UNIVERSIDADE FEDERAL DE SÃO CARLOS
Centro de Ciências Exatas e de Tecnologia
Departamento de Computação

Arquiteturas de Alto Desempenho
Prática 3
Implementação do Jogo da Vida de Conway em FPGA

Guilherme Gomes Arcencio - 769731 - Engenharia de Computação
Guilherme Silva Castro - 769763 - Ciência da Computação

1 Introdução

Neste experimento, foi implementado o Jogo da Vida de Conway em uma FPGA (Field-Programmable Gate Array), com a saída visualizada através de uma interface VGA (Video Graphics Array). O Jogo da Vida, um autômato celular de regra simples, mas com comportamento emergente complexo, oferece um excelente estudo de caso para explorar a arquiteturas de processamento paralelo. O uso de FPGA permite a síntese de hardware dedicado, otimizando o desempenho para a simulação do jogo em tempo real, aproveitando sua capacidade intrínseca de processamento paralelo para atualizar simultaneamente o estado de cada célula na grade.

2 Desenvolvimento teórico

O Jogo da Vida de Conway [1] é um autômato celular que se desenrola em uma grade bidimensional infinita, na qual cada célula existe em um de dois estados: viva ou morta. A cada instante, as células evoluem simultaneamente para um novo estado de acordo com regras simples, mas cuja interação gera comportamentos complexos e inesperados. A configuração inicial, chamada de "semente", pode ser qualquer conjunto de células vivas distribuídas no tabuleiro, e a dinâmica resultante desse arranjo revela como padrões aparentemente caóticos podem, ao longo de várias gerações, se organizar em estruturas estáticas, oscilantes ou até mesmo viajantes.

As regras que governam essa evolução baseiam-se exclusivamente no número de vizinhas vivas que cada célula possui. Se uma célula viva tiver exatamente dois ou três vizinhos vivos, ela permanece viva; caso contrário, morre, seja por solidão (menos de dois vizinhos) ou por superpopulação (mais de três vizinhos). Por outro lado, uma célula morta se torna viva somente se estiver rodeada por exatamente três células vivas. Todas as demais células mortas permanecem mortas. Essas condições conferem ao sistema uma capacidade surpreendente de gerar comportamentos auto-organizados.

Mesmo com regras locais e determinísticas, o Jogo da Vida apresenta fenômenos emergentes que variam de padrões estáveis, que não mudam com o tempo, até os osciladores, que retornam periodicamente a estados anteriores, passando pelos "*gliders*" — estruturas que se deslocam pela grade. Há ainda configurações capazes de refletir operações lógicas e simular, em tese, qualquer cálculo que uma máquina de Turing possa executar.

3 Desenvolvimento prático

Inicialmente, foi escolhida uma semente inicial para a simulação do Jogo da Vida, mostrada na Figura 1, que consistia em uma "*glider gun*", um padrão que gera *gliders* inifinitamente em uma direção. A grade foi criada com dimensões de 64 por 48 céluas; visto que a resolução da interface VGA é 640x480 pixels, foi possível mapear cada célula para um quadrado de 10x10 pixels na tela.

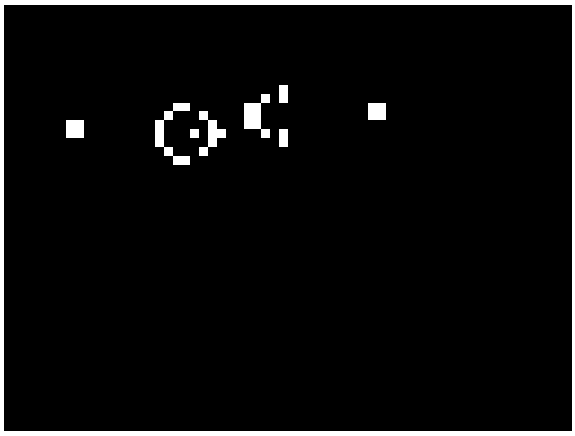

Figura 1: Configuração inicial do Jogo da Vida a ser simulado.

Para fazer a conversão entre coordenadas do VGA (em pixels) para coordenadas do jogo (em células) foi criado um módulo simples em Verilog:

```
module pixel_to_cell(
        input [9:0] row,
        input [9:0] column,
        output [5:0] c_row,
        output [5:0] c_column
);
        assign c_row = row / 10;
        assign c_column = column / 10;
endmodule
```

Além disso, o *clock* de 50Mhz usado pelo módulo VGA fornecido é muito rápido para que a evolução do jogo seja perceptível. Por isso, foi criado em Verilog um módulo contador que gera um sinal de *clock* muito mais lento:

```
module clock_counter(
        input clk,
        output advance_clk
```

```
);
        reg [22:0] counter;

        always @(posedge clk) begin
                counter <= counter + 1;
        end

        assign advance_clk = counter[22];
endmodule
```

Por fim, a implementação do jogo em si foi feita utilizando um registrador para cada célula do tabuleiro. Foi gerado, para cada registrador/célula, um circuito combinacional que lê o valor dos registradores vizinhos e calcula o estado da célula na próxima geração de acordo com as regras do jogo.

Apesar de resultar em uma grande quantidade de uso de *hardware*, essa estratégia permite que o cálculo de cada etapa ocorra de forma completamente paralela entre as células. O código em Verilog desenvolvido é mostrado a seguir:

```
module jogo(
  input advance_clk,
  input reset,
  input [5:0] row,
  input [5:0] column,
  output pixel
);
  parameter ROWS = 48;
  parameter COLUMNS = 64;

  reg [COLUMNS-1:0] board [0:ROWS-1];
  reg [COLUMNS-1:0] next_board [0:ROWS-1];

  always @(posedge advance_clk or posedge reset) begin
    integer i, j;

    if (reset) begin
            // Aqui são instanciados
            // os valores da configuração
            // inicial
    end else begin
```

```
22        for (i = 0; i < ROWS; i = i + 1) begin
23          for (j = 0; j < COLUMNS; j = j + 1) begin
24            board[i][j] <= next_board[i][j];
25          end
26        end
27      end
28    end
29
30    always @(*) begin
31      integer i, j, neighbors;
32
33      for (i = 1; i < ROWS - 1; i = i + 1) begin
34        for (j = 1; j < COLUMNS - 1; j = j + 1) begin
35          neighbors = board[i-1][j-1] + board[i][j-1] + board[i+1][j-1] +
36                  board[i-1][j]   +                 board[i+1][j]   +
37                  board[i-1][j+1] + board[i][j+1] + board[i+1][j+1];
38
39          if (board[i][j]) begin
40            next_board[i][j] = neighbors == 2 || neighbors == 3;
41          end else begin
42            next_board[i][j] = neighbors == 3;
43          end
44        end
45      end
46    end
47
48    assign pixel = board[row][column];
49  endmodule
```

O diagrama do projeto completo é mostrado na Figura 2.

4 Discussão dos resultados

Após a programação da placa Altera DE10 com o circuito projetado, esta foi conectada ao monitor através de um cabo VGA. A exibição da simulação é mostrada na Figura 3.

A simulação fluiu de forma suave. O excesso de hardware gerado pelo circuito foi compensado pelo processamento totalmente paralelo das etapas do jogo.

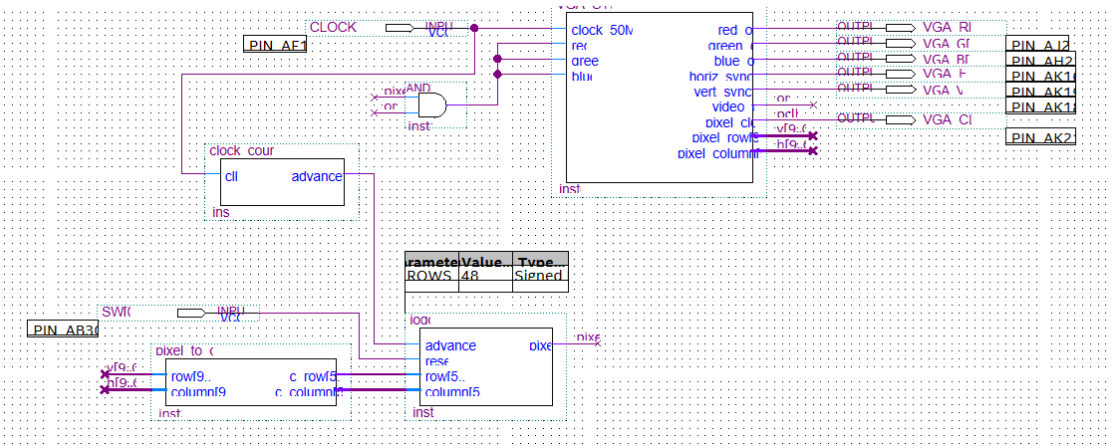

Figura 2: Diagrama completo da simulação paralela do Jogo da Vida.

5 Conclusões

A implementação do Jogo da Vida de Conway em uma FPGA demonstrou com sucesso a viabilidade do uso de hardware dedicado para o processamento paralelo dessa simulação. A estratégia de design, que atribuiu um registrador e um circuito combinacional para cada célula, permitiu que a atualização do estado do tabuleiro ocorresse de forma totalmente paralela, resultando em uma simulação fluida e em tempo real. Apesar do uso intensivo de recursos de hardware, a abordagem paralela contornou a necessidade de um processamento sequencial, garantindo o desempenho necessário para a dinâmica do Jogo da Vida.

Referências

1 GARDNER, Martin. The fantastic combinations of John Conway's new solitaire game "life". **Scientific American**, v. 223, n. 120-123, p. 11, 1970.

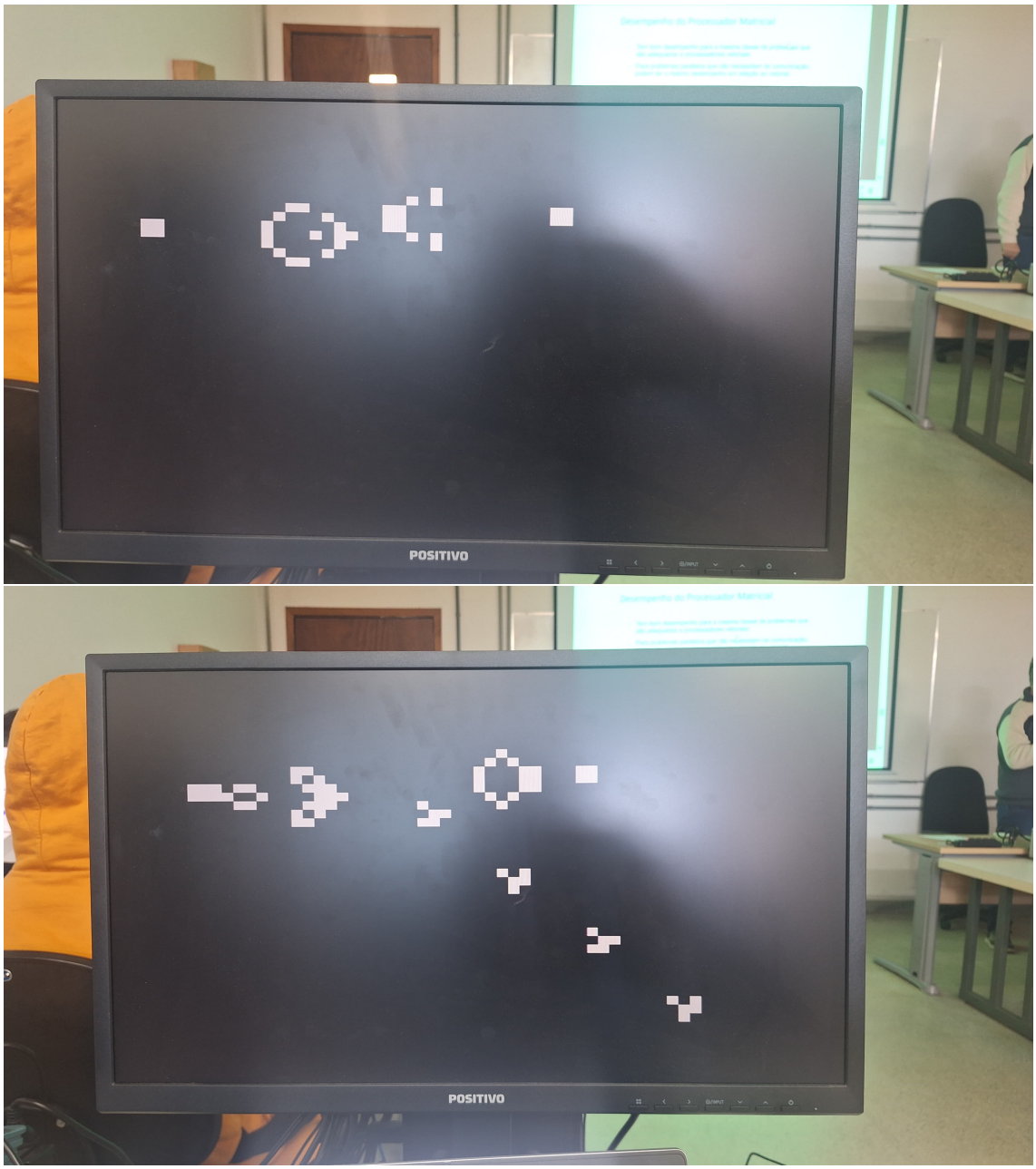

Figura 3: Exibição da configuração inicial do Jogo da Vida e etapa posterior, respectivamente.

Apêndice

1. Faça uma pesquisa comparativa entre Processadores Vetoriais e Matriciais, destacando algumas arquiteturas comerciais e suas principais características; Processadores vetoriais exploram paralelismo de dados no tempo: uma única instrução é aplicada sequencialmente a todos os elementos de um vetor, graças a registradores vetoriais, pipelines e múltiplas "lanes" de ULA que permitem sobreposição de cargas e operações. Entre as arquiteturas comerciais mais conhecidas estão o Cray-1, que oferecia 64 registradores de 64 bits e diversas unidades em pipeline para multiplicação e adição, e a série NEC SX, que seguiu filosofia semelhante de vetores longos e alto throughput.

Já os processadores matriciais, também chamados de Array Processors, exploram paralelismo de dados no espaço: várias unidades de processamento elementar (PEs) executam a mesma instrução sobre diferentes dados simultaneamente. Existem dois tipos principais: os Attached Array Processors, acoplados a computadores convencionais para acelerar cargas numéricas (por exemplo, o VAX-11 com interface FSP-164/Max), e os SIMD Array Processors, em que centenas de PEs independentes são controlados por uma única unidade de controle comum e conectados por redes dedicadas, como a topologia de toro 2D do Illiac IV. Cada PE costuma ter memória local, ULA e registradores próprios, e a comunicação entre PEs é tratada por mecanismos de forwarding ou switches de I/O configuráveis.

2. Dada a arquitetura apresentada no slide 26, mostre um problema de comunicação que pode ocorrer, por exemplo, para uma dada multiplicação de dois vetores X e Y de tamanho n que são distribuídos de acordo com seu índice i. Por trabalhar com PEs que têm memória local, um simples produto de vetores pode incorrer em tráfego intenso sempre que um operando não estiver "no lugar certo". Suponha que os elementos de X e Y estejam mapeados de modo distinto sobre as PEs (por exemplo, X[i] distribuído por linhas e Y[i] por colunas). Para calcular Z[i] = X[i] x Y[i], cada PE que contém X[i] terá de buscar Y[i] na memória de uma PE vizinha. Se muitos PEs fizerem isso simultaneamente, vários pacotes confluirão nas mesmas conexões, gerando congestionamento e elevando o tempo de comunicação muito além do tempo de cálculo local.

3. Aborde sobre as vantagens e desvantagens de arquiteturas MSIMD. Vantagens:

- Escalabilidade elevada: combina o poder de vários vetores independentes, permitindo expansão flexível conforme a demanda. Isso resulta em grande largura de banda agregada para computação científica e numérica.

- Performance ótima em cargas vetoriais massivas: cada SIMD lida eficientemente com operações de vetores longos, maximizando throughput.
- Flexibilidade de controle: embora cada vetor seja processado de forma uniforme, diferentes unidades podem executar algoritmos ou etapas distintas da aplicação.

Desvantagens:

- Complexidade de interconexão: requer redes de comunicação sofisticadas para coordenar as unidades SIMD e movimentar dados entre elas, o que eleva custo e dificultam balanceamento de carga.
- Overhead de sincronização: apesar de cada SIMD operar de forma autônoma, a coordenação geral exige mecanismos de controle complexos e podem introduzir latência.
- Uso restrito a domínios específicos: ideal para tarefas científicas ou numéricas que exploram vetores massivos, não é eficiente para workloads irregulares ou código dinâmico.

4. Explique a ideia do uso de memória intercalada em arquitetura de computadores e compare o diagrama de tempo de uma arquitetura com memória convencional x uma arquitetura com intercalamento de memória. A memória intercalada (interleaving) consiste em dividir o espaço de endereçamento em k "bancos" independentes que podem operar simultaneamente, de modo que cada endereço i é mapeado para o banco (i mod k). isso permite que, enquanto um banco realiza a operação de leitura ou escrita (que pode levar vários ciclos de clock), a CPU inicie acessos a outros bancos sem precisar esperar o término do primeiro, aumentando significativamente a largura de banda disponível.

Em uma organização convencional de memória, há apenas um banco: cada acesso ocupa o barramento e o controlador por todo o seu tempo de ciclo (por exemplo, 3 ciclos). Em contraste, com quatro bancos intercalados ($k = 4$) e mesma latência de 3 ciclos por banco, podemos escalonar as requisições: a cada ciclo de clock começa um novo acesso a um banco diferente, e, embora cada um demore 3 ciclos para responder, a CPU recebe um dado por ciclo após o pipeline inicial. isso reduz o intervalo efetivo entre respostas de 3 ciclos para 1 ciclo, multiplicando por k a largura de banda sem alterar a latência individual de cada banco.

5. Para o exemplo de cálculo mostrado no slide 10, prove que as operações destacadas em vermelho irão ocorrer nos ciclos 9 e 10 respectivamente. Seja:

- Mk,s = estágio s da multiplicação para o k-ésimo elemento do vetor
- Ak,s = estágio s da adição para o k-ésimo elemento do vetor

Em um pipeline puro de 4 estágios, cada novo elemento entra no estágio 1 uma unidade de tempo (ciclo) depois do anterior. Assim:

1. A multiplicação do elemento 1 (k = 1) passa pelos estágios M1,1 no ciclo 1, M1,2 no ciclo 2, ..., até M1,4 no ciclo 4.
2. Graças ao encadeamento, o resultado útil de M1,4 pode ser imediatamente encaminhado como entrada para A1,1 no ciclo 5, seguindo A1,2 no 6, A1,3 no 7 e A1,4 no 8.

Em paralelo, o segundo elemento (k = 2) iniciou M2,1 no ciclo 2, M2,2 no 3, M2,3 no 4 e M2,4 no 5. Com isso:

- M3,3 ocorre no ciclo (3 + (3-1)) = 5
- M4,2 ocorre no ciclo (2 + (4-1)) = 5

e assim por diante, sempre seguindo a fórmula ciclo de $M_{k,s} = s + (k - 1)$. Na ilustração do slide, as operações vermelhas correspondem, respectivamente, a:

- M6,3 (o terceiro estágio de multiplicação para k = 6)
- A6,1 (o primeiro estágio de adição para k = 6)

Aplicando a fórmula:

- Para M6,3: ciclo = 3 + (6-1) = 9
- Para A6,1: como a adição de um elemento só pode começar uma vez que o estágio 4 da multiplicação termine (e mais um ciclo para forwarding), temos ciclo de A6,1 = 4 + (6-1) + 1 = 10.

10. Considere a multiplicação entre duas matrizes de tamanho 100x100 em um processador vetorial. Assim, pede-se o número de produtos escalares que haverá no referido sistema. Na multiplicação de duas matrizes 100x100, cada elemento da matriz-resultado é obtido por um produto escalar entre uma linha de A e uma coluna de B. Como a matriz-resultado tem 100x100 = 10000 elementos, haverá 10000 produtos escalares. Cada um desses produtos escalares envolve 100 multiplicações escalares, totalizando 1000000 de multiplicações escalares no processo.

11. Quantos ciclos de clock são necessários para processar um produto escalar, referente a um elemento de saída, correspondente à multiplicação de duas matrizes de tamanho 100x100 no pipeline vetorial dado no slide 10. Para um produto escalar de comprimento n num pipeline vetorial de 4 estágios (multiplicação) seguido de 4 estágios (adição) com encadeamento de resultados, cada elemento k entra no estágio M1 no ciclo k, gera a saída de multiplicação (M4) no ciclo k + 3, e inicia a adição (A1) no ciclo k + 4, terminando em A4 no ciclo k + 7. Logo, o último elemento (k = 100) conclui a soma em 100 + 7 = 107.

Assim, são necessários 107 ciclos de clock para obter o resultado do produto escalar referente a um elemento de saída na multiplicação de duas matrizes 100x100.

12. Mostre como fazer o intercalamento de endereços de memória de uma arranjo de 1024 palavras de dados em 4 bancos distintos. Para intercalar as 1024 palavras em 4 bancos, numeramos os bancos de 0 a 3 e usamos dois campos no endereço: os dois bits menos significativos determinam o banco e os bits restantes formam o deslocamento dentro de cada banco.

Assim, por exemplo:
endereço 0 → banco 0, offset 0
endereço 1 → banco 1, offset 0
endereço 2 → banco 2, offset 0
endereço 3 → banco 3, offset 0
endereço 4 → banco 0, offset 1

13. Qual é o tipo de hazard que pode ser evitado em arquiteturas vetoriais? Justifique. Em arquiteturas vetoriais, o hazard de controle pode ser praticamente eliminado. Num código escalar tradicional, cada iteração de um laço exige uma instrução de desvio (branch) e, portanto, toda vez que o processador encontra esse desvio ele precisa prever o resultado, possivelmente desperdiçando ciclos se a previsão falhar. Já numa operação vetorial, você emite uma única instrução que opera sobre todo o vetor de uma só vez: há apenas um desvio para entrar e sair da rotina vetorial, não um desvio por elemento. Com isso, elimina-se por completo o custo de previsões de desvio e o risco de penalidades de flush de pipeline causadas por branches dentro de laços longos.

14. Dê um exemplo de mascaramento para uma instrução IF utilizada no contexto de computação vetorial. Suponha que temos dois vetores X e Y e queremos calcular um vetor Z tal que, para cada elemento i, se X[i] > Y[i] então Z[i] = X[i] – Y[i], senão Z[i] = X[i] + Y[i]. Primeiro, geramos um vetor de máscara M onde M[i] = 1 se X[i] > Y[i], caso contrário M[i] = 0. Em seguida executamos duas

operações vetoriais completas, mas só atualizamos Z onde a máscara vale 1 numa, e onde vale 0 na outra:

1. M = CMPGT(X, Y)
2. V1 = SUB(X, Y)
3. V2 = ADD(X, Y)
4. Z = SELECT(M, V1, V2)

Dessa forma, a instrução IF por elemento é substituída por um compare vetorial que produz M, seguido de uma operação condicional por máscara que evita branches e processa todo o vetor em poucos ciclos.

15. Dê um exemplo de uso de Scatter-Gather no contexto de tratamento de matrizes esparsas. Considere uma matriz esparsa A armazenada em formato compacto, com um vetor val contendo os valores não-nulos, um vetor idx com os índices de linha/coluna correspondentes e um vetor x com os elementos de um vetor para multiplicação A * x. Gather é usado para buscar os valores de x correspondentes às posições não-nulas de A, e Scatter pode ser usado para acumular os resultados parciais no vetor resultado y.

```
para i = 0 ate nnz-1:
    xi = GATHER(x, idx[i])       ; busca x[idx[i]]
    pi = val[i] * xi             ; produto escalar parcial
    SCATTER_ADD(y, row[i], pi)   ; acumula pi em y[row[i]]
```