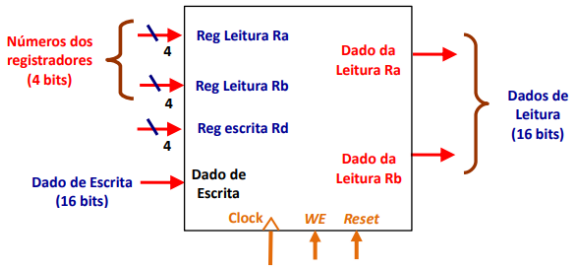

Figura 3. Esquema do Banco de Registradores, retirado do roteiro

Depois, construímos o banco de registradores, que consideramos o módulo mais complexo do processador por apresentar sete entradas diferentes - Ra e Rb (registradores de leitura, 4 bits cada), Rd (registrador de escrita, 4 bits), D (dado de escrita, 4 bits), Clock, WE (escolha entre escrita e leitura, 1 bit) e Reset. Utilizamos o tipo *reg* de Verilog para criar 16 registradores de 16 bits cada.

A presença do bit de escolha entre leitura e escrita se mostra de extrema importância na medida em que códigos de máquina mal pensados poderiam resultar na escrita indesejada de algum registrador e a leitura durante uma escrita é inconfiável.

Definimos que a leitura do banco de registradores ocorreria de forma assíncrona, a todo tempo desde que W $\neq$ 1 ou esteja fora da subida de clock.

O funcionamento da leitura se dá, portanto, a partir do uso de Ra e Rb para selecionar um dos registradores, cujo valor aprece na saída A ou B de 16 bits.

A escrita no banco de registradores ocorre de forma contrária, apenas quando W = 1 e haja uma subida de clock. Para tal, utilizamos um comando *always* condicionado pela borda de subida do clock. Como o Reset também é síncrono, também o incluímos dentro do comando *always*; Quando Reset = 1, todos os registradores assumem o mesmo valor do Registrador 0, que sempre tem valor zero, quando Reset = 0, ocorre a escrita. Para mantermos o Registrador 0 em zero apenas não o incluímos como possibilidade de escrita.

O funcionamento da escrita se dá utilizando o valor em Rd para selecionar um dos registradores, que recebe o valor presente na entrada D.

2.2.2. Registrador PC

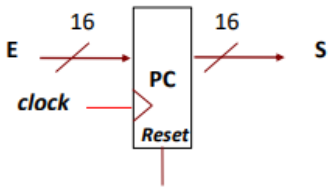

Figura 4. Esquema do registrador PC, retirado do roteiro

Criamos o módulo PC com um registrador de 16 bits. O módulo possui três entradas: E (16 bits), Reset e Clock. Utilizamos um comando *always* condicionado pela borda de subida do clock, onde quando Reset está ativado o registrador tem seu valor zerado, e quando Reset está desativado o registrador salva o valor em E; A saída S do PC sempre tem o valor do registrador, definido através de um comando *assign*.

2.2.3. Memória de Instruções

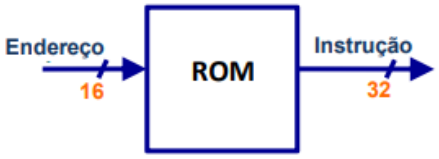

Figura 5. Esquema da Memória de Instruções (ROM), retirado do roteiro

Inicialmente, o módulo ROM foi criado utilizando um comando always com *casex* utilizando uma entrada Adds (o endereço, 16 bits) como seleção, ocorrendo em uma saída pré-determinada para Inst (32 bits). A estrutura geral é indicada abaixo:

```
module rom(
input [15:0] Adds,
output wire [31:0] Inst);
always @ (*)
casex (Adds)
16'h0000: Inst = 32'hXXXXXXXX
...
16'hXXXX: Inst = 32'hXXXXXXXX
endcase
endmodule
```

Onde X = Don't Care.

Entretanto, posteriormente passamos a utilizar o template de exemplo de ROM em Verilog disponibilizado, de forma a ler diretamente de arquivos; Para tal, ainda contamos com a entrada E de 16 bits e a saída Inst de 32 bits. Inicialmente, definimos uma ROM de 1024 posições, salvamos nela dados contidos em arquivos .dat utilizando o comando *initial $readmemh* e depois utilizamos um comando *assign* para atribuir a saída Inst ao dado apontado pela entrada Adds na ROM. É importante notar que a ROM se trata de outra estrutura assíncrona.

2.3. Bloco de Controle

Tabela 1. Campos das intruções de 32 bits

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rb				Ra				Rd				Opcode			

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Imediato															

O Bloco de Controle é mais um circuito combinacional, onde recebemos a entrada Inst (instrução, 32 bits) e através dela geramos diversas saídas necessárias para a realização de cada instrução específica. Cada parte da instrução foi separada conforme identificado no roteiro (Tabela 1) de forma a ser utilizada nas entradas de outros módulos; Para tal, utilizamos a seguinte fórmula na descrição de cada parte:

$$assign\ Parte = (Inst\,\%\,(Bi+2)'b10^{Bi+1}) >> Bf; \quad (1)$$

Onde Bi = posição do bit inicial da parte e Bf = posição do bit final da parte (Considerando posição inicial da instrução igual a 0).

Nessa fórmula, o módulo limita a quantidade de bits pela esquerda, enquanto o operador $>>$ (shift right) desloca os bits para a direita, eliminando bits indesejáveis pelo outro lado.

Além de criarmos saídas para cada parte definida pelo roteiro, criamos a entrada Op para a ULA, a partir dos 3 primeiros bits do Opcode. Criamos também identificadores de

saltos condicionais e incondicionais, e identificadores do tipo específico de salto (beq, bne, bge, blt ou jalr), tendo em vista a necessidade desses identificadores na determinação da próxima instrução de PC; Todos foram criados com base em condicionais de Verilog utilizando o Opcode, também criamos a entrada WE do Banco de Registradores pelo mesmo método.

2.4. Interligação dos componentes

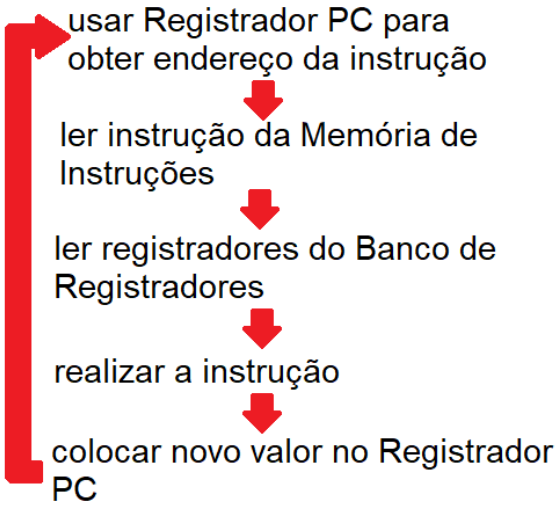

Figura 6. Etapas do funcionamento do processador

Nessa fase da implementação do Zeptoprocessador-V é imprescindível ter em mente o funcionamento geral do processador, descrito na figura acima - o utilizamos como guia na sua construção. Na interligação de todos os módulos, criamos um módulo *main*, que servirá de Top-Level do projeto. O módulo recebe como entrada KEY[0] (Reset, 1 bit), KEY[3] (Clock manual, 1 bit), SW[3:0] (SW[0] = seleção de clock, SW[2:1] = seleciona valor apresentado no display, SW[3] = seleção de clock automático), CLOCK_50 (Clock de 50MHz da placa FPGA, 1 bit), HEX0 a HEX7 (display de 7 segmentos, 7 bits) e LEDR[0].

```
module main(
input [3:0] KEY,
input [3:0] SW,
input CLOCK_50,
output wire [6:0] HEX0,
output wire [6:0] HEX1,
output wire [6:0] HEX2,
output wire [6:0] HEX3,
output wire [6:0] HEX4,
output wire [6:0] HEX5,
output wire [6:0] HEX6,
output wire [6:0] HEX7,
output wire [1:0] LEDR);
```

Primeiramente buscamos criar o clock do processador. Passamos o $Clock_50$ por um bloco fdiv.v, disponível nos materiais complementares, gerando a saída div com frequência de 1HZ. Definimos um fio clk para o clock do processador, que recebe o valor do clock manual caso SW[0] = 0; Caso SW[0] = 1, o valor é dependente de SW[3] - se SW[3] = 0, recebe o valor de CLOCK_50, se SW[3] = 1, recebe o valor de div. Isso permite que possamos rodar programas rapidamente, ou rodá-los de forma lenta, para que o processo possa ser visualizado. Definimos a saída LEDR[0] como o valor do clock, para que seja possível identificar sua frequência.

```
// Clock automático / manual
wire clk;
wire div;
fdiv U0(.clkin(CLOCK_50),.clkout(div));
always @(*)
begin
if (SW[0] == 1'b0)
begin
if (SW[3] == 1'b0)
begin
clk = CLOCK_50;
end
if (SW[3] == 1'b1)
begin
clk = div;
end
end
if (SW[0] == 1'b1)
begin
clk = KEY[3];
end
end
assign LEDR[0] = clk;
```

Atribuímos ao fio Reset o valor de KEY[0] negado, uma vez que na placa FPGA botões naturalmente emitem o valor 1 e passam ao valor 0 quando pressionados, e nossos componentes são resetados quando Reset = 1. Iniciando a etapa 1 da Figura 6, definimos a entrada E de PC com 16 bits como um fio (inicialmente com valor igual a 0). Utilizamos o módulo de PC dentro de main, utilizando E, clk e Reset como entradas, e conectamos diretamente ao módulo da Memória de Instruções, realizando a etapa 2 da Figura 6.

```
// PC e ROM
wire [15:0] E; // Entrada de PC
wire [15:0] Adds; // Endereço
wire [31:0] Inst; // Instrução
wire Reset;
assign Reset =  KEY[0];
pc U1(.E(E),.clk(clk),.Saida(Adds),.Reset(Reset));
rom U2(.Adds(Adds),.Inst(Inst));
```

Da Memória de Instruções, a instrução é levada ao Bloco de Controle, onde ela é desmembrada de forma a gerar os diversos sinais responsáveis por controlar o funcionamento do processador, mostrando-se necessário para o uso dos componentes seguintes.

```
wire [2:0] Op;
wire [3:0] Ra;
wire [3:0] Rb;
wire [3:0] Rd;
wire [15:0] Imm;
wire WE;
wire Jump;
wire CJump;
wire beq;
wire bne;
wire bge;
wire blt;
wire jalr;
controle U3(.Inst(Inst),.Op(Op),.Ra(Ra),.Rb(Rb),
.Rd(Rd),.Imm(Imm),.WE(WE),.Jump(Jump),.CJump(CJump),
.beq(beq),.bne(bne),.bge(bge),.blt(blt),.jalr(jalr));
```

O Banco de Registradores recebe entradas definidas pela Memória de Instruções (Ra, Rb, Rd, WE), Reset, clk e uma nova entrada D, que inicialmente possui valor 0 - seu valor, portanto, não é confiável nesse momento. Suas saídas A e B são direcionadas ao comparador e depois à ULA, onde ocorrem duas operações: uma com A e B, outra com o resultado (S) e o imediato da instrução, ocasionando na saída P.

```
wire [15:0] D;
wire [15:0] A;
wire [15:0] B;
registrador U5(.D(D),.Ra(Ra),.Rb(Rb),.Rd(Rd),.WE(WE),
.Reset(Reset),.clk(clk),.A(A),.B(B));
```

Após tais processos, é preciso realizar a próxima etapa, realizando a instrução e descobrindo o próximo valor de PC. Assim, definimos um fio com o valor que será somado a PC (Sadd, 16 bits), utilizamos um comando *always* e fazemos uso de condicionais a partir das saídas do Bloco de Controle.

Verificamos se ocorrerá um salto ou não; Caso não ocorra, a entrada D anteriormente definida para o Banco de Registradores assume o valor de P e Sadd obtém o valor 1. Caso ocorra um salto, identificamos se ele é condicional ou não - se for um salto condicional, aplicamos as condicionais de Verilog com a saída do comparador, Sadd adquirindo o valor do imediato se a saída do comparador for igual a 1 e Sadd adquirindo o valor 1 caso a saída seja igual a 0. Caso o salto não seja condicional, verificamos o valor de jalr, em ambos os casos definimos D como o endereço atual de PC+1, porém, se jalr = 1, então Sadd adquire o valor A - endereço atual de PC + imediato, de forma que, quando somado à PC, obtenha o valor de A + imediato; Se jalr = 0, Sadd adquire o valor do imediato.

```
wire [15:0] Sadd; // Soma ao endereço de PC
always @(*)
begin
if (Jump == 1'b0)
begin
D = P;
Sadd = 16'b1;
end
else
begin
if (CJump == 1'b1)
begin
if (beq == 1'b1)
begin
Sadd = SComp[3] ? Imm : 16'b1;
end
if (bne == 1'b1)
begin
Sadd = SComp[2] ? Imm : 16'b1;
end
if (bge == 1'b1)
begin
Sadd = SComp[1] ? Imm : 16'b1;
end
if (blt == 1'b1)
begin
Sadd = SComp[0] ? Imm : 16'b1;
end
end
```

```
else begin
if (jalr == 1'b1)
begin
D = Adds + 1;
Sadd = A - Adds + Imm;
end
else
begin
D = Adds + 1;
Sadd = Imm;
end
end
end
end
```

Por fim, utilizamos a ULA para somar o valor de Sadd ao endereço atual de PC, resultando em E (nova entrada do PC). Assim, cumprimos todas as etapas e o processador retorna à etapa 1 com o novo endereço de PC.

```
ula U9(.Op(3'b000),.A(Adds),.B(Sadd),.S(E)); // Coloca novo endereço de PC
```

Para poder visualizar os resultados, criamos fios de 7 bits cada com o valor do endereço atual de PC, a instrução, o registrador A e o registrador B, passados por um decodificador de hexadecimal para display de 7 segmentos, disponibilizado no material auxiliar. Depois, utilizamos um comando always que determina qual valor aparecerá nos displays de 7 segmentos através dos bits de escolha SW[2:1]. Se SW[2:1] = 00, o endereço de PC é selecionado, se SW[2:1] = 01, a instrução é selecionada, se SW[2:1] = 10, o registrador B é selecionado, e caso SW[2:1] = 11, então o registrador A é selecionado.

2.5. Algoritmos de Teste

Desenvolvemos programas que realizam multiplicação, divisão, resto e MDC de forma a testar o funcionamento do processador. Em cada um dos programas utilizamos a seguinte base disponibilizada:

Endereço	Código Hexadecimal	Instrução	Comentário
0x0000	0x0000 0010	addi R1,R0,R0,0	R1 = 0 Resultado
0x0001	0xXXXX 0020	addi R2,R0,R0,X	R2 = X
0x0002	0xYYYY 0030	addi R3,R0,R0,Y	R3 = Y
0x0003	0x0002 00FB	jal R15,Proc	R15=0x0004 PC=Proc
0x0004 Fim:	0x0000 110B	jal R0,0	J Fim - mostra R1 e R1
0x0005 Proc:	...	...	...
...	...	...	...
0xZZZZ	0x0000 0F0C	jalr R0,R15,0	Retorna resultado em R1

2.5.1. Multiplicação

Como a multiplicação deve ocorrer entre números de mesmo sinal, transformamos ambos os números da multiplicação em positivos e contamos a quantidade de números negativos, de forma que descubramos o sinal do valor final.

Em seguida, utilizamos um algoritmo básico de multiplicação, onde definimos um contador e somamos o valor de R3 à R1, incrementando cada vez que ocorre uma soma e parando tal soma quando o contador atinge o valor em R2. Tal algoritmo é expresso abaixo:

```
R4 = 0;
Enquanto R4 ≠ R2:
R1 = R1 + R3;
R4 = R4 + 1;
```

Onde R4 é o contador, R2 é X, R3 é Y e R1 é o resultado.

Adicionamos uma parte anterior no código que verifica se R2 é igual a 0, interrompendo automaticamente o código, uma vez que nossa comparação R4 $\neq$ R2 ocorre apenas após a soma R4 + 1 em código de máquina, de forma que o código não chegaria ao final caso contrário. Por fim, adequamos o sinal do resultado R1 de acordo com a quantidade de números negativos.

O código, em Assembly, apresentou-se da seguinte forma:

```
beq r2, r0, -1
bge r2, r0, 4
xori r2, r2, r0, -1
addi r2, r2, r0, 1
addi r5, r5, r0, 1
bge r3, r0, 4
xori r3, r3, r0, -1
addi r3, r3, r0, 1
addi r5, r5, r0, 1
addi r1, r3, r1, 0
addi r4, r4, r0, 1
blt r4, r2, -2
addi r5, r0, r0, 1
bne r5, r6, 3
xori r1, r1, r0, -1
addi r1, r1, r0, 1
```

2.5.2. Divisão

Inicialmente, tentamos utilizar a multiplicação para criar o algoritmo de divisão: utilizar um contador crescente para achar qual número multiplicado ao divisor chega mais

próximo do dividendo. Entretanto, nos deparamos com problemas de *overflow* ao utilizar o número -32768: nosso método de multiplicação faz uso de números positivos e não conseguimos representar o número +32768 em nosso processador (com 16 bits).

Modificamos então o algoritmo para um mais simples: dessa vez, transformamos todos os números em negativos e guardamos a quantidade de positivos; Então, utilizamos o algoritmo expresso abaixo:

```
Enquanto R3 ≥ R2:
R2 = R2 - R3;
R1 = R1 + 1;
```

Onde R1 é o resultado, R2 é X e R3 é Y. Observa-se que trocamos a posição de R2 e R3 do algoritmo convencional por estarmos utilizando números negativos.

Adicionamos também partes no código que identificam quando o divisor é -32768, resultando em 0 a menos que o numerador também seja -32768; Também uma que identifica quando o dividendo é 0, deslocando-se diretamente ao final do código, haja vista que a comparação (R3 ≥ R2) em código de máquina ocorre somente após a subtração. Por fim, caso o valor de R3 se torne positivo diminuímos o valor de R1 e atualizamos o resultado R1 para o sinal adequado com base no contador de números positivos.

```
addi r4, r0, r0, -32768
addi r5, r0, r0, 0
blt r2, r0, 4
xori r2, r2, r0, -1
addi r2, r2, r0, 1
addi r5, r5, r0, 1
blt r3, r0, 4
xori r3, r3, r0, -1
addi r3, r3, r0, 1
addi r5, r5, r0, 1
bne r3, r4, 5
beq r2, r4, 2
jalr r0, r15, 0
addi r1, r0, r0, 1
jalr r0, r15, 0
beq r2, r0, -1
subi r2, r2, r3, 0
addi r1, r1, r0, 1
bge r3, r2, -2
addi r6, r6, r0, 1
bne r5, r6, 3
xori r1, r1, r0, -1
addi r1, r1, r0, 1
```

2.5.3. Resto

O resto foi de simples resolução após o desenvolvimento do algoritmo de divisão, uma vez que o próprio algoritmo de divisão já apresenta o resto - apenas o reutilizamos com algumas diferenças: fazemos com que R1 assuma o valor de R2 no início do código e o sinal final é alterado para o mesmo de R2.

```
R1 = R2;
Enquanto R3 ≥ R1:
R1 = R1 - R3;
```

Onde R1 é o resultado, R2 é X e R3 é Y.

O código em Assembly que reflete esse algoritmo (a partir de Proc) está descrito abaixo.

```
addi r4, r0, r0, -32768
addi r1, r2, r0, 0
blt r1, r0, 3
xori r1, r1, r0, -1
addi r1, r1, r0, 1
blt r3, r0, 3
xori r3, r3, r0, -1
addi r3, r3, r0, 1
bne r3, r4, 5
addi r1, r2, r0, 0
bne r2, r3, 2
addi r1, r0, r0, 0
jalr r0, r15, 0
bge r3, r1, 3
addi r1, r2, r0, 0
jalr r0, r15, 0
subi r1, r1, r3, 0
bge r3, r1, -1
blt r1, r0, 2
addi r1, r0, r0, 0
blt r2, r0, 3
xori r1, r1, r0, -1
addi r1, r1, r0, 1
```

2.5.4. MDC

Para realizar o MDC utilizamos o Algoritmo de Euclides, segundo [Wikipedia 2021], ”o algoritmo de Euclides é um método simples e eficiente de encontrar o máximo divisor comum entre dois números inteiros diferentes de zero”. É um dos algoritmos mais antigos conhecidos e foi escolhido por sua simplicidade, pois só utiliza números positivos e não

requer nenhuma fatoração. Como esse algoritmo utiliza apenas números positivos, o sinal pôde ser desconsiderado na sua implementação, simplificando-o.

Esse algoritmo é descrito abaixo em pseudocódigo:

```
R1 = R2
Enquanto R3 ≠ 0: (
R4 = R1;
Enquanto R4 ≥ R3: (
R4 = R4 - R3; )
R1 = R3;
R3 = R4; )
```

Em Assembly, o algoritmo se apresentou da seguinte forma (dentro da seção Proc):

```
addi r1, r2, r0, 0
beq r0, r3, -2
addi r4, r1, r0, 0
blt r4, r3, r0
subi r4, r4, r3, 0
blt r3, r4, -1
addi r1, r3, r0, 0
addi r3, r4, r0, 0
bne r0, r3, -6
```

2.6. Temporização

Para que o circuito se encontre estável, mostra-se necessário adotar uma estratégia de temporização que delimita a frequência de clock máxima do circuito. Os tempos de propagação que devem ser obedecidos em cada parte do processador podem ser definidos assim, de forma geral:

Figura 7. Tempos entre circuitos sequenciais

Portanto, para que o funcionamento do processador se dê sem erros, a frequência de clock precisa respeitar o atraso da propagação da saída da primeira lógica sequencial dado o clock, o tempo de atraso de lógica combinacional e o setup time da segunda lógica sequencial. Assim, o clock deve conter uma frequência que abranja o maior atraso encontrado entre duas lógicas sequenciais.

3. Resultados Obtidos

Cada módulo resultou nos seguintes circuitos, vistos a partir da ferramente RTL Viewer do Quartus-II:

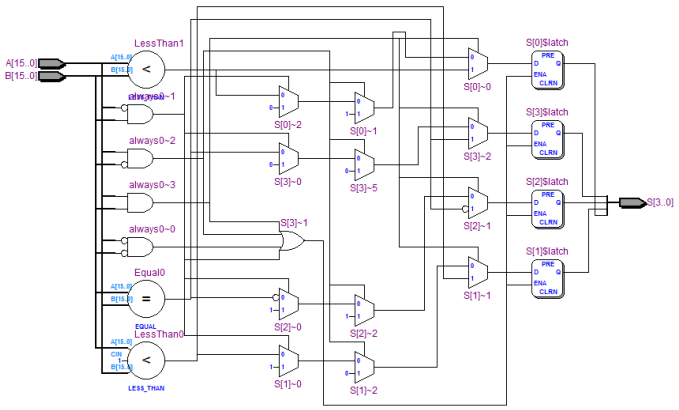

Figura 8. Circuito do Comparador no Quartus-II

Figura 9. Circuito da Ula no Quartus-II

Ainda que tenhamos tentado descrever ambos os circuitos do comparador e ULA utilizando lógica combinacional, o Quartus-II continuou a utilizar latches em sua composição pela natureza do comando *always*. Entretanto, a descrição do que se fazer em cada entrada ainda garante seu comportamento combinacional.

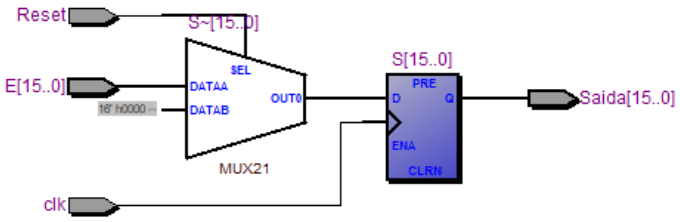

Figura 10. Circuito do Registrador PC no Quartus-II

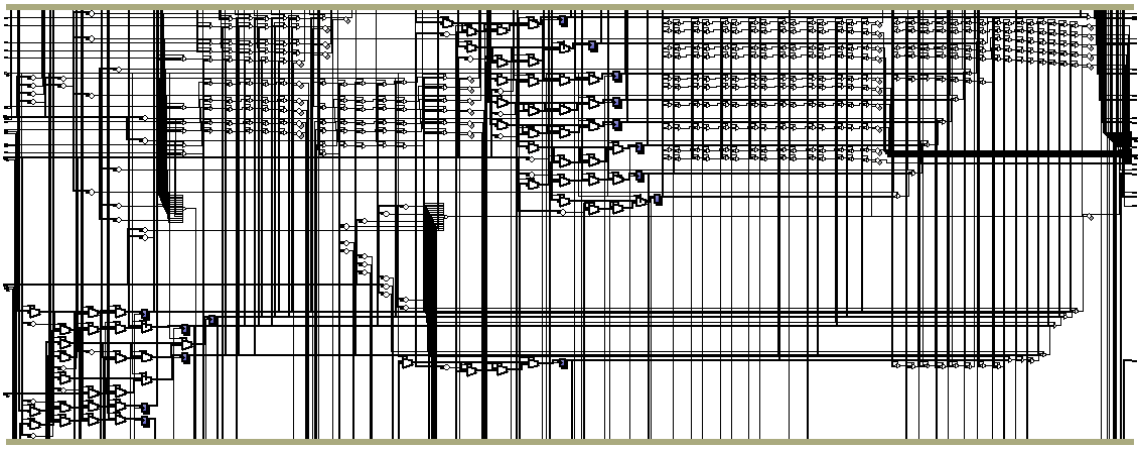

Figura 11. Circuito do Banco de Registradores no Quartus-II

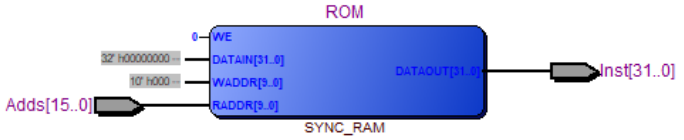

Figura 12. Circuito da Memória de Instruções no Quartus-II

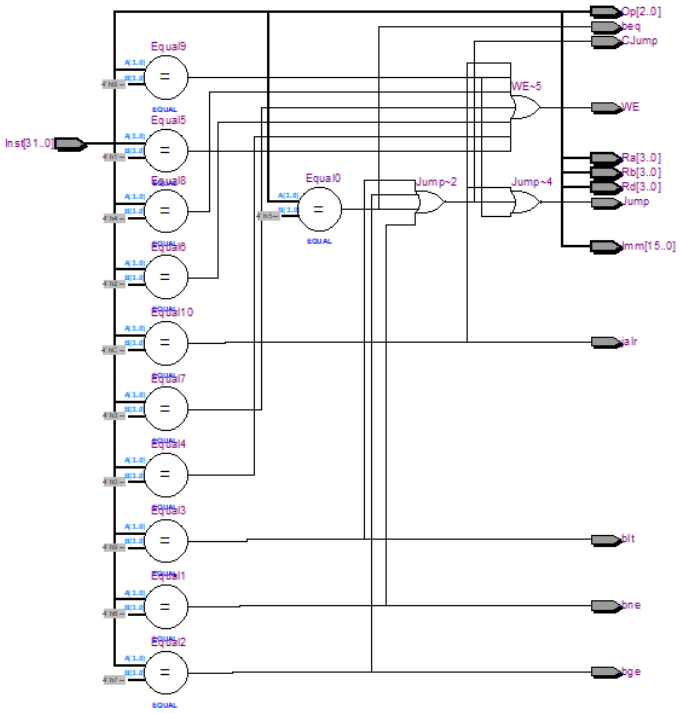

Figura 13. Circuito do Bloco de Controle no Quartus-II

É notável que, embora a ULA tenha tido uma descrição de Hardware tão simples, seu circuito (Figura 9) apresentou grande tamanho. Como esperado, o circuito do Banco de Registradores (Figura 11) foi o maior dentre os componentes do processador. O Bloco de Controle (Figura 13) e a ROM (Figura 12) foram os únicos circuitos que não utilizarem latches em sua composição.

Utilizando cada um desses módulos em sua composição, encontramos o seguinte circuito para o processador:

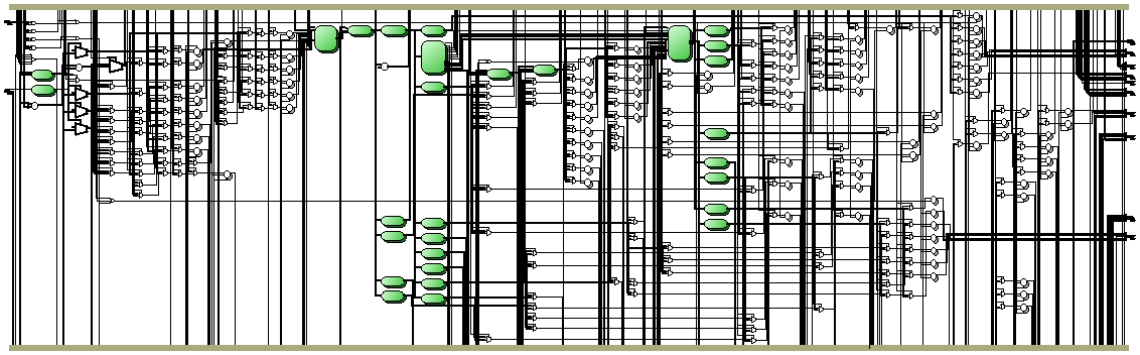

Figura 14. Circuito do Processador no Quartus-II

Para verificar o funcionamento dos registradores, fizemos uso de alguns programas de teste. De forma a observar o valor do resultado mais facilmente, criamos uma saída VerRA, que mostra o valor presente em A (saída do Banco de Registradores que apresenta o valor de Ra) a partir de um comando Assign.

3.1. Resultados da Temporização

Utilizando a metodologia demarcada na seção 2.6, descobrimos o tempo de clock mínimo para a execução confiável do circuito.

Os tempos necessários para a descoberta desse tempo de clock mínimo estão dispostos abaixo:

Circuito Sequencial	Tempo de Setup	Tempo de Hold
PC	0ns	0ns
Banco de Registradores	0ns	0ns
Divisor de Frequência	68.921ns	0ns

Circuito Combinacional	Tempo de Propagação
Comparador	6.619ns
ULA	13.770ns
Memória de Instruções	11.847ns
Bloco de Controle	9.501ns

Como o clock passando pelo Divisor de Frequência é pré-estabelecido, ele não será considerado nos cálculos. Portanto, cada caminho dos pares de circuitos sequenciais apresentam o seguinte atraso:

Caminho	Atraso
PC $\rightarrow$ BR	9.501ns
BR $\rightarrow$ PC	41.31ns

Onde BR = Banco de Registradores

Assim, o caminho mais longo leva 41.31ns. Portanto, podemos retirar disso que a frequência de clock mínima para a execução de qualquer possível programa é:

$$Fmin = \frac{1}{41.31ns} = \frac{1}{41.31x10^{-}9} \cong 24.2MHz \tag{2}$$

3.2. Resultados do contador e da soma de ímpares

Utilizamos o programa contador de -16 a 16 e a soma dos números ímpares de 0 a 15 já disponíveis no roteiro do projeto para verificar o funcionamento do processador. Nesta verificação, utilizamos apenas a forma de onda funcional do Quartus-II pois nosso foco ainda não tratava de questões temporais.

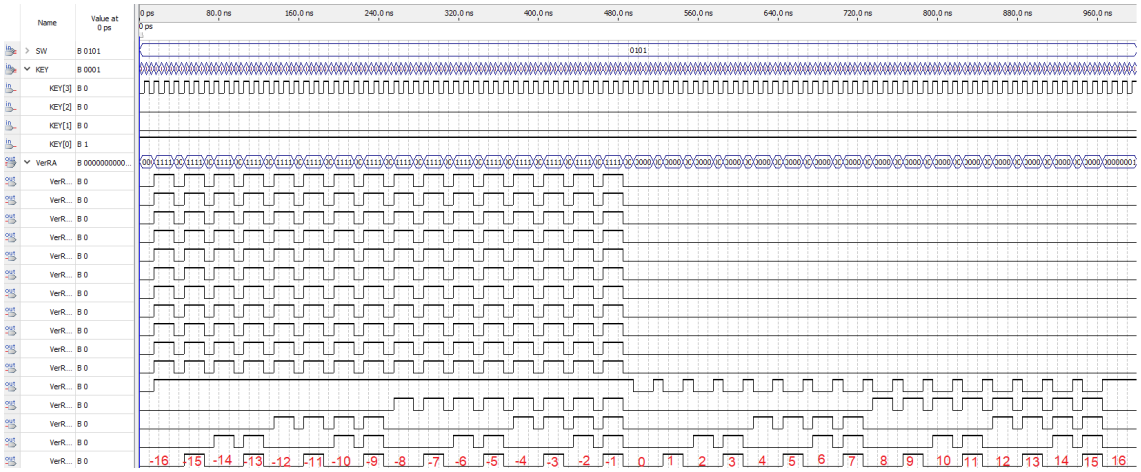

Figura 15. Forma de onda funcional do algoritmo de contagem aplicado no processador

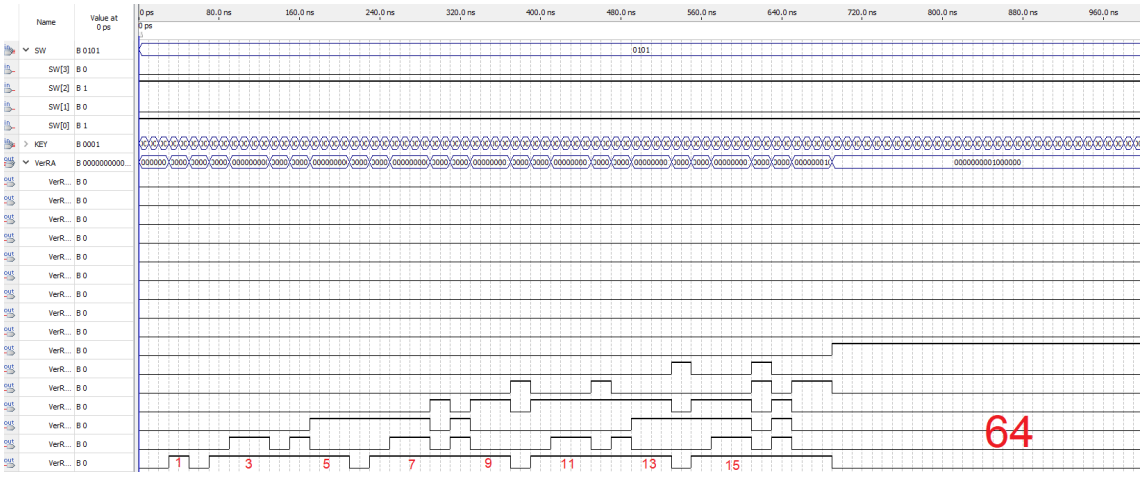

Figura 16. Forma de onda funcional do algoritmo da soma de ímpares aplicado no processador

Na Figura 15, a contagem (em decimal) está apresentada em vermelho abaixo dos bits da contagem em binário. Na Figura 16, os valores dos números ímpares de 0 a 15 e a soma final estão apresentados em vermelho abaixo de seus bits respectivos.

Ambas as formas de onda apresentaram o comportamento esperado com base no algoritmo: o contador (Figura 15) exibindo a contagem crescente de -16 até 16 em Ra nos momentos em que Ra apresenta o valor do registrador 1 e a soma dos números ímpares (Figura 16) expondo continuamente os números ímpares de 0 até 15 em Ra até que finaliza o programa mostrando o resultado da soma (64).

3.3. Resultados do algoritmo de multiplicação

Após notificarmos o funcionamento correto do processador em tais códigos específicos, buscamos realizar as formas de onda funcional e temporal do algoritmo de multiplicação (descrito na Seção 2.5.1) com uma conta simples, 1 x 0.

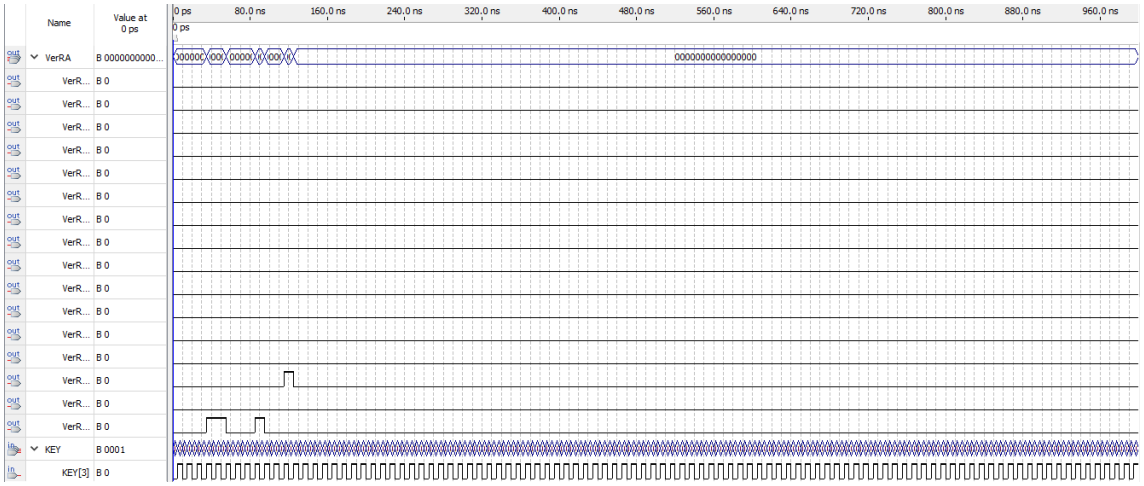

Figura 17. Forma de onda funcional do algoritmo de multiplicação no processador

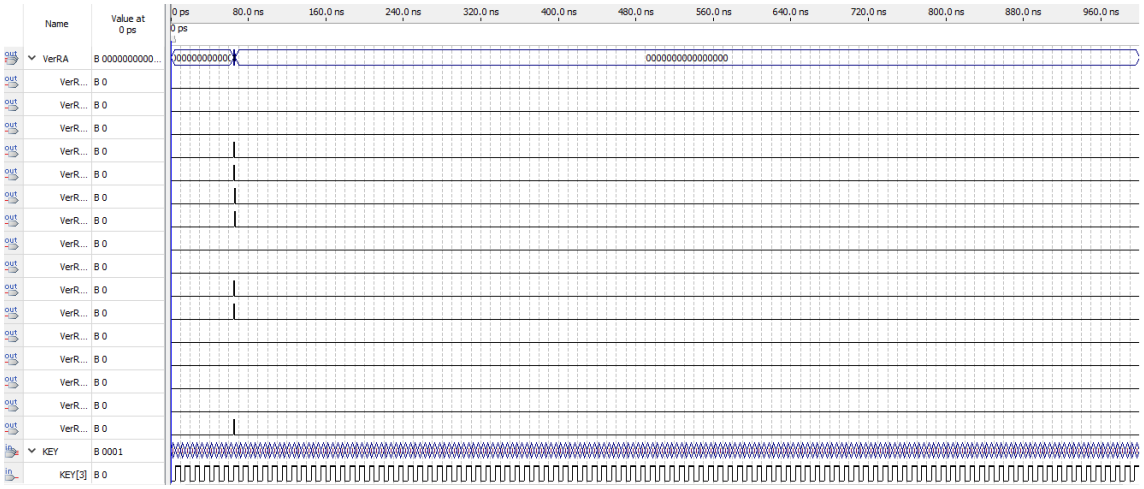

Figura 18. Forma de onda temporal do algoritmo de multiplicação no processador

Em ambas as formas de onda, utilizamos o clock manual e permanecemos com Reset = 1.

Identificamos que ambas as formas de onda funcional (Figura 17) e temporal apresentaram o resultado 0 ao final do circuito. Entretanto, o circuito temporal se apresentou mais condizente com o resultado esperado por alcançar o resultado após 7 subidas de clock (7 instruções). O funcionamento correto do circuito temporal se deve ao clock (de 20ns nessa forma de onda) estar dentro dos limites de frequência - 79.33MHz para esse circuito.

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	79.33 MHz	79.33 MHz	CLOCK_50
2	84.19 MHz	84.19 MHz	pc:U1|S[0]

Figura 19. Frequência máxima com Mult detectada pelo Quartus-II

Ainda assim, a forma de onda mostra pequenos erros, que denunciam que não tivemos sucesso na descrição de um circuito onde a entrada não se alterasse durante a subida de clock - isso leva a uma pequena falta de confiabilidade, entretanto, a olho nu tal erro do circuito não pode ser observado.

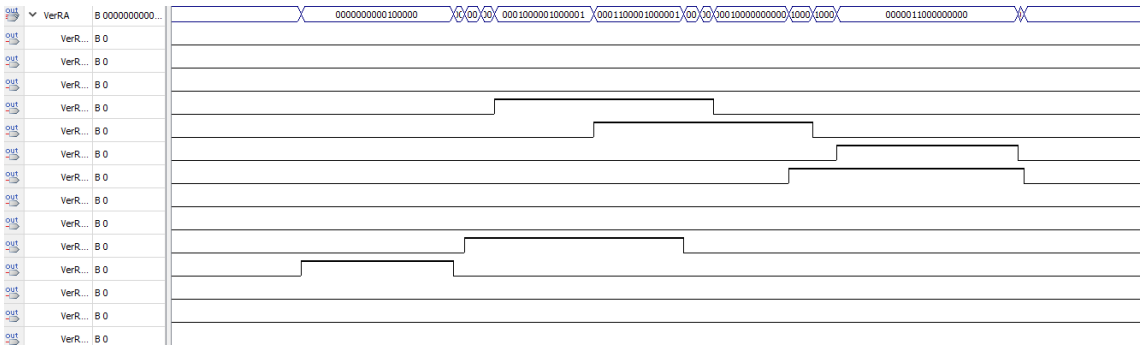

Figura 20. Erros na forma de onda temporal do algoritmo de multiplicação

Depois, sintetizamos o circuito na placa FPGA DE2, cuja gravação pode ser vista aqui. Realizamos as multiplicações -180 x -180, -180 x 180, 180 x 1 e 180 x 0, todas obtiveram o resultado esperado de sua multiplicação (32400, -32400, 180 e 0 respectivamente).

3.4. Resultados do algoritmo de divisão

Utilizando esse algoritmo, a frequência máxima do circuito é 72.75 MHz.

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	72.75 MHz	72.75 MHz	pc:U1|S[0]
2	77.04 MHz	77.04 MHz	CLOCK_50

Figura 21. Frequência máxima com Div detectada pelo Quartus-II

Sintetizamos esse circuito na placa FPGA DE2, sua gravação pode ser vista aqui. Realizamos as operações -32768/-32768, -32768/32767, -32768/1 e 0/-32768, obtendo os resultados 1, -1, -32768 e 0, funcionando corretamente.

3.5. Resultados do algoritmo de resto

Com o algoritmo do resto na Memória de Instruções, o circuito assume como frequência máxima 66.24 MHz.

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	66.24 MHz	66.24 MHz	pc:U1|S[0]
2	75.84 MHz	75.84 MHz	CLOCK_50

Figura 22. Frequência máxima com Rem detectada pelo Quartus-II

O circuito do processador com esse algoritmo foi gerado na placa FPGA DE2, a gravação de seu funcionamento está disponível aqui. Foram realizadas as operações 32767 % -32768, -32768 % -32768, -32768 % 32767, -32768 % 1 e 0 % -32768; Foram obtidos os resultados 32767, 1, -1, 0 e 0, que demarcam o funcionamento do circuito.

3.6. Resultados do algoritmo de MDC

O algoritmo de MDC no processador resulta em uma frequência máxima de 76.18MHz.

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name
1	76.18 MHz	76.18 MHz	CLOCK_50
2	90.2 MHz	90.2 MHz	pc:U1|S[0]

Figura 23. Frequência máxima com MDC detectada pelo Quartus-II

Por fim, sintetizamos o algoritmo de MDC na placa. Sua gravação pode ser vista aqui. Realizamos as operações MDC(32767,32767), MDC(32767,0), MDC(32767,1) e MDC(32767,2); Os resultados foram satisfatórios, uma vez que obtivemos como resposta 32767, 32767, 1 e 1.

4. Conclusão

Neste trabalho, implementamos o "ZeptoProcessador-V"em SystemVerilog com a assistência do software Quartus-II e um kit de desenvolvimento FPGA DE2. Todos os módulos especificados na seção 1.1 como necessários para a implementação do "ZeptoProcessador-V"foram abordados no texto e implementados.

O resultado final foi exatamente o que queríamos: um processador simples, de acordo com a especificação, com palavras de 16 bits, 11 instruções, 16 registradores, programável, e que passasse em todos os nossos testes.

Referências

[Wikipedia 2021] Wikipedia (2021). Algoritmo de euclides — wikipedia, a enciclopédia livre. `https://pt.wikipedia.org/wiki/Algoritmo_de_Euclides`. [Online; acessado em 2022-09-29].