Sudoku em FPGA

Tiago Vallejo

***Resumo* – Este artigo descreve a implementação de solucionadores de puzzles Sudoku, desenvolvidos no âmbito dos sistemas reconfiguráveis. Deste modo, foram projetadas três arquiteturas: *Simples*, apenas capaz de resolver puzzles simples, *Tentativa e Erro*, que implementa um algoritmo de *Breadth-FirstSearch* para solucionar puzzles mais complexos, e, por fim,*Tentativa e Erro com Processamento Paralelo*. A implementação e teste foram realizadosnuma *FPGA* da família *Spartan-3E* da *Xilinx*, usando, para isso, uma placa de prototipagem da *Digilent*. Os resultados obtidos foram comparados entre as três implementações e outros solucionadores existentes. O projeto foi desenvolvido no âmbito da Dissertação de Mestrado "Sudoku em *FPGA*".**

***Abstract* – This paper describes the implementation ofa Sudoku solver, developed in the context of reconfigurable systems. Thus, three solvers were developed: *Simple*, only able to solve simple puzzles, *Trial and Error*, which implements a *Breadth-First Search* algorithm, being able to solve more complex puzzles, and, finally, *Trial and Error solver with the possibility of parallel processing*. For the implementation and test an *FPGA* of the *Xilinx Spartan-3E* family was used, using for the purpose a prototyping board from *Digilent*. The results were compared between the three implementations as well as with other state-of-the-art solvers. The project was developed within the Master Thesis "Sudoku em *FPGA*".**

I. INTRODUÇÃO

Sudoku é um puzzle quebra-cabeças baseado na colocação lógica dos algarismos. O puzzle foi projetado por Howard Garns, baseando-se no quadrado latino [1].

Esse puzzle consiste numa matriz 9x9, subdividida em 9 caixas (pequenas matrizes 3x3) onde são colocados números de 1 a 9. O objetivo é preencher as células vazias do puzzle, respeitando a seguinte regra: cada linha, coluna e caixa deverão conter todos os números de 1 a 9 sem repetição.

Além desta configuração mais comum, existem variações nas suas dimensões, podendo-se descrever um puzzle pela sua ordem N. Esta define o número de linhas e colunas existentes na matriz N^2xN^2, dividida em N^2 caixas. Assim, um puzzle de matriz 9x9 consiste num puzzle de ordem N = 3.

Recentemente, com a competição *Sudoku Solvers onFPGA* na *Conferência InternacionalField-Programmable Technology 2009* (*FPT'09*) [2], surgiu um maior interesse em criar solucionadores eficazes e capazes de aproveitar ao máximo o desempenho da *FPGA*.

II. TRABALHO RELACIONADO

Vários projetos de pesquisa, baseados em *FPGAs*, implementam algoritmos computacionalmente exigentes, como é o exemplo do Sudoku. De seguida são apresentadas algumas abordagens na implementação de solucionadores que concorreram na *FPT'09*.

O solucionador *TU Delft* [3] recorre a uma *FPGAXilinx Virtex2P-30* para a implementação de um algoritmo de força-bruta. Os resultados foram obtidos a uma frequência de 50 *MHz*, tendo resolvido com sucesso puzzles até ordem N = 8. Este método, apesar de realizável, não é aplicável a puzzles de maior complexidade.

Muito similar a este, o solucionador descrito em [4] recorre a algumas técnicas de redução do número de candidatos, juntamente com força-bruta. Os resultadosforam obtidos usando uma *FPGAVirtex-II Pro* da *Xilinx*, demonstrando uma grande eficiência na resolução de puzzles até ordem N = 11. Por forma a melhorar os resultados obtidos, uma possível otimização passará pela implementação de mais métodos heurísticos.

Um outro solucionador é apresentado em [5], consistindo este num solucionador probabilístico, cujo algoritmo se baseia num método designado *simulatedannealing*. Este solucionador foi implementado tanto em *software* como em *hardware*, usando para isso uma *FPGA Virtex-II Pro* da *Xilinx*. Os resultados foram obtidos à máxima frequência de operação, 53 *MHz*, tendo sido resolvidos puzzles de ordem N = 6 até N = 12. De notar que a *FPGA* foi mais rápida que o *software* para puzzles até à ordem N = 8.

Mais tarde, já fora do âmbito da competição, surgiu uma outra abordagem, sendo esta feita através de uma matriz de incidência *ECP* (*Exact Cover Problem*) [6]. Este algoritmo revelou-se bastante eficiente, resolvendo puzzles até ordem N = 14. Os resultados, obtidos com uma *FPGA Spartan-3E* da *Xilinx*, foram comparados com solucionadores *state-of-the-art*, implementados em *software*, tendo estes sido mais rápidos apenas nos puzzles mais complexos.

Neste projeto, a resolução de puzzles será feita através de um algoritmo de pesquisa, *Breadth-First Search*, juntamente com alguns métodos heurísticos, por forma a reduzir o número de candidatos por célula. Apenas serão considerados os puzzlesde ordem N = 3.

III. ALGORITMOS

Com base nas células do puzzle já preenchidas, é gerada a lista de possibilidades para as células vazias. Esta lista é constituída pelo conjunto de números passíveis de ser colocados em cada célula, respeitando as regras do puzzle. Através desta lista, é possível implementar um conjunto de métodos capazes de reduzir o número de candidatos ou mesmo solucionar o puzzle.

Os métodos implementados foram os seguintes:

- *Singles*: trata de averiguar qual ou quais as células que contêm um único candidato permitido. Se este for o caso, o candidato é preenchido de imediato, e o mesmo terá que ser retirado da lista de possibilidades da célula, como também das células da linha, coluna e caixa.
- *HiddenNumber*: averigua se um dado número surge apenas uma vez na lista de possibilidades das células de uma dada linha, coluna e caixa. Se assim for, este deverá ser preenchido na célula que o possui e removido de todas as listas de possibilidades da mesma linha, coluna e caixa.

Estes métodos, sendo apenas heurísticos, não garantem a solução do puzzle. Como tal, é necessário recorrer ao método de tentativa e erro. Este método, surge, então, quando nenhum dos outros métodos é capaz de preencher mais nenhuma célula ou reduzir o seu número de candidatos. Assim:

1. Encontra a célula com menor número possível de candidatos;
2. São efetuadas tantas cópias do puzzle quanto o número de candidatos existentes nessa célula, preenchendo cada uma com um candidato diferente;
3. Cada cópia é agora tratada como um puzzle distinto, ao qual é aplicado novamente estes três métodos, até que seja encontrada uma solução, ou se atinja uma situação de bloqueio (células livres sem candidatos).

Os métodos supracitados interagem entre si, formando o sistema solucionador de puzzles. Basicamente, foram desenvolvidos dois algoritmos: *Simples*, apenas capaz de resolver puzzles simples através dos dois métodos heurísticos supracitados, e *Tentativa e Erro*, que implementa um algoritmo de *Breadth-First Search* para solucionar puzzles mais complexos.

Os fluxogramas destes dois algoritmos podem ser observados nas Figuras 1 e 2.

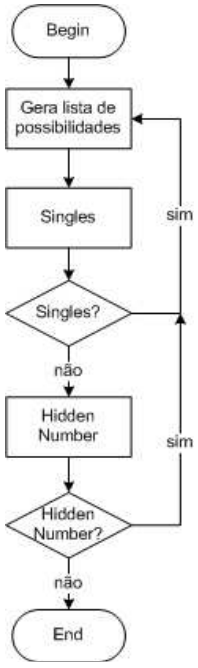

Fig. 1 – Fluxograma do algoritmo Simples.

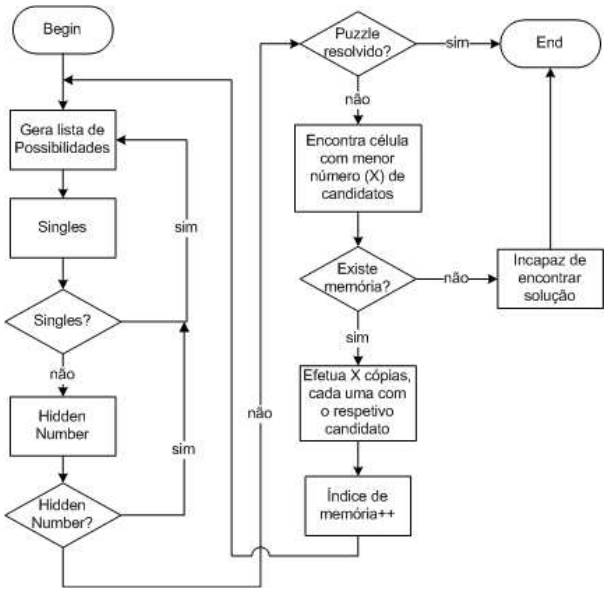

Fig. 2 – Fluxograma do algoritmo Tentativa e Erro.

IV. ARQUITETURA

Com base nos algoritmos das Figuras 1 e 2, foram implementadas 3 arquiteturas. A primeira, correspondente ao *Solucionador Simples*, que implementa o algoritmo da Figura 1 e, consequentemente, este só será capaz de resolver puzzles através dos dois métodos heurísticos já mencionados. O segundo solucionador, *Tentativa e Erro*, implementa o algoritmo da Figura 2, sendo este capaz de solucionar puzzles mais complexos, estando apenas limitado pela memória da *FPGA*. Por fim, o terceiro

solucionador implementa o algoritmo da Figura 2 com possibilidade de processar vários nós da árvore de pesquisa em paralelo, isto é, após efetuar X cópias do puzzle, todos estes X puzzles serão processados em paralelo. Por sua vez, se estes X puzzles derem origem a outras Y cópias, estas serão processadas em paralelo com os puzzles que já se encontram a ser processados.

A. *Organização dos dados em memória*

Os dados serão armazenados em blocos de memória do tipo *Single Port RAM*, mantendo a simplicidade das memórias usadas. Os seus barramentos, quer de endereços quer de dados, variarão de dimensão consoante a aplicação desta.

Na Tabela 1 podem ser observados os dados que são necessários guardar para a implementação dos solucionadores.

Tabela 1 – Dados necessários armazenar em memória.

Aplicação	Descrição
Puzzle(s)	Números constituintes do puzzle
Puzzle(s) Auxiliar	Cópias do puzzle necessárias aos solucionadores com tentativa e erro
Lista de Possibilidades	Candidatos de cada célula do puzzle
Mapa de Colunas	Identificadores de colunas
Mapa de Caixas	Identificadores de caixas
Mapa de Mínimos	Ordenação crescente das células do puzzle por número de candidatos
Mapa de Índices	Indica quais os puzzles (memórias) livres para os solucionadores com tentativa e erro

O puzzle é armazenado numa memória de 81 posições (9x9), com words de 4 bits. Uma vez que o puzzle auxiliar é usado para efetuar a salvaguarda dos valores do puzzle original, este é exatamente igual ao puzzle principal (Figura 3). Cada número é representado pelo seu equivalente em código binário, sendo que 0000_2 é reservado para indicar células vazias.

A lista de possibilidades é projetada para o pior caso, isto é, quando são passíveis de colocar 9 números numa dada célula. Deste modo, esta conterá 81 posições (9x9 células), tendo cada word 36 bits (9 números x 4 bits/número), tal como se pode observar na Figura 4.

Os mapas de colunas e caixas (Figura 5) consistem em matrizes 9x9 onde cada posição possui um identificador que corresponde ao número da coluna ou da caixa da célula indicada pelo barramento de endereços. Estas memórias são, também, de 81 words, de 4 bits cada.

O mapa de mínimos contém a ordenação crescente das células que possuem menor número de candidatos, bastante útil na implementação do algoritmo *Tentativa e Erro*, nomeadamente na identificação da célula com menor número de possibilidades. Esta memória encontra-se organizada em 81 *words* de 7 bits cada.

Por fim, o mapa de índices, responsável por indicar quais as memórias que se encontram livres para armazenar mais puzzles, apenas surge nos solucionadores com tentativa e erro. Esta memória é composta por 32 *words* de 1 bit cada, no entanto, esta é facilmente dimensionável através de um sinal auxiliar, consoante a memória que se encontra disponível na *FPGA*.

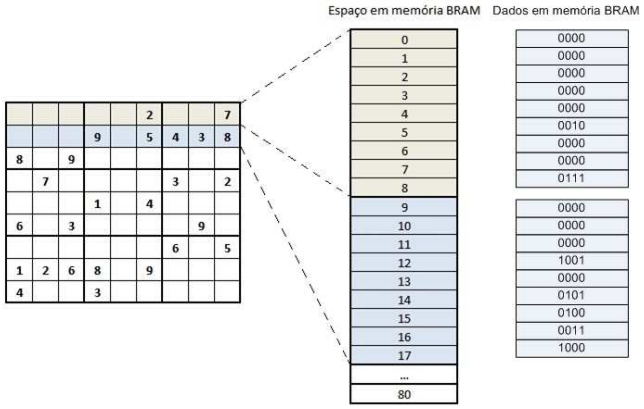

Fig. 3 – Armazenamento dos dados do puzzle em memória.

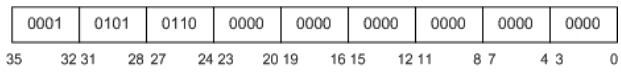

Fig. 4 – Estrutura da *word* da lista de possibilidades.

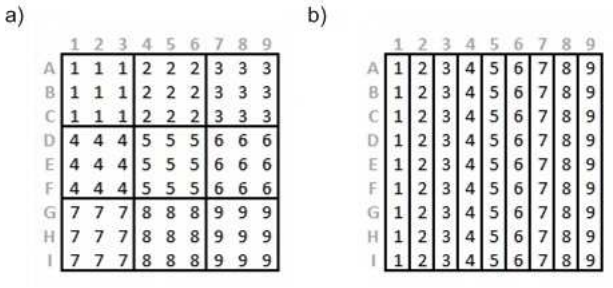

Fig. 5 – Mapas: a) mapa de caixas; b) mapa de colunas.

B. *Arquitetura dos solucionadores*

O diagrama de blocos *do Solucionador Simples* pode ser observado na Figura 6. Uma vez que este implementa o algoritmo *Simples* (Figura 1), o número de processos intervenientes é reduzido, sendo apenas o processo *Singles*, *HiddenNumber* e *Lista de Possibilidades*.

Este algoritmo encontra-se implementado na *Unidade de Controlo*, através da gestão dos sinais *enable*dos processos e acesso aos blocos de memórias (puzzle, lista de possibilidades, mapa de caixas e colunas). O acesso a estes blocos é feito através de *multiplexers* de endereços, dados e sinais *writeenable*, assim como também pelo codificador de prioridade, sendo, o acesso, disputado entre

uma interface externa (*USB*, por exemplo) e os três processos supracitados. Tanto os processos, como o algoritmo em si, foram implementados através de máquinas de estados finitos.

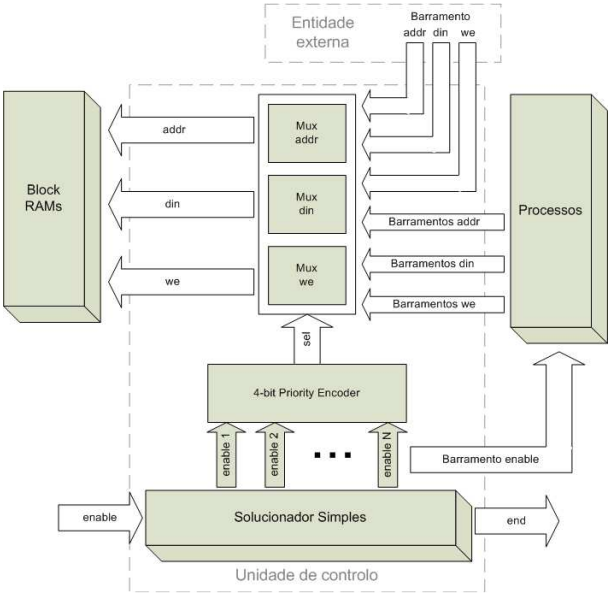

Figura 6 – Diagrama de blocos da *Unidade de Controlo* do *Solucionador Simples*.

O segundo solucionador, *Solucionador Tentativa e Erro* (Figura 7 e Figura 8) sofre algumas alterações relativamente ao *Solucionador Simples*, nomeadamente:

- A memória Puzzle passará a constituir um bloco de memórias iguais desmultiplexadas através do valor do índice atual (Figura 8);
- Adição do mapa de índices, contendo a informação das memórias livres/ocupadas;
- Reestruturação do *multiplexer* de endereços e codificador de prioridade, por forma a interagir com mais processos.

Além destas alterações, surgem novos processos como *Mínimos*, responsável por gerar o mapa de mínimos e o *Check Solution*, responsável por validar a solução do puzzle.

Como já referido, este solucionador aplica um algoritmo de *Breadth-First Search* quando nenhum dos processos heurísticos (*Singles* e *Hidden Number*) é capaz de preencher mais nenhuma célula. Assim, através do mapa de mínimos é possível determinar a célula com o menor número de candidatos, efetuando tantas cópias quanto o número de candidatos existentes nessa célula. Cada uma destas cópias é processada sequencialmente pelo solucionador, de acordo com o algoritmo apresentado na Figura 2, criando, eventualmente, novas cópias que serão processadas entretanto. Este processo termina quando o puzzle se encontrar completamente preenchido, sendo verificada a validade da solução, por parte do processo *Check Solution*. Se a solução for válida, o solucionador é interrompido; caso contrário, são processadas outras cópias efetuadas.

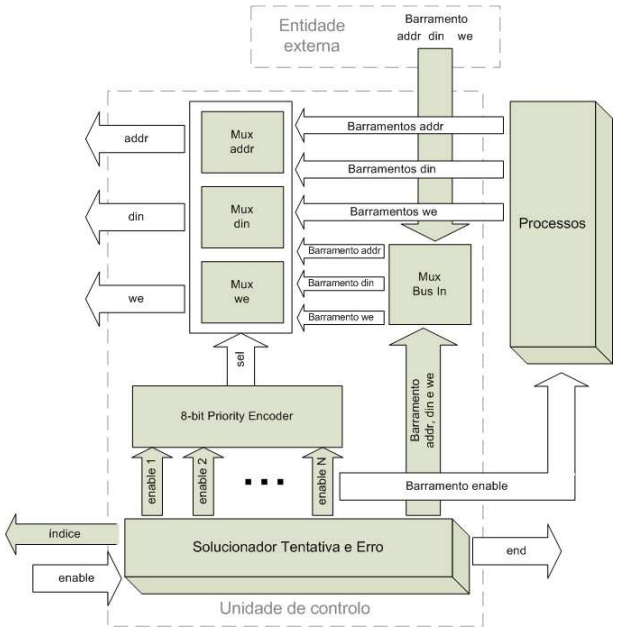

Figura 7 – Diagrama de blocos da *Unidade de Controlo* do *Solucionador Tentativa e Erro*.

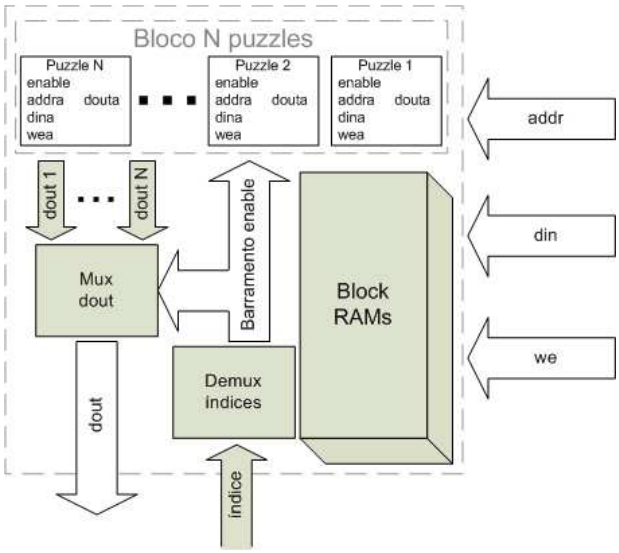

Figura 8 – Diagrama de blocos da memória do *Solucionador Tentativa e Erro*.

Com esta arquitetura é possível resolver um número considerável de puzzles, uma vez que existe memória suficiente para efetuar o número de cópias necessário aos puzzles mais comuns. O acesso a cada uma das cópias do puzzle é gerido pela *Unidade de Controlo*, que define o índice do puzzle a ser avaliado. Este índice atua como sinal de seleção dos *multiplexers* do bloco de memórias (Figura 8), pelo que, deste modo, os processos continuam

a aceder a este como se fosse um só puzzle e não um bloco.

A última implementação, constitui uma variante do *Solucionador Tentativa e Erro*, sendo este implementado com a possibilidade de processar puzzles em paralelo. Assim, a base de funcionamento é a mesma, isto é, baseado numa árvore de pesquisa *Breadth-First Search*.

Neste, ter-se-ão vários solucionadores responsáveis por executar os processos heurísticos *Singles* e *Hidden Number* e ainda criar o mapa de mínimos, sem que possuam tentativa a erro. A fase de tentativa e erro será entregue à *Unidade de Controlo* (Figura 9), sendo que esta surge quando um solucionador não consegue resolver o puzzle, indicando a posição da célula com menor número de candidatos possíveis.

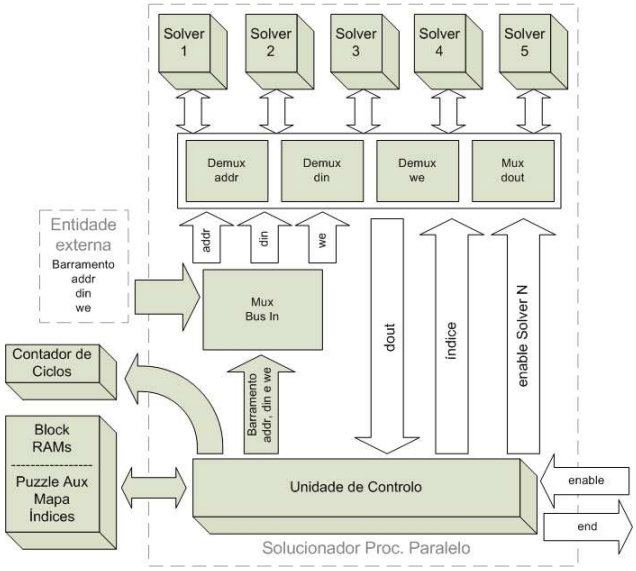

Figura 9 – Diagrama de blocos da *Unidade de Controlo* do *Solucionador Tentativa e Erro com Processamento Paralelo.*

A *Unidade de Controlo*efetua, assim, tantas cópias quanto o número de possibilidades para índices (solucionadores nesta implementação) livres, isto é, que não se encontrem a ser utilizados na resolução do puzzle. Após a cópia dos puzzles, ativa os respetivos solucionadores, dando inicio ao processo de resolução. Assim que um solucionador tenha chegado à solução, a *Unidade de Controlo* é notificada, terminando todos os outros solucionadores.

Com esta arquitetura é possível resolver um vasto número de puzzles, no entanto, estes não deverão necessitar de efetuar um número elevado de cópias do puzzle original, dado que grande parte da memória é usada na implementação dos solucionadores em paralelo (Figura 9). O acesso aos solucionadores é gerido, tal como anteriormente, pela *Unidade de Controlo*, através do índice do solucionador que atua como seleção dos *demultiplexers*.

V. RESULTADOS

Os resultados foram obtidos usando uma placa de prototipagem *Nexys 2*, da *Digilent*, que contém uma *FPGA XC3S1200E*. Esta *FPGA* possui 28 *BlockRAMs*embutidos, de 18 Kbit [7].

Cada entidade descrita na Tabela 1 é armazenada numa *Block RAM*, configuração esta que permite implementar facilmente o *Solucionador Simples*, ter disponíveis 21 índices no *Solucionador Tentativa e Erro* e, ainda, implementar 5 solucionadores em paralelo no *Solucionador Tentativa e Erro com Processamento Paralelo*.

Para efeitos de teste foram usados puzzles que necessitam de um máximo de 12 índices para ser resolvidos, sendo este valor suficiente para a obtenção de resultados.

Os recursos da *FPGA* ocupados por cada solucionador encontram-se descritos na Tabela 2. Como seria de esperar, à medida que se aumenta a complexidade do solucionador, aumentam-se, também, os recursos por este ocupados.

Tabela 2 – Recursos ocupados por cada um dos solucionadores

Recursos	Solucionadores		
	Simples	**Tentativa e Erro**	**Tentativa e Erro com Processamento Paralelo**
Slices	529 (6%)	961 (11%)	4297 (49%)
BlockRAMs	4 (14%)	19 (67%)	27 (96%)
$F_{máx}$ (MHz)	55.6	43.4	51.7

As Tabelas 3 e 4 apresentam os resultados obtidos para os puzzles que servem como instância de teste [8-10]. Os algoritmos das Figuras 1 e 2 foram, também, implementados em *software*, por forma a comparar o desempenho com a *FPGA*, usando para o efeito um PC *HP EliteBook 2730p*, com uma *CPU Intel Core Duo 1.87 GHz*.

Observando as Tabelas 3 e 4 constata-se que as implementações em *hardware* dos solucionadores *Simples* e *Tentativa e Erro* são menos eficientes quando comparadas com o seu equivalente em *software*. Estes resultados estão de acordo com os esperados, visto que estes solucionadores não exploram um elevado nível de paralelismo de operações.

Relativamente ao *Solucionador Tentativa e Erro com Processamento Paralelo*, os resultados obtidos em *hardware* são bastante próximos dos obtidos em *software*, evidenciando o desempenho da *FPGA* quando explorado o paralelismo de operações.

Note-se que estes testes foram efetuados numa *FPGAlow-cost*, que suporta frequências de operação muito limitadas (37 vezes mais lenta que o PC usado). Se o mesmo projecto for migrado para uma *FPGA* mais avançada dever-se-ão obter melhores resultados.

Tabela 3 – Tabela dos resultados do tempo de processamento para a implementação do *Solucionador Simples*

Puzzle	Software (ms)	Hardware (ms)
Easy Puzzle 1	0.93	1.43
Easy Puzzle 2	0.59	0.79
Easy Puzzle 3	0.77	1.20
Easy Puzzle 4	0.98	1.68
Medium Puzzle 1	2.04	3.92
Medium Puzzle 2	3.76	1.57
Medium Puzzle 3	1.73	2.03
Medium Puzzle 4	2.87	3.36

Tabela 4 – Tabela dos resultados do tempo de processamento para as implementações com tentativa e erro

Puzzle	X	Software (ms)	Hardware Sem Proc. Paralelo (ms)	Hardware Com Proc. Paralelo (ms)
Hard Puzzle 5	4	5.63	14.16	5.65
Hard Puzzle 9	4	6.94	20.60	5.42
Hard Puzzle 12	2	4.34	8.15	4.91
Expert Puzzle 1	3	3.99	6.68	4.14
Expert Puzzle 2	2	4.78	22.06	4.98
Extreme Puzzle 5	1	3.39	11.01	5.14
Evil Puzzle 1	1	5.53	22.14	9.57
Evil Puzzle 2	3	5.40	10.54	7.22
Extra Chall 13	1	4.92	18.86	7.25
Evil Puzzle 17	1	4.15	6.21	5.22
Evil Puzzle 5	2	5.22	9.25	5.63
Hard Puzzle 10	3	3.36	7.34	5.08
Expert Puzzle 10	1	2.75	15.21	3.96

Além destes, foram também obtidos resultados para o puzzle 3a das instâncias de teste usadas para a*FPT'09* [11]. O *Solucionador Simples* foi cinco vezes mais rápido que o solucionador *TU Delft* [3], duas vezes mais rápido que [4] mas mais lento que osolucionador [6]. O solucionador *Tentativa e Erro*, tal como se verificou, é bastante mais lento. Não obstante o seu resultado, este é satisfatório, tendo-se aproximado do resultado do solucionador [4], cuja implementação é bastante idêntica.

VI. CONCLUSÃO

Neste trabalho foram implementadas três arquitecturas diferentes para os solucionadores de puzzles Sudoku em *FPGA*.

Apesar dos resultados obtidos em *hardware* serem mais lentos, foi possível obter resultados muito próximos do *software* através da exploração do paralelismo de operações, mesmo usando uma*FPGAlow-cost*. Com isto conclui-se que este deve ser mais aprofundado, por forma a obter melhores resultados.

Como trabalho futuro propõe-se uma otimizaçao do uso das *BlockRAMs*, aproveitando melhor o seu espaço, podendo implementar um número superior de solucionadores em paralelo, assim como também tornar estes solucionadores capazes de resolver puzzles de ordem superior.

REFERÊNCIAS

[1] Wikipedia, Sudoku [Online]: http://pt.wikipedia.org/wiki/Sudoku.

[2] The 2009 International Conference on Field-Programmable Technology (FPT'09) [Online]: http://fpt09.cse.unsw.edu.au/.

[3] K. van der Bok, M. Taouil, P. Afratis, and I. Sourdis, "The TU Delft Sudoku Solver on FPGA", Proc. FPT'2009, pp. 526-529, Australia, 2009.

[4] C. González, J. Olivito, and J. Resano, "An Initial Specific Processor for Sudoku Solving", Proc. FPT'2009, pp. 530-533, Australia, 2009.

[5] P. Malakonakis, M. Smerdis, E. Sotiriades, and A. Dollas, "An FPGA-Based Sudoku Solver based on Simulated Annealing Methods", Proc. FPT'2009, pp. 522-525, Australia, 2009.

[6] M. Dittrich, T.B. Preußer, and R.G. Spallek, "Solving Sudokus through an Incidence Matrix on an FPGA", Proc. FPT'2010, pp. 465-469, Beijing, China, 2010. [7] Trenz Electronic GmbH, Products. [Online]: www.trenz-electronic.de/home/indexen.htm.

[7]Xilinx. (2009). Spartan-3E FPGA Family Datasheet. [Online]: www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[8] Web Sudoku benchmarks [Online]: http://www.websudoku.com/.

[9] Feenstra, M., Sudoku puzzles collection [Online]: http://www.sudoku.ws/.

[10] Extra Challenging (Very Hard) Sudoku Puzzles [Online]: http://puzzles.about.com/library/sudoku/blprsudokux04.htm.

[11] FPT'09. (2009). International Conference on Field-Programmable Technology - Design Competition - Benchmarks. Consultado a 25 Out, 2011: http://fpt09.cse.unsw.edu.au/comp/benchmarks.html.l