



UNIVERSIDADE FEDERAL DA BAHIA
ESCOLA POLITÉCNICA
Curso de Graduação em Engenharia de Computação

**Implementação e Validação de uma Plataforma para Jogos com
Gráficos 2D Embarcada em FPGA**

Lucas Vilas Boas Alves

Salvador
Julho de 2018

Lucas Vilas Boas Alves

Implementação e Validação de uma Plataforma para Jogos com Gráficos 2D Embarcada em FPGA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação da Universidade Federal da Bahia como parte dos requisitos para a obtenção do título de Engenheiro de Computação.

Universidade Federal da Bahia
Escola Politécnica
Curso de Graduação em Engenharia de Computação

Orientador: Prof. Dr. Wagner Luiz Alves de Oliveira

Salvador
Julho de 2018

Lucas Vilas Boas Alves

Implementação e Validação de uma Plataforma para Jogos com Gráficos 2D Embarcada em FPGA

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Engenheiro de Computação e aprovado em sua forma final pela Comissão Examinadora e pelo Colegiado do Curso de Graduação em Engenharia de Computação da Universidade Federal da Bahia.

Salvador, 27 de Julho de 2018

Wagner Luiz Alves de Oliveira, D.Sc.
(Orientador)

**Paulo César Machado de Abreu
Farias, D.Sc.**

Edmar Egidio Purcino de Souza, M.Sc.

Salvador
Julho de 2018

Aos meus pais, a quem tudo devo, dedico este trabalho.

Agradecimentos

Apesar de não serem suficientes para demonstrar minha gratidão, registro aqui algumas frases de agradecimento à todos aqueles que de alguma forma me auxiliaram na conclusão desta graduação, etapa tão importante de minha vida.

Agradeço a Deus, pela concessão da energia necessária, quando eu não imaginava ter, e pela coragem para prosseguir.

Um agradecimento especial a todos os meus familiares, por cada palavra de incentivo e por compreenderem minha ausência enquanto me dedicava a esta graduação. Em especial à minha mãe Eunice, ao meu pai Eduardo e às minhas irmãs, por estarem sempre cuidando de mim, me ajudando e sendo a base forte que eu preciso para seguir em frente. Obrigado Bianca por cada carona, e pelos momentos de descontração, sou grato por ter você em minha vida.

Agradeço à minha companheira Julia pelo carinho, paciência, e por acreditar em minha capacidade. Obrigado pela presença constante, sempre se esforçando para me ajudar, e pelos momentos de paz e risos em meio às turbulências de cada semestre.

Aos meus amigos, em especial ao Cambada, que mesmo quando distantes me trazem a certeza de não estar sozinho neste mundo.

Aos professores do curso que contribuíram durante a minha vida acadêmica, especialmente ao professor Wagner pela orientação e incentivo que tornaram este trabalho possível.

*“Nossa maior fraqueza está em desistir.
O caminho mais certo para vencer é tentar mais uma vez”
(Thomas Alva Edison)*

Resumo

Desde seus primórdios, na década de 1950, os jogos eletrônicos despertam a atenção das pessoas. Atualmente, este mercado movimenta bilhões de dólares, superando o faturamento de indústrias mais antigas e consolidadas, como a fonográfica e a cinematográfica. Na base dos consoles de jogos eletrônicos recentes estão circuitos digitais complexos, capazes de executar diversas funções e reproduzir gráficos realistas. O presente trabalho usa esses sistemas como referência e apresenta uma plataforma para a execução de jogos simples, com gráficos em duas dimensões, implementada sobre um kit de desenvolvimento em FPGA. São apresentadas as características do projeto, a montagem física do protótipo e a lógica desenvolvida para programar o FPGA. Por último são demonstrados os *softwares* elaborados e os testes realizados para validar o circuito.

Palavras-chave: Jogos Eletrônicos. Videogame. Circuito Digital. FPGA. Verilog HDL. IP Core.

Abstract

Since its beginnings, in the 1950s, electronic games attract people's attention. Today, this market moves billions of dollars, surpassing the revenues of older and consolidated industries, such as phonographic and cinematic. At the base of the recent gaming consoles are complex digital circuits, capable of performing several functions and reproducing realistic graphics. The following work uses as reference these systems and presents a platform for execute simple games, with graphics in two dimensions, implemented in a FPGA development board. The characteristics of the project, the prototype physical assembly and the logic developed for FPGA programming are presented. Finally, the software coded and the tests performed to validate the circuit are demonstrated.

Keywords: Electronic Games. Video Game. Digital Circuit. FPGA. Verilog HDL. IP Core.

Listas de ilustrações

Figura 1 – Dispositivos utilizados no jogo <i>Tennis for Two</i> .	16
Figura 2 – <i>Magnavox Odyssey</i> e acessórios que o acompanhavam.	17
Figura 3 – Estrutura conceitual de um dispositivo FPGA.	20
Figura 4 – Diagrama conceitual de uma célula lógica baseada em <i>Look-Up Table</i> .	21
Figura 5 – Composição de uma imagem por <i>background</i> e <i>sprites</i> .	23
Figura 6 – Arquitetura conceitual da plataforma para jogos.	25
Figura 7 – Interface do <i>software Terasic DE2-115 Control Panel</i> .	27
Figura 8 – Dispositivos usados na montagem do protótipo.	28
Figura 9 – Detalhe das conexões entre o controle de <i>Sega Mega Drive</i> e a placa <i>DE2-115</i> .	28
Figura 10 – Diagrama dos módulos implementados no FPGA.	29
Figura 11 – Conexões entre os <i>slide switches</i> e o FPGA na placa DE2-115.	31
Figura 12 – Representações do módulo <i>Reset_Synchronizer</i> .	32
Figura 13 – Representação em bloco do módulo <i>Memory_Arbiter</i> .	34
Figura 14 – Modelos de arquitetura para acesso à memória.	35
Figura 15 – Diagrama de estados do módulo <i>Memory_Arbiter</i> .	35
Figura 16 – Representação em bloco do módulo <i>Interrupt_Controller</i> .	37
Figura 17 – Diagrama de estados do módulo <i>Interrupt_Controller</i> .	38
Figura 18 – Primeira etapa do <i>waveform</i> gerado pelo módulo <i>Interrupt_Controller</i> .	38
Figura 19 – Segmentos da memória de dados e a máscara de interrupção.	39
Figura 20 – Segmentos da memória de programa e o vetor de interrupções.	40
Figura 21 – Segunda etapa do <i>waveform</i> gerado pelo módulo <i>Interrupt_Controller</i> .	41
Figura 22 – Terceira etapa do <i>waveform</i> gerado pelo módulo <i>Interrupt_Controller</i> .	42
Figura 23 – <i>Waveform</i> com temporizações do sinal SELECT.	43
Figura 24 – Representação em bloco do módulo <i>Genesis_6button_Interface</i> .	44
Figura 25 – Segmentos da memória de dados e os estados dos botões do controle.	44
Figura 26 – Diagrama de estados do módulo <i>Genesis_6button_Interface</i> .	46
Figura 27 – Conexões entre a memória SRAM e o FPGA na placa DE2-115.	47
Figura 28 – Representação em bloco do módulo <i>SRAM_Interface</i> .	48
Figura 29 – Diagrama simplificado das conexões do módulo <i>SRAM_Interface</i> .	48
Figura 30 – Construção de uma imagem.	49
Figura 31 – Especificações de tempo na varredura de uma imagem.	50
Figura 32 – Conexões entre o FPGA e a saída de vídeo VGA na placa DE2-115.	52
Figura 33 – Representação em bloco do módulo <i>VGA_Interface</i> .	53
Figura 34 – Construção e posicionamento de um <i>sprite</i> .	55
Figura 35 – Representação em bloco do módulo <i>Sprite_Shape_Reader</i> .	56

Figura 36 – Segmentos da memória de dados e os <i>shapes</i> dos <i>sprites</i>	56
Figura 37 – Diagrama de estados do módulo <i>Sprite_Shape_Reader</i>	57
Figura 38 – Representação em bloco do módulo <i>Sprite_Processor</i>	58
Figura 39 – Representação em bloco do módulo <i>Processor_Controller</i>	60
Figura 40 – Diagrama de estados do módulo <i>Processor_Controller</i>	61
Figura 41 – Representação do registrador de status <i>RFlags</i>	61
Figura 42 – Formatos das instruções executadas pelo <i>Processor_Controller</i>	62
Figura 43 – Representações dos módulos <i>IP_ADD</i> e <i>IP_SUB</i>	68
Figura 44 – Representação em bloco do módulo <i>IP_MULT</i>	70
Figura 45 – Representação em bloco do módulo <i>IP_DIVIDE</i>	72
Figura 46 – Representação em bloco do módulo <i>IP_COMPARE</i>	76
Figura 47 – Representação em bloco do módulo <i>IP_ROM_Program</i>	78
Figura 48 – Representação em bloco do módulo <i>IP_RAM_Data</i>	79
Figura 49 – Representação em bloco do módulo <i>IP_PLL</i>	80
Figura 50 – Foto do protótipo montado.	84

Lista de quadros

Quadro 1 – Estados e sinais para leitura do controle de 6 botões do <i>Sega Mega Drive</i>	43
Quadro 2 – Descrição das instruções de transferência de dados	63
Quadro 3 – Descrição das instruções aritméticas	64
Quadro 4 – Descrição das instruções lógicas	65
Quadro 5 – Descrição das instruções gráficas	65
Quadro 6 – Descrição das instruções de transferência de controle	67
Quadro 7 – Tabela verdade do <i>IP core LPM_DIVIDE</i> : LPM_REMAINDERPOSITIVE = ‘TRUE’	73
Quadro 8 – Tabela verdade do <i>IP core LPM_DIVIDE</i> : LPM_REMAINDERPOSITIVE = ‘FALSE’	74

Lista de tabelas

Tabela 1 – Portas da memória SRAM	47
Tabela 2 – Portas de entrada dos módulos <i>IP_ADD</i> e <i>IP_SUB</i>	69
Tabela 3 – Portas de saída dos módulos <i>IP_ADD</i> e <i>IP_SUB</i>	69
Tabela 4 – Parâmetros do <i>IP core LPM_ADD_SUB</i> utilizados no somador	69
Tabela 5 – Parâmetros do <i>IP core LPM_ADD_SUB</i> utilizados no subtrator	70
Tabela 6 – Portas de entrada do módulo <i>IP_MULT</i>	71
Tabela 7 – Porta de saída do módulo <i>IP_MULT</i>	71
Tabela 8 – Parâmetros do <i>IP core LPM_MULT</i> utilizados no multiplicador	71
Tabela 9 – Portas de entrada do módulo <i>IP_DIVIDE</i>	72
Tabela 10 – Portas de saída do módulo <i>IP_DIVIDE</i>	73
Tabela 11 – Parâmetros do <i>IP core LPM_DIVIDE</i> utilizados no divisor	75
Tabela 12 – Portas de entrada do módulo <i>IP_COMPARE</i>	76
Tabela 13 – Portas de saída do módulo <i>IP_COMPARE</i>	77
Tabela 14 – Parâmetros do <i>IP core LPM_COMPARE</i> utilizados no comparador	77
Tabela 15 – Parâmetros do <i>IP core ALTSYNCRAM</i> utilizados na memória ROM	78
Tabela 16 – Parâmetros do <i>IP core ALTSYNCRAM</i> utilizados na memória RAM	79
Tabela 17 – Parâmetros do <i>IP core ALTPPLL</i> utilizados no PLL	81

Lista de abreviaturas e siglas

VLSI	Very Large Scale Integration
HDL	Hardware Description Language
EDA	Electronic Design Automation
FPGA	Field-Programmable Gate Array
ASIC	Application-Specific Integrated Circuit
LUT	Look-Up Table
RTL	Register-Transfer Level
IEEE	Institute of Electrical and Electronics Engineers
VGA	Video Graphics Array
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RAM	Random-Access Memory
SRAM	Static Random-Access Memory
ISR	Interrupt Service Routine
PLL	Phase-Locked Loop
DAC	Digital-to-Analog Converter
USB	Universal Serial Bus
FSM	Finite State Machine
MIF	Memory Initialization File
DUT	Design Under Test

Listas de símbolos

\mathbb{Z} Conjunto dos números inteiros

\mathbb{Z}^* Conjunto dos inteiros não-nulos

\in Pertence

Sumário

1	INTRODUÇÃO	16
2	FUNDAMENTAÇÃO TEÓRICA	20
2.1	FPGA	20
2.2	Ferramentas EDA e Verilog HDL	21
3	IMPLEMENTAÇÃO DO PROTÓTIPO	23
3.1	Especificação do Projeto	23
3.2	Montagem do Protótipo	26
3.3	Módulos Implementados	30
3.3.1	Sincronizador de Reset	30
3.3.2	Árbitro de Memória	32
3.3.3	Controlador Programável de Interrupções	36
3.3.4	Interface para Controle	42
3.3.5	Interface para Memória SRAM	45
3.3.6	Interface de Vídeo VGA	48
3.3.7	Leitor de Sprites	54
3.3.8	Processador de Sprites	58
3.3.9	Unidade de Controle do Processador	58
3.4	IP cores Utilizados	68
3.4.1	Somador/Subtrator	68
3.4.2	Multiplicador	70
3.4.3	Divisor	72
3.4.4	Comparador	76
3.4.5	Memória ROM	77
3.4.6	Memória RAM	78
3.4.7	PLL	79
3.5	Softwares Desenvolvidos	80
3.5.1	Conversor de Imagens Bitmap	80
3.5.2	Assembler	82
4	VALIDAÇÃO DO PROTÓTIPO	83
5	CONCLUSÃO	85
	REFERÊNCIAS	86

1 Introdução

Considerado um dos primeiros jogos eletrônicos da história, *Tennis for Two*, também conhecido como *Tennis Programming*, surgiu em 1958 criado pelo físico William Higinbotham no laboratório estadunidense *Brookhaven National Laboratory*. Era um jogo muito simples, jogado por meio de um osciloscópio e processado por um computador analógico. Uma quadra de tênis vista lateralmente era simulada, com a bola sendo rebatida em uma linha horizontal na parte inferior da tela, e uma linha vertical no centro representando a rede. Não havia placar, e a tela era o cinescópio de fósforo verde monocromático de um osciloscópio. Duas caixas controlavam o jogo, cada uma contendo um potenciômetro e um botão. Os potenciômetros afetavam o ângulo da bola, que era rebatida de volta para o outro lado da tela através do botão. A bola caia na rede quando o jogador errava o ângulo. Um botão de *reset* permitia o reinício do jogo, fazendo com que a bola reaparecesse do lado oposto da tela (BATISTA et al., 2007). A Figura 1 ilustra os instrumentos usados para jogar *Tennis for Two*.



(a) *Tennis for Two* em um osciloscópio.



(b) Recriação moderna do controlador.

Figura 1 – Dispositivos utilizados no jogo *Tennis for Two*.

Fonte: ([WIKIPEDIA](#), 2018)

Em 1972 é lançado o *Magnavox Odyssey*, idealizado por Ralph Baer e apontado como o primeiro console caseiro criado. O equipamento oferecia doze jogos, armazenados em placas de circuito impresso distintas, permitindo ao usuário escolher qual ele queria jogar (BATISTA et al., 2007). Acompanhavam o console folhas de papel padronizadas, para anotar o placar do jogo, e cartões plásticos coloridos chamados de *overlay*, que deveriam ser fixados na frente da tela da televisão para simular o espaço do jogo. Em uma partida de tênis, por exemplo, um plástico verde deveria ser fixado para parecer com grama (REIS,

2005). Grande parte dos jogos tinha esportes como tema, mas outros gêneros também foram desenvolvidos, como jogos de tiro. Estes eram oferecidos separadamente e traziam um rifle, do tipo *light gun*, para ser usado como periférico do console (BATISTA et al., 2007).

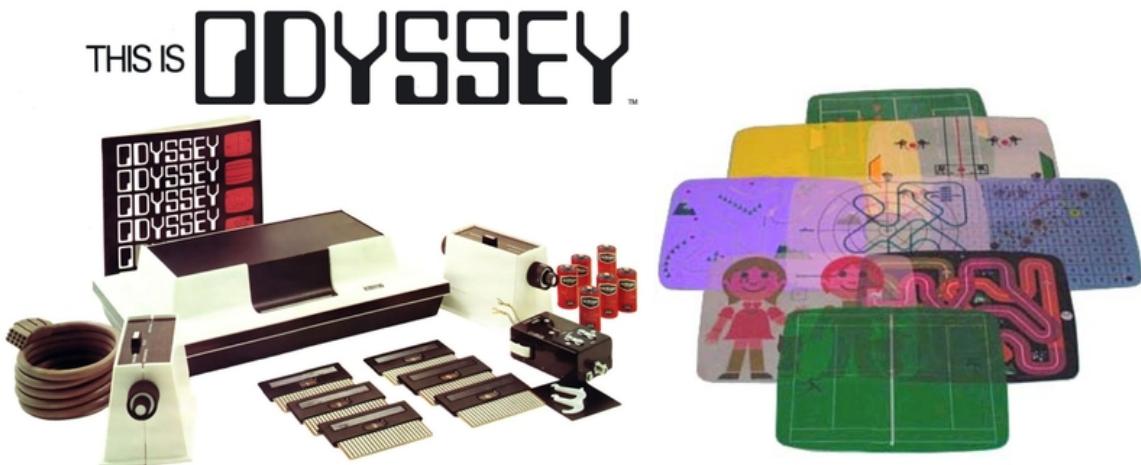


Figura 2 – *Magnavox Odyssey* e acessórios que o acompanhavam.

Fonte: (AMARAL, 2018)

O *Magnavox Odyssey* produzia imagens que podem ser consideradas hoje como de baixa resolução, pois exibia na tela apenas elementos geométricos simples, como quadrados. Os componentes, personagens e o enredo dos jogos não eram representados de forma realista, além de não possuir som. Contudo, este aparelho fez muito sucesso, vendendo cerca de 100 mil consoles e 20 mil rifles (BATISTA et al., 2007).

Hoje, o mercado mundial de jogos eletrônicos movimenta bilhões de dólares, superando o faturamento de indústrias consolidadas, como a fonográfica e a cinematográfica. Em 2015 as receitas mundiais com música gravada somaram US\$ 15 bilhões, enquanto que os 100 filmes mais lucrativos lançados em 2016 renderam, juntos, US\$ 25.6 bilhões. Neste mesmo ano, os jogos eletrônicos arrecadaram US\$ 91 bilhões, considerando todos os dispositivos e plataformas (HARADA, 2018). Contudo, este ainda é um segmento em franco crescimento, inclusive no Brasil. Em 2017 a receita mundial cresceu para US\$108.9 bilhões, e o país obteve uma renda de US\$1.334 bilhões, figurando na décima terceira posição do ranking de países com maior faturamento por jogos eletrônicos (NEWZOO, 2017).

A indústria de *games* pode ser segmentada em plataformas como computadores pessoais, dispositivos móveis e consoles. Esta última, em 2017, foi responsável por uma fatia de aproximadamente 31% do mercado, ou o equivalente a US\$33.5 bilhões, um número bastante expressivo ([NEWZOO, 2017](#)). Este segmento conta atualmente com equipamentos avançados, como o *Sony PlayStation 4* ou o *Microsoft Xbox One*, que além de executarem jogos com gráficos ultra realistas, são capazes de acessar a internet e reproduzir recursos multimídia. A diversidade de funções oferecidas por estes aparelhos, em um tamanho reduzido, é devida em grande parte à evolução da tecnologia no desenvolvimento de circuitos digitais.

O progresso das técnicas de fabricação dos circuitos integrados possibilitaram uma miniaturização crescente dos dispositivos eletrônicos, permitindo a combinação em um único *chip* de dezenas de bilhões destes elementos, construídos em escala nanométrica ([NVIDIA, 2017](#)). A tecnologia empregada na concepção de circuitos altamente densos é chamada de VLSI (*Very Large Scale Integration*). Circuitos integrados podem se tornar bastante complexos, e a utilização de métodos de projeto e verificação assistidos por computador são essenciais para o desenvolvimento desses sistemas.

A utilização de HDLs (*Hardware Description Languages*), sigla em inglês para linguagens de descrição de *hardware*, e dos softwares de EDA (*Electronic Design Automation*), permitem automatizar a síntese dos circuitos e possibilitam a verificação funcional destes. O processo de fabricação de circuitos integrados tem alto custo e, depois de produzidos, a estrutura interna destes componentes não pode ser modificada. Assim, a validação do projeto é etapa fundamental antes da confecção do *chip*.

No caso particular dos circuitos digitais é possível simular fisicamente projetos sofisticados, implementando-os em FPGAs (*Field-Programmable Gate Arrays*) e verificando como operam em um dispositivo real. O uso de protótipos com FPGAs é útil para detectar e corrigir erros no projeto, reduzindo as chances de problemas após a construção final do circuito integrado. Dispositivos FPGA são úteis também em aplicações digitais onde pode haver a necessidade de reconfiguração da lógica interna do circuito, seja para a atualização de protocolos ou para melhorias de desempenho, por exemplo, tornando inviável a utilização de ASICs (*Application-Specific Integrated Circuits*).

Com o objetivo de aprofundar o conhecimento nas etapas envolvidas no projeto de um circuito digital, buscou-se implementar em FPGA uma plataforma para jogos com gráficos em duas dimensões. O projeto embarcado no FPGA foi elaborado com a plataforma *Quartus Prime Lite*, fornecida pela *Intel*, e simulado com o apoio do software *ModelSim - INTEL FPGA STARTER EDITION*, produzido pela *Mentor Graphics*. O contato com estas ferramentas proporcionou o desenvolvimento de novas habilidades ligadas à operação de softwares de EDA, além de propiciar uma maior aptidão no uso de *Verilog HDL*, linguagem de descrição de *hardware* escolhida para elaborar os módulos do

projeto.

Competências ligadas à construção de *softwares* também puderam ser evoluídas a partir da criação de um *Assembler* e de uma aplicação para a conversão de imagens, produzidas nas linguagens *Python* e *MATLAB*, respectivamente. Por fim, conhecimentos de eletrônica precisaram ser aplicados durante a montagem do protótipo na placa de desenvolvimento *Altera DE2-115*, permitindo o uso correto dos periféricos da placa e a integração de dispositivos externos adequadamente.

2 Fundamentação Teórica

Este capítulo descreve as tecnologias e conceitos centrais utilizados durante a concepção do projeto. As definições apresentadas são embasadas no material bibliográfico revisado, que serviu de apoio no desenvolvimento de um trabalho fundamentado nas teorias existentes.

2.1 FPGA

O termo FPGA (*Field-Programmable Gate Array*), que pode ser traduzido livremente como Matriz de Portas Programáveis no Local, se refere a um circuito integrado contendo uma matriz bidimensional de células lógicas genéricas e comutadores programáveis. Cada uma destas células pode ser programada para executar funções lógicas simples, enquanto que os comutadores são capazes de fornecer interconexões entre as células lógicas. A partir da especificação da função de cada célula lógica, e das conexões configuradas em cada comutador programável, é possível implementar um circuito personalizado.

Com o projeto e a síntese do circuito disponíveis, a configuração das células lógicas e comutadores pode ser carregada no FPGA, que passará a trabalhar conforme o circuito projetado. Dado que esta implementação pode ser feita no ambiente de operação, ao invés de em uma fábrica, o dispositivo é dito programável no local (*Field-Programmable*) ([CHU, 2008](#)). O conceito da estrutura de um dispositivo FPGA é exibido na [Figura 3](#).

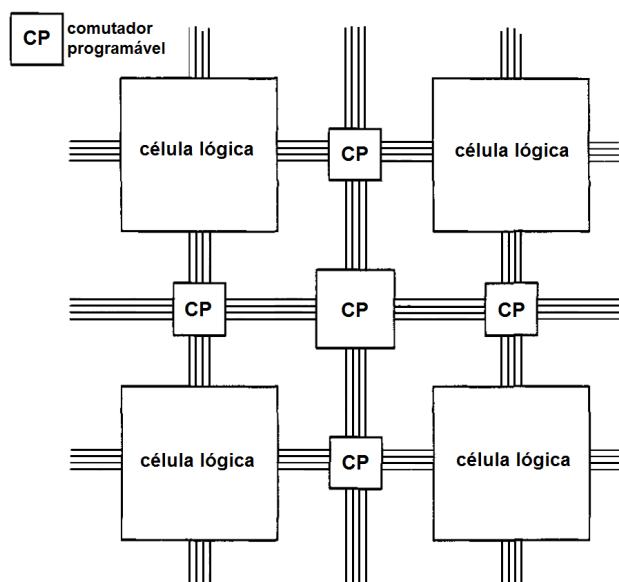


Figura 3 – Estrutura conceitual de um dispositivo FPGA.

Fonte: Adaptado de ([CHU, 2008](#))

As células lógicas geralmente são formadas por um pequeno circuito combinacional configurável associado a um *flip-flop* tipo D. O circuito combinacional configurável é comumente implementado através de uma tabela de consulta, conhecida como LUT (*Look-Up Table*). Uma LUT com n entradas funciona como uma pequena memória, de 2^n endereços por 1 bit de largura, e segue o conceito de uma tabela verdade, produzindo uma saída conforme o endereço formado pelos valores das n entradas. Assim, qualquer função combinacional pode ser implementada através de uma LUT, dependendo apenas do conteúdo gravado em sua memória. O diagrama conceitual de uma célula lógica de três entradas baseada em *Look-Up Table* é mostrado na Figura 4. É possível observar que a saída da LUT pode ser usada diretamente ou armazenada no *flip-flop* tipo D, empregado na implementação de circuitos sequenciais (CHU, 2008).

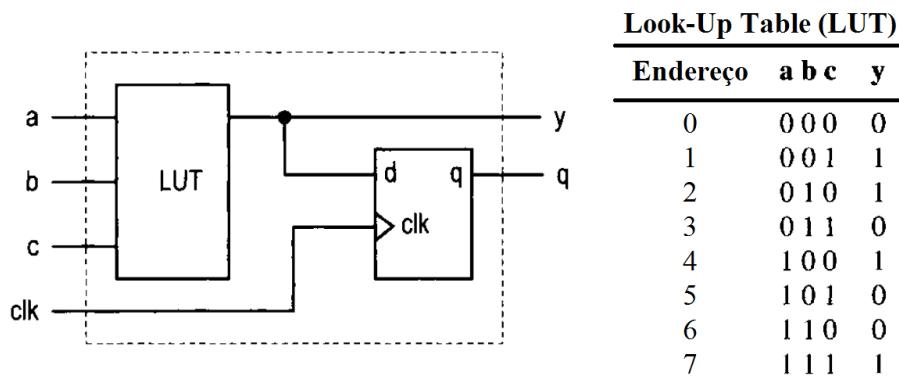


Figura 4 – Diagrama conceitual de uma célula lógica baseada em *Look-Up Table*.

Fonte: Adaptado de (CHU, 2008)

A maioria dos dispositivos FPGA também incorpora algumas macro-células, ou macro-blocos, projetadas e fabricadas em nível de transistor, com funcionalidades que complementam as células lógicas genéricas. As macro-células mais comuns incluem blocos de memória, multiplicadores combinacionais, circuitos de gerenciamento de *clock* e circuitos de interface de entrada e saída. Dispositivos FPGA avançados podem até embutir um ou mais núcleos de processador pré-fabricados (CHU, 2008).

2.2 Ferramentas EDA e Verilog HDL

O projeto de circuitos digitais evoluiu rapidamente nas últimas décadas. O advento da tecnologia VLSI (*Very Large Scale Integration*) tornou possível o projeto de *chips* com mais de 100.000 transistores. Devido à complexidade desses circuitos, técnicas de verificação e projeto assistidas por computador tornaram-se essenciais no desenvolvimento de circuitos digitais com altíssima escala de integração.

Atualmente são comuns *softwares* para posicionamento e roteamento automático dos componentes eletrônicos, e os simuladores lógicos surgiram para viabilizar a verificação funcional desses circuitos antes de serem fabricados em um *chip*. Com os projetos se tornando cada vez maiores e mais complexos, a simulação lógica assumiu um papel importante no processo de desenvolvimento, possibilitando a correção de falhas funcionais da arquitetura antes de prosseguir com a elaboração do *chip* (PALNITKAR, 2003). Estas ferramentas de auxílio no projeto de sistemas eletrônicos, como circuitos integrados, são conhecidas como EDA (*Electronic Design Automation*).

As linguagens de descrição de *hardware*, chamadas também de HDLs (*Hardware Description Languages*), surgiram da necessidade de uma linguagem para descrever circuitos digitais. As HDLs permitem modelar a simultaneidade de processos encontrada em elementos de *hardware*, em oposição às linguagens de programação convencionais, que são de natureza sequencial.

A chegada da síntese lógica no final da década de 1980 possibilitou o uso de HDLs na síntese de circuitos digitais descritos em nível RTL (*Register-Transfer Level*). Isto permitiu o desenvolvimento de projetos complexos descrevendo apenas como o circuito processa os dados e como estes fluem entre os registradores. Os detalhes das portas lógicas, e de suas interconexões para implementar o circuito, são automaticamente extraídos da descrição RTL por ferramentas de síntese lógica dos *softwares* de EDA.

Entre as linguagens de descrição de hardware o *Verilog HDL* e o *VHDL* figuram entre as mais populares. O *Verilog HDL* surgiu em 1983 na *Gateway Design Automation* e hoje é um padrão IEEE aceito (PALNITKAR, 2003). Em 1995 o padrão original IEEE 1364-1995 foi aprovado, sendo atualizado em seguida nos anos de 2001, 2005 e 2009, com melhorias acrescentadas ao padrão original.

3 Implementação do Protótipo

Este capítulo descreve a elaboração da plataforma para jogos 2D. Inicialmente é feita uma pequena introdução sobre o funcionamento do projeto e são especificadas suas principais características. Posteriormente é exposto o procedimento de conexão dos elementos de interface com a placa de prototipagem *DE2-115*, tais como o controle para console do video game *Sega Mega Drive* e o monitor de vídeo VGA. Em seguida são detalhados os módulos de descrição de *hardware* implementados em *Verilog HDL* e são identificados os *IP cores* utilizados, ambos empregados na configuração do *FPGA*. Finalmente, são apresentados os elementos de *software* concebidos durante este trabalho.

3.1 Especificação do Projeto

Apesar de composta por diversos módulos, a plataforma elaborada pode ser dividida conceitualmente em dois segmentos, chamados de processador gráfico e processador de propósito geral. O processador gráfico é responsável por produzir as imagens exibidas pelo protótipo. Estas imagens são compostas basicamente por uma grande figura de fundo fixa, denominada *background*, e pequenos elementos gráficos móveis que sobrepõem a figura de fundo, designados como *sprites*. Estes dois componentes, usados na construção da imagem, são suficientes para a criação de cenários, placares, personagens e outros recursos necessários no enredo de um jogo simples, com gráficos em duas dimensões.

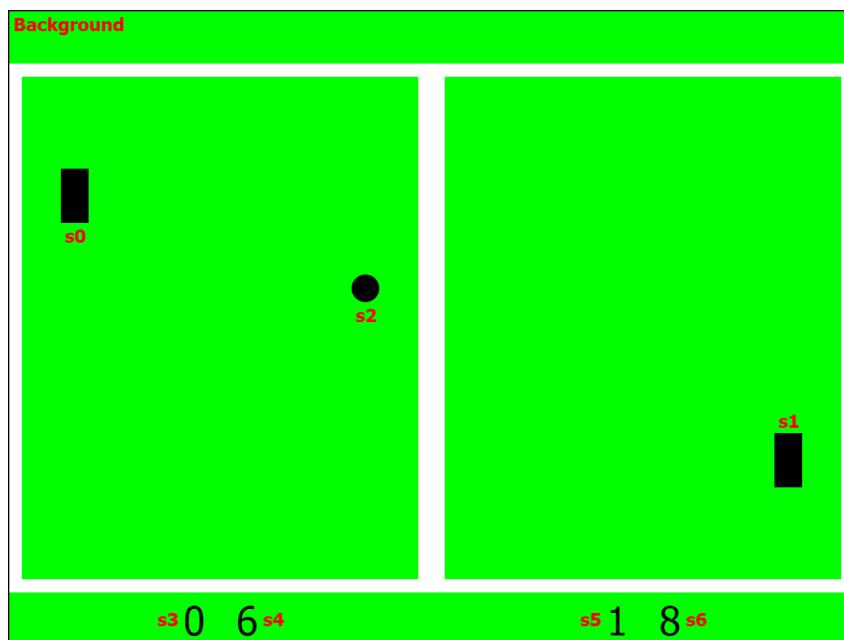


Figura 5 – Composição de uma imagem por *background* e *sprites*.

A [Figura 5](#) ilustra a composição de uma imagem através de um *background* simples, contendo a visão superior de um campo de tênis com piso de grama, e *sprites* na cor preta, representando dois jogadores, os placares de cada um, e a bola. É possível observar que neste cenário foram utilizados 7 *sprites*, nomeados de *s0* a *s6*. A figura de fundo exibida durante o jogo deve ser carregada previamente na memória da plataforma, enquanto que as formas dos *sprites* podem ser definidas em tempo de execução, através do processador de propósito geral, ou também inicializadas na memória antecipadamente. São destacados abaixo os principais atributos do processador gráfico:

- Exibição de imagens com profundidade de cor de 16 bits (*High Color*), resolução de 640x480 *pixels* e taxa de atualização de 60 Hz;
- Exibição de elementos gráficos móveis (*sprites*), com tamanho de 16x16 *pixels*, cores com 16 bits de profundidade (*High Color*) e possibilidade de assumir qualquer formato;
- Imagem de fundo (*background*) carregada na memória antecipadamente, no momento da programação da plataforma;
- Formatos dos *sprites*, também chamados de *shapes*, podem ser programados através do processador de propósito geral ou inicializados previamente na memória;
- Exibição de até 16 *sprites* simultaneamente na tela, em diferentes níveis de sobreposição, e com capacidade de todos se sobrepor em concomitância.

Os 16 bits utilizados na representação das cores são divididos entre três componentes primários. São reservados 5 bits para as componentes vermelha e azul, e 6 bits para a componente verde. Este padrão foi adotado por conta da maior sensibilidade do olho humano ao verde e, portanto, a resolução adicional proporcionada pelo bit extra é melhor captada se usada neste elemento.

O processador de propósito geral tem como função executar o algoritmo do jogo, que coordena seu comportamento através de instruções lógicas, aritméticas, gráficas, de transferência de dados e de transferência de controle. O código do jogo deve ser salvo na memória no momento da programação da plataforma, visto que seu funcionamento se baseia em seguir as instruções armazenadas. As informações do controle manipulado pelo jogador também são interpretadas por esta etapa do protótipo, que reage conforme programado pelo desenvolvedor do jogo. As características centrais deste processador são resumidas nos pontos a seguir:

- Operação com *clock* de 50 MHz;
- Arquitetura RISC (*Reduced Instruction Set Computer*);

- Instruções de 32 bits;
- Instruções gráficas para manipulação dos 16 *sprites* e para sincronismo com a exibição das imagens;
- Execução serial de instruções, sem uso de *pipeline*;
- Operações internas em 16 bits;
- Operações aritméticas em ponto fixo, apenas com números inteiros;
- 32 registradores de 16 bits com propósito geral, definidos como *R0* a *R31*;
- Registrador especial de 8 bits, designado *RFlags*, para armazenamento de sinalizações (*flags*) referentes à operação do processador;
- Modos de endereçamento: a registrador, imediato e base-deslocamento.

Os dois processadores descritos exercem funções complementares e trabalham de maneira unificada, estando inseridos em uma estrutura maior. A Figura 6 demonstra de forma conceitual a arquitetura elaborada para a plataforma de jogos.

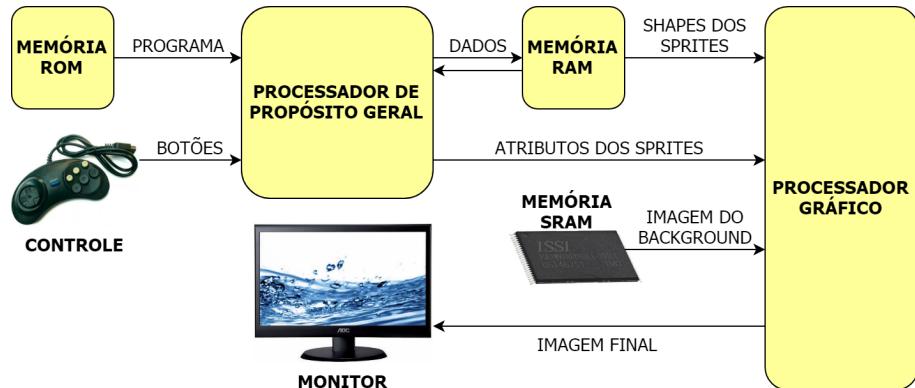


Figura 6 – Arquitetura conceitual da plataforma para jogos.

Esta arquitetura é detalhada nas seções subsequentes e suas especificações são resumidas nos tópicos a seguir:

- Arquitetura *Harvard*, com memórias e barramentos distintos para dados e instruções do processador de propósito geral;
- Memória de programa do tipo ROM (*Read-Only Memory*), com 65.536 endereços e 32 bits de largura. Utilizada para armazenar as instruções do algoritmo que o processador de propósito geral deve executar;

- Memória de dados do tipo RAM (*Random-Access Memory*), com 65.536 endereços e 16 bits de largura. Utilizada para armazenar *shapes* dos *sprites*, registradores de comunicação de entrada e saída, máscaras de interrupção e dados em geral;
- Arbitragem do acesso à memória de dados, devido ao seu compartilhamento entre diversos módulos;
- Memória do tipo SRAM (*Static Random-Access Memory*) para armazenamento de 4.800 kb da imagem de fundo (*background*);
- Comunicação via mecanismo de interrupção, com controle de mascaramento de interrupções através da memória de dados compartilhada;
- Vetor de interrupções na memória de programa para atender até 4 ISRs (*Interrupt Service Routines*);
- Controlador dedicado para o dispositivo de controle externo, com comunicação baseada em memória compartilhada, no qual os registradores de dados são mapeados em endereços da memória de dados.

3.2 Montagem do Protótipo

O protótipo produzido usou como base a placa de desenvolvimento *Altera DE2-115*, que tem como principal recurso um FPGA *Altera* da série *Cyclone IV E*. Este dispositivo, de modelo EP4CE115F29, conta com 114.480 elementos lógicos, 3.888 kb de memória embarcada, 4 PLLs (*Phase-Locked Loops*) e 266 multiplicadores de 18 x 18 bits. No projeto implementado, o circuito sintetizado e carregado no FPGA usou parcialmente todos estes recursos.

A *Altera DE2-115* possui ainda uma série de outros componentes úteis agregados ao FPGA, entretanto, apenas uma pequena parcela precisou ser explorada neste projeto. Os recursos empregados no protótipo são listados abaixo:

- **Memória SRAM:** É disponibilizado um *chip* com 1.048.576 endereços e 16 bits de largura, totalizando 2 MB de capacidade. Esta memória é utilizada para armazenar a imagem de fundo (*background*), entretanto, apenas 4.800 kb são consumidos (640 x 480 *pixels* x 16 bits);
- **Slide Switches:** A *DE2-115* disponibiliza 18 *slide switches*, mas apenas o SW17 foi aproveitado, usado na produção do sinal de *reset* para o circuito sintetizado no FPGA;
- **Oscilador de 50MHz:** Utilizado como fonte de *clock* para o FPGA;

- **Conversores Digital-Analógico e conector de saída VGA:** Um *chip*, com três DACs (*Digital-to-Analog Converters*) integrados, é aplicado na placa para converter as saídas digitais do FPGA em sinais analógicos de vídeo, que são direcionados ao conector de saída VGA. Esta saída de vídeo fornece as imagens produzidas pela plataforma e deve ser ligada a um monitor com entrada VGA;
- **Conector de Expansão de 40 pinos:** Permite a conexão de dispositivos externos ao FPGA. O conector possui 36 pinos conectados diretamente ao FPGA, além de fornecer alimentação com tensões de 3,3V e 5V, e dois pinos de terra (GND). O controle manipulado pelo jogador é ligado ao protótipo através deste conector.

O primeiro passo para utilizar a plataforma é carregar a imagem do *background* na memória SRAM, o que é feito através do *software Terasic DE2-115 Control Panel*, fornecido pelo fabricante da placa e ilustrado na [Figura 7](#). Este programa carrega um circuito no FPGA que permite, através de sua interface, o controle de diversos componentes da placa. Entre os dispositivos controlados está a memória SRAM, que pode ser inicializada através de um arquivo de texto com extensão *.hex* inserido no *software*.



Figura 7 – Interface do *software Terasic DE2-115 Control Panel*.

O arquivo não possui nenhuma formatação específica, os dados devem estar representados em notação hexadecimal e escritos de forma contínua na mesma sequência em que serão salvos na memória, isto é, do menor endereço para o maior. Como a memória possui uma largura de 16 bits, cada grupo de 4 caracteres hexadecimais representa os dados salvos em um endereço. Logo, os primeiros 4 caracteres do arquivo serão salvos na primeira posição da memória, os 4 caracteres seguintes serão armazenados na segunda posição, e assim por diante.

Devido ao funcionamento do *software Terasic DE2-115 Control Panel* depender de um circuito carregado no FPGA, a escrita da imagem de *background* na memória SRAM deve ser efetuada previamente, antes de configurar o FPGA com o circuito da plataforma

de jogos. Outro ponto de observação está na volatilidade da memória SRAM e, portanto, na perda do conteúdo se a placa for desligada.

Conforme demonstrado na [Figura 8](#), a placa *DE2-115* deve ser conectada em um monitor com entrada de vídeo VGA, e com um controle do console *Sega Mega Drive*, por meio dos quais o usuário interage com o jogo. A conexão do monitor com a placa é feita por meio de um cabo padrão VGA simples. Por outro lado, a ligação do controle com a *DE2-115* precisa ser feita através de um cabo adaptado, construído conforme as informações indicadas na [Figura 9](#).

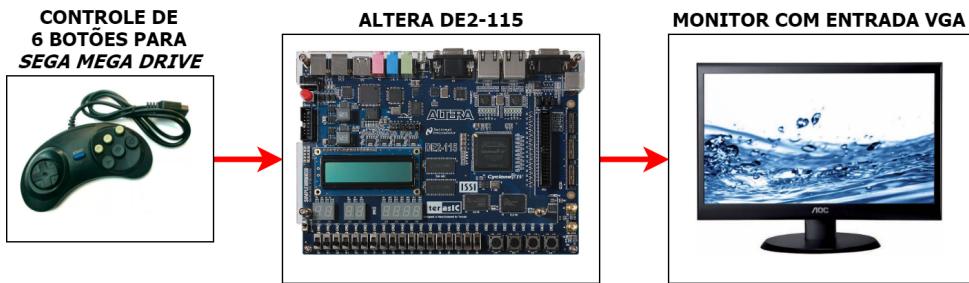


Figura 8 – Dispositivos usados na montagem do protótipo.

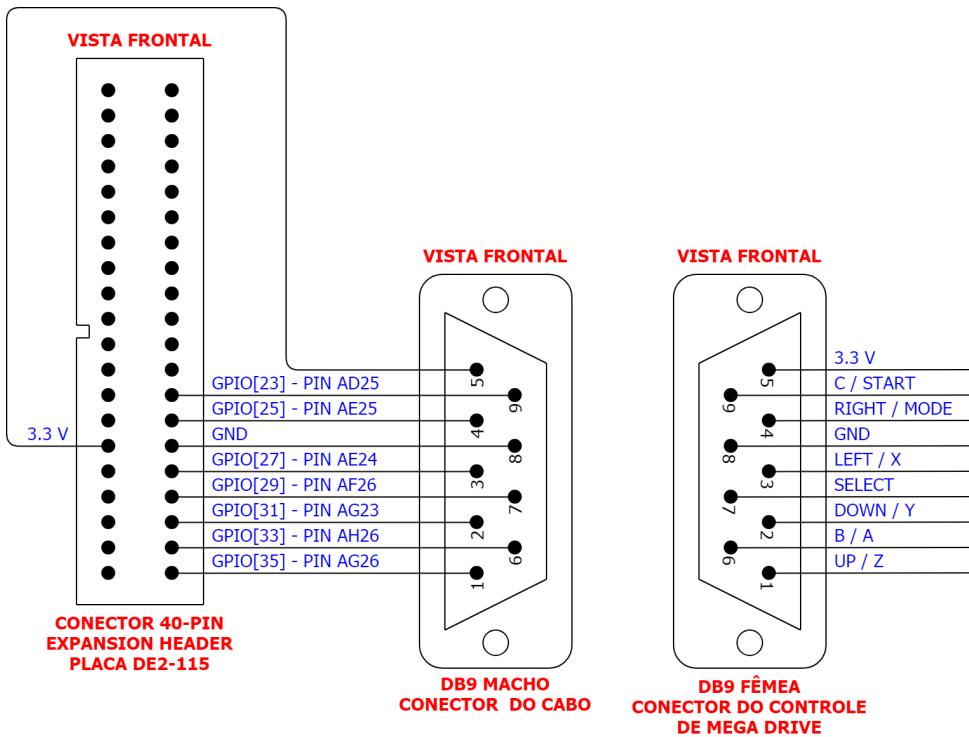


Figura 9 – Detalhe das conexões entre o controle de *Sega Mega Drive* e a placa *DE2-115*.

A configuração do FPGA é feita através de um cabo USB, conectando a placa *DE2-115* ao computador, e da ferramenta *Programmer*, disponível no *software Quartus Prime Lite*. Um diagrama dos módulos implementados no FPGA é apresentado na [Figura 10](#), e seus blocos serão detalhados na [seção 3.3](#) e na [seção 3.4](#).

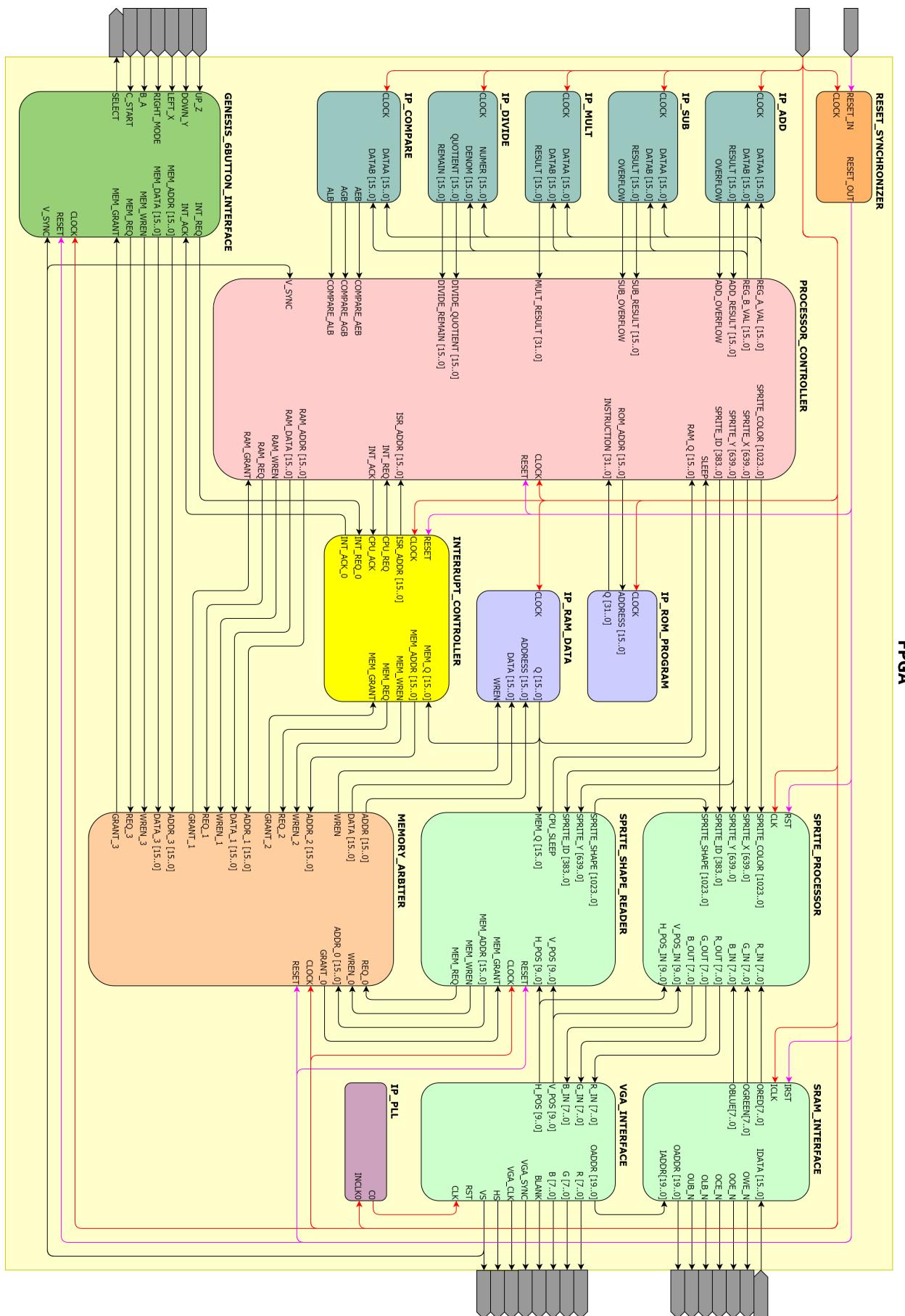


Figura 10 – Diagrama dos módulos implementados no FPGA.

3.3 Módulos Implementados

Para atender aos requisitos do projeto e viabilizar a arquitetura desejada, foi necessário o desenvolvimento de alguns módulos específicos cujas atividades não poderiam ser desempenhadas pelos *IP cores* disponíveis. A implementação destes módulos foi realizada no *software Quartus Prime Lite*, utilizando a linguagem de descrição de *hardware Verilog HDL*, e o funcionamento de cada um deles é descrito a seguir.

3.3.1 Sincronizador de Reset

O principal objetivo do sinal de *reset* em um circuito digital é forçá-lo para um estado conhecido. Existem algumas estratégias de *reset* que podem ser adotadas no projeto destes circuitos e suas características devem ser consideradas antes de definir qual a forma mais adequada. De modo geral existem dois modelos básicos, o *reset* síncrono e o *reset* assíncrono.

O *reset* assíncrono incorpora nos *flip-flops* um pino de *reset* que quando ativo, seja em nível alto ou baixo, faz com que estes elementos de memória sejam redefinidos para o estado de *reset*, sem levar em consideração o sinal de *clock*. Em alguns projetos esta independência do *clock* é uma vantagem, tornando-se por vezes uma necessidade. Uma outra vantagem é que o caminho percorrido pelos dados fica livre da lógica adicional que é inserida para tratar o *reset* síncrono. Isto reduz os atrasos no trajeto dos dados e permite atender a requisitos de temporização mais críticos.

O maior problema com os *reset*s assíncronos é que tanto a sua ativação quanto sua desativação são assíncronos e, caso haja uma mudança no nível do sinal de *reset* na região próxima à borda ativa do *clock*, os *flip-flops* podem ser levados para um estado de metaestabilidade, deixando o circuito em um estado desconhecido. Outra desvantagem são os *reset*s espúrios devido a ruídos ou oscilações no sinal, dependendo de sua fonte. Por fim, devido aos diferentes atrasos nos caminhos percorridos pelos sinais em um circuito digital, há a possibilidade dos *flip-flops* entrarem e saírem do estado de *reset* em momentos distintos, o que pode levar o circuito a se comportar de uma maneira não esperada.

O *reset* síncrono é baseado na premissa de que o sinal de *reset* só afetará ou redefinirá o estado dos *flip-flops* na borda ativa do sinal de *clock*. Uma das vantagens desta estratégia é que o *clock* funciona como um filtro para pequenas oscilações no sinal de *reset* que venham a ocorrer entre os pulsos de sincronismo. Entretanto, se essas oscilações ocorrerem próximas da borda ativa do sinal de *clock*, o *flip-flop* pode ir para um estado de metaestabilidade, assim como ocorre com todos os outros sinais de entrada. Qualquer sinal que viola os requisitos de *setup* e *hold* pode causar metaestabilidade. Outra vantagem é que a lógica de *reset* síncrona tende a sintetizar circuitos menores e permite que a ferramenta de síntese realize os testes de temporização para este sinal.

Esta estratégia pode ser desvantajosa em projetos onde há a possibilidade do pulso de *reset* não ser largo o suficiente para ser detectado durante a borda ativa do sinal de *clock*. Neste caso, o *reset* precisará de uma lógica adicional para ser alongado por alguns ciclos de *clock* de forma que passe a ser reconhecido. Este modelo pode não ser adequado também para circuitos onde é possível desativar o sinal de *clock* para economizar energia, por exemplo. Neste cenário, caso o sinal de *clock* esteja desativado e ocorra um pulso de *reset*, este não será identificado. Nesta situação apenas um *reset* assíncrono funcionará ([CUMMINGS; MILLS; GOLSON, 2003](#)).

No projeto implementado neste trabalho, o *reset* é ativo em nível alto e controlado através do SW17 (*slide switch* 17), ou seja, o circuito é levado ao estado de *reset* quando o pino Y23 do FPGA, ao qual o *slide switch* 17 está conectado, recebe nível lógico 1. Como mostrado na [Figura 11](#), quando as chaves estão posicionadas para baixo (mais próximas da borda da placa), elas fornecem um nível lógico baixo para o FPGA e, quando estão posicionadas para cima, fornecem um nível lógico alto ([TERASIC, 2017](#)).

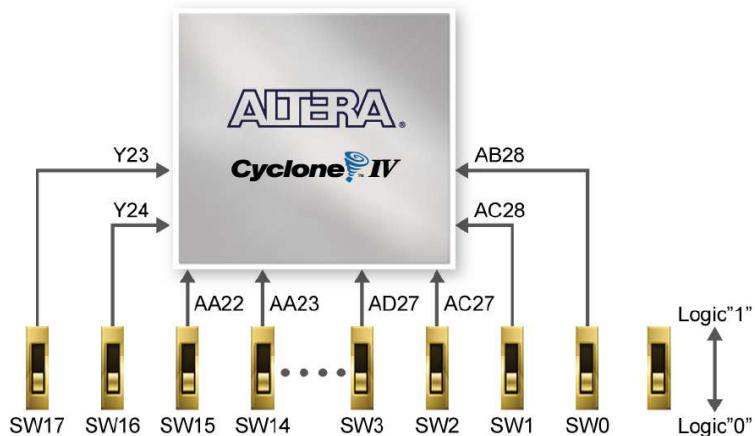


Figura 11 – Conexões entre os *slide switches* e o FPGA na placa DE2-115.

Fonte: ([TERASIC, 2017](#))

O nível lógico fornecido por um *slide switch* tem uma característica assíncrona, visto que o usuário da placa pode mudar o estado da chave a qualquer momento e o FPGA receberá esta mudança imediatamente, independente da região em que o sinal de *clock* se encontra. Assim, caso o sinal gerado por um *slide switch* seja utilizado diretamente como fonte para o *reset* dos módulos contidos no projeto, teremos um circuito com *reset* assíncrono.

Durante o desenvolvimento deste projeto o modelo assíncrono foi experimentado e vários problemas surgiram como resultado desta prática. Verificou-se um comportamento inesperado do circuito após o *reset*, exibindo os *sprites* com movimentos aleatórios e distintos daqueles que foram programados. A imagem de fundo também passou a ser exibida de maneira deformada, com cortes horizontais e verticais que variavam a cada ativação do sinal

de *reset*. Estas reações visuais sinalizaram que o sinal de *reset* estava conduzindo o circuito para estados desconhecidos por conta de algum dos motivos apontados anteriormente como desvantagens na utilização dessa estratégia. Além disso, como consequência da natureza mecânica dos *slide switches*, o nível lógico fornecido por estes elementos tende a oscilar bastante durante a sua movimentação, o que torna ainda menos recomendado a utilização da forma assíncrona quando este componente for utilizado como origem do sinal de *reset*.

Como solução para as falhas citadas foi desenvolvido um módulo capaz de sincronizar o sinal de *reset* gerado pelo *slide switch* com o sinal de *clock* do circuito. Este módulo foi chamado de *Reset_Synchronizer* e suas representações em bloco e RTL podem ser vistas na Figura 12. A sincronização do *reset* é feita através de dois *flip-flops*, como mostrado na Figura 12(b). O sinal *reset_in* é propagado para a saída do *flip-flop* q1 sempre que ocorre uma borda de subida do sinal de *clock*, produzindo o sincronismo desejado entre os dois sinais. Entretanto, caso o sinal *reset_in* oscile próximo à borda de subida do sinal de *clock*, o *flip-flop* q1 pode entrar em um estado de metaestabilidade e gerar um sinal de saída oscilante, o que não é desejado. Nesta situação, para que este sinal oscilatório não seja propagado pelo circuito, um segundo *flip-flop* é adicionado após a saída do *flip-flop* q1. Desta maneira, como a entrada do *flip-flop* *reset_out~reg0* só será propagada na próxima borda de subida do sinal de *clock*, existe tempo para que o *flip-flop* q1 saia da metaestabilidade e consolide o seu sinal de saída. Isso faz com que o sinal *reset_out* esteja sempre estável, mesmo quando violações de *setup* ou de *hold* ocorrerem no sinal *reset_in*, produzindo para os demais módulos do circuito um sinal de *reset* consistente e sincronizado.

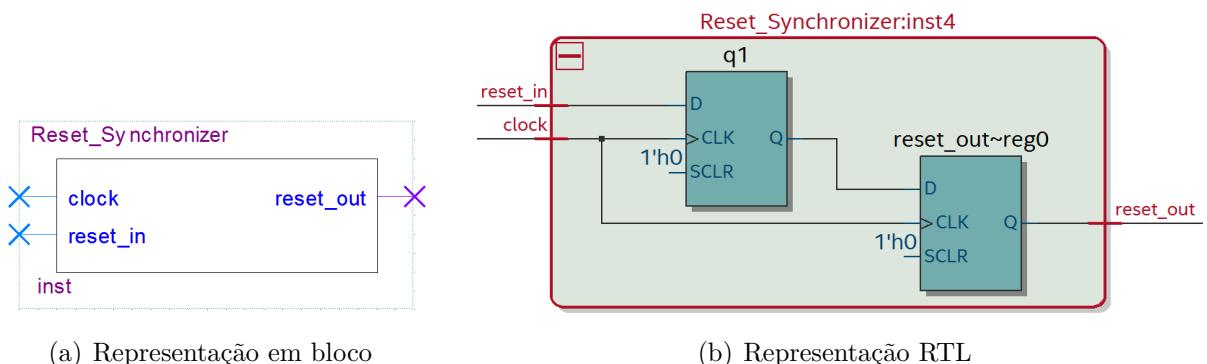


Figura 12 – Representações do módulo *Reset_Synchronizer*.

3.3.2 Árbitro de Memória

Nos sistemas computacionais é comum o compartilhamento de uma memória por vários módulos, permitindo que tenham acesso a um elemento onde possam ler e escrever dados. Para que isto seja possível, usualmente a memória é conectada com os demais dispositivos através de um barramento compartilhado e, como consequência, apenas um

círcuito pode transmitir informações com sucesso pelo barramento a cada momento. Entretanto, podem ocorrer situações em que módulos distintos precisem controlar o barramento simultaneamente para acessar a memória, sendo necessário algum método de arbitragem.

De forma geral, estes métodos podem ser classificados como centralizados ou distribuídos. Em uma estratégia centralizada, um único dispositivo de *hardware*, chamado de controlador ou árbitro de barramento, é responsável por liberar o acesso ao barramento. Já em um esquema distribuído, os módulos atuam juntos para compartilhar o barramento, com cada módulo respeitando uma lógica de controle de acesso, dispensando a necessidade do controlador central ([STALLINGS, 2010](#)).

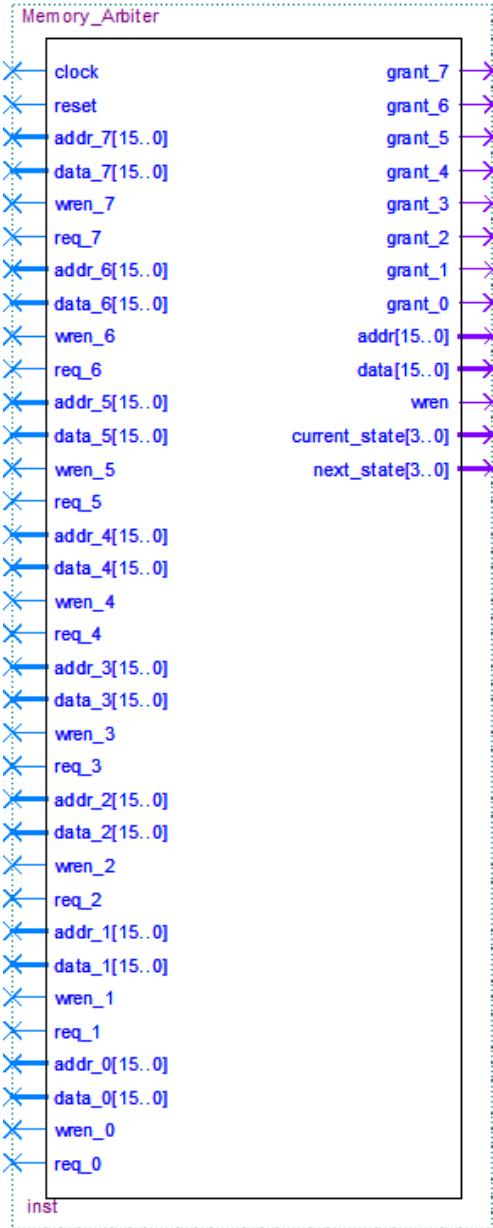
Na arquitetura desenvolvida foi adotada a técnica de arbitragem centralizada, com o módulo *Memory_Arbiter* sendo responsável pelo gerenciamento do acesso à memória de até oito dispositivos distintos. Este componente pode ser visto na [Figura 13](#).

De forma genérica, para ler e escrever dados na memória um módulo precisa receber o sinal dos dados a serem lidos e fornecer os sinais de endereço, dados para escrita e controle, onde seleciona se está escrevendo ou lendo a memória. Os sinais descritos geralmente são ligados na forma de barramentos compartilhados que conectam a memória com os dispositivos que a acessam. Este modelo de acesso à memória é ilustrado na [Figura 14\(a\)](#).

Um módulo que não está transmitindo no barramento compartilhado precisa desconectar suas saídas do barramento, de modo que não haja interferência nas informações que estão sendo propagadas por outro módulo. Para isto, o dispositivo deve deixar suas saídas num estado chamado de alta impedância, onde não há a produção de níveis lógicos altos e baixos, desligando efetivamente as saídas do barramento. Este comportamento é chamado de lógica *tri-state*, visto que três estados podem ser assumidos pela porta de saída, níveis lógicos alto e baixo quando está transmitindo ou alta impedância quando não está transmitindo.

Entretanto, a utilização de sinais *tri-state* internos não é recomendada em FPGAs porque a arquitetura destes dispositivos não inclui lógica *tri-state* internamente. Caso seja utilizada em um projeto, este tipo de lógica geralmente é convertida em portas *OR* ou em uma lógica de multiplexação ([INTEL, 2018](#)).

Devido tal característica, neste projeto os sinais de endereço, dados para escrita e controle não foram concebidos na forma de barramentos compartilhados, pois precisariam ser implementados como portas de saída *tri-state*. Ao invés disso, foi utilizada uma arquitetura como a ilustrada na [Figura 14\(b\)](#). Neste modelo, estes sinais chegam primeiramente ao módulo *Memory_Arbiter*, que os encaminha para a memória após arbitrar qual dispositivo deve ter o acesso concedido. Já o sinal de dados lidos da memória foi projetado como um barramento compartilhado, visto que somente a memória escreve neste barramento.

Figura 13 – Representação em bloco do módulo *Memory_Arbiter*.

O gerenciamento do acesso à memória realizado pelo módulo *Memory_Arbiter* segue um protocolo simples, e seu funcionamento é demonstrado através da máquina de estados finita detalhada na Figura 15. O módulo permanece em estado ocioso até receber uma requisição de acesso à memória de algum dos dispositivos. Neste momento ele roteia para a memória os sinais de endereço, dados para escrita e controle vindos do dispositivo e sinaliza que o acesso foi concedido. Este estado se mantém enquanto o sinal de requisição do dispositivo estiver ativo. No instante em que o acesso à memória é finalizado, o dispositivo desativa o sinal de requisição e o *Memory_Arbiter* volta para o estado de ociosidade, desativando o sinal de acesso concedido.

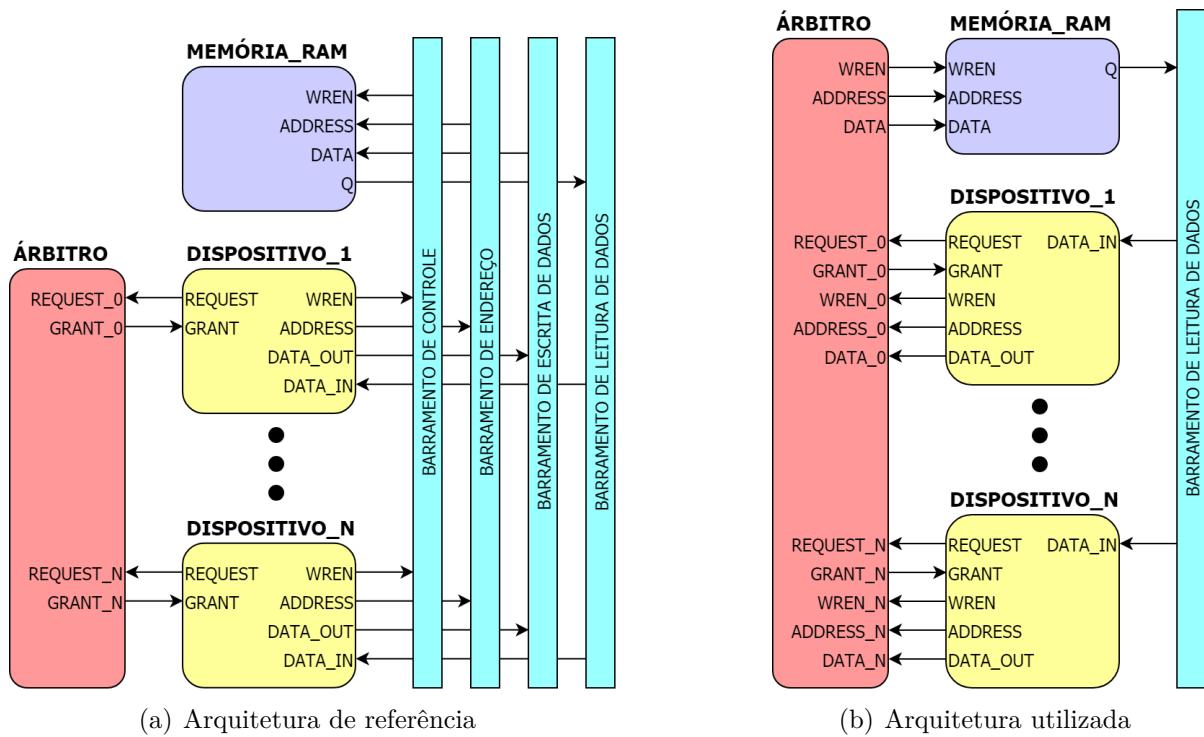
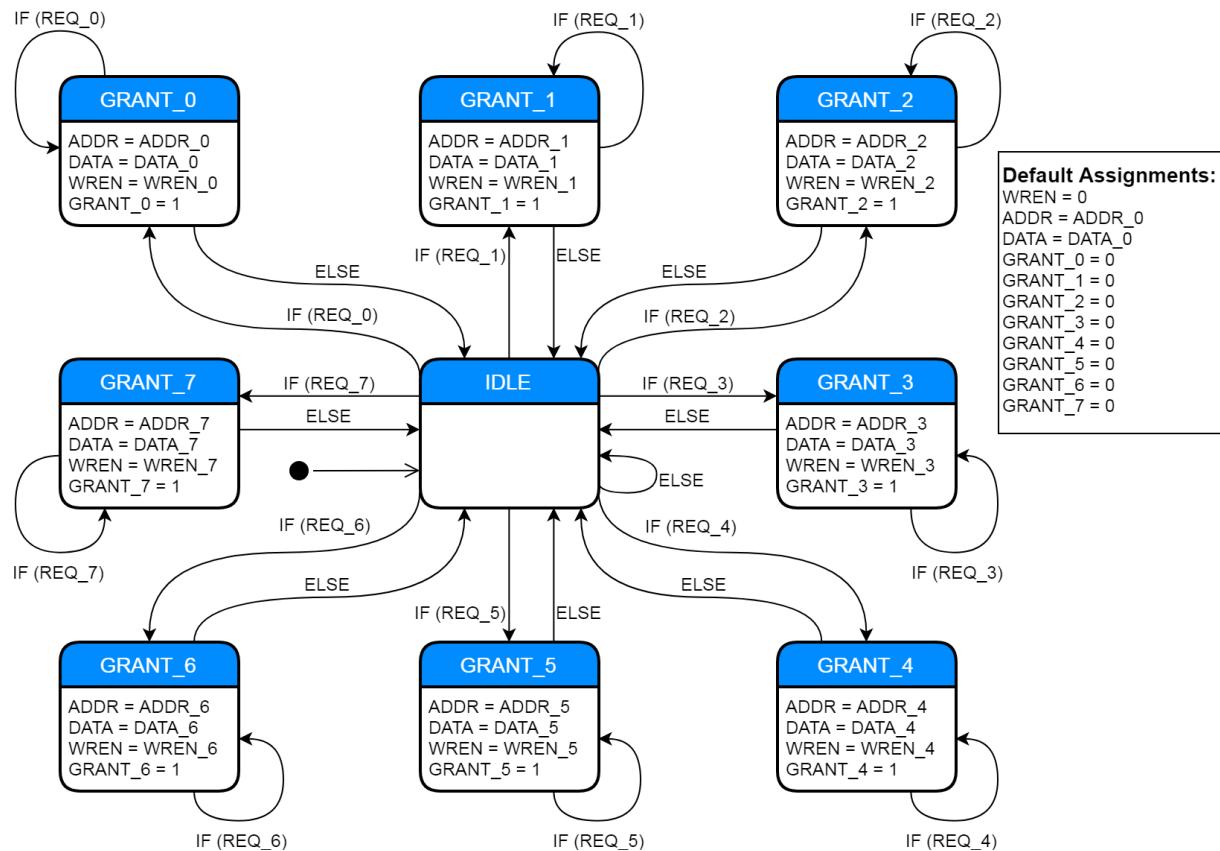


Figura 14 – Modelos de arquitetura para acesso à memória.

Figura 15 – Diagrama de estados do módulo *Memory_Arbiter*.

3.3.3 Controlador Programável de Interrupções

No contexto da computação, as interrupções são alterações no fluxo de controle do processador que não são causadas pelo programa em execução, mas por algum outro elemento, em geral dispositivos de entrada e saída ([TANENBAUM, 2007](#)). Usualmente as arquiteturas computacionais oferecem este mecanismo como forma de permitir que outros módulos possam interromper a sequência de processamento normal do processador.

Este recurso é fornecido primordialmente como um modo de melhorar a eficiência do processamento. A maioria dos dispositivos externos é muito mais lenta do que o processador e este, com a utilização das interrupções, pode executar outras instruções enquanto aguarda a conclusão de uma operação externa em andamento. Quando o dispositivo externo estiver pronto para ser atendido, ou seja, quando estiver preparado para receber ou fornecer mais dados, enviará um sinal de requisição de interrupção ao processador. Este, por sua vez, reage suspendendo a operação do algoritmo atual e desviando a execução para uma rotina que atenderá ao dispositivo que fez a requisição. Após o atendimento da solicitação, o processador retoma a execução original.

Para lidar com a ocorrência de múltiplas interrupções, duas técnicas podem ser utilizadas. A primeira é desativar as interrupções enquanto uma interrupção estiver sendo tratada, o que significa que o processador ignorará novas requisições de interrupção. Uma interrupção que ocorrer neste momento permanecerá pendente e será verificada pelo processador depois que as interrupções tiverem sido reabilitadas. Desta maneira, caso haja uma interrupção durante a execução de um algoritmo, o processador desvia a sua execução para a rotina que tratará este evento e desabilita imediatamente novas interrupções. Depois que a rotina de tratamento de interrupção for concluída, as interrupções são habilitadas novamente e o processador verifica se houve interrupções adicionais. Caso isto não tenha ocorrido, o algoritmo que estava em execução inicialmente é retomado.

Esta técnica é simples e funcional, pois as interrupções são tratadas na medida em que vão ocorrendo, de forma sequencial. A desvantagem é que este método não leva em consideração a prioridade relativa ou requisitos de tempo crítico. Para sistemas onde estes pontos são essenciais uma segunda técnica pode ser utilizada, definindo prioridades para as interrupções e permitindo que uma interrupção de maior prioridade faça com que um tratamento de interrupção com menor prioridade seja interrompido ([STALLINGS, 2010](#)).

O módulo *Interrupt_Controller* mostrado na [Figura 16](#) é o responsável pelo gerenciamento das interrupções no projeto descrito neste trabalho. Ele funciona como uma interface entre o processador e os dispositivos externos, recebendo as requisições de interrupção de até quatro dispositivos distintos e comunicando ao processador qual deve ser atendida. Esta seleção é baseada primeiramente em quais interrupções foram habilitadas e em seguida pela prioridade, caso mais de uma requisição ocorra simultaneamente.

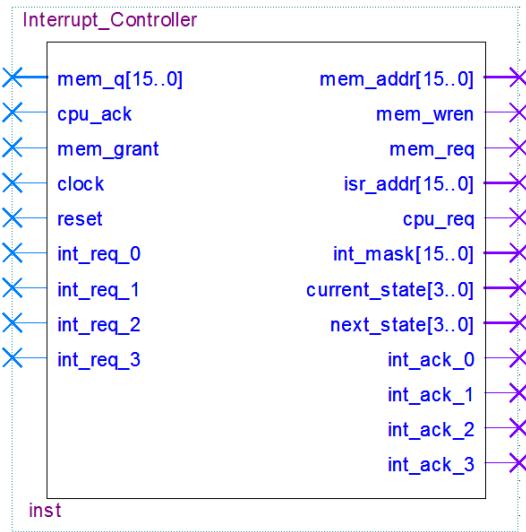


Figura 16 – Representação em bloco do módulo *Interrupt_Controller*.

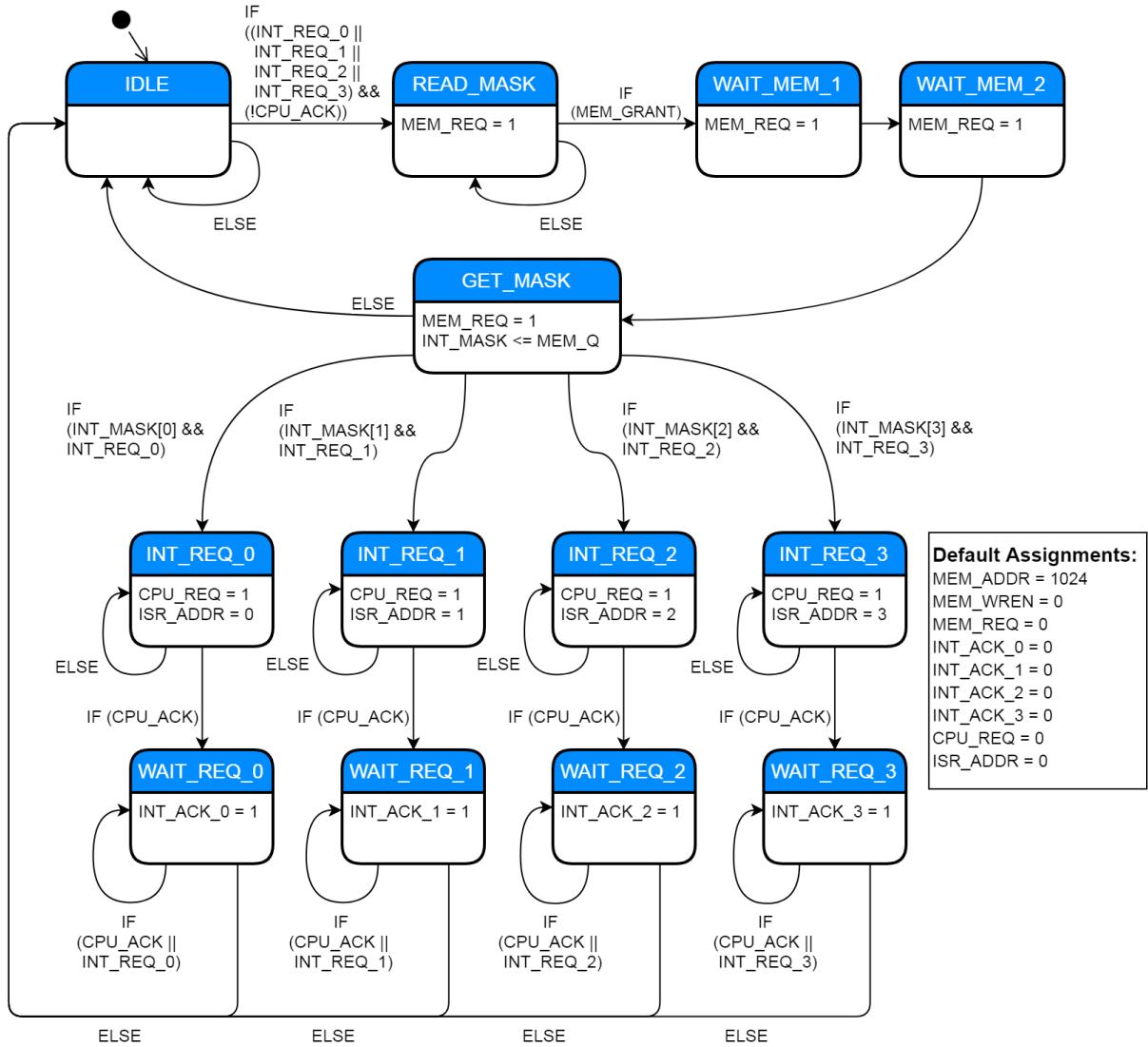
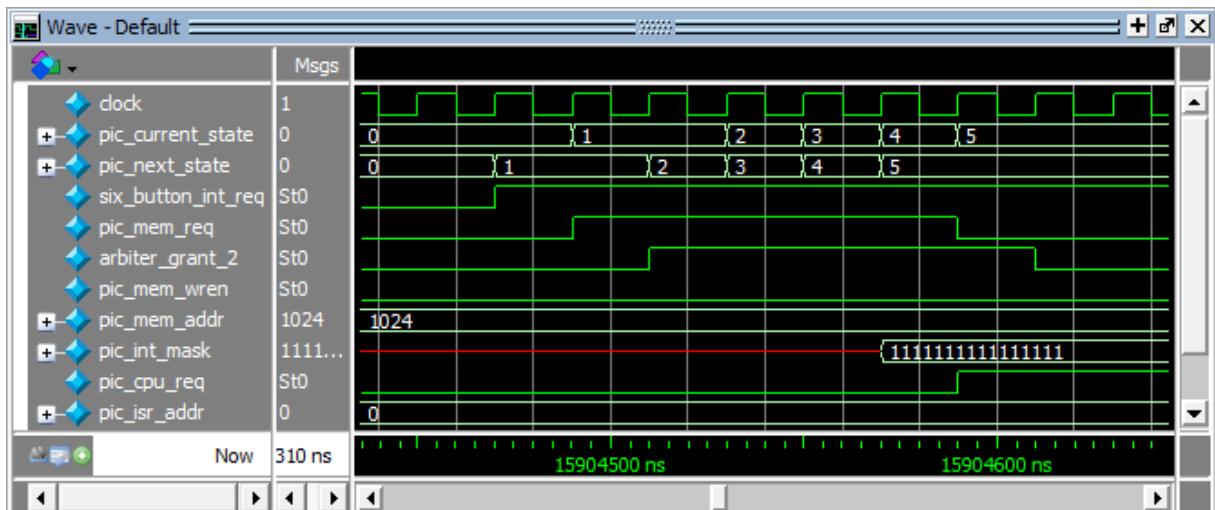
A Figura 17 descreve resumidamente na forma de uma *FSM* (*Finite State Machine*), ou máquina de estados finita, o comportamento deste módulo, que é detalhado em seguida.

A Figura 18 demonstra através de um *waveform* a primeira etapa da sequência de operações realizadas por este bloco. Inicialmente o módulo permanece em um estado de ociosidade, aguardando uma requisição de interrupção de um dos quatro dispositivos suportados. Neste exemplo, um elemento externo solicita a interrupção ativando o sinal *six_button_int_req*.

Após o reconhecimento deste evento, o módulo busca na memória de dados a máscara de interrupções, que é uma palavra de 16 bits encarregada de indicar quais interrupções estão habilitadas ou desabilitadas pelo processador. Nesta palavra, o bit com nível lógico 1 indica que a interrupção associada a este bit está habilitada, e o nível lógico 0 indica que está desabilitada. Como a memória de dados é compartilhada por vários módulos, para realizar a leitura é necessário antes solicitar ao árbitro de memória o acesso a este recurso. Este requerimento é feito com a ativação do sinal *pic_mem_req* e, caso o recurso esteja disponível, o árbitro libera a utilização ativando o sinal *arbiter_grant_2*. Com o acesso à memória de dados autorizado, a palavra armazenada no endereço 1024, onde é salva a máscara de interrupções, é lida e salva no registrador *pic_int_mask*.

Apesar da palavra recuperada ser de 16 bits, apenas os 4 bits menos significativos são considerados no mascaramento, dado que o módulo gerencia as interrupções de apenas quatro dispositivos. O bit menos significativo, o bit 0, efetua o mascaramento da interrupção 0. O bit 1, efetua o mascaramento da interrupção 1. O bit 2, efetua o mascaramento da interrupção 2 e, por fim, o bit 3, efetua o mascaramento da interrupção 3. A Figura 19 detalha as regiões da memória de dados e destaca a estrutura da máscara de interrupção.

Com a finalização da etapa de leitura da máscara de interrupções, e dispondendo desta

Figura 17 – Diagrama de estados do módulo *Interrupt_Controller*.Figura 18 – Primeira etapa do waveform gerado pelo módulo *Interrupt_Controller*.

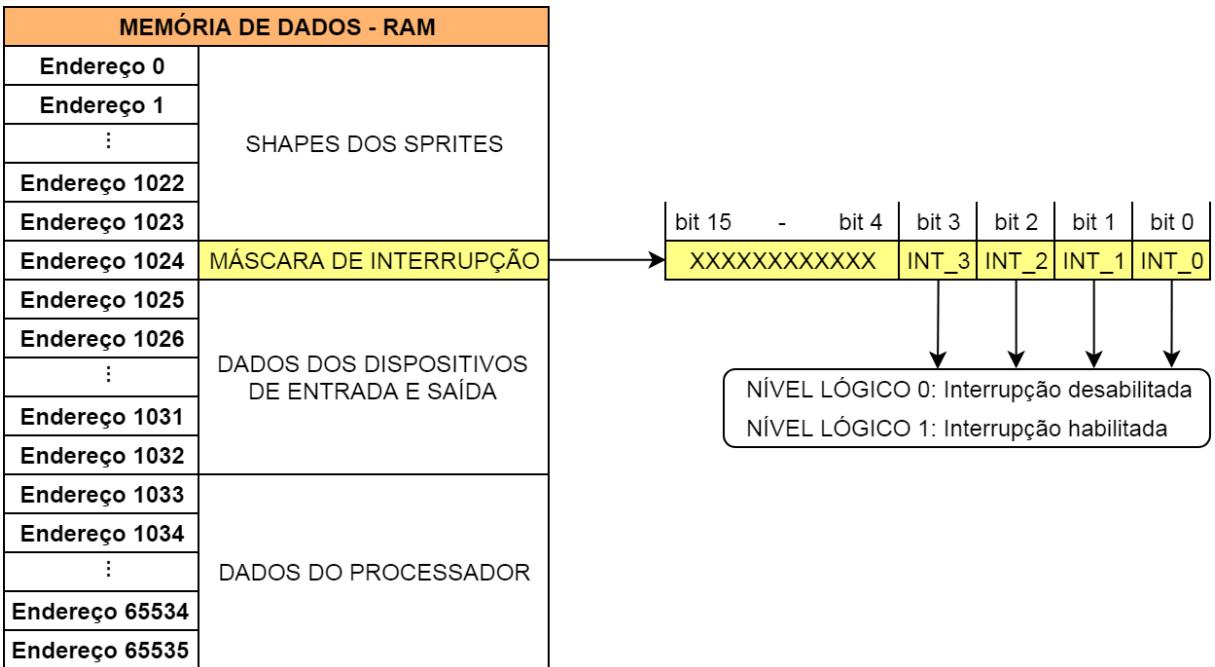


Figura 19 – Segmentos da memória de dados e a máscara de interrupção.

informação no registrador *pic_int_mask*, o módulo verifica através dela se a requisição de interrupção recebida está habilitada ou desabilitada pelo processador. Caso esteja desabilitada, o módulo ignora a solicitação e retorna para o estado de ociosidade. Por outro lado, se a interrupção estiver habilitada, o módulo ativa a linha *pic_cpu_req* para sinalizar ao processador que há uma requisição de interrupção pendente e informa através do barramento *pic_isr_addr* em qual endereço da memória de programa está o ponteiro para a rotina responsável por tratar a interrupção.

Na hipótese de duas ou mais requisições ocorrerem simultaneamente, é adotado o critério de prioridade para a escolha de qual solicitação deve ser atendida primeiro. A interrupção 0 é a de maior prioridade, seguida pela interrupção 1, e assim sucessivamente até a interrupção 3, que possui a menor prioridade.

No exemplo demonstrado na Figura 18 é possível observar que todas as interrupções foram habilitadas pelo processador, visto que todos os bits do registrador *pic_int_mask* estão configurados com nível lógico 1. Como consequência, após a requisição de interrupção gerada pela ativação do sinal *six_button_int_req*, o módulo ativa a linha *pic_cpu_req* e sinaliza no barramento *pic_isr_addr* o endereço 0. Isto indica que o processador deve ler o endereço 0 da memória de programa para ser direcionado à rotina que tratará a interrupção 0.

Na arquitetura desenvolvida neste trabalho, os quatro primeiros endereços da memória de programa foram reservados para o vetor de interrupções, como destacado na Figura 20. Cada endereço deste vetor está associado a uma interrupção e contém uma

instrução de desvio do processador para o endereço da rotina de interrupção correspondente. Ou seja, no endereço 0 da memória de programa haverá uma instrução de desvio para que o processador execute a primeira instrução da rotina dedicada à interrupção 0. No endereço 1 haverá um desvio para a primeira instrução da rotina dedicada à interrupção 1, e assim por diante.

MEMÓRIA DE PROGRAMA - ROM	
Endereço 0	JMP .ISR0 / IRET
Endereço 1	JMP .ISR1 / IRET
Endereço 2	JMP .ISR2 / IRET
Endereço 3	JMP .ISR3 / IRET
Endereço 4	CÓDIGO
Endereço 5	
Endereço 6	
:	
Endereço 65533	
Endereço 65534	
Endereço 65535	

Figura 20 – Segmentos da memória de programa e o vetor de interrupções.

Caso uma rotina de interrupção, também conhecida como *ISR (Interrupt Service Routine)*, não tenha sido declarada no código, o *assembler* posicionará no endereço associado do vetor de interrupções uma instrução para que o processador retome o código que estava sendo executado inicialmente. Como exemplo, se o processador recebe uma requisição da interrupção 0 mas nenhuma rotina para tratá-la tiver sido declarada, a execução do código inicial será retomada.

A Figura 20 ilustra este mecanismo com a representação das instruções *JMP* e *IRET*. Se uma rotina de interrupção foi codificada, o endereço do vetor de interrupções relacionado armazenará uma instrução *JMP* seguida do endereço de início da rotina de tratamento. Isso fará com que o processador seja direcionado para a execução da *ISR*. Em contrapartida, se uma rotina de interrupção não tiver sido definida, o endereço ligado a esta interrupção guardará uma instrução *IRET*, sinalizando ao processador que ele deve retornar para a execução do código anterior.

Dando seguimento ao aviso de interrupção pendente através do sinal *pic_cpu_req*, a segunda etapa das ações do módulo é iniciada com a espera por uma indicação do processador de que a solicitação foi reconhecida e está sendo atendida. Isto é feito através da linha *cpu_int_ack*, que permanece ativada durante todo o período em que o processador

estiver executando a rotina de interrupção. No momento em que o módulo reconhece a ativação do sinal *cpu_int_ack* é efetuada a desativação da linha *pic_cpu_req* e a ativação da linha *pic_int_ack_0*. A primeira ação é tomada para que o processador não interprete que uma nova interrupção foi requisitada. Dado que uma interrupção já está sendo atendida, o processador só deverá ser interrompido novamente após a finalização deste atendimento. A segunda ação indica ao dispositivo externo que a interrupção requisitada está sendo atendida pelo processador. Como consequência, o dispositivo externo desativa o sinal *six_button_int_req* para não haver o julgamento de que uma nova interrupção foi requisitada. O comportamento descrito é ilustrado na [Figura 21](#).

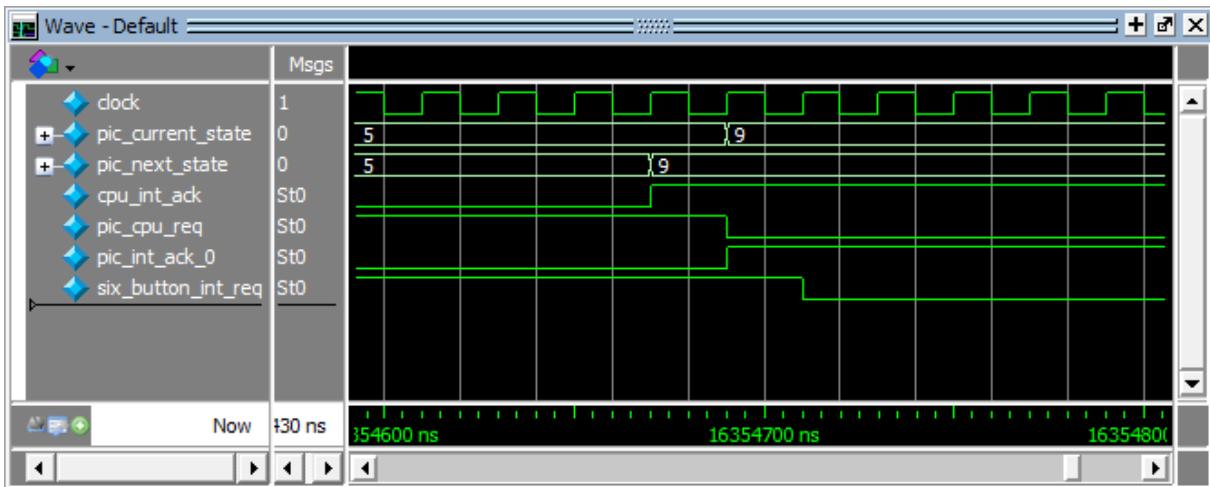


Figura 21 – Segunda etapa do *waveform* gerado pelo módulo *Interrupt_Controller*.

No momento em que o processador finaliza a execução da rotina de interrupção é realizada a desativação do sinal *cpu_int_ack*, sinalizando este evento ao módulo. A [Figura 22](#) demonstra esta ação, que é traduzida como uma indicação de que o processador está pronto para receber novas requisições de interrupções. Como consequência, o módulo repassa esta informação para o dispositivo externo com a desativação do sinal *pic_int_ack_0*, indicando a conclusão da interrupção e que uma nova requisição pode ser feita.

É importante notar que os sinais de requisição (*request*) e de reconhecimento (*acknowledge*) são utilizados como meio de comunicação entre o módulo *Interrupt_Controller* e os módulos com que faz interface. Este sinais seguem um protocolo em que há uma espécie de intertravamento. Desta forma, o módulo que faz uma requisição deve ativar seu sinal de *request* e mantê-lo ativado somente enquanto aguarda a ativação do sinal de *acknowledge*. Assim que este sinal for identificado deve haver a desativação do sinal de *request*, e uma nova requisição só poderá ser executada quando o sinal de *acknowledge* tiver sido desativado. O módulo que recebe uma requisição deve apenas ativar o sinal *acknowledge*, assim que identificar a ativação do sinal de *request*, e mantê-lo ativado

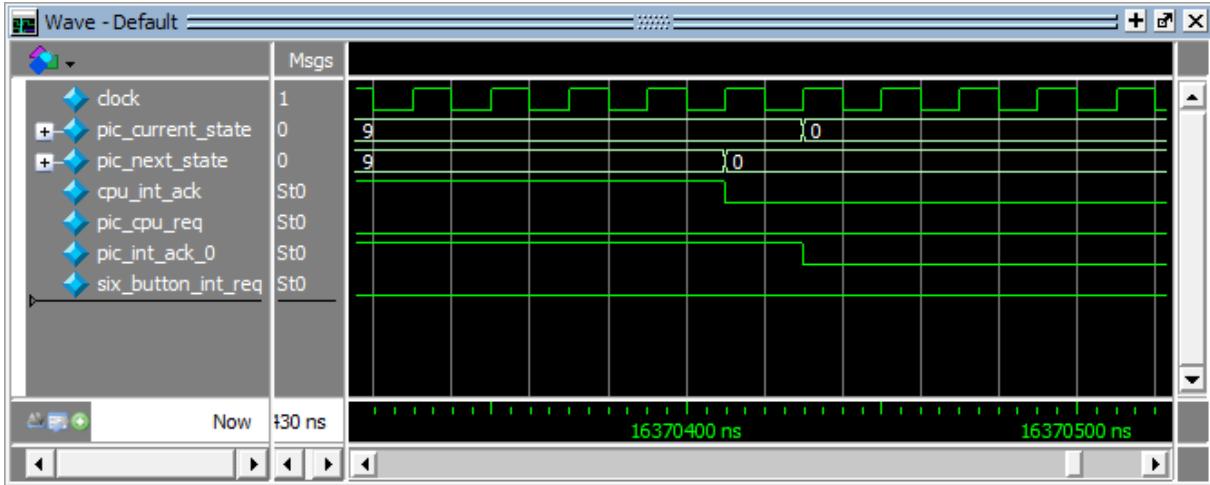


Figura 22 – Terceira etapa do *waveform* gerado pelo módulo *Interrupt_Controller*.

enquanto estiver atendendo a requisição. Este protocolo garante que o processador não seja novamente interrompido quando já estiver atendendo a uma interrupção.

3.3.4 Interface para Controle

Em uma plataforma de jogos eletrônicos é indispensável a existência de algum dispositivo de entrada que permita a interação do usuário com o aparelho. É comum que este papel seja exercido por controles, com formatos como o de *joysticks* ou de *gamepads*, por meio dos quais são enviados comandos ao equipamento.

No projeto desenvolvido o usuário interage com a plataforma através de um controle do videogame *Sega Mega Drive*, também conhecido como *Sega Genesis* na América do Norte. O controle deste equipamento conta com versões de 3 botões e de 6 botões, sendo este segundo modelo o escolhido para a integração com o protótipo. A conexão é feita através de um conector DB9 fêmea, onde 2 pinos são utilizados para a alimentação do dispositivo e 7 pinos são empregados na comunicação dos comandos, totalizando os 9 pinos disponíveis. Entretanto, o controle possui ao todo 12 botões com funcionalidades distintas que precisam ser transmitidas, exigindo que os estados dos botões sejam codificados de alguma maneira.

A leitura completa do controle é realizada em oito etapas, usando seis linhas para a transmissão de dados, nomeadas como uma sequência de D0 a D5, e uma sétima linha denominada SELECT, que determina quais botões terão seus estados transmitidos nas linhas de dados. A linha de seleção SELECT é normalmente mantida em nível alto e deve ser pulsada para que os oito estados do controle sejam percorridos. Os pulsos são gerados com a alternância entre níveis lógicos alto e baixo em intervalos de 20 microsegundos e o processo é repetido aproximadamente a cada 16,7 milissegundos, em virtude do controle geralmente ser lido com uma frequência de 60 Hz ([SARNOFF, 2009](#)). A Figura 23 ilustra

através de um *waveform* as temporizações descritas e o comportamento que o sinal SELECT deve apresentar durante a leitura do controle.

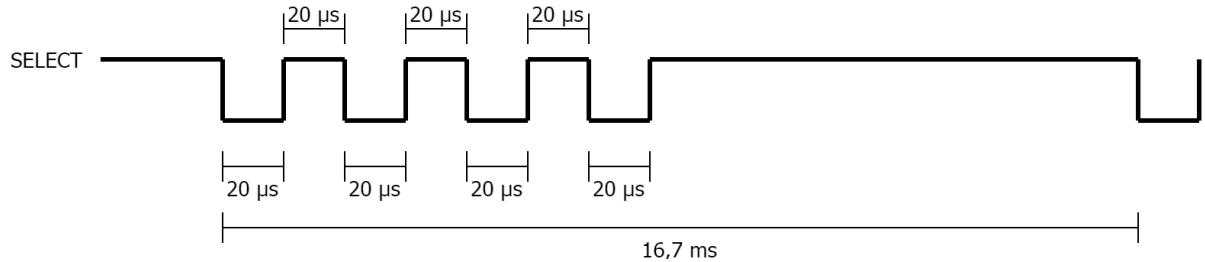


Figura 23 – *Waveform* com temporizações do sinal SELECT.

O [Quadro 1](#) exibe os níveis lógicos aplicados na linha SELECT e quais botões podem ser lidos nas linhas de dados em cada um dos estados assumidos pelo controle. Uma linha de dados assume nível lógico baixo quando o botão correspondente está pressionado e nível lógico alto na ocasião em que o botão correlato não está acionado.

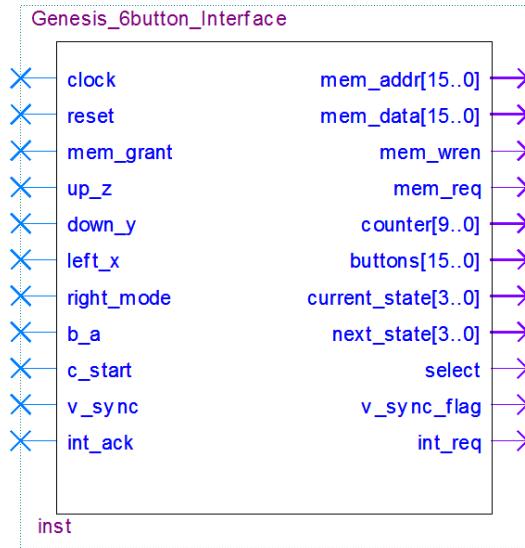
Quadro 1 – Estados e sinais para leitura do controle de 6 botões do *Sega Mega Drive*

Estado	Entrada	Saídas						
		Pino 7 Select	Pino 1 D0	Pino 2 D1	Pino 3 D2	Pino 4 D3	Pino 6 D4	Pino 9 D5
0	HIGH	Up	Down	Left	Right	B	C	
1	LOW	Up	Down	0	0	A	Start	
2	HIGH	Up	Down	Left	Right	B	C	
3	LOW	Up	Down	0	0	A	Start	
4	HIGH	Up	Down	Left	Right	B	C	
5	LOW	0	0	0	0	A	Start	
6	HIGH	Z	Y	X	Mode	B	C	
7	LOW	-	-	-	-	A	Start	

Fonte: ([SEGA RETRO, 2012](#))

A representação gráfica do módulo *Genesis_6button_Interface*, desenvolvido para fazer interface com o controle, é exposta na [Figura 24](#). Este bloco é o responsável por produzir o sinal aplicado à linha SELECT e por decodificar os estados dos botões a partir dos dados recebidos através das linhas D0 a D5.

A condição dos botões é salva na memória de dados em uma palavra de 16 bits que indica quais deles estão pressionados. Nesta palavra adotou-se uma representação oposta à praticada pelo controle, ou seja, o bit com nível lógico 1 indica que o botão associado a este bit está acionado, e o nível lógico 0 indica que o botão não está acionado. Como a memória de dados é compartilhada por vários módulos, para realizar a escrita é necessário antes solicitar ao árbitro de memória o acesso a este recurso. A requisição é feita através do pino *mem_req* e, caso o recurso esteja disponível, o árbitro libera a utilização ativando

Figura 24 – Representação em bloco do módulo *Genesis_6button_Interface*.

o pino *mem_grant*. Com o acesso à memória de dados autorizado, a palavra é armazenada no endereço 1025 e fica disponível para ser lida pelo processador.

Apesar de 16 bits serem escritos na memória apenas os 12 bits menos significativos são manipulados, dado que o controle possui somente 12 botões cujos estados precisam ser armazenados. A Figura 25 detalha as regiões da memória de dados e destaca como a condição de cada botão é representada.

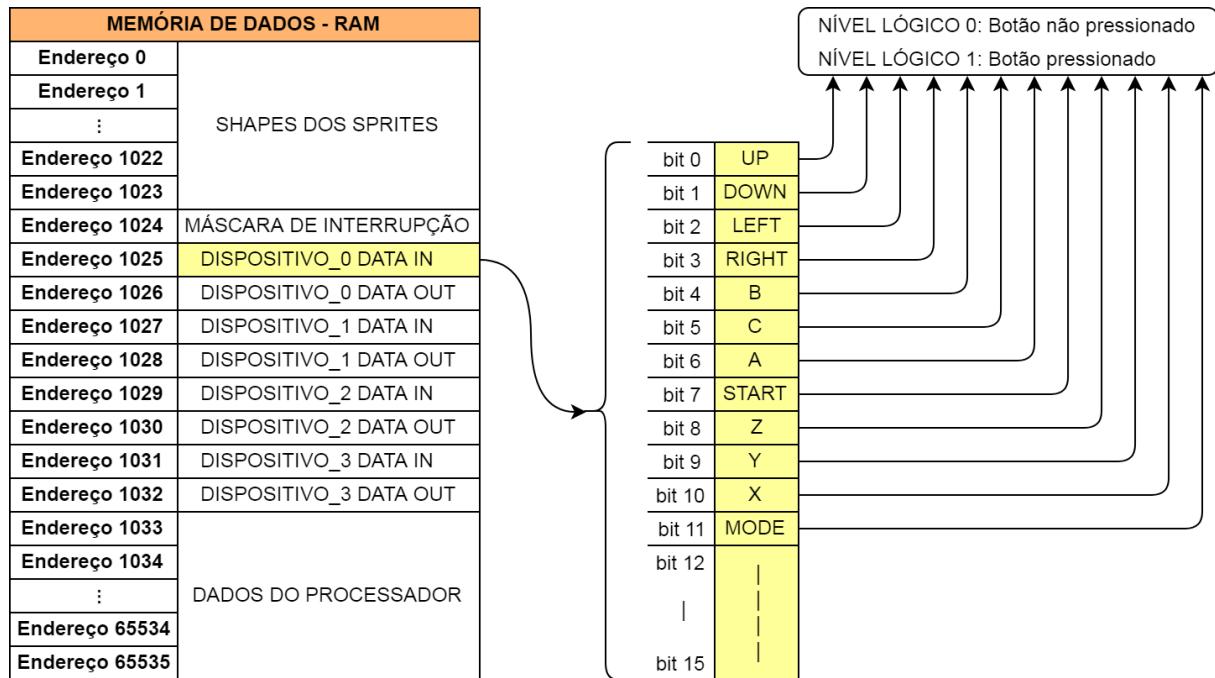


Figura 25 – Segmentos da memória de dados e os estados dos botões do controle.

Por conta do compartilhamento da memória para o armazenamento de dados,

almeja-se que o acesso a este recurso seja feito de forma ágil e o menor número de vezes possível. A fim de atender tais objetivos, o módulo foi projetado para funcionar com *clock* de 50 MHz e escrever um novo dado na memória apenas se este for diferente do que já está salvo.

Após a escrita dos dados na memória, o módulo faz uma requisição de interrupção ao controlador programável de interrupções usando o pino *int_req*, com o objetivo de informar ao processador que existem novos dados para serem lidos. A sinalização de que a interrupção está sendo atendida é feita pelo controlador de interrupções através do pino *int_ack*, de forma que o módulo produza novas interrupções apenas quando a interrupção gerada anteriormente tiver sido integralmente resolvida.

Duas estratégias foram aplicadas para medir a temporização dos ciclos necessários na geração do sinal SELECT. A primeira foi o uso de um contador de pulsos de *clock* no cálculo do período de 20 microsegundos, que deve existir na alternância entre níveis lógicos alto e baixo, exigidos para a leitura dos estados do controle. A frequência de 50 MHz do *clock* no qual o módulo trabalha requer que o contador seja incrementado até assumir o valor 1000 para a obtenção do intervalo desejado.

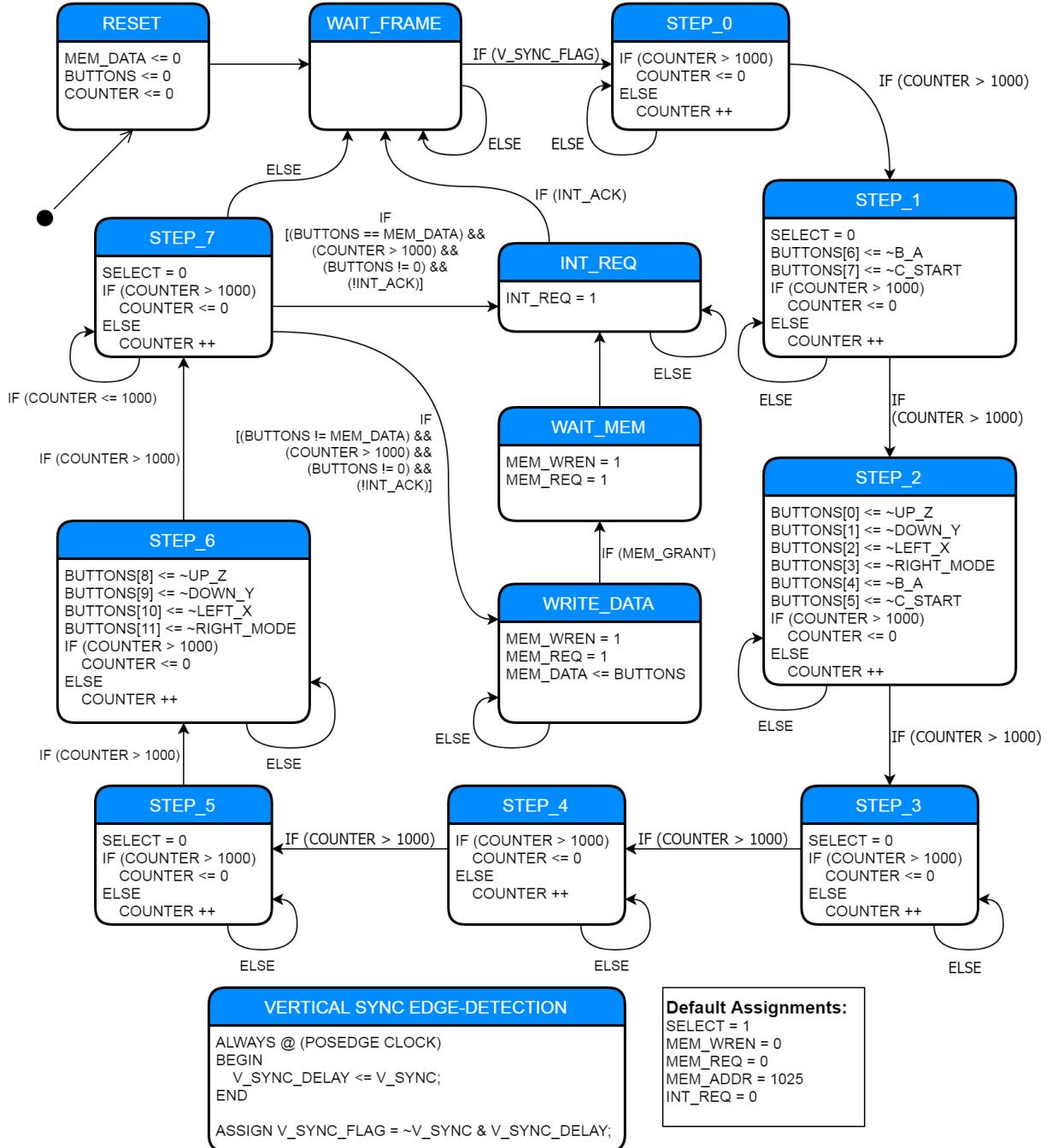
O segundo método se deu pelo aproveitamento do sinal de sincronismo vertical, fornecido pelo módulo da interface de vídeo VGA, na determinação do intervalo de leitura do controle. Este processo deve ser repetido aproximadamente a cada 16,7 milissegundos, mesmo intervalo em que o pulso de sincronismo vertical é gerado para atualizar os quadros da imagem em uma frequência de 60 Hz. O pino de entrada *v_sync* recebe este sinal de sincronização e um circuito interno de detecção de borda faz o reconhecimento do momento do pulso para indicá-lo à lógica do bloco. O funcionamento do módulo é resumido por meio de uma máquina de estados finita na [Figura 26](#).

3.3.5 Interface para Memória SRAM

A placa *DE2-115* possui um *chip* de memória SRAM externo ao FPGA com capacidade de 2 MB e largura de dados de 16 bits, sendo capaz de operar numa frequência máxima de aproximadamente 125 MHz ([TERASIC, 2017](#)). As características deste dispositivo tornaram-no adequado para armazenar a imagem de fundo do sinal de vídeo produzido pela plataforma e, por conta disso, precisa ser constantemente acessado.

A [Figura 27](#) demonstra as conexões existentes da memória SRAM com o FPGA, e a [Tabela 1](#) descreve a função de cada porta do *chip*. O módulo *SRAM_Interface*, ilustrado na [Figura 28](#), foi desenvolvido exclusivamente para fazer a interface da memória SRAM com os demais elementos do circuito, produzindo os sinais necessários na leitura da memória e convertendo os dados recebidos para o formato adequado aos demais módulos.

O bloco opera de forma bastante simplificada, dado que o acesso à memória é feito

Figura 26 – Diagrama de estados do módulo *Genesis_6button_Interface*.

apenas para ler dados. A saída *Write Enable* é mantida sempre em nível alto, indicando a operação de leitura. As demais saídas de controle *Chip Enable*, *Output Enable*, *Upper Byte* e *Lower Byte* são conservadas em nível baixo com o objetivo de ativar o *chip* da memória e habilitar completamente a porta por onde os dados são lidos. O endereço de memória que deve ser lido é gerado externamente, pelo módulo da interface de vídeo VGA, e conectado à entrada *iADDR* do módulo. O sinal é então diretamente encaminhado para a saída *oADDR*, que está conectada à porta de endereço da memória SRAM.

Cada endereço da memória armazena 16 bits de dados referentes a um *pixel* da

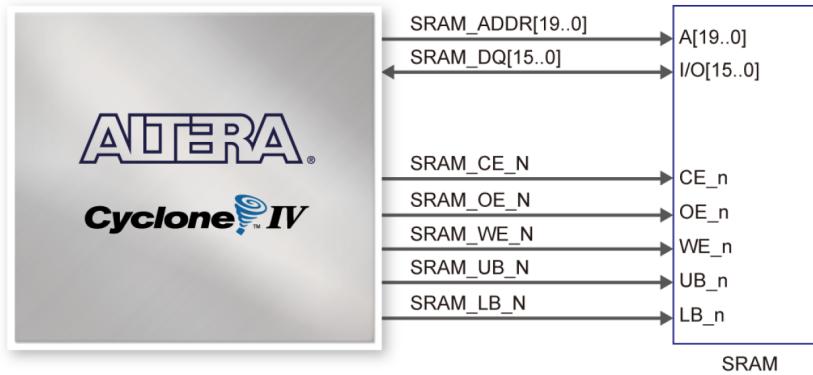


Figura 27 – Conexões entre a memória SRAM e o FPGA na placa DE2-115.

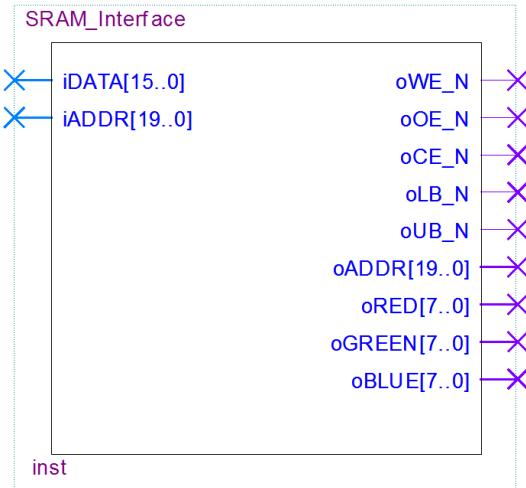
Fonte: ([TERASIC, 2017](#))

Tabela 1 – Portas da memória SRAM

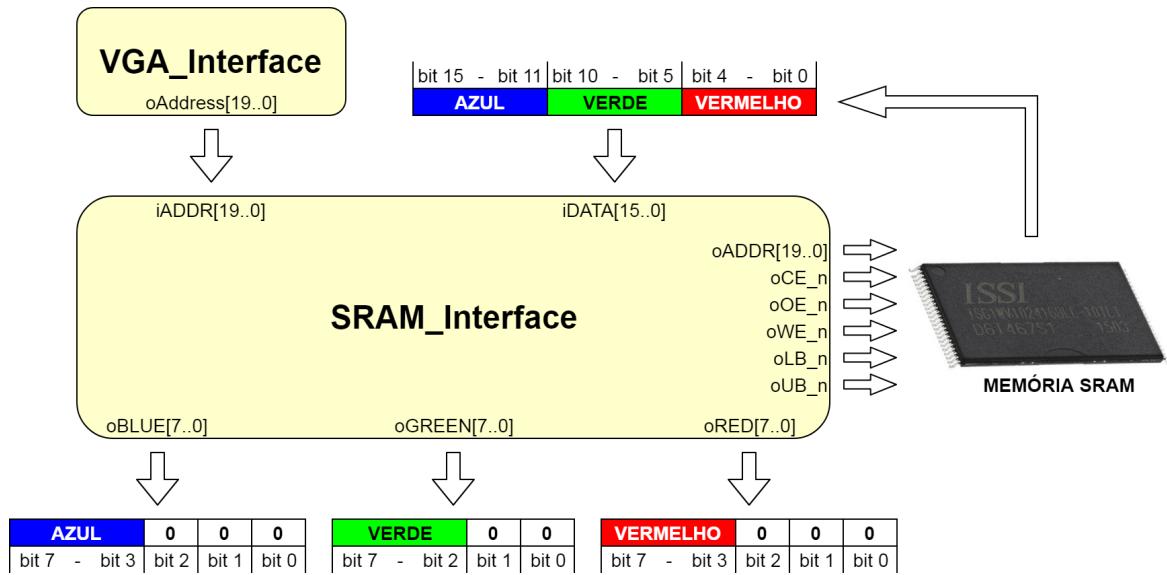
Nome da porta	Descrição
A[19..0]	Entrada de endereço.
I/O[15..0]	Entrada e saída de dados.
CE_n	Entrada de controle <i>Chip Enable</i> . Pino ativo em nível baixo. Quando em nível alto (desativado), o dispositivo assume um modo de espera com baixo consumo de energia.
OE_n	Entrada de controle <i>Output Enable</i> . Pino ativo em nível baixo. Controla a porta de dados nas operações de leitura. Quando em nível alto (desativado), mantém a porta de dados em estado de alta impedância.
WE_n	Entrada de controle <i>Write Enable</i> . Pino ativo em nível baixo. Controla tanto a escrita quanto a leitura da memória.
UB_n	Entrada de controle <i>Upper-byte</i> . Pino ativo em nível baixo. Permite acesso ao byte superior do dado (I/O[15..8]).
LB_n	Entrada de controle <i>Lower-byte</i> . Pino ativo em nível baixo. Permite acesso ao byte inferior do dado (I/O[7..0]).

Fonte: ([ISSI, 2014](#))

imagem, e o processo de leitura segue a sequência de exibição do módulo de interface de vídeo VGA. Assim, após a recuperação e reprodução de um *pixel*, o endereço seguinte é encaminhado para a memória de modo que o novo *pixel* seja lido e projetado em seguida. A imagem armazenada representa as componentes de cor vermelha e azul usando 5 bits e a componente verde utilizando 6 bits, totalizando os 16 bits de largura dos dados. Porém, a saída de vídeo VGA da plataforma deve receber cada componente de cor em barramentos de 8 bits e, por conta disso, os sinais trafegam pelo circuito com esta configuração. Consequentemente, é necessário adicionar 3 bits às componentes vermelha e azul e 2 bits à componente verde. Para este fim, os dados recebidos da memória por meio da entrada *iDATA* são divididos nas três componentes de cor e direcionados para as portas

Figura 28 – Representação em bloco do módulo *SRAM_Interface*.

oRED, *oGREEN* e *oBLUE*, onde os 8 bits de cada barramento são completados definindo como zero os bits menos significativos de cada saída. A Figura 29 ilustra as conexões do módulo e demonstra seu funcionamento de forma simplificada.

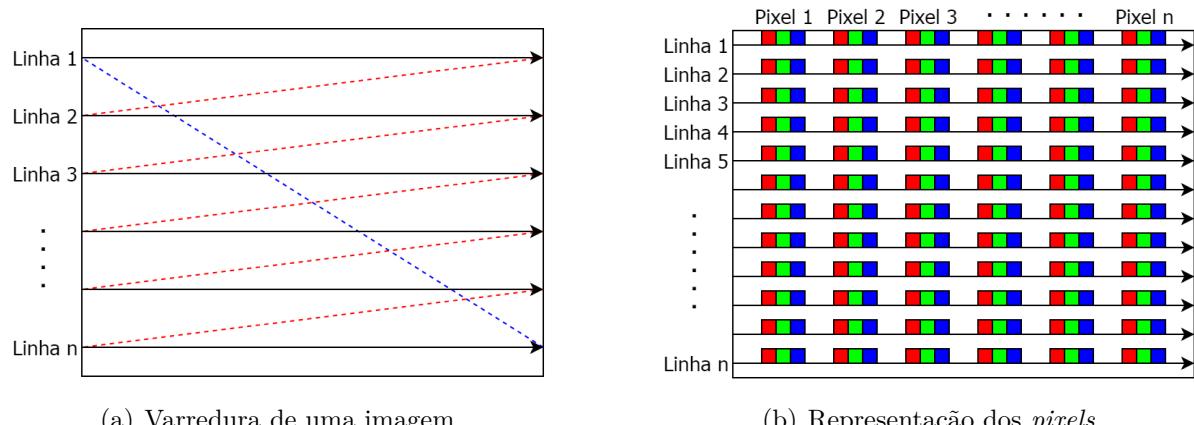
Figura 29 – Diagrama simplificado das conexões do módulo *SRAM_Interface*.

3.3.6 Interface de Vídeo VGA

Usualmente as plataformas de jogos eletrônicos reagem às ações do usuário através de estímulos visuais, provocados por imagens exibidas em algum tipo de tela. Deste modo, a interface de vídeo pode ser considerada elemento essencial para uma interação satisfatória com o equipamento, permitindo a visualização de respostas aos comandos dados pelo jogador.

A placa *DE2-115*, utilizada na construção do protótipo, conta com uma saída de vídeo VGA (*Video Graphics Array*) que foi empregada na exibição das imagens produzidas pela plataforma. O VGA é um padrão de vídeo analógico com funcionamento baseado no processo de varredura, ou seja, a imagem é formada através do desenho sequencial de pontos na tela nos sentidos horizontal e vertical. A varredura destes pontos ocorre da esquerda para a direita, formando uma linha, e de cima para baixo, criando o conjunto de linhas que forma um quadro da imagem. Este comportamento é ilustrado na [Figura 30\(a\)](#), onde os pontilhados vermelhos indicam o fim da varredura de uma linha e o início da linha subsequente, e o pontilhado azul indica o final de um quadro e o início do quadro seguinte.

Os pontos que formam as linhas, e consequentemente as imagens, são chamados de *pixels* (*picture elements*) e estão representados na [Figura 30\(b\)](#). Em telas coloridas estes elementos geralmente são construídos com capacidade de emitirem luz nas cores vermelha, verde e azul. A combinação dessas três componentes, com seus diferentes níveis de brilho, permite que cada ponto da imagem possa assumir uma infinidade de cores. O tamanho reduzido dos *pixels*, e alta frequência em que são varridos, possibilita que o processo ocorrido durante a formação da imagem na tela seja imperceptível ao olho humano.



[Figura 30 – Construção de uma imagem.](#)

Para construir a imagem corretamente, o monitor precisa das sinalizações de quando ocorre o final de uma linha e do momento de conclusão de um quadro. No padrão VGA estas indicações são feitas respectivamente pelos sinais de sincronismo horizontal e de sincronismo vertical. O primeiro especifica o tempo requerido para percorrer uma linha, e o segundo controla o tempo necessário para percorrer toda a tela. A temporização destes sinais depende de parâmetros como a resolução e a frequência de atualização da imagem ([CHU, 2008](#)). Este projeto adotou a especificação do vídeo com resolução de 640 colunas por 480 linhas e varredura dos *pixels* com taxa de 25 MHz, produzindo uma atualização de quadros com frequência de aproximadamente 60 Hz.

Em um sinal de vídeo com as características adotadas, o sincronismo horizontal apresenta um período de $32 \mu\text{s}$ enquanto que o sincronismo vertical possui período de 16,8 ms. Considerando a varredura dos *pixels* em uma taxa de 25 MHz observa-se que o período do sincronismo horizontal é suficiente para atualizar 800 *pixels*, e que o período do sincronismo vertical é o bastante para percorrer 525 linhas. Isto significa que o monitor precisa de algum tempo além do necessário para exibir as 640 colunas e 480 linhas de *pixels* visíveis. Este intervalo adicional é explicado através da divisão dos sinais de sincronismo horizontal e vertical em quatro regiões chamadas de *back porch*, intervalo de exibição, *front porch* e pulso de sincronismo.

No sinal de sincronismo horizontal, o *back porch* é a área que forma a borda esquerda da tela, com comprimento de 48 *pixels* ou $1,92 \mu\text{s}$. O intervalo de exibição é a região onde os *pixels* são realmente exibidos na tela, e possui comprimento de 640 *pixels* ou $25,6 \mu\text{s}$. O *front porch* é a área que forma a borda direita da tela, apresentando comprimento de 16 *pixels* ou $0,64 \mu\text{s}$. Por fim, o pulso de sincronismo indica ao monitor o fim de uma linha, para que retorne à borda esquerda e exiba a linha seguinte. O comprimento deste pulso é de 96 *pixels* ou $3,84 \mu\text{s}$ (CHU, 2008). A Figura 31 representa estes intervalos por linhas vermelhas, e a soma deles produz o período de $32 \mu\text{s}$, ou 800 *pixels*, do sincronismo horizontal citado anteriormente.

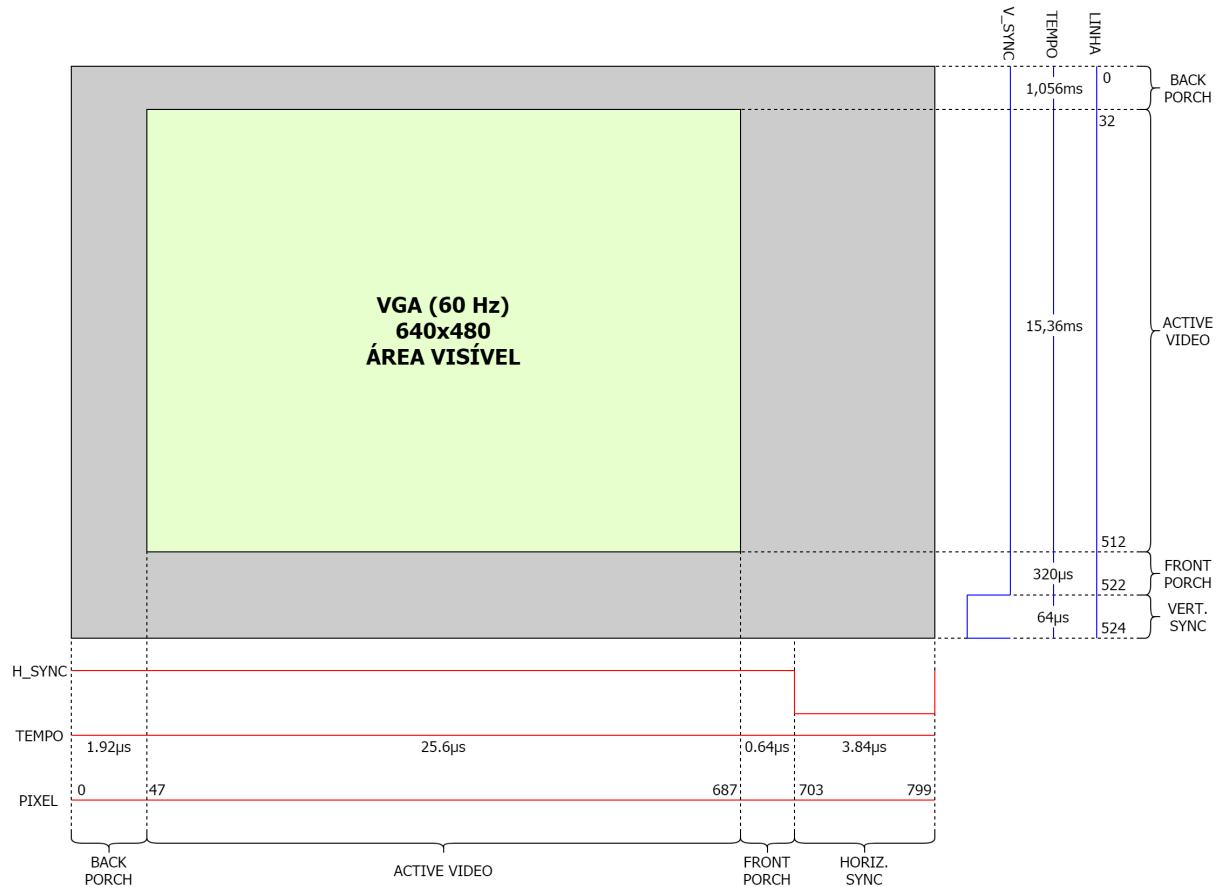


Figura 31 – Especificações de tempo na varredura de uma imagem.

O sinal do sincronismo vertical tem formato semelhante ao do sinal de sincronismo horizontal apresentado. O *back porch* é a área que forma a borda superior da tela, com comprimento de 33 linhas ou 1,056 ms. O intervalo de exibição é a região onde as linhas horizontais são realmente exibidas na tela, e possui comprimento de 480 linhas ou 15,36 ms. O *front porch* é a área que forma a borda inferior da tela, apresentando comprimento de 10 linhas ou 320 μ s. Finalmente, o pulso de sincronismo indica ao monitor o fim de um quadro, para que retorne ao topo da tela e exiba o quadro seguinte. O comprimento deste pulso é de 2 linhas ou 64 μ s (CHU, 2008). Estes intervalos são representados por linhas azuis na [Figura 31](#), e somados produzem o período de 16,8 ms, ou 525 linhas, do sincronismo vertical apontado previamente.

Um monitor capaz de exibir imagens com padrão VGA necessita apenas dos sinais de sincronismo vertical e horizontal, e das componentes vermelha, verde e azul de cada *pixel* que será exibido. Estes sinais da saída de vídeo VGA estão disponíveis na placa *DE2-115* através de um conector D-SUB de 15 pinos, que deve ser usado na conexão com um monitor.

Os sinais de sincronismo vertical e horizontal possuem uma natureza digital, dado que devem permanecer normalmente em nível alto e assumir nível baixo apenas durante um intervalo específico de tempo, retornando em seguida para o nível alto e gerando os pulsos de sincronismo. Esta característica permite que estes sinais sejam gerados por um circuito programado no FPGA e conectados diretamente ao conector da saída VGA.

Por outro lado, como consequência do VGA ser um padrão de vídeo analógico, os sinais das componentes vermelha, verde e azul dos *pixels* são igualmente analógicos. Isso exige um elemento intermediário que converta os dados digitais produzidos pelo FPGA em sinais analógicos, e este papel é assumido pelo *chip* ADV7123, da *Analog Devices*, presente na placa *DE2-115*. As conexões entre o FPGA, o ADV7123 e o conector da saída de vídeo VGA na placa *DE2-115* são ilustrados na [Figura 32](#).

O *chip* ADV7123 contém internamente três conversores digital-analógico, também chamados de DACs (*Digital-to-Analog Converters*). Cada DAC possui uma porta de entrada separada, com 10 bits de largura, e é responsável por converter, na borda de subida de cada ciclo de *clock*, as informações de uma das componentes de cor. Assim, os 30 bits de dados de um *pixel* devem ser conduzidos aos pinos de entrada R0 a R9, para o conversor do elemento vermelho, G0 a G9, para o conversor do elemento verde, e B0 a B9, para o conversor do elemento azul. Os resultados das conversões são propagados para três saídas analógicas, nomeadas como IOR, IOG e IOB, contendo respectivamente os sinais das componentes vermelha, verde e azul do sinal de vídeo VGA (ANALOG DEVICES, 2010).

Apesar do DAC disponibilizar entradas de 10 bits para converter cada componente de cor, na placa *DE2-115* apenas os 8 bits mais significativos de cada porta foram

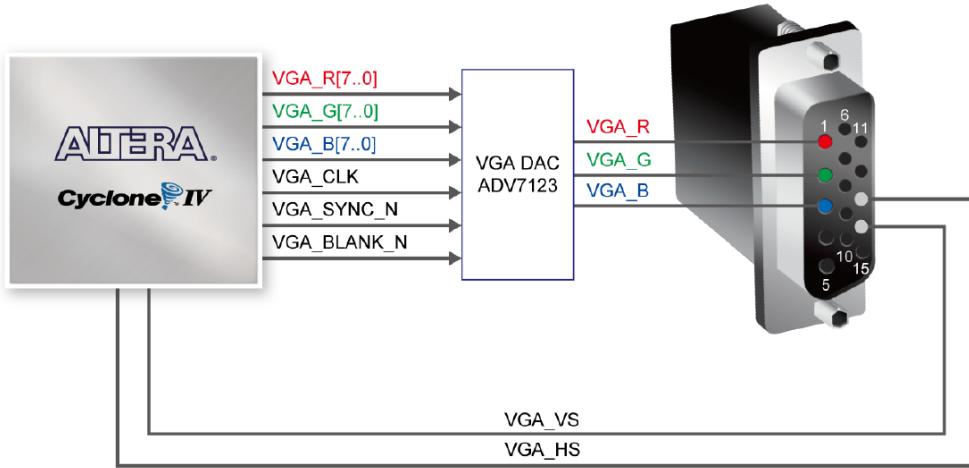


Figura 32 – Conexões entre o FPGA e a saída de vídeo VGA na placa DE2-115.

Fonte: (TERASIC, 2017)

conectados ao FPGA. Os pinos R0, R1, G0, G1, B0 e B1 do ADV7123 são conectados ao sinal de terra da placa, e as saídas analógicas IOR, IOG e IOB são ligadas ao conector D-SUB da saída de vídeo VGA. Com cada componente do sinal de vídeo sendo representada por 8 bits, suas respectivas saídas analógicas podem assumir individualmente 256 níveis distintos e, quando combinadas, podem produzir *pixels* de 16.777.216 cores diferentes.

Além dos três conversores digital-analógico integrados, o *chip* ADV7123 conta também com uma lógica de controle baseada nas entradas *BLANK* e *SYNC*. O pino de controle *BLANK* é ativo em nível lógico baixo e, quando acionado, ignora as entradas de *pixel* R0 a R9, G0 a G9 e B0 a B9, mantendo as saídas analógicas IOR, IOB e IOG em nível de apagamento (ANALOG DEVICES, 2010). Esta entrada deve ser ativada nos momentos em que não há exibição de *pixels* na tela, ou seja, durante o intervalo formado pelo *front porch*, pulso de sincronismo e *back porch* dos sincronismos horizontal e vertical.

O sinal de controle *SYNC* é utilizado para embutir um sinal de sincronismo composto, contendo a combinação dos sincronismos horizontal e vertical, à saída analógica da componente verde (ANALOG DEVICES, 2010). Alguns fabricantes utilizam esta tecnologia para reduzir o número de conexões exigidas na transmissão do sinal de vídeo, visto que os sinais de sincronismo horizontal e vertical não precisam ser enviados em linhas separadas. No projeto desenvolvido não foi elaborada nenhuma lógica para o controle do pino *SYNC*, por não ser considerado um sinal essencial ao funcionamento de grande parte dos dispositivos que seguem o padrão VGA.

O módulo *VGA_Interface*, ilustrado na Figura 33, foi concebido para gerenciar todos os sinais necessários ao funcionamento da saída de vídeo VGA. Devido à resolução e taxa de quadros adotados, este é o único bloco da arquitetura que trabalha com *clock* de 25 MHz, visto que os *pixels* precisam ser varridos nesta frequência. Deste modo, a cada

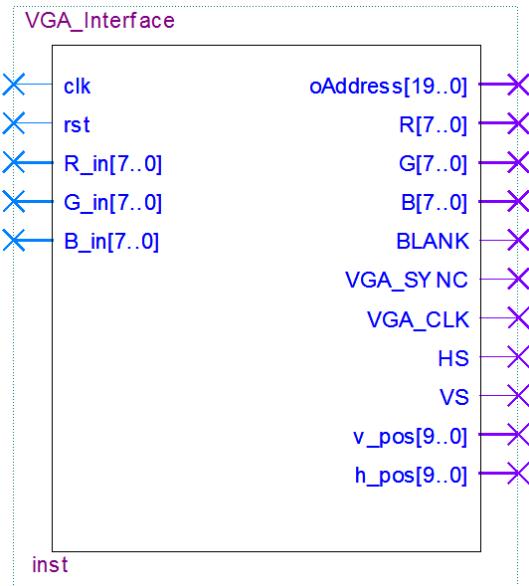


Figura 33 – Representação em bloco do módulo *VGA_Interface*.

ciclo de *clock* os dados presentes nas entradas *R_in*, *G_in* e *B_in* são encaminhados para as saídas *R*, *G* e *B*, produzindo um novo *pixel*. Estes pinos de saída são conectados ao *chip* ADV7123, assim como o pino *VGA_CLK*, que direciona o *clock* de 25 MHz recebido na entrada *clk*.

Os sinais de controle produzidos por este bloco são baseados em dois contadores de 10 bits, que indicam as posições relativas das varreduras horizontal e vertical, especificando a localização do *pixel* corrente. O contador *h_pos* controla a varredura horizontal da tela, sendo incrementado a cada borda de subida do *clock* de 25 MHz que alimenta o módulo. Dado que o período do sincronismo horizontal é de 32 μ s, ou o equivalente a 800 *pixels*, este contador é zerado ao atingir o valor de 799.

A varredura vertical da tela é controlada pelo contador *v_pos*, incrementado toda vez que o contador *h_pos* alcança o valor de 799. O contador *v_pos* é zerado sempre que chega ao valor de 524, visto que o período do sincronismo vertical é de 16,8 ms, ou o correspondente a 525 linhas. Os sinais de saída destes contadores ficam disponíveis nos pinos *h_pos* e *v_pos* do bloco, sendo utilizados por outros módulos que precisam de alguma referência relacionada à varredura da tela.

Conforme ilustrado na Figura 31, a contagem dos *pixels* durante a varredura horizontal começa partindo da região de *back porch*, trecho que ocorre entre os *pixels* 0 ao 47. Em seguida acontece o intervalo de exibição entre os *pixels* 48 ao 687, sendo sucedido pela área de *back porch* do *pixel* 688 ao 703. Por fim ocorre o pulso de sincronismo horizontal entre os *pixels* 704 ao 799, intervalo em que o sinal *HS* do bloco deixa o nível lógico alto e é mantido em nível lógico baixo.

De maneira similar, a contagem das linhas durante a varredura vertical se inicia no intervalo de *back porch*, entre as linhas 0 a 32. Logo após, entre as linhas 33 a 512, ocorre o intervalo de exibição, que é acompanhado pela região de *front porch* entre as linhas 513 a 522. Finalmente, o pulso de sincronismo vertical é gerado durante as linhas 523 e 524, período em que o sinal *VS* do bloco é alternado do nível lógico alto e mantido em nível lógico baixo. Esta contagem também é demonstrada na [Figura 31](#).

O sinal de apagamento *BLANK*, fornecido ao *chip* ADV7123, é mantido em nível lógico baixo apenas durante o intervalo formado pela união do *front porch*, pulso de sincronismo e *back porch*, dos sincronismos horizontal e vertical. Deste modo, o pino *BLANK* do módulo assume estado lógico 0 durante as linhas 0 a 32 e 513 a 524, assim como nos *pixels* 0 a 47 e 688 a 799 de todas as linhas.

Além de gerenciar os sinais da saída de vídeo VGA, este módulo é responsável também por produzir os endereços da memória SRAM, que devem ser acessados para recuperar os dados dos *pixels* da imagem de fundo que foi programada. Estes endereços são gerados através de um contador, que é incrementado a cada pulso do *clock*, e funciona apenas na região visível do vídeo, isto é, entre os *pixels* 48 a 687 das linhas 33 a 512. Esta área contém 480 linhas e 640 colunas, resultando num total 307.200 *pixels*. A imagem de fundo que deve ser exibida possui as mesmas dimensões, e cada um de seus *pixels* é armazenado em endereços distintos e sequenciais da memória. Como consequência, apenas os primeiros 307.200 endereços da memória SRAM precisam ser acessados, levando o contador a ser zerado sempre que alcança o valor 307.199. O valor deste contador é propagado para a saída de endereço *oAddress* do bloco.

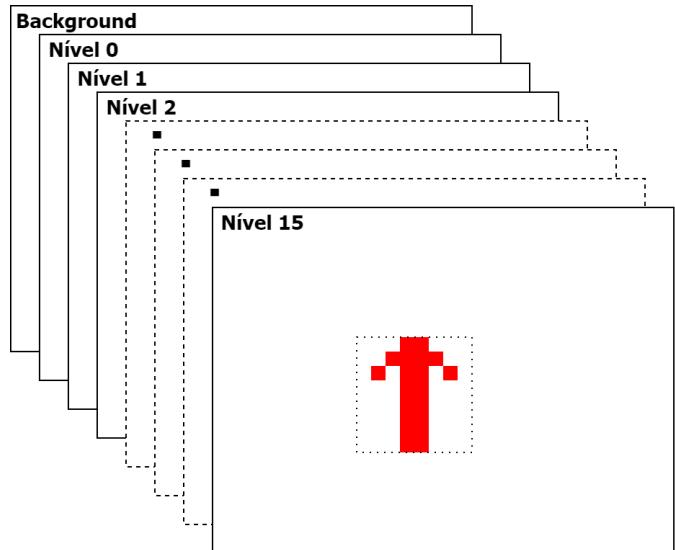
3.3.7 Leitor de Sprites

Os jogos executados na plataforma desenvolvida são animados através de elementos gráficos móveis chamados de *sprites*. Cada *sprite* possui apenas uma cor, e é construído a partir de uma matriz quadrada com tamanho fixo de 16 por 16 *pixels*. O formato desejado para o componente pode ser obtido através da ativação ou desativação dos *pixels* que compõem a matriz. Este processo de formatação do *sprite* é modelado através da associação de uma matriz de 16 por 16 bits com a matriz de 16 por 16 *pixels* citada. Assim, o bit 1 em dada posição da matriz de bits indica que o *pixel* associado na matriz de *pixels* deve ser exibido, enquanto que o bit 0 indica que o *pixel* deve permanecer desativado. Este procedimento é ilustrado na [Figura 34\(a\)](#), onde foi concebido um *sprite* na cor vermelha com o formato de uma seta apontada para cima.

A arquitetura elaborada é capaz de exibir até 16 *sprites* simultaneamente na tela, em diferentes níveis de sobreposição. Deste modo, além do posicionamento horizontal e vertical do *sprite*, é possível definir também em qual nível de sobreposição ele será exibido. Esta técnica permite uma maior flexibilidade no desenvolvimento dos gráficos,

possibilitando, por exemplo, a união de vários *sprites* para formar objetos com tamanhos maiores que 16 por 16 *pixels* e com mais de uma cor.

0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	1	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0
0	0	1	1	1	0	1	1	1	1	0	1	1	0	0	0	0
0	0	1	1	0	0	1	1	1	1	0	0	1	1	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0

(a) Formatação de um *sprite*.

(b) Camadas de sobreposição da imagem.

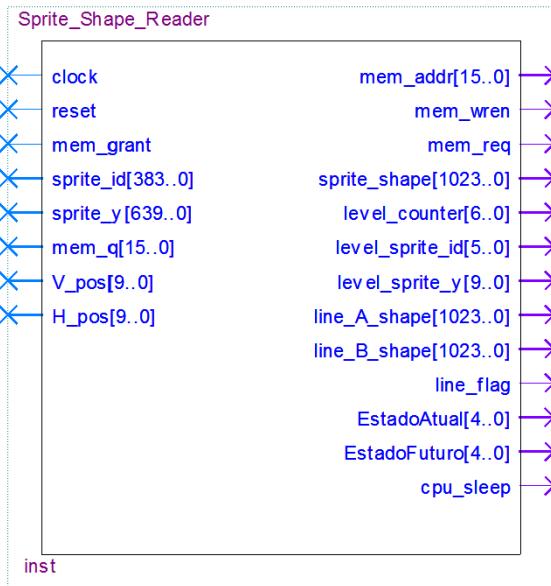
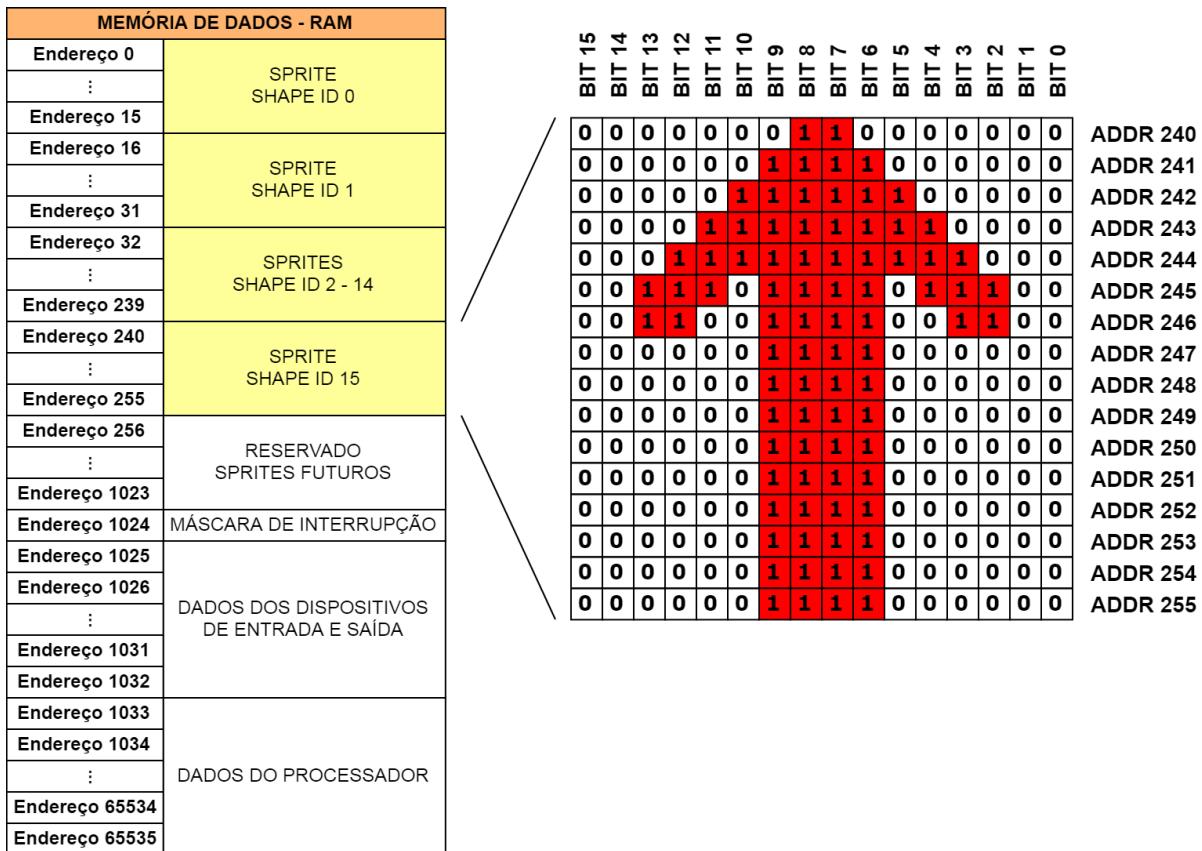
Figura 34 – Construção e posicionamento de um *sprite*.

Cada nível de sobreposição pode ter apenas um *sprite* associado, e a imagem final produzida pelo protótipo é construída a partir da composição da imagem de fundo (*background*) com os 16 níveis de sobreposição dos *sprites*. Estes 17 planos seguem uma ordem de prioridade durante a exibição, com o *sprite* do nível 15 sendo o mais prioritário e o *background* assumindo a menor prioridade. Por consequência, o *background* sempre será sobreposto pelos *sprites* e, se houver interseção na posição de dois *sprites*, os *pixels* ativados do *sprite* mais prioritário serão exibidos e ocultarão o *sprite* da camada inferior. A Figura 34(b) demonstra o arranjo dos planos que formam a imagem.

As matrizes de bits que definem os formatos dos *sprites* são armazenadas na memória de dados do sistema, e a leitura destes dados precisa ser feita de modo que estejam disponíveis para exibição no tempo correto. O módulo *Sprite_Shape_Reader*, exibido na Figura 35, foi desenvolvido com a finalidade de recuperar da memória os *shapes* dos *sprites* com a antecedência necessária e disponibilizá-los para serem exibidos.

A Figura 36 exibe os segmentos da memória de dados e a região destinada aos *shapes* dos *sprites*. Cada endereço de memória é capaz de armazenar uma linha do *shape*, sendo necessários 16 endereços de memória para salvá-lo completamente. Deste modo, os 16 *sprites* são registrados entre os endereços 0 ao 255, e os endereços 256 ao 1023 foram reservados para salvar novos *shapes* em aplicações futuras.

De modo a prover os *shapes* no período adequado, o módulo *Sprite_Shape_Reader* trabalha entre as linhas 32 e 511 da imagem, iniciando e finalizando o processo de leitura dos *sprites* com uma linha de antecedência ao intervalo de exibição do vídeo. Os *shapes* são

Figura 35 – Representação em bloco do módulo *Sprite_Shape_Reader*.Figura 36 – Segmentos da memória de dados e os *shapes* dos *sprites*.

lidos linha após linha, e dois registradores são utilizados para salvar os dados recuperados. O primeiro armazena a linha, de cada *shape*, que está sendo exibida naquele momento, e o segundo armazena a linha, de cada *shape*, que será exibida em seguida. A leitura de um *shape* é iniciada durante a linha de vídeo anterior à linha em que o *sprite* será

exibido, ou seja, um *sprite* exibido nas linhas 50 a 65 tem seu *shape* lido entre as linhas 49 e 64. As linhas dos *shapes* disponíveis nos registradores são direcionadas para o módulo *Sprite_Processor*, que fará a composição dos *sprites* com o *background* e produzirá a imagem final. A Figura 37 expõe uma máquina de estados finita com a lógica de funcionamento do *Sprite_Shape_Reader*.

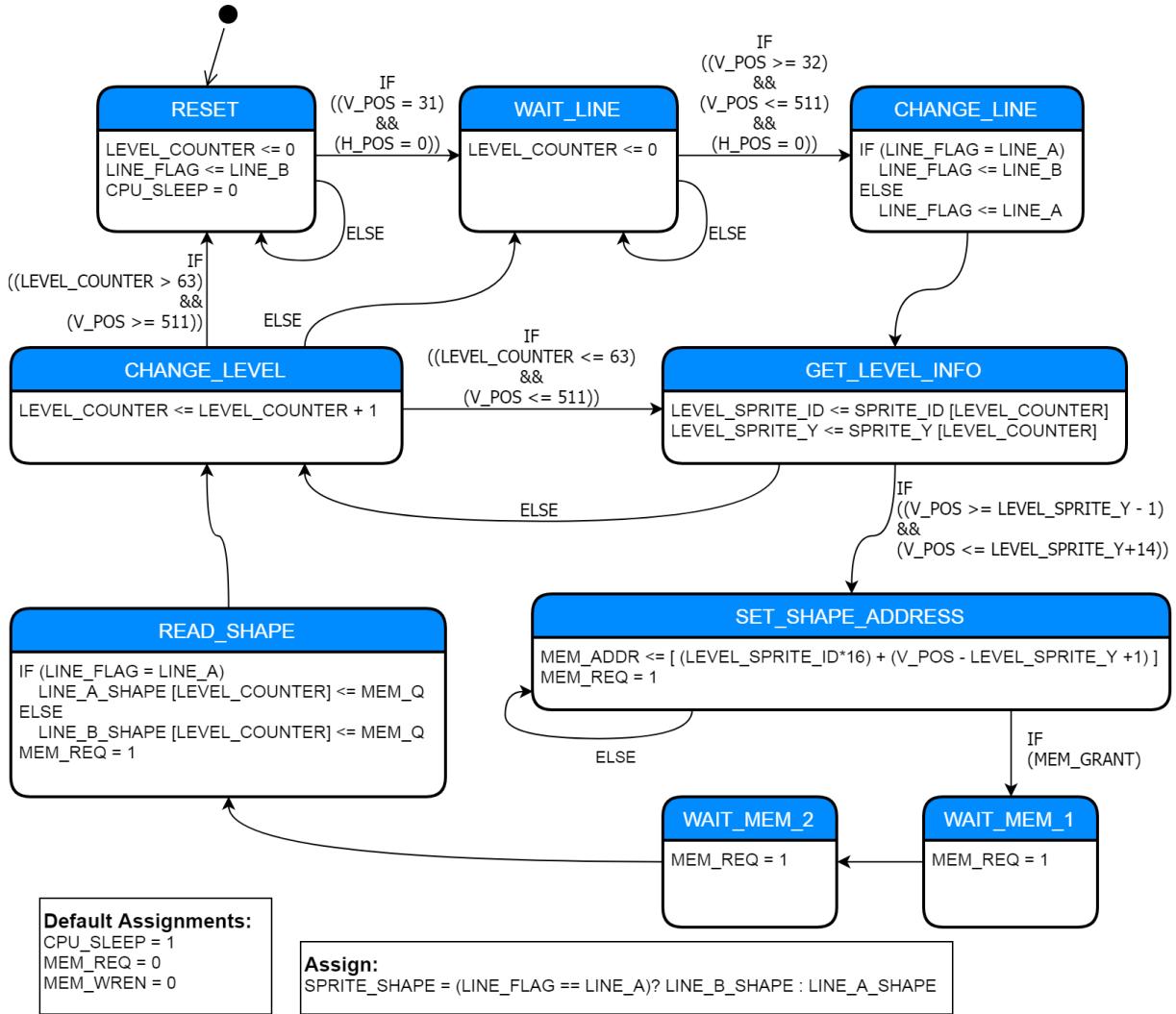


Figura 37 – Diagrama de estados do módulo *Sprite_Shape_Reader*.

O bloco *Sprite_Shape_Reader* produz também um sinal para controle do módulo *Processor_Controller*, chamado de *cpu_sleep*. Este sinal tem como função ativar o *Processor_Controller* apenas no período inativo da imagem, de modo que não hajam mudanças nos atributos dos *sprites* durante a construção da área visível da imagem, o que poderia provocar falhas perceptíveis no vídeo.

3.3.8 Processador de Sprites

A consolidação dos elementos gráficos, para construir a imagem final exibida pela plataforma, é realizada pelo módulo *Sprite_Processor*. O bloco tem um funcionamento simplificado, fazendo a priorização do que deve ser exibido a partir das componentes de cor da imagem de *background* e dos atributos de todos os *sprites*. Estes atributos são a cor, as posições horizontal e vertical, o nível de sobreposição ao qual está associado e a linha do *shape* que está sendo exibida. A cada pulso de *clock* são produzidas as componentes de cor de um *pixel* da imagem final. Este *pixel* será referente ao elemento de maior prioridade de exibição, podendo ser da imagem de *background* ou de um dos *sprites*, dependendo de como estes elementos estão sobrepostos. No caso dos *sprites*, o bit 0 de um *shape* funciona como uma transparência, permitindo que elementos de camadas inferiores sejam exibidos quando houver sobreposição. A Figura 38 ilustra o módulo *Sprite_Processor*.

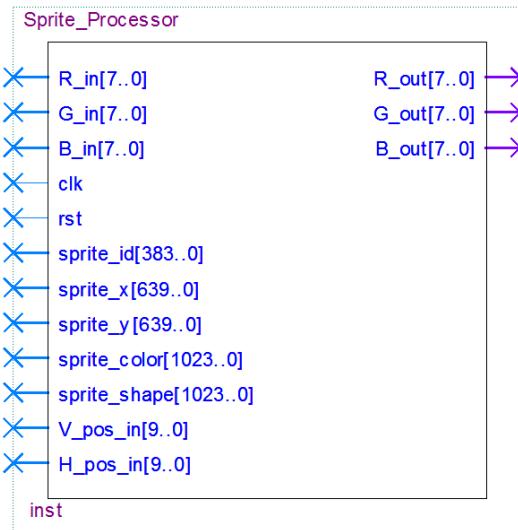


Figura 38 – Representação em bloco do módulo *Sprite_Processor*.

3.3.9 Unidade de Controle do Processador

A função de um processador é executar programas armazenados na memória, buscando suas instruções, examinando-as e então executando-as uma após a outra. Um processador é composto por várias partes distintas. A unidade de controle é responsável por buscar instruções na memória e determinar seu tipo. A unidade lógica e aritmética efetua operações como adição e AND booleano para executar as instruções.

O processador também contém uma pequena memória de alta velocidade usada para armazenar resultados temporários e informações de controle. Esta memória é dividida em registradores, cada um deles com certo tamanho e função, que podem ser lidos e escritos rapidamente porque são internos ao processador. Cada registrador pode armazenar um

valor, cujo máximo é determinado pela quantidade de bits que ele contém (TANENBAUM, 2007).

Na arquitetura concebida, o módulo *Processor_Controller* funciona como um processador, gerenciando os registradores e executando as funções da unidade de controle e da unidade lógica e aritmética. Cinco *IP Cores* foram acoplados externamente a este bloco para auxiliá-lo na realização das operações de soma, subtração, multiplicação, divisão e comparação. Estes *IP Cores* recebem os dados do *Processor_Controller*, efetuam a operação, e devolvem o resultado para que o módulo conclua a execução da instrução. Dado que todas as demais operações são realizadas internamente no bloco *Processor_Controller*, os *IP Cores* podem ser considerados uma extensão da unidade lógica aritmética.

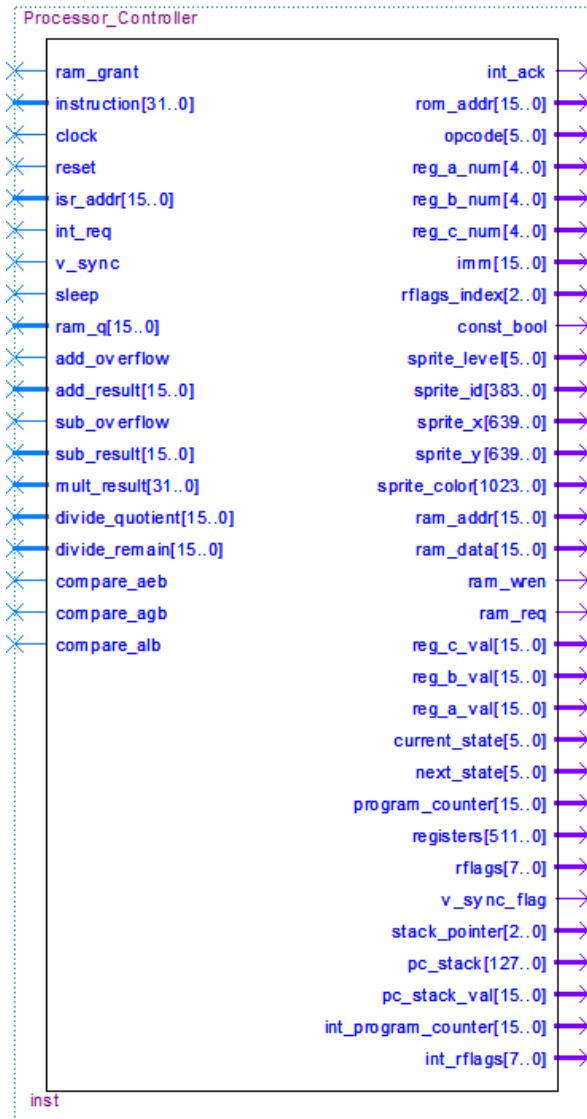
O módulo *Processor_Controller*, exibido na Figura 39, trabalha com *clock* de 50 MHz e foi modelado como um processador de arquitetura RISC (*Reduced Instruction Set Computer*), capaz de executar apenas 24 instruções simples. A execução das instruções ocorre de forma serial, isto é, uma nova instrução só é executada após a finalização da instrução anterior. Este modelo foi adotado para simplificar o projeto, dado as complexidades existentes na concepção de um processador com *pipeline*.

A Figura 40 expõe o funcionamento do bloco através de uma máquina de estados finita. Em benefício da simplicidade, os estados de execução das 24 instruções foram ocultados e unificados em um único estado de execução, colorido em vermelho.

É possível observar que, além da busca, decodificação e execução das instruções, o módulo conta também com uma lógica para atender à interrupções. Quando acionado pelo bloco *Interrupt_Controller*, o processador sinaliza o atendimento da requisição, salva o contexto dos registradores de status *RFlags* e de contador de programa *Program_Counter*, e desvia a execução para o endereço *ISR_Addr* informado pelo *Interrupt_Controller*. Quando a rotina de interrupção é finalizada, através da instrução *IRET*, o processador comunica o fim da requisição e restaura os valores dos registradores *RFlags* e *Program_Counter*.

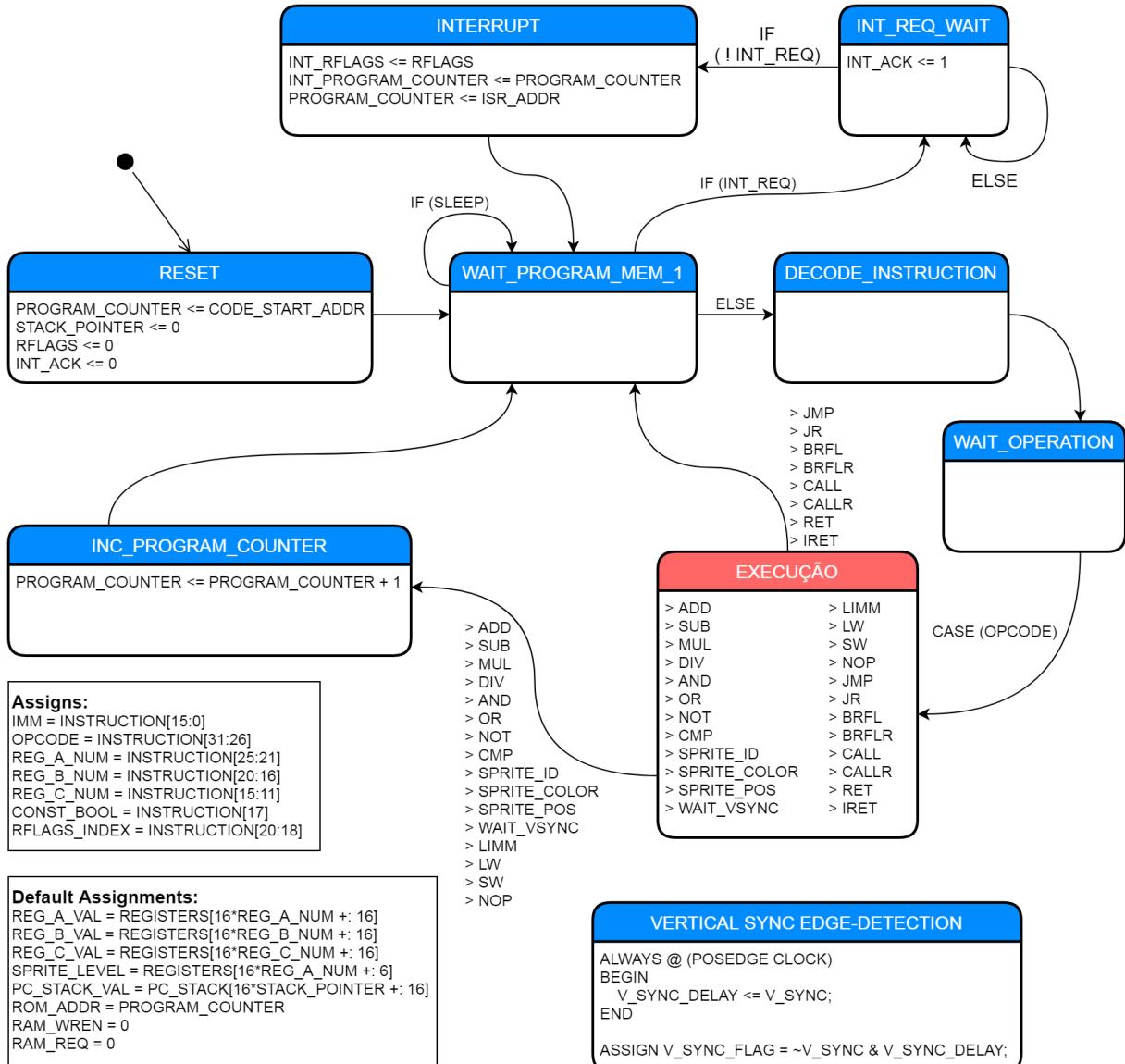
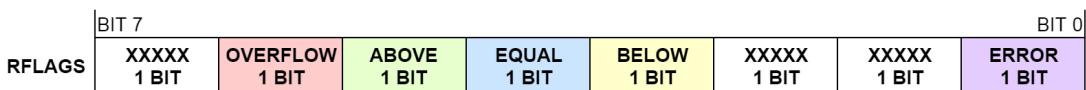
O registrador de status *RFlags*, apresentado na Figura 41, tem como papel armazenar as sinalizações geradas pelas operações executadas no processador. Apesar de contar com 8 bits, apenas 5 deles são utilizados. O bit 6 deste registrador indica quando ocorre *overflow* nas operações aritméticas, isto é, quando o resultado da operação tem valor maior que o possível de ser representado nos 16 bits dos registradores de propósito geral. O bit 0 sinaliza um erro da operação de divisão, que ocorre quando o valor do denominador é igual a zero. Por fim, os bits 3, 4 e 5 são usados na operação de comparação, para indicar quando o operando A é menor, igual ou maior que o operando B, respectivamente. Os dados registrados no *RFlags* podem ser lidos através das instruções *BRFL* e *BRFLR*, usadas para fazer desvios na execução do código.

Além do registrador especial *RFlags*, existem 32 registradores de 16 bits que são

Figura 39 – Representação em bloco do módulo *Processor_Controller*.

de propósito geral. Estes registradores são designados como R0 a R31 e estão acessíveis para serem lidos e escritos através das instruções do processador. Esta quantidade de bits permite que os registradores armazenem valores entre 0 e 65.535 ou entre -32.768 e 32.767, quando utilizada a representação de complemento de dois. A representação aplicada irá depender da instrução que está usando o registrador. Instruções aritméticas, por exemplo, irão utilizar a representação de complemento de dois. Já instruções de acesso à memória utilizam a faixa de 0 a 65.535, visto que não faz sentido um valor de endereço negativo.

Para permitir que o desenvolvedor de um jogo possa sincronizar o funcionamento do processador com os quadros da imagem, foi embutido no módulo um circuito de detecção do pulso de sincronismo vertical do vídeo, como pode ser visto na Figura 40. A sinalização produzida por este circuito é usada pela instrução *WAIT_VSYNC*, que suspende a execução do processador até o reconhecimento do próximo pulso de sincronismo

Figura 40 – Diagrama de estados do módulo *Processor_Controller*.Figura 41 – Representação do registrador de status *RFlags*.

vertical.

Por se tratar de um processador capaz de executar instruções gráficas, algumas peculiaridades precisam ser atendidas para evitar defeitos na imagem. Devido à alta velocidade do processador, que trabalha com frequência de 50 MHz, em comparação com a taxa de atualização da tela, de aproximadamente 60 Hz, milhares de instruções poderiam ser executadas durante o intervalo de exibição de apenas um quadro da imagem. Se entre os comandos executados houvessem instruções gráficas, modificando os atributos dos *sprites* ao longo da formação de um quadro, é possível que pequenas falhas fossem

observadas na imagem. Por conta desta característica o processador deve ser desativado durante o intervalo de exibição do vídeo, o que pode ser feito através do pino *sleep*. O controle deste sinal é feito pelo bloco *Sprite_Shape_Reader*, que mantém o processador em funcionamento apenas nos intervalos verticais inativos do vídeo.

As instruções executadas pelo *Processor_Controller* possuem 32 bits de largura e são formatadas conforme os modelos apresentados na [Figura 42](#). Cada instrução deve ser acompanhada dos dados necessários para sua execução, entretanto, estes dados podem ser diferentes dependendo da operação que precisa ser realizada. Como consequência, três modelos de instruções foram desenvolvidos para atender aos requisitos de todas as operações. A função de cada campo das instruções é descrita abaixo:

- *OPCODE*: usado pelo processador para identificar a instrução e quais ações devem ser tomadas;
- *REG_A*, *REG_B* e *REG_C*: indicam o número dos registradores que serão utilizados na operação;
- *IMM*: Valor de 16 bits armazenado diretamente na instrução, ao invés de estar salvo em um registrador. É usado em instruções de transferência de dados e de transferência de controle na indicação de endereços de memória;
- *RFLAGS_INDEX*: indica o índice de um dos bits do registrador especial *RFlags*. Este campo é usados nas instruções *BRFL* e *BRFLR* para especificar qual bit deve ser lido;
- *CONST_BOOL*: Usado pelas instruções *BRFL* e *BRFLR* para especificar o valor booleano que deve ser comparado com o bit *RFLAGS_INDEX* do registrador *RFlags*.

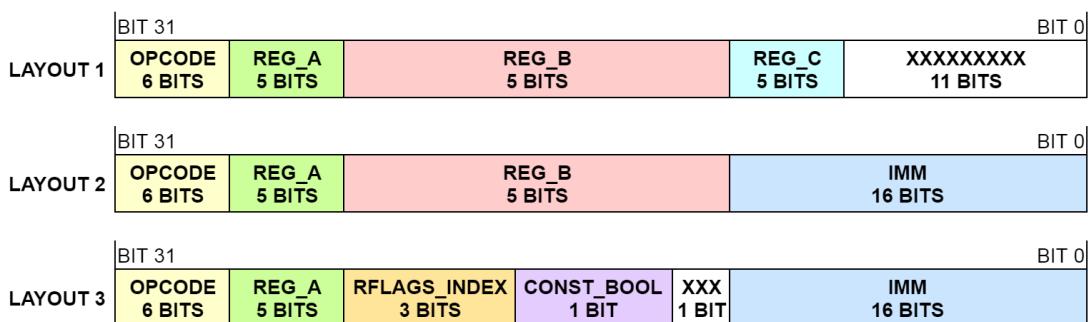


Figura 42 – Formatos das instruções executadas pelo *Processor_Controller*.

A seguir são resumidas em quadros as características das instruções, que foram divididas de acordo com a função. O [Quadro 2](#) apresenta as instruções de transferência de dados.

Quadro 2 – Descrição das instruções de transferência de dados

Instruções de Transferência de Dados			
Opcode	Mnemônico	Operandos	Operação
001100	LIMM	Ra, Imm	$Ra \leftarrow Imm$
001001	LW	Ra, Imm (Rb)	$Ra \leftarrow MEM[Rb + Imm]$
001010	SW	Ra, Imm (Rb)	$MEM[Rb + Imm] \leftarrow Ra$

A instrução *LIMM* armazena o valor imediato *Imm* de 16 bits no registrador de número *Ra*. A instrução *LW* carrega no registrador *Ra* o valor salvo no endereço de memória especificado, enquanto que a instrução *SW* salva no endereço de memória especificado o valor do registrador *Ra*. O endereço pode ser especificado de três modos diferentes, que são através do registrador *Rb*, por meio do valor imediato *Imm* ou a partir da soma dos dois valores, usando o modo base-deslocamento.

As instruções aritméticas, especificadas no [Quadro 3](#), são capazes de fazer operações apenas com números inteiros. Todas as instruções deste grupo se comportam de maneira semelhante, com o resultado da operação entre os registradores *Ra* e *Rb* sendo salvos no registrador *Ra*. Caso ocorra *overflow* nas operações, o registrador *Ra* assume valor -1 e o bit 6 do registrador *RFlags* é configurado com valor 1. Na operação de divisão, em especial, o registrador *Ra* também assume valor -1 se o valor do registrador *Rb* for 0, visto que a divisão por zero é indefinida. Este erro é sinalizado por meio do bit 0 do registrador *RFlags*, que assume valor 1 nestes casos.

O [Quadro 4](#) traz a descrição das instruções lógicas. As instruções *AND*, *OR* e *NOT* realizam as operações booleanas homônimas de forma bit a bit, com os resultados sendo salvos no registrador *Ra*. No caso das instruções *AND* e *OR*, cada bit do registrador *Ra* é operado com o bit correspondente do registrador *Rb*, ao passo que na instrução *NOT* todos os bits do registrador *Ra* são invertidos. A operação *CMP* analisa o valor do registrador *Ra* comparado ao *Rb* e configura os bits 3, 4 e 5 do registrador especial *RFlags*. Estes bits assumem valor 1 para indicar que *Ra* é menor, igual ou maior que *Rb*, respectivamente.

O [Quadro 5](#) demonstra as operações disponíveis para controlar os elementos gráficos da plataforma. A exibição de um *sprite* depende basicamente da definição de três atributos, que são o formato (*shape*), a cor e a posição na tela. Estas configurações podem ser feitas através das instruções *SPRITE_ID*, *SPRITE_COLOR* e *SPRITE_POS*, respectivamente. Como as definições de apenas um *sprite* podem ser associadas a cada nível de sobreposição da imagem, é preciso especificá-lo em todas estas instruções através do registrador *Ra*.

A instrução *SPRITE_ID* define através do registrador *Rb* a identificação do *sprite* associado a determinado nível, isto é, o número que identifica o *shape* que será exibido em dado nível de sobreposição.

A instrução *SPRITE_COLOR*, por sua vez, define através do registrador *Rb* a

Quadro 3 – Descrição das instruções aritméticas

Instruções Aritméticas			
Opcode	Mnemônico	Operandos	Operação
010001	ADD	Ra, Rb	$Ra \leftarrow Ra + Rb$ $RFlags[6] \leftarrow 0$ <i>Overflow:</i> $Ra \leftarrow -1$ $RFlags[6] \leftarrow 1$
010010	SUB	Ra, Rb	$Ra \leftarrow Ra - Rb$ $RFlags[6] \leftarrow 0$ <i>Overflow:</i> $Ra \leftarrow -1$ $RFlags[6] \leftarrow 1$
010100	MUL	Ra, Rb	$Ra \leftarrow Ra * Rb$ $RFlags[6] \leftarrow 0$ <i>Overflow:</i> $Ra \leftarrow -1$ $RFlags[6] \leftarrow 1$
010101	DIV	Ra, Rb	$Ra \leftarrow Ra / Rb$ $RFlags[0] \leftarrow 0$ $RFlags[6] \leftarrow 0$ <i>Overflow:</i> $Ra \leftarrow -1$ $RFlags[0] \leftarrow 0$ $RFlags[6] \leftarrow 1$ <i>Rb = 0 Error:</i> $Ra \leftarrow -1$ $RFlags[0] \leftarrow 1$ $RFlags[6] \leftarrow 0$

cor associada a determinado nível. O *sprite* associado a este mesmo nível assumirá a cor configurada. Os 16 bits do registrador são divididos de modo a representar as três componentes básicas de cor, com os bits 0 ao 4 representando o vermelho, os bits 5 ao 10 representando o verde, e os bits 11 ao 15 representando o azul.

Por fim, a instrução *SPRITE_POS* configura a posição do *sprite* associado a determinado nível. Este posicionamento é feito tomando como base a linha e a coluna da extremidade superior esquerda do *sprite*. O registrador *Rb* é usado para definir a linha enquanto que o registrador *Rc* define a coluna.

Uma referência de tempo pode ser obtida durante a execução do código através da

Quadro 4 – Descrição das instruções lógicas

Instruções Lógicas			
Opcode	Mnemônico	Operandos	Operação
100001	AND	Ra, Rb	$Ra \leftarrow Ra \text{ AND } Rb$
100010	OR	Ra, Rb	$Ra \leftarrow Ra \text{ OR } Rb$
100101	NOT	Ra	$Ra \leftarrow \text{NOT } Ra$
100100	CMP	Ra, Rb	<p><i>Se $Ra < Rb$:</i> $\text{RFlags}[3] \leftarrow 1$ $\text{RFlags}[4] \leftarrow 0$ $\text{RFlags}[5] \leftarrow 0$</p> <p><i>Se $Ra = Rb$:</i> $\text{RFlags}[3] \leftarrow 0$ $\text{RFlags}[4] \leftarrow 1$ $\text{RFlags}[5] \leftarrow 0$</p> <p><i>Se $Ra > Rb$:</i> $\text{RFlags}[3] \leftarrow 0$ $\text{RFlags}[4] \leftarrow 0$ $\text{RFlags}[5] \leftarrow 1$</p>

instrução *WAIT_VSYNC*. Quando executada, o processador é suspenso até o próximo pulso de sincronismo vertical da imagem, sinalizando a velocidade de exibição dos quadros. Esta instrução pode ser utilizada, por exemplo, para controlar a velocidade do movimento dos *sprites*.

Quadro 5 – Descrição das instruções gráficas

Instruções Gráficas			
Opcode	Mnemônico	Operandos	Operação
110001	SPRITE_ID	Ra, Rb	<i>Define o sprite associado a um nível:</i> $\text{Sprite_Level} \leftarrow Ra$ $\text{Sprite_ID} \leftarrow Rb$
110010	SPRITE_COLOR	Ra, Rb	<i>Define a cor associada a um nível:</i> $\text{Sprite_Level} \leftarrow Ra$ $\text{Sprite_Color} \leftarrow Rb$
110100	SPRITE_POS	Ra, Rb, Rc	<i>Define a posição do sprite associado a um nível:</i> $\text{Sprite_Level} \leftarrow Ra$ $\text{Sprite_Row} \leftarrow Rb$ $\text{Sprite_Col} \leftarrow Rc$
110111	WAIT_VSYNC	Nenhum	<i>Suspende o processador até o próximo pulso de sincronismo vertical da imagem.</i>

As instruções de transferência de controle, exibidas no [Quadro 6](#), têm como incumbência desviar a sequência de execução do código. A exceção é a instrução *NOP*, que não

executa nenhuma operação e é utilizada apenas para consumir tempo de processamento do processador. As instruções *JR* e *JMP* quando executadas agem de forma similar, realizando um desvio incondicional para o endereço especificado. Entretanto, a primeira define o endereço do desvio através do registrador *Ra*, enquanto a segunda utiliza um *label* para esta função. O *label* é uma espécie de rótulo posicionado no código e convertido para um endereço pelo *assembler*, sendo armazenado em seguida no campo *IMM* da instrução.

As instruções *BRFL* e *BRFLR* são utilizadas para executar desvios condicionais. As duas também funcionam de maneira semelhante, com a primeira definindo o endereço do salto através do registrador *Ra*, e a segunda utilizando um *label* para esta função. O desvio é condicionado à comparação do valor booleano *const_bool* com o valor de um dos bits do registrador *RFlags*, definido através da índice *rflags_index*. O desvio é feito caso os valores sejam iguais e, caso não sejam, a execução continua de modo sequencial.

As instruções *CALL* e *CALLR* são aplicadas na chamada de sub-rotinas. Estas instruções se diferenciam das instruções *JR* e *JMP*, pois armazenam o endereço da instrução seguinte antes de realizarem o salto incondicional. Deste modo, ao final da sub-rotina, a instrução *RET* é executada e o processador retorna para a posição salva. A instrução *CALL* define o endereço de início da sub-rotina através de um *label*, enquanto que a instrução *CALLR* utiliza o registrador *Ra* para isto. Os endereços de retorno das sub-rotinas são armazenados em uma pilha construída no interior do processador. Esta pilha tem capacidade para salvar até oito endereços, o que limita a chamada de até oito sub-rotinas de forma encadeada.

A instrução *IRET* tem como função indicar ao processador o final de uma rotina de interrupção. Ao ser executada, os registradores *Program_Counter* e *RFlags* são restaurados para os valores anteriores à interrupção, permitindo ao processador retomar a execução do código no mesmo ponto em que foi interrompido.

Quadro 6 – Descrição das instruções de transferência de controle

Instruções de Transferência de Controle			
Opcode	Mnemônico	Operandos	Operação
101110	NOP	Nenhum	Nenhuma operação
101001	JMP	Label	$PC \leftarrow Label$
011001	JR	Ra	$PC \leftarrow Ra$
101010	BRFL	Label, rflags_index, const_bool	<i>Se RFlags[rflags_index] = const_bool:</i> $PC \leftarrow Label$
011010	BRFLR	Ra, rflags_index, const_bool	<i>Se RFlags[rflags_index] = const_bool:</i> $PC \leftarrow Ra$
101011	CALL	Label	$PC_Stack[SP] \leftarrow PC + 1$ $PC \leftarrow Label$ $SP \leftarrow SP + 1$
011011	CALLR	Ra	$PC_Stack[SP] \leftarrow PC + 1$ $PC \leftarrow Ra$ $SP \leftarrow SP + 1$
101100	RET	Nenhum	$PC \leftarrow PC_Stack[SP-1]$ $SP \leftarrow SP - 1$
101101	IRET	Nenhum	$Int_Ack \leftarrow 0$ $RFlags \leftarrow Int_RFlags$ $PC \leftarrow Int_PC$

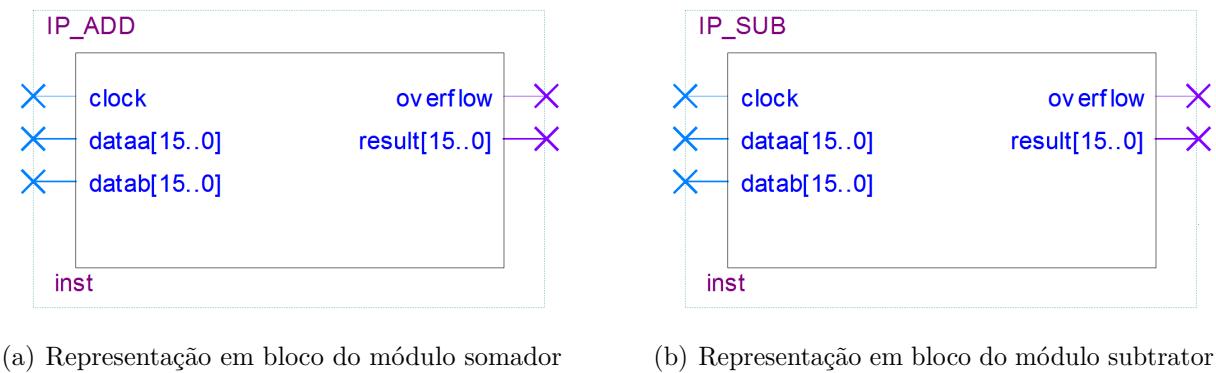
3.4 IP cores Utilizados

Através de seu *software* de projeto *Quartus Prime Lite*, a *Intel* fornece uma série de *IP cores* para as mais diversas funções. Por se tratarem de blocos otimizados e validados, utilizá-los reduz o tempo de desenvolvimento e maximiza o desempenho do circuito. Estas vantagens estimularam o uso de *IP cores* nas atividades em que havia viabilidade, como no gerenciamento do *clock*, operações aritméticas e implementação de memórias *on-chip*. A ferramenta *MegaWizard Plug-In Manager*, disponível no *software Quartus Prime Lite*, foi empregada para auxiliar a personalização e parametrização de todos os *IP cores* integrados no projeto.

3.4.1 Somador/Subtrator

As operações de soma e subtração requisitadas ao processador são executadas por instâncias do *IP core LPM_ADD_SUB*. Este módulo pode funcionar como um somador ou um subtrator de números inteiros, obedecendo aos parâmetros definidos no momento de sua implementação.

Um somador e um subtrator foram utilizados na arquitetura a partir de duas instâncias do *LPM_ADD_SUB*. Estes módulos são ilustrados na Figura 43 e uma descrição de suas entradas e saídas podem ser consultadas respectivamente na Tabela 2 e na Tabela 3.



(a) Representação em bloco do módulo somador

(b) Representação em bloco do módulo subtrator

Figura 43 – Representações dos módulos *IP_ADD* e *IP_SUB*.

Os dois blocos são capazes de fazer operações sobre dados de 16 bits com sinal e utilizam o recurso de *pipeline*. Este mecanismo foi configurado com latência de 1 ciclo de *clock* a fim de sincronizar as saídas dos módulos e evitar problemas com a temporização do circuito. A Tabela 4 e a Tabela 5 contêm os valores dos parâmetros utilizados para configurar as instâncias do somador e do subtrator, respectivamente.

Tabela 2 – Portas de entrada dos módulos *IP_ADD* e *IP_SUB*

Nome da porta	Descrição
dataaa[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTH.
datab[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTH.
clock	Entrada para uso em pipeline. Fornece a entrada de sincronismo para uma operação em pipeline. Para valores do parâmetro LPM_PIPELINE diferentes de 0 (padrão), a porta de <i>clock</i> deve estar ativada.

Fonte: ([INTEL, 2017b](#))Tabela 3 – Portas de saída dos módulos *IP_ADD* e *IP_SUB*

Nome da porta	Descrição
result[15..0]	Saída de dados. O tamanho da porta de saída depende do valor do parâmetro LPM_WIDTH.
overflow	Saída de exceção de <i>overflow</i> . A porta de <i>overflow</i> é ativada quando os resultados excedem a precisão disponível, e é usada somente quando o valor do parâmetro LPM REPRESENTATION é SIGNED.

Fonte: ([INTEL, 2017b](#))Tabela 4 – Parâmetros do *IP core LPM_ADD_SUB* utilizados no somador

Nome do parâmetro	Tipo	Valor	Descrição
LPM_WIDTH	Integer	16	Especifica as larguras das portas dataaa[], datab[] e result[].
LPM_DIRECTION	String	ADD	Especifica a função de somador ou subtrator. Os valores são ADD, SUB e UNUSED.
LPM REPRESENTATION	String	SIGNED	Especifica o tipo de adição/subtração executada. Os valores são SIGNED e UNSIGNED. Se omitido, o valor padrão é SIGNED. Quando este parâmetro é definido como SIGNED, o somador/subtrator interpreta a entrada de dados como complemento de dois com sinal.
LPM_PIPELINE	Integer	1	Especifica o número de ciclos de <i>clock</i> de latência associados à saída result[]. Um valor zero (0) indica que não existe latência e que uma função puramente combinacional será instanciada. Se omitido, o valor padrão será 0 (sem pipeline).

Fonte: ([INTEL, 2017b](#)) com valores utilizados pelo autor.

Tabela 5 – Parâmetros do *IP core LPM_ADD_SUB* utilizados no subtrator

Nome do parâmetro	Tipo	Valor	Descrição
LPM_WIDTH	Integer	16	Especifica as larguras das portas dataa[], datab[] e result[].
LPM_DIRECTION	String	SUB	Especifica a função de somador ou subtrator. Os valores são ADD, SUB e UNUSED.
LPM REPRESENTATION	String	SIGNED	Especifica o tipo de adição/subtração executada. Os valores são SIGNED e UNSIGNED. Se omitido, o valor padrão é SIGNED. Quando este parâmetro é definido como SIGNED, o somador/subtrator interpreta a entrada de dados como complemento de dois com sinal.
LPM_PIPELINE	Integer	1	Especifica o número de ciclos de <i>clock</i> de latência associados à saída result[]. Um valor zero (0) indica que não existe latência e que uma função puramente combinacional será instanciada. Se omitido, o valor padrão será 0 (sem pipeline).

Fonte: ([INTEL, 2017b](#)) com valores utilizados pelo autor.

3.4.2 Multiplicador

Uma instância do *IP core LPM_MULT* foi utilizada para executar as operações de multiplicação demandadas ao processador. Este módulo é capaz de multiplicar dois números inteiros recebidos em suas entradas e gerar o produto da operação como saída. A [Figura 44](#) ilustra este bloco e uma descrição da finalidade de suas portas de entrada e de saída pode ser encontrada respectivamente na [Tabela 6](#) e na [Tabela 7](#).

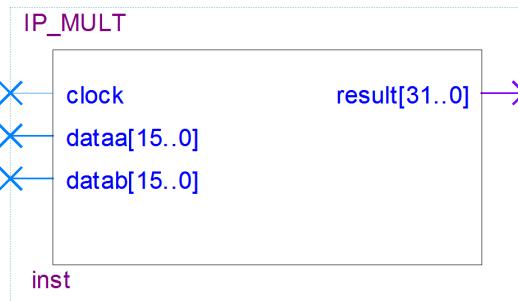
Figura 44 – Representação em bloco do módulo *IP_MULT*.

Tabela 6 – Portas de entrada do módulo *IP_MULT*

Nome da porta	Descrição
dataaa[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTHA.
datab[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTHB.
clock	Entrada para uso em pipeline. Fornece a entrada de sincronismo para uma operação em pipeline. Para valores do parâmetro LPM_PIPELINE diferentes de 0 (padrão), a porta de <i>clock</i> deve estar ativada.

Fonte: ([INTEL, 2017b](#))Tabela 7 – Porta de saída do módulo *IP_MULT*

Nome da porta	Descrição
result[31..0]	Saída de dados. O tamanho da porta de saída depende do valor do parâmetro LPM_WIDTHP. Se LPM_WIDTHP < (LPM_WIDTHA + LPM_WIDTHB), apenas os LPM_WIDTHP bits mais significativos estarão presentes.

Fonte: ([INTEL, 2017b](#))Tabela 8 – Parâmetros do *IP core LPM_MULT* utilizados no multiplicador

Nome do parâmetro	Tipo	Valor	Descrição
LPM_WIDTHA	Integer	16	Especifica a largura da porta dataaa[].
LPM_WIDTHB	Integer	16	Especifica a largura da porta datab[].
LPM_WIDTHP	Integer	32	Especifica a largura da porta result[].
LPM REPRESENTATION	String	SIGNED	Especifica o tipo de multiplicação executada. Os valores são SIGNED e UNSIGNED. Quando este parâmetro é definido como SIGNED, o multiplicador interpreta a entrada de dados como complemento de dois com sinal.
LPM_PIPELINE	Integer	1	Especifica o número de ciclos de <i>clock</i> de latência associados à saída result[]. Um valor zero (0) indica que não existe latência e que uma função puramente combinacional será instanciada. Se omitido, o valor padrão será 0 (sem pipeline).

Fonte: ([INTEL, 2017b](#)) com valores utilizados pelo autor.

A Tabela 8 resume a descrição e o valor dos parâmetros utilizados no momento da instanciação do *IP core*, que foi configurado para receber dados de 16 bits com sinal e produzir dados de 32 bits na saída. Além disso, o recurso de *pipeline* foi utilizado com

latência de 1 ciclo de *clock* a fim de sincronizar a saída do módulo e evitar problemas com a temporização do circuito.

3.4.3 Divisor

O *IP core LPM_DIVIDE* é empregado na arquitetura para efetuar as operações de divisão que precisam ser executadas pelo processador. O divisor implementado por este módulo produz valores inteiros de quociente e resto a partir de números inteiros recebidos nas entradas de numerador e denominador. Uma representação deste bloco pode ser verificada na [Figura 45](#) e a finalidade de suas portas de entrada e de saída é descrita respectivamente na [Tabela 9](#) e na [Tabela 10](#).

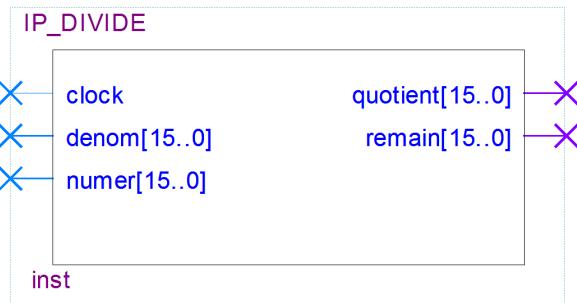


Figura 45 – Representação em bloco do módulo *IP_DIVIDE*.

Tabela 9 – Portas de entrada do módulo *IP_DIVIDE*

Nome da porta	Descrição
numer[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTN.
denom[15..0]	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro LPM_WIDTHD.
clock	Entrada para uso em pipeline. Fornece a entrada de sincronismo para uma operação em pipeline. Para valores do parâmetro LPM_PIPELINE diferentes de 0 (padrão), a porta de <i>clock</i> deve estar ativada.

Fonte: ([INTEL, 2017b](#))

O módulo *IP_DIVIDE* é uma instância do *IP core LPM_DIVIDE* gerada usando os parâmetros e valores descritos na [Tabela 11](#). As entradas de numerador e denominador e as saídas de quociente e resto foram configuradas para tratarem dados de 16 bits, que são interpretados seguindo a representação binária de complemento de dois com sinal.

Para garantir a sincronização das saídas do bloco e minimizar problemas com a temporização do circuito, foi empregado o recurso de *pipeline*. Inicialmente buscou-se que o módulo produzisse resultados com latência de 1 ciclo de *clock*, mas os requisitos de

Tabela 10 – Portas de saída do módulo *IP_DIVIDE*

Nome da porta	Descrição
quotient[15..0]	Saída de dados. O tamanho da porta de saída depende do valor do parâmetro LPM_WIDTHN.
remain[15..0]	Saída de dados. O tamanho da porta de saída depende do valor do parâmetro LPM_WIDTHD.

Fonte: ([INTEL, 2017b](#))

tempo não foram atendidos devido ao uso do *clock* de 50 MHz e ao grande número de elementos lógicos combinacionais necessários para a implementação deste *IP core*. Com a finalidade de reduzir a lógica combinacional entre registradores e atender às exigências de temporização, o bloco foi finalmente configurado para utilizar um *pipeline* com latência de 4 ciclos de *clock*.

Como descrito na [Tabela 11](#), o parâmetro LPM_REMAINDERPOSITIVE pode ser configurado como TRUE para que o valor da porta remain[] seja sempre maior ou igual a zero. A Intel recomenda que esta configuração seja utilizada em operações em que o resto da divisão deve ser positivo ou onde o resto não é importante, de modo a melhorar a velocidade e reduzir a área utilizada pelo circuito ([INTEL, 2017b](#)).

O [Quadro 7](#) e o [Quadro 8](#) demonstram através de tabelas verdade, para alguns valores de numerador e denominador, o comportamento do circuito quando o parâmetro LPM_REMAINDERPOSITIVE é configurado como ‘TRUE’ e como ‘FALSE’.

Quadro 7 – Tabela verdade do *IP core LPM_DIVIDE*:
LPM_REMAINDERPOSITIVE = ‘TRUE’

Entradas		Saídas	
numer[]	denom[]	quotient[]	remain[]
3	3	1	0
3	2	1	1
3	1	3	0
3	0	x	x
3	-1	-3	0
3	-2	-1	1
3	-3	-1	0
-3	3	-1	0
-3	2	-2	1
-3	1	-3	0
-3	0	x	x
-3	-1	3	0
-3	-2	2	1
-3	-3	1	0

Fonte: ([ALTERA, 2007](#))

Quadro 8 – Tabela verdade do *IP core LPM_DIVIDE*:
LPM_REMAINDERPOSITIVE = ‘FALSE’

Entradas		Saídas	
numer[]	denom[]	quotient[]	remain[]
3	3	1	0
3	2	1	1
3	1	3	0
3	0	x	x
3	-1	-3	0
3	-2	-1	1
3	-3	-1	0
-3	3	-1	0
-3	2	-1	-1
-3	1	-3	0
-3	0	x	x
-3	-1	3	0
-3	-2	1	-1
-3	-3	1	0

Fonte: ([ALTERA, 2007](#))

A divisão euclidiana define que dados $a \in \mathbb{Z}$, $b \in \mathbb{Z}^*$ existem $q, r \in \mathbb{Z}$ com $0 \leq r < |b|$ e $a = bq + r$. Tais q e r estão unicamente determinados e são chamados de quociente e resto da divisão de a por b . Se $b > 0$ podemos definir $q = \lfloor a/b \rfloor$ e se $b < 0$, $q = \lceil a/b \rceil$, com $r = a - bq$ em qualquer caso. A notação $\lfloor x \rfloor$ denota o único inteiro k tal que $k \leq x < k + 1$ e $\lceil x \rceil$ o único inteiro k tal que $k - 1 < x \leq k$ ([MOREIRA; SALDANHA, 1999](#)).

Nas divisões de -3 por 2 e de -3 por -2 é possível notar que as saídas de quociente e resto apresentadas no [Quadro 7](#) são distintas das exibidas no [Quadro 8](#), ou seja, dependendo da configuração do parâmetro LPM_REMAINDERPOSITIVE o módulo produz resultados distintos. No entanto, o comportamento do *IP core* é coerente com a definição da divisão euclidiana quando o parâmetro LPM_REMAINDERPOSITIVE é definido como ‘TRUE’, e, por conta disto, esta foi a configuração utilizada no projeto.

Com o módulo definido desta maneira foi possível produzir resultados com latência de 4 ciclos de *clock* consumindo 445 células lógicas do FPGA. Em testes com o parâmetro LPM_REMAINDERPOSITIVE definido como ‘FALSE’ foram necessárias 439 células lógicas e uma latência mínima de 4 ciclos de *clock* para a geração das saídas. Assim, especificamente para o circuito sintetizado neste projeto não foi possível observar o incremento da velocidade e a redução da área utilizada quando LPM_REMAINDERPOSITIVE é definido como ‘TRUE’.

Tabela 11 – Parâmetros do *IP core LPM_DIVIDE* utilizados no divisor

Nome do parâmetro	Tipo	Valor	Descrição
LPM_WIDTHN	Integer	16	Especifica a largura das portas numer[] e quotient[]. Podem ser usados valores de 1 a 64.
LPM_WIDTHD	Integer	16	Especifica a largura das portas denom[] e remain[]. Podem ser usados valores de 1 a 64.
LPM_NREPRESENTATION	String	SIGNED	Representação de sinal da entrada de numerador. Os valores são SIGNED e UNSIGNED. Quando este parâmetro é definido como SIGNED, o divisor interpreta a entrada de numer[] como complemento de dois com sinal.
LPM_DREPRESENTATION	String	SIGNED	Representação de sinal da entrada de denominador. Os valores são SIGNED e UNSIGNED. Quando este parâmetro é definido como SIGNED, o divisor interpreta a entrada de denom[] como complemento de dois com sinal.
LPM_PIPELINE	Integer	4	Especifica o número de ciclos de <i>clock</i> de latência associados às saídas quotient[] e remain[]. Um valor zero (0) indica que não existe latência e que uma função puramente combinacional será instanciada. Se omitido, o valor padrão será 0 (sem pipeline). Não é possível especificar um valor para o parâmetro LPM_PIPELINE maior que LPM_WIDTHN.
LPM_REMAINDERPOSITIVE	String	TRUE	Parâmetro específico da <i>Intel</i> . O parâmetro LPM_HINT deve ser usado para especificar o parâmetro LPM_REMAINDERPOSITIVE em arquivos de design VHDL. Os valores são TRUE ou FALSE. Se este parâmetro estiver definido como TRUE, então o valor da porta remain[] deverá ser maior ou igual a zero. Se este parâmetro estiver configurado como FALSE, então o valor da porta remain[] é zero ou tem o mesmo sinal, positivo ou negativo, que o valor da porta numer[]. Para reduzir a área e melhorar a velocidade, a <i>Intel</i> recomenda configurar este parâmetro como TRUE em operações em que o resto deve ser positivo ou onde o resto não é importante.

Fonte: ([INTEL, 2017b](#)) com valores utilizados pelo autor.

3.4.4 Comparador

O *IP core LPM_COMPARE* compara dois dados de entrada e sinaliza a relação entre os valores através das portas de saída. Seis tipos de comparação podem ser configuradas, permitindo indicar que a entrada A é menor que a entrada B (*alb*), a entrada A é igual a entrada B (*aeb*), a entrada A é maior que a entrada B (*agb*), a entrada A é maior ou igual à entrada B (*ageb*), a entrada A não é igual a entrada B (*aneb*) e, por fim, a entrada A é menor ou igual à entrada B (*aleb*).

A [Figura 46](#) ilustra as portas do módulo *IP_COMPARE*, com a [Tabela 12](#) e a [Tabela 13](#) apresentando respectivamente as descrições das entradas e saídas do bloco.

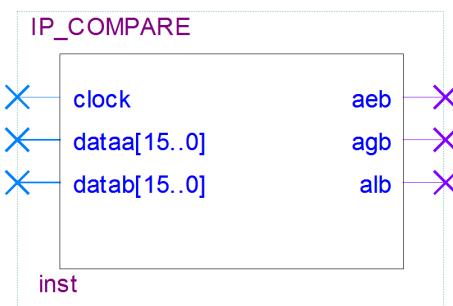


Figura 46 – Representação em bloco do módulo *IP_COMPARE*.

Tabela 12 – Portas de entrada do módulo *IP_COMPARE*

Nome da porta	Descrição
<i>dataaa[15..0]</i>	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro <i>LPM_WIDTH</i> .
<i>datab[15..0]</i>	Entrada de dados. O tamanho da porta de entrada depende do valor do parâmetro <i>LPM_WIDTH</i> .
<i>clock</i>	Entrada para uso em pipeline. Fornece a entrada de sincronismo para uma operação em pipeline. Para valores do parâmetro <i>LPM_PIPELINE</i> diferentes de 0 (padrão), a porta de <i>clock</i> deve estar ativada.

Fonte: ([INTEL, 2017b](#))

O *IP_COMPARE* foi criado como uma instância do *IP core LPM_COMPARE* e concebido para executar as comparações solicitadas ao processador da plataforma. As portas de entrada suportam dados de 16 bits representados como complemento de dois com sinal, e apenas as saídas *alb*, *aeb* e *agb* foram habilitadas. O mecanismo de *pipeline* foi aplicado com latência de 1 ciclo de *clock* para sincronizar devidamente as saídas do bloco com o restante do circuito, reduzindo problemas de temporização. A [Tabela 14](#) resume os parâmetros e valores utilizados na configuração do módulo para obter o comportamento descrito.

Tabela 13 – Portas de saída do módulo *IP_COMPARE*

Nome da porta	Descrição
aeb	Porta de saída do comparador. Ativado se a entrada A é igual a entrada B.
agb	Porta de saída do comparador. Ativado se a entrada A é maior que a entrada B.
alb	Porta de saída do comparador. Ativado se a entrada A é menor que a entrada B.

Fonte: ([INTEL, 2017b](#))

Tabela 14 – Parâmetros do *IP core LPM_COMPARE* utilizados no comparador

Nome do parâmetro	Tipo	Valor	Descrição
LPM_WIDTH	Integer	16	Especifica as larguras das portas dataa[] e datab[].
LPM REPRESENTATION	String	SIGNED	Especifica o tipo de comparação executada. Os valores são SIGNED e UNSIGNED. Se omitido, o valor padrão é UNSIGNED. Quando este parâmetro é definido como SIGNED, o comparador interpreta a entrada de dados como complemento de dois com sinal.
LPM_PIPELINE	Integer	1	Especifica o número de ciclos de <i>clock</i> de latência associados às saídas aeb, agb e alb. Um valor zero (0) indica que não existe latência e que uma função puramente combinacional será instanciada. Se omitido, o valor padrão será 0 (sem pipeline).

Fonte: ([INTEL, 2017b](#)) com valores utilizados pelo autor.

3.4.5 Memória ROM

A memória ROM (*Read-Only Memory*) é uma memória somente de leitura que armazena dados permanentemente, ou seja, é possível ler uma memória ROM mas não há a possibilidade de escrever algo novo nela. O FPGA utilizado neste projeto possui 3.888 kbits de memória embarcada divididos em blocos que podem ser configurados para se comportarem como uma memória ROM.

O *IP core ALTSYNCRAM* foi utilizado para instanciar o módulo *IP_ROM_Program*, uma memória ROM de 2.048 kbits responsável por armazenar o algoritmo executado pelo processador. As instruções do programa são inicializadas na memória através de um arquivo *MIF* (*Memory Initialization File*) indicado no momento da criação do bloco.

A [Figura 47](#) apresenta o bloco *IP_ROM_Program*. Esta memória é dividida em 65.536 endereços com palavras de 32 bits, requerendo que a porta de entrada de endereço

$address[15..0]$ possua 16 bits de largura e que a porta de saída de dados $q[31..0]$ apresente largura de 32 bits. Foram adicionados registradores a esta porta de saída durante a configuração do bloco, permitindo a previsibilidade do momento em que os dados estão disponíveis para leitura e evitando dificuldades na temporização do circuito. Os parâmetros utilizados para a elaboração do módulo estão descritos na Tabela 15.

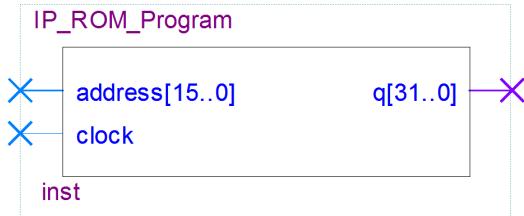


Figura 47 – Representação em bloco do módulo *IP_ROM_Program*.

Tabela 15 – Parâmetros do *IP core ALTSYNCRAM* utilizados na memória ROM

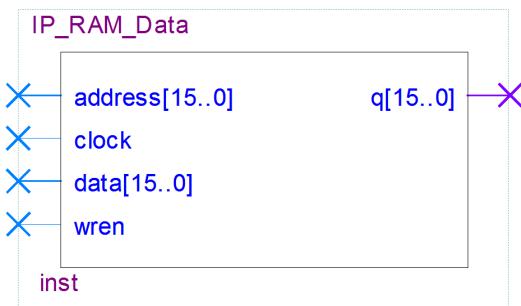
Nome do parâmetro	Tipo	Valor	Descrição
NUMWORDS_A	Integer	65536	Número de palavras de dados no bloco de memória da porta A.
OPERATION_MODE	String	ROM	Modo de operação do bloco de memória.
OUTDATA_REG_A	String	CLOCK0	<i>Clock</i> para os registradores de saída de dados da porta A.
WIDTHAD_A	Integer	16	Largura do endereço da porta A.
WIDTH_A	Integer	32	Largura de dados da porta A.

Fonte: (INTEL, 2017a) com valores utilizados pelo autor.

3.4.6 Memória RAM

A memória RAM (*Random-Access Memory*) é um elemento de armazenamento que permite a leitura e a escrita de dados a qualquer instante e em qualquer endereço da memória. Em um FPGA, as células lógicas e os blocos de memória embarcados podem ser programados para funcionarem como um módulo de memória RAM.

O *IP core ALTSYNCRAM* foi parametrizado como uma memória RAM de 1.024 kbits e utilizado para instanciar o módulo *IP_RAM_Data*, ilustrado na Figura 48. Esta memória é compartilhada por vários módulos e empregada para armazenar os *shapes* dos *sprites*, os registradores de comunicação de entrada e saída, a máscara de interrupção e dados em geral. A inicialização de dados nesta memória pode ser realizada através de um arquivo *MIF* (*Memory Initialization File*) apontado durante a criação do bloco, e este recurso foi explorado para salvar as formas dos *sprites* que serão exibidos.

Figura 48 – Representação em bloco do módulo *IP_RAM_Data*.

Para que todos os 65.536 endereços da memória sejam acessíveis, é exigido que a porta de entrada de endereço *address[15..0]* tenha 16 bits de largura. A porta de entrada de dados *data[15..0]* e a porta de saída de dados *q[15..0]* apresente largura de 16 bits como consequência do armazenamento de palavras de 16 bits. O pino *wren* é essencial para sinalizar ao módulo se a operação requisitada é de leitura, quando em nível baixo, ou de escrita, quando em nível alto. Por fim, durante a configuração do bloco foram adicionados registradores à porta de saída *q[15..0]*, de modo que haja previsibilidade do momento em que os dados estão disponíveis para leitura e evitando dificuldades na temporização do circuito. A Tabela 16 descreve os parâmetros utilizados na elaboração do módulo.

Tabela 16 – Parâmetros do *IP core ALTSYNCRAM* utilizados na memória RAM

Nome do parâmetro	Tipo	Valor	Descrição
NUMWORDS_A	Integer	65536	Número de palavras de dados no bloco de memória da porta A.
OPERATION_MODE	String	SINGLE_PORT	Modo de operação do bloco de memória.
OUTDATA_REG_A	String	CLOCK0	<i>Clock</i> para os registradores de saída de dados da porta A.
WIDTHAD_A	Integer	16	Largura do endereço da porta A.
WIDTH_A	Integer	16	Largura de dados da porta A.

Fonte: ([INTEL, 2017a](#)) com valores utilizados pelo autor.

3.4.7 PLL

O PLL (*Phase-Locked Loop*) é um circuito de controle por realimentação que ajusta automaticamente a fase de um sinal gerado localmente para sincronizá-lo com a fase de um sinal de entrada. Os PLLs operam produzindo uma frequência de oscilador síncrona com a frequência de um sinal de entrada. Nesta condição encadeada, qualquer pequena mudança no sinal de entrada aparece como uma diferença de fase entre o sinal de entrada e o gerado pelo oscilador. Este deslocamento de fase atua como um sinal de erro que altera

a frequência do oscilador local do PLL para sincronizá-lo ao sinal de entrada ([ALTERA, 2017](#)).

A placa de desenvolvimento *Altera DE2-115*, fabricada pela *Terasic* e utilizada como base deste projeto, inclui um oscilador que produz sinal de *clock* de 50 MHz. Este sinal é distribuído e conectado aos pinos de entrada de *clock* do FPGA, permitindo que seja usado como fonte de sincronismo para os circuitos PLL embarcados.

Apesar da maior parte do circuito projetado operar com *clock* de 50 MHz, o módulo de interface VGA desenvolvido requer um *clock* de 25 MHz para produzir o sinal de vídeo com a resolução e frequência desejadas. Este novo sinal de *clock* é gerado por um circuito PLL implementado pelo módulo *IP_PLL*, garantindo a sincronia com o *clock* de 50 MHz fornecido aos demais blocos.

O módulo *IP_PLL*, ilustrado na [Figura 49](#), é uma instância do *IP core ALTPPLL*. Durante a configuração do bloco, apenas as portas *inclk0* e *c0* foram habilitadas. A primeira recebe o *clock* de entrada, que referencia o sinal produzido, e a segunda é a saída de *clock* produzida pelo PLL. A [Tabela 17](#) descreve os parâmetros utilizados para a elaboração do módulo.

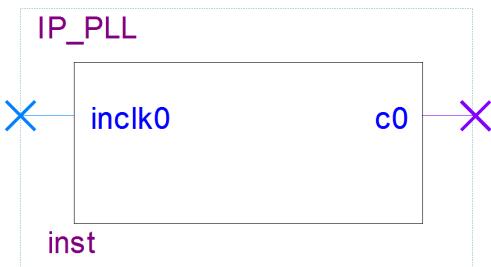


Figura 49 – Representação em bloco do módulo *IP_PLL*.

3.5 Softwares Desenvolvidos

O desenvolvimento de jogos para a plataforma é apoiado em dois *softwares* que foram produzidos. O primeiro é um conversor de imagem bitmap em arquivo *.hex*, utilizado para carregar a imagem de *background* do jogo no protótipo. O segundo é um *Assembler*, empregado na conversão do código *assembly* do jogo criado para a plataforma, em um arquivo *.mif* com as instruções em código binário. Este arquivo é usado para inicializar a memória ROM do projeto.

3.5.1 Conversor de Imagens Bitmap

A imagem de fundo do jogo, o *background*, deve ser carregada na memória SRAM do kit de desenvolvimento DE2-115 através do *software Terasic DE2-115 Control Panel*,

Tabela 17 – Parâmetros do *IP core ALTPLL* utilizados no PLL

Nome do parâmetro	Tipo	Valor	Descrição
PLL_TYPE	String	AUTO	Especifica o tipo de PLL a ser instanciado. Os valores são AUTO, ENHANCED, FAST, TOP_BOTTOM e LEFT_RIGHT. Se omitido, o padrão é AUTO.
OPERATION_MODE	String	NORMAL	Especifica a operação do PLL. Os valores são EXTERNAL_FEEDBACK, NO_COMPENSATION, NORMAL, ZERO_DELAY_BUFFER e SOURCE_SYNCHRONOUS. Se omitido, o padrão é NORMAL.
COMPENSATE_CLOCK	String	CLK0	Especifica a porta de saída de <i>clock</i> que deve ser compensada.
BANDWIDTH_TYPE	String	AUTO	Especifica o tipo de largura de banda para o parâmetro BANDWIDTH. Os valores são AUTO, LOW, MEDIUM ou HIGH. Se omitido, o valor padrão é AUTO.
CLK0_MULTIPLY_BY	Integer	1	Especifica o fator de multiplicação para a frequência do VCO da porta de saída de <i>clock</i> correspondente. O valor do parâmetro deve ser maior que 0. Se omitido, o padrão é 0.
CLK0_DIVIDE_BY	Integer	2	Especifica o fator de divisão para a frequência do VCO da porta de saída de <i>clock</i> correspondente. O valor do parâmetro deve ser maior que 0. Se omitido, o padrão é 0.
CLK0_PHASE_SHIFT	String	0	Especifica o deslocamento de fase em picossegundos (ps) para a porta de saída de <i>clock</i> correspondente. Se omitido, o padrão é 0.
CLK0_DUTY_CYCLE	Integer	50	Especifica o ciclo de trabalho em porcentagem para a porta de saída de <i>clock</i> correspondente. Se omitido, o padrão é 50.

Fonte: ([ALTERA, 2017](#)) com valores utilizados pelo autor.

disponibilizado pela fabricante da placa. Esta aplicação suporta o carregamento de arquivos com extensão *.hex*, que são arquivos de texto com caracteres ASCII representando valores hexadecimais para serem salvos na memória. Por exemplo, um arquivo contendo a linha 0123456789ABCDEF define oito valores hexadecimais de 8 bits: 01, 23, 45, 67, 89, AB, CD, EF. Estes valores serão carregados consecutivamente na memória ([TERASIC, 2017](#)).

O algoritmo para converter uma imagem em um arquivo *.hex* foi desenvolvido na forma de um *script* do *software Matlab*, devido à praticidade e à grande quantidade de funções embutidas nesta ferramenta. O *script* faz a leitura de uma imagem, no formato *bitmap* com resolução de 640 x 480 e 24 bits de profundidade de cor, e transforma as informações dos *pixels* em 3 matrizes de 480 linhas por 640 colunas. Cada matriz armazena valores de 8 bits de uma das componentes de cor, totalizando os 24 bits da profundidade de cor do *pixel*. Entretanto, a memória SRAM tem apenas 16 bits de largura e, para

armazenar um *pixel* em cada posição da memória como desejado, foi necessário eliminar alguns bits menos significativos da representação de cor.

O padrão seguido foi o mesmo adotado em outros elementos da plataforma, mantendo 5 bits para as cores vermelha e azul, e 6 bits para a cor verde. Deste modo, o *script* converte os valores das matrizes e os agrupa no formato adequado em vetores de 16 bits com a representação de um *pixel*. Os valores destes vetores são então escritos no formato hexadecimal e de modo sequencial em um arquivo de texto com extensão *.hex*.

3.5.2 Assembler

A aplicação do *Assembler* foi implementada na linguagem *Python*, por conta de sua simplicidade e variedade de funções que agilizam a codificação. Através da interface gráfica do programa, nomeado como *asm2mif Assembler*, o usuário tem acesso a uma janela de navegação para escolher o arquivo de texto *.asm* que contém o código *assembly* do jogo. Em seguida, também por meio de uma janela de navegação, o usuário deve escolher o nome e o local onde será salvo o arquivo de texto *.mif*, contendo o código traduzido para o formato binário. Após o fim do processo de conversão e criação do novo arquivo, a aplicação exibe uma mensagem indicando o sucesso da operação. Caso ocorra algum problema na conversão, uma mensagem de erro é apresentada.

Por trás da interface gráfica, o algoritmo faz a leitura do arquivo de texto *.asm* e realiza diversos tratamentos, como o reconhecimento e conversão dos *labels* em endereços de memória, a eliminação de comentários do código, remoção de espaços vazios e, por fim, a conversão das instruções *assembly* em códigos binários, que devem seguir o formato definido nos modelos de instruções da plataforma. Esta última etapa exige uma conversão adequada de números decimais para binários, além do reconhecimento de símbolos como vírgulas, espaços e parênteses, que separam os elementos de uma instrução.

4 Validação do Protótipo

Antes de ser implementado na placa DE2-115, o projeto desenvolvido no *software Quartus Prime Lite* foi testado funcionalmente com o apoio da ferramenta *ModelSim - INTEL FPGA STARTER EDITION*, produzida pela *Mentor Graphics*. O teste funcional se baseou na elaboração de um *test bench* em linguagem *Verilog HDL*, provendo estímulos às entradas e salvado em arquivos de texto algumas das saídas do *DUT* (*Design Under Test*), nome dado ao projeto que está sendo testado.

Foram simulados pelo *test bench* um *clock* de 25 MHz, usado no módulo *VGA_Interface*, e um *clock* de 50 MHz, utilizado pelo restante dos módulos do projeto. O comportamento do controle de *Sega Mega Drive* foi simulado, e os sinais produzidos pelo acionamento de alguns botões foram aplicados à entrada do *DUT*. Além disso, um código *assembly* específico foi desenvolvido para ser executado pelo processador durante o teste.

O algoritmo executado pelo *DUT* deveria conter todas as instruções suportadas e utilizar a maior variedade de operandos possíveis, de modo a verificar o funcionamento do processador de modo confiável. Como um grande número de instruções seriam necessárias, um código em *Python* foi criado para automatizar a geração do código *assembly*, com as instruções e combinações de operandos desejadas. O código *assembly* produzido também faz uma série de verificações das operações aritméticas, comparando os resultados produzidos pelo processador com os calculados pelo código em *Python*. Em caso de erro, a execução do processador é desviada para um endereço específico, onde fica em *loop* para sinalizar o problema.

Com o algoritmo *assembly* adequado e com os estímulos de entrada do *DUT* sendo simulados, o passo seguinte foi verificar o comportamento das saídas. Para este propósito, dois arquivos de texto são produzidos pelo *test bench*. O primeiro contém os valores de cada *pixel* produzido na saída do *DUT*. Este arquivo de texto é formatado de modo que os dados assumam um formato semelhante ao de uma imagem, sendo possível visualizar a posição dos *sprites* exibidos. O segundo arquivo armazena de forma sequencial todas as instruções executadas pelo processador do *DUT*, permitindo seguir o caminho da execução caso alguma anomalia seja detectada. Além dos arquivos produzidos, foram adicionados ao *test bench* algumas mensagens para sinalizar erros e a ocorrência de eventos, como interrupções.

Após a verificação funcional do projeto por meio do *test bench*, o circuito sintetizado pelo *software Quartus Prime Lite* foi finalmente carregado no FPGA da placa de desenvolvimento. No teste do protótipo foram conectados ao kit o controle de *Sega Mega Drive* e um monitor LCD com entrada VGA. Uma imagem de *background* personalizada

foi convertida e carregada na memória SRAM da DE2-115, e o mesmo código *assembly* da verificação funcional foi executado. O protótipo se comportou como programado, com alguns *sprites* configurados se movimentando conforme o acionamento dos botões do controle. A [Figura 50](#) apresenta uma foto do protótipo montado.

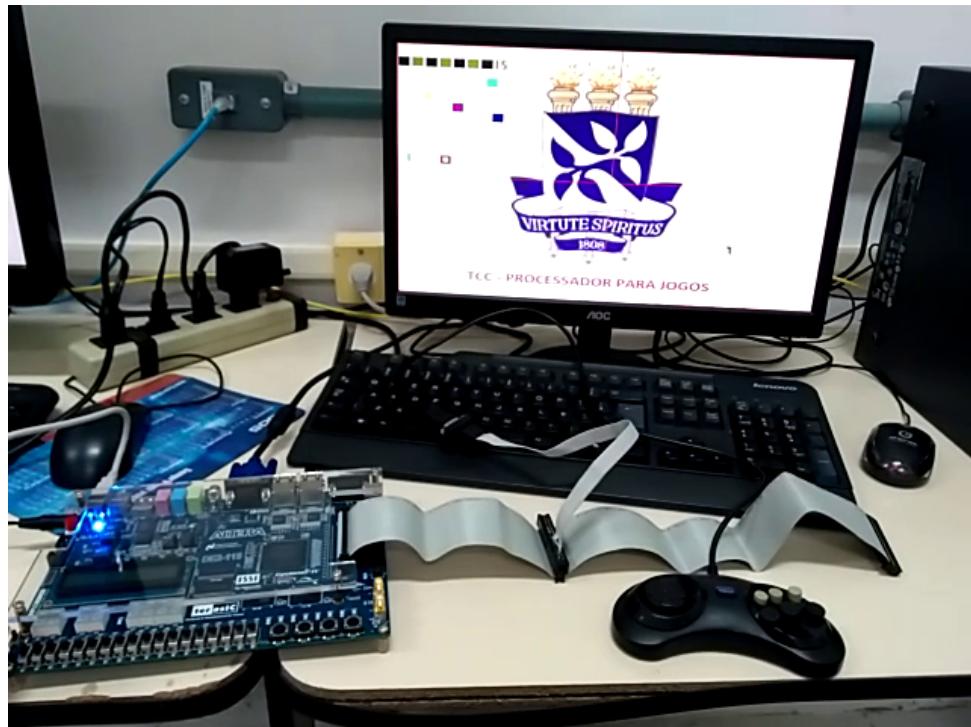


Figura 50 – Foto do protótipo montado.

5 Conclusão

Neste trabalho foram apresentadas as características, a implementação e a validação do projeto de uma plataforma para jogos com gráficos em duas dimensões. O protótipo elaborado com a utilização da placa de desenvolvimento *Altera DE2-115* funcionou de forma adequada em todos testes, comportando-se de acordo ao código programado para execução pelo processador da plataforma.

Apesar do protótipo, na forma final, ser capaz de exibir apenas 16 *sprites* simultaneamente, a arquitetura planejada previa a apresentação de até 64 *sprites* ao mesmo tempo. Entretanto, não foi possível sintetizar no FPGA o circuito projetado para exibir esta quantidade de elementos gráficos e, por conta disso, o número foi reduzido. O desenvolvimento de uma nova arquitetura para a etapa gráfica, que possibilite o uso de mais *sprites*, é uma possível melhoria futura.

Referências

- ALTERA CORPORATION. *lpm_divide Megafunction*: User guide. [S.l.], 2007. 32 p. Disponível em: <https://www.altera.com/zh_CN/pdfs/literature/ug/ug_lpm_divide_mf.pdf>. Acesso em: Mai/2018. Citado 2 vezes nas páginas 73 e 74.
- ALTERA CORPORATION. *ALTPPLL (Phase-Locked Loop) IP Core User Guide*. [S.l.], 2017. 63 p. Disponível em: <https://www.altera.com/en_US/pdfs/literature/ug/ug_altpll.pdf>. Acesso em: Mai/2018. Citado 2 vezes nas páginas 80 e 81.
- AMARAL, L. *Magnavox Odyssey: o primeiro videogame do mundo*. 2018. Disponível em: <<http://www.gameblast.com.br/2018/03/magnavox-odyssey-primeiro-videogame.html>>. Acesso em: Jun/2018. Citado na página 17.
- ANALOG DEVICES INC. *ADV7123: CMOS, 330 MHz Triple 10-Bit High Speed Video DAC*. [S.l.], 2010. 24 p. Disponível em: <<http://www.analog.com/media/en/technical-documentation/data-sheets/ADV7123.pdf>>. Acesso em: Jun/2018. Citado 2 vezes nas páginas 51 e 52.
- BATISTA, M. de L. S. et al. Um estudo sobre a história dos jogos eletrônicos. *Revista Eletrônica da Faculdade Metodista Granbery* — Faculdade de Sistemas de Informação, Juiz de Fora, Minas Gerais, n. 3, jul/dez 2007. ISSN 1981-0377. Disponível em: <<http://re.granbery.edu.br/artigos/MjQ4.pdf>>. Acesso em: Jun/2018. Citado 2 vezes nas páginas 16 e 17.
- CHU, P. P. *FPGA prototyping by Verilog examples*: Xilinx spartan-3 version. Hoboken, New Jersey: John Wiley & Sons, Inc., 2008. ISBN 9780470185322. Citado 5 vezes nas páginas 20, 21, 49, 50 e 51.
- CUMMINGS, C. E.; MILLS, D.; GOLSON, S. Asynchronous & synchronous reset design techniques - part deux. In: *SNUG Boston 2013*. [s.n.], 2003. Disponível em: <http://www.sunburst-design.com/papers/CummingsSNUG2003Boston_Resets.pdf>. Acesso em: Mai/2018. Citado na página 31.
- HARADA, J. *Que indústria fatura mais: do cinema, da música ou dos games?* 2018. Disponível em: <<https://super.abril.com.br/cultura/que-industria-fatura-mais-do-cinema-da-musica-ou-dos-games/>>. Acesso em: Jun/2018. Citado na página 17.
- INTEGRATED SILICON SOLUTION INC. *1M X 16 HIGH-SPEED ASYNCHRONOUS CMOS STATIC RAM WITH 3.3V SUPPLY (IS61WV102416ALL / IS61WV102416BLL / IS64WV102416BLL)*. [S.l.], 2014. 20 p. Disponível em: <<http://www.issi.com/WW/pdf/61WV102416ALL.pdf>>. Acesso em: Jun/2018. Citado na página 47.
- INTEL CORPORATION. *Embedded Memory (RAM: 1-PORT, RAM: 2-PORT, ROM: 1-PORT, and ROM: 2-PORT) User Guide*. [S.l.], 2017. 48 p. Disponível em: <https://www.altera.com/en_US/pdfs/literature/ug/ug_ram_rom.pdf>. Acesso em: Mai/2018. Citado 2 vezes nas páginas 78 e 79.

INTEL CORPORATION. *Intel FPGA Integer Arithmetic IP Cores User Guide*. [S.l.], 2017. 101 p. Disponível em: <https://www.altera.com/en_US/pdfs/literature/ug/ug_lpm_alt_mfug.pdf>. Acesso em: Mai/2018. Citado 8 vezes nas páginas 69, 70, 71, 72, 73, 75, 76 e 77.

INTEL CORPORATION. *Intel Quartus Prime Standard Edition Handbook Volume 1: Design and synthesis*. [S.l.], 2018. 1103 p. Disponível em: <https://www.altera.com/en_US/pdfs/literature/hb/qts/qts-qps-5v1.pdf>. Acesso em: Mai/2018. Citado na página 33.

MOREIRA, C. G. T. de A.; SALDANHA, N. C. Primos de mersenne (e outros primos muito grandes). In: *22º Colóquio Brasileiro de Matemática*. Rio de Janeiro: IMPA, 1999. Disponível em: <<http://www.mat.puc-rio.br/~nicolau/papers/mersenne.pdf>>. Acesso em: Mai/2018. Citado na página 74.

NEWZOO. *2017 GLOBAL GAMES MARKET REPORT*: Trends, insights, and projections toward 2020. [S.l.], 2017. 21 p. Disponível em: <http://progamedev.net/wp-content/uploads/2017/06/Newzoo_Global_Games_Market_Report_2017_Light.pdf>. Acesso em: Jun/2018. Citado 2 vezes nas páginas 17 e 18.

NVIDIA CORPORATION. *NVIDIA TESLA V100 GPU ARCHITECTURE*. [S.l.], 2017. 58 p. Disponível em: <<http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>>. Acesso em: Jun/2018. Citado na página 18.

PALNITKAR, S. *Verilog HDL: A Guide to Digital Design and Synthesis*. 2. ed. [S.l.]: Prentice Hall PTR, 2003. ISBN 0130449113. Citado na página 22.

REIS, G. dos. *Videogame*: história, gêneros e diálogo com o cinema. 190 f. Dissertação (Mestrado em Comunicação) — Faculdade de Comunicação, Educação e Turismo, Universidade de Marília, São Paulo, 2005. Disponível em: <<http://www.unimar.br/pos/trabalhos/arquivos/4fad589f032b377ef35f6d7850966007.pdf>>. Acesso em: Jun/2018. Citado na página 17.

SARNOFF, M. *usb adapter for sega genesis controller*. 2009. Disponível em: <<http://www.msarnoff.org/gen2usb/>>. Acesso em: Jun/2018. Citado na página 42.

SEGA RETRO. *Six Button Control Pad (Mega Drive)*. 2012. Disponível em: <[https://segaretro.org/Six_Button_Control_Pad_\(Mega_Drive\)](https://segaretro.org/Six_Button_Control_Pad_(Mega_Drive))>. Acesso em: Jun/2018. Citado na página 43.

STALLINGS, W. *Arquitetura e organização de computadores*. 8. ed. São Paulo: Pearson Pratice Hall, 2010. ISBN 9788576055648. Citado 2 vezes nas páginas 33 e 36.

TANENBAUM, A. S. *Organização estruturada de computadores*. 5. ed. São Paulo: Pearson Pratice Hall, 2007. ISBN 9788576050674. Citado 2 vezes nas páginas 36 e 59.

TERASIC TECHNOLOGIES INC. *DE2-115 User Manual*. [S.l.], 2017. 121 p. Disponível em: <http://www.terasic.com.tw/cgi-bin/page/archive_download.pl?Language=English&No=502&FID=cd9c7c1feaa2467c58c9aa4cc02131af>. Acesso em: Mai/2018. Citado 5 vezes nas páginas 31, 45, 47, 52 e 81.

WIKIPEDIA. *Tennis for Two*. 2018. Disponível em: <https://pt.wikipedia.org/wiki/Tennis_for_Two>. Acesso em: Jun/2018. Citado na página 16.