Universidade Federal do Rio Grande do Sul
Instituto de Informática

Implementação em VHDL do Neander com Modificações

Alexandre Lima
Gert Folz
Willian Reichert

INF01175 - Sistemas Digitais
Prof. Fernanda Kastensmidt

Porto Alegre
Maio de 2019

1. Apresentação do problema

O objetivo deste trabalho é implementar o circuito, em VHDL, da máquina hipotética Neander, criada pelo professor Raul Weber, com algumas modificações em uma placa Digilent Basys 2, que contém a FPGA Spartan-3E:

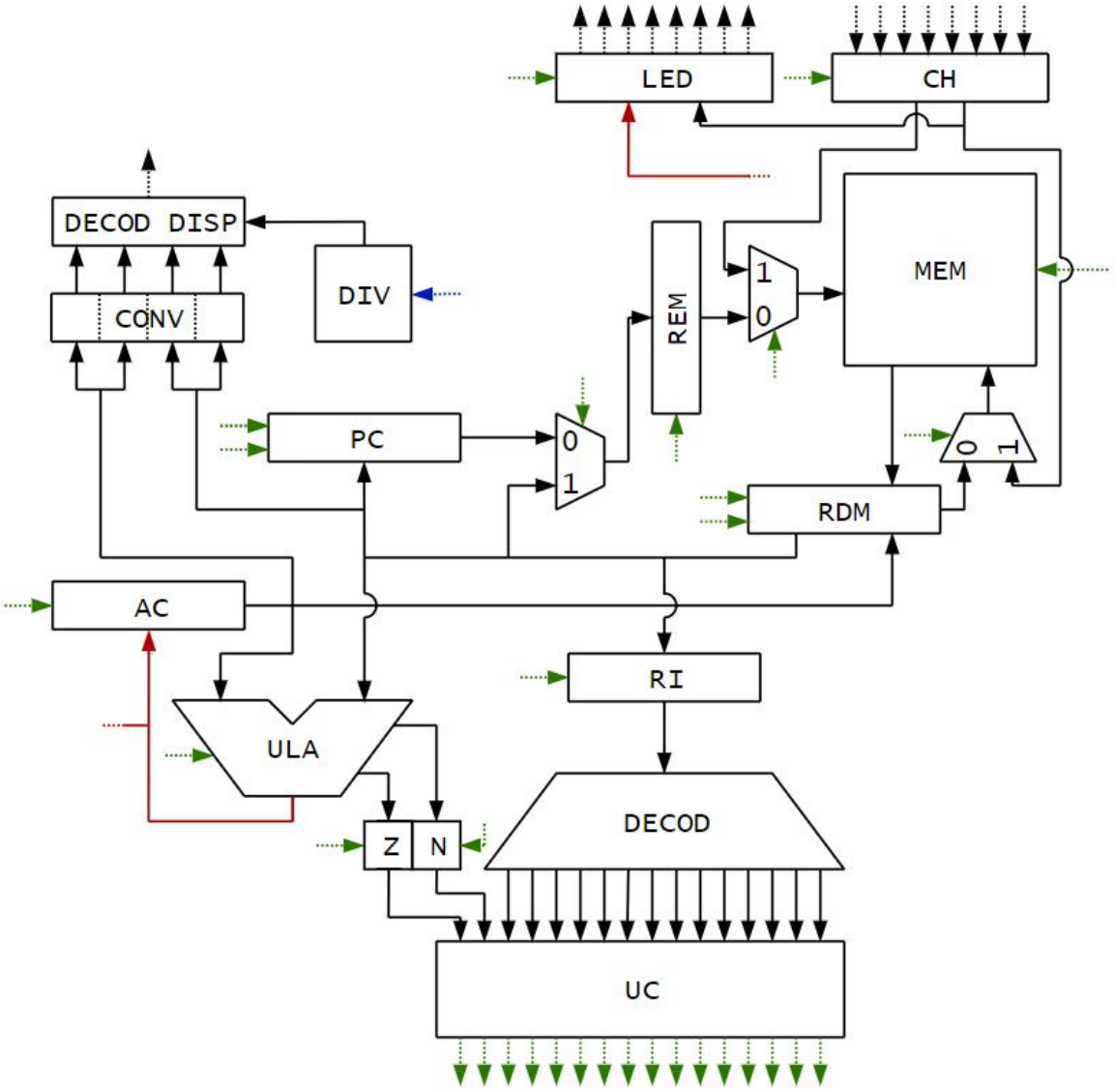


Imagem 1: diagrama do Neander modificado

Legenda do diagrama

- caminhos dos dados
- sinais de controle provenientes da UC
- saída da ULA
- clock de 50 MHz

2. Implementação

Para implementar em VHDL, o circuito do Neander foi separado em vários módulos para melhor organização e teste dos códigos:

- **Unidade de Controle (UC)**

É o maior bloco do Neander, sendo responsável por controlar todo o fluxo de dados da máquina através dos 16 sinais de controle. Foi implementado como uma *finite state machine* (FSM) que possui dois grandes estados: o de leitura dos dados iniciais e o de execução dos programas carregados na memória.

O estado de leitura é o responsável por controlar o preenchimento de 8 endereços da REM com valores iniciais através das chaves e do botão BTN0 antes da execução do programa ser iniciada. O sinal *reset1*, enviado pelo botão BTN3, reinicia o Neander por aqui.

Já o estado de execução controla todos os outros componentes da máquina de forma que as instruções armazenadas sejam executadas até encontrar o *HLT*. Cada aperto no botão BTN0 executa uma única instrução, o que foi feito para permitir a visualização dos valores do AC e PC a cada passo. O sinal *reset0*, enviado pelo botão BTN2, reinicia o Neander por aqui e utiliza os mesmos dados inseridos anteriormente na execução.

O agrupamento de certas instruções em caminhos definidos da FSM levou em consideração as operações semelhantes realizadas entre elas. Todas as instruções, sem exceção, passam pelo bloco de estados gerais **e0**, **e1**, **e2**, **e3** e **ef**.

Os estados **erem**, **write** e **eri** foram colocados para permitir que certas operações fossem realizadas sem problemas, sendo elas o armazenamento do endereço no REM, a escrita dos dados na memória durante a instrução STA e a passagem dos códigos das instruções para o RI antes que a máquina entrasse em um caminho definido (que dependem desses códigos).

- **Program Counter (PC)**

Apontador de programa em português, é um *process* sensível ao *clock* e a um sinal de *reset* assíncrono. Como o nome sugere, ele é utilizado para percorrer a memória durante a execução do programa, servindo como um apontador para os endereços.

Geralmente o fluxo de execução é linear, entretanto as instruções que realizam *jumps* podem alterar o valor desse registrador para executar blocos condicionais ou laços, por exemplo.

- **Acumulador (AC)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. É responsável por armazenar os valores resultantes das operações realizadas na unidade lógica

e aritmética ao receber o sinal *ac_carga* da unidade de controle. O dado salvo é mostrado nos displays e pode ser armazenado na RAM através do registrador de dados da memória.

- **Unidade Lógica e Aritmética (ULA)**

Bloco puramente combinacional que realiza operações lógicas e aritméticas, além de detectar quando o resultado é zero ou negativo para enviar sinais aos registradores das duas *flags* N e Z. O sinal *ula_seletor*, enviado pela UC, seleciona a operação a ser realizada, sendo as opções:

	b2	***operação***
ADD	000	AC ← AC + RDM
AND	001	AC ← AC ***and*** RDM
OR	010	AC ← AC ***or*** RDM
NOT	011	AC ← ***not*** AC
LDA	100	AC ← RDM
SUB	101	AC ← AC – RDM
DEC	110	AC ← AC – 1
SHL	111	AC ← AC + AC

Tabela 1: operações realizadas na ULA do Neander modificado

Os dois operandos da ULA são o valor atual do acumulador (AC) e o dado lido da memória armazenado no registrador de dados (RDM). Algumas operações utilizam somente um operando, ao passo que o restante utiliza dois.

As condições e os respectivos valores atibuídos às *flags* são:

	condição	***atribuição***
N	AC < 0	N ← 1
	AC ≥ 0	N ← 0
Z	AC = 0	Z ← 1
	AC ≠ 0	Z ← 0

Tabela 2: condições e atribuições das *flags* de controle

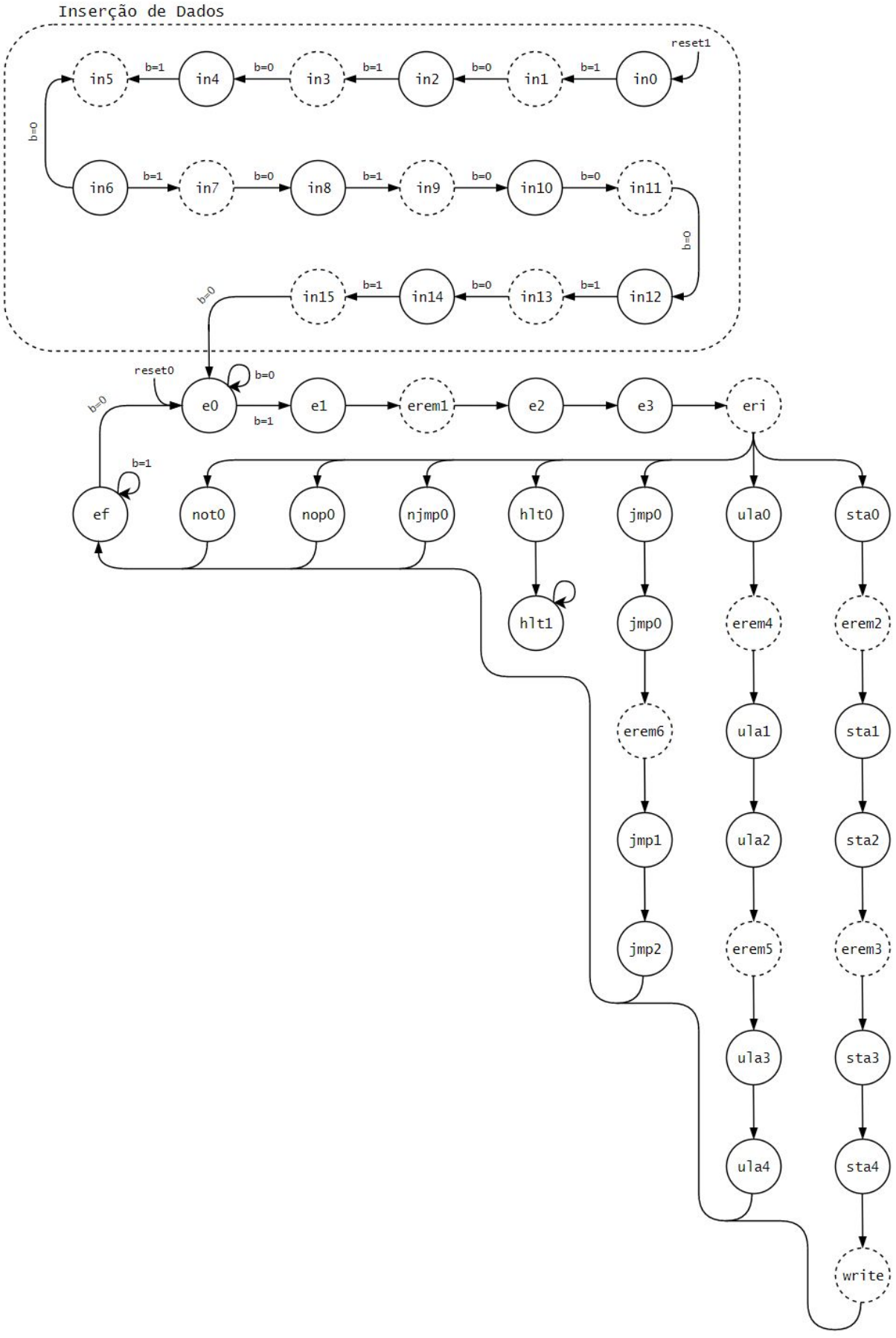


Imagem 2: diagrama da FSM da unidade de controle

- **Decodificador de Instruções (DECOD)**

Process puramente combinacional responsável por propagar sinais para a unidade de controle que indicam qual das 14 instruções será realizada. Cada uma das instruções possui um bit que é ativado dependendo do código recebido do registrador de instruções. São elas:

	b2	***b10***	***operação***
NOP	0000	000	*nenhuma operação*
STA	0001	016	MEM*[end]* ← AC
LDA	0010	032	AC ← MEM*[end]*
ADD	0011	048	AC ← AC **+** MEM*[end]*
OR	0100	064	AC ← AC ***or*** MEM*[end]*
AND	0101	080	AC ← AC ***and*** MEM*[end]*
NOT	0110	096	AC ← ***not*** AC
JMP	1000	128	PC ← *end*
JN	1001	144	PC ← *end* ***if*** N = 1
JZ	1010	160	PC ← *end* ***if*** Z = 1
SUB	1011	176	AC ← AC **-** MEM*[end]*
DEC	1100	192	AC ← AC **-** 1
SHL	1101	208	AC ← AC **+** AC
HLT	1111	240	*parar processamento*

Tabela 3: instruções do Neander modificado

- **Registrador de Instruções (RI)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. O RI armazena o código da instrução a ser executada lido da memória, e o propaga para o DECOD quando a UC envia o sinal *ri_carga*.

Apesar de somente o *nibble* mais significativo do vetor ser utilizado (as instruções necessitam de 4 bits, mas os dados na memória possuem uma largura de 8 bits), todo o *byte* é enviado para o decodificador de instruções, que então é analisado.

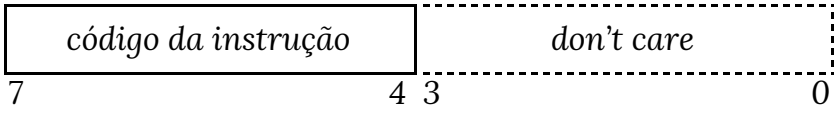


Imagem 3: esquema de um vetor de instruções

- **Registrador da Flag Negativo (N)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. É o registrador responsável por armazenar o *bit* da *flag* que indica se o resultado da operação realizada na ULA foi negativo ao receber o sinal *reg_n_carga* da UC.

- **Registrador da Flag Zero (Z)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. Possui a mesma função do registrador anterior, mas armazena a *flag* que indica se o resultado foi nulo ao receber o sinal *reg_z_carga* da unidade de controle.

- **Multiplexadores**

Processes puramente combinacionais. Foram utilizadas três instâncias de um multiplexador, cuja função é selecionar uma entrada utilizando um sinal enviado pela UC e propagá-la para a saída.

O primeiro mux é responsável por selecionar o dado que será enviado para o REM como endereço de leitura/escrita, podendo ser o valor do PC ou um valor lido da memória, dependendo da instrução sendo executada.

O segundo mux fica entre o REM e a memória, e define se o endereço passado para ela será o armazenado no registrador ou o enviado pelo controlador das chaves. A seleção depende de qual estado a UC está executando no momento (leitura e escrita de dados iniciais ou execução do programa).

O terceiro e último mux fica entre o RDM e a memória, e tem um propósito semelhante ao segundo. A diferença é que o seletor escolhe qual dado será enviado para escrita na memória, o que também depende dos dois grandes estados da unidade de controle.

- **Memória (MEM)**

O Neander possui uma única memória RAM, compartilhada entre instruções e dados, que contém 256 endereços para salvar dados de 8 bits, além de 2 registradores separados (que foram implementados como *processes* sensíveis ao clock e a sinais de reset assíncronos): um para o endereçamento do dado na memória (REM) e outro para escrita e leitura de dados (RDM).

Em instruções que não necessitam de um segundo operando, os dados são dispostos da seguinte maneira:

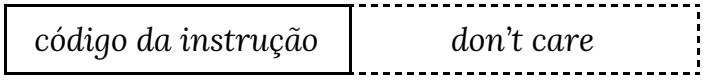


Imagem 4: um vetor de uma instrução no endereço 0

Já em instruções que necessitam de um segundo operando (endereço), os dados são dispostos da seguinte maneira:

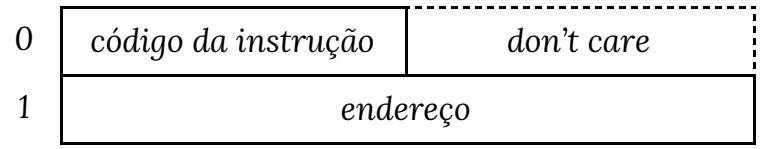


Imagem 5: dois vetores de uma instrução nos endereços 0 e 1

No projeto, utilizamos a ferramenta Core Generator, contida no ISE da Xilinx, para uso da BRAM com o modo *write first*. Foi implementada uma memória *RAM Single Port*, com uma entrada para a escrita (*dina*) e outra para a leitura de dados (*douta*). Os sinais para a leitura/escrita (*wea*) e carga dos registradores partem da unidade de controle.

As instruções do programa a ser executado se localizam na memória, que é carregada com elas pelo arquivo .coe, usado para inicializar dados na BRAM. Um exemplo de programa é uma soma de duas matrizes 2x2:

```
MEMORY_INITIALIZATION_RADIX=10;
MEMORY_INITIALIZATION_VECTOR=0,32,244,48,248,16,252,32,245,48,249,16,2
53,32,246,48,250,16,254,32,247,48,251,16,255,240;
```

Imagem 6: exemplo de um arquivo .coe

- **Registrador de Endereços da Memória (REM)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. Armazena o endereço que será utilizado para ler/escrever ao receber o sinal *reg_end_carga* da UC. O endereço poderá ser o valor do PC, ou um dado lido da memória no caso das instruções que necessitam de um valor.

- **Registrador de Dados da Memória (RDM)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono. É um registrador bidirecional: fornece o dado a partir da memória ao receber o sinal rdm_read, e prepara o valor do AC para a escrita quando recebe o sinal *rdm_carga* da unidade de controle.

- **Divisor de Frequência (DIV)**

Process sensível ao *clock* e a um sinal de *reset* assíncrono, O divisor foi implementado como um contador de 0 a 100.000. Assim como no Montador

Daedalus, o registrador AC e o PC são mostrados nos displays da placa. Para isso, foram utilizados os códigos criados no primeiro trabalho para acionamento dos mesmos.

Como os 4 displays possuem os pinos dos segmentos em comum, faz-se necessário o uso de um divisor para chaveamento dos displays. A frequência de relógio original da 50 MHz da placa é dividida por $1 * 10^5$, o que resulta em 500 Hz e torna possível a visualização "contínua" dos números.

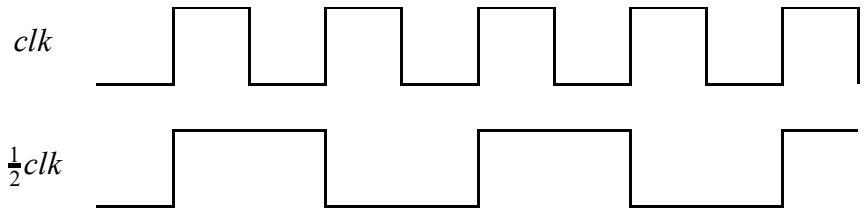


Imagem 7: exemplo de um divisor de frequência

- **Conversores BCD (CONV)**

Processes puramente combinacionais. Foram utilizados quatro conversores, sendo que cada um recebe 4 bits e os converte para o equivalente hexadecimal para serem exibidos. Os valores em hexadecimal são armazenados em 8 bits, sendo sete deles para os segmentos e um para o ponto (DP).

CA	CB	CC	CD	CE	CF	CG	DP

Imagem 8: vetor dos segmentos e do ponto dos displays

É importante notar que os segmentos e o ponto acendem com 0 e apagam com 1, ou seja, o vetor "01100011" corresponde à letra C. Também há somente um vetor que é compartilhado para todos os displays, um de cada vez, e por isso faz-se necessário o uso de um decodificador.

- **Decodificador dos Displays (DECOD DISP)**

Process sensível ao *clock* de 500 Hz gerado pelo divisor de frequência e a um sinal de *reset* assíncrono. O decodificador alterna a exibição dos displays utilizando um vetor para mostrar os dígitos do AC, que ficam nos dois displays mais significativos, e do PC, que ficam nos displays menos significativos. Cada display é iluminado por 0,002 s, ou 2 ms ($\frac{1}{500\ Hz}$).

É importante notar que o display selecionado recebe o valor 0, ou seja, o vetor "1101" corresponde ao segundo display sendo iluminado.

AN0	AN1	AN2	AN3

Imagem 9: vetor dos ânodos dos displays

- **Controlador das Chaves (CH)**

Process puramente combinacional controlado pelo sinal *ch_contador*. É utilizado na leitura e escrita dos dados iniciais utilizados pelo programa a ser executado. Cada chave ocupa 1 bit em um vetor de 8 bits no formato "SW7 SW6 SW5 SW4 SW3 SW2 SW1 SW0", que é enviado para a memória como dado a ser escrito.

SW7	SW6	SW5	SW4	SW3	SW2	SW1	SW0

Imagem 10: vetor das chaves (*switches*)

O usuário escolhe o número manipulando as chaves (chave pra cima é 1 e para baixo é 0), e aperta o botão BTN0 para armazenar o dado na RAM. A unidade de controle se encarrega de incrementar o endereço de escrita a cada armazenamento, sendo [244, 251] o intervalo de memória utilizado.

- **Controlador dos LEDs (LED)**

Process puramente combinacional. Não foi criado como um módulo separado, apesar de se comportar como um.

Ele seleciona qual vetor será mostrado nos 8 leds na placa, sendo o vetor das chaves durante o estado de leitura e escrita de dados iniciais, e o vetor de saída da ULA (que foi o último dado utilizado nos testes que executamos) durante a execução do programa no Neander.

LD7	*LD6*	*LD5*	*LD4*	*LD3*	*LD2*	*LD1*	*LD0*

Imagem 11: vetor dos LEDs

3. Resultados

- **Dados de área**

Os dados de área foram retirados diretamente do *Design Summary* e do *console* do ISE, e se encontram na próxima página.

- **Dados de frequência e período**

Foram retirados diretamente do *console* do ISE durante a síntese.

```
Minimum period: 13.210ns (Maximum Frequency: 75.700MHz)
Minimum input arrival time before clock: 4.553ns
Maximum output required time after clock: 16.517ns
Maximum combinational path delay: 6.140ns
```

Imagem 12: dados de frequência e período de relógio

Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	184	1,920	9%
Number used as Flip Flops	139		
Number used as Latches	45		
Number of 4 input LUTs	333	1,920	17%
Number of occupied Slices	209	960	21%
Number of Slices containing only related logic	209	209	100%
Number of Slices containing unrelated logic	0	209	0%
Total Number of 4 input LUTs	366	1,920	19%
Number used as logic	333		
Number used as a route-thru	33		
Number of bonded IOBs	32	83	38%
IOB Flip Flops	1		
Number of RAMB16s	1	4	25%
Number of BUFGMUXs	2	24	8%
Average Fanout of Non-Clock Nets	3.12		

Imagem 13: dados de área da implementação do Neander

```
Advanced HDL Synthesis Report

Macro Statistics
# FSMs                                  : 1
# ROMs                                  : 5
 16x8-bit ROM                          : 4
 8x8-bit ROM                           : 1
# Adders/Subtractors                    : 2
 32-bit adder                          : 1
 8-bit addsub                          : 1
# Counters                              : 1
 8-bit up counter                      : 1
# Registers                             : 87
 Flip-Flops                            : 87
# Latches                               : 1
 47-bit latch                          : 1
# Comparators                           : 2
 32-bit comparator less                : 2
# Multiplexers                          : 1
 8-bit 8-to-1 multiplexer              : 1
```

Imagem 14: mais dados de área da implementação da máquina

- **Simulações**

As simulações foram feitas com e sem atraso, com o *clock* comum no decodificador dos displays para podermos visualizar os resultados. Os sinais observados para analisar esses resultados foram os enviados para os displays da

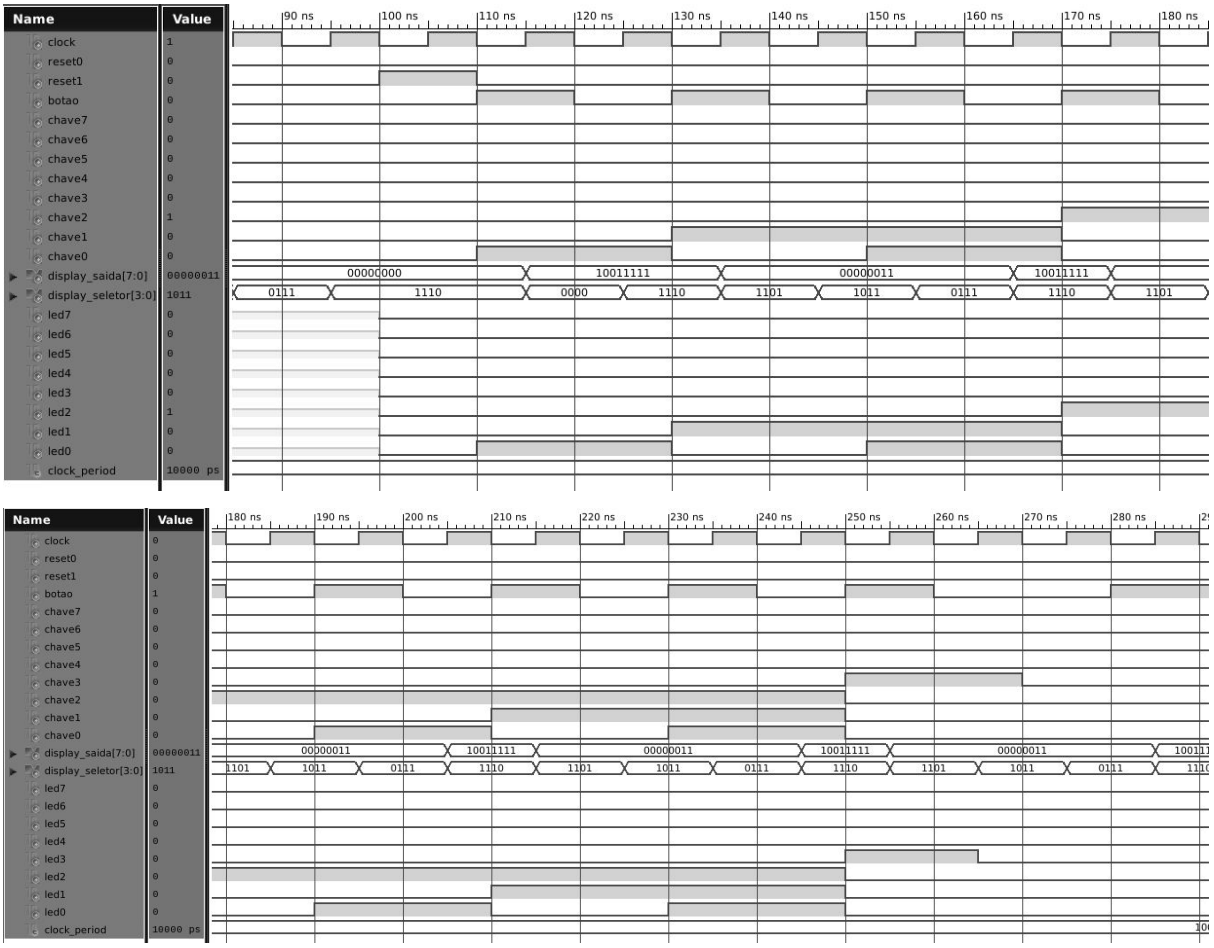
placa, juntamente com o vetor de seleção dos displays. Para isso, precisamos saber o que os bytes enviados para o DECOD DISP representam:

representação hexadecimal	***byte***
0	00000011
1	10011111
2	00100101
3	00001101
4	10011001
5	01001001
6	01000001
7	00011111
8	00000001
9	00001001
A	00010001
B	11000001
C	0110001
D	10000101
E	01100001
F	01110001

Tabela 4: tabela de conversão BCD

O programa utilizado para as simulações foi o de soma de duas matrizes 2x2, e como os *testbenches* são longos, somente incluímos as *screenshots* que contém a inserção inicial dos dados pelas chaves, e as que contém os valores do AC e do PC após as realizações das instruções ADD e STA (são oito no total). Essas instruções foram marcadas na **Imagem 6**.

As *screenshots* a seguir são dos testes sem atraso:



Imagens 15 e 16: podemos ver a inserção dos valores de 1 a 8 nas oito posições de memória utilizadas para dados iniciais. Os LEDs possuem a mesma configuração de onda das chaves, mostrando que as chaves para cima acendem seus respectivos LEDs no modo leitura

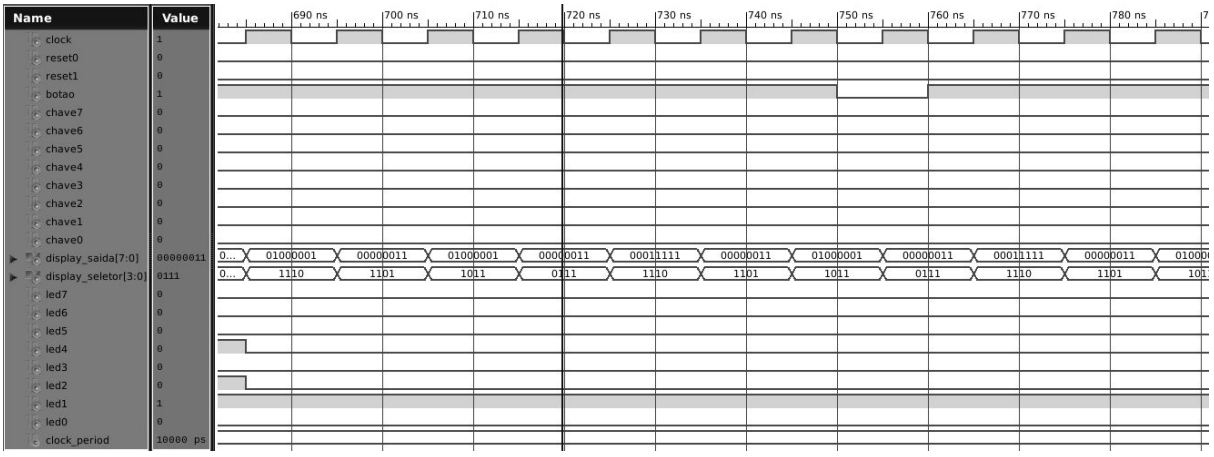


Imagem 17: no intervalo [730 ns, 760 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **07** e nos do AC temos **06**, o que mostra que a soma 1 + 5 dos elementos

1x1 foi feita corretamente (a primeira matriz possui os elementos 1, 2, 3 e 4, e a segunda os elementos 5, 6, 7 e 8), e que uma instrução NOP e outras três que utilizam um operando foram realizadas com sucesso

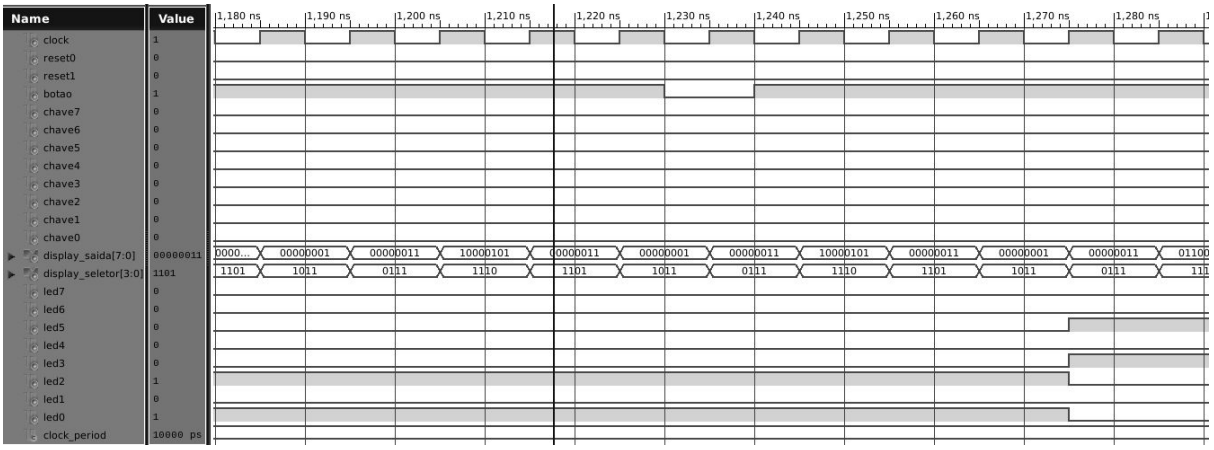


Imagem 18: no intervalo [1.210 ns, 1.240 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **0D** e nos do AC temos **08**, o que mostra que a soma 2 + 6 dos elementos 1x2 foi feita corretamente, e que as três instruções foram realizadas com sucesso

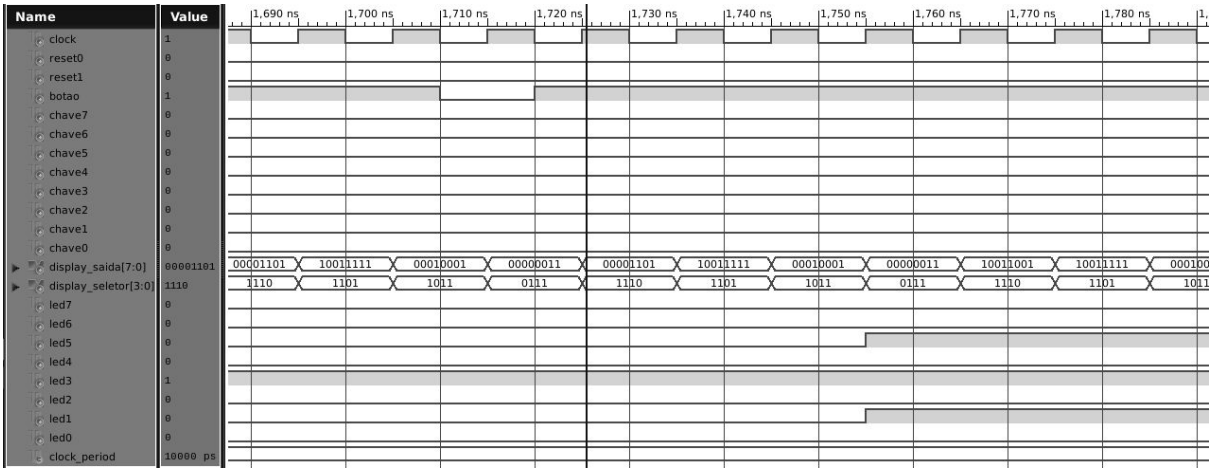


Imagem 19: no intervalo [1.730 ns, 1.760 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **13** e nos do AC temos **0A**, o que mostra que a soma 3 + 7 dos elementos 2x1 foi feita corretamente, e que as três instruções foram realizadas com sucesso

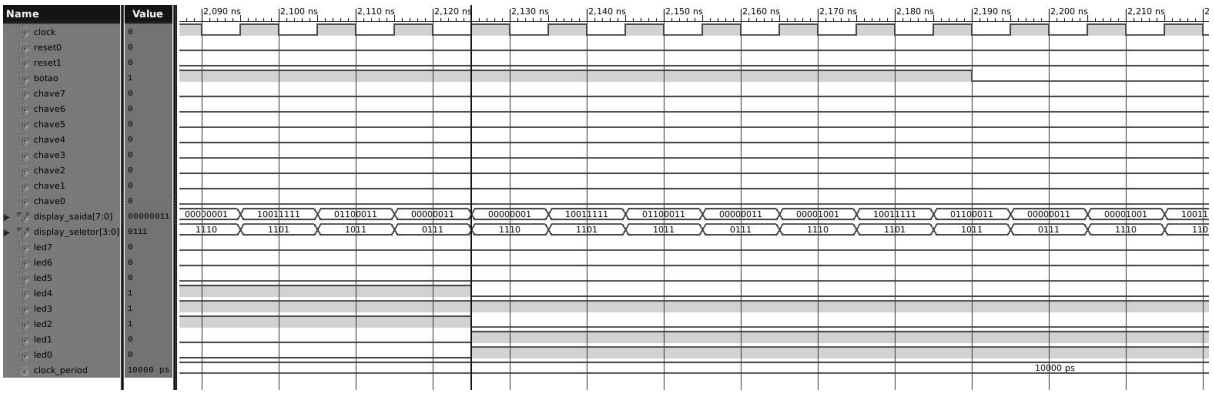
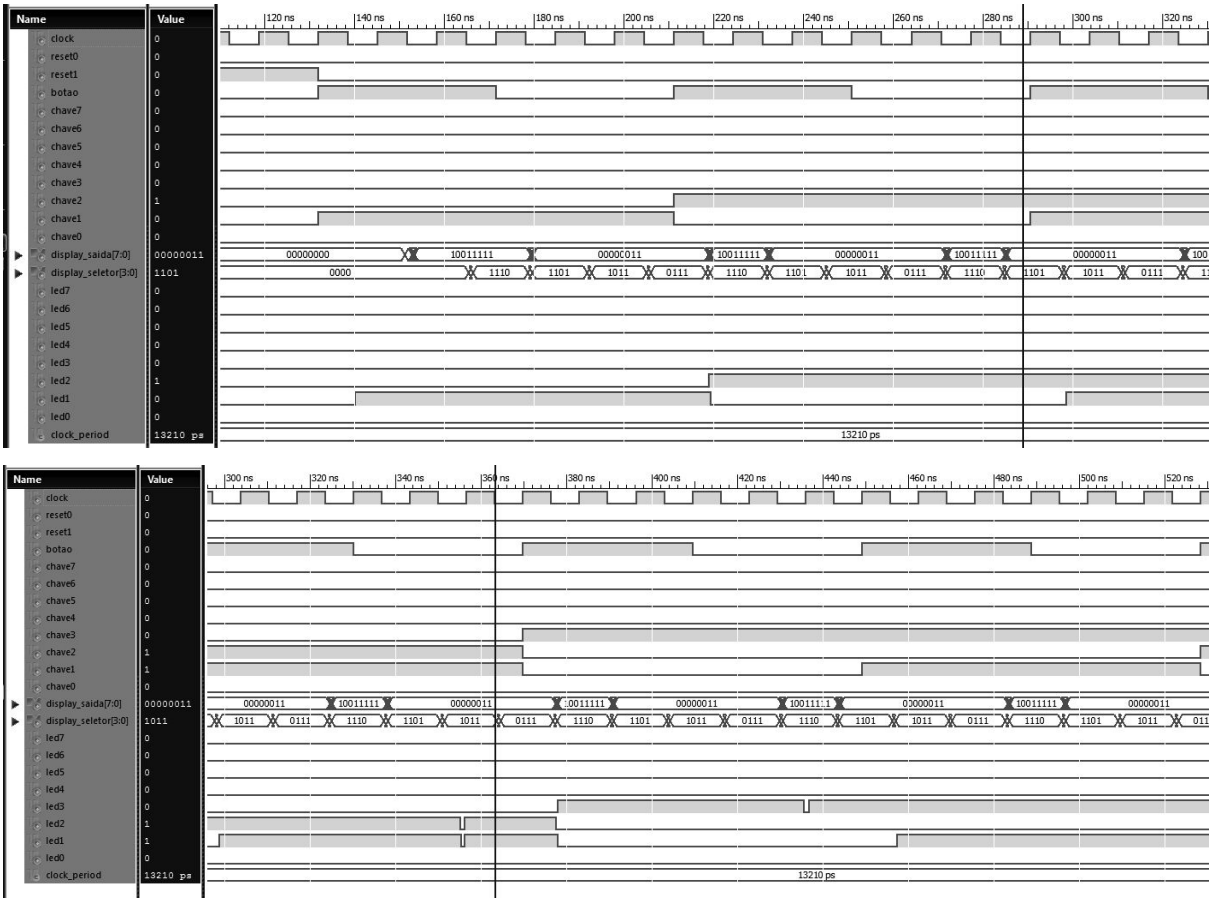
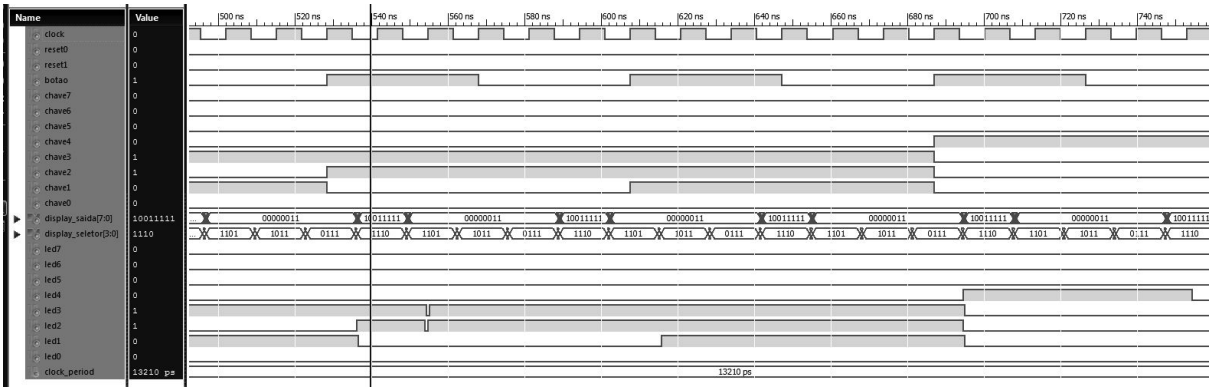


Imagem 20: no intervalo [2.170 ns, 2.200 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **19** e nos do AC temos **0C**, o que mostra que a soma 4 + 8 dos elementos 2x2 foi feita corretamente, e que as três instruções foram realizadas com sucesso

Para a simulação com atraso, as matrizes foram inicializadas com os valores 2, 4, 6, 8, 10, 12, 14 e 16, seguindo a mesma lógica da simulação anterior:





Imagens 21, 22 e 23: podemos ver a inserção dos valores 2, 4, 6, 8, A, C, E e 10 nas oito posições de memória utilizadas para dados iniciais. Os LEDs possuem a mesma configuração de onda das chaves com um pequeno atraso de aproximadamente 8 ns, mostrando que as chaves para cima acendem seus respectivos LEDs no modo leitura

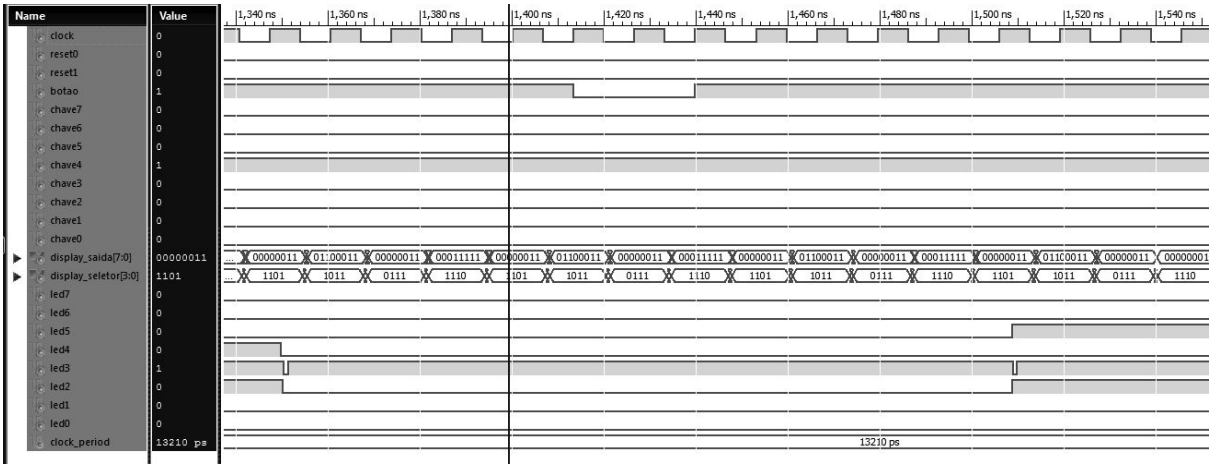


Imagem 24: no intervalo [1.440 ns, 1.480 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD, STA e uma NOP. Nos displays do PC temos **07** e nos do AC temos **0C**, o que mostra que a soma 2 + A dos elementos 1x1 foi feita corretamente, e que as três instruções foram realizadas com sucesso. A saída nos displays quase não possui atraso em relação às seleções do decodificador, sendo de aproximadamente 0,2 ns

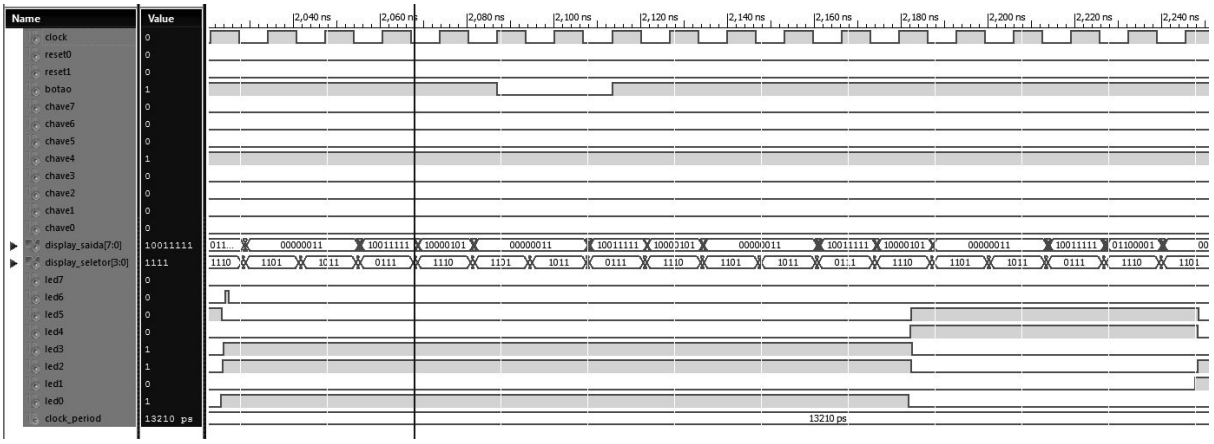


Imagem 25: no intervalo [2.120 ns, 2.160 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **0D** e nos do AC temos **10**, o que mostra que a soma 4 + C dos elementos 1x2 foi feita corretamente, e que as três instruções foram realizadas com sucesso. Na mudança do display 1101 para o 1011 não há troca do vetor de 8 bits, o que indica que os displays estão mostrando o mesmo valor (nesse caso, o AC e o PC possuem um zero)

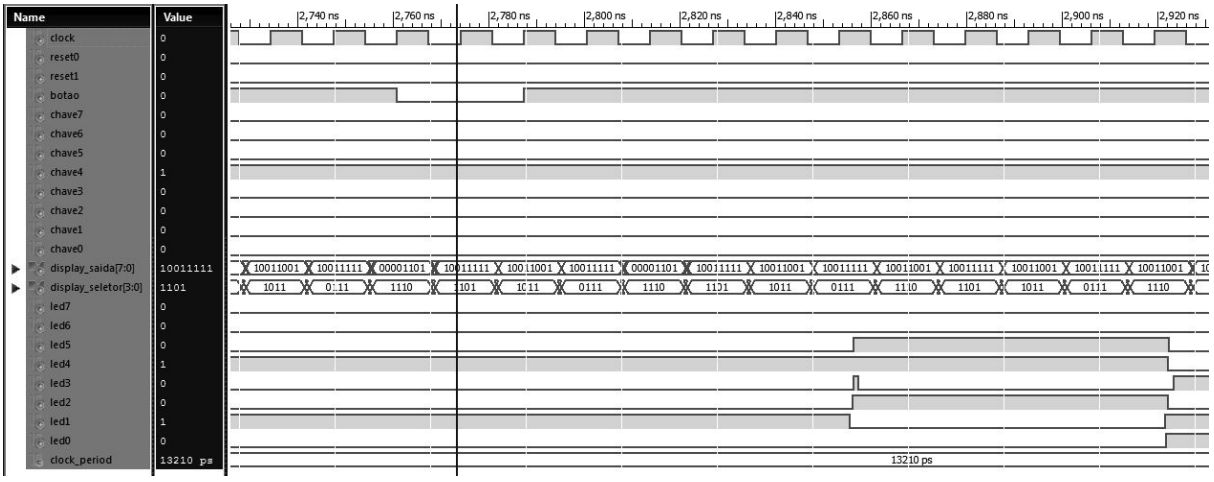


Imagem 26: no intervalo [2.810 ns, 2.860 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **13** e nos do AC temos **14**, o que mostra que a soma 6 + E dos elementos 2x1 foi feita corretamente, e que as três instruções foram realizadas com sucesso

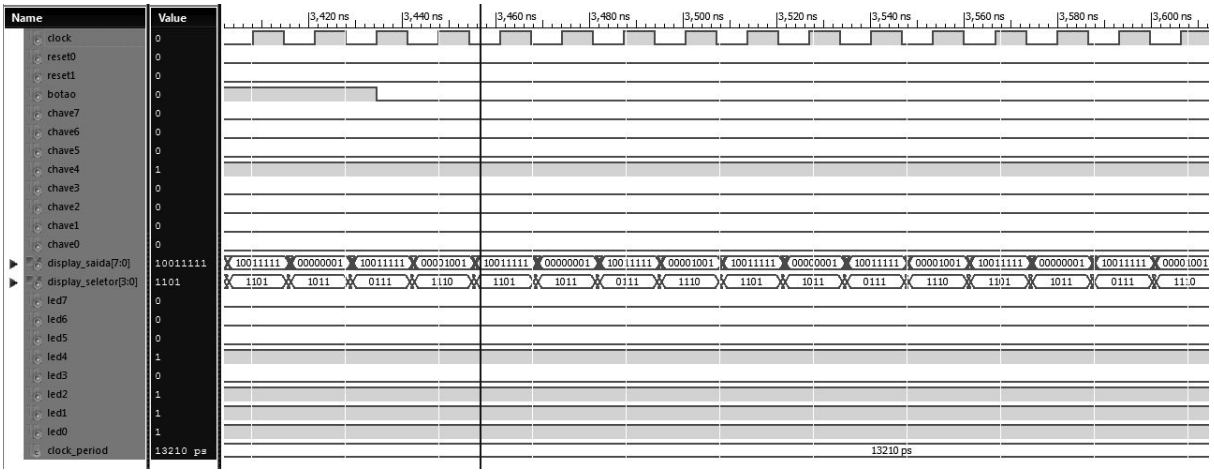
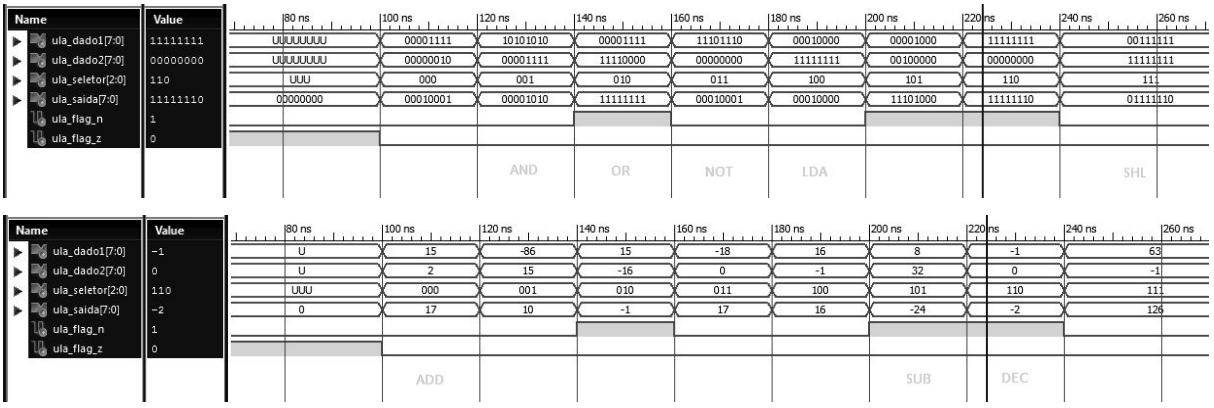


Imagem 27: no intervalo [3.440 ns, 3.490 ns] todos os quatro displays são mostrados após a realização das três instruções básicas LDA, ADD e STA. Nos displays do PC temos **19** e nos do AC temos **18**, o que mostra que a soma 8 + 10 dos elementos 2x2 foi feita corretamente, e que as três instruções foram realizadas com sucesso

O período de relógio utilizado na simulação com atraso foi o período mínimo estipulado pelo *ISE* durante a síntese: 13,210 ns.



Imagens 28 e 29: o *testbench* da ULA mostra que as novas instruções adicionadas, SHL, SUB e DEC funcionam corretamente. Os sinais a se observar são o *ula_dado1*, que corresponde ao AC, o *ula_dado2*, que corresponde ao dado lido da memória que foi armazenado no RDM, e o *ula_saida*, que corresponde ao resultado da operação

4. Referências

[1] Pedroni, Volnei. *Circuit Design with VHDL, 1. ed.* 2004.

[2] XILINX. *Documentation about project navigator*. Disponível em <https://www.xilinx.com/support.html#documentation>
[3] MOODLE. *Página do Moodle da disciplina de Sistemas Digitais*. Disponível em <https://moodle.inf.ufrgs.br/course/view.php?id=307>
[4] Weber, Raul. *Fundamentos de Arquitetura de computadores. 4. ed.* 2012.