

Desenvolvimento de uma Arquitetura Baseada em Sprites para criação de Jogos 2D em Ambientes Reconfiguráveis utilizando dispositivos FPGA

Gabriel B. Alves, Anfranserai M. Dias

Universidade Estadual de Feira de Santana (UEFS)

Departamento de Tecnologia (DTEC)

Feira de Santana, Bahia, Brasil

Email: bielbarretoalves@gmail.com, anfranserai@ecomp.uefs.br

Victor T. Sарinho

Universidade Estadual de Feira de Santana (UEFS)

Lab. de Entretenimento Digital Aplicado (LEnDA)

Feira de Santana, Bahia, Brasil

Email: vsarinho@uefs.br

Resumo—In the last years, it's possible to see an increase in the development of new electronic devices. The architecture of everyone is composed of digital electronic elements and software for controls. The notions of digital elements such as registers, memory, multiplexers, processors, and programming aspects, help engineers and students in the building process of a digital system. To understand these notions, the FPGA combined with the use of games help to assimilate the techniques and approaches used in this process. This paper presents a Sprite Based Architecture that aims the development of 2D games in reconfigurable environments using FPGA devices. Section IV shows all details of the architecture, and present some functions and resources that help to develop a game using the architecture proposed; Section V the experiments and tests.

Index Terms—FPGA, 2D Games, Digital Systems.

I. INTRODUÇÃO

Nos últimos anos ocorreu um aumento considerável no desenvolvimento de novos equipamentos eletrônicos com o intuito de auxiliar a vida cotidiana das pessoas. Esses produtos utilizam em grande parte a eletrônica digital como base de sua modelagem. Para o desenvolvimento desses equipamentos é essencial o domínio dos elementos relacionados à criação de sistemas e circuitos digitais. Considerando o mercado mundial de dispositivos eletrônicos, um fator importante que ocasionou seu crescimento exponencial foi o avanço das técnicas de fabricação de circuitos integrados. Essas técnicas garantem a miniaturização dos componentes, permitindo a combinação de vários sistemas em um único chip construído em escala nanométrica. Diante desse fator, houve também um aumento na complexidade do processo de modelagem e verificação, tornando os métodos de projeto e verificação auxiliados por computador essenciais para o desenvolvimento desses projetos.

Diante disso, em cursos como Engenharia de Computação, Elétrica e Eletrônica, as universidades utilizam dispositivos FPGA (*Field-Programmable Gate Array*) como uma ferramenta de aprendizagem em prol do desenvolvimento acadêmico dos discentes sobre os elementos de software, hardware e técnicas de integração. Os FPGA permitem realizar a simulação dos projetos de circuitos e sistemas digitais além de possibilitar a verificação do comportamento do sistema

em um dispositivo físico. Também são instrumentos capazes de fornecer os meios necessários para promover atividades lúdicas e interativas que facilitam a assimilação das técnicas de modelagem e integração de forma mais prática e objetiva.

Um formato de atividade consiste na criação e no uso de jogos como uma ferramenta de auxílio à aprendizagem. Os jogos estão dentro da categoria de atividades que promovem grande flexibilidade na abordagem e compreensão de um conteúdo. Além disso, contribui com o desenvolvimento de estratégias e habilidades importantes, como raciocínio dedutivo e memorização [1]. Com eles é possível aplicar conceitos de programação e implementação de sistemas digitais de uma forma mais descontraída, menos densa e motivadora para os alunos, além de fortalecer o desenvolvimento lógico, cognitivo, técnico, a criatividade, entre outras aptidões [2].

A metodologia de uso das FPGA proporciona algumas vantagens como a utilização de um único sistema CAD (*Computer-Aided Design*) para o desenvolvimento, validação dos projetos e programação dos chips FPGA. Além disso, encontra-se no mercado diversas placas de desenvolvimento acopladas com esses chips, e estruturadas com periféricos como displays, botões, e outras interfaces de entrada e saída. Esses periféricos auxiliam na interação com o sistema desde a entrada de dados à visualização gráfica dos resultados obtidos. Essa variedade possibilita adquirir placas de acordo com o orçamento disponível e recursos necessários para o desenvolvimento dos projetos em laboratório.

Com a contribuição dos jogos e as vantagens apresentadas pelo uso das FPGA, desenvolver jogos em ambientes reconfiguráveis torna-se uma excelente forma prática de consolidar os conceitos relacionados a modelagem de circuitos e sistemas digitais, além da integração entre hardware e software.

Neste contexto, esse trabalho tem por objetivo apresentar a modelagem de uma arquitetura baseada em *sprites* que visa desenvolver jogos bidimensionais em ambientes reconfiguráveis utilizando dispositivos FPGA permitindo a compreensão dos elementos de hardware digital, assim como aspectos de programação. Para fins de avaliação, foram criados dois jogos baseados nos clássicos *Asteroids* e *Space Invaders*, pois

a mecânica simples viabilizou a implementação a partir dos recursos disponíveis no projeto.

II. TRABALHOS RELACIONADOS

Quando mencionamos a respeito da utilização dos jogos eletrônicos como ferramenta de aprendizagem, têm-se em vista os benefícios que podem ser alcançados de acordo com a metodologia utilizada. Como por exemplo, a criação de experiências concretas que facilitam o entendimento de algo abstrato, e o aprimoramento de habilidades cognitivas e trabalho em equipe [2]. Essas experiências com os jogos podem influenciar os aspectos cognitivos resultando em um processo de aprendizagem mais eficiente. Isso possibilita a aplicação de métodos de aprendizagem com o uso de jogos de maneira intencional em um contexto de treinamento de habilidades e/ou sala de aula [3].

Devido a essas observações, muitos projetos de âmbito acadêmico são realizados em universidades utilizando a metodologia de criação de jogos eletrônicos em dispositivos FPGA. Jogos clássicos como, *Snake* [4], *Galaxian* [5], *Space Shoot* [6] e *Pong* [7] são alguns exemplos. Tais projetos impulsionam nos alunos o desenvolvimento das habilidades de modelagem, verificação e testes de circuitos e sistemas digitais. Grande parte dos projetos são modelados através das linguagens HDL (*Hardware Description Language*) como Verilog ou VHDL (*VHSIC Hardware Description Language*) [8] e sintetizados através de um sistema CAD, como o *Intel Quartus Prime*. No entanto, essa abordagem é limitada em relação à dificuldade de criação de novos jogos, necessitando o desenvolvimento de um novo hardware de controle e animação em cada projeto.

Outra abordagem consiste no uso de microcontroladores. O desenvolvimento é realizado por meio de uma placa de circuito impresso acoplada com periféricos de I/Os (*Inputs and Outputs*) e um microcontrolador como os da linha Arduino. APIs (*Application Programming Interface*) também são disponibilizadas para facilitar a criação, controle dos jogos e acesso ao hardware. Os dispositivos *GameBuino* [9] e o *Arduino Esplora* [10] são alguns exemplos. Contudo, essa abordagem dificulta o uso de *sprites* e a realização de melhorias no hardware para criação de jogos mais complexos.

Em vista desses aspectos, a arquitetura proposta apresenta uma solução para essas limitações através de um ambiente de desenvolvimento que permitirá a criação de jogos por meio da estruturação em hardware, programação em alto-nível ou a combinação de ambas as abordagens. O usuário terá a possibilidade de aprimorar o uso das técnicas de programação e consolidar conceitos de modelagem e integração entre hardware e software.

III. FUNDAMENTAÇÃO TEÓRICA

Alguns conceitos são fundamentais para o desenvolvimento da arquitetura, tais como o entendimento da estruturação de um Dispositivo FPGA, Fluxo de Projeto Digital, e funcionamento do padrão VGA (*Video Graphic Array*) para geração de sinais de vídeo.

A. Estrutura de um Dispositivo FPGA

As FPGA consistem em um arranjo bidimensional de várias células lógicas programáveis e independentes, interconectadas para criação de uma função através da configuração das células e utilização de chaves para conexão entre elas (Figura 1). Cada célula trabalha, normalmente, com até quatro ou cinco variáveis de entrada. A maioria das FPGA utilizam uma tabela de consulta (LUT, do inglês, look-up table) para criar as funções lógicas desejadas. Uma LUT funciona como uma tabela-verdade, no sentido de que a saída é programada para criar a função combinacional ao armazenar o 0 ou 1 adequado a cada combinação de entrada. Os blocos de I/O (*Input/Output*) podem ser configurados para fornecer recursos de entrada, saída ou mesmo bidirecionais; e também registradores internos podem ser usados para armazenar dados que entram ou saem [11]. Essas configurações permitem ao projetista desenvolver qualquer sistema digital, obtendo uma arquitetura flexível e totalmente reconfigurável.

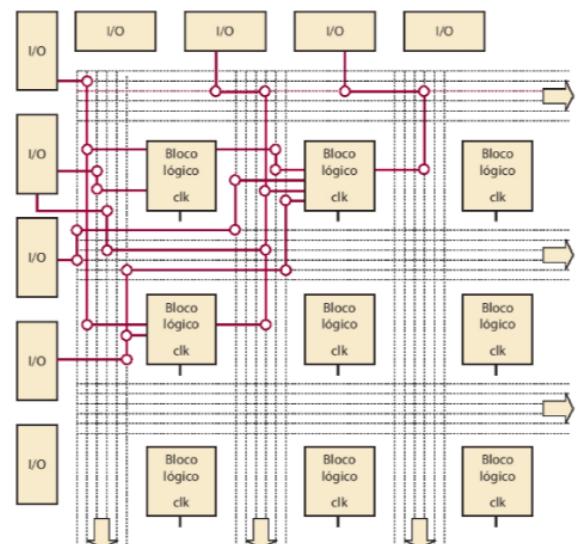


Figura 1: Representação da Estrutura Interna de uma FPGA.

Fonte: Retirado de [11].

B. Fluxo de Projeto Digital

Com a criação das linguagens de descrição de hardware e as ferramentas de síntese, o processo de desenvolvimento de um sistema digital em hardware tornou-se semelhante à criação de um software. Todo o projeto pode ser implementado utilizando uma única linguagem, como Verilog, VHDL ou System Verilog, no entanto, para o uso de tais linguagens não utiliza-se dos conceitos de programação, mas das regras e conceitos voltados para o projeto de circuitos digitais. O fluxo de desenvolvimento do projeto foi definido através das seguintes etapas etapas, especificação, design em alto nível, codificação, verificação/testes e sintetização:

Especificação: A etapa de especificação consiste na primeira tarefa desenvolvida pelo projetista ou engenheiro. Nesse

momento será definido todos os requisitos funcionais e não funcionais do sistema digital, além de todas as ferramentas e tecnologias que serão utilizadas durante o projeto. Os requisitos funcionais abrangem todas as ações que o sistema ou componente individual irá exercer. Os não funcionais não estão diretamente ligados às funcionalidades de um sistema, mas, a requisitos como desempenho, portabilidade, segurança, dentre outros. O engenheiro/projetista deverá especificar tais requisitos para cada módulo que compõe o sistema.

Design em Alto Nível: É construído uma representação em alto nível da arquitetura do sistema digital, demonstrando como todos os módulos estão interligados e como se comunicam.

Codificação: Para os sistemas digitais projetados em FPGA a etapa de codificação acontecerá através de alguma HDL. Toda a arquitetura e funcionalidades definidas na parte inicial do fluxo de projeto são desenvolvidas nesta etapa.

Verificação/Testes: Processo de validação da arquitetura através de alguma ferramenta de simulação com o intuito de garantir antes da etapa de síntese o total funcionamento dos circuitos digitais implementados. O objetivo é verificar se cada módulo desenvolvido atende aos parâmetros dos requisitos funcionais e não funcionais que foram especificados para cada um de forma individual.

Sintetização: Consiste na sintetização de todo o projeto através de uma ferramenta de síntese lógica, como por exemplo, o software *Intel Quartus Prime*. O projeto é carregado em um kit de desenvolvimento que possua um chip FPGA e realiza-se mais um processo de verificação (agora em um ambiente real, não por simulação) com o intuito de validar o funcionamento geral do hardware desenvolvido. Caso algum erro seja identificado, retorna-se à etapa de codificação e segue-se novamente novos testes e sintetização até que todo o sistema esteja funcionando corretamente.

C. Padrão VGA para Geração de Sinais de Vídeo

Esse padrão foi desenvolvido em 1987 pela IBM e tornou-se bastante difundido no mercado, pois esteve presente na maioria dos monitores de tubo CRT (*Cathode Ray Tube*), bem como foram implementados nos primeiros monitores LCD (*Liquid Crystal Display*). Neste padrão são transmitidos os sinais de geração de vídeo, tais como sinais de sincronização vertical e horizontal, e sinais de cores RGB (Red – Vermelho, Green – Verde, Blue – Azul). Para o controle e geração desses sinais um Controlador de Vídeo é necessário. O sinal de sincronismo horizontal (*hsync*) especifica o tempo requerido para percorrer uma linha do monitor. Ele é ativo em nível lógico 0 e indica que uma linha foi finalizada e que a próxima linha será iniciada. O sinal de sincronismo vertical (*vsync*) controla o tempo necessário para percorrer toda a tela. Ele é ativo em nível lógico 0 e indica que um *Frame* foi finalizado e que o próximo será iniciado.

Na Figura 2 é possível observar a estrutura de um *Frame* utilizando o padrão VGA. A partir dessa observação temos a definição dos seguintes parâmetros:

- 1) **Área Ativa:** Corresponde ao espaço de exibição dos *pixels* na tela.
- 2) **Front Porch e Back Porch:** Corresponde aos tempos de espera que devem ser implementados para delimitar a região Ativa. Esses tempos dependem da resolução e frequência de atualização da tela.



Figura 2: Representação de um *Frame* de Vídeo utilizando o padrão VGA.

Fonte: Próprio autor.

Para o controle do sincronismo é necessário obter o tempo correspondente a cada sinal de vídeo, e assim usá-los para expressar matematicamente os valores de contagem da varredura horizontal e vertical. Tomando como base a resolução de 640x480 *pixels* com taxa de atualização de 60Hz, temos por padrão os valores da Tabela I.

Tabela I: Definição dos parâmetros de resolução da tela

Elemento	Sigla	Pixels
Tamanho Horizontal	HD	640
Front Porch Horizontal	HF	16
Back Porch Horizontal	HB	48
Pulso <i>hsync</i>	HR	96
Total de Pixels Horizontais	HT	800
Tamanho Vertical	VD	480
Front Porch Vertical	VF	11
Back Porch Vertical	VB	31
Pulso <i>vsync</i>	VR	2
Total de Pixels Verticais	VT	524

1) *Valor Limite para o Contador da Varredura Horizontal:* Defini-se o valor máximo de contagem do elemento HT (Total de *Pixels* Horizontais) como:

$$HT = HD + HF + HB + HR \quad (1)$$

2) *Valor Limite para o Contador da Varredura Vertical:* Defini-se o valor máximo de contagem do elemento VT (Total de *Pixels* Verticais) como:

$$VT = VD + VF + VB + VR \quad (2)$$

3) *Período de Acionamento do Sinal de Sincronismo Horizontal*: O sinal de sincronismo horizontal (*hsync*) deve ser acionado quando o contador horizontal atender à seguinte condição:

$$H_{counter} = HD + HF \quad (3)$$

4) *Fim da Permanência do Sinal de Sincronismo Horizontal*: O sinal *hsync* deve ser desabilitado quando o contador horizontal atender à seguinte condição:

$$H_{counter} = HT - HB \quad (4)$$

5) *Período de Acionamento do Sinal de Sincronismo Vertical*: O sinal de sincronismo vertical (*vsync*) deve ser acionado quando a seguinte condição booleana for atendida:

$$[V_{counter} = (VD + VF)] \& (H_{counter} = HT) \quad (5)$$

6) *Fim da Permanência do Sinal de Sincronismo Vertical*: O sinal *vsync* deve ser desabilitado quando a seguinte condição booleana for atendida:

$$[V_{counter} = (VT - VB)] \& (H_{counter} = HT) \quad (6)$$

7) *Área Visível da Tela*: Para determiná-la é preciso ter uma visão geral do controle de sincronismo. Esta área é determinada como sendo a região delimitada pelos intervalos de *front porch* e *back porch* vertical e horizontal. Dessa forma, área visível pode ser determinada a partir da seguinte relação booleana:

$$(H_{counter} < HD) \& (V_{counter} < VD) \quad (7)$$

IV. METODOLOGIA

A. Modelagem e Organização da Arquitetura

A primeira etapa do processo de renderização consistiu na inicialização de duas memórias responsáveis por armazenar os bits de cores RGB dos *sprites* e *background* da tela. A segunda etapa corresponde ao controle de impressão dos elementos da tela, como por exemplo, os *sprites*. O padrão gráfico escolhido foi o VGA com uma resolução de 640x480 *pixels*. Como a varredura de um monitor VGA é realizada da esquerda para a direita e de cima para baixo, ao finalizar a varredura de toda a área ativa da tela, os *sprites* podem ser atualizados, de forma que, no próximo *Frame*, eles sejam redesenhadados de acordo às novas definições.

Cada jogo implementado por meio da arquitetura proposta deve ser programado utilizando a linguagem C através de um processador de propósito geral. Como as informações dos *sprites* são armazenadas a nível de hardware, as atualizações são feitas através de instruções (comandos) enviadas ao módulo gráfico responsável pelo processo de renderização dos jogos. O processo de renderização é independente, executa todo o controle sem a necessidade de intervenções via software. Com isso, o código em linguagem C ficará responsável em definir a lógica do jogo e enviar os comandos de atualização das informações pertencentes aos objetos a serem renderizados.

Com base nesse princípio de funcionamento, temos a modelagem da arquitetura apresentada na Fig. 3. Sua estrutura

consiste em um processador de propósito geral, duas FIFOs (*First In First Out*), uma PLL (*Phase Locked Loop*) e um Processador Gráfico. Para atuar como processador de propósito geral foi escolhido o Nios II. Consiste em um processador *softcore* RISC de 32 bits com arquitetura Harvard desenvolvida pela empresa Altera. Sua função é executar o código-fonte em linguagem C dos jogos que serão programados. O Processador Gráfico é responsável por gerenciar o processo de renderização dos jogos e executar um conjunto de instruções que permitem inicialmente mover e controlar *sprites*, como também modificar o *layout* do *background* da tela e renderizar polígonos do tipo Quadrado e Triângulo. As principais saídas do Processador Gráfico consistem nos sinais de sincronização horizontal (*h_sync*) e vertical (*v_sync*) do monitor VGA, e os bits de cores RGB. Como o Nios II e o Processador Gráfico possuem frequências de *clock* e controle distintos, as FIFOs estão sendo utilizadas como dispositivos intermediários para comunicação. A PLL é responsável por gerar as frequências de *clock* necessárias para o correto funcionamento da arquitetura.

O Nios II armazena nas FIFOs todas as instruções que devem ser executadas pelo Processador Gráfico. O módulo Gerador de Pulso é responsável por gerar um único pulso de escrita em sincroniza com o sinal *wrclk*, por consequência, os dados de instrução presentes nos barramentos *dataA* e *dataB* serão armazenados nas FIFOs uma única vez. Cada FIFO possui inicialmente a capacidade de armazenar 16 palavras de 32-bits. Quando o sinal *wrfull* está em nível lógico alto significa que as FIFOs alcançaram sua capacidade máxima. Desta forma, o circuito interno de proteção das FIFOs é ativado automaticamente para evitar possíveis *Overflows*. As FIFOs facilitam o processo de modelagem de novas instruções, não sendo necessário a inclusão de novos barramentos e/ou dispositivos intermediários entre o Nios II e o Processador Gráfico.

B. Arquitetura do Processador Gráfico

É possível observar a representação da estrutura interna deste módulo na Fig. 4. Inicialmente, temos a Unidade de Controle. Ela consiste em uma Máquina de Estados responsável por gerenciar o processo de leitura, decodificação e execução das instruções recebidas. O Banco de Registradores armazena temporariamente as informações (coordenadas, offset de memória, e um bit de ativação) associadas a cada *sprite*. São 32 registradores no total, sendo o primeiro reservado para o armazenamento da cor de *background* da tela e os outros 31 reservados aos *sprites*. Com isso, o Processador Gráfico consegue administrar o uso de 31 *sprites* em um mesmo *Frame* de tela. O Módulo de Desenho é responsável por gerenciar o processo de desenho dos *pixels* no monitor VGA. O Controlador VGA é responsável por gerar os sinais de sincronização *v_sync* e *h_sync* da VGA, além de fornecer as coordenadas x e y do processo de varredura do monitor. Considerando os tempos de sincronização vertical e horizontal, cada tela é impressa a cada 16.768ms. Logo, temos uma taxa de aproximadamente 60 *Frames* por segundo. A Memória de *Sprites* armazena o *bitmap* para cada *sprite*. Sua dimensão

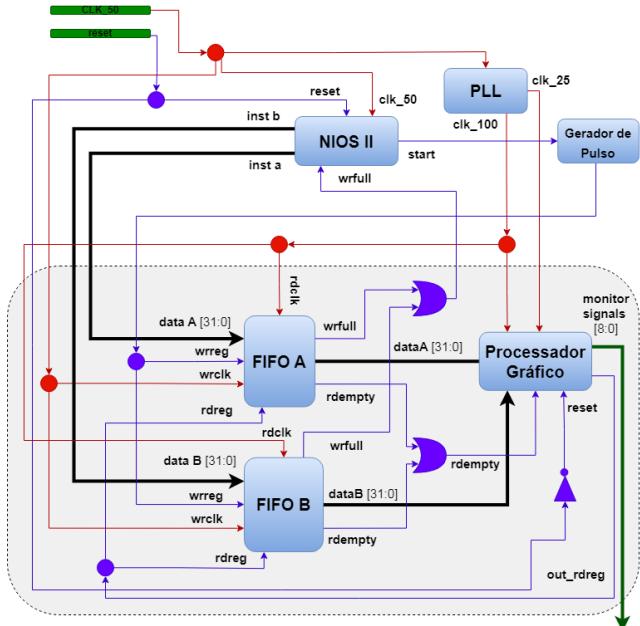


Figura 3: Representação da Arquitetura.

consiste em 12.800 palavras de 9-bits, 3 bits para cada componente RGB. Cada *sprite* deve possuir o tamanho de 20x20, cada um ocupando 400 posições de memória. Desta forma, é possível armazenar 32 diferentes *sprites* para uso em um jogo. A Memória de *Background* é utilizada para modificar pequenas partes do *background* e consiste em 4.800 palavras de 9-bits. A inicialização das memórias é realizado durante o processo de síntese do projeto no software Quartus. Na Figura 4, também encontra-se um Co-Processador no qual é responsável por gerenciar a construção de polígonos convexos do tipo Quadrado e Triângulo. Tais polígonos serão renderizados na tela de um monitor VGA em conjunto com os *sprites* e *background*.

C. Arquitetura do Co-Processador

Um dos problemas enfrentados no processo de construção de polígonos é a análise para definir os pontos (*pixels*) que estão dentro da sua área de renderização. Pelo fato dos quadrados e triângulos serem convexos, essa propriedade ajuda na construção de uma solução simples para este problema por meio da Análise de Colinearidade dos pontos em relação às arestas dos polígonos.

Por definição, temos que, 3 pontos são colineares se eles pertencem a uma mesma reta. Em geometria cartesiana, prova-se que os pontos $P_1 = (X_1, Y_1)$, $P_2 = (X_2, Y_2)$ e $P_3 = (X_3, Y_3)$ são colineares, se somente se, o determinante formado pela matriz 8 for igual a 0.

$$\begin{bmatrix} 1 & X_1 & Y_1 \\ 1 & X_2 & Y_2 \\ 1 & X_3 & Y_3 \end{bmatrix} \quad (8)$$

Verifica-se que se esses 3 pontos não são colineares, o sinal desse mesmo determinante é precisamente a orientação do

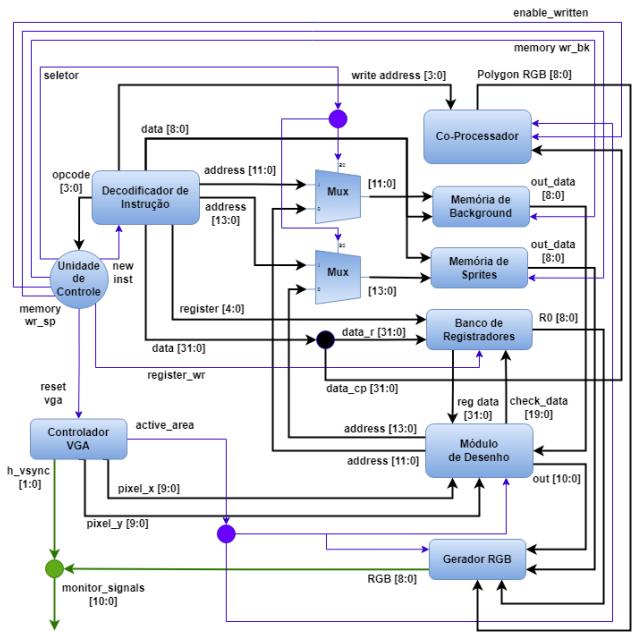


Figura 4: Representação da Estrutura Interna do Processador Gráfico.

triângulo formado pela ligação desses pontos. Caso o determinante $\Delta(P_1, P_2, P_3) > 0$ considera-se a orientação como positiva, caso $\Delta(P_1, P_2, P_3) < 0$ considera-se orientação negativa. Com isso, a partir de um ponto Q qualquer, calculase o determinante formado por este ponto e por dois vértices que definem uma aresta do polígono. O número de determinantes calculados depende da quantidade de arestas que cada polígono possui. Logo, teremos 4 determinantes para os quadrados e 3 para os triângulos. Aplicando essa metodologia pode-se afirmar que, sendo Q um pixel da tela e V_1 e V_2 vértices de uma aresta do polígono, caso o $\Delta(V_1, V_2, Q) = 0$ temos que o pixel Q pertence a aresta formada pelos vértices V_1 e V_2 ou é igual a um deles. Caso $\Delta(V_1, V_2, Q) < 0$, temos que o pixel está à esquerda da aresta. Caso $\Delta(V_1, V_2, Q) > 0$, o pixel está à direita da aresta. Para a análise de orientação positiva ou negativa, seguiu-se o formato apresentado na Figura 5. Desta forma, se os determinantes calculados forem negativos ou iguais a 0, chegamos à conclusão que o pixel Q está dentro da área de renderização do polígono. Se um deles for maior que 0, o pixel Q não pertence ao polígono desejado.

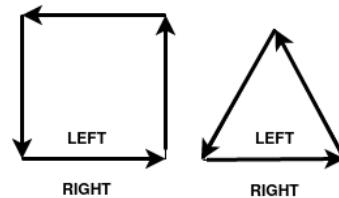


Figura 5: Orientação utilizada para Análise de Colinearidade.

Para o processamento dos polígonos são necessários as seguintes etapas de cálculo:

Definição de base e altura dos polígonos: Por simplicidade os valores de base e altura são iguais. Como padrão, adotou-se o valor de 20 pixels para os dois parâmetros, no entanto, via instrução é possível definir esse dimensionamento entre 20 e 320 pixels, sendo a escolha realizada com valores espaçados de 20 pixels.

Cálculo de vértices: Após a definição do dimensionamento calcula-se os vértices de acordo ao polígono desejado. Um dos parâmetros utilizados além da base e altura para esse cálculo consiste em um ponto de referência definido como o ponto central do polígono. Sendo $ref(x)$ e $ref(y)$ as coordenadas do ponto de referência e $size$ a definição de base e altura, logo, para cada polígono temos que:

Nos Quadrados:

$$V_1(x, y) = (ref(x) - size, ref(y) - size) \quad (9)$$

$$V_2(x, y) = (ref(x) - size, ref(y) + size) \quad (10)$$

$$V_3(x, y) = (ref(x) + size, ref(y) + size) \quad (11)$$

$$V_4(x, y) = (ref(x) + size, ref(y) - size) \quad (12)$$

Nos Triângulos:

$$V_1(x, y) = (ref(x), ref(y) - size) \quad (13)$$

As equações de $V_2(x, y)$ e $V_3(x, y)$ para os Triângulos são iguais aos do Quadrado.

Cálculo de determinantes: Calculando o determinante formado pela matriz 8 utilizando um par de vértices do polígono e um ponto Q qualquer, chega-se à seguinte equação utilizada na Análise de Colinearidade:

$$\Delta(V_1, V_2, Q) = (Y_Q - Y_1).(X_2 - X_1) - (X_Q - X_1).(Y_2 - Y_1) \quad (14)$$

Todas essas etapas de processamento demandam uma quantidade considerável de pulsos de *clock*, não sendo possível analisar todos os polígonos antes que o próximo pixel da tela seja gerado pelo Controlador VGA. Diante disso, para resolver esse problema, o Co-Processador trabalha com um *clock* de 100MHz, além de adotar uma estrutura em *Pipeline* para as etapas de cálculo apresentadas anteriormente.

A representação da arquitetura do Co-Processador pode ser vista na Figura 6. Todas as informações referentes aos polígonos são enviadas via barramento de instrução (Figura 3), e posteriormente, decodificadas e enviadas ao módulo do Co-Processador (Figura 4). O Co-Processador foi estruturado utilizando a técnica de *Pipeline* sendo composto por 5 vias, possibilitando assim que a cada ciclo de *clock* e em um determinado estágio até 5 instruções estejam sendo processadas ao mesmo tempo. Na Figura 6, pode-se observar a Memória de Instrução. Ela é responsável por armazenar os dados referentes aos polígonos, são eles: tamanho, cor, coordenadas X e Y do ponto de referência e forma (Quadrado ou Triângulo). Ela pode armazenar até 16 palavras de 32-bits,

em que, cada palavra define um polígono para renderização. Os módulos do Co-Processador nomeados como *Unidades de Cálculo* são responsáveis por executar as etapas de definição e Análise de Colinearidade dos polígonos em relação aos pixels da tela. Os Bancos de Registrador são responsáveis por armazenar as informações de cada polígono no momento que será realizado o processamento de um novo *frame* de tela. Os dados armazenados na Memória de Instrução são distribuídos previamente em cada Banco antes que se inicie o processamento de um novo *frame*, de forma que, cada via do Co-Processador processe uma determinada quantidade de polígonos. A Unidade de Controle consiste em uma Máquina de Estados responsável por gerenciar o processo de leitura da Memória de Instrução, distribuição dos dados entre os Bancos de Registrador e o sincronismo de execução das instruções. Para que a análise dos polígonos seja feita durante a renderização de um *frame*, é utilizado na saída do Co-Processador uma Memória de Pixels. Essa memória consiste de 640 palavras de 9-bits respectivo às cores RGB de um pixel, além de possuir de forma separada sinais de *clock*, *enable*, e barramentos de endereço específicos para escrita e leitura. O processo de escrita é gerenciado pelo Co-Processador à medida que os polígonos são processados, enquanto o processo de leitura é feito de forma incremental utilizando a frequência de 25MHz que corresponde à velocidade de geração dos pixels pelo Controlador VGA. Desta forma, enquanto o Controlador VGA realiza o sincronismo dos sinais de video e renderiza uma linha do *frame*, o Co-Processador atualiza os endereços que já foram lidos com novos valores referentes à próxima linha a ser renderizada. Interligado nas Unidades de Cálculo, está o módulo Comparador que possui o encargo de verificar qual resultado será escrito na memória após o término do processamento dos polígonos em um determinado pixel. O Contador conectado nas vias do *Pipeline* é responsável por gerar as coordenadas X e Y de acordo à execução de cada instrução, desse modo, a análise de todos os polígonos será realizada a cada pixel. O Contador conectado na Memória de Pixels é responsável por gerar o respectivo endereço de escrita no momento em que o módulo Comparador disponibilizar os dados de cores RGB e o sinal de escrita for habilitado. Como visto na Figura 4, a saída do Co-Processador está conectada ao módulo Gerador RGB, pois internamente será escolhido qual pixel será renderizado na tela dentre a seguinte ordem de prioridade: pixel de *sprite*, polígono e *background*.

A estruturação em *Pipeline* foi definida para que fosse possível realizar o processamento de todos os polígonos para cada pixel de uma linha antes do término do intervalo de varredura horizontal do Controlador VGA (equação 1). Na Figura 7, pode-se visualizar uma representação da temporização do *Pipeline* implementado. Para cada uma das 5 vias, essa análise de temporização é igual devido ao paralelismos das instruções. A cada pulso de *clock* uma nova instrução é lida dos Bancos de Registradores e introduzida nas Unidades de Cálculo, de forma que, nos pulsos subsequentes, é determinado o tamanho dos polígonos que estão sendo processados, calcule-se seus vértices e a Análise de Colinearidade para verificar

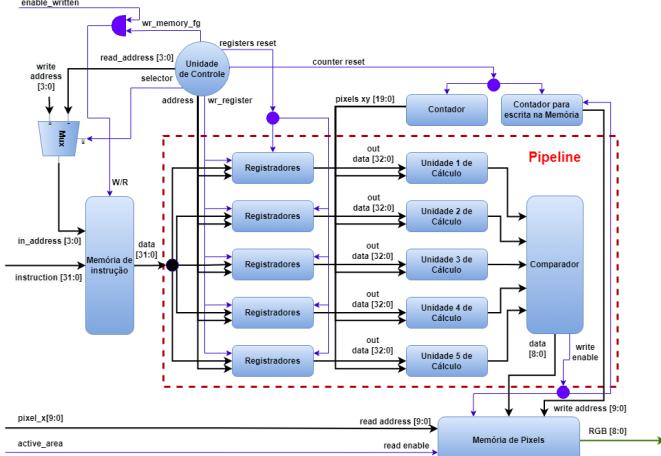


Figura 6: Representação da Arquitetura do Co-Processador.

se um determinado *pixel* está dentro da área de renderização do polígono. O cálculo da equação (14) é dividido em etapas distintas de modo que em cada pulso de *clock* uma parte é realizada. A latência do *Pipeline* implementado corresponde a 8 pulsos de *clock*, que se inicia desde a leitura das instruções distribuídas entre os Bancos de Registrador até o instante em que é liberado o primeiro resultado na saída do módulo Comparador entre 70 e 80ns. Quando o *Pipeline* está cheio, são necessários 3 pulsos para processar todos os polígonos e 1 pulso de bolha. Essa bolha, como visto na Figura 7, serve para especificar que para um determinado *pixel* já foram processados todos os polígonos, logo, o resultado deve ser armazenado na Memória de Pixels. Com o Co-Processador funcionando com uma frequência de *clock* de 100MHz, isso proporciona um limite de 3200 pulsos durante a varredura horizontal. Diante desse limite, uma via do *Pipeline* consegue processar somente 3 polígonos, assim, adotou-se uma estrutura em paralelo com mais 4 vias (Figura 6). Isso viabiliza o processamento de mais 12 polígonos utilizando o mesmo tempo de latência em todas as vias implementadas. Consequentemente, para solucionar o problema da concorrência e assim avaliar qual resultado será armazenado na Memória de Pixels, foi adicionado o módulo Comparador. Deste modo, o processo de escolha dos resultados utiliza como critério de prioridade a ordem das Unidades de Cálculo, ou seja, do primeiro ao último; e o último polígono que entra nas vias do *Pipeline* possui mais prioridade do que o seu antecessor. Isso garante um nível de abstração relativamente simples para tratar o problema de sobreposição de polígonos no momento da renderização. As 3 primeiras instruções da Memória são lançadas para a 1^a via do *Pipeline*, as 3 subsequentes para a 2^a via, e assim por diante. Logo, sabendo das regras pré-definidas para escolha dos polígonos no módulo Comparador e a distribuição das instruções nas vias do *Pipeline*, o controle de sobreposição pode ser feito alocando as instruções na Memória de Instrução de acordo aos polígonos que devem ter precedência no momento da exibição em um *frame*.

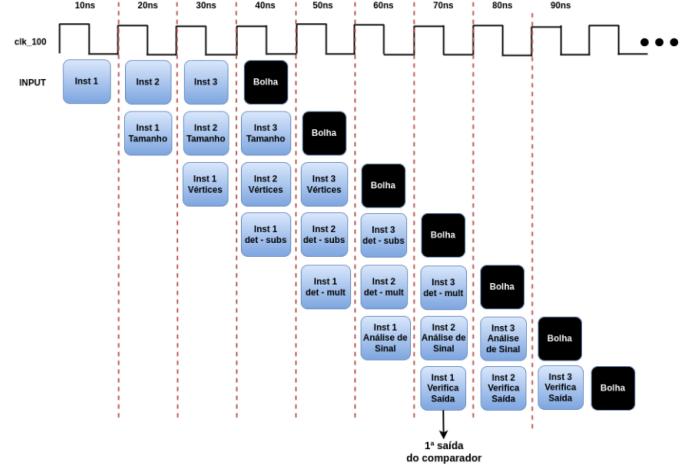


Figura 7: Análise temporal do *Pipeline*.

D. Instruções do Processador Gráfico

Nesta primeira versão do projeto, o Processador Gráfico possui as seguintes instruções:

Escrita no Banco de Registradores (WBR): Essa instrução é responsável por configurar os registradores que armazenam as informações dos *sprites* e a cor base do *background*. Como essa cor base é armazenada no primeiro registrador do Banco, a instrução WBR segue a estrutura apresentada na Fig. 8. As posições com valor 0 representam posições não utilizadas. Para configurar um *sprite* segue-se a estrutura apresentada na Fig. 9. O campo **opcode** define qual instrução será executada pelo Processador Gráfico. Para esta instrução, o valor é configurado em 0000. A escolha do *sprite* é feita através do campo **offset**. Esse *offset* indica a localização do *sprite* na memória. O campo **registrador** é utilizado para definir em qual registrador os parâmetros de impressão serão armazenados. A posição do *sprite* é definida através das coordenadas X e Y. O campo **sp** permite habilitar/desabilitar o desenho de um *sprite* na tela. Na Fig. 8 os campos **R**, **G** e **B** configuram as componentes RGB da cor base do *background*.

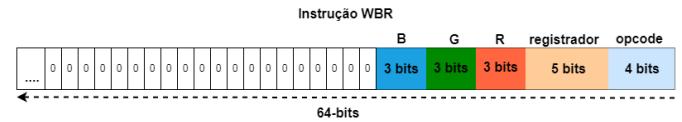


Figura 8: Uso da Instruções WBR para modificação da cor base do *background*.



Figura 9: Uso da Instruções WBR para configurar um *sprite*.

Escrita na Memória de Sprites (WSM): Essa instrução armazena ou modifica o conteúdo presente na Memória de *Sprites*

(Fig. 10). O campo **opcode** é semelhante à instrução anterior, no entanto, seu valor é configurado em 0001. O campo **endereço de memória** especifica qual local da memória será alterado. Os campos **R**, **G** e **B** definem as novas componentes RGB para o local desejado.

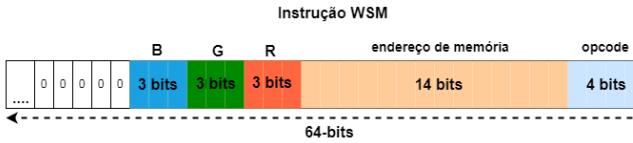


Figura 10: Representação da Instrução WSM.

Escrita na Memória de Background (WBM): Essa instrução armazena ou modifica o conteúdo presente na Memória de *Background*. Sua função é configurar valores RGB para o preenchimento de áreas do *background*. Seus campos são semelhantes ao da instrução WSM (Fig. 10), a única diferença está no campo **endereço de memória** com tamanho de 12 bits. O valor do **opcode** é configurado como 0010. O *background* é dividido em pequenos blocos de 8x8 *pixels* e cada endereço de memória corresponde a um bloco. Sendo a resolução de 640x480 *pixels*, temos uma divisão de 80x60 blocos. Isso permite que o *background* seja configurado de formas diferentes de acordo ao preenchimento da memória (Fig. 11). Se um endereço for preenchido com o valor 0b11111110 = 510, o Módulo de Desenho entenderá que o bloco correspondente está desabilitado, assim ocupando os *pixels* da área com a cor base do *background*, um polígono ou *sprite*, caso suas coordenadas coincidam com o bloco.

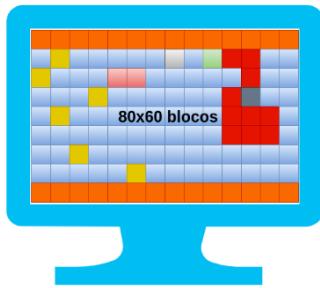


Figura 11: Divisão da área do *Background*.

Definição de um Polígono (DP): Essa instrução é utilizada para modificar o conteúdo da Memória de Instrução do Co-Processador (Fig. 6), de forma a definir os dados referentes a um polígono que deve ser renderizado. O valor do **opcode** é configurado como 0011. O campo **endereço** é utilizado para escolha da posição de memória em que a instrução será armazenada, possibilitando o controle da sobreposição dos polígonos. Os campos **ref_point X** e **ref_point Y** são usados para definir as coordenadas do ponto de referência do polígono. O campo **tamanho** define a dimensão do polígono. Caso seu valor seja configurado como 0b0000, logo, o polígono que foi definido estará desabilitado. Por último, as componentes RGB definem a cor do polígono, e o bit

de **forma** define se o polígono corresponde a um Quadrado = 0 ou Triângulo = 1. Logo abaixo, na tabela II, temos as dimensões que atualmente são possíveis de utilização com o Co-Processador:

Tabela II: Dimensões dos Polígonos

Campo tamanho	Dimensão (Base e altura)
0b0000	polígono desabilitado
0b0001	20x20
0b0010	30x30
0b0011	40x40
0b0100	50x50
0b0101	60x60
0b0110	70x70
0b0111	80x80
0b1000	90x90
0b1001	100x100
0b1010	110x110
0b1011	120x120
0b1100	130x130
0b1101	140x140
0b1110	150x150
0b1111	160x160

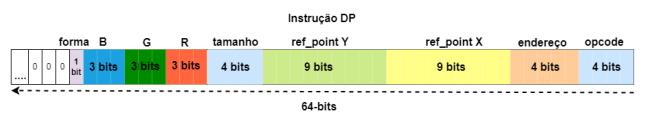


Figura 12: Uso da instrução DP para definição de um polígono.

E. Protocolo de Comunicação entre Nios II e Processador Gráfico

Afim de realizar o envio de instruções para o Processador Gráfico, criou-se um protocolo de comunicação simples utilizando as FIFOs e o módulo Gerador de Pulso, vistos na (Fig. 3). Como os jogos serão desenvolvidos em linguagem C, é possível realizar acesso às GPIOs (General-Purpose Input/Output) do sistema por meio da técnica de Mapeamento de Memória. Através da ferramenta *Platform Design* presente no software Quartus Prime Lite, é realizado a associação de endereços de memória às entradas e saídas do sistema. Desta forma, é obtido acesso a todos os sinais e barramentos ligados ao processador Nios II na Fig. 3. Esse controle de acesso para escrita e leitura é realizado pelo controlador de barramento do Nios II. Com esses endereços e utilizando as Instruções Customizadas do Nios II, é realizado a distribuição dos campos das instruções do Processador Gráfico dentro dos barramentos **dataA** e **dataB** no momento de envio. Como temos instruções com mais de 32-bits, foi decidido manter dois barramentos de 32-bits separados, deixando assim linhas de transmissão suficientes para a implementação de novas instruções de até 64-bits. Nas Fig. 13, 14, 15 e 16 pode-se observar como os dados são distribuídos. O barramento **dataA** é utilizado para **opcodes** e endereçamento do Banco de Registrador e Memórias; e o barramento **dataB** é utilizado para envio de dados a serem armazenados e/ou atualizados. Após a inserção

da instrução nos barramentos **dataA** e **dataB**, deve-se colocar o sinal **start** em nível lógico alto, isso irá ativar o Módulo Gerador de Pulso, no qual, irá habilitar a escrita nas FIFOs durante um único pulso de *clock*. Isso garante que as instruções sejam escritas somente uma vez, fazendo com que não existam instruções iguais, ocasionando a perca de espaço. Logo após, deve-se colocar o sinal **start** novamente em nível lógico baixo, o módulo Gerador de Pulso irá reiniciar, aguardando assim, a próxima mudança na sua entrada de forma a gerar um novo pulso para escrita. Não é possível utilizar um pulso de escrita vindo direto do Nios II, pois as Instruções Customizadas levam em torno de 6 pulsos de *clock* para refletir o novo valor na respectiva saída, isso refletiria em várias escritas nas FIFOs, ocasionando redundâncias de instruções.

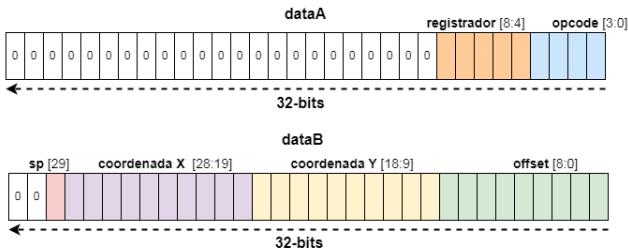


Figura 13: Envio da instrução WBR para definição de um *sprite*.

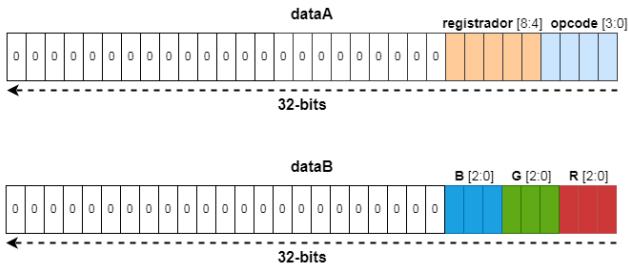


Figura 14: Envio da instrução WBR para definição da cor de *background*.

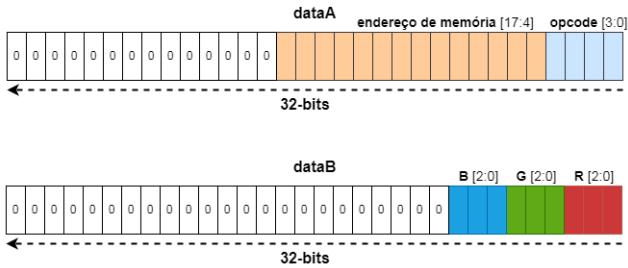


Figura 15: Envio da instrução WSM

F. API Simplificada para auxiliar na produção de Jogos Bidimensionais

Após o término do processo de modelagem e implementação da arquitetura, deu-se início a construção

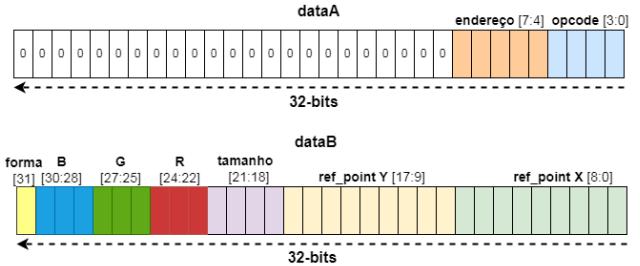


Figura 16: Envio da instrução DP

de recursos em software como funções, constantes e tipos compostos (*struct*); tendo por objetivo abstrair aspectos de baixo nível da arquitetura e assim auxiliar na programação dos jogos em linguagem C. Esses recursos estão disponíveis em um arquivo de cabeçalho (*graphic_processor.h*).

Inicialmente, na Listing 1, temos a definição de algumas constantes utilizadas para fixar valores inteiros que informam a direção ou ângulo de movimento de um *sprite*. Logo após, temos duas *structs* utilizadas para armazenar informações referentes aos *sprites*. A *struct* do tipo *Sprite*, é usada para os *sprites* móveis. Enquanto a *Sprite_Fixed* para os *sprites* fixos, ou seja, não possuem movimentação durante o jogo.

```
#define LEFT      0
#define RIGHT     4
#define UP        2
#define DOWN     6
#define UPPER_RIGHT 1
#define UPPER_LEFT  3
#define BOTTOM_LEFT 5
#define BOTTOM_RIGHT 7
typedef struct {
    int coord_x, coord_y;
    int direction, offset, data_register;
    int step_x, step_y;
    int ativo, collision;
} Sprite;
typedef struct {
    int coord_x, coord_y, offset;
    int data_register, ativo;
} Sprite_Fixed;
```

Listing 1: Definição de Constantes e Estruturas Compostas.

O atributo **coord_x** e **coord_y** armazenam as coordenadas x e y. O **direction** armazena um número inteiro indicando o ângulo de movimento. O **offset** indica o offset de memória utilizado para a escolha do *bitmap* armazenado no Processador Gráfico. O **data_register** indica o registrador do Banco de Registradores (Fig. 4) no qual as informações do *sprite* serão armazenadas. O **step_x** e o **step_y** informam o número de *pixels* que o *sprite* irá se deslocar a cada comando de movimentação. O **ativo** habilita/desabilita a impressão do *sprite* em um determinado momento. O **collision** informa se o *sprite* sofreu alguma colisão.

Na Listing 2, encontram-se as funções que auxiliam no desenvolvimento dos jogos e controle dos *sprites*.

```
int set_sprite( int registrador, int x, int y, int offset, int activation_bit);
```

```

int set_background_block( int column, int line,
    int R, int G, int B);

int set_background_color(int R, int G, int B);

void increase_coordinate(Sprite *sp, int mirror);

int collision(Sprite *sp1, Sprite *sp2);

```

Listing 2: Funções Auxiliares.

A função **set_sprite** é usada para posicionar um *sprite* na tela através da instrução *WBR* do Processador Gráfico. Seus parâmetros são iguais aos campos dessa instrução (Fig. 9). O campo **Opcode** é definido internamente, antes do envio do comando ao Processador. Seu retorno é 1 caso a operação tenha sido enviada com sucesso, e 0 caso contrário.

A função **set_background_block** é usada para modelar o *background* através do preenchimento dos blocos de 8x8 *pixels* apresentados na Fig. 11. Seus parâmetros consistem nas componentes **R**, **G**, **B** e no valor de linha e coluna que representa o bloco que será preenchido. Internamente, os parâmetros de linha e coluna são utilizados para o cálculo do endereço correspondente na Memória de *Background*.

A função **set_background_color** é usada para configurar a cor base do *background*. Seus parâmetros consistem nas suas componentes RGB. Para a configuração, utiliza-se a instrução *WBR* (Fig. 8) do Processador Gráfico. Como o registrador usado para a cor base é fixo, o campo **Register** é definido internamente, assim como o **Opcode**. Seu retorno é 1 caso a operação tenha sido enviada com sucesso, e 0 caso contrário.

A função **increase_coordinate** é responsável por atualizar as coordenadas x e y de um *sprite* móvel de acordo ao seu ângulo de movimento e valor de deslocamento; atributos **direction**, **step_x** e **step_y** respectivamente. O ângulo de movimento do *sprite* é comparado com as constantes apresentadas na Listing 1. Seus parâmetros consistem na passagem por referência de uma variável do tipo *Sprite* e um valor inteiro que informa se as coordenadas do *sprite* devem ser espelhadas ao sair da área ativa do monitor VGA. Após realizada a atualização, deve-se usar a função **set_sprite** para desenhar o *sprite* nas novas coordenadas.

A função **collision** é usada para verificar se ocorreu uma colisão entre dois *sprites* quaisquer. Ela implementa a técnica de Sobreposição de Retângulos, utilizando altura e largura de um *sprite* para definir os limites de área. É verificado se a área de impressão de um *sprite* está dentro da área de impressão do outro (Fig. 17). Seus parâmetros consistem na passagem por referência de duas variáveis do tipo *Sprite* com seus atributos **coord_x** e **coord_y** devidamente preenchidos. Seu retorno é 1 caso a colisão seja detectada, e 0 caso contrário.

V. EXPERIMENTOS E TESTES

A. Simulação da estrutura em Pipeline do Co-Processador

Como forma de validação da estrutura em paralelo dos núcleos do Co-Processador usou-se a ferramenta *Simulation WaveForm Editor* do software Quartus Prime Lite para realizar

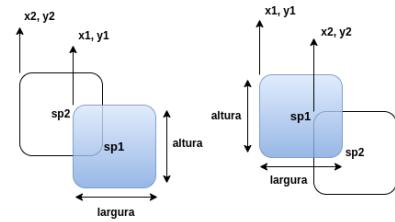


Figura 17: Exemplo da Sobreposição de Retângulos aplicada a 2 *sprites*.

uma análise temporal do circuito digital proposto. O resultado da simulação pode ser vista na Figura 18. É possível visualizar que a cada pulso de *clock* são inseridas todas as informações referentes a um polígono. A cada pulso de *clock* de 100MHz uma nova instrução é inserida nos *Pipelines*. Por simplicidade, adotou-se a mesma coordenada Y (linha 16) de referência em todas as entradas, somente variando as coordenadas X (linhas 9 a 14). No pulso do *clock* em 30ns uma bolha é colocada no *Pipeline* para que o resultado das instruções anteriores seja avaliada e escrita na Memória de Pixels. Logo após, outras entradas são inseridas e uma nova bolha é colocada nos *Pipelines*. Nota-se que em 74ns o resultado das 5 primeiras instruções inseridas no primeiro pulso de *clock* é liberada na saída do módulo Comparador (linha 17). Nos próximos pulsos, a saída é modificada, isso significa que um novo polígono inserido no *Pipeline* também possui em sua área de renderização as coordenadas X e Y atuais, fazendo com o resultado anterior seja sobreescrito. Nota-se também que, em aproximadamente 164ns, após a ativação do sinal de área ativa (linha 4) a primeira leitura na Memória de Pixels é liberada (linha 21). O valor lido corresponde justamente ao último polígono detectado dentre as instruções inseridas antes da primeira inserção do pulso de bolha. Assim, a cada pulso de *clock* de 25MHz, um novo valor é liberado. Quando nenhum polígono é detectado para um determinado pixel o valor 510 é inserido na Memória. Este valor é utilizado para representar uma cor "invisível", que não deve ser renderizada. Desta forma, comprova-se o funcionamento da estrutura em paralelo dos 5 núcleos utilizados na arquitetura do Co-Processador em conjunto com o módulo Comparador e a Memória de Pixels, assim como, o tempo de latência do *Pipeline*.

B. Simulação da Unidade de Controle do Co-Processador

Na Figura 19, é possível visualizar o processo de leitura dos dados na Memória de Instrução em conjunto com a escrita nos Bancos de Registrador. Logo, após o sinal de **reset** (linha 3) ser desabilitado (ativo em nível lógico 0), o módulo passa por um estado intermediário utilizado para habilitar a leitura na Memória de Instrução (linha 7), configurar o sinal de seleção do multiplexador (linha 8) para que seja selecionado os endereços de leitura, habilitar escrita para os Bancos de Registradores (linha 9), e desabilitar o seu respectivo sinal de **reset** (linha 10). Posteriormente, a partir dos 70ns é iniciado o processo de leitura na Memória de Instrução e distribuição

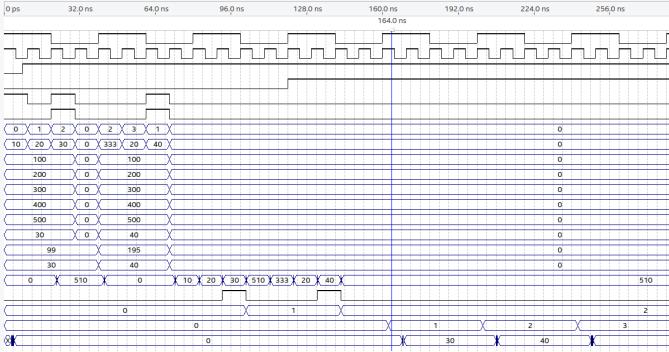


Figura 18: Simulação do processamento dos polígonos utilizando os núcleos do Co-Processador.

dos dados em cada Banco de Registrador. Nas linhas 12, 13 e 14 pode-se observar a geração dos endereços de memória em conjunto o endereço de escrita nos Bancos. Na linha 15, temos os valores lidos da Memória de Instrução. Tais valores são meramente incrementais para fins de validação do processo em análise. No instante localizado em 70ns é realizado a primeira leitura, logo após, no pulso seguinte, o respectivo dado é escrito em um dos Bancos de acordo a faixa de endereço que cada um possui. Dando seguimento, nas bordas de descida do pulsos de escrita, novos endereços são gerados para que uma nova leitura possa ocorrer. Isso se repete até que todas as 15 instruções da memória estejam distribuídas entre os Bancos.

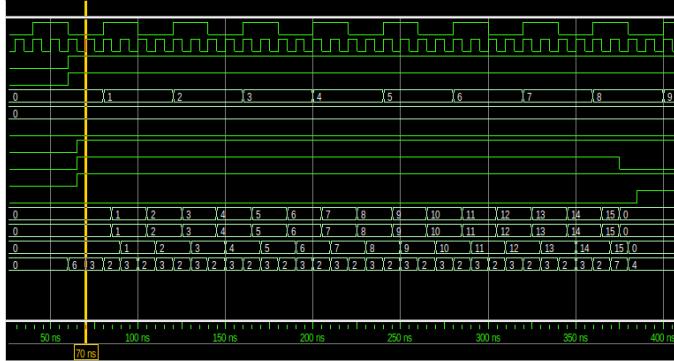


Figura 19: Simulação da leitura na Memória de Instrução e Escrita de dados nos Bancos de Registrador

Após o término da distribuição das instruções entre os Bancos de Registrador percebe-se na Figura 20, na linha 12, que a Unidade de Controle entra no estado de Processamento das Instruções. Esse estado consiste em realizar a leitura das instruções em todos os Bancos de Registrador e direcioná-las aos núcleos do Co-Processador. Como mencionado anteriormente, cada núcleo processa 3 instruções que representam polígonos que devem ser renderizados, logo, são realizadas 3 leituras e uma 1 bolha é lançada no *Pipeline* (isso pode ser observado nos instantes em que as entradas são zeradas). Também observa-se que, enquanto a tela está sendo renderi-

zada (linhas 4 e 6), todos as instruções são lidas repetidamente a cada pixel (linha 5), sendo consequentemente lançadas no *Pipeline*. Desta forma, é possível processar novos *pixels* que já foram renderizados, deste modo, não distorcendo a saída no monitor VGA. A Unidade de Controle permanece nesse estado de processamento até que todos os *pixels* da nova linha que estão dentro da área de renderização de cada polígono estejam devidamente processados. Após isso, a Unidade de Controle permanece em um estado de espera até o instante em que todo o processamento deve ser reiniciado.

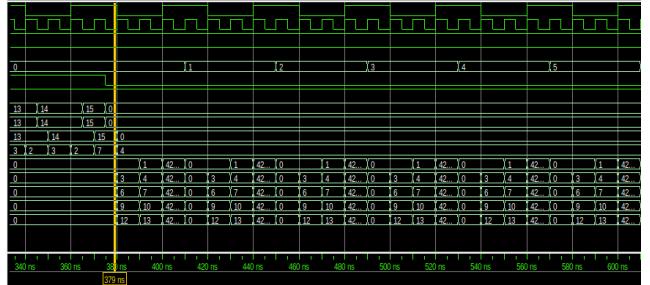


Figura 20: Simulação da leitura nos Bancos de Registrador em sincronia com a geração dos *pixels*.

Com esses testes, prova-se o funcionamento da arquitetura proposta para o Co-Processador. Sua arquitetura utilizando o conceito de *Pipeline* permitiu o processamento em tempo real dos polígonos, e sua estrutura modular com mais de uma via possibilitou tornar a arquitetura escalável em relação à quantidade de instruções por ciclo de *clock*, pois permite que novas vias sejam inseridas, aumentando desta forma o número de polígonos possíveis de serem processados. Como não temos dependência de dados no *Pipeline*, essa inserção se torna simples desde que o chip FPGA possua elementos lógicos suficientes para a inclusão de novas vias, e bits de memória que atenda à quantidade de instruções desejadas na Memória de Instrução.

C. Validação da Arquitetura

Para validação da arquitetura proposta nesse trabalho foram criados dois jogos baseados nos clássicos *Asteroids* e *Space Invaders*. Na Fig. 21, é possível visualizar uma representação geral do fluxo de execução de ambos os jogos desenvolvidos.

A primeira ação realizada é a inicialização dos *sprites* que serão utilizados durante o jogo. Na Fig. 22, é possível visualizar 3 tipos de *sprites* que foram utilizados para construção do jogo *Asteroids*; e na Fig. 23, temos 4 exemplos de *sprites* utilizados no jogo *Space Invaders*. No momento em que os *bitmaps* são gerados, a cor preta é substituída pela cor invisível mencionada na seção V, de forma que, somente o corpo dos objetos sejam renderizados. O algoritmo de geração dos *bitmaps* ficará a cargo do desenvolvedor. Como o *bitmap* dos *sprites* já estão carregados na Memória de *Sprites* do Processador Gráfico, o processo de inicialização em software consiste em declarar e inicializar variáveis de tipos compostos que irão armazenar suas informações. Para isso, foi

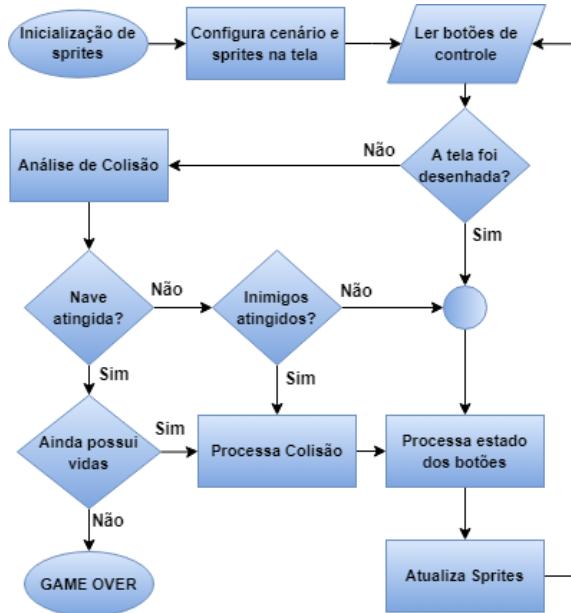


Figura 21: Fluxo de Execução dos Jogos.

utilizada a *struct Sprite* apresentada na seção anterior. Em seguida, através das funções *set_sprite*, *set_background_color* e *set_background_block*, são inseridos na tela os *sprites* iniciais, a cor base de fundo e construção do cenário (para o caso do *Space Invaders*).

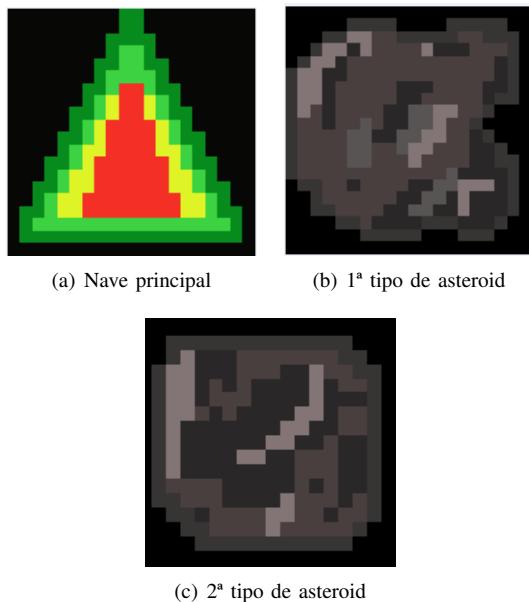


Figura 22: Exemplo de *Sprites* utilizados no jogo *Asteroids*.

Dando início ao loop principal, temos a leitura dos botões de controle. Essa leitura é feita através da função *IORD(base, 0)* disponível no arquivo *altera_avalon_pio_regs.h*. O parâmetro *base* consiste no endereço de memória dos registros que armazenam o estado de cada botão. Esses endereços são

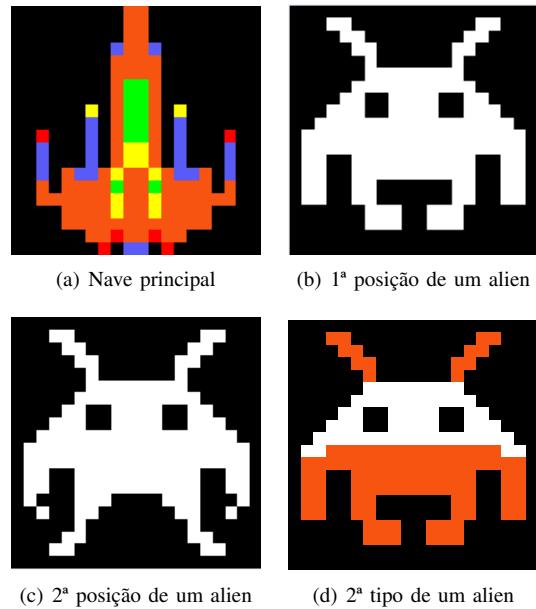


Figura 23: Exemplo de *Sprites* utilizados no jogo *Space Invaders*.

encontrados no arquivo *system.h*. As duas bibliotecas foram desenvolvidas pela empresa Altera para facilitar o acesso aos periféricos conectados ao sistema do processador Nios II. O acesso a essas bibliotecas é feito após a criação do projeto na IDE (*Integrated development environment*) Eclipse integrada à plataforma Quartus Prime.

Realizada a leitura dos botões, é verificado se o tempo de varredura de uma tela já foi atingido. A Listing 3 mostra um exemplo da leitura de todos os botões mapeados em memória e a verificação do tempo de varredura da tela.

```

#include "altera_avalon_pio_regs.h"
#include "system.h"

#define A_BASE 0x11160
#define B_BASE 0x11130
#define X_BASE 0x11110
#define Y_BASE 0x11120
#define SELECT_BUTTON_BASE 0x110f0
#define START_BASE 0x11100
#define DIRECTION_ANALOGIC_BASE 0x11170
#define TL_BASE 0x11140
#define TR_BASE 0x11150
#define SCREEN_BASE 0x11180

int main(){
    int start_pause = IORD(START_BASE, 0);
    int TR      = IORD(TR_BASE, 0);
    int TL      = IORD(TL_BASE, 0);
    int A       = IORD(A_BASE, 0);
    int move   = IORD(DIRECTION_ANALOGIC_BASE, 0);
    int screen_number = 0;
    if(IORD(SCREEN_BASE, 0) == 1){
        // tempo atingido.
        screen_number++;
    }
    return 0;
}

```

Listing 3: Leitura de Botões e Status de Varredura da Tela

De acordo à Fig. 21, caso o tempo de varredura não seja atingido, é realizado o processamento de colisão entre os *sprites*. Para o jogo *Asteroids* o jogador só irá possuir uma vida, desta forma, se a nave for destruída o jogo é encerrado. Para o *Space Invaders* o jogador terá 3 vidas. Caso a nave seja atingida e o mesmo possua vidas extras, o jogo é retornado à sua execução normal após um período de 4 segundos a partir do estágio em que o jogador se encontra.

Caso o tempo de varredura tenha sido atingido, são processados os estados de cada botão de controle, e os *sprites* são atualizados de acordo às novas informações. A Listing 4 mostra um exemplo de atualização dos *sprites*.

```

if(number_screen == 1) {
    screen_number = 0;
    if(IORD(WRFULL_BASE,0) == 0) {
        if(state_game == 0) {
            increase_coordinate(&ast_1,1);
            increase_coordinate(&ast_2,1);
            // atualizar sprite 1
            set_sprite(sp_1.data_register,
                       sp_1.coord_x,sp_1.coord_y,
                       sp_1.offset,
                       sp_1.ativo
            );
            // atualizar sprite 2
            set_sprite(sp_2.data_register,
                       sp_2.coord_x,sp_2.coord_y,
                       sp_2.offset,
                       sp_2.ativo
            );
        }
    }
}

```

Listing 4: Atualização de *Sprites*.

O endereço de memória armazenado em *WRFULL_BASE* é utilizado na função IORD para verificar o estado de armazenamento (0 = vazio, 1 = cheio) das FIFOs de instrução (Fig. 3). Caso uma tela tenha sido desenhada, as FIFOs estejam vazias e o jogo em andamento, então os *sprites* são atualizados através da função *increase_coordinate* e os comandos de atualização transmitidos ao Processador Gráfico por meio da função *set_sprite*.

Toda a arquitetura foi compilada e sintetizada na plataforma Quartus Prime Lite 20.1. Os sinais para controle de movimentação, tiros da nave, e operações de pausar e retomar o jogo são obtidos por meio de um hardware externo com botões e um *joystick* (Fig. 24). A Fig 25 mostra o resultado final alcançado após o processo de desenvolvimento e testes.

Para os testes, utilizou-se a placa de desenvolvimento DEO-Nano acoplada com o chip FPGA *Altera Cyclone IV EP4CE22F17C6N*. Foi possível perceber que realizar uma animação com o limite de 31 *sprites* na tela ocasiona uma perda na velocidade de atualização devido ao limite de envio de 13 instruções por *Frame*. Pretende-se aumentar esse valor, tendo por objetivo possibilitar o controle de mais *sprites* em um mesmo *Frame*, desta forma alcançando um maior desempenho. Com a arquitetura implementada e validada, na tabela III, pode-se visualizar uma análise comparativa em relação a arquitetura desenvolvida e os projetos mencionados na seção II.



Figura 24: Protótipo Inicial.



(a) Jogo Asteroids.



(b) Tela de GAME OVER do jogo Asteroids.



(c) Jogo Space Invaders.

Figura 25: Jogo Space Invaders e Asteroids.

VI. CONCLUSÃO

Este artigo apresentou a arquitetura, o conjunto de instruções e a validação de um Processador Gráfico baseado em sprites que permite mover e controlar elementos em um monitor VGA com resolução de 640x480 pixels, além da modelagem e simulação dos módulos de um Co-Processador que permite desenhar dois tipos de polígonos convexos (Quadrado e Triângulo) utilizando uma estruturação baseada no

Tabela III: Análise Comparativa com a Arquitetura desenvolvida

Análise Comparativa	Arquitetura Desenvolvida	Jogos em linguagens HDL	Jogos em MCU
Permite Sprites	Sim	Sim	Não
Necessita novo hardware de animação	Não	Sim	Não
Permite modificar hardware externo para iteração	Sim	Sim	Não
Permite aprimorar habilidades de programação	Sim	Não	Sim
Permite aprimorar habilidades de integração entre hardware e software	Sim	Não	Não
Permite integração com novos módulos em hardware	Sim	Sim	Não

conceito de *Pipeline*. Este projeto provou ser uma solução viável para o desenvolvimento de jogos e a construção de uma plataforma de ensino utilizando os jogos como principal motivação. Este projeto ainda está em desenvolvimento, com algumas outras etapas a serem aprimoradas e validadas, como por exemplo, a implementação de um módulo HDMI (*High-Definition Multimedia Interface*). Isso permitiria exibir no display LCD que pode ser visto acoplado ao hardware externo da Figura 24 todos os jogos desenvolvidos. Pretende-se também desenvolver melhorias no armazenamento dos *sprites*, de forma a proporcionar maior flexibilidade ao programador no momento de realizar o processo de upload dos *sprites* em cada jogo, sem a necessidade de recompilar toda a arquitetura.

REFERÊNCIAS

- [1] C. A. Paiva and R. Tori, “Jogos digitais no ensino: processos cognitivos, benefícios e desafios,” *XVI Simpósio Brasileiro de Jogos e Entretenimento Digital*, pp. 1–4, 2017.
- [2] E. Braga, M. Santana, S. L. O. Duarte, and L. C. Lacerda, “Jogos eletrônicos,” *Revista Diálogos: Economia e Sociedade* (ISSN: 2594-4320), vol. 4, no. 2, pp. 55–72, 2020.
- [3] D. K. Ramos and F. S. C. Pimentel, “Cognição, aprendizagem e jogos digitais,” *BG BUSINESS GRAPHICS EDITORA*, p. 13, 2021.
- [4] N. Singla and M. S. Narula, “Fpga implementation of snake game using verilog hdl,” 2018.
- [5] S.-M. Xia, X.-L. Xu, L. Qin, and C.-H. Liu, “Galaxian game on altera de2-115 fpga architecture.”
- [6] A. Mishra, A. Kumarb, and R. Parihar, “Design and fpga implementation of space shoot game,” *International Journal of Control Theory and Applications*, vol. 10, no. 30, 2017.
- [7] A. Sharma, “One dimensional pong game using finite state machine on a field programmable gate array.”
- [8] C. J. Jiménez-Fernández, C. B. Oliva, P. P. Fernández, A. G. Soto, F. E. P. Ordóñez, and M. V. Barrero, “Learning vhdl through teamwork fpga game design,” in *2020 XIV Technologies Applied to Electronics Teaching Conference (TAEE)*, 2020, pp. 1–5.
- [9] “Gamebuino,” <https://gamebuino.com/>, accessed in: 2022-01-18.
- [10] “Arduino esplora,” <https://www.arduino.cc/en/Main.ArduinoBoardEsplora>, accessed in: 2022-01-18.
- [11] G. L. Ronald J.Tocci, Neal S.Widmer, *Sistemas Digitais Princípios e Aplicações*. Pearson, 2019.