UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

KAUÊ CHRISTMANN CAMPOS

Implementação de Máquinas Hipotéticas (NEANDER e AHMES) e Interface VGA

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Renato Perez Ribas

Porto Alegre
2016

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Vladimir Pinheiro do Nascimento
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Raul Fernando Weber
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“It does not matter how slowly you go
as long as you do not stop.”

— CONFUCIUS

AGRADECIMENTOS

A esta universidade, seu corpo docente, direção e administração que me oportunizaram as condições necessárias para que eu alcançasse meus objetivos.

Ao meu orientador por todo o tempo que dedicou a me ajudar durante este trabalho.

Aos meus pais pelo amor, ensinamento e apoio.

Emfim, a todos que contribuíram para a realização deste trabalho, seja de forma direta ou indireta, fica registrado aqui, o meu muito obrigado!

RESUMO

Este trabalho busca fazer uma abordagem didática sobre as diferentes formas de implementação das máquinas hipotéticas (Ahmes e Neander), assim como das linguagens de descrição de hardware (Verilog e VHDL) utilizadas na implementação destas. Também é exposto neste trabalho uma forma de apresentar as arquiteturas desenvolvidas fazendo uso de um kit de desenvolvimento FPGA e um monitor. Ainda neste escopo será descrito um driver para mostrar caracteres de texto ASCII em um monitor VGA com a finalidade de facilitar o uso deste em projetos diversos e também é definida e implementada uma interface genérica.

Palavras-chave: FPGA. máquinas hipotéticas. Ahmes. Neander. Verilog. VHDL. VGA.

Implementation of the hypothetical machines (NEANDER and Ahmes) and VGA interface

ABSTRACT

This work seeks a didactic approach of the different implementation methods for the hypothetical machines (Ahmes and Neander) and the hardware description languages (Verilog and VHDL) that implement these computers. It is also exposed within this work a way of presenting the designed architectures using a FPGA developer kit. Still, in this scope it will be developed a driver for displaying ASCII text characters on the VGA display for the purpose of facilitating its use on a variety of projects and also a generic interface is defined and implemented as well.

Keywords: FPGA, hypothetical machines, Ahmes, Neander, Verilog, VHDL, VGA.

LISTA DE ABREVIATURAS E SIGLAS

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
FPGA	Field Programmable Gate Array
ROM	Read Only Memory
RAM	Random Access Memory
FSM	Finite State Machine
VGA	Video Graphics Array

LISTA DE FIGURAS

Figura 2.1 Menu de Memória 1 17
Figura 2.2 Menu de Memória 2 17
Figura 2.3 Menu de Memória 3 18
Figura 2.4 Menu de Memória 4 19
Figura 2.5 Menu de Memória 5 19
Figura 2.6 Menu de Memória 6 20
Figura 2.7 Menu de Memória 7 20
Figura 2.8 Menu de Memória 8 21
Figura 2.9 Ciclo de leitura de memória...23
Figura 2.10 Ciclo de escrita na memória 24

Figura 3.1 Diagrama de Blocos...27
Figura 3.2 Arquitetura UC temporizador...44
Figura 3.3 Diagrama de estados da FSM para ROM 46
Figura 3.4 Diagrama de estados da FSM para RAM 48

Figura 4.1 Varredura do quadro VGA...54
Figura 4.2 Restrições de tempo H_SYNC 55
Figura 4.3 Restrições de tempo V_SYNC 56
Figura 4.4 Arquitetura driver VGA..58
Figura 4.5 Arquitetura Gerador de Pixels 60
Figura 4.6 Exemplo de desenho de caractere((CHU, 2008)) 61

Figura 5.1 Arquitetura divisor de frequências...65
Figura 5.2 Simulação do divisor de frequências 66
Figura 5.3 Frequência de H_SYNC no osciloscópio 68
Figura 5.4 Frequência de V_SYNC no osciloscópio 69
Figura 5.5 Demonstração projeto VGA genérico no monitor............................69
Figura 5.6 Demonstração de Ahmes no monitor 72

LISTA DE TABELAS

Tabela 3.1 Tabela de Instruções 26
Tabela 3.2 Execução das instruções 29
Tabela 3.3 Estados e Ações 47
Tabela 3.4 Estados e Ações 49

Tabela 4.1 Resoluções e suas Especificações 57

SUMÁRIO

1 INTRODUÇÃO ... **11**
1.1 Motivação ... **12**
1.2 Objetivo ... **12**
1.3 Estrutura ... **12**
2 MEMÓRIA ... **14**
2.1 Memória ROM ... **14**
2.2 Memória RAM ... **16**
2.2.1 Implementação via Ferramenta ... 16
2.2.2 Implementação Via Inferência ao Compilador ... 21
2.2.3 Simulação de Funcionamento ... 23
3 MÁQUINAS HIPOTÉTICAS ... **25**
3.1 Tabela de Instrucções ... **25**
3.2 Arquitetura ... **27**
3.3 Execução das Instruções ... **27**
3.4 Implementação de Blocos da Arquitetura ... **29**
3.4.1 PC ... 30
3.4.2 REM, RDM, RI, AC e Registrador de Estados ... 32
3.4.3 MUX ... 35
3.4.4 Decodificador ... 36
3.4.5 ULA ... 39
3.5 Unidades de Controle ... **43**
3.5.1 Unidades de Controle para ROM ... 43
3.5.1.1 Temporizador ... 43
3.5.1.2 FSM ... 45
3.5.2 Unidades de Controle para RAM ... 47
3.5.2.1 Temporizador ... 47
3.5.2.2 FSM ... 48
3.6 Simulação Testbench ... **49**
3.7 Empacotador de Memória ... **52**
4 VGA (VIDEO GRAPHICS ARRAY) ... **54**
4.1 Especificações VGA ... **54**
4.2 Arquitetura Driver VGA ... **57**
4.3 Controle VGA ... **58**
4.3.1 Sinais de Sincronização ... 58
4.3.2 Requerimento de Pixel ... 59
4.4 Gerador de Pixels ... **60**
4.4.1 Memória de Caracteres ... 60
4.4.2 Memória de Tela ... 61
4.4.3 Funcionamento ... 62
5 PROJETO VGA GENÉRICO ... **64**
5.1 Interface e Driver VGA ... **64**
5.1.1 Divisor de Frequências ... 64
5.1.2 Interface ... 66
5.2 Projeto VGA ... **67**
5.3 Computadores Hipotéticos no Monitor ... **70**
6 CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS ... **73**
REFERÊNCIAS ... **74**
7 APÊNDICE ... **75**

1 INTRODUÇÃO

Para o aprendizado de arquitetura de computadores de microprogramação em vez de utilizar máquinas reais são usados simuladores de máquinas hipotéticas. No processo da aprendizagem estes simuladores facilitam no material de estudo, que está sempre em expansão, e também aos recursos materiais que se tornam mais acessíveis. Os simuladores de CPU auxiliam o aprendizado ativo ao permitir que os alunos usem máquinas hipotéticas para programar, executar e depurar software de sistema e usar simulação para entender o funcionamento de máquinas reais (WOLFFE et al., 2002). Atualmente, existem vários simuladores: Neander, Ahmes, Ramses, Cesar (WEBER, 2009); PIPPIN machine (DECKER; HIRSHFIELD, 2001); CPU Sim (SKRIEN, 1994). O intento destes é o de ensinar os fundamentos de arquitetura de hardware, lógica de microprogramação e padrões de expressão de valores lógicos.

Os FPGAs (Field Programmable Gate Arrays) são dispositivos semicondutores que baseiam-se em torno de uma matriz de blocos lógicos configuráveis (CLBs) que são conectados por meio de interconexões programáveis. Esses dispositivos podem ser reprogramados de acordo com os requisitos de aplicação ou funcionalidade desejados após a fabricação. Esse recurso distingue os FPGAs dos ASICs (Application Specific Integrated Circuits), que são fabricados sob medida para tarefas específicas (FPGA..., 2016).

A reprogramação do FPGA é feita através das ferramentas disponibilizadas por seu fabricante. Atualmente, os dois maiores fabricantes são Xilinx (XILINX..., 2016) e Altera (ALTERA..., 2016), sendo suas plataformas de desenvolvimento ISE e Quartus II respectivamente. As plataformas de desenvolvimento oferecem diferentes métodos para reprogramação do FPGA e dentre esses métodos encontram-se a programação via diagrama de blocos e as linguagens de descrição de hardware VHDL e Verilog.

Um kit de desenvolvimento FPGA consiste em uma placa onde são acoplados diversos componentes e interfaces que auxiliam a programação e utilização de um chip FPGA. Existem kits com diferentes características que são utilizados tanto para o aprendizado, a prototipação de um produto e testes de conceito como para aplicação final. Alguns exemplos de kits de desenvolvimento e seus diferenciais são: Altera DE2 Board com decodificador para TV, entrada Ethernet e porta RS232 (VGA); Altera DE0 Board com portas RS232 (VGA) e PS/2; Spartan-3A Starter Kit Board com conector de áudio, tela LCD 16x2 caracteres, portas RS232 (VGA) e PS/2;

1.1 Motivação

As disciplinas introdutórias dos cursos de Engenharia da Computação e Ciências da Computação da UFRGS fazem uso de máquinas hipotéticas, criadas por professores da instituição para fins didáticos, contribuindo em disciplinas do currículo como fundamentos de arquitetura de computadores, circuitos digitais e organização de computadores.

Motivado pelos conhecimentos adquiridos no decorrer do curso, tendo em vista que já existem trabalhos que implementam esses computadores hipotéticos, por exemplo, Implementação em hardware da arquitetura do computador hipotético CESAR (ORTH, 2010). Este trabalho se propõe a desenvolver um projeto para auxiliar tanto o aprendizado do aluno quanto a transmissão de conhecimento pelo professor.

1.2 Objetivo

Este trabalho tem como objetivo integrar o conteúdo de diferentes disciplinas para analisar alguns destes computadores hipotéticos comparando e destacando suas diferenças. Além disso, propõe-se a potencializar o aproveitamento dos recursos disponíveis nos laboratórios a fim de melhor demonstrar o funcionamento destes computadores hipotéticos amplamente utilizados no currículo dos cursos de Ciências da Computação e Engenharia da Computação da Universidade Federal do Rio Grande do Sul.

Para tal, este trabalho busca analisar tanto o conhecimento teórico-pedagógico quanto o conhecimento prático no uso das ferramentas disponíveis. Na implementação deste trabalho, os projetos tiveram base no uso do kit de desenvolvimento DE0 e no ambiente de desenvolvimento Quartus II.

1.3 Estrutura

O Capítulo 2 apresenta uma breve explicação sobre os variados tipos de memória, pois os projetos envolvidos neste trabalho necessitam deste componente. Ainda neste capítulo são apresentadas variadas formas de implementação para diferentes tipos de memória.

No Capítulo 3 as máquinas hipotéticas Ahmes e Neander são caracterizadas. Os diversos componentes que integram a arquitetura destas máquinas são analisados e im-

plementados em duas linguagens de descrição de hardware. Devido ao fato de Ahmes e Neander serem ferramentas que simulam máquinas hipotéticas existem vários modelos de implementação a nível de hardware, neste capítulo também serão descritas diferentes formas de implementação destes.

Com a intenção de demonstrar o funcionamento dos computadores implementados, o Capítulo 4 apresenta uma breve fundamentação teórica sobre VGA com o propósito de desenvolver um meio de fazer uso deste recurso disponível na placa DE0. A concepção e implementação de um driver VGA é desenvolvida.

A fim de criar uma forma prática de utilizar o driver VGA, o Capítulo 5 define e implementa uma interface para o uso do driver, inclusive é apresentado um tutorial para o uso desta. Também é exemplificada a demonstração de um dos computadores implementados em um monitor.

Finalmente, o Capítulo 6 relata os objetivos alcançados e dificuldades enfrentadas durante o desenvolvimento deste trabalho, propondo ainda sugestões de trabalhos futuros.

2 MEMÓRIA

A memória é um componente fundamental, tanto em um computador, pois ela armazena as instruções e os dados que serão utilizados na execução de um programa pela máquina, quanto em um sistema de vídeo, onde ela armazena as informações que são mostradas na tela. Neste capítulo serão apresentadas duas categorias de memória que definem essencialmente suas funcionalidades.

A ROM (Read Only Memory) é uma memória que após a gravação de seus dados não é possível apagar ou editar nenhuma informação, permitindo somente acessar a mesma. Diferentemente, a RAM (Random Access Memory) é aquela que permite a gravação e regravação de seus dados.

Considerando as categorias expostas acima, existem diferentes tipos de dispositivos com características distintas, como por exemplo: PROM (Programmable ROM); EEPROM (Electrically-Erasable ROM); SRAM (Static RAM); DRAM (Dynamic RAM), SDRAM (Synchronous Dynamic RAM);

Partindo do ponto de vista de funcionamento lógico estas memórias podem ser implementadas um uma FPGA. Porém, buscando atingir um dos objetivos definidos, grande parte das implementações apresentadas visam utilizar o componente SDRAM presente na placa DE0, a fim de possibilitar a construção de memórias maiores sem utilizar os recursos da FPGA.

2.1 Memória ROM

Atento ao fato de que a ROM é um elemento que permite apenas a leitura e considerando um ambiente FPGA, onde toda lógica é reprogramável, o termo memória ROM fica designado a uma memória que não é modificada durante a execução um projeto.

A seguir são apresentadas duas opções de implementação. A primeira delas é a simplificação de uma ROM em um circuito combinacional onde para cada valor de entrada existe uma saída previamente definida. O código 2.1 descreve uma ROM utilizando a linguagem VHDL.

Código 2.1 – Memória ROM

```
library ieee;
use ieee.std_logic_1164.all;

```

```vhdl
entity  MemoriaRom is
     port  (endr:  in   std_logic_vector (7  downto 0  );
             dado:  out   std_logic_vector (  7  downto 0));
end MemoriaRom;


architecture   memoria of MemoriaRom is
begin
process (  endr  )
begin
     case  endr  is
          WHEN "00000000" => dado <= "00100000" ; --LDA
          WHEN "00000001" => dado <= "10000000" ; --ender
          WHEN "00000010" => dado <= "00110000" ; --ADD
          WHEN "00000011" => dado <= "10000001" ; --ender
          WHEN "00000100" => dado <= "10100000" ; --JZ
          WHEN "00000101" => dado <= "00001100" ; --ender
          WHEN "00000110" => dado <= "10010000" ; --JN
          WHEN "00000111" => dado <= "00001100" ; --ender
          WHEN "00001000" => dado <= "01000000" ; --OR
          WHEN "00001001" => dado <= "10000010" ; --ender
          WHEN "00001010" => dado <= "10010000" ; --JN
          WHEN "00001011" => dado <= "00001101" ; --ender
          WHEN "00001100" => dado <= "11110000" ; --HLT
          WHEN "00001101" => dado <= "01100000" ; --NOT
          WHEN "00001110" => dado <= "01010000" ; --AND
          WHEN "00001111" => dado <= "10000010" ; --ender
          WHEN "00010000" => dado <= "10100000" ; --JZ
          WHEN "00010001" => dado <= "00010011" ; --ender
          WHEN "00010010" => dado <= "11110000" ; --HLT
          WHEN "00010011" => dado <= "01100000" ; --NOT
          WHEN "00010100" => dado <= "00010000" ; --STA
          WHEN "00010101" => dado <= "10000011" ; --ender
          WHEN "00010110" => dado <= "10000000" ; --JMP
          WHEN "00010111" => dado <= "00011001" ; --ender
          WHEN "00011000" => dado <= "11110000" ; --HLT
          WHEN "00011001" => dado <= "01010000" ; --AND
          WHEN "00011010" => dado <= "10000100" ; --ender
          WHEN "00011011" => dado <= "00010000" ; --STA
          WHEN "00011100" => dado <= "00100000" ; --ender
```

```
45          WHEN "00011101" => dado <= "00000000" ; --NOP
            WHEN "00011110" => dado <= "00000000" ; --NOP
47          WHEN "00011111" => dado <= "00000000" ; --NOP
            WHEN "00100000" => dado <= "00000000" ; --NOP
49          WHEN "10000000" => dado <= "00001010" ; --dados
            WHEN "10000001" => dado <= "00000001" ; --dados
51          WHEN "10000010" => dado <= "10001111" ; --dados
            WHEN "10000011" => dado <= "00000000" ; --dados
53          WHEN "10000100" => dado <= "11110000" ; --dados

55          WHEN others => dado <= "00000000";
        end case;
57  end process;
    end memoria;
```

Referente ao funcionamento, pelo fato de não depender de um relógio, este método de implementação pode ser considerado com um circuito combinacional, onde o dado fica disponível na saída no mesmo ciclo de relógio em que o endereço de memória foi atualizado.

Esta forma de implementação utiliza os recursos da FPGA, outras implementações deste tipo de memória podem ser vistas nos anexos 7.1 e 7.2.

A segunda opção de implementação consiste em usar uma memória RAM para emular uma ROM, os tempos de leitura de dados na memória devem ser respeitados de acordo com a especificação da memória.

2.2 Memória RAM

Nesta Subseção serão descritos dois métodos de implementação para uma RAM e um estudo sobre seu funcionamento e tempos de acesso à memória. Esses dois métodos são formas diferentes de indicar que o compilador deve utilizar o componente de memória presente no kit de desenvolvimento.

2.2.1 Implementação via Ferramenta

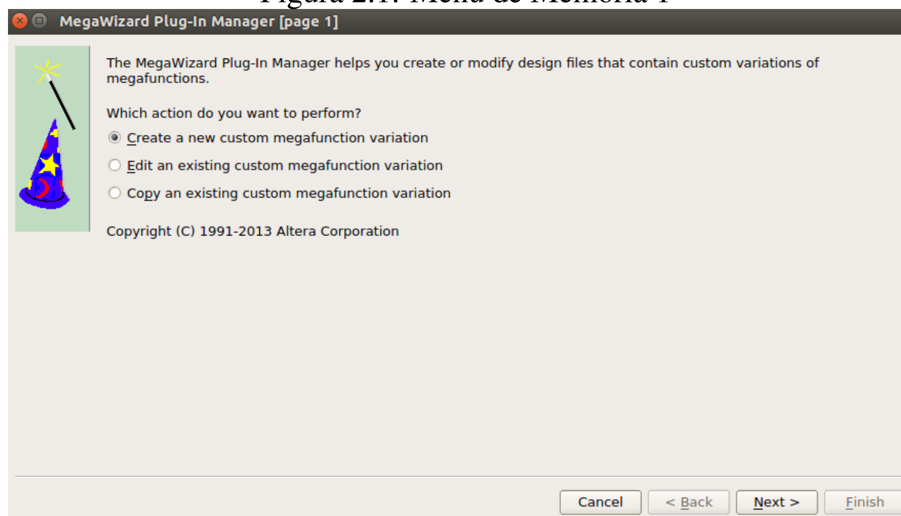
Para tal, a plataforma de desenvolvimento (Quartus II) oferece uma ferramenta para criação das mega funções de IP da ALTERA, utilizando-se desta ferramenta é possí-

vel criar uma memória RAM com aspectos diversos.

A seguir será explicado o fluxo para criação de uma memória RAM com 256 endereços de memória de 8 bits cada, com uma entrada de endereço de 8 bits, uma entrada de dado de 8 bits, uma entrada de 1 bit para habilitar escrita e uma saída de dado de 8 bits.

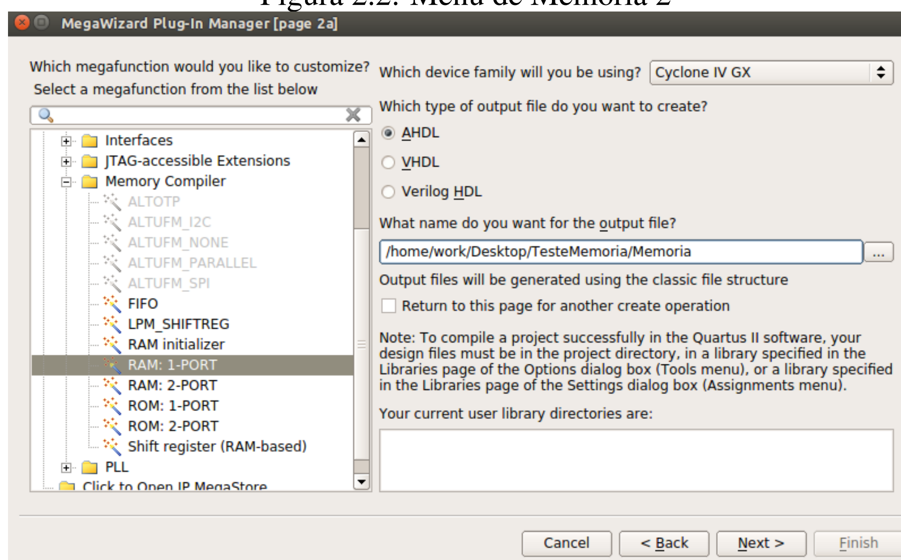
Dentro do Quartus II a aba de menu Tools→MegaWizard Plug-In Manager inicializa a ferramenta para criação das mega funções de IP da ALTERA, Figura 2.1, escolhendo a opção para criar uma nova mega função.

Figura 2.1: Menu de Memória 1



No menu da Figura 2.2 é feita a escolha da mega função desejada, a linguagem na qual ela deve ser implementada e o nome do módulo.

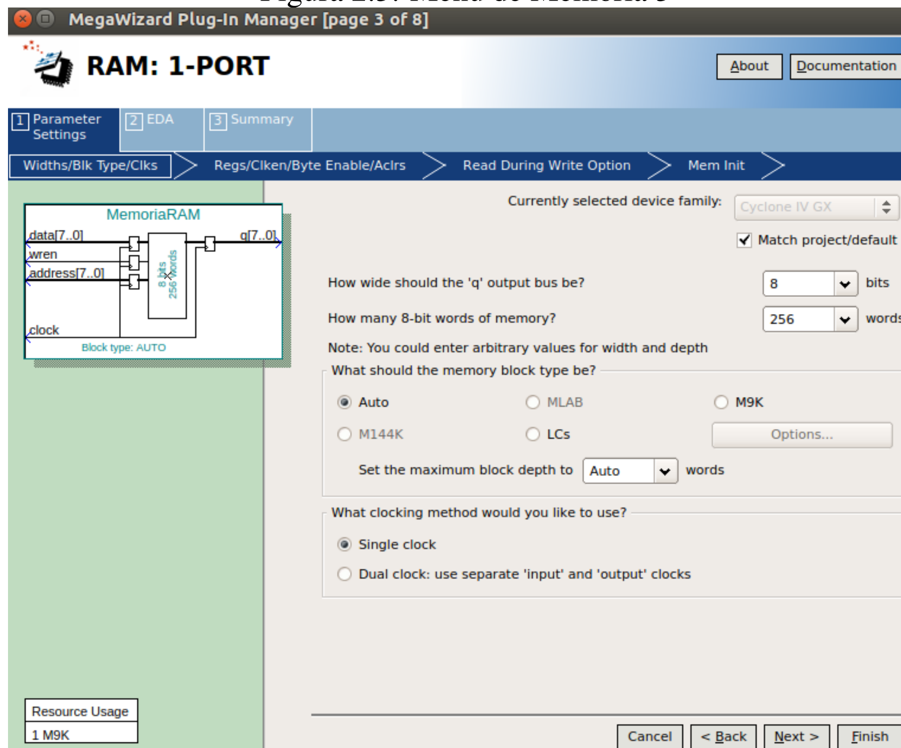
Figura 2.2: Menu de Memória 2



No menu da Figura 2.3, é selecionado o dispositivo-alvo(placa FPGA). A escolha de tamanho da palavra de memória e quantidade de palavras da memória. O tipo

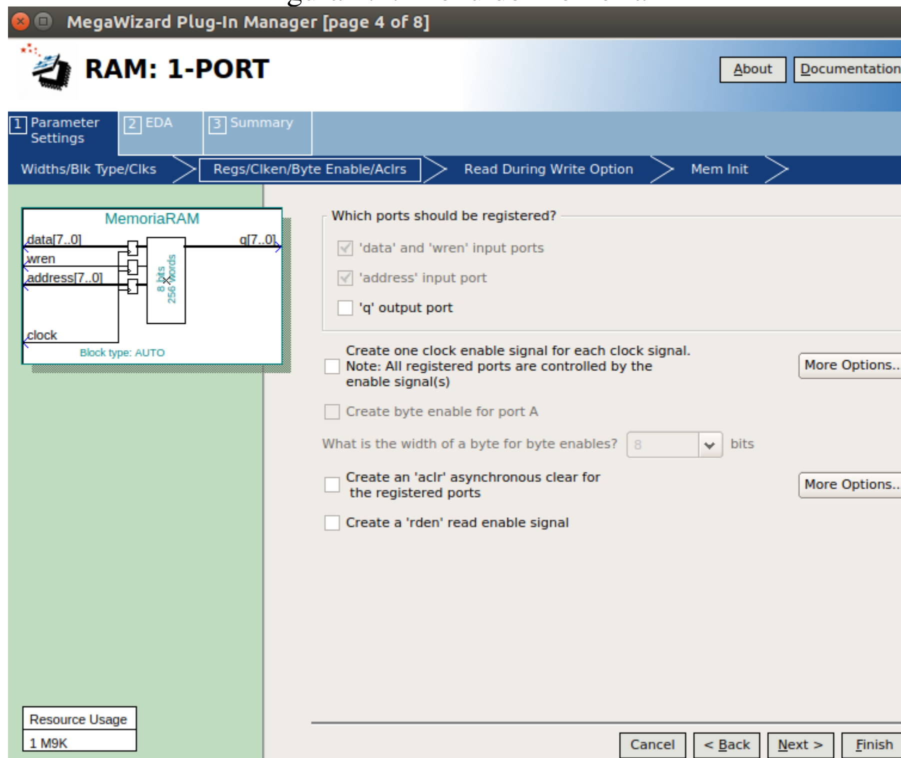
de bloco utilizado pelo dispositivo, sendo LCs(Logic Cells) utilizadas pela FPGA para implementar qualquer tipo de lógica e M9K células especificas para implementar memória. Método de relógio, único: entradas e saída controladas pelo mesmo relógio; duplo: entradas controladas por um relógio e saída por outro.

Figura 2.3: Menu de Memória 3



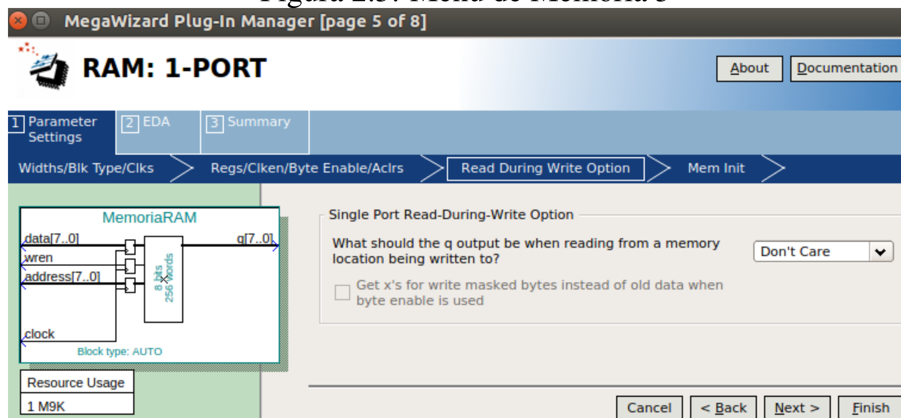
No Menu da Figura 2.4, na escolha de quais portas terão registradores, é retirado o registrador de saída da memória(porta 'q') pois na arquitetura definida o REM exerce essa função. Há também opções referentes a sinais de controle adicionais que não são utilizados pela arquitetura já definida dos comutadores Neander e Ahmes.

Figura 2.4: Menu de Memória 4



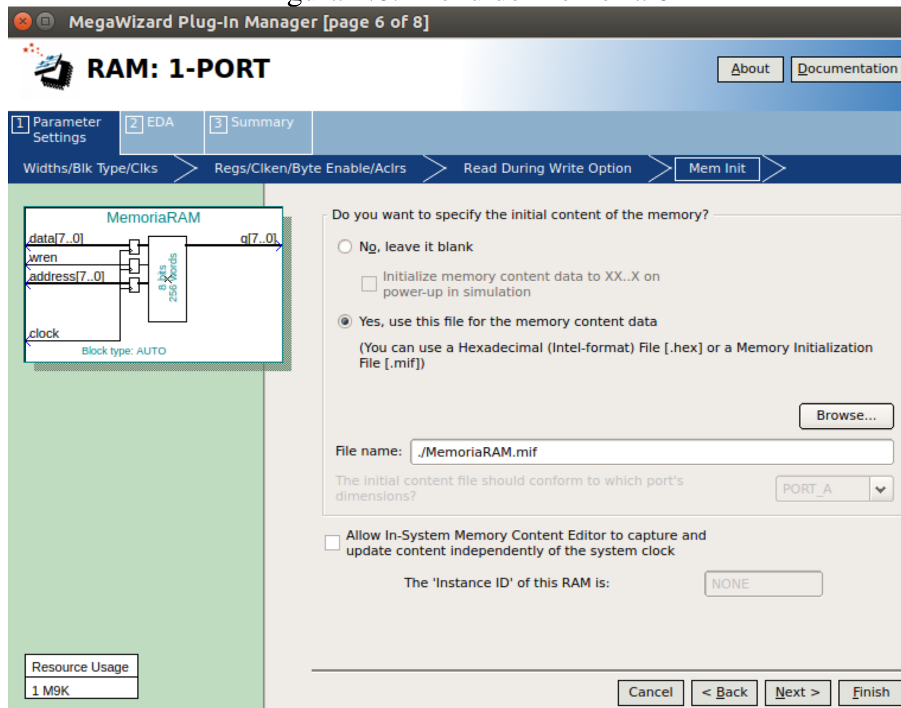
No menu da Figura 2.5 é possível escolher qual dado estará disponível na saída 'q' durante a operação de escrita na memória.

Figura 2.5: Menu de Memória 5



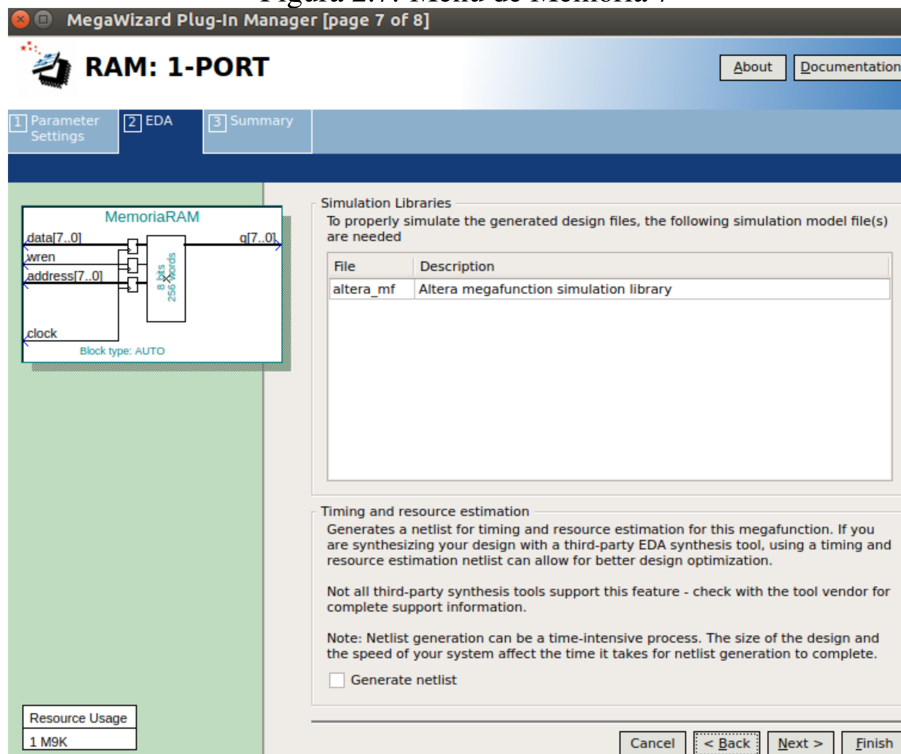
No menu da Figura 2.6 é selecionado o arquivo de inicialização da memória que pode ser em dois formatos: .mif (Memory Initialization File), e o arquivo de texto ASCII que especifica os valores iniciais para cada endereço de memória; Intel HEX, é o formato de arquivo usado para descrever o conteúdo de memória definido pela Intel Corporation.

Figura 2.6: Menu de Memória 6



No menu da Figura 2.7, existe a opção para gerar arquivos de estimativa de tempo e tamanho da memória no caso da síntese ser feita por uma ferramenta que não seja da ALTERA.

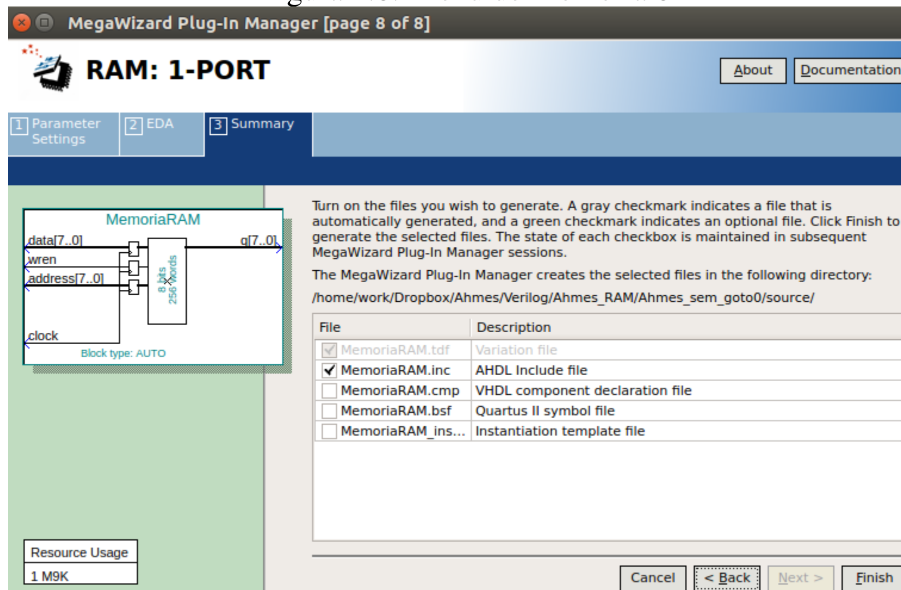
Figura 2.7: Menu de Memória 7



No menu da Figura 2.8 são escolhidos os arquivos que serão gerados pela ferra-

menta.

Figura 2.8: Menu de Memória 8



2.2.2 Implementação Via Inferência ao Compilador

Este método faz uso das capacidades do compilador existente na plataforma de desenvolvimento (Quartus II). A partir de um arquivo de texto, descrito tanto em Verilog quanto em VHDL, é possível inferir ao compilador que ele deve utilizar o componente de memória disponível no kit de desenvolvimento.

Utilizando o modelo descrito no código 2.2, em Verilog, é possível reproduzir a memória feita pela ferramenta para criação das mega funções. No código 2.3 a mesma é representada em Verilog.

Código 2.2 – Memória RAM por Inferência em Verilog

```
module MemoriaRAM(q, address, data, wren, clock);
    output reg [7:0] q;
    input      [7:0] data;
    input      [7:0] address;
    input      wren, clock;


     // Build a 2-D array type for the RAM
    reg [7:0] mem [255:0];

     always @(posedge clock) begin
          if (wren)
```

```
            mem[address] <= data;
        q <= mem[address];
    end
endmodule
```

Código 2.3 – Memória RAM por Inferência em VHDL

```
library ieee;
use ieee.std_logic_1164.all;

entity MemoriaRAMinferred is

   generic
   (
      DATA_WIDTH : natural := 8;
      ADDR_WIDTH : natural := 6
   );

   port
   (
      clock    : in std_logic;
      address  : in natural range 0 to 2**ADDR_WIDTH - 1;
      data    : in std_logic_vector ((DATA_WIDTH-1) downto 0);
      wren     : in std_logic := '1';
      q    : out std_logic_vector ((DATA_WIDTH -1) downto 0)
   );

end entity;

architecture rtl of MemoriaRAMinferred is

   -- Build a 2-D array type for the RAM
   subtype word_t is std_logic_vector ((DATA_WIDTH-1) downto 0);
   type memory_t is array(2**ADDR_WIDTH-1 downto 0) of word_t;

   -- Declare the RAM signal.
   signal ram : memory_t;

   -- Register to hold the address
   signal addr_reg : natural range 0 to 2**ADDR_WIDTH-1;

begin
```

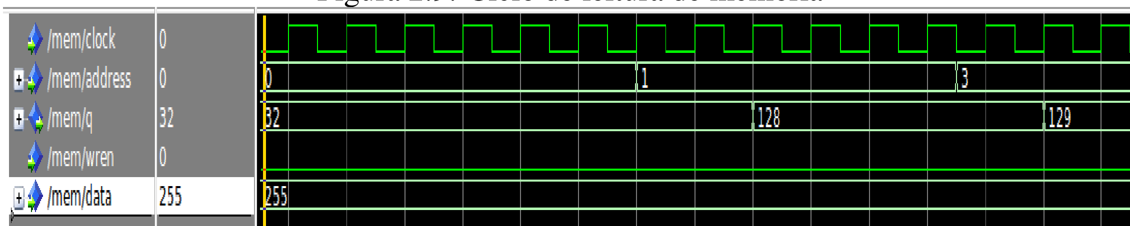
```
36
    process (clock)
38  begin
    if ( rising_edge (clock)) then
40      if (wren = '1') then
            ram(address) <= data;
42      end if;

44      -- Register the address for reading
        addr_reg <= address;
46  end if;
    end process;
48
    q <= ram(addr_reg);
50
end rtl ;
```

2.2.3 Simulação de Funcionamento

Para poder utilizar esta memória nas arquiteturas projetadas é preciso primeiro fazer um estudo do comportamento desta memória criada a partir das ferramentas oferecidas pelo Quartus II.

A Figura 2.9 demonstra um trecho da simulação da memória RAM desenvolvida. Nela podemos observar que o dado da memória 'q' estará disponível um ciclo de relógio 'clock' após a primeira borda de subida do mesmo em que o endereço do dado estiver estável.

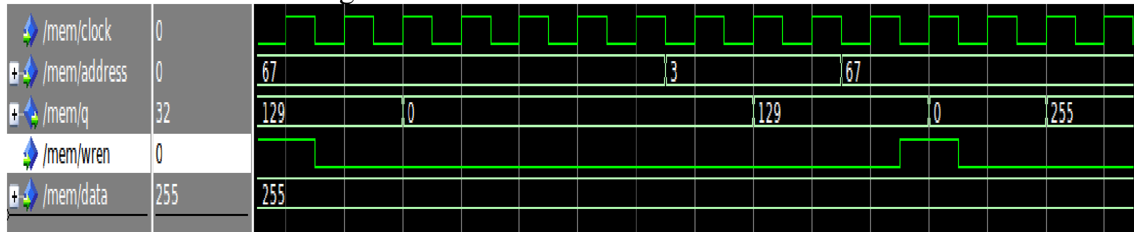
Figura 2.9: Ciclo de leitura de memória



A Figura 2.10 demonstra a simulação de uma operação de escrita na memória. Pode-se observar que a escrita acontece quando "wren"está ativo. Para garantir a que a operação seja executada corretamente deve-se garantir que o endereço e o dado a serem

escritos estejam estáveis, e "wren"deve estar ativo durante a borda de subida do relógio.

Figura 2.10: Ciclo de escrita na memória



3 MÁQUINAS HIPOTÉTICAS

Neander e Ahmes são ferramentas que simulam computadores hipotéticos. Estas ferramentas tem sua arquitetura e funcionamento descritos no livro Fundamentos de Arquitetura de Computadores (WEBER, 2009). Neste capítulo é descrito suas especificações, arquiteturas e formas de implementação.

Ambos são computadores de 8 bits com dados representados em complemento de dois.

O modo de endereçamento implementado neles é o modo direto. Isso significa que a palavra que segue o código da instrução contém o endereço de memória do operando a ser utilizado. Nas instruções de desvio a palavra que segue o código da instrução contém o endereço da próxima instrução a ser executada.

Estes computadores possuem registradores de estado. Esses registradores guardam informações sobre a última operação executada pela ULA (únidade lógica aritmética). Essa informação é usada para operações de desvio condicional.

O computador Neander tem 2 registradores de estados: N (negativo) indica se a última operação resultou em um numero negativo e Z (zero) que indica que a última operação resultou em zero.

O Computador Ahmes possui 5 registradores de estado, os mesmos N e Z do Neander mais os registradores C (carry) que indica se houve "vai-um"na soma, V (overflow) indica se houve estouro de representação na soma ou subtração e B (borrow) indica se houve "empresta-um"na subtração.

3.1 Tabela de Instrucções

O computador Neander utiliza apenas os 4 bits mais significativos da palavra para decodificar a instrução.

O computador Ahmes é uma extensão do Neander, onde a palavra de 1 byte é utilizada para decodificar instruções, estendendo o conjunto de instruções resultando em poucas modificações em relação à arquitetura do Neander. Essas modificações são mencionadas durante a análise e implementação dos módulos da arquitetura.

A Tabela 3.1 descreve todas as operações dos computadores Neander e Ahmes.

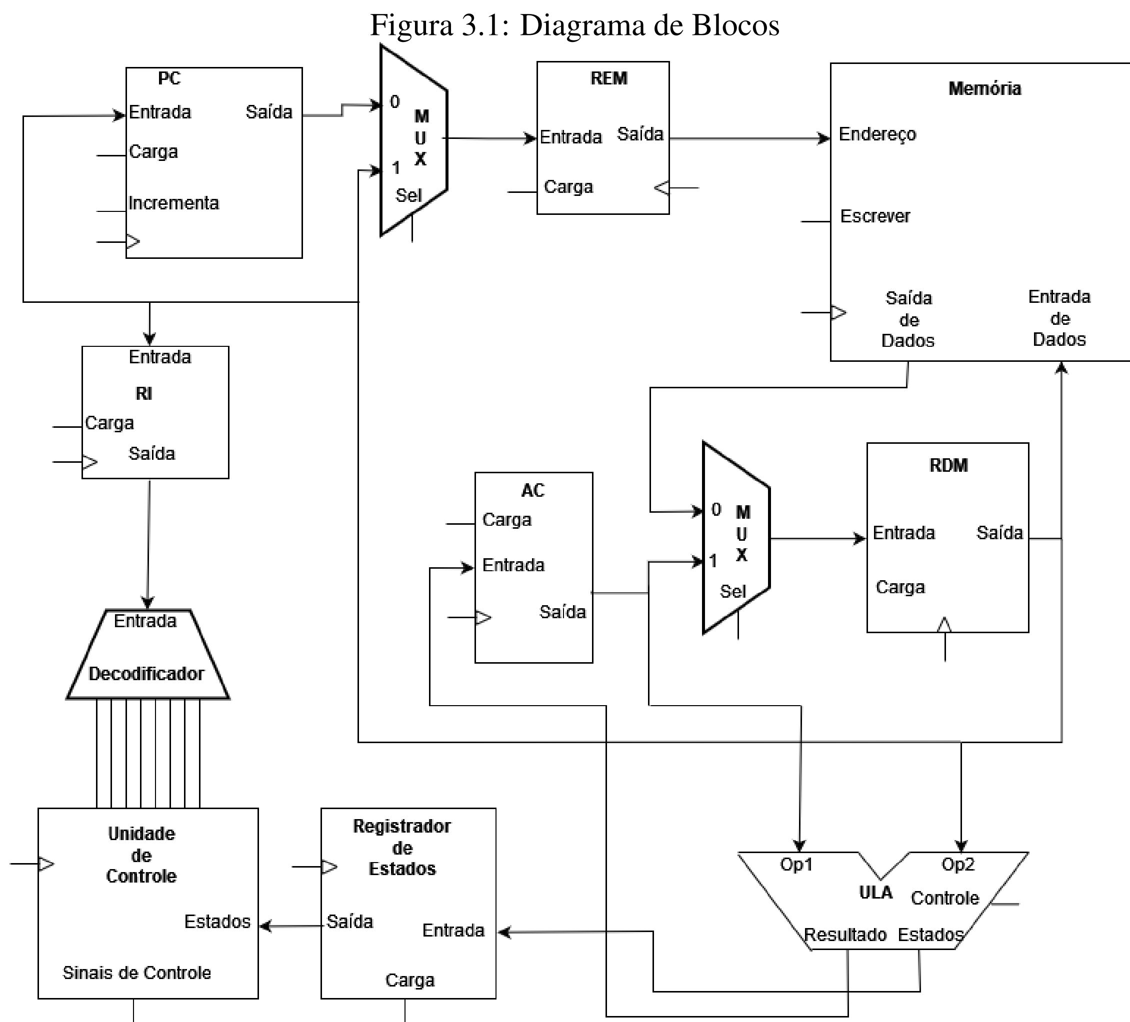
Tabela 3.1: Tabela de Instruções

Codigo	Instrução Neander	Instrução Ahmes	Descrição
0000 0000	NOP	NOP	nenhuma operação
0001 0000	STA end	STA end	armazena acumulador na memória
0010 0000	LDA end	LDA end	carrega acumulador
0011 0000	ADD end	ADD end	soma acumulador com dado da memoria
0100 0000	OR end	OR end	operação lógica "or"entre acumulador e dado da memoria
0101 0000	AND end	AND end	operação lógica "and"entre acumulador e dado da memoria
0110 0000	NOT	NOT	inverte acumulador
0111 0000	———	SUB end	subtrai dado da memoria do acumulador
1000 0000	JMP end	JMP end	desvio incondicional
1001 0000	JN end	JN end	desvio condicional (pula se negativo)
1001 0100	———	JP end	desvio condicional (pula se positivo)
1001 1000	———	JV end	desvio condicional (pula se overflow)
1001 1100	———	JNV end	desvio condicional (pula se não ocorreu overflow)
1010 0000	JZ end	JZ end	desvio condicional(pula se zero)
1010 0100	———	JNZ end	desvio condicional (pula se não zero)
1011 0000	———	JC end	desvio condicional(pula se carry)
1011 0100	———	JNC end	desvio condicional (pula se não carry)
1011 1000	———	JB end	desvio condicional(pula se borrow)
1011 1100	———	JNB end	desvio condicional (pula se não borrow)
1110 0000	———	SHL	desloca acumulador para direita
1110 0001	———	SHR	desloca acumulador para esquerda
1110 0010	———	ROL	rotação do acumulador para direita
1110 0011	———	ROR	rotação do acumulador para esquerda
1111 0000	HLT	HLT	termino da execução

3.2 Arquitetura

A Figura 3.1 ilustra o diagrama de blocos que é compatível tanto para o Neander quanto para o Ahmes.

Figura 3.1: Diagrama de Blocos



Estes computadores foram implementados em VHDL e Verilog. A implementação de cada um dos componentes do diagrama de blocos da Figura 3.1 é apresentada na Seção 3.4.

3.3 Execução das Instruções

As instruções seguem um fluxo dentro da arquitetura, através da carga de registradores e operações aritméticas. Primeiro é feita a busca pela instrução, cuja sequência é sempre igual, independente da instrução que será executada.

REM ← PC

RDM ← MEM, PC ← PC + 1

RI ← RDM

A partir desta etapa começa a execução da instrução. Ambos os computadores Ahmes e Neander possuem os mesmos fluxos de execução, a diferença está na quantidade de instruções que é mais abrangente no Ahmes. No restante desta Subseção serão descritos os fluxos para o computador Ahmes, e estes podem ser facilmente adaptados para o computador Neander ao retirar as instruções que não o pertencem.

As instruções STA, NOP, e HLT têm seu próprio fluxo de execução e as restantes podem ser organizadas em grupos que seguem o mesmo fluxo de execução:

Instruções de Desvio, onde ocorre o desvio (Desvio sim):

- JMP
- JN, N = 1
- JZ, Z = 1
- JV, V = 1
- JC, C = 1
- JB, B = 1
- JP, N = 0
- JNZ, Z = 0
- JNV, V = 0
- JNC, C = 0
- JNB, B = 0

Instruções de Desvio onde não ocorre o desvio (Desvio não):

- JN, N = 0
- JZ, Z = 0
- JV, V = 0
- JC, C = 0
- JB, B = 0
- JP, N = 1
- JNZ, Z = 1
- JNV, V = 1
- JNC, C = 1

- JNB, B = 1

Instruções de operação da ULA apenas sobre acumulador (ULA dado):

- NOT
- SHL
- SHR
- ROL
- ROR

Instruções de operação da ULA entre acumulador e dado da memória (ULA acumulador):

- LDA
- ADD
- OR
- AND
- SUB

A Tabela 3.2 organiza a atividade dos componentes da arquitetura durante os fluxos de execução de cada tipo de instrução.

Tabela 3.2: Execução das instruções

	STA	ULA dado	Desvio sim	Desvio não	ULA acumulador	NOP	HLT
0	REM← PC	REM← PC	REM← PC	REM← PC	REM← PC	REM← PC	REM← PC
1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1	RDM← MEM PC←PC+1
2	RI←RDM	RI←RDM	RI←RDM	RI←RDM	RI←RDM	RI←RDM	RI←RDM
3	REM←PC	REM←PC	REM←PC	PC←PC+1	AC←ULA		Fim de Execução
4	RDM←MEM PC←PC+1	RDM←MEM PC←PC+1	RDM←MEM				
5	REM←RDM	REM←RDM	PC←RDM				
6	RDM←AC	RDM←MEM					
7	MEM←RDM	AC←ULA					

3.4 Implementação de Blocos da Arquitetura

Nesta Seção estão descritos todos os elementos ilustrados na Figura 3.1, definindo suas entradas, saídas e as funções que desempenham.

Os módulos de Memória e Unidade de Controle são analisados em uma Seção a parte, ver Seção 3.5.

3.4.1 PC

O PC (Contador de Programa) é composto por um registrador de 8 bits e um somador. Tem como função manter o controle sobre o endereço de memória que contém a próxima instrução a ser executada. Entradas :

- Clk : relógio do sistema, ativa o módulo em toda borda de subida do relógio;
- Entrada : endereço a ser carregado;
- Carga : sinal de comando para carregar Endereço;
- Incrementa : sinal de comando para incrementar o valor atual em uma unidade;

Saída:

- Saída : próximo endereço de memória;

O Código 3.1 implementa o módulo PC em Verilog.

Código 3.1 – PC Verilog

```
module count8_PCv(
        clk ,
        rst ,
        carga ,
        incrementa ,
        entrada ,
        pc
    );
    input  clk ;
    input  rst ;
    input  carga ;
    input  incrementa ;
    input  [7:0] entrada ;
    output  [7:0] pc ;
    reg  [7:0]  regist ;
    assign  pc  =  regist ;
    always  @ (  posedge clk  or  posedge  rst )
    begin
        if  (  rst  == 1'b1  )
            begin
                regist  <= {  1'b0  };
            end
        else
```

```
                begin
25              if  ( clk  )
                    begin
27                      if  ( incrementa  == 1'b1  )
                            begin
29                              regist  <= entrada;
                            end
31                      else
                            begin
33                              if  ( carga  == 1'b1  )
                                    begin
35                                      regist  <= (  regist  + 1'b1  );
                                    end
37                          end
                    end
39          end
        end
41  endmodule
```

O Código 3.2 implementa o módulo PC em VHDL.

Código 3.2 – PC VHDL

```
1   library  IEEE;
    use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.STD_LOGIC_ARITH.ALL;
    use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
    ENTITY count8_PC is
7       PORT (
            clk  :  in  std_logic ;
9           rst  :  in  std_logic ;
            incrementa  :  in   STD_LOGIC;
11      carga  :  in  std_logic ;
        entrada  :  in  std_logic_vector  (7  downto 0);
13      pc  :  out   std_logic_vector  (7  downto 0)  );
    end count8_PC;
15
    architecture  Behavioral  of  count8_PC is
17
    signal  reg  :  std_logic_vector  (7  downto 0)  :=  ( others  => '0') ;
19
    begin
```

```
21
    pc <= reg;
23
    process (clk, rst)
25  begin
      if rst = '1' then
27      reg <= (others => '0');
      elsif (clk'event and clk = '1') then
29      if (carga = '1') then
          reg <= entrada;
31      elsif (incrementa = '1') then
          reg <= reg + 1;
33      end if;
      end if;
35
    end process;
37 end Behavioral;
```

3.4.2 REM, RDM, RI, AC e Registrador de Estados

Estes módulos possuem a mesma implementação, sendo utilizada para armazenar valores diferentes durante o fluxo de execução da arquitetura. A funcionalidade de suas portas é descrita abaixo:

- Entrada: valor à ser armazenado;
- Carga : sinal de comando para armazenar o valor da Entrada;
- clk : relógio do sistema, toda borda de subida do relógio ou armazena novo dado (Carga = 1) ou mantém valor atual (Carga = 0);
- Saída : valor atual armazenado no registrador;

A seguir cada um deles é descrito de acordo com sua função:

REM (Registrador do Endereço de Memória) é um registrador de 8 bits tem como função armazenar o endereço de memória que será lido pela memória.

RDM (Registrador de Memória) é um registrador de 8 bits tem como função armazenar o dado que vem da memória ou o dado que será escrito na memória.

RI (Registrador de Instrução) é um registrador de 8 bits tem como função armazenar a instrução que está sendo executada.

Acumulador é um registrador de 8 bits tem como função armazenar o resultado das operações realizadas pela ULA.

O Registrador de Estados é um registrador de 5 bits (AHMES) ou 2 bits (NEANDER), tem como função armazenar o estado da última operação realizada pela ULA.

O Código 3.3 implementa o módulo Registrador em Verilog.

Código 3.3 – Registrador Verilog

```
module register8bits (
        clk,
        rst,
        carga,
        entrada,
        saida
    );

    input clk;
    input rst;
    input carga;
    input [7:0] entrada;
    output [7:0] saida;
    reg [7:0] register ;

    assign saida = register ;

    always @ (posedge clk or posedge rst )
    begin
        if ( rst == 1'b1 )
            begin
                register <= { 1'b0 };
            end
        else
            begin
                if ( clk )
                    begin
                        if ( carga == 1'b1 )
                            begin
                                register <= entrada;
                            end
                    end
            end
    end
```

```
endmodule
```

O Código 3.4 implementa o módulo Registrador em VHDL.

Código 3.4 – Registrador VHDL

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Register8bits is
  Port (
    clk : in std_logic;
    rst : in std_logic;
    carga : in std_logic;
    entrada : in std_logic_vector (7 downto 0);
    saida : out std_logic_vector (7 downto 0)
  );
end Register8bits;

architecture Behavioral of Register8bits is

  signal reg : std_logic_vector (7 downto 0) := (others => '0');

begin
  saida <= reg;

  Process(clk, rst)
  begin
    if rst = '1' then
      reg <= (others => '0');
    elsif clk'event and clk = '1' then
      if carga = '1' then
        reg <= entrada;
      end if;
    end if;
  end process;
end Behavioral;
```

3.4.3 MUX

O MUX é um multiplexador que faz uma escolha entre qual das duas entrada de 8 bits será disponibilizada na saída, a funcionalidade de suas portas é descrito abaixo:

- Sel: sinal de comando para definir qual entrada é disponibilizada na saída;
- Entrada 0: valor que estará disponível na saída quando Sel = 0;
- Entrada 1: valor que estará disponível na saída quando Sel = 1;
- Saída : valor selecionado;

Este módulo é instanciado em duas ocasiões, a função de cada uma delas é descrita a seguir:

- O MUX que precede o REM tem como função selecionar qual dado vai ser enviado à porta de entrada do REM (vindo do PC ou do RDM);
- O MUX que precede o RDM tem como função selecionar qual dado vai ser enviado à porta de entrada do RDM (vindo da Memória ou do Acumulador).

O Código 3.5 implementa o módulo MUX em Verilog.

Código 3.5 – MUX Verilog

```
module mux2_8(
        sel ,
        entrada0 ,
        entrada1 ,
        saida
    );
    input  sel ;
    input  [7:0] entrada0 ;
    input  [7:0] entrada1 ;
    output  [7:0] saida ;

    assign  saida  =  ( sel  ==  1'b0)   ?    entrada0  :
                                               entrada1   ;


endmodule
```

O Código 3.6 implementa o módulo MUX em VHDL.

Código 3.6 – MUX VHDL

```
library  IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity mux2_8 is
  Port (
    sel : in STD_LOGIC;
    entrada0 : in STD_LOGIC_VECTOR (7 downto 0);
    entrada1 : in STD_LOGIC_VECTOR (7 downto 0);
    saida : out STD_LOGIC_VECTOR (7 downto 0)
  );
end mux2_8;

architecture Behavioral of mux2_8 is

begin
  saida <= entrada0 when sel = '0' else entrada1 ;

end Behavioral ;
```

3.4.4 Decodificador

O Decodificador transforma a informação obtida de determinada maneira em outra forma de código que possam ser usadas pelos circuitos seguintes.

Este módulo tem uma implementação diferente para cada computador, pois sua função é decodificar a instrução armazenada em RI gerando um sinal para cada instrução possível do computador.

O Neander possui um conjunto de 11 instruções de acordo com a Tabela 3.1. Apenas os quatro bits mais significativos do RI são utilizados para gerar os 11 sinais de saída do decodificador, cada um destes sinais representa uma instrução.

O Ahmes possui um conjunto de 24 instruções de acordo com a Tabela 3.1. Os oito bits de RI são utilizados para gerar os 24 sinais de saída do decodificador, cada um destes sinais representa uma instrução.

O Código 3.7 implementa o módulo Decodificador para Ahmes em Verilog.

Código 3.7 – Decodificador Verilog

```
module demux8_decoder(
        entrada ,
        decodificado
```

```
4       );

6       input  [7:0] entrada ;
        output  [23:0] decodificado ;
8       reg  [23:0] decodificado ;

10      always  @( entrada  )
            begin
12              case(  entrada  )
            8'h00 :  decodificado  = 24'h000001;//NOP
14          8'h10 :  decodificado  = 24'h000002;//STA
            8'h20 :  decodificado  = 24'h000004;//LDA
16          8'h30 :  decodificado  = 24'h000008;//ADD
            8'h40 :  decodificado  = 24'h000010;//OR
18          8'h50 :  decodificado  = 24'h000020;//AND
            8'h60 :  decodificado  = 24'h000040;//NOT
20          8'h70 :  decodificado  = 24'h000080;//SUB
            8'h80 :  decodificado  = 24'h000100;//JMP
22          8'h90 :  decodificado  = 24'h000200;//JN
            8'h94 :  decodificado  = 24'h000400;//JP
24          8'h98 :  decodificado  = 24'h000800;//JV
            8'h9C :  decodificado  = 24'h001000;//JNV
26          8'hA0 :  decodificado  = 24'h002000;//JZ
            8'hA4 :  decodificado  = 24'h004000;//JNZ
28          8'hB0 :  decodificado  = 24'h008000;//JC
            8'hB4 :  decodificado  = 24'h010000;//JNC
30          8'hB8 :  decodificado  = 24'h020000;//JB
            8'hBC :  decodificado  = 24'h040000;//JNB
32          8'hE0 :  decodificado  = 24'h080000;//SHR
            8'hE1 :  decodificado  = 24'h100000;//SHL
34          8'hE2 :  decodificado  = 24'h200000;//ROR
            8'hE3 :  decodificado  = 24'h400000;//ROL
36          8'hF0 :  decodificado  = 24'h800000;//HLT
             default  :  decodificado  = 24'h000000;
38        endcase
            end
40
endmodule
```

O Código 3.8 implementa o módulo Decodificador para Ahmes em VHDL.

Código 3.8 – Decodificador VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity demux8_decoder is
  Port (
     entrada : in STD_LOGIC_VECTOR (7 downto 0);
     decodificado : out STD_LOGIC_VECTOR (23 downto 0)
  );
end demux8_decoder;

architecture Behavioral of demux8_decoder is

begin
 with entrada select
 decodificado <=     x"000001" when x"00", -- NOP
                   x"000002" when x"10",     -- STA
              x"000004" when x"20", -- LDA
                   x"000008" when x"30", --ADD
                   x"000010" when x"40",     --OR
                   x"000020" when x"50",     --AND
              x"000040" when x"60",     --NOT
                   x"000080" when x"70",     --SUB
              x"000100" when x"80",     --JMP
              x"000200" when x"90",     --JN
              x"000400" when x"94",     --JP
              x"000800" when x"98",     --JV
                   x"001000" when x"9C",     --JNV
              x"002000" when x"A0",     --JZ
                   x"004000" when x"A4", --JNZ
              x"008000" when x"B0", --JC
                   x"010000" when x"B4", --JNC
              x"020000" when x"B8", --JB
                   x"040000" when x"BC", --JNB
                   x"080000" when x"E0", --SHR
                   x"100000" when x"E1", --SHL
                   x"200000" when x"E2", --ROR
                   x"400000" when x"E3", --ROL
              x"800000" when x"F0", --HLT
                   x"000000" when others;
end Behavioral;
```

3.4.5 ULA

A ULA (Unidade Lógica Aritmética) é um bloco de lógica combinacional que realiza a operação aritmética definida pela instrução atual, a funcionalidade de suas portas é descrito abaixo:

- Op 1 : recebe dado do Acumulador;
- Op 2 : recebe dado do RDM;
- Controle : recebe os sinais de instrução decodificados pelo decodificador;
- Resultado : resultado da operação realizada;
- Estados : estados resultantes da operação realizada;

Este módulo tem uma implementação diferente para cada computador, pois o computador Ahmes possui mais operações aritméticas que o Neander.

O Código 3.9 implementa o módulo ULA para o computador Ahmes em Verilog.

Código 3.9 – ULA Verilog

```
module alu(
        op1,
        op2,
        controle ,
        estados ,
        resultado
    );
    input  [7:0] op1;
    input  [7:0] op2;
    input  [23:0] controle ;
    output  [4:0] estados ;
    output  [7:0] resultado ;

    wire  [7:0]  opRol, opRor, opShr, opShl, opNot, opAnd, opOr, opPass;
    wire  [8:0]  opAdd, opSub;
    wire stN, stZ, stC, stV, stB;
    reg  [7:0]  result ;

    assign  resultado  =  result ;

    assign  estados [0]  = stN;
    assign  estados [1]  = stZ;
    assign  estados [2]  = stV;
```

```
24  assign  estados [3]  = stC;
    assign  estados [4]  = stB;
26


28  assign  opNot =  ~( op1);
    assign  opAnd = ( op1 & op2 );
30  assign  opOr = ( op1 | op2 );
    assign  opAdd = ( op1 + op2 );
32  assign  opPass = op2;
    assign  opSub = (1'b0 & op1) − (1'b0 & op2);
34  assign  opShr =  1'b0 & op1 [7:1];
    assign  opShl =  op1 [6:0]  & 1'b0 ;
36  assign  opRor =  op1 [0]  & op1 [7:1];
    assign  opRol =  op1 [6:0]  & op1 [7];
38

40

42
    assign  stN =  result [7];
44  assign  stZ = ( result  == 8'b00000000) ? 1'b1:  1'b0;
    assign  stB = ( controle  == 23'h000080) ? opSub[8] :  1'b0;
46
    assign  stC = ( controle  == 23'h000008) ? opAdd[8] :
48            ( controle  == 23'h080000) ? op1 [0]  :
              ( controle  == 23'h100000) ? op1 [7]  :
50            ( controle  == 23'h200000) ? op1 [0]  :
              ( controle  == 23'h400000) ? op1 [7]  :
52            1'b0;

54

56  assign  stV = ( controle  == 23'h000008) ?  (op1[7]  & op2[7] & ~result [7])  | (~op1[7]  & ~op2
    [1] & result [7])   :
              ( controle  == 23'h000080) ? (op1[7]  & ~op2[7] & ~result [7])  | (~op1[7]  & op2[1] &
     result [7])   :
58   1'b0;

60

    always @( controle ,  opRol,  opRor,  opShr,  opShl  ,opSub,  opPass,  opAdd, opOr,opAnd, opNot )
62      begin
```

```
            case(  controle  )
64          23'h000004 :  result  = opPass;
            23'h000008 :  result  = opAdd[7:0];
66          23'h000010 :  result  = opOr;
            23'h000020 :  result  = opAnd;
68          23'h000040 :  result  = opNot;
            23'h000080 :  result  = opSub[7:0];
70          23'h080000 :  result  = opShr;
            23'h100000 :  result  = opShl;
72          23'h200000 :  result  = opRor;
            23'h400000 :  result  = opRol;
74      default  :  result  = 8'b11111111;
            endcase
76        end

78 endmodule
```

O Código 3.10 implementa o módulo ULA para o computador Ahmes em VHDL.

Código 3.10 – ULA VHDL

```
   library  IEEE;
2  USE IEEE.STD_LOGIC_1164.ALL;
   USE IEEE.STD_LOGIC_SIGNED.ALL;
4  USE IEEE.STD_LOGIC_ARITH.ALL;

6  entity  alu  is
     Port  (
8       op1   :  in   STD_LOGIC_VECTOR (7 downto 0);
        op2  :  in   STD_LOGIC_VECTOR (7 downto 0);
10      controle  :  in   STD_LOGIC_VECTOR (23 downto 0);
        estados  :  out   std_logic_vector  (4  downto 0);
12      output   :  out   STD_LOGIC_VECTOR (7 downto 0));
   end  alu;
14
   architecture   Behavioral  of  alu  is
16
   signal  opRol, opRor, opShr, opShl, opNot, opAnd, opOr, opPass,  result  :  std_logic_vector  (7
        downto 0);
18 signal  opAdd, opSub :  std_logic_vector (8  downto 0);
   signal  carry ,  overflow :  std_logic ;
20 begin
```

```
22  output <= result ;

24  opNot <= not op1;
    opAnd <= op1 and op2;
26  opOr <= op1 or op2;
    opAdd <= ("0"&op1) + ("0"&op2);
28  opPass <= op2;
    opSub <= ("0"&op1) - ("0"&op2);
30  opShr <= '0' & op1(7 downto 1);
    opShl <= op1(6 downto 0) & '0' ;
32  opRor <= op1(0) & op1(7 downto 1);
    opRol <= op1(6 downto 0) & op1(7);
34
    overflow <= (op1(7) and op2(7) and not result (7)) or (not op1(7) and not op2(7) and result (7))
            when controle = x"000008" -- ADD
36              else (op1(7) and not op2(7) and not result (7)) or (not op1(7) and op2(7) and result (7)
        ) when controle = x"000080" --SUB
                else '0';
38
    estados (0) <= result (7); -- N
40  estados (1) <= '1' when result = "00000000" else '0'; -- Z
    estados (2) <= overflow; -- V
42  estados (3) <= carry; --C
    estados (4) <= opSub(8) when controle = x"000080" else '0'; -- B
44
    with controle select
46  carry <= opAdd(8) when x"000008", --ADD
          op1(0) when x"080000", --SHR
48        op1(7) when x"100000", --SHL
          op1(0) when x"200000", --ROR
50        op1(7) when x"400000", --ROL
          '0' when others;
52
    with controle select
54  result <=     opPass when x"000004", -- LDA
            opAdd(7 downto 0) when x"000008", -- ADD
56          opOr when x"000010", -- OR
            opAnd when x"000020", -- AND
58          opNot when x"000040", -- NOT
            opSub(7 downto 0) when x"000080", -- SUB
60          opShr when x"080000", -- SHR
```

```
        opShl  when x"100000", -- SHL
62      opRor  when x"200000", -- ROR
        opRol  when x"400000", --ROL
64      ( others  => '0')  when others;

66


68 end Behavioral;
```

3.5 Unidades de Controle

A unidade de controle pode ser implementada de várias maneiras diferentes. Neste trabalho serão descritas três formas: uma implementação utilizando uma máquina de estados e as outras duas com temporizador.

Sendo a UC (Unidade de Controle) o módulo que define a ordem e o tempo em que os outros módulos da arquitetura serão acionados, é necessário definir o tipo de memória que será utilizada para incluir suas restrições de tempo no controle de fluxo da UC.

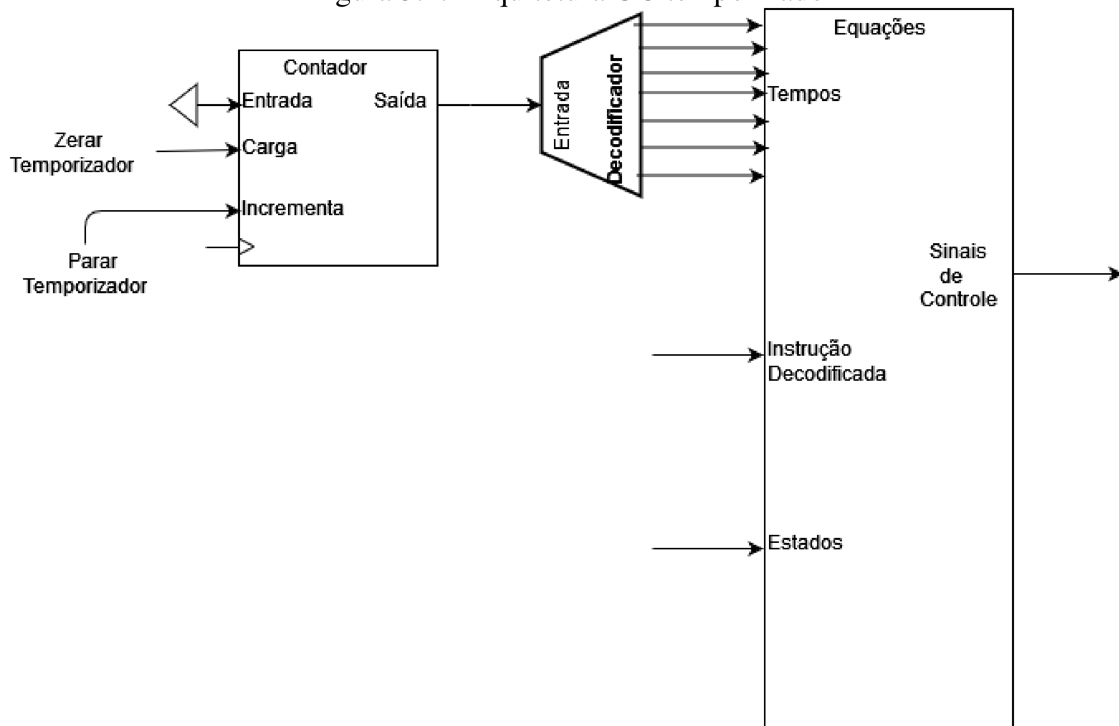
Para cada solução de UC proposta será desenvolvida uma versão que prevê o uso de uma memória ROM construída de acordo com a especificada na Seção 2.1. E outra versão que prevê o uso de uma memória RAM de acordo com a especificada na Seção 2.2.

3.5.1 Unidades de Controle para ROM

3.5.1.1 Temporizador

Esta solução se constitui em 3 blocos. Um contador com as mesmas funcionalidades do PC (Program Counter) porém menor, de três bits contando de 0 a 7. O decodificador do contador com entrada de 3 bits e saída de 8 bits, onde em determinado tempo apenas um bit dos 8 está ativo. O bloco de equações contém a lógica combinacional necessária para gerar os sinais de controle a partir do contador decodificado, sinais de estado e instrução a ser executada. A Figura 3.2 ilustra essa arquitetura.

Figura 3.2: Arquitetura UC temporizador



Dentro desta implementação da Unidade de Controle existem duas variações: a primeira, após gerar os sinais necessários para executar a instrução o computador passa o restante do tempo inativo até que o contador passe por todos os tempos e se reinicie; a segunda é uma otimização da primeira, quando uma instrução termina de ser executada um sinal de controle reinicia o contador, reduzindo assim a quantidade de ciclos de relógio necessários para executar algumas instruções.

A lógica combinacional do bloco de equações é construída a partir da Tabela 3.2. Os sinais de controle necessários para controlar arquitetura dos computadores Neander e Ahmes são os mesmos e são citadas abaixo juntamente com suas respectivas equações:

- $Carrega_REM = T0 \vee (T3 \wedge (STA \vee ULA_dado \vee Desvio_sim)) \vee (T5 \wedge (STA \vee ULA_dado))$
- $Mux_REM = T5 \wedge (STA \vee ULA_dado)$
- $Carrega_RDM = T1 \vee (T4 \wedge (STA \vee ULA_dado \vee Desvio_sim)) \vee (T6 \wedge (STA \vee ULA_dado))$
- $Mux_RDM = T6 \wedge STA$
- $Incrementa_PC = T1 \vee (T4 \wedge (STA \vee ULA_dado)) \vee (T3 \wedge Desvio_nao)$
- $Carrega_PC = T5 \wedge Desvio_sim$

- $Carrega_RI = T2$
- $Escreve_Memoria = T7 \wedge STA$
- $Carrega_AC = (T3 \wedge opACC) \vee (T7 \wedge ULA_dado)$
- $Parar_Temporizador = \neg opHLT$
- $Zerar_Temporizador = (T3 \wedge (opNOP \vee opACC \vee Desvio_nao)) \vee load_pc \vee (T6 \wedge ULA_dado)$

Observar que para a implementação da unidade de controle mais simples, ou seja, sem o controle Zerar_Temporizador basta apenas não implementar sua equação ou deixar este sinal de controle desativado (= 0).

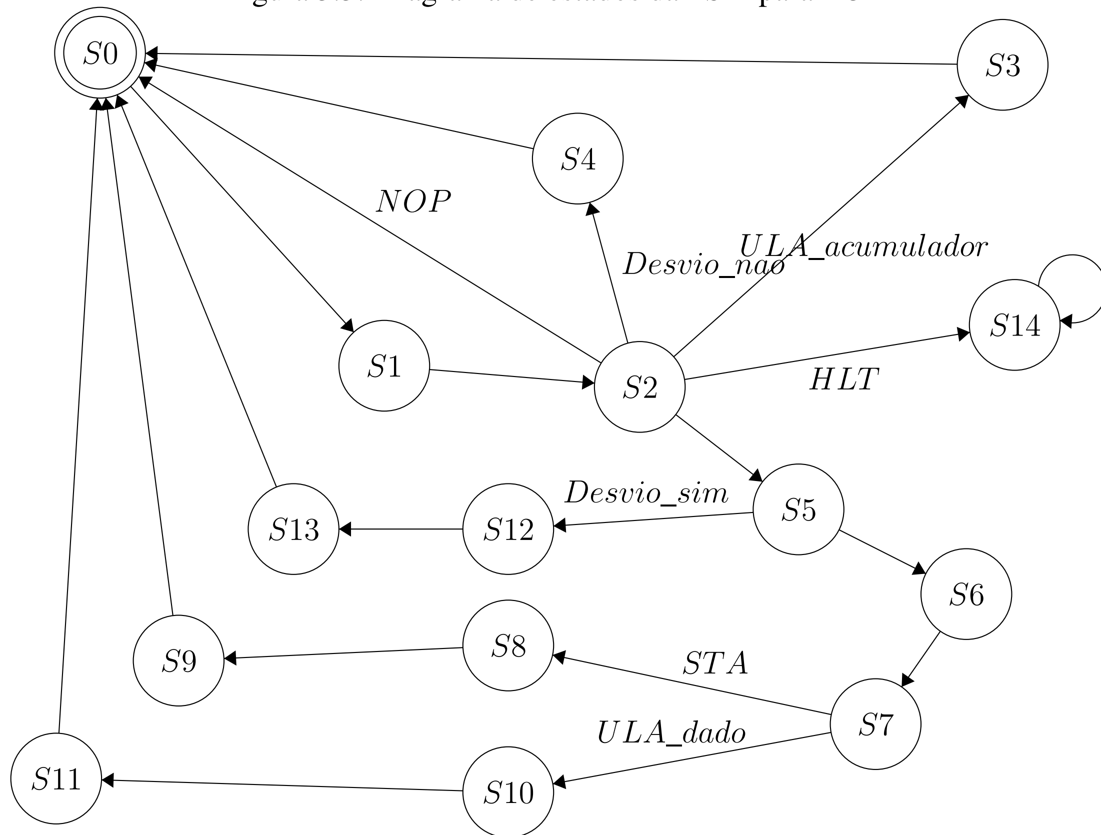
3.5.1.2 FSM

A máquina de estados ilustrada na Figura 3.5.1.2 é deduzida da Tabela 3.2. Nesta FSM Mealy os fluxos de execução das instruções são agrupados nos estados enquanto os sinais de controle que devem ser gerados são os mesmos, e se dividem quando diferem, mas sempre retornando ao estado inicial S0, exceto na execução da instrução HLT em que a máquina de estados entra em loop.

Abaixo são listados os fluxos de execução de cada grupo de instruções:

- STA : $S0 \rightarrow S1 \rightarrow S2 \rightarrow S5 \rightarrow S6 \rightarrow S7 \rightarrow S8 \rightarrow S9 \rightarrow S0$
- ULA dado: $S0 \rightarrow S1 \rightarrow S2 \rightarrow S5 \rightarrow S6 \rightarrow S7 \rightarrow S10 \rightarrow S11 \rightarrow S0$
- Desvio sim : $S0 \rightarrow S1 \rightarrow S2 \rightarrow S5 \rightarrow S12 \rightarrow S13 \rightarrow S0$
- ULA acumulador: $S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S0$
- Desvio não : $S0 \rightarrow S1 \rightarrow S2 \rightarrow S4 \rightarrow S0$
- NOP : $S0 \rightarrow S1 \rightarrow S2 \rightarrow S0$
- HLT : $S0 \rightarrow S1 \rightarrow S2 \rightarrow S14 \rightarrow S14 \rightarrow S14...$

Figura 3.3: Diagrama de estados da FSM para ROM



A Tabela 3.3 explica o que ocorre em cada estado da máquina.

Tabela 3.3: Estados e Ações

Estado	Ação
S0	REM ← PC
S1	PC ← PC + 1 RDM ← MEM
S2	RI ← RDM
S3	AC ← ULA
S4	PC ← PC + 1
S5	REM ← PC
S6	PC ← PC + 1 RDM ← MEM
S7	REM ← RDM
S8	RMD ← AC
S9	MEM ← RDM
S10	RDM ← MEM
S11	AC ← ULA
S12	RDM ← MEM
S13	PC ← RDM
S14	HALT

3.5.2 Unidades de Controle para RAM

3.5.2.1 Temporizador

O temporizador de 3 bits precisa ser estendido para 4 bits pois as operações de escrita e leitura precisam de um ciclo de relógio a mais para serem executada na memória RAM e em consequência desta mudança o decodificador deve ser adaptado ao novo temporizador.

Ao reescrever as equações adicionando tempo nas operações necessárias, a quantidade de ciclos de relógio utilizados aumenta de 8 para 11.

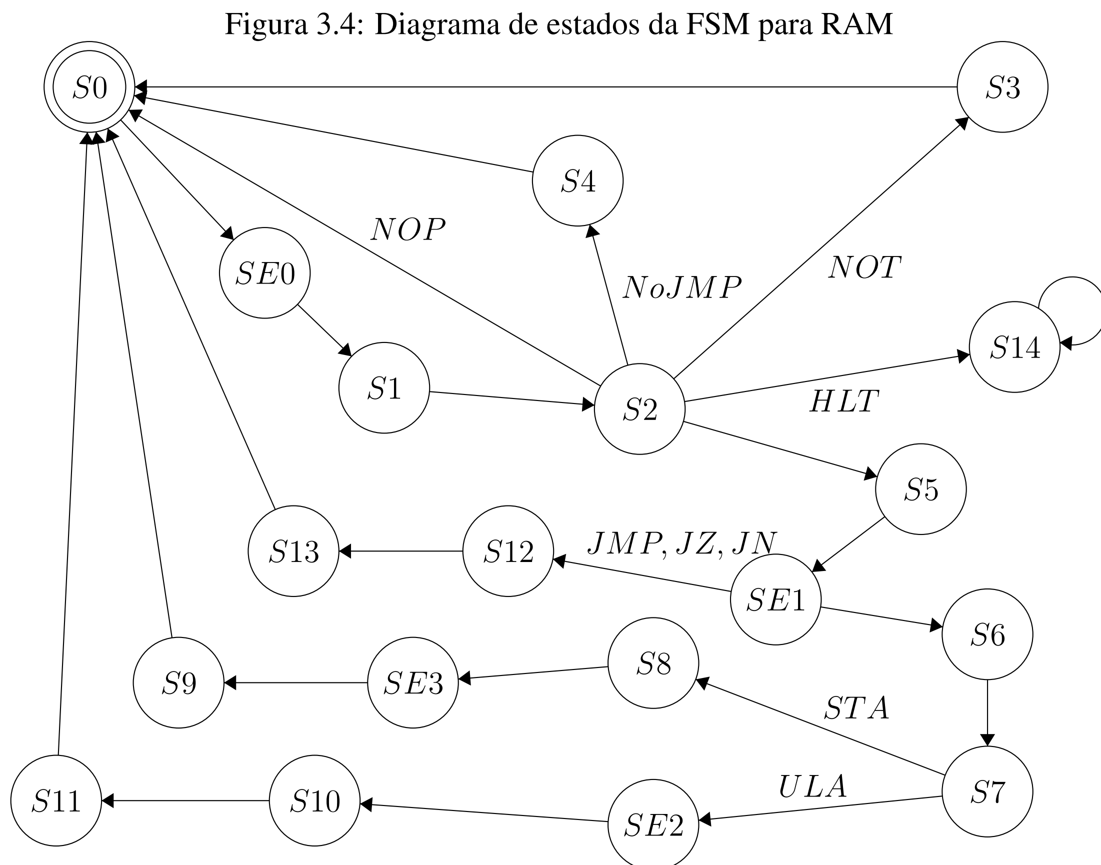
- $CarregaREM = T0 \vee (T4 \wedge (STA \vee ULA_dado \vee Desvio_sim)) \vee (T7 \wedge STA)$
- $MuxREM = T7 \wedge (STA \vee ULA_dado)$
- $CarregaRDM = T2 \vee (T4 \wedge (STA \vee ULA_dado \vee Desvio_sim)) \vee (T7 \wedge STA)$

- $MuxRDM = T7 \land STA$
- $IncrementaPC = T1 \lor (T5 \land (STA \lor ULA_dado \lor Desvio_sim)) \lor (T4 \land Desvio_nao)$
- $CarregaPC = T7 \land Desvio_sim$
- $CarregaRI = T3$
- $EscreveMemoria = (T9 \lor T10) \land STA$
- $CarregaAC = (T4 \land opACC) \lor (T10 \land ULA_dado)$
- $PararTemporizador = \neg opHLT$
- $ZerarTemporizador = (T5 \land (opNOP \lor opACC \lor Desvio_nao)) \lor load_pc \lor T10$

3.5.2.2 FSM

A FSM necessita de estados adicionais para acomodar as especificações da memória RAM. Na Figura 3.5.2.2 o novo grafo da máquina de estados pode ser observado.

Figura 3.4: Diagrama de estados da FSM para RAM



E a Tabela 3.4 demonstrando as ações executadas em cada estado.

Tabela 3.4: Estados e Ações

Estado	Ação
S0	REM $\leftarrow$ PC
SE0	
S1	PC $\leftarrow$ PC + 1 RDM $\leftarrow$ MEM
S2	RI $\leftarrow$ RDM
S3	AC $\leftarrow$ ULA
S4	PC $\leftarrow$ PC + 1
S5	REM $\leftarrow$ PC
SE1	
S6	PC $\leftarrow$ PC + 1 RDM $\leftarrow$ MEM
S7	REM $\leftarrow$ RDM
SE2	
S8	RMD $\leftarrow$ AC
SE3	MEM $\leftarrow$ RDM
S9	MEM $\leftarrow$ RDM
S10	RDM $\leftarrow$ MEM
S11	AC $\leftarrow$ ULA
S12	RDM $\leftarrow$ MEM
S13	PC $\leftarrow$ RDM
S14	HALT

3.6 Simulação Testbench

As ferramentas utilizadas para compilar e simular foram respectivamente Quartus II 13.1 Web Edition e Modelsim ALTERA STARTER EDITION 10.4. Para realizar o teste da arquitetura foram desenvolvidos dois programas, um utilizando o simulador Neander e outro utilizando o Ahmes. Ambos com o objetivo de testar todas as instruções de cada um dos computadores. A seguir é apresentado o arquivo .txt que pode ser exportado dos simuladores Ahmes e Neander contendo a descrição do programa desenvolvido dentro desses ambiante de simulação.

Segmento de programa de 0 a 127 (as posições de memória não descritas aqui contém 0, ou seja instrução NOP) e segmento de dados de 128 a 255 (as posições de memória não descritas aqui contém 0):

Código 3.11 – Testbench.txt

```
0    32 128   LDA 128 -- Carrega ender 128, AC = 10
2    48 129   ADD 129  -- 10 + 1, AC = 11
4   160   12   JZ  12 -- Se zero desvia  para  HLT
6   144   12   JN  12 -- Se negativo desvia  para  HLT
8    64 130   OR  130 -- 11 or 143,  AC = 143
10   144  13   JN  13 -- se nao negativo  segue  para  HLT
12   240       HLT  -- Termina execucao
13    96       NOT  -- not 0, AC = 112
14    80 130   AND 130  -- 112 and 143, AC = 0
16   160  19   JZ  19 -- se nao zero  segue  para  HLT
18   240       HLT  -- Termina execucao
19    96       NOT  -- not 0, AC = 255
20    16 131   STA 131  -- guarda 255 em ender 131
22   128  25   JMP 25 -- desvia para ender  25
24   240       HLT  -- Termina execucao
25    80 132   AND 132  -- 255 and 240, AC = 240
27    16  32   STA 32 -- guarda 240(HLT) em ender 32
29     0       NOP
30     0       NOP
31     0       NOP
32     0       NOP
128   10
129    1
130  143
131    0
132  240
```

Os sinais necessários para estimular a arquitetura são um período de relógio constante e um pulso de reset para o setup inicial da arquitetura. Estes estímulos são gerados no arquivo de testbench 3.12.

Código 3.12 – Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY Ahmes_RAM1_vhd_tst IS
```

```
END Ahmes_RAM1_vhd_tst;
ARCHITECTURE Ahmes_RAM1_arch OF Ahmes_RAM1_vhd_tst IS
-- constants
CONSTANT clock_period : time := 10 ns;
-- signals
SIGNAL AC : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL CLK : STD_LOGIC;
SIGNAL CTRL : STD_LOGIC_VECTOR(10 DOWNTO 0);
SIGNAL MEM : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL PC : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL RDM : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL \REM\ : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL RI : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL clock : STD_LOGIC := '0' ;
SIGNAL rst :  STD_LOGIC;
COMPONENT Ahmes_RAM1
  PORT (
  AC :  OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  CLK : OUT STD_LOGIC;
  CTRL : OUT STD_LOGIC_VECTOR(10 DOWNTO 0);
  MEM : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  PC :  OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  RDM : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  \REM\ : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  RI  :  OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
  clock  :  IN STD_LOGIC;
   rst   :  IN STD_LOGIC
  );
END COMPONENT;
BEGIN
  i1  :  Ahmes_RAM1
  PORT MAP (
-- list  connections  between master  ports  and  signals
  AC => AC,
  CLK => CLK,
  CTRL => CTRL,
  MEM => MEM,
  PC => PC,
  RDM => RDM,
  \REM\ => \REM\,
  RI => RI,
```

```
    clock  => clock,
    rst  => rst
    );



clock <= not clock after clock_period/2;

 init : PROCESS
-- variable declarations
BEGIN
        -- code that executes only once
    rst <= '1';
    wait for 30 ns;
    rst <= '0';
WAIT;
END PROCESS init;

END Ahmes_RAM1_arch;
```

Ao final da simulação é esperada a execução de 3 instruções NOP e depois HLT, caso haja algum erro nas instruções implementadas

3.7 Empacotador de Memória

Para criar uma memória ROM e gerar os arquivos necessários para inicializar a memória RAM criada na Subseção 2.2.1 foi desenvolvido um script que recebe como entrada um arquivo exportado dos simuladores Ahmes e Neander e retorna esses arquivos.

Este script no Apêndice 7.3 foi desenvolvido na linguagem Python e recebe como parâmetro o programa desenvolvido no simulador (o arquivo exportado do simulador deve ser um .TXT e estar como os dados descritos em números decimais), os arquivos retornados são:

- MemoriaROM.vhd ;
- MemoriaRAM.mif ;

Um exemplo de arquivo usado como parâmetro já exposto neste trabalho no Código 3.11, é processado pelo script através da linha de comando:

```
$ ./memparcer Testbench.txt
```

Gerando os arquivos MemoriaROM.vhd descrito no Código 2.1, MemoriaRAM.mif.

4 VGA (VIDEO GRAPHICS ARRAY)

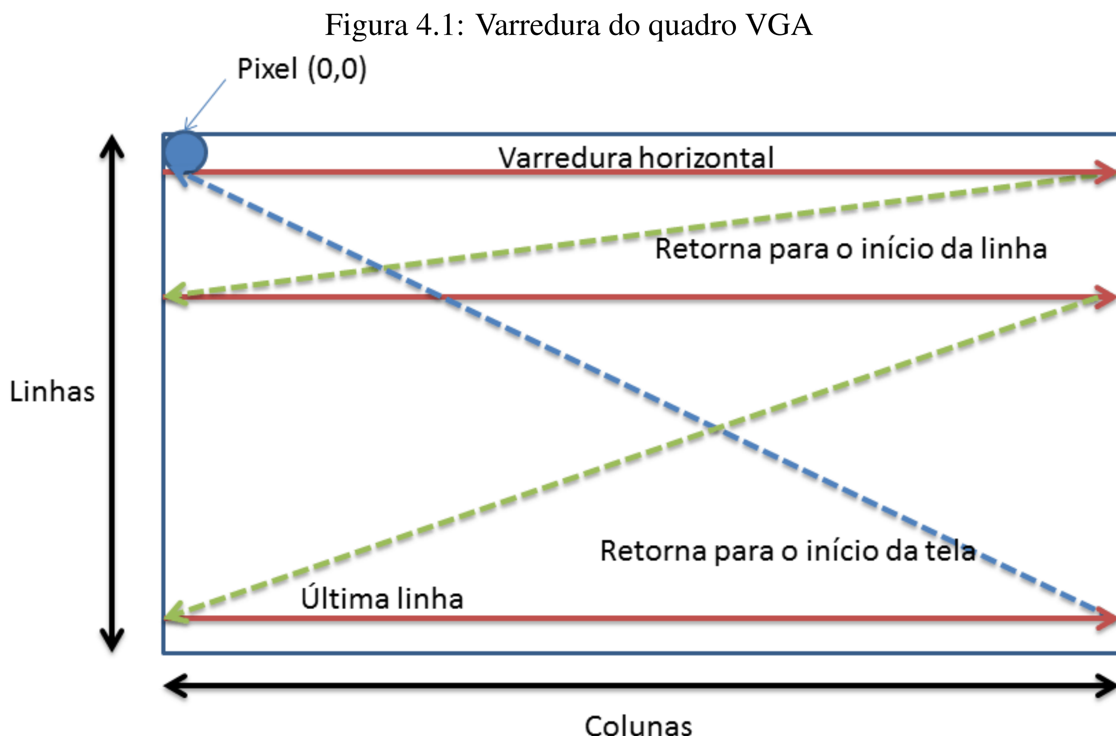
O VGA (Video Graphics Array) é um padrão de gráficos de computadores introduzido em 1987 pela IBM.

Neste capítulo serão descritos o desenvolvimento de um controlador para VGA assim como a análise das especificações padrão para seu funcionamento e o mapeamento da tela para possibilitar a exibição de caracteres ASCII.

4.1 Especificações VGA

Para entender a solução proposta neste trabalho é necessário primeiro estudar a teoria básica sobre o VGA. O formato de vídeo VGA é um fluxo de quadros: cada quadro é feito de uma série de linhas horizontais, e cada linha é feita de uma série de pixels. As linhas de cada quadro são transmitidas de cima para baixo, e os pixels de cada linha da esquerda para direita. Ilustrado na Figura 4.1.

Figura 4.1: Varredura do quadro VGA



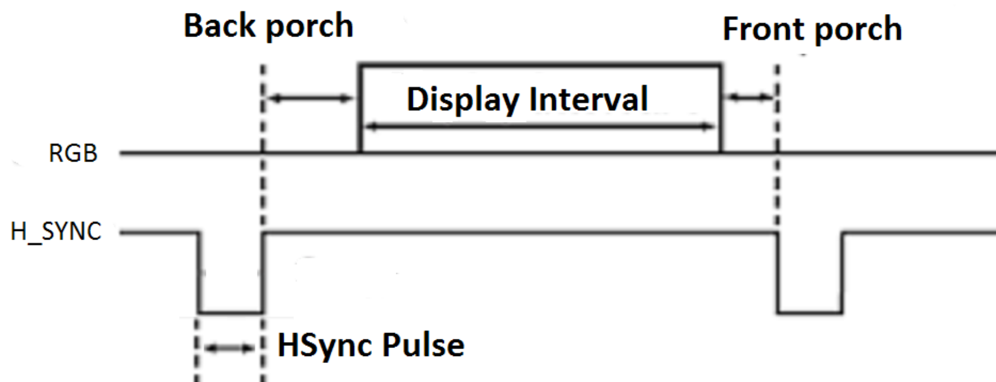
O sistema de cores adotado pelo VGA é o RGB, utilizando 3 sinais analógicos para representar as cores: RED, GREEN e BLUE. O padrão VGA utiliza 2 sinais para sincronização de vídeo: sincronização horizontal (H_SYNC) especifica o tempo neces-

sário para transmitir uma linha; sincronização vertical (V_SYNC) especifica o tempo necessário para transmitir todas as linhas(um quadro).

Para o correto funcionamento do VGA os sinais de sincronização devem respeitar algumas restrições físicas do monitor. As restrições descritas abaixo e ilustradas na Figura 4.2 são referentes ao sinal H_SYNC, estas restrições de tempo também podem ser medidas em pixels (contando um pixel por ciclo de relógio):

- Front porch: Período de tempo que antecede o Sync Pulse, durante este tempo o RGB deve estar desligado (0 V).
- HSync Pulse: Pulso reverso com intervalo de tempo definido, indica o final de uma linha e o inicio de outra.
- Back porch: Período de tempo que precede o Sync Pulse, durante este tempo o RGB deve estar desligado (0 V).
- Display Interval: Período de tempo em que os dados (RGB) são mostrados na tela.

Figura 4.2: Restrições de tempo H_SYNC

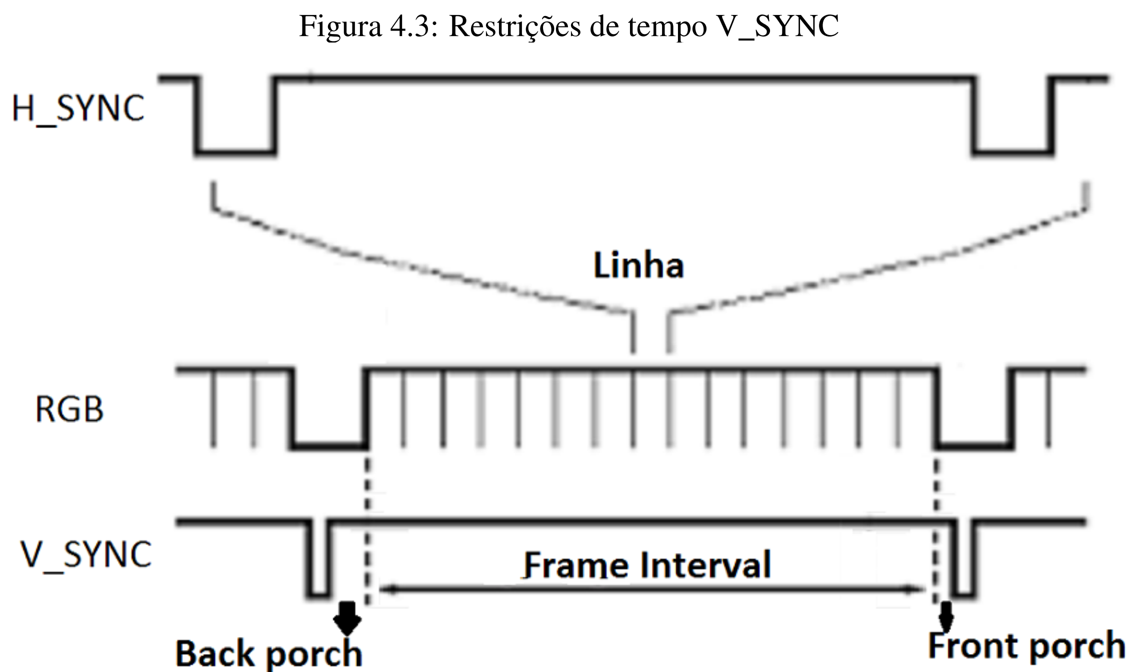


As restrições descritas abaixo e ilustradas na Figura 4.3 referem-se ao sinal V_SYNC, os períodos de tempo do sinal V_SYNC são diretamente dependentes de H_SYNC podendo também serem medidos em número de linhas:

- Front porch: Período de tempo que antecede o VSync Pulse, durante este tempo o RGB deve estar desligado (0 V).
- VSync Pulse: Pulso reverso com intervalo de tempo definido, indica o final de um quadro e o início de outro.
- Back porch: Período de tempo que precede o VSync Pulse, durante este tempo o RGB deve estar desligado (0 V)

- Frame Interval: Período de tempo em que todas as linhas são exibidas na tela formando um quadro completo.

Figura 4.3: Restrições de tempo V_SYNC



A resolução da tela é determinada pela taxa de pixel (relógio disponível no controlador). Quanto maior a frequência do relógio mais pixels poderão ser enviados ao monitor aumentando assim a resolução. Por ser um padrão amplamente utilizado o VGA possui uma grande quantidade de resoluções de tela já definidas com suas restrições de tempo para os sinais de controle. O VESA (VESA. . . , 2016) é utilizado pelos fabricantes como padrão, e as especificações sobre quais resoluções são suportadas por determinado monitor estão no manual do produto.

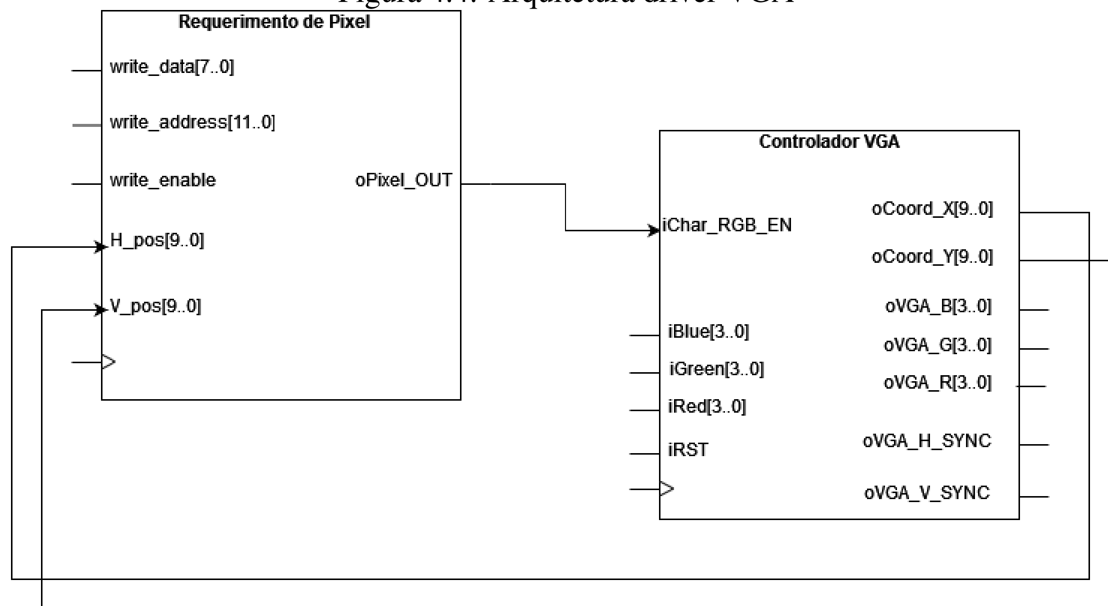
Tabela 4.1: Resoluções e suas Especificações

Format	Pixel Clock (MHz)	Horizontal(in pixels)				Vertical (in Lines)			
		Active Video	Front Porch	Sync Pulse	Back Porch	Active Video	Front Porch	Sync Pulse	Back Porch
640x480, 60Hz	25.175	640	16	96	48	480	11	2	31
640x480, 72Hz	31.500	640	24	40	128	480	9	3	28
640x480, 75Hz	31.500	640	16	96	48	480	11	2	32
640x480, 85Hz	36.000	640	32	48	112	480	1	3	25
800x600, 56Hz	38.100	800	32	128	128	600	1	4	14
800x600, 60Hz	40.000	800	40	128	88	600	1	4	23
800x600, 72Hz	50.000	800	56	120	64	600	37	6	23
800x600, 75Hz	49.500	800	16	80	160	600	1	2	21
800x600, 85Hz	56.250	800	32	64	152	600	1	3	27
1024x768, 60Hz	65.000	1024	24	136	160	768	3	6	29
1024x768, 70Hz	75.000	1024	24	136	144	768	3	6	29
1024x768, 75Hz	78.750	1024	16	96	176	768	1	3	28
1024x768, 85Hz	94.500	1024	48	96	208	768	1	3	36

4.2 Arquitetura Driver VGA

Seguindo as especificações descritas no manual da placa DE0, foi adotada a resolução sugerida, 640x480 pixels (480 linhas de 640 pixels cada uma). Este driver é dividido em dois módulos, controle VGA e Gerador de pixels. A Figura 4.4 ilustra a arquitetura desenvolvida.

Figura 4.4: Arquitetura driver VGA



4.3 Controle VGA

Este módulo define a resolução da tela. Gera os sinais necessários para sincronizar a imagem mostrada no monitor(H_SYNC e V_SYNK) e requerimento de valor do pixel para definir o valor RGB de cada pixel da tela.

4.3.1 Sinais de Sincronização

A fim de gerar os sinais de sincronização o controlador implementa contadores para monitorar as restrições de tempo para V_SYNC e H_SYNC, estas especificações seguem o padrão VESA e estão descritos na Tabela 4.1.

O sinal H_SYNC fica desativado ('0' lógico) quando o valor do contador horizontal está entre 0 e 95 (período de H_SYNC_PULSE) e ativo ('1' lógico) entre de 96 a 799. O contador horizontal zera seu valor ao chegar em 799 que equivale a $H_SYNC_PULSE + H_BACK_PORCH + DISPLAY_INTERVAL + H_FRONT_PORCH$.

O contador vertical é incrementado toda vez que o contador horizontal é reiniciado, indicando o início de uma nova linha. O sinal V_SYNC fica desativado ('0' lógico) quando o valor do contador vertical está entre 0 e 2 (período de V_SYNC_CYC) e ativo ('1' lógico) entre de 3 a 524. O contador vertical zera seu valor ao chegar em 524 que

equivale a $V_SYNC_PULSE + V_BACK_PORCH + FRAME_INTERVAL + V_FRONT_PORCH$.

4.3.2 Requerimento de Pixel

O Requerimento de valor do pixel é feito de acordo com a posição de coordenada (X,Y) em que ele aparece na tela, sendo X um valor entre 0 e 639 e Y um valor entre 0 e 479. Estas coordenadas são enviadas ao Gerador de Pixels e este retorna o valor do pixel com um atraso de 2 ciclos de relógio. Os registradores coord_X e coord_Y são utilizados quando o controlador estiver no intervalo de área visível. Isto ocorre enquanto:

- Registrador horizontal entre 142 ($H_SYNC_PULSE + H_BACK_PORCH - 2$) e 784 ($H_SYNC_PULSE + H_BACK_PORCH + DISPLAY_INTERVAL - 2$);
- Registrador vertical entre 35 ($V_SYNC_PULSE + V_BACK_PORCH$) e 515 ($V_SYNC_PULSE + V_BACK_PORCH + FRAME_INTERVAL$)

Devido ao tempo de resposta do módulo Gerador de Pixels o controlador precisa ajustar o tempo de requisição dos pixels das linhas para que o valor retornado corresponda ao pixel atual da área visível. Este ajuste é feito ao fazer a requisição do pixel dois ciclos de relógio antes do valor real de sua posição.

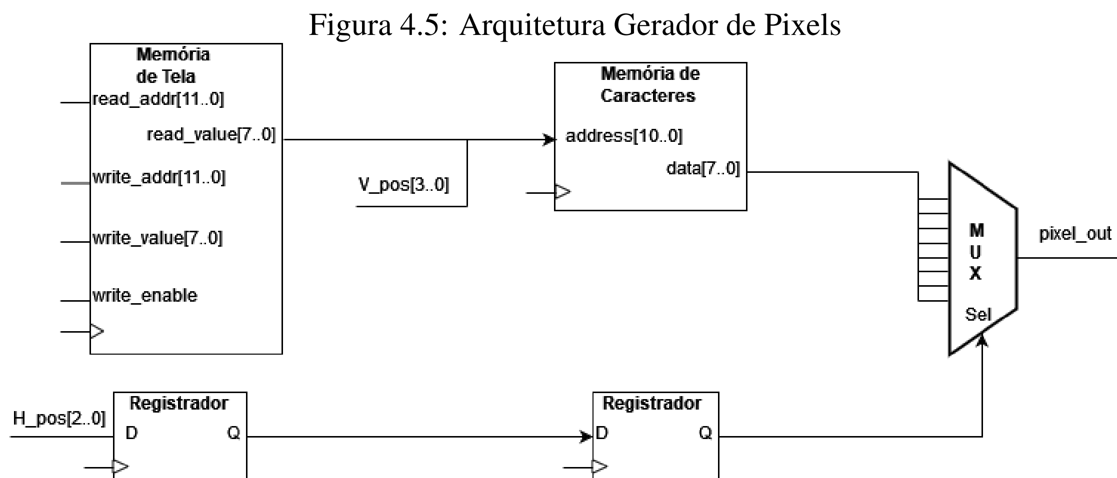
O Valor de cada cor RGB(RED, GREEN, BLUE) é dado por um valor de 4 bits. Pelo fato da tela a ser específica para texto, o fundo é definido como preto (R = 0,G = 0, B = 0) e o valor do pixel de caractere é definido com como branco (R = 16, G = 16, B = 16). De acordo com a resposta recebida do Gerador de Pixels a cor preta ou a branca é enviada para a tela. Os valores de RGB são enviados para a tela enquanto:

- Registrador horizontal entre 142 ($H_SYNC_PULSE + H_BACK_PORCH$) e 784 ($H_SYNC_PULSE + H_BACK_PORCH + DISPLAY_INTERVAL$);
- Registrador vertical entre 35 ($V_SYNC_PULSE + V_BACK_PORCH$) e 515 ($V_SYNC_PULSE + V_BACK_PORCH + FRAME_INTERVAL$)

4.4 Gerador de Pixels

Este módulo tem a função de armazenar a memória da tela e disponibilizar o valor do pixel requerido pelo módulo. A arquitetura do Gerador de Pixels, é composta por uma memória de tela, uma memória de caracteres, 2 flip-flops para buferização e um multiplexador para escolha do pixel requerido. Essa arquitetura está ilustrada na Figura 4.5

Figura 4.5: Arquitetura Gerador de Pixels



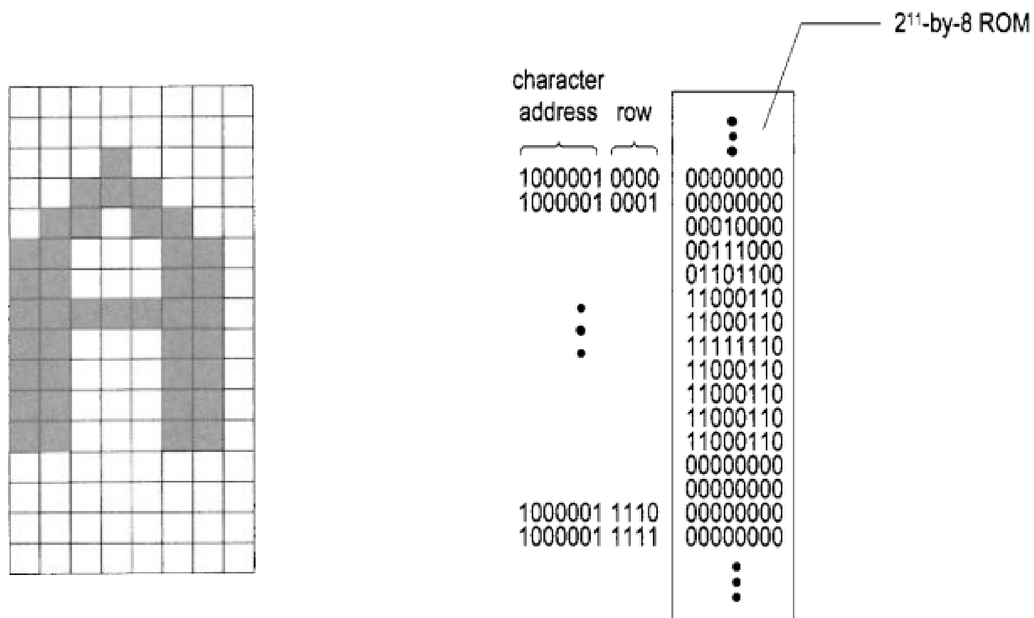
4.4.1 Memória de Caracteres

A memória de caracteres é uma memória ROM contendo os desenhos caracteres. Está memória tem $2^{11} = 2048$ endereços e cada palavra da memória tem 8 bits.

Cada caractere é desenhado em uma área de 8X16 pixels, a Figura 4.6 exemplifica o desenho de um caractere e sua posição na memória. Pelo fato de o caractere ocupar 16 linhas de pixel da tela, na memória ele ocupara 16 endereços de memória. Sendo assim, a memória de 2048 endereços pode mapear até 128 caracteres. Com o objetivo de mapear os caracteres de acordo com os valores ASCII, a entrada de endereço da memória que tem 11 bits é utilizada da seguinte forma:

- Bits de endereço de 11 a 4: Endereço do caractere conforme valor da Tabela ASCII;
- Bits de endereço de 3 a 0: acesso à linha de pixels do caractere;

Figura 4.6: Exemplo de desenho de caractere((CHU, 2008))



4.4.2 Memória de Tela

A Memória de Tela é uma memória RAM contendo o mapa da tela em caracteres ASCII. Está memória tem $2^{12} = 4096$ endereços e cada palavra da memória tem 8 bits.

Esta memória tem como objetivo mapear a área da tela por caractere, cada palavra da memória contém o valor de um caractere ASCII. Sabendo que a resolução da tela é de 640X480 e que cada caractere tem a resolução de 8X16 pixels são realizados os cálculos para quantas linhas de caracteres cabem na tela e quantos caracteres cabem em cada linha:

- Linhas por tela: $\frac{480}{16} = 30$;
- Caracteres por linha: $\frac{640}{8} = 80$;

Sendo assim, seria necessária uma memória de 2400 (80X30) endereços. Porém, a fim de facilitar o acesso, o mapeamento da memória é projetado estendendo o numero de linhas e o número de caracteres por linha para números em potência de 2, gerando uma memória com 4096 (128X32) endereços.

Ressalta-se que, construir a memória deste modo acarreta em desperdício de memória, pois o controlador não acessa todos os endereços da mesma. A entrada de endereço da memória é composta por 12 bits e é utilizada da seguinte forma:

- Bits de endereço de 11 a 7: Endereça as linhas da tela;

- Bits de endereço de 6 a 0: Endereça os caracteres dentro da linha;

Esta memória tem três entradas para controle de escrita (endereço de escrita, habilitar escrita e valor a ser escrito), podendo assim modificar a informação mostrada na tela.

4.4.3 Funcionamento

As entradas deste módulo, Gerador de Pixels, são listadas e descritas a seguir:

- H_pos : tamanho 10 bits, posição horizontal do pixel a ser mostrado na tela;
- V_pos : tamanho 10 bits, posição vertical do pixel a ser mostrado na tela;
- Write_Address: tamanho 12 bits, posição da tela em que o caractere deve ser armazenado.
- Write_Enable: tamanho 1 bit, sinal que habilita escrita na memória de caracteres
- Write_Data: tamanho 8 bits, valor de um caractere ASCII;

A saída deste módulo, Gerador de Pixels, é:

- Pixel_Valor: tamanho 1 bit, indica ao controlador se o pixel deve ser mostrado ou não.

Para fazer o acesso ao valor do pixel a ser retornado na saída, os valores de posição recebidos na entrada (H_pos e V_pos) são divididos em partes menores, pois cada conjunto de bits de seus valores é utilizado para fazer acesso a uma das memórias. Estas duas entradas são divididas da seguinte forma:

- V_pos, bits de 8 a 4: indicam a linha do caractere, são conectados aos bits de 11 a 7 da entrada read_addr da Memória de Tela;
- H_pos, bits de 9 a 3: indicam a posição do caractere dentro da linha, são conectados aos bits de 6 a 0 da entrada read_addr da Memória de Tela;
- V_pos, bits de 3 a 0: indicam a linha de pixels do caractere atual, são conectadas aos bits de 3 a 0 da entrada address da Memória de caracteres;
- H_pos, bits de 2 a 0: indicam o pixel atual dentro da linha de pixels, são conectados ao seletor do multiplexador, este seleciona o pixel requerido pelo controlador dentre os oito recebidos da Memória de Caracteres. Para adequar a temporização das memórias é necessário sincronizar o seletor com o atraso de resposta das memórias,

utilizando dois registradores para isso.

As entradas para escrita de caractere na memória são conectadas diretamente às suas correspondentes no módulo Memória de Tela.

A saída Pixel_Valor recebe seu valor do multiplexador.

5 PROJETO VGA GENÉRICO

Este projeto tem como objetivo simplificar o uso do Driver VGA, tornando possível utilizar a placa DE0 para simular um módulo mostrar suas saídas em um monitor VGA.

Nesta Seção serão descritos os componentes e módulos adicionais necessários para produzir um projeto de fácil utilização juntamente com um tutorial para seu uso.

5.1 Interface e Driver VGA

Este módulo reúne o módulo de Driver VGA com um módulo de interface e o divisor de frequências necessário para o sincronismo da tela. Tem como função abstrair o endereçamento da Memória de Tela e controle de sincronismo do VGA, oferecendo um número limitado de registradores mapeados no monitor como entrada e tem como saída os sinais de controle que serão enviados ao monitor VGA.

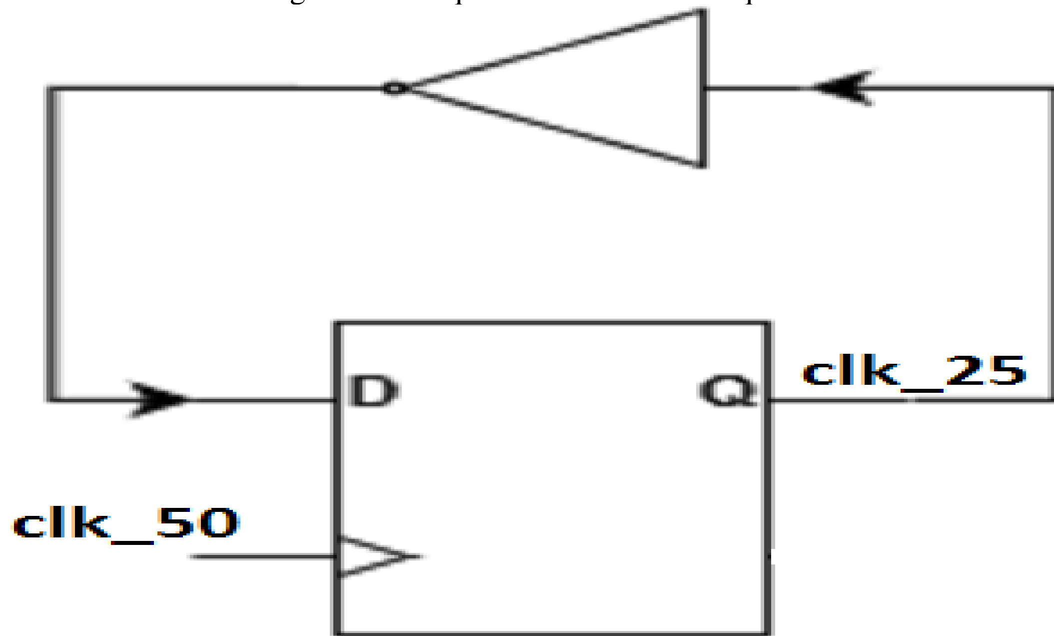
A seguir são descritos a concepção de um módulo de interface para utilizar o Driver VGA previamente descrito e o divisor de frequências para o sincronismo do controle VGA.

5.1.1 Divisor de Frequências

Este módulo é necessário para adaptar a frequência do relógio disponibilizado pela placa DE0 à frequência de relógio utilizada pelo Driver VGA e pela interface desenvolvidos.

O relógio disponível na placa DE0 possui uma frequência de 50MHz e a frequência necessária para o funcionamento dos módulos desenvolvidos é de 25MHz, isto significa que é necessário reduzir a frequência disponível pela metade. A Figura 5.1 ilustra a arquitetura.

Figura 5.1: Arquitetura divisor de frequências



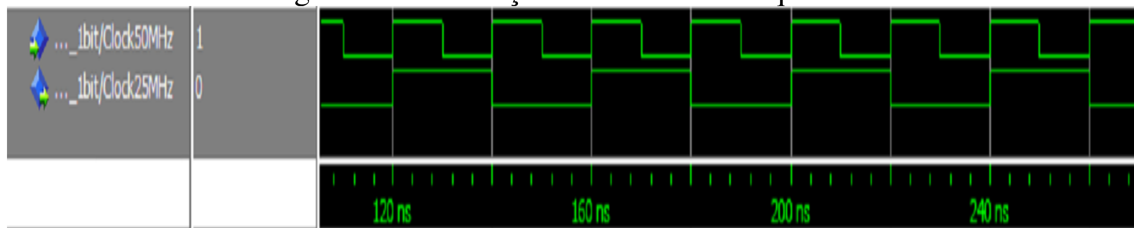
Considerando que 50MHz tem um ciclo de relógio de 20ns, utiliza-se um flip-flop onde o relógio de 50MHz é utilizado para carregar o flip-flop e a porta de saída deste passa por uma inversão lógica que realimenta a entrada. Desta forma dois ciclos do relógio de carga geram um ciclo de relógio de 40ns na saída, obtendo-se assim a frequência de 25 MHz necessária. O código 5.1 implementa este módulo em Verilog e a Figura 5.2 apresenta sua simulação.

Código 5.1 – Divisor de frequências em Verilog

```
module div_freq(
    input      clk_50,
      input      rst ,
      output  reg  clk_25
    );

    always @ (  posedge clk_50 or posedge  rst )
    begin
        if  (  rst  == 1'b1  )
            begin
                clk_25 <= {  1'b0  };
            end
        else
```

```
            begin
15          if ( clk_50 )
                begin
17                  clk_25 <= ~ clk_25;
                end
19      end
    end
21 endmodule
```

Figura 5.2: Simulação do divisor de frequências



5.1.2 Interface

O módulo de Interface tem com função receber o valor dos registradores do usuário e atualizar seus valores na Memória de Tela. Para desenvolver esta interface é necessário definir algumas especificações sobre como estas informações serão visualizadas no monitor. Estas especificações e seus respectivos valores são listadas a seguir:

- Número de registradores mostrados na tela: 12 registradores nomeados de R0 a R11;
- Tamanho dos registradores: 16 bits cada registrador;
- Formato do valor de cada registrador na tela: representado em base binária, com espaçamento de 4 em 4 bits, ex.: 0000 0000 0000 0000;
- Posição dos nomes dos registradores na tela: a partir da linha 2 até a 24 com espaçamento de 1 linha entre eles (2,4,6,8,..,24). Os nomes dos registradores são inicializados diretamente em suas posições na Memória de Tela e não são alterados por esta interface.
- Posição dos valores dos registradores na tela: nas mesmas linhas de seus respectivos nomes, cada bit do registrador tem uma posição fixa;

Pelo fato de o Driver VGA possuir apenas uma entrada de escrita é necessário

atualizar um caractere da memória de cada vez. Para isso é usado um contador que é incrementado a cada ciclo de relógio. De acordo com o valor atual do contador, um dos bits de um dos registradores é atualizado na Memória de Tela. Considerando que a quantidade de informações que devem ser atualizadas na tela é de 192 (12×16), número de registradores pela quantidade de bits de cada um, é necessário um contador com 8 bits de tamanho contando de 0 a 255.

A implementação deste módulo consiste em um bloco *switch case*. Nele estão descritos que de acordo com o valor do contador:

- Um dos bits de um dos registradores de entrada é selecionado;
- A linha em que este bit será salvo na Memória de tela;
- A posição dentro da linha em que ele será salvo na Memória;
- Ativa o sinal de escrita na Memória de tela;

Pelo fato deste contador ser reiniciado ao chegar ao seu valor máximo (255) ser maior que a quantidade de dados a serem atualizados (192), existe um período ocioso entre o intervalo de 192 a 255 do contador.

Após a seleção do dado a ser atualizado e sua posição na tela, é feita a formatação de seus valores para corresponder aos dados esperados pelas entradas de escrita da memória de tela, o sinal de escrita na memória não precisa ser formatado.

Sendo o valor esperado como dado de entrada para escrita na Memória de Tela o valor de um caractere ASCII, o valor do bit selecionado é convertido para seu valor ASCII.

Os Valores de linha e de posição dento da linha são concatenados de acordo com a especificação de endereçamento da Memória de tela, ou seja, os bits de endereço de 11 a 7 recebem valor da linha e endereço de 6 a 0 recebem valor da posição dentro da linha.

5.2 Projeto VGA

A ferramenta utilizada para o desenvolvimento de trabalhos e para a programação da placa DE0 é o Quartus II.

Após o desenvolvimento de um projeto tendo como alvo a programação da placa, é necessário utilizar a ferramenta de mapeamento das entradas e saídas do projeto desenvolvido para seus respectivos pinos na placa. Estas especificações estão disponíveis no manual de uso da placa de desenvolvimento DE0.

Este projeto foi desenvolvido com base na ferramenta Quartus II para a programação e uso da Placa DE0. Utiliza a linguagem Verilog para conectar o módulo de Interface e Driver VGA descritos acima com o módulo usuário que terá suas saídas enviadas para o monitor VGA. Algumas entradas disponíveis na placa DE0 estão declarados e mapeados, assim como os pinos de saída H_SYNC, V_SYNC e RGB já estão mapeados ao conector.

O apêndice 7 é um tutorial que descreve o uso deste projeto.

As frequências dos sinais de controle VGA, geradas pela placa DE0 com este projeto, foram medidas com um osciloscópio. A Figura 5.3 mostra H_SYNC e a Figura 5.4 mostra V_SYNC.

Figura 5.3: Frequência de H_SYNC no osciloscópio

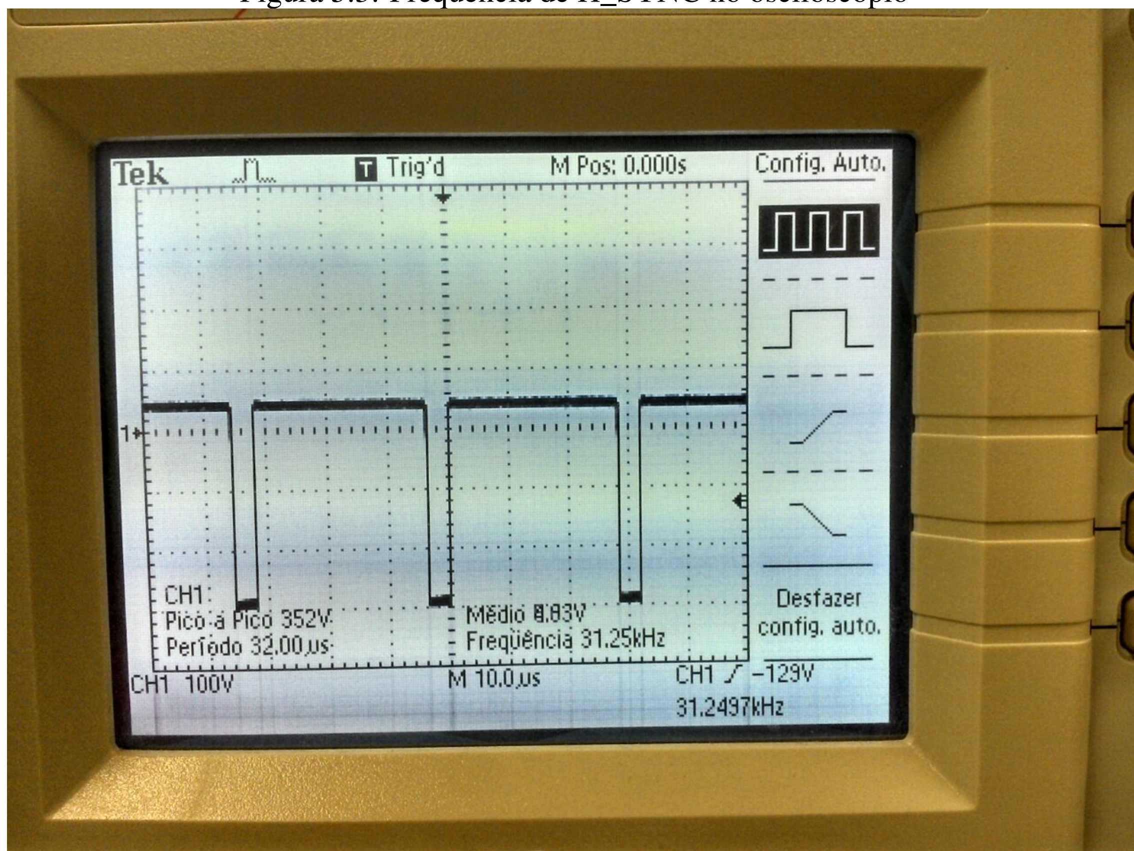
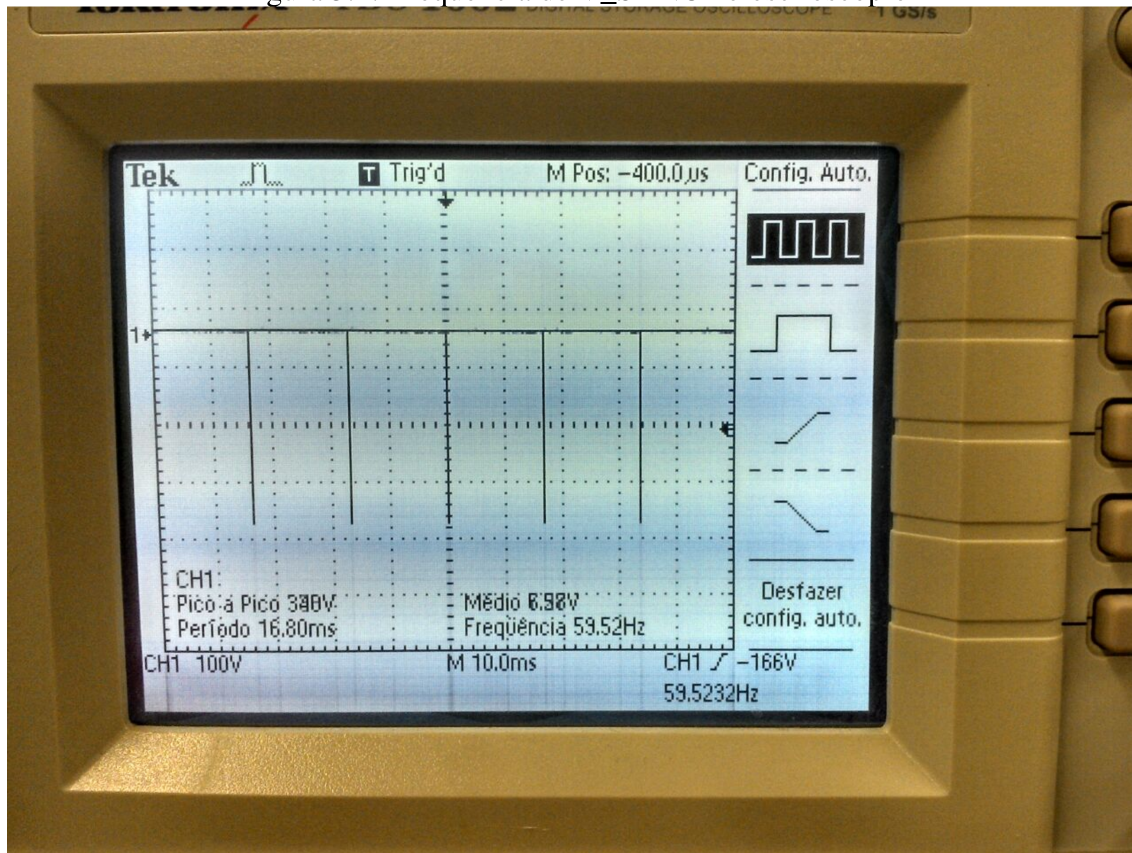
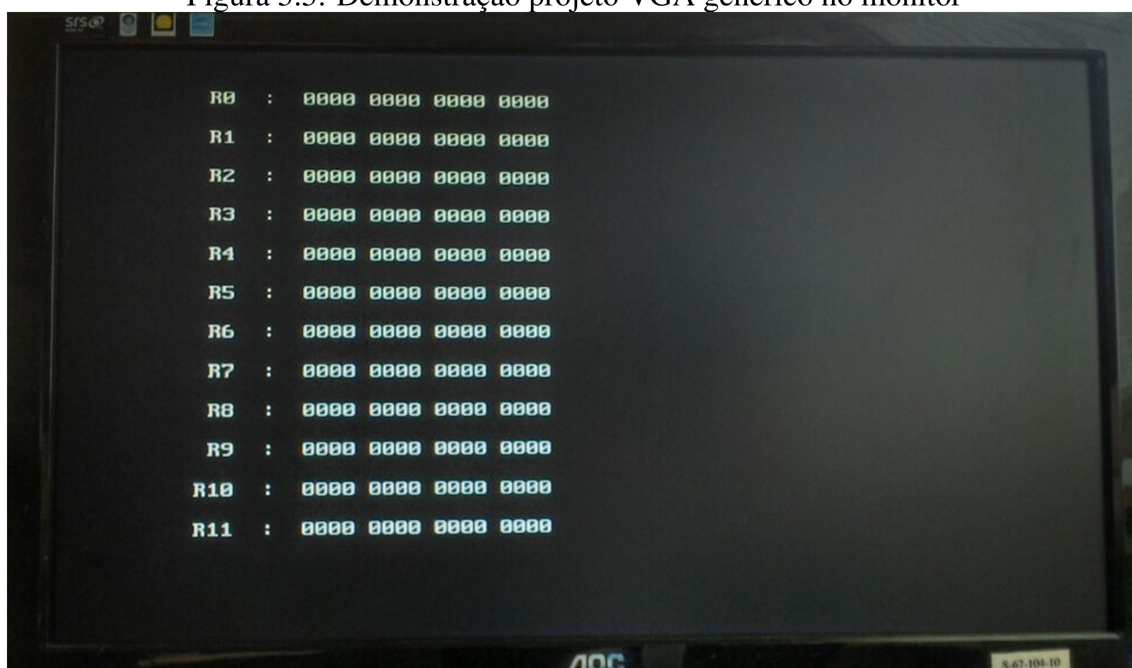


Figura 5.4: Frequência de V_SYNC no osciloscópio



A Figura 5.5 mostra a disposição desses doze registradores genéricos em um monitor.

Figura 5.5: Demonstração projeto VGA genérico no monitor



5.3 Computadores Hipotéticos no Monitor

A fim de mostrar o funcionamento da máquina hipotética Ahmes com unidade de controle temporizada com RAM, através do fluxo descrito na Seção 5.2 realiza-se a demonstração do valor de seus registradores no monitor, conforme ilustra a Figura 5.6.

O código 5.2 demonstra os botões da placa e os registradores de interface que foram utilizados.

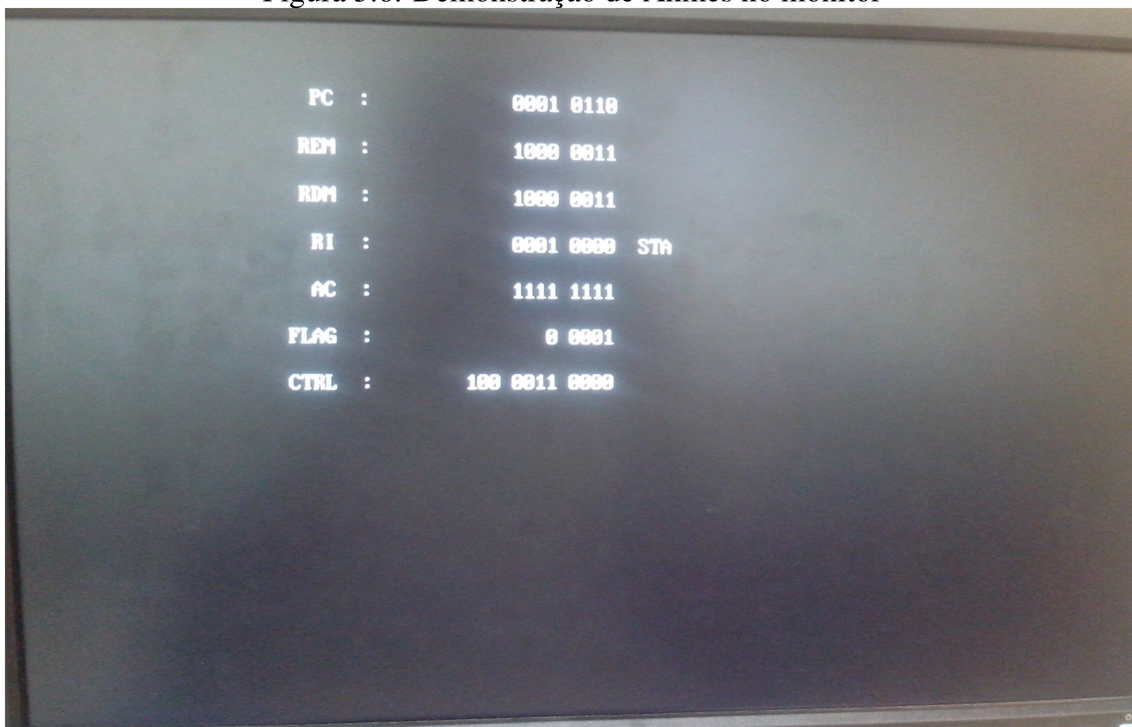
Código 5.2 – Conexão entre Driver VGA e Ahmes

```
module VGA_TopLevelEntity
    (
        input  CLOCK_50,
      input  botao0,
      input  botao1,
      input  botao2,
      input  chave0,
      input  chave1,
      input  chave2,
      input  chave3,
      input  chave4,
      input  chave5,
      input  chave6,
      input  chave7,
      input  chave8,
      input  chave9,
        output [3:0]    oVGA_RED,
        output [3:0]    oVGA_GREEN,
        output [3:0]    oVGA_BLUE,
        output      oVGA_HS,
        output      oVGA_VS
      );

//************************************************************
// declaracao das ligacoes entre design do usuario e modulo VGA
//************************************************************
  wire    [15:0]   R0 ,
            R1 ,
            R2 ,
            R3 ,
            R4 ,
```

```
                R5 ,
33              R6 ,
                R7 ,
35              R8 ,
                R9 ,
37              R10 ,
                R11 ;
39  //*****************************************************

41
    //***********************************************************
43  //Declare seu modulo
    //***********************************************************
45
    Ahmes_com_goto0 user_design(
47       .clock (botao1),  //  input
         . rst   (~botao0),   //  input
49       .PC    (R0[7:0]),   //  output  [7:0]
         .REM   (R1[7:0]),   //  output  [7:0]
51       .RDM   (R2[7:0]),   //  output  [7:0]
         .RI    (R3[7:0]),   //  output  [7:0]
53       .AC    (R4[7:0]),   //  output  [7:0]
         .FLAGS (R5[4:0]),  //  output  [4:0]
55       .CTRL ()   //  output  [10:0]
    );
57  //*****************************************************
    s
59  //**********************************************************
    // declaracao da interface + driver VGA
61  //**********************************************************

63    vga_interface_driver     vga_interface_driver_inst
          (
65          .oVGA_RED(oVGA_RED) , // output [3:0] oVGA_RED
            .oVGA_GREEN(oVGA_GREEN) , // output [3:0] oVGA_GREEN
67          .oVGA_BLUE(oVGA_BLUE) , // output [3:0] oVGA_BLUE
            .oVGA_HS(oVGA_HS) , // output  oVGA_HS
69          .oVGA_VS(oVGA_VS),  // output  oVGA_VS
            .CLOCK_50(CLOCK_50) , // input  CLOCK_50
71          .iRst (1'b1)  ,  //  input   stuck  at  1
            .iR0(R0) ,    //  input  [15:0]  iR0
```

```
73          .iR1(R1) ,     // input [15:0] iR1
            .iR2(R2) ,     // input [15:0] iR2
75          .iR3(R3) ,     // input [15:0] iR3
            .iR4(R4) ,     // input [15:0] iR4
77          .iR5(R5) ,     // input [15:0] iR5
            .iR6(R6) ,     // input [15:0] iR6
79          .iR7(R7) ,     // input [15:0] iR7
            .iR8(R8) ,     // input [15:0] iR8
81          .iR9(R9) ,     // input [15:0] iR9
            .iR10(R10) ,   // input [15:0] iR10
83          .iR11(R11)     // input [15:0] iR11
        );
85  //************************************************************

87

89

91  endmodule
```

Figura 5.6: Demonstração de Ahmes no monitor



6 CONCLUSÃO E SUGESTÕES PARA TRABALHOS FUTUROS

Analisando as etapas de implementação envolvidas neste trabalho, concluí que a motivação de criar um módulo para VGA foi decorrente da necessidade de demonstrar o funcionamento das máquinas hipotéticas desenvolvidas. Isto me relembrou as dificuldades de apresentar projetos de sistemas digitais durante a graduação e levou a proposta de definir uma interface simplificada para utilizar o driver VGA.

O fato de ter começado pelo desenvolvimento das máquinas hipotéticas também ajudou durante a concepção e implementação do projeto VGA. A implementação de diversas versões das máquinas hipotéticas Neander e Ahmes levaram a uma análise sobre as memórias ROM e RAM que são fundamentais para o funcionamento de um dispositivo de vídeo.

Ainda nesta primeira etapa, o uso das diferentes linguagens de descrição de hardware me proporcionou escolher o Verilog, por ter uma sintaxe mais simplificada que o VHDL, para descrever o arquivo que liga o módulo usuário à interface do projeto VGA.

Apesar de ter implementado apenas duas das quatro máquinas hipotéticas (Neander, Ahmes, Ramses e Cesar), a análise e implementação destas fornece base de conhecimentos necessários para implementar máquinas mais complexas.

Sobre trabalhos futuros, na perspectiva de aprofundamento em arquiteturas de computadores seria válido implementar Ramses e Cesar. Buscando expandir a quantidade de dispositivos de entrada disponíveis no kit de desenvolvimento DE0, um módulo para entrada de texto utilizando um teclado poderia ser desenvolvido e adaptado ao driver VGA.

REFERÊNCIAS

ALTERA Website. 2016. <https://www.altera.com>. [Online; accessed 19-12-2016].

CHU, P. **FPGA Prototyping by VHDL Examples: Xilinx Spartan-3 Version**. Wiley, 2008. ISBN 9780470231623. Available from Internet: <https://books.google.ro/books?id=mwUV7ZK9l9gC>.

DECKER, R.; HIRSHFIELD, S. The pippin machine: Simulations of language processing. **J. Educ. Resour. Comput.**, ACM, New York, NY, USA, v. 1, n. 4, p. 4–17, dec. 2001. ISSN 1531-4278. Available from Internet: <http://doi.acm.org/10.1145/514144.514706>.

FPGA Definition Xilinx. 2016. <https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array>. [Online; accessed 19-12-2016].

ORTH, G. K. Implementação em hardware da arquitetura do computador hipotético cesar. 2010.

SKRIEN, D. Cpu sim a computer simulator for use in an introductory computer organization-architecture class. **Journal of Computing in Higher Education**, v. 6, n. 1, p. 3–13, 1994. ISSN 1042-1726. Available from Internet: <http://dx.doi.org/10.1007/BF03035480>.

VESA Website. 2016. <http://www.vesa.org/>.

WEBER, R. **Fundamentos de Arquitetura de Computadores - Vol.8: Série Livros Didáticos Informática UFRGS**. Bookman, 2009. ISBN 9788540701434. Available from Internet: <https://books.google.com.br/books?id=UAiPQ60dRjMC>.

WOLFFE, G. S. et al. Teaching computer organization/architecture with limited resources using simulators. **SIGCSE Bull.**, ACM, New York, NY, USA, v. 34, n. 1, p. 176–180, feb. 2002. ISSN 0097-8418. Available from Internet: <http://doi.acm.org/10.1145/563517.563408>.

XILINX Website. 2016. <https://www.xilinx.com>. [Online; accessed 19-12-2016].

7 APÊNDICE

Código 7.1 – MemoriaROM em Verilog

```
module MemoriaRom (
ender ,
dado
);
input [7:0] ender;
output [7:0] dado;

reg [7:0] dado ;

always @ (ender)
begin
  case (ender)
    0 : dado = 10;
    1 : dado = 55;
    2 : dado = 244;
    3 : dado = 0;
    4 : dado = 1;
    5 : dado = 8'hff;
    6 : dado = 8'h11;
    7 : dado = 8'h1;
    8 : dado = 8'h10;
    9 : dado = 8'h0;
    10 : dado = 8'h10;
    11 : dado = 8'h15;
    12 : dado = 8'h60;
    13 : dado = 8'h90;
    14 : dado = 8'h70;
    15 : dado = 8'h90;
   default : dado = 8'h00;
  endcase
end
```

Código 7.2 – MemoriaROM em VHDL com with select

```
library ieee;
use ieee.std_logic_1164.all;

```

```
5  entity MemoriaRom is
       port (endr: in std_logic_vector (7 downto 0 );
7              dado: out std_logic_vector ( 7 downto 0));
   end MemoriaRom;
9

11 architecture memoria of MemoriaRom is
   begin
13
       with endr select
15     dado<= "00100000" WHEN "00000000" , --LDA
              "10000000" WHEN "00000001" , --ender
17            "00110000" WHEN "00000010" , --ADD
              "10000001" WHEN "00000011" , --ender
19            "10100000" WHEN "00000100" , --JZ
              "00001100" WHEN "00000101" , --ender
21            "10010000" WHEN "00000110" , --JN
              "00001100" WHEN "00000111" , --ender
23            "01000000" WHEN "00001000" , --OR
              "10000010" WHEN "00001001" , --ender
25            "10010000" WHEN "00001010" , --JN
              "00001101" WHEN "00001011" , --ender
27            "11110000" WHEN "00001100" , --HLT
              "01100000" WHEN "00001101" , --NOT
29            "01010000" WHEN "00001110" , --AND
              "10000010" WHEN "00001111" , --ender
31            "10100000" WHEN "00010000" , --JZ
              "00010011" WHEN "00010001" , --ender
33            "11110000" WHEN "00010010" , --HLT
              "01100000" WHEN "00010011" , --NOT
35            "00010000" WHEN "00010100" , --STA
              "10000011" WHEN "00010101" , --ender
37            "10000000" WHEN "00010110" , --JMP
              "00011001" WHEN "00010111" , --ender
39            "11110000" WHEN "00011000" , --HLT
              "01010000" WHEN "00011001" , --AND
41            "10000100" WHEN "00011010" , --ender
              "00010000" WHEN "00011011" , --STA
43            "00100000" WHEN "00011100" , --ender
              "00000000" WHEN "00011101" , --NOP
45            "00000000" WHEN "00011110" , --NOP
```

```
                "00000000"   WHEN "00011111"  , --NOP
47              "00000000"   WHEN "00100000"  , --NOP
                "00001010"   WHEN "10000000"  , --dados
49              "00000001"   WHEN "10000001"  , --dados
                "10001111"   WHEN "10000010"  , --dados
51              "00000000"   WHEN "10000011"  , --dados
                "11110000"   WHEN "10000100"  , --dados
53
                "00000000"    WHEN others;
55


57  end memoria;
```

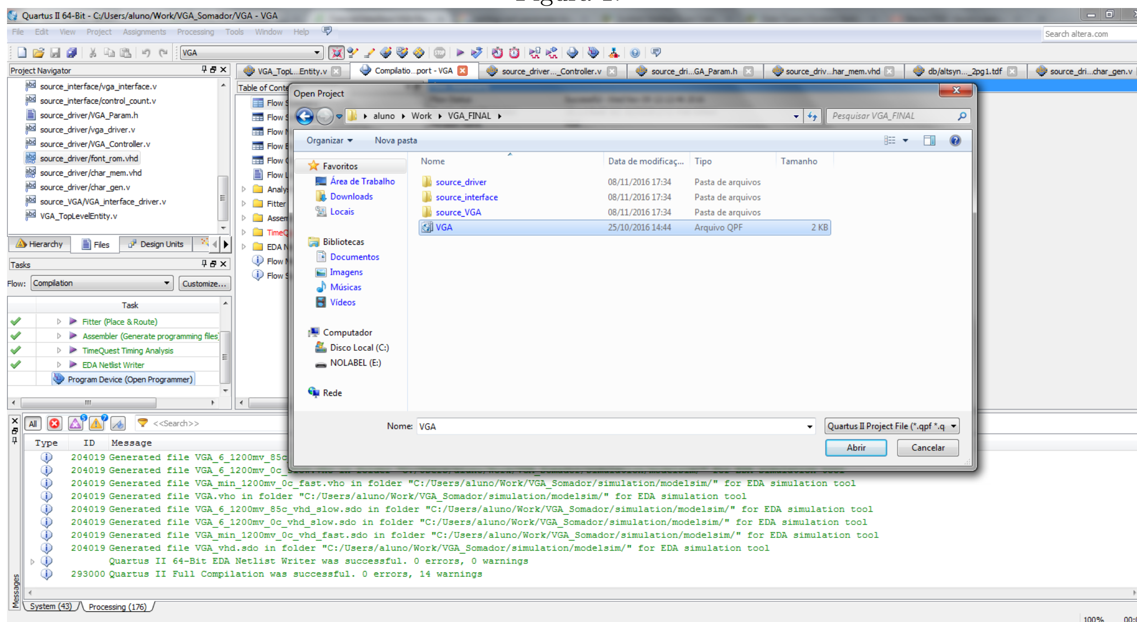
Tutorial Interface VGA Para DE0

Este tutorial apresenta uma metodologia rápida para utilização da interface VGA genérica disponibilizada, a qual foi desenvolvida para placa DE0. A interface foi desenvolvida na linguagem de hardware Verilog, e pode ser usada para visualizar módulos descritos em Verilog, VHDL e arquivos .bdf(Block Diagram File).

1 Organizando Arquivos e Projeto

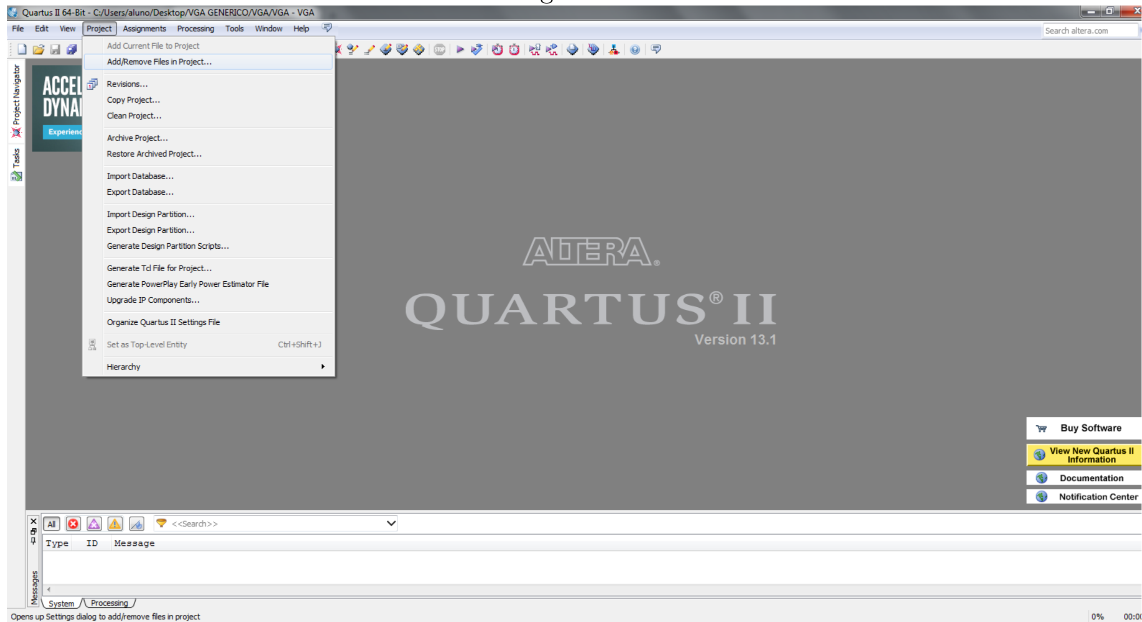
- Abra o projeto "VGA"no Quartus II "File→Open Project...", Figura 1

Figura 1:



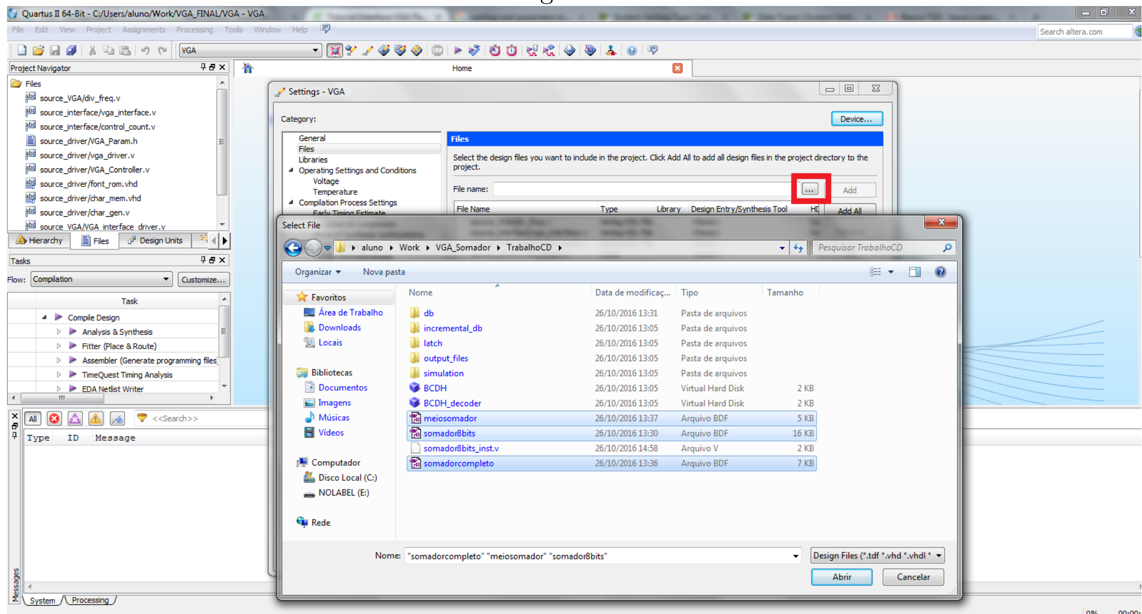
- Abra o menu "Project→Add/Remove Files in Project...", Figura 2.

Figura 2:



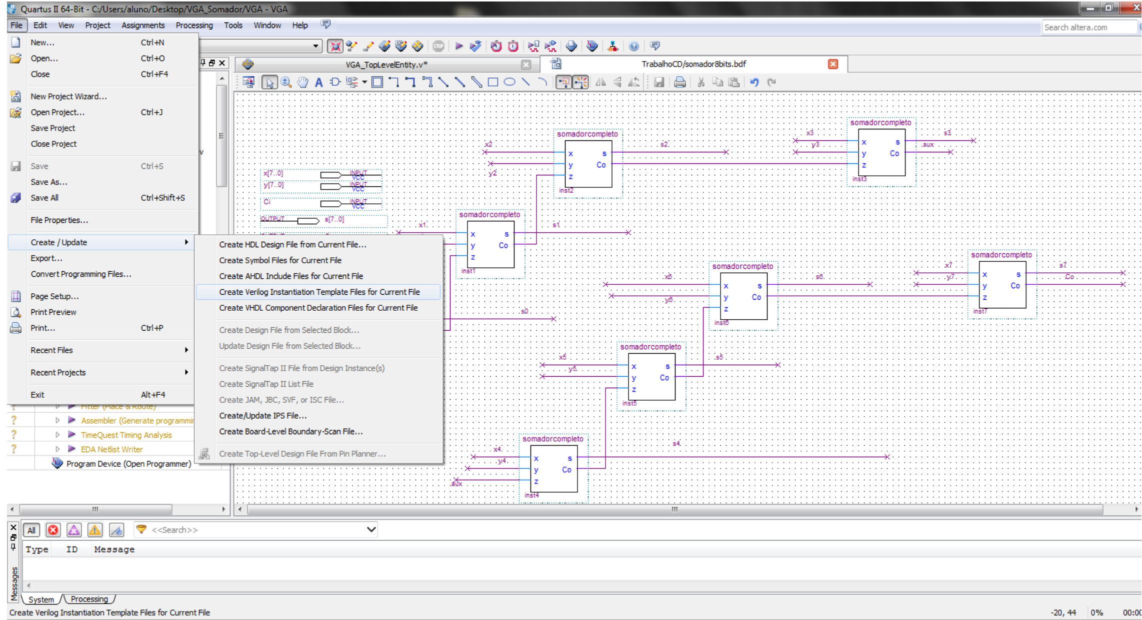
- Adicione os arquivos contendo seus módulos, Figura 3.

Figura 3:



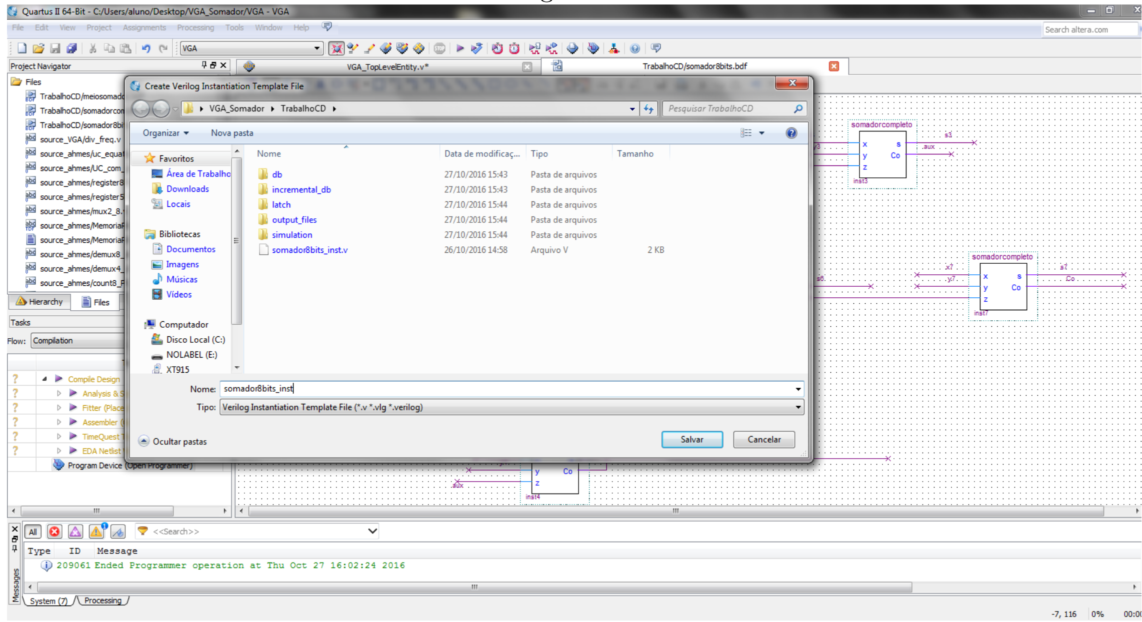
- Abra seu modulo topo que será visualizado na tela e vá em "File→Create/Update→Create Verilog Instantiation Template Files for Current File"para criar a instância Verilog para este modulo. 4

Figura 4:



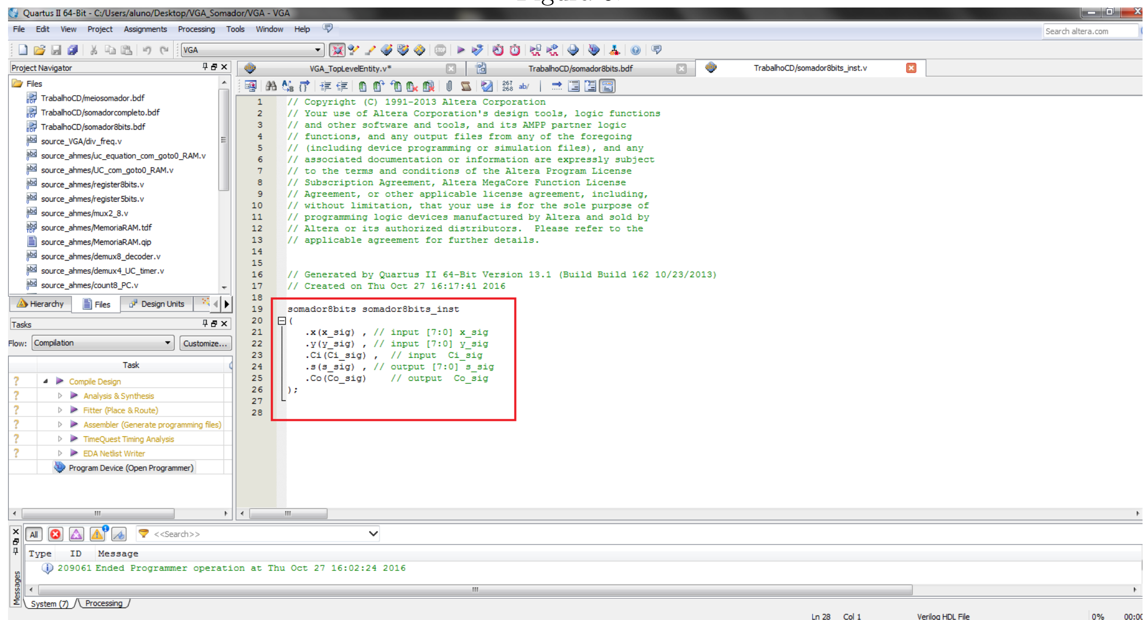
- Salvar a instancia do módulo no arquivo "'nome_do_modulo'_inst.v". Figura 5.

Figura 5:



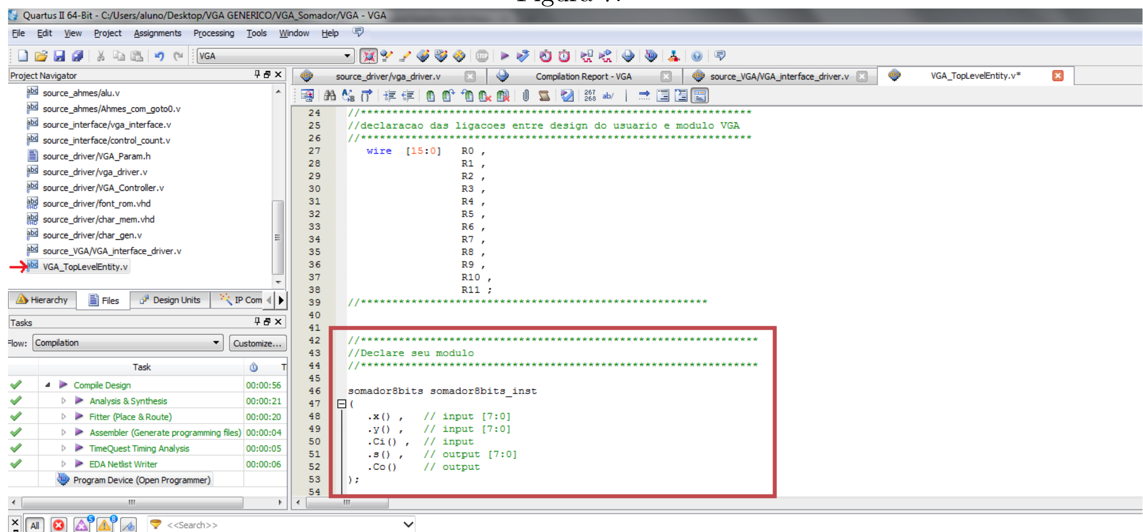
- Abra o arquivo "'nome_do_modulo'_inst.v"e copie a instância do módulo. Figura 6

Figura 6:



- Cole na seção indicada no módulo topo do projeto VGA, arquivo "VGA_TopLevelEntity.v",Figura 7.
- Remova os sinais criados pela ferramenta, eles serão substituídos a seguir.Figura 7

Figura 7:



2 Conectando a Interface VGA com o Módulo do Usuário:

No arquivo "VGA_TopLevelEntity.v"faça as conexões entre a interface VGA e o seu modulo, descritas a seguir:

- Saídas:

  Esta interface possui 12 registradores de ligação para entrada de dados(R1 a R11), cada registrador é composto por 16 bits.

  As saídas (outputs) do módulo do usuário devem ser ligadas nos registradores de ligação, esta operação está ilustrada na Figura 8.

Figura 8:

```
Text Editor - C:/Users/aluno/Desktop/VGA GENERICO/VGA_Somador/VGA - VGA - [VGA_TopLevelEntity.v]*
File  Edit  View  Project  Processing  Tools  Window  Help

24   //*****************************************************************
25   //declaracao das ligacoes entre design do usuario e modulo VGA
26   //*****************************************************************
27      wire  [15:0]   R0 ,
28                     R1 ,
29                     R2 ,
30                     R3 ,
31                     R4 ,
32                     R5 ,
33                     R6 ,
34                     R7 ,
35                     R8 ,
36                     R9 ,
37                     R10 ,
38                     R11 ;
39   //*********************************************************
40
41
42   //******************************************************************
43   //Declare seu modulo
44   //******************************************************************
45
46   somador8bits somador8bits_inst
47   (
48      .x() ,           // input [7:0]
49      .y() ,           // input [7:0]
50      .Ci() ,          // input
51      .s(R7[7:0]) ,    // output [7:0]
52      .Co(R8[0])       // output
53   );
54
55   Ahmes_com_goto0 user_design(
56          .clock   (),           // input
57          .rst     (),           // input
58          .PC      (R0[7:0]),    // output [7:0]
59          .REM     (R1[7:0]),    // output [7:0]
60          .RDM     (R2[7:0]),    // output [7:0]
61          .RI      (R3[7:0]),    // output [7:0]
62          .AC      (R4[7:0]),    // output [7:0]
63          .FLAGS   (R5[4:0]),    // output [4:0]
64          .CTRL    (R6[10:0]),   // output [10:0]
65   );
66   //*********************************************************
67
68
```

1. Obs.: Quando fizer a conexão de um registrador de ligação com uma saída do módulo de usuário deve-se especificar quantos bits do registrador de ligação serão utilizados. Ex: saída do usuário possui 8 bits, então a conexão deve ser com R[7:0]. Linha 51 Figura 8
2. Obs.: Os registradores de ligação são declarados como big endian [N:0].

- Entradas:

  Este projeto contém as chaves e botões disponíveis na placa e já estão mapeados, são eles:

  – Sinais referentes às 10 chaves disponíveis declarados como chave0, chave1, chave2, chave3, chave4, chave5, chave6, chave7, chave8, chave9;

  – Sinais referentes aos 3 botões disponíveis declarados como botao0, botao1, botao2 (os botões estão fixos em "1"lógico e quando pressionados ocorre a transição para "0"lógico);

  As entradas (inputs) do módulo do usuário devem ser ligadas nos sinais descritos a cima ,ou definir valores fixos, de acordo com a necessidade do usuário, Figura 9.

Figura 9:

```
Text Editor - C:/Users/aluno/Desktop/VGA GENERICO/VGA_Somador/VGA - VGA - [VGA_TopLevelEntity.v]*
File  Edit  View  Project  Processing  Tools  Window  Help

24  //***************************************************************
25  //declaracao das ligacoes entre design do usuario e modulo VGA
26  //***************************************************************
27     wire  [15:0]    R0 ,
28                     R1 ,
29                     R2 ,
30                     R3 ,
31                     R4 ,
32                     R5 ,
33                     R6 ,
34                     R7 ,
35                     R8 ,
36                     R9 ,
37                     R10 ,
38                     R11 ;
39  //********************************************************
40
41
42  //****************************************************************
43  //Declare seu modulo
44  //****************************************************************
45
46  somador8bits somador8bits_inst
47  (
48     .x({chave7,chave6,chave5,chave4,chave3,chave2,chave1,chave0}) ,   // input [7:0]
49     .y(8'h00) ,     // input [7:0]
50     .Ci(~botao2) , // input
51     .s(R7[7:0]) ,  // output [7:0]
52     .Co(R8[0])     // output
53  );
54
55  Ahmes_com_goto0 user_design(
56         .clock   (botao1),    // input
57         .rst     (~botao0),   // input
58         .PC      (R0[7:0]),   // output [7:0]
59         .REM     (R1[7:0]),   // output [7:0]
60         .RDM     (R2[7:0]),   // output [7:0]
61         .RI      (R3[7:0]),   // output [7:0]
62         .AC      (R4[7:0]),   // output [7:0]
63         .FLAGS   (R5[4:0]),   // output [4:0]
64         .CTRL    (R6[10:0]),  // output [10:0]
65  );
66  //********************************************************
67
68
```

1. Obs.: Para concatenar sinais deve-se utilizar a sintaxe {chaveN,...,chave3,chave2,chave1,chave0}.
   Ex: linha 48, Figura 9;
2. Obs.: Para fixar valores deve-se utilizar a sintaxe N'Kxxx, sendo :
   N = número de bits;
   K = indica a base: "h"para hexadecimal, "b"para binário, "d"para decimal;
   xxx = valor
   Ex: linha 49, Figura 9;
3. Obs.: Para utilizar um dos botões como reset ativo em "1"lógico, é necessário fazer a negação.
   Ex: linha 57, Figura 9;
4. Obs.: Para criar um clock utilize um dos botões.
   Ex: linha 56, Figura 9;

3 Compilando e programando FPGA:

- Compile o projeto "Processing→Start Compilation".
- Abra a ferramenta para programar a placa em "Tools→Programmer".
- Clique em "Auto Detect"e a seguir em "Start"para programar, Figura 10.

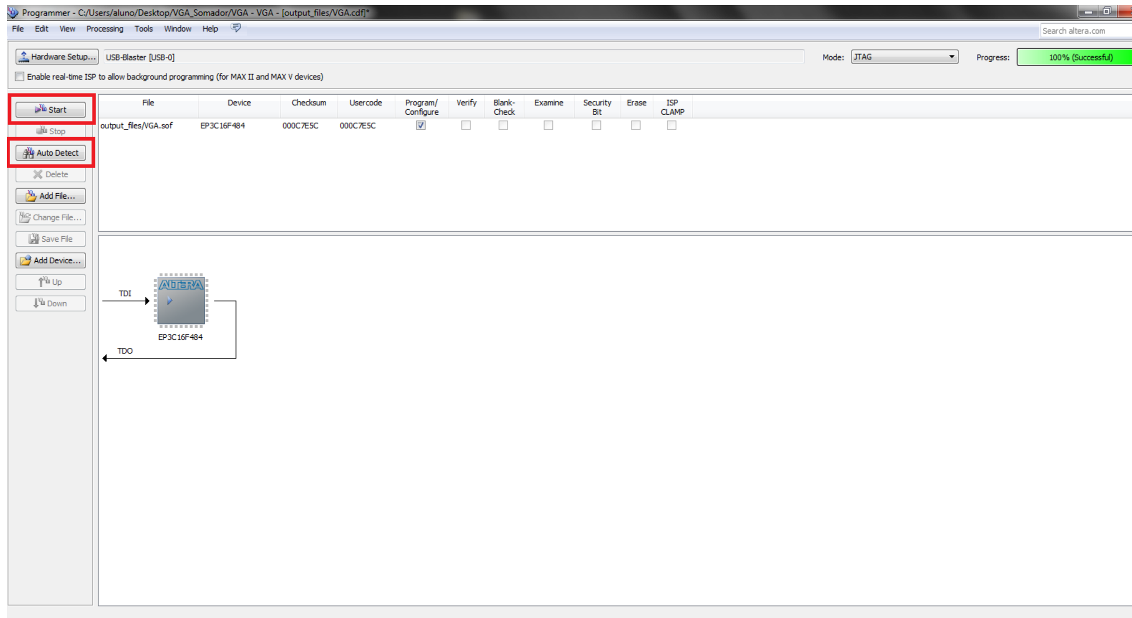


Figura 10:

- Utilize as chaves e botões da placa para interagir com o seu trabalho.

Mais referências sobre o uso podem ser encontradas no manual da placa DE0.

Código 7.3 – Script parcer de memória

```
#!/usr/bin/env python
import sys
import os

def criaMemoriaROM_esqueleto(fROM):
    fROM.write( """ library ieee;
use ieee.std_logic_1164.all;

 entity MemoriaRom is
    port (endr: in std_logic_vector (7 downto 0 );
          dado: out std_logic_vector ( 7 downto 0));
end MemoriaRom;

 architecture memoria of MemoriaRom is
begin
process( endr )
begin
    case endr is
""")
    return

def caseMemoriaROMDEC(fread, fwrite):
    #para cada linha do arquivo gera uma entrada na memoria
    fread.seek(0)
    for linha in fread:
        mementry = linha.split()
        if len(mementry) == 3:
            endereco = "{0:08b}".format(int(mementry[0]))
            operacao = "{0:08b}".format(int(mementry[1]))
            fwrite.write("           WHEN \""+\
                            endereco+\
                            "\" => dado <= \""+\
                            operacao+\
                            "\" ; --"+\
                            mementry[2]+\
                            "\n"
                           )
        elif len(mementry) == 5:
            endereco = "{0:08b}".format(int(mementry[0]))
            operacao = "{0:08b}".format(int(mementry[1]))
```

```
41                      fwrite . write ("              WHEN \""+\
                                          endereco+\
43                                        "\"  => dado <= \""+\
                                          operacao+\
45                                        "\"  ;  --"+\
                                          mementry[3]+\
47                                        "\n"
                                         )
49                      endereco_maisum = "{0:08b}".format( int (mementry[0])+1)
                        enderecodado = "{0:08b}".format( int (mementry[2]))
51                      fwrite . write ("              WHEN \""+\
                                          endereco_maisum+\
53                                        "\"  => dado <= \""+\
                                          enderecodado+\
55                                        "\"  ;  --"+\
                                          "ender"+\
57                                        "\n"
                                         )
59                  elif  len(mementry) == 2:
                        endereco  = "{0:08b}".format( int (mementry[0]))
61                      operacao  =  "{0:08b}".format( int (mementry[1]))
                        fwrite . write ("              WHEN \""+\
63                                        endereco+\
                                          "\"  => dado <= \""+\
65                                        operacao+\
                                          "\"  ;  "+\
67                                        "-- aread de dados"+\
                                          "\n"
69                                       )
        fwrite . write ("""              WHEN others => dado <= "00000000";
71      end case;
   end process;
73 end memoria;
                            """)
75      print  "MemoriaROM.vhd.....OK"
        return
77
   def criaMIFDEC(fread,fwrite):
79      fread . seek(0)
        #cria  parametrios  pra  memoria
81      fwrite . write ("DEPTH = 256 ;                    -- Size of memory in word\n")
```

```
    fwrite.write("WIDTH = 8;                          -- The size of data in bits\n")
83  fwrite.write("ADDRESS_RADIX = BIN;                -- The radix for address values\n" )
    fwrite.write("DATA_RADIX = BIN;                   -- The radix for data values\n")
85  fwrite.write("CONTENT                             -- start of (address : data pairs)\n")
    fwrite.write("BEGIN\n")
87  for linha in fread:
        mementry = linha.split()
89      if len(mementry) == 3:
            endereco = "{0:08b}".format(int(mementry[0]))
91          operacao = "{0:08b}".format(int(mementry[1]))
            fwrite.write(endereco +\
93                          " : "+\
                            operacao +\
95                          "; --" +\
                            mementry[2]+\
97                          "\n"
                          )
99      elif len(mementry) == 5:
            endereco = "{0:08b}".format(int(mementry[0]))
101         operacao = "{0:08b}".format(int(mementry[1]))
            fwrite.write(endereco+\
103                         " : "+\
                            operacao+\
105                         "; -- "+\
                            mementry[3]+\
107                         "\n"
                          )
109         endereco_maisum = "{0:08b}".format(int(mementry[0])+1)
            enderecodado = "{0:08b}".format(int(mementry[2]))
111         fwrite.write(endereco_maisum+\
                            " : "+\
113                         enderecodado+\
                            "; --"+\
115                         "ender"+\
                            "\n"
117                       )
        elif len(mementry) == 2:
119         endereco = "{0:08b}".format(int(mementry[0]))
            operacao = "{0:08b}".format(int(mementry[1]))
121         fwrite.write(endereco+\
                            " : "+\
```

```
                                operacao+\
                                ";   "+\
                                "-- aread de dados"+\
                                "\n"
                               )
    fwrite.write("END;")
    print "MemoriaROM.mif.....OK"
    return

def main():
    # lista de parametros
    param = sys.argv[1:]
    # testa parametro recebido para saber se e txt
    if param[0].endswith(".txt") or param[0].endswith(".TXT"):
        fNeander = open(param[0], "r")
        print ("argumentos recebitos ... .. OK")
    else:
        print ("Argumento invalido, extencao deve ser .txt")
        sys.exit()

    #Cria MemmoriaROM.vhd
    fROM = open("MemoriaROM.vhd", "w")
    criaMemoriaROM_esqueleto(fROM)
    #Cria MemoriaRAM.mif
    fMIF = open("MemoriaRAM.mif", "w")
    #Prenche MemoriaROM.vhd
  caseMemoriaROMDEC(fNeander, fROM)
  criaMIFDEC(fNeander, fMIF)

    fNeander.close()
    fROM.close()
    fMIF.close()

if __name__ == "__main__":
        main()
```