Universidade de Aveiro
2011
Departamento de Electrónica, Telecomunicações e Informática

Tiago Vallejo dos Santos

Sudoku em FPGA

Universidade de Aveiro
2011
Departamento de Electrónica, Telecomunicações e Informática

Tiago Vallejo dos Santos

Sudoku em FPGA

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica da Prof. Doutora Iouliia Skliarova, Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Dedico este trabalho aos meus pais e amigos.

o júri

presidente

Prof. Doutor Joaquim João Estrela Ribeiro Silvestre Madeira
Professor auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Profª. Doutora Iouliia Skliarova
Professora auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Prof. Doutor António José Duarte Araújo
Professor auxiliar do Departamento de Engenharia Electrotécnica e de Computadores da Faculdade de Engenharia da Universidade do Porto

agradecimentos

Aos meus pais, pelo apoio, suporte e toda a confiança depositada em mim ao longo destes 5 anos, tornando possível chegar aqui.

À minha irmã, Black, Serra e Rafa, por todos os fins-de-semana bem passados, permitindo abstrair e fugir da rotina universitária semanal.

A todos os meus amigos que me apoiaram ao longo destes anos, que acompanharam os bons e maus momentos e vice-versa, e pelos momentos de paródia e de estudo.

Por fim, mas não menos importante, a todos os professores desta minha casa, pelo conhecimento transmitido, em especial à Professora Iouliia Skliarova por esta oportunidade.

A todos vocês, que me fizeram crescer,

Muito Obrigado!

palavras-chave

Sistemas Reconfiguráveis, Sudoku, *Field Programmable Gate Array*, *Breadth-First Search*, Computação Paralela.

resumo

Este trabalho, desenvolvido no âmbito dos sistemas reconfiguráveis, tem como objetivo a implementação de um solucionador de puzzles Sudoku, quer em *software* quer em *hardware*, tentando minimizar o seu tempo de solução.
Deste modo, foram desenvolvidos três solucionadores: *Simples*, apenas capaz de resolver puzzles simples, *Tentativa e Erro*, que implementa um algoritmo de *Breadth-First Search* para solucionar puzzles mais complexos, e, por fim, o solucionador *Tentativa e Erro* com capacidade para processamento paralelo, também este capaz de solucionar puzzles mais complexos.
Todos estes solucionadores foram implementados e testados numa *FPGA* da família *Spartan-3E* da *Xilinx*, usando, para isso, uma placa de prototipagem da *Digilent*.
Os resultados obtidos foram comparados entre as várias implementações abordadas, assim como com outros solucionadores existentes.

keywords

Reconfigurable Systems, Sudoku, Field Programmable Gate Array, Breadth-First Search, Parallel Computing.

abstract

This work, developed in the context of reconfigurable systems, has as an objective the implementation of a Sudoku solver, both in software and hardware, and attempting to minimize its solution time.
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, the *Trial and Error solver with the possibility of parallel processing*, being also able to solve complex puzzles.
All these solvers were implemented and tested on an FPGA of Xilinx Spartan--3E family, using for this purpose a prototyping board from Digilent.
The results were compared between the various implementations, as well as with other state-of-the-art solvers.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

ADDR Address

ASIC Application Specific Integrated Circuit

BF Brute-force

BFS Breadth-first Search

CLB Configurable Logic Block

CPU Central Processing Unit

DEMUX Demultiplexer

DFS Depth-first Search

DIN Data In

DOUT Data Out

DSP Digital Signal Processor

ECP Exact Cover Problem

EOC End Of Count

EOCS End Of Check Solution

EOF End Of Filled

EOH End Of Hidden

EOM End Of Minimums

EONE End Of Not Empty

EOPL End Of Possibilities List

EOS End Of Singles

EOV End Of Validate

FIFO First In, First Out

FPGA Field Programmable Gate Array

FPT'09 Field-Programmable Technology 2009

FSM Finite State Machine

GPP General Purpose Processor

HDL Hardware Description Language

ID Identification

IOB Bounded Input/Output

LC Logic Cell

LFSR Linear Feedback Shift Register

LUT Lookup Table

MUX Multiplexer

PC Personal Computer

RAM Random Access Memory

SAT Satisfiability Problem

SRL Shift Register Lookup Table

USB Universal Serial Bus

VHDL Very High Speed Integrated Circuit Hardware Description Language

WE Write Enable

1 Introdução

1.1 Enquadramento

Antes da lógica programável, os circuitos lógicos eram construídos em placas de circuitos, utilizando componentes, ou pela integração de portas lógicas, em circuitos integrados para aplicações específicas [1].

Com o surgimento das *FPGAs*, (*Field-Programmable Gate Array*) em meados da década de 80, em conjunto com as linguagens de descrição de *hardware* (*HDL*), tornou-se possível desenvolver circuitos lógicos com maior facilidade e flexibilidade. Originalmente, estas eram usadas para verificar o projeto dos circuitos que, mais tarde, viriam a ser fabricados. Com o passar dos anos, porém, a sua configurabilidade, flexibilidade e capacidade computacional sofreram um enorme desenvolvimento [2, 3].

Hoje em dia, com o acompanhamento da *Lei de Moore*[1][4], as *FPGAs* conseguem competir com *ASICs* (*Application Specific Integrated Circuits*) em vários domínios de aplicação, como, por exemplo, processamento de imagem e vídeo, processamento de dados de satélites, compressão de imagem e vídeo, implementação de filtros ou mesmo interface de comunicação, etc…[5] Todas estas áreas constituem o domínio da computação de alto desempenho.

Estas características, juntamente com o preço e a flexibilidade, tornaram a utilização das *FPGAs* viável numa vasta gama de aplicações, constituindo uma boa alternativa aos sistemas baseados em microprocessadores [3].

1.2 Motivação

Neste contexto, o aumento da introdução de *FPGAs* em cada vez mais aplicações requer que as capacidades destas sejam muito bem aproveitadas, nomeadamente em aplicações exigentes e de alto desempenho.

Existem vários paradigmas para colocar as *FPGAs* a funcionar num ambiente computacional de alto desempenho. No entanto, a sua flexibilidade e capacidade tornaram-nas capazes de executar operações em paralelo em larga escala [6], capacidade esta fulcral

[1] O número de transístores num chip duplica a cada 18 a 24 meses [4].

para este projeto. A execução de operações em paralelo constitui a abordagem mais comum para o aumento da *performance* das *FPGAs*, rentabilizando o aproveitamento das suas capacidades quando comparadas com os processadores atuais.

Por forma a testar o desempenho destas, surgiu uma competição na *International Conference on Field-Programmable Technology*, em 2009, que consistiu no desenvolvimento de um solucionador de puzzles Sudoku baseado em *FPGAs*. Assim, cada equipa teve de implementar um solucionador, usando, para isso, a abordagem que considerasse mais adequada e, por fim, apresentar os resultados para os tempos de resolução dos puzzles, utilizando as instâncias de teste [7] disponibilizadas para o efeito.

Usar uma *FPGA* para resolver puzzles Sudoku constitui um desafio interessante e um teste ao seu desempenho na execução de algoritmos. O seu uso, para este propósito, tem crescido ao longo dos anos e provou-se ser mais eficiente em muitos dos casos. No entanto, as *FPGAs* são mais lentas que os *GPPs* (*General Purpose Processors*) relativamente à sua frequência de operação, pelo que o melhor desempenho é obtido através do paralelismo das operações e da capacidade de manter os dados locais, isto é, na própria *FPGA* [8].

1.3 Objetivos

O presente trabalho tem como objetivos os a seguir enunciados:

- Implementação dos métodos lógicos de resolução de puzzles Sudoku em *software* e *hardware*;

- Construção de uma rotina solucionadora capaz de resolver puzzles utilizando os métodos lógicos anteriores, tanto em *software* como em *hardware*;

- Estudo e implementação de um bloco solucionador capaz de gerir e processar os métodos anteriores em paralelo;

- Comparação de resultados de desempenho e recursos usados entre solucionadores e outros trabalhos realizados anteriormente.

1.4 Organização da tese

Para além do capítulo de introdução, esta dissertação estrutura-se em mais seis capítulos:

Capítulo 2 – Estado da arte – é constituído por uma breve introdução teórica sobre os sistemas digitais reconfiguráveis, abordando a descrição e evolução, ao longo dos anos, das *FPGAs*. Integra, ainda, uma introdução ao Sudoku, apresentando uma breve nota histórica e suas características. Paralelamente, são também abordados alguns métodos de resolução de puzzles e outras ferramentas necessárias à implementação destes métodos, tanto em *software* como em *hardware*. Por fim, são apresentados alguns estudos e trabalhos desenvolvidos, que exigem a implementação de algoritmos computacionalmente exigentes, como é o caso do Sudoku.

Capítulo 3 – Estrutura de dados – apresenta a discussão sobre que dados necessitam ser armazenados e a forma como serão guardados.

Capítulo 4 – Métodos lógicos de resolução – *software* – é constituído pela descrição das implementações em *software* dos métodos lógicos e ferramentas apresentadas no capítulo II. É descrita, também, a implementação do solucionador com e sem tentativa e erro.

Capítulo 5 – Métodos lógicos de resolução – *hardware* – destina-se à descrição das implementações em *hardware* dos métodos lógicos e ferramentas apresentadas no capítulo II. É, ainda, descrita a implementação do solucionador, com e sem tentativa e erro, e, ainda, com tentativa e erro implementado com processamento paralelo.

Capítulo 6 – Resultados – apresenta os resultados para cada uma das implementações dos solucionadores, em *software* e *hardware*, sendo também efetuada a comparação entre eles. Além disto, são também comparados os resultados obtidos com os de outros trabalhos realizados na mesma área.

Capítulo 7 – Conclusão – destina-se à apresentação das conclusões principais deste projeto, bem como as propostas para trabalho futuro e publicações.

2 Estado da arte

Neste capítulo, será feita uma breve introdução aos sistemas digitais reconfiguráveis, sendo, assim, abordada a descrição e evolução, ao longo dos anos, das *FPGAs*. Será aqui, também, introduzido o puzzle *Sudoku*, apresentando uma breve nota histórica e descrevendo os seus principais métodos de resolução.

Por fim, serão apresentadas algumas implementações de solucionadores, levadas a cabo por várias universidades, assim como os seus resultados e conclusões.

2.1 FPGAs

FPGA é um circuito integrado que contém um grande número (milhares) de blocos lógicos configuráveis. Estas unidades podem ser vistas como componentes padrão que são configurados independentemente e conetados entre si, através de uma matriz de trilhas condutoras e *switches* programáveis (Figura 2.1) [1].

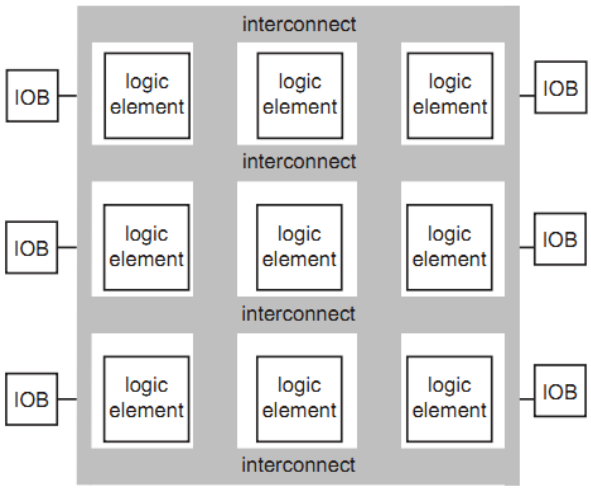


Figura 2.1 – Estrutura genérica de uma *FPGA*[9]

Por forma a configurar estas unidades lógicas, é gerado um arquivo binário, a partir de ferramentas de projeto assisto por computador, que contém a informação necessária para especificar a função de cada unidade lógica e para, seletivamente, fechar os *switches* da matriz. Este arquivo é então carregado no dispositivo, especificando assim a forma

como este funcionará. Em suma, a matriz de unidades lógicas e a matriz de conexão, que podem ser programadas e reprogramadas, formam a estrutura básica da *FPGA* [1].

Nas famílias da *Xilinx*, a menor unidade lógica configurável é denominada de *slice*. Esta é bastante versátil e pode ser configurada para operar como *Lookup Table* (*LUT*), *RAMs* (*Random Access Memory*) distribuídas e *Shift Registers* (Figura 2.2).

Na operação como *LUT,* podem ser utilizados recursos adicionais, como flip-flops tipo D, *multiplexers*, lógica de transporte (*carry*) dedicado e portas lógicas para implementação de funções booleanas, multiplicadores e somadores com palavras de comprimento bastante flexível. Na operação como *SRL* (*Shift Register LUT*), estes recursos adicionais podem ser usados para implementar contadores, conversores, entre outras funcionalidades [1]. O limite para o conjunto de funções e componentes implementados é determinado pelo número total de unidades lógicas disponíveis na *FPGA*, isto é, limitado pelos seus recursos.

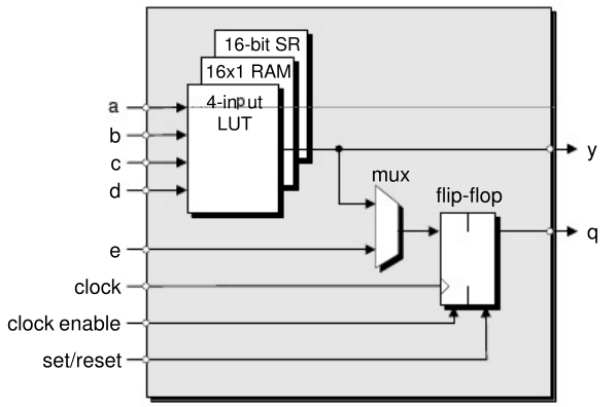


Figura 2.2 – Arquitetura interna simplificada de uma célula logica da família *Virtex-II* [10]

A programação da *FPGA* é feita recorrendo a um conjunto de ferramentas de projeto assistido por computador que possibilitam um certo grau de abstração, permitindo, ao projetista, até certo ponto, implementar o algoritmo desejado, em vez de se preocupar com os circuitos que serão implementados. Este projeto pode ser feito através de linguagens de descrição de *hardware* (*HDL*), como *VHDL* (*Very High Speed Integrated Circuit Hardware Description Language*) ou *Verilog*, modelação de sistemas (*System Generator*), etc.

Importa referir que *Field Programmable* significa que as funções da *FPGA* são definidas por um programa do utilizador e não pelo fabricante do dispositivo, como, por exemplo, em circuitos integrados típicos (*ASIC*), em que a implementação é realizada no processo de fabrico, tendo estes apenas uma única função, ao longo da sua vida, enquanto chip. Esta flexibilidade de programação, associada às potentes ferramentas de desenvolvimento e modelação, possibilita, aos utilizadores, o acesso ao projeto de circuitos integrados complexos sem os altos custos de engenharia associados aos *ASICs* [1].

2.2 Evolução das FPGAs

Ao longo de cerca de 40 anos da existência da *Lei de Moore*, esta acompanhou o desenvolvimento da lógica programável, tendo permitido aos fabricantes de *FPGAs* adicionar mais funcionalidades num único chip (Figura 2.3), aumentando a quantidade de lógica enquanto o seu custo continuava a diminuir (Tabela 2.1) [11].

Figura 2.3 – Aspeto de uma *FPGA*[12]

Desta forma, ao longo dos anos, as *FPGAs* têm sofrido uma grande evolução (Tabela 2.1), fazendo, agora, parte de um vasto número de sistemas embutidos.

Atualmente várias empresas dedicam-se ao fabrico de *FPGAs*, como, por exemplo, a *Xilinx*, *Altera*, *Lattice Semiconductor*, *Atmel*, *QuickLogic*, etc…[13] A *Xilinx* e *Altera* são hoje os líderes do mercado, controlando 89% deste, sendo que 50% pertencem apenas à *Xilinx* [13] (Figura 2.4). Como a *Xilinx* é líder deste sector e também a empresa responsável pela *FPGA* contida na placa de prototipagem usada neste projeto, ser-lhe-á dada maior ênfase.

Tabela 2.1 – Evolução do número de transístores nas *FPGAs* [14-16]

FPGA	Transístores	Ano	Fabricante
Virtex	~70 000 000	1997	Xilinx
Virtex-E	~200 000 000	1998	Xilinx
Virtex-II	~350 000 000	2000	Xilinx
Virtex-II PRO	~430 000 000	2002	Xilinx
Virtex-4	1 000 000 000	2004	Xilinx
Virtex-5	1 100 000 000	2006	Xilinx
Stratix IV	2 500 000 000	2008	Altera
Virtex-6	2 500 000 000	2009	Xilinx
Stratix V	3 800 000 000	2011	Altera
Virtex-7	6 800 000 000	2011	Xilinx

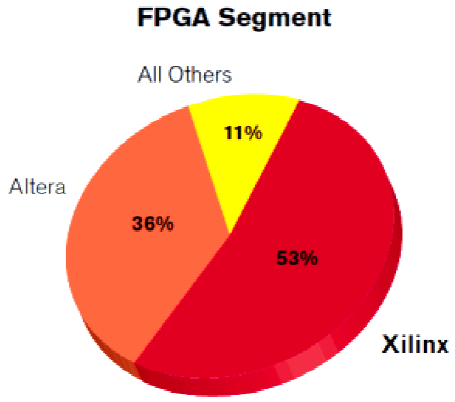


Figura 2.4 – Percentagem de mercado de cada fabricante, 2010 [17]

A primeira *FPGA* foi introduzida pela *Xilinx* em 1985. Esta continha 64 blocos lógicos e 58 entradas/saídas (*IOBs*). Hoje em dia, as *FPGAs* contêm entre 150 000 e 2 000 000 células lógicas e cerca de 600 *IOBs* [18] em adição ao vasto número de blocos especializados que expandiram as suas capacidades (Figura 2.5). No entanto, este aumento de capacidade foi acompanhado por mudanças críticas na arquitetura das mesmas, tendo

assim sido adicionados *DSPs* (*Digital Signal Processor*), *Tranceivers*, interfaces *PCI Express*, etc.

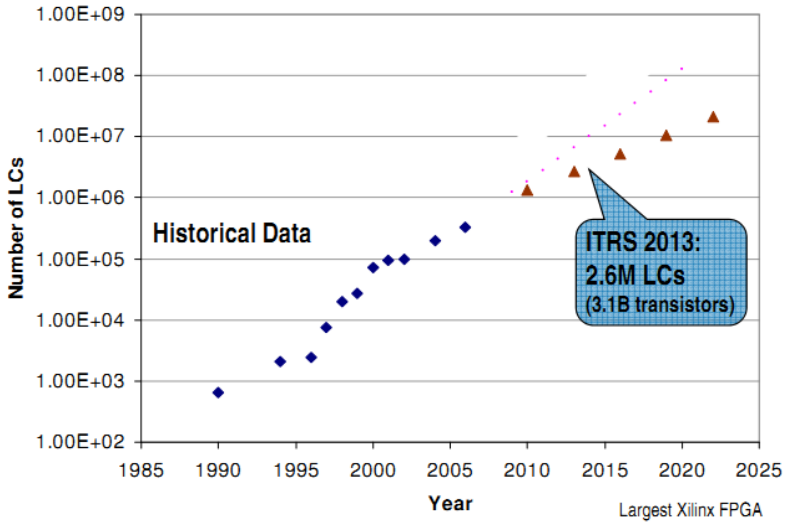


Figura 2.5 – Evolução do número de células lógicas nas *FPGAs* da *Xilinx* [19]

Na atualidade, a *Xilinx* possui três grandes famílias: *Virtex*, que visa o alto desempenho, *Artix*, mais económica mas também com bastantes capacidades e *Kintex*, intermédia que apresenta uma boa relação custo - *performance*. Estas correspondem às famílias da série 7 da *Xilinx*.

Na Figura 2.6 podem ser observadas as características principais de algumas *FPGAs* de famílias de séries anteriores, como a *Virtex* e *Spartan*.

Pode assim concluir-se que estes dispositivos lógicos programáveis são hoje capazes de suportar sistemas lógicos de grande capacidade, permitindo implementar algoritmos e sistemas de elevada complexidade e exigência com apenas uma *FPGA*. Além da elevada capacidade e flexibilidade, o seu preço tornou-se cada vez mais acessível, sendo estas economicamente viáveis para inúmeras aplicações.

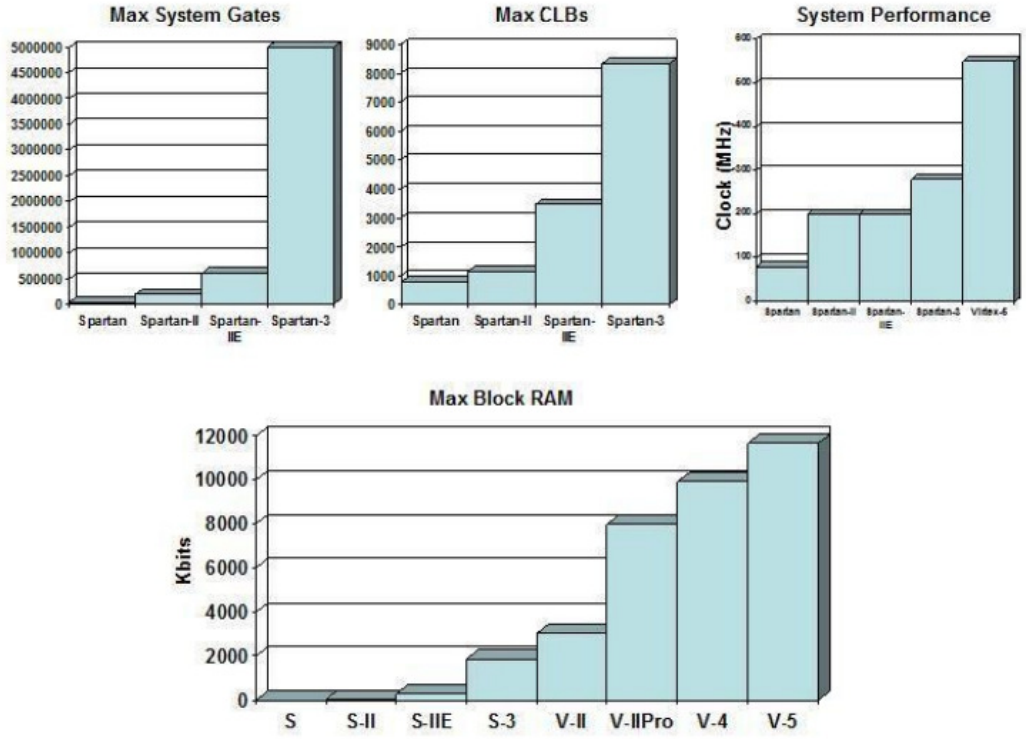


Figura 2.6 – Características principais de algumas *FPGAs* da *Xilinx* [20]

2.3 Sudoku

2.3.1 Introdução

Sudoku é um puzzle quebra-cabeças baseado na colocação lógica de números. O puzzle foi concebido por Howard Garns, baseando-se no quadrado latino, uma construção matemática criada pelo suíço Leonhard Euler no século XVIII. Garns apresentou uma variação do mesmo, que consiste no puzzle Sudoku tal como o conhecemos hoje em dia [21].

As primeiras publicações do puzzle ocorreram nos Estados Unidos, no final da década de 70, na revista norte-americana *Math Puzzle and Logic Problems*, da editora *Dell Magazines*. A editora deu ao jogo o nome de *Number Place*, que é a designação ainda hoje usada nos Estados Unidos.

Em 1984, a *Nikoli*, maior empresa japonesa de quebra-cabeças, descobriu o jogo e decidiu levá-lo até ao seu país. O nome Sudoku surgiu então, sendo este a abreviatura da

frase "*suuji wa dokushin ni kagiru"* que significa "os dígitos devem permanecer únicos" [21].

Apesar de toda a popularidade no Japão, o Sudoku não atraiu a mesma atenção no Ocidente até finais de 2004, quando Wayne Gould viajou a Londres para convencer os editores do jornal *The Times* a publicar o Sudoku. Gould tinha desenvolvido um programa de computador que gerava os puzzles com vários níveis de dificuldade e não estava a cobrar nada por ele. Em Portugal, o Sudoku começou a ser publicado em Maio de 2005 pelo jornal Público [21].

Este puzzle consiste numa matriz 9x9, composta por 9 matrizes 3x3, onde serão colocados números de 1 a 9. O objetivo é preencher as células vazias (Figura 2.7) 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 do puzzle da seguinte forma: N^2 x N^2. Assim, um puzzle de matriz 9x9 consiste num puzzle de ordem N = 3.

7				4			8	9
5		1			8			
8				6	1	7	5	
3	5					9	7	
		7	8			6		
	1		7	3				5
	8		4	9		2		
	3	9			7			6
		2	6				9	3

Figura 2.7 – Exemplo da matriz de um puzzle Sudoku [22]

Por forma a preencher corretamente o puzzle, existem vários métodos lógicos de resolução. Estas metodologias podem ser de dois tipos: eliminação do candidato, onde o progresso é feito através de sucessivas eliminações do número de candidatos de uma ou mais células para deixar apenas uma opção, ou tentativa e erro, onde, para uma dada célula com um número reduzido de candidatos, é feita uma suposição quanto ao seu valor [23].

2.3.2 Métodos lógicos de resolução

2.3.2.1 Singles

De todos os métodos lógicos de solução de puzzles Sudoku, este é talvez o mais básico. Esta técnica, *Singles*, ou candidato único, é assim chamada uma vez que trata de averiguar qual ou quais as células que contêm somente um candidato permitido [24]. Por forma a ilustrar este processo, observe-se o exemplo da Figura 2.8, que consiste numa caixa de um puzzle de ordem N = 3.

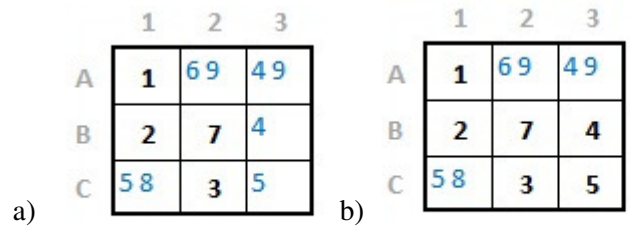


Figura 2.8 a) e b) – Exemplo ilustrativo de candidatos únicos, *Singles* [25]

Como se pode observar, a célula C3, Figura 2.8a, assim como a célula imediatamente acima desta, B3, possuem um só candidato. Esta situação resulta da avaliação dos candidatos possíveis para cada célula, tendo em conta os valores já preenchidos nas células do puzzle. Desta forma, estas deverão ser preenchidas com o valor do seu respetivo candidato (Figura 2.8b).

Algumas células poderão possuir estes candidatos únicos desde o início de resolução do puzzle. No entanto, a situação mais comum é existirem múltiplos candidatos por célula, sendo que estes irão sendo reduzidos à medida que outros métodos lógicos de resolução são aplicados. Como exemplo, note-se que, aquando do preenchimento das duas células supracitadas com os seus respetivos candidatos (valor 4 e 5), as células C1 e A3 ficarão reduzidas a um só candidato, tal como ilustrado na Figura 2.9a.

a)

	1	2	3
A	1	6 9	9
B	2	7	4
C	8	3	5

b)

	1	2	3
A	1	6	9
B	2	7	4
C	8	3	5

Figura 2.9 a) e b) – Exemplo ilustrativo do processo iterativo de candidatos únicos, *Singles* [25]

Assim, a célula A2 ficará também reduzida a um candidato apenas (Figura 2.9b). Através deste processo iterativo, foi possível obter esta caixa corretamente preenchida.

2.3.2.2 Hidden Number

Esta técnica, também designada de *Hidden Single* ou candidato único não direto, apesar de um pouco mais complexa, consiste também num método lógico de resolução básico e é igualmente de fácil compreensão.

Como sabemos, se existir apenas um só candidato para uma dada célula, então esta deverá assumir o seu valor (ver secção 2.3.2.1). No entanto, mesmo existindo mais do que um candidato para uma dada célula, nalguns casos é possível facilmente determinar o valor que esta deverá assumir, bastando que, para isso, o candidato surja uma e uma só vez na linha, coluna ou caixa em que essa célula se encontra [26]. Isto ocorre uma vez que cada linha, coluna e caixa deverão conter números de 1 a 9 sem repetição.

Para uma melhor compreensão, tome-se o exemplo da caixa representada na Figura 2.10.

	1	2	3
A	4	7	1 5 9
B	3	8	1 5 6 9
C	2	1 9	1 5 9

Figura 2.10 – Exemplo ilustrativo do processo iterativo de candidatos únicos não diretos, *Hidden Number* [24]

Como se pode observar, e tendo em conta todos os candidatos para todas as células livres do exemplo da Figura 2.10, o número 6, residente na célula B3, surge apenas uma e uma só vez nesta caixa. Significa isto que a célula B3, apesar de possuir vários candidatos, deve assumir o valor 6 (Figura 2.11). Por outras palavras, se o valor 6 não for atribuído à célula B3, mais nenhuma célula da matriz poderá assumi-lo e esta ficará incompleta.

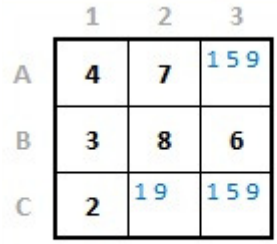


Figura 2.11 – Exemplo ilustrativo do processo iterativo de candidatos únicos não diretos, *Hidden Number*, para uma caixa

Repare-se agora o exemplo da Figura 2.12, desta vez para uma linha. Observando a linha B, constata-se rapidamente que o número 7 surge uma só vez. Assim sendo, a célula B8 deverá assumir o valor 7. O mesmo raciocínio deverá ser aplicado para as colunas constituintes do puzzle.

Apesar de simples, na prática, esta técnica não é imediata, uma vez que, para cada linha, coluna e caixa, é necessário efetuar a contagem de cada um dos nove números, por forma a determinar qual ou quais surgem uma e uma só vez. No entanto, a maioria dos puzzles consegue ser resolvida com recurso a esta técnica e à técnica *Singles*.

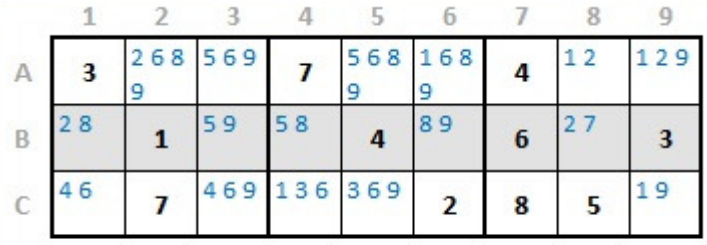


Figura 2.12 – Exemplo ilustrativo do processo iterativo de candidatos únicos não diretos, *Hidden Number*, para uma linha [27]

2.3.2.3 Tentativa e erro

Com os dois métodos anteriores, é possível resolver muitos puzzles, todavia, se o grau de dificuldade for elevado, é necessário recorrer ao método de tentativa e erro frequentemente.

Este método de resolução surge quando nenhum dos outros métodos é capaz de preencher mais nenhuma célula ou reduzir o seu número de candidatos. Nesta técnica, é então escolhida uma célula, de preferência com o menor número possível de candidatos, e, com base nestes, é feita uma suposição quanto ao seu valor (Figura 2.13).

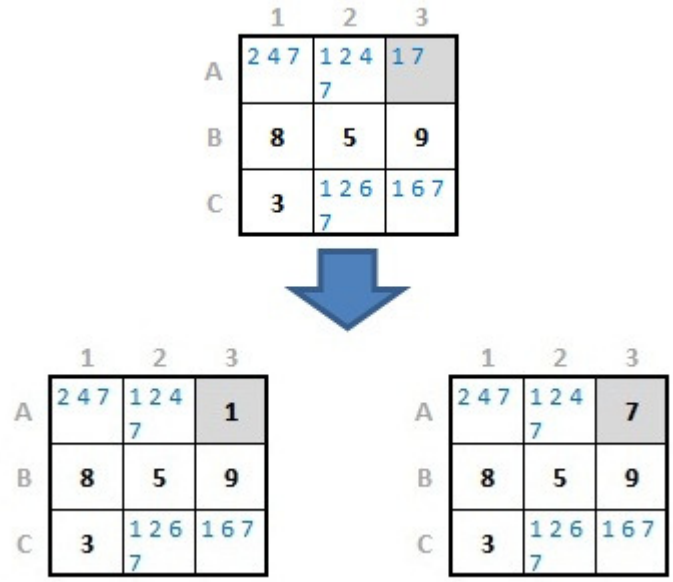


Figura 2.13 – Exemplo ilustrativo do método de tentativa e erro

De acordo com a Figura 2.13, são assim gerados dois puzzles com duas soluções distintas, com base na célula A3, aos quais deverão ser aplicados os métodos já apresentados, por forma a determinar qual a solução correta e descartar a solução errada. De notar que este método poderá ser usado quantas vezes necessário até a solução final ser atingida.

2.3.2.4 Outros métodos necessários

a) Validate

Este método é talvez a peça mais importante em todo o processo de resolução de puzzles *Sudoku*. Consiste a mesma na verificação do correto preenchimento do puzzle, isto é, permite verificar se uma dada linha, coluna e caixa não apresentam números repetidos e, portanto, se, até um dado momento, o preenchimento se encontra de acordo com as regras do puzzle.

Como exemplo de uma situação de preenchimento incorreto de uma célula, observe-se a Figura 2.14.

	1	2	3	4	5	6	7	8	9
A						2			7
B				9		5	4	3	8
C	8		9				9		
D		7					3		2
E				1		4			
F	6		3					9	
G							6		5
H	1	2	6	8		9			
I	4			3					

Figura 2.14 – Exemplo do puzzle Sudoku mal preenchido [28]

De notar que, na linha C, se repete o número 9 duas vezes (posições C3 e C7), pelo que a principal regra do puzzle é assim violada. Deste modo, este puzzle nunca será corretamente preenchido, ou seja, a sua solução não será uma solução válida. Assim, esta função apresenta duas importantes funcionalidades:

1. Permite verificar se os valores preenchidos para uma dada célula são válidos, isto é, se o seu preenchimento se encontra de acordo com as regras do puzzle;
2. Permite verificar quais os valores válidos para uma dada célula, que será útil na fase de preenchimento da lista de possibilidades/candidatos para cada célula.

b) Lista de possibilidades ou candidatos

A lista de possibilidades, também chamada de lista de candidatos, é de extrema importância quando se está perante um puzzle Sudoku. Através desta, é possível rapidamente saber quais os números passíveis de serem colocados nas mais diversas células constituintes do puzzle.

Assim, a cada célula está associada uma lista de números (que varia de célula para célula), e que contém os números que podem ser colocados em cada uma das células de acordo com as regras do puzzle Sudoku. Excetuam-se células já preenchidas, uma vez que já não necessitam de candidatos.

Por observação do exemplo da Figura 2.15, nota-se que a primeira célula do puzzle apenas contém dois candidatos: o número 5 e o número 3. Isto ocorre uma vez que o número 1, 4, 6 e 8 já existem na mesma coluna, o número 2 e 7 na mesma linha, e o número 9 na mesma caixa (tal como novamente o 8). Assim sendo, sobram apenas os dois candidatos presentes na lista de possibilidades para a primeira célula.

3 5	1 3 4 5 6	1 4 5	4 6	1 3 4 6 8	2	1 5 9	1 5 6	7
2 7	1 6	1 2 7	9	1 7 6	5	4	3	8
8	1 3 4 5 6	9	4 6 7	1 3 4 6 7	1 3 6 7	1 2 5	1 2 5 6	1 6
5 9	7	1 4 5 8	5 6	5 6 8 9	6 8	3	1 4 5 6 8	2
2 5 9	5 8 9	2 5 8	1	2 3 5 6 7 8 9	4	5 7 8	5 6 7 8	6
6	1 4 5 8	3	2 5 7	2 5 7 8	7 8	1 5 7 8	9	1 4
3 7 9	3 8 9	7 8	2 4 7	1 2 4 7	1 7	6	1 2 4 7 8	5
1	2	6	8	4 5 7	9	7	4 7	3 4
4	5 8 9	5 7 8	3	1 2 5 6 7	1 6 7	1 2 7 8 9	1 2 7 8	1 9

Figura 2.15 – Puzzle Sudoku e sua respetiva lista de possibilidades [29]

c) Mínimos

Esta rotina é responsável por gerar o mapa de mínimos do puzzle. Este mapa será uma matriz 9x9 que conterá a ordem crescente das células que possuem menor número de possibilidades. Para melhor perceber o funcionamento deste processo, observe-se a Figura 2.16.

	1	2	3	4	5	6	7	8	9
A	53	13456	145	46	13486	**2**	59	56	**7**
B	27	16	127	**9**	176	**5**	**4**	**3**	**8**
C	**8**	3456	**9**	467	3467	367	25	256	**1**
D	59	**7**	1458	56	689	68	**3**	158	**2**
E	259	589	258	**1**	23789	**4**	58	578	**6**
F	**6**	158	**3**	257	278	78	158	**9**	**4**
G	379	389	78	247	1247	17	**6**	128	**5**
H	**1**	**2**	**6**	**8**	**5**	**9**	**7**	**4**	**3**
I	**4**	58	578	**3**	1267	167	128	128	**9**

Figura 2.16 – Possibilidades de um puzzle Sudoku

Por observação da Figura 2.16 constata-se que algumas células possuem 2 candidatos, 3 candidatos, etc. No entanto, como veremos, aquando da criação do mapa de mínimos, não existe nenhuma célula com uma só possibilidade, visto que estas foram já preenchidas pela rotina *Singles*. Assim, as células com 2 candidatos serão as primeiras a ser preenchidas no mapa de mínimos, com 3 candidatos as segundas, e todas as outras depois. Observe-se então o mapa de mínimos para o puzzle apresentado na Figura 2.17.

De acordo com o mapa, pode-se observar que as células com 2 candidatos apenas são as primeiras do mapa a serem preenchidas. Note-se, por exemplo, a célula A1 que, sendo esta a primeira célula com 2 candidatos, será aquela preenchida com o primeiro índice do mapa. A célula A4 é a segunda célula a possuir apenas 2 candidatos e, portanto, a esta

corresponde o segundo índice do mapa. Após este raciocínio, tem-se então o mapa resultante apresentado na Figura 2.17, que será bastante útil na fase de tentativa e erro.

	1	2	3	4	5	6	7	8	9
A	1	44	16	2	45		3	4	
B	5	6	17		18				
C		40		19	46	20	7	21	
D	8		41	9	22	10		23	
E	24	25	26		47		11	27	
F		28		29	30	12	31		
G	32	33	13	34	42	14		35	
H									
I		15	36		43	37	38	39	

Figura 2.17 – Mapa de mínimos do puzzle apresentado na Figura 2.16

d) Filled

Esta função é responsável por verificar se o puzzle se encontra totalmente preenchido ou não. É bastante útil na fase de tentativa e erro, pois determina se o processo de tentativa e erro terminou ou se necessita de continuar. A sua implementação é bastante simples e rápida, quer em *software* quer em *hardware*.

e) Not Empty

Esta função é responsável por verificar se o puzzle não se encontra vazio. É bastante útil na fase de tentativa e erro, tal como iremos ver, uma vez que permite verificar se um dado puzzle apresenta células preenchidas, e, portanto, se é passível de se aplicar o algoritmo de tentativa e erro.

f) Check Solution

Tal como o próprio nome indica, esta função é responsável por verificar se a solução apresentada para o puzzle é válida. Este processo é deveras importante, nomeadamente para a fase de tentativa e erro.

2.4 Revisão do estado da arte

Apesar das *FPGAs* terem vindo a revelar um grande desempenho num vasto número de aplicações, os *CPUs* recentes apresentam também um grande potencial, nomeadamente para aplicações computacionalmente exigentes. Assim, a alta *performance* das *FPGAs* advém da sua flexibilidade, que torna possível realizar circuitos otimizados para cada aplicação, suportando um grande paralelismo de operações [30]. Por este motivo, estas conseguem atingir desempenhos elevados, apesar da sua baixa frequência de operação, pelo que a solução para aumentar esse desempenho em algoritmos exigentes é através da exploração do paralelismo das operações [31].

A resolução de puzzles Sudoku, recorrendo a uma *FPGA,* constitui um desafio no que concerne à sua implementação e ao seu desempenho, uma vez que a resolução deste recorre a um algoritmo computacionalmente exigente. Deste modo, existem várias abordagens para o projeto destes solucionadores em *hardware*.

De seguida, são abordadas algumas implementações apresentadas para a competição de *Sudoku Solvers on FPGA* na Conferência Internacional *Field-Programmable Technology 2009 (FPT'09)* [32]. Para esta, foram apresentados puzzles de dois tipos (A e B) para teste dos solucionadores. Estes puzzles variam quer nas duas dimensões quer no grau de dificuldade, sendo os puzzles do tipo B, equivalentes em termos de dimensão aos do tipo A, mais complexos. O tempo de execução é medido após a receção do último *byte* de dados do puzzle via *RS-232* e termina com o preenchimento do último *byte* de dados da solução.

2.4.1 Solucionador TU Delft

O solucionador *TU Delft* [8], desenvolvido na *Delft University of Technology,* na Holanda, consiste num solucionador de puzzle Sudoku que recorre a uma *FPGA* para implementação de um algoritmo de *força-bruta*[2]. Desta forma, de acordo com o conjunto de números passíveis de serem colocados em cada célula, é especulado o valor desta. As células são preenchidas até que ocorra um conflito no preenchimento do puzzle, como, por exemplo, a existência de células livres sem candidatos a ela associados, ou o puzzle seja resolvido. Na situação de conflito, retorna-se à última célula preenchida, alterando o valor especulado para a mesma por um outro possível.

Os resultados foram obtidos (Tabela 2.2, Tabela 2.5) usando a *FPGA Virtex2P-30* da *Xilinx* e uma frequência de operação de 50 *MHz*.

Tabela 2.2 – Utilização de recursos da *FPGA Virtex2P-30* para o solucionador *TU Delft* [8]

Recursos	Usados	Utilização
Slices	2 436	17%
Block RAMs	110	80%

Os puzzles usados para efeito de teste foram os disponibilizados como instância de teste na *FPT'09* [7], tendo sido resolvidos os puzzles do Tipo A de ordem N = 3 até N = 8, e o puzzle Tipo B de ordem N = 3. Os melhores resultados foram obtidos para os puzzles de ordem N = 3, tal como se pode observar na Tabela 2.5. Para os restantes puzzles, não foram obtidos os seus tempos de execução.

Como conclusão, o algoritmo *força-bruta* consiste num método realizável e capaz, todavia para puzzles de maior complexidade não é aplicável. Uma possível solução para a otimização deste algoritmo passará por implementar métodos de eliminação de candidatos, sendo estes usados juntamente com *força-bruta*.

[2] Em ciência da computação, força bruta (ou busca exaustiva) é um algoritmo trivial, mas com muito uso, que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema.

2.4.2 Processador específico para resolver Sudoku

Este solucionador de puzzles *Sudoku* [33], desenvolvido nas Universidades de Madrid e de Zaragoza, em Espanha, recorre a algumas técnicas de redução do número de candidatos (como, por exemplo, *Singles*, *Hidden Singles*, *Hidden Pairs*, etc...[23]) usadas juntamente com *força-bruta*, surgindo esta como complemento às técnicas de redução de candidatos, pois, sendo estas heurísticas, não garantem que o puzzle seja resolvido.

Desta forma, são preenchidas as células com os valores dos candidatos possíveis para cada uma até se atingir uma situação em que não existe nenhum candidato para uma célula livre, sendo necessário regressar à última célula preenchida e assumir outro valor para a mesma. É, assim, gerada uma árvore de pesquisa do tipo *DFS* (*Depth-First Search*) [34], sendo analisado cada ramo até detetar um conflito ou até o ramo produzir a solução do puzzle.

Os resultados foram obtidos usando uma *FPGA XC2VP30 Virtex-II Pro* da *Xilinx,* incluída na placa *XUP Development System* [33].

Na Tabela 2.3, podem ser observados os recursos ocupados na *FPGA* à medida que se foram adicionando novos métodos de resolução. De notar que os novos métodos de resolução exigem um número superior de *Block RAMs*, diminuindo a ordem máxima dos puzzles possíveis de resolver.

Os resultados para os tempos de execução de cada puzzle podem ser observados na Tabela 2.5. Os puzzles usados foram os disponibilizados como instância de teste na *FPT'09* [7].

Tabela 2.3 – Tabela de recursos usados com a adição de novos métodos de resolução [33]

Técnicas	Ordem Máxima (N)	Block RAMs	Slices e percentagem	
Branch&Bound	15	29	1 832	13%
Singles	15	29	2 183	16%
Hidden Singles	11	134	7 675	56%
Hidden Pairs	11	134	9 187	67%
Hidden Triplets	11	134	1 080	78%
Hidden Quartets	11	134	12 265	90%

Os resultados demonstraram uma grande eficiência na resolução de puzzles do Tipo A, tendo sido resolvidos todos os puzzles deste tipo até ordem N = 11. No entanto, no que

diz respeito aos puzzles do Tipo B, este algoritmo revela-se ineficiente, tendo sido apenas resolvido o puzzle de ordem N = 3, devendo-se este facto à estratégia *Branch&Bound* (*força-bruta*), que se torna extensa na procura da solução de puzzles complexos. Uma possível otimização passará pela implementação de mais métodos heurísticos, por forma a reduzir o número de possibilidades por célula, que se traduzirá num melhor desempenho da estratégia supracitada. Além desta, o uso de memória externa, por forma a lidar com puzzles de ordem superior a 12, constitui também uma otimização para este solucionador [33].

2.4.3 Solucionador SSAS

O *SSAS* (*Sudoku Simulated Annealing Solver*) [35], desenvolvido na Universidade de Creta, na Grécia, consiste num solucionador probabilístico, cujo algoritmo se baseia num método designado *Simulated Annealing* [36], em vez de árvores de pesquisa como, por exemplo, *DFS* ou *BFS* (*Breadth-First Search*), por forma a superar a exigência combinatória destas últimas. No entanto, este não garante que a solução seja atingida.

Esta abordagem baseia-se na perturbação do sistema no seu estado inicial, de modo a minimizar o tempo de execução até encontrar a solução, não sendo esta garantida. Assim, o algoritmo é constituído por três estados de resolução:

- **Estado 1 –** Consiste na inicialização das células vazias. Durante o mesmo, todos os blocos 3x3 são preenchidos com números de 1 a 9, inicializando assim cada bloco. Desta forma, cada um destes contém os dados corretos, ou seja, não existe repetição de células em cada bloco nem células vazias. Obviamente que as células já preenchidas no puzzle original não são alteradas.

- **Estado 2 –** Após todas as células serem preenchidas existem conflitos (*threats*) resultantes da repetição de números por linhas e colunas. Neste estado, é então calculado o número de repetições existentes, de acordo com a seguinte equação:

$$Threats = \frac{(count) \times (count - 1)}{2} \text{ [35]}$$

- **Estado 3 –** Neste estado, o algoritmo executa uma série de trocas aleatórias entre os números que não fazem parte do puzzle inicial, por forma a reduzir o número de conflitos para zero. Para esta troca aleatória, é usado o gerador de números pseudoaleatórios *Galois LFSR*[3][37, 38], indicando este quais as células que permutarão o seu valor. Para cada troca, é calculada a diferença entre o valor da *threat* atual com o valor anterior (ΔE), determinando, desta forma, se a troca é ou não vantajosa. Baseada neste valor, a troca terá que ocorrer sempre que ΔE <0 (reduzindo o número de conflitos totais).

Cada conjunto destes três estados corresponde a um *core*, isto é, a um núcleo de processamento do solucionador.

Os resultados foram obtidos com a implementação do algoritmo em *software* (recorrendo à linguagem *Java*) e em *hardware*, usando para isso a *FPGA Virtex-II Pro* da *Xilinx*, incluída na placa *XUP* da *Digilent*. O projeto, na sua totalidade, possui três *cores* independentes. Apesar da redundância, uma vez que a troca de números é feita de forma aleatória, cada *core* seguirá um caminho diferente. O tempo de execução do algoritmo é determinado pelo *core* mais rápido a atingir a solução.

Na Tabela 2.4 podem ser observados os recursos ocupados com todos os três *cores*.

Tabela 2.4 – Tabela de recursos usados pelos três *cores* [35]

Recursos	Usados	Disponíveis	Percentagem
Slices	11 769	13 696	85%
4-LUT	14 096	27 392	51%
BRAM	96	136	70%

A máxima frequência de operação é estabelecida nos 53 *MHz*, contudo foi usado um *Clock Divider* para reduzir para metade a frequência do cristal da placa (100 *MHz*) [35]. De acordo com este, foram obtidos os resultados apenas para os puzzles do Tipo A, desde a ordem N = 6 até N = 12. Todos os outros puzzles disponibilizados como instâncias de teste na *FPT'09* [7] não foram resolvidos em tempo útil. De notar que *t_max* indica que o

[3] *Linear Feedback Shift Register*, muito usado para gerar números pseudoaleatórios, tendo como base o estudo de campos finitos de *Évariste Galois*.

solucionador necessita de tempo superior ao limite imposto na competição ($0{,}0003 \times n^6$) para resolver o puzzle.

Comparativamente aos resultados obtidos em *software* (processador *P4 dual-core*, 2.86 *GHz*), com o mesmo algoritmo implementado em *Java*, estes obtiveram um rácio variável, entre 0.50 e 4.01 (*speedup*) [35], evidenciando-se a capacidade da *FPGA* competir com um *PC* que apresenta uma frequência de operação 57x superior.

Por fim, a falta de desempenho deste método (*simulated annealing*) é evidente nos puzzles mais difíceis (Tipo B), pelo que a sua otimização constitui uma proposta para trabalho futuro, assim como a implementação de alguns métodos heurísticos [35].

2.4.4 Análise de resultados da FPT'09

Nesta competição, estiveram presentes três equipas de diferentes universidades, usando cada uma delas uma abordagem diferente para a implementação dos solucionadores: *força-bruta* [8], processador específico para resolver *Sudoku* [33] e *simulated annealing* [35]. Cada uma destas equipas usou equipamentos idênticos de acordo com as normas da competição, por forma a que esta fosse justa [32]. Além destas, competiram também dois solucionadores *state-of-the-art*, implementados em *software* (solucionadores *SAT*), tendo estes como objetivo a comparação de resultados com o *hardware*. Mais tarde, em 2010, surgiu uma outra abordagem na implementação do solucionador, sendo esta feita através de uma matriz de incidência *ECP* (*Exact Cover Problem*) [39].

Na Tabela 2.5, podem ser observados os resultados dos tempos de execução obtidos, em *hardware*, para cada equipa, juntamente com o solucionador *ECP* [39] (BF – *TU Delft*, SP – *Processador específico para resolver Sudoku*, SA – *Solucionador SSAS,* ECP – *Exact Cover Problem*) e *software* (MS – *MiniSat 2 beta*, MSp – *Preprocessing & MiniSat 2 beta*).

Por observação da tabela, os melhores resultados em *hardware*, quer em tempos de execução, quer na capacidade de resolver puzzles de maior ordem, foram obtidos por parte do solucionador *ECP*.

Tabela 2.5 – Resultados da competição *FPT'09* em ms [39]

	BF	SP	SA	ECP	MS	MSp
3a	0,020821	0,008600		0,000257	0,001999	≈ 0
4a	0,221379	0,024761		0,001306	0,016997	0,000999
5a		0,060281		0,009064	0,105983	0,002999
6a	0,115347	0,117894	0,145	0,002838	0,38894	0,001999
7a	0,211343	0,224523	0,223	0,007863	1,3208	0,001999
8a	0,096424	0,058375	0,658	0,016489	3,9564	0,002999
9a		0,300835	5,662	0,075043	10,3444	0,008998
10a		0,322603	19,780	0,11694	24,2893	0,016997
11a		0,381182	147,490	0,183406	-	0,035994
12a		-	557,520	0,558018	-	0,069989
13a		-		0,802226	-	0,112982
14a		-		2,017806	-	0,182972
15a		-			-	0,276957
3b	0,012460	0,009762		0,003178	0,001999	≈ 0
4b				(1,27567)	0,020996	0,001999
5b				(28,8915)	(0,116982)	0,004999
6b					(1008,2)	0,017997
7b					-	0,044993
8b					-	0,117982
9b					-	0,277957
10b		-			-	1,59091
11b		-			-	1,16082
12b		-			-	2,28465
14b		-			-	7,2

2.5 Conclusões

Neste capítulo, fez-se uma breve descrição do que é uma *FPGA*, assim como também se retratou a sua evolução ao longo dos anos.

Além disto, foi apresentada uma breve nota histórica do puzzle *Sudoku*, assim como alguns métodos lógicos de resolução deste, nomeadamente os métodos *Singles*, *Hidden Number* e *Tentativa e Erro*. Para a implementação destes, é necessário recorrer a outras técnicas, também aqui apresentadas, servindo estas de base para as futuras implementações em *software* e *hardware*.

Foram, também, neste capítulo, abordadas algumas implementações apresentadas para a competição *Sudoku Solvers on FPGA* na Conferência Internacional *FPT'09* e seus resultados.

3 Estrutura de dados

Neste capítulo, será abordada a forma como os dados serão armazenados em memória. O modo como os dados são guardados dita o método de funcionamento de cada uma das rotinas implementadas.

Para armazenar os dados, serão usados os blocos de memória embutidos na *FPGA*, *Block RAMs* [40]. Estes blocos permitem evitar desperdícios da capacidade lógica da *FPGA* em implementar memórias distribuídas.

Foi escolhido o tipo de memória *Single Port RAM* (Figura 3.1), com um porto apenas, por forma a manter a simplicidade das memórias e porque também não serão necessárias memórias mais complexas como iremos ver na implementação do processo de resolução de puzzles, na secção 5.2.

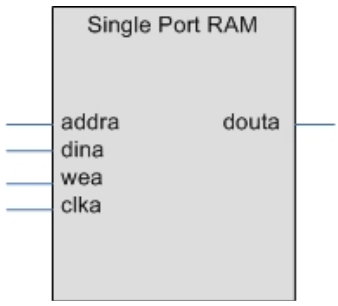


Figura 3.1 – Bloco de uma memória *Single Port RAM*

Para a sua implementação utilizou-se o *Block Memory Generator v3.8* que está inserido no núcleo de propriedade intelectual da *Xilinx* (*Xilinx Intellectual Property cores*).

Os tamanhos dos barramentos de endereços e de dados são variáveis consoante a aplicação da memória desejada. De seguida, são apresentados os cálculos, para o tamanho do barramento de endereços, e dados, para cada uma das memórias necessárias.

3.1 Dados que são necessários armazenar em memória

A primeira questão que se coloca é: que dados são necessários guardar em memória? A resposta a esta questão pode ser rapidamente observada na Tabela 3.1.

Tabela 3.1 – Dados que necessitam ser armazenados em memória

Aplicação	Dados a guardar
Puzzle Sudoku	Números constituintes do puzzle Sudoku.
Puzzle Auxiliar	Memória responsável por manter uma cópia do puzzle. Necessária para alguns algoritmos implementados.
Lista de Possibilidades	Candidatos de cada célula do puzzle.
Mapa de colunas	Identificadores de cada coluna do puzzle.
Mapa de caixas	Identificadores de cada caixa do puzzle.
Mapa de mínimos	Ordenação crescente das células do puzzle que possuem menor número de possibilidades.
Mapa de índices	Responsável por indicar quais os puzzles livres e quais os ocupados. Surgirá apenas na fase de tentativa e erro.

O acesso a estas memórias, quer ao barramento de dados quer ao de endereços, ou ao dos seus sinais de controlo, é gerido por uma unidade de controlo que resultará no solucionador do puzzle (Figura 3.2). A sua implementação será abordada nos capítulos seguintes.

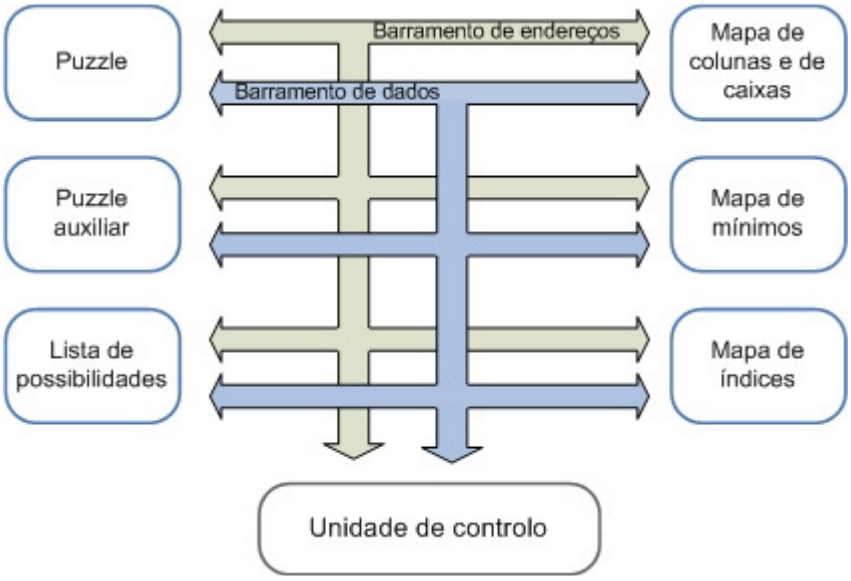


Figura 3.2 – Esquema ilustrativo do acesso às *Block RAMs* por parte da *Unidade de Controlo*

3.2 Organização dos dados em memória

3.2.1 Puzzle e puzzle auxiliar

O puzzle Sudoku consiste numa matriz 9x9, o que perfaz um total de 81 números. Na implementação em *software*, esta matriz é facilmente implementada através de um *array* bidimensional de 9 linhas e 9 colunas (ex.: puzzle[9][9]). No que diz respeito à implementação em *hardware*, os números serão guardados por sequências de linhas da matriz, isto é, as primeiras 9 posições da memória corresponderão à primeira linha do puzzle e assim sucessivamente, tal como se pode observar na Figura 3.3.

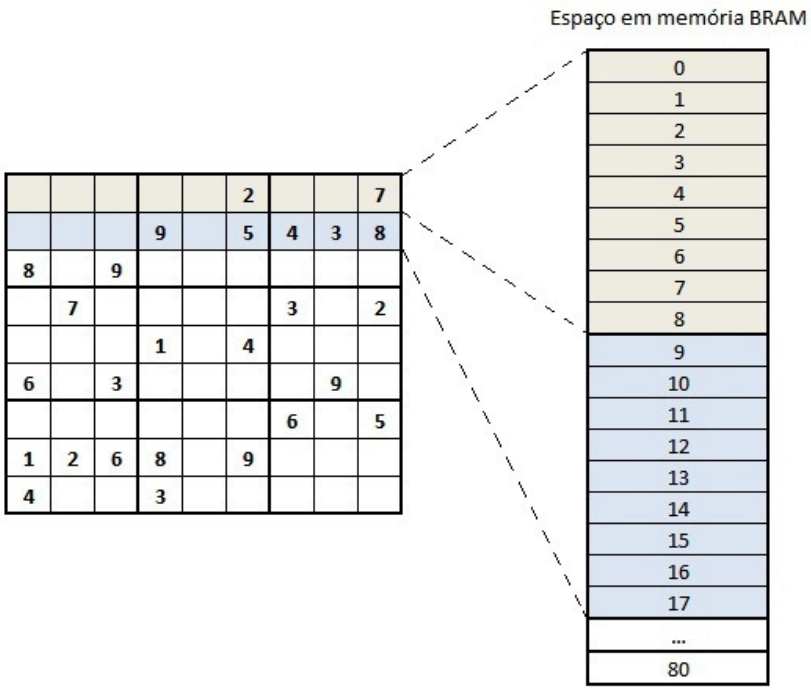


Figura 3.3 – Armazenamento dos dados do puzzle em *Block RAM*

É então necessária uma memória com capacidade para 81 posições, por forma a guardar todos os números do puzzle, e cada posição terá que armazenar um número de 1 a 9.

$$\text{Bits do barramento de endereços} = \lceil \log_2(81) \rceil = 7 \text{ bits}$$

$$\text{Bits do barramento de dados} = \lceil \log_2(9) \rceil = 4 \text{ bits}$$

Assim, serão necessários 7 bits para o barramento de endereços, por forma a armazenar todos os 81 números do puzzle e cada posição (*word* da memória) necessitará de 4 bits, para descrever todos os números.

Relativamente ao puzzle auxiliar, este é exatamente igual ao puzzle principal, uma vez que servirá para efetuar a salvaguarda dos valores do mesmo em determinadas etapas do algoritmo de tentativa e erro.

3.2.2 Lista de possibilidades

A lista de possibilidades consiste numa lista de 9 números para cada célula do puzzle. No entanto, apesar de não existirem 9 números passíveis de ser colocados numa célula, dadas as regras do Sudoku, esta foi projetada para o pior caso. Uma vez que interessa manter a correspondência entre células do puzzle e as da lista, isto é, para que uma posição *Y* de uma dada célula do puzzle corresponda à mesma célula de posição *Y* na lista de possibilidades (*hardware*); ou para que uma célula (m,n) do *array* bidimensional, que é o puzzle, corresponda exatamente à mesma célula (m,n) da lista de possibilidades, os dados terão que ser guardados tendo por base o mesmo *array* bidimensional já referido. Em termos de *software*, isto é facilmente implementado, recorrendo a um *array* tridimensional onde a cada posição da matriz bidimensional 9x9 correspondem 9 posições para a escrita de números que são passíveis de ser colocados na respetiva célula (ex.: lista[9][9][9]), sendo que este corresponde ao pior caso. Com esta implementação, a mesma posição (m,n) do puzzle corresponde à mesma célula (m,n) da lista de possibilidades, e assim, os índices dos *arrays* mantêm-se.

No que diz respeito à implementação em *hardware*, os números serão guardados da mesma forma que o puzzle (ver Figura 3.3) e, como tal, será necessário espaço em memória para as mesmas 81 posições e, portanto, o barramento de endereços terá os mesmos 7 bits. Desta forma, o mesmo endereço de posição de memória para aceder a uma célula do puzzle permite aceder também às suas respetivas possibilidades. No entanto, como cada posição necessita de armazenar no máximo 9 números, não poderá contar apenas 4 bits, como acontecia no caso do puzzle. Assim, tem-se que:

$$\text{Bits para representar um número de 1 a 9} = \lceil \log_2(9) \rceil = 4 \text{ bits}$$

$$\text{Bits para representar nove números} = 9 \times 4 \text{ bits} = 36 \text{ bits}$$

Com uma *word* de 36 bits por posição de memória é permitido armazenar os 9 números possíveis. O preenchimento/acesso a cada candidato será feito através de *shifts* consecutivos de 4 bits, pois os números serão guardados de acordo com a configuração apresentada na Figura 3.4.

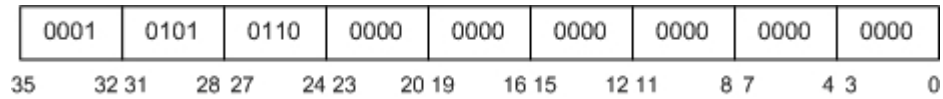


Figura 3.4 – Organização do barramento de dados para a lista de possibilidades

Esta configuração necessita de obedecer a um conjunto de regras para que todos os blocos que precisem de aceder à lista de possibilidades o possam fazer corretamente e de forma unânime:

1. Os números serão preenchidos começando pelos bits mais significativos e terminando nos menos significativos do barramento de dados da lista de possibilidades;
2. Os números terão que ser preenchidos por ordem crescente por forma a otimizar futuras procuras por número;
3. Não poderão existir números "em branco", isto é, número zero entre dois números diferentes de zero, ou seja, todos os números candidatos a uma determinada célula terão que estar nas "casas" múltiplas de 4 consecutivas, tal como demonstrado na Figura 3.4. Isto constitui não só uma forma de organização dos dados como também permite uma certa otimização aquando da procura de determinados números (ex.: após surgir o primeiro zero na lista, não será necessário procurar as casas seguintes).

3.2.3 Mapas de colunas e de caixas

Estes mapas consistem em matrizes 9x9 onde cada posição possui um identificador (id), que corresponde ao número da coluna ou da caixa cuja posição é endereçada pelo

barramento de endereços, por forma a permitir identificar se uma dada célula pertence a que coluna ou caixa.

Do ponto de vista de *software*, estes não são necessários, mas já o são em *hardware*, devido a existir uma maior dificuldade na identificação das colunas e das caixas através apenas de lógica combinatória.

É então necessária uma memória com capacidade para 81 posições, por forma a guardar os identificadores de cada célula do puzzle, e cada posição da memória (*word*) terá que armazenar um número de 1 a 9, pelo que serão necessários 7 bits para o barramento de endereços e 4 bits para o barramento de dados, tal como na memória para o puzzle. As memórias são já inicializadas com os identificadores preenchidos, tal como demonstrado na Figura 3.5 e Figura 3.6.

	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	9
B	1	2	3	4	5	6	7	8	9
C	1	2	3	4	5	6	7	8	9
D	1	2	3	4	5	6	7	8	9
E	1	2	3	4	5	6	7	8	9
F	1	2	3	4	5	6	7	8	9
G	1	2	3	4	5	6	7	8	9
H	1	2	3	4	5	6	7	8	9
I	1	2	3	4	5	6	7	8	9

Figura 3.5 – Mapa de colunas

	1	2	3	4	5	6	7	8	9
A	1	1	1	2	2	2	3	3	3
B	1	1	1	2	2	2	3	3	3
C	1	1	1	2	2	2	3	3	3
D	4	4	4	5	5	5	6	6	6
E	4	4	4	5	5	5	6	6	6
F	4	4	4	5	5	5	6	6	6
G	7	7	7	8	8	8	9	9	9
H	7	7	7	8	8	8	9	9	9
I	7	7	7	8	8	8	9	9	9

Figura 3.6 – Mapa de caixas

3.2.4 Mapa de mínimos

Este mapa consiste numa matriz 9x9 necessária à fase de tentativa e erro. Esta matriz conterá a ordenação crescente das células que possuem menor número de possibilidades,

isto é, à célula com menor número de possibilidades é atribuído o valor "1", à segunda célula com menor número de possibilidades o valor "2", e assim sucessivamente. Interessa, portanto, manter a correspondência direta entre células do puzzle e da lista de possibilidades, pelo que a configuração do barramento de endereços será a mesma. Na pior das hipóteses (puzzle vazio), é necessário que a ordem das células vá desde 1 até 81 e, portanto, serão necessários 7 bits para descrever estes valores (*word* da memória), assim como também 7 bits para endereçar as 81 células, tal como acontecia anteriormente. No entanto, em geral, os puzzles já trazem metade das casas preenchidas, tendo cerca de 41 casas livres, e, por isso, com 6 bits é possível escrever valores da ordem das células que variam entre 1 e 64, não havendo assim necessidade de 7 bits para o barramento de dados.

No que diz respeito à implementação em *software*, este mapa consiste apenas numa matriz bidimensional 9x9 (9 linhas, 9 colunas) tal e qual como o puzzle.

3.2.5 Mapa de índices

Este mapa apenas surgirá no processo de tentativa e erro e é responsável por indicar quais os puzzles que se encontram ocupados e quais os que se encontram livres (secção 4.2.2). Estes puzzles (*Block RAMs*), com a mesma configuração do Puzzle já mencionado, possuem, associados a si, um índice. Inicialmente, estes puzzles encontrar-se-ão sinalizados como vazios e, à medida que forem sendo preenchidos, isto é, usados no algoritmo de tentativa e erro, o seu respetivo índice será sinalizado como ocupado neste mapa, permitindo, assim, ao sistema, identificar rapidamente quais os índices dos puzzles ainda disponíveis para usar o algoritmo.

Assim, em termos de barramento de dados, apenas será necessário 1 bit para sinalizar puzzle desocupado ou puzzle ocupado. Já no que diz respeito ao barramento de endereços, uma vez que corresponde ao número de índices passíveis de ser armazenados, este pode variar consoante o número de *Block RAMs* disponíveis na *FPGA*, e também com o próprio algoritmo implementado. No entanto, o barramento de endereços pode ser arbitrado com um valor elevado, mesmo que permita endereçar mais índices do que o número de *Block RAMs* disponíveis, e adicionalmente usar um sinal com o valor real do número de *Block RAMs* que se encontram disponíveis, tendo este que ser ajustado para cada caso. Esta organização é particularmente útil para explorar vários tipos de algoritmos, tais como

Tentativa e Erro com Processamento Paralelo (secção 5.2.3), sem necessidade de recriar o mapa de índices com novas características. Deste modo, serão assim arbitrados 5 bits para o barramento de endereços (permite endereçar 32 índices).

Do ponto de vista do *software*, este corresponderá a uma matriz linha de 32 posições (*array* unidimensional).

3.3 Diagrama de blocos

Antes de apresentar o diagrama de blocos da organização das *Block RAMs*, os parâmetros de cada uma das memórias usadas podem ser sintetizados nas tabelas 3.2 e 3.3.

Tabela 3.2 – Tabela síntese dos parâmetros dos *Arrays* consoante a aplicação

Aplicação	Tipo de array	Dimensões
Puzzle	Bidimensional	9 x 9
Puzzle Auxiliar	Bidimensional	9 x 9
Lista de Possibilidades	Tridimensional	9 x 9 x 9
Mapa de mínimos	Bidimensional	9 x 9
Mapa de índices	Unidimensional	32

Tabela 3.3 – Tabela síntese dos parâmetros das *Block RAMs* consoante a aplicação

Aplicação	Word (bits)	Nº de Words	Endereço (bits)
Puzzle	4	81	7
Puzzle Auxiliar	4	81	7
Lista de Possibilidades	36	81	7
Mapa de colunas	4	81	7
Mapa de caixas	4	81	7
Mapa de mínimos	6	81	7
Mapa de índices	1	32	5

Tendo em conta o barramento de dados e o barramento de endereços já especificado na Tabela 3.3, o esquema ilustrativo de acesso às *Block RAMs* da Figura 3.2 pode-se traduzir no diagrama de blocos simplificado da Figura 3.7.

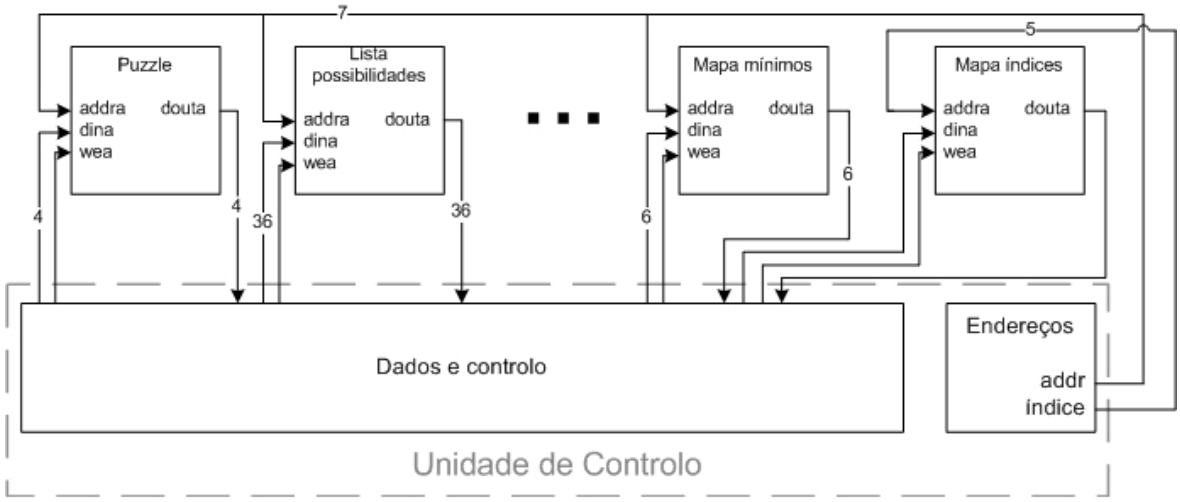


Figura 3.7 – Diagrama de blocos simplificado da estrutura de dados

De notar que o diagrama (Figura 3.7) não apresenta todos os blocos referidos neste capítulo apenas por uma questão de simplificação. No entanto, todos os blocos apresentados fazem parte desta estrutura e deverão ser levados em conta na implementação do solucionador.

Nos capítulos seguintes será descrita a construção do bloco *Unidade de Controlo*, que fará a gestão dos acessos às diferentes memórias e implementará o solucionador de puzzles.

3.4 Conclusões

Foram, neste capítulo, discutidos que dados necessitariam ser guardados, bem como o modo como o seriam. Desta forma, os dados descritos na Tabela 3.1 serão guardados através do uso de *Block RAMs*, do tipo *Single Port RAM*, por forma a manter a simplicidade das mesmas.

O acesso a estas memórias, quer ao barramento de dados, como ao de endereços, quer aos seus sinais de controlo, é gerido pela *Unidade de Controlo*, que resultará no solucionador de puzzles. Posto isto, as *Block RAMs* foram projetadas com os parâmetros descritos na Tabela 3.3, resultando no diagrama de blocos representado na Figura 3.7.

4 Métodos lógicos de resolução – software

Neste capítulo, serão descritas as implementações em *software* dos métodos lógicos de resolução abordados na secção 2.3.2. Será também descrito o modo como estes se interligam formando o sistema global solucionador de puzzles.

4.1 Rotinas de resolução

4.1.1 Singles

A implementação deste método lógico em *software* obedece aos seguintes passos ilustrados no pseudocódigo da Figura 4.1.

```
FOR m = 0 : 8
 FOR n = 0 : 8
    IF célula (m,n) == 0
       IF possibilidade célula (m,n) for singular
          célula (m,n) = possibilidade célula (m,n);
          possibilidade célula (m,n) = 0;
```

Figura 4.1 – Pseudocódigo da implementação em *software* da função *Singles*

Tal como ilustrado, a implementação desta função em *software* é bastante simples. Esta percorre todas as posições do *array* bidimensional que é o puzzle e, para cada célula, são lidas apenas a primeira e segunda possibilidades, recorrendo para isso ao *array* de possibilidades (secção 3.2.2) que, de acordo com o modo como as possibilidades são armazenadas, se a primeira possibilidade for diferente de zero e a segunda igual a zero então estamos perante uma situação de singularidade de candidatos para a célula em questão. Se for este o caso, então a célula é de imediato preenchida com o valor da possibilidade, e esta é removida do *array* de possibilidades, deixando então de existir possibilidades para esta célula que, agora, se encontra preenchida. O mesmo raciocínio é aplicado a todas as outras células do puzzle.

4.1.2 Hidden Number

A implementação deste método em *software* pode ser dividida em três fases, todas elas muito idênticas. A primeira fase corresponde à verificação da existência de candidatos únicos não diretos por linhas, a segunda por colunas e, finalmente, a terceira corresponde à verificação por caixas, ou mais especificamente, por cada caixa constituinte do puzzle.

Seja *m* o valor da linha atual, *n* o valor da coluna atual e *i* o índice da possibilidade a ser lida para uma dada célula, tem-se assim, para a primeira fase, a seguinte implementação:

```
FOR m = 0 : 8
 FOR número = 1 : 9
    count = 0;
    FOR n = 0 : 8
       i = 0;
       WHILE possibilidade célula (m,n,i) ≠ 0
          IF possibilidade célula (m,n,i) == número
             count++;
          i++;

    IF count == 1
       FOR i = 0 : 8
          IF célula (m,i) == 0
             FOR p = 0 : 8
                IF possibilidade célula (m,i,p) == número
                   célula (m,i) = número;
```

Figura 4.2 – Pseudocódigo da implementação em *software* da função *Hidden Number*, para a verificação por linhas

Tal como ilustrado na Figura 4.2, esta implementação, apesar de mais complexa face à implementação da função *Singles*, é também bastante simples. Este algoritmo percorre cada uma das posições de uma dada linha e, para cada número, efetua a contagem do número de vezes que este surge como candidato nas listas de possibilidades das diferentes células constituintes dessa mesma linha. Se este surgir apenas uma vez, estamos então perante um candidato único não direto e, como tal, basta percorrer novamente as posições da linha e encontrar a célula que possua este como candidato na sua lista de possibilidades e preenchê-la com o seu valor.

Para a segunda fase, o mesmo raciocínio é aplicado para a verificação por colunas. Em termos de implementação, basta apenas trocar os ciclos *for*, que fazem variar a posição

nas colunas, pelos ciclos *for*, que fazem variar a posição nas linhas, e vice-versa. Tal pode ser observado na Figura 4.3.

```
FOR n = 0 : 8
 FOR número = 1 : 9
    count = 0;
    FOR m = 0 : 8
       i = 0 ;
       WHILE possibilidade célula (m,n,i) ≠ 0
          IF possibilidade célula (m,n, i) == número
             count++;
          i++;

    IF count == 1
       FOR i = 0 : 8
          IF célula (i,n) == 0
             FOR p = 0 : 8
                IF possibilidade célula (i,n,p) == número
                   célula (i,n) = número;
```

Figura 4.3 - Pseudocódigo da implementação em *software* da função *Hidden Number*, para a verificação por colunas

A terceira fase, que consiste na verificação por caixas, é um pouco mais complexa, uma vez que, ao contrário do que acontece nas outras duas fases, esta apresenta a necessidade de determinar se posição atual se encontra dentro ou fora da caixa que se pretende verificar. Para tal, recorre-se ao mapa de caixas (ver secção 3.2.3), que em *software* não é nada mais que uma simples função que, para uma dada posição, devolve o número da caixa (*id*) a que essa posição corresponde.

Assim, o raciocínio anterior sofre ligeiras alterações, como demonstrado na Figura 4.4.

De acordo com a Figura 4.4, nesta terceira fase, são percorridas cada uma das caixas do puzzle, fazendo variar a posição das várias células e averiguando se estas pertencem ou não à caixa pretendida e, para cada número, é contado o número de vezes que este surge como candidato nas diferentes listas de possibilidades de cada célula constituinte dessa mesma caixa. Após isto, verifica-se se o resultado da contagem é unitário, e, em caso afirmativo, estamos então perante um candidato único não direto, pelo que são novamente percorridas todas as células da caixa, por forma a encontrar a célula que possua este como candidato na sua lista de possibilidades. Encontrada a célula, esta é preenchida com o valor do candidato.

```
FOR caixa = 1 : 9
 FOR número = 1 : 9
    count = 0;
    FOR m = 0 : 8
       FOR n = 0 : 8
          IF idCaixa (m,n) == caixa
             i = 0;
             WHILE possibilidade célula (m,n,i) ≠ 0
                IF possibilidade célula (m,n,i) == número
                   count++;
                i++;

    IF count == 1
       FOR m = 0 : 8
          FOR n = 0 : 8
             IF idCaixa (m,n) == caixa
                IF célula (m,n) == 0
                   FOR p = 0 : 8
                      IF possibilidade célula (m,n,p) == número
                         célula (m,n) = número;
```

Figura 4.4 - Pseudocódigo da implementação em *software* da função *Hidden Number*, para a verificação por caixa

Estas três fases, em conjunto, constituem a rotina *Hidden Number*, responsável por averiguar a existência de candidatos únicos não diretos para linhas, colunas e caixas.

4.1.3 Validate

Do ponto de vista da implementação em *software*, esta rotina pode ser dividida em três fases distintas. A primeira fase corresponde à validação dos números existentes numa caixa, a segunda, à validação dos números numa linha e a terceira corresponde à validação dos números numa coluna.

Seja *m* o valor atual da linha e *n* o valor atual da coluna, parâmetros de entrada desta função, e sejam *x*, *y*, *k* e *j* variáveis auxiliares tem-se o seguinte pseudocódigo da Figura 4.5 como implementação para a primeira fase.

```
IF m < 3
  x = 0;
IF m < 6 && m ≥ 3
  x = 3;
IF m ≥ 6
  x = 6;

IF n < 3
  y = 0;
IF n < 6 && n ≥ 3
  y = 3;
IF n ≥ 6
  y = 6;

FOR k = x : x + 3
  FOR j = y : y + 3
     IF m ≠ k || n ≠ j
        IF célula (m,n) == célula (k,j)
           RETURN false;

RETURN true;
```

Figura 4.5 - Pseudocódigo da implementação em *software* da função *Validate*, para a validação por caixa

Tal como ilustrado na Figura 4.5, esta implementação começa por averiguar a que caixa corresponde a célula (m,n) passada como parâmetro de entrada. Este passo corresponde aos vários *ifs* iniciais. Ao mesmo tempo, é definida a primeira posição da mesma caixa, através das variáveis *x* e *y*.

Após definir a posição inicial da caixa em questão, são percorridas todas as células constituintes do puzzle, comparando cada uma destas células (k,j) com a célula (m,n) inicial (Figura 4.6). Dependendo do resultado desta comparação, é retornado um valor booleano que será avaliado mais tarde, como veremos.

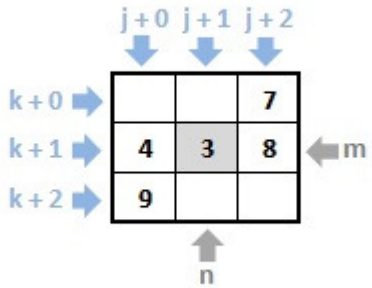


Figura 4.6 – Exemplo ilustrativo da verificação por caixa

Para a segunda fase, validação por linhas, é aplicado o mesmo raciocínio. Seja *m* o valor atual da linha e *n* o valor atual da coluna, parâmetros de entrada desta função, e *j* uma variável auxiliar, tem-se para a implementação desta segunda fase o pseudocódigo da Figura 4.7.

```
FOR j = 0 : 8
  IF j ≠ n
     IF célula (m,n) == célula (m,j)
        RETURN false;

RETURN true;
```

Figura 4.7 – Pseudocódigo da implementação em *software* da função *Validate*, para a validação por linha

Observando-se a Figura 4.7, esta função começa por incrementar uma variável auxiliar *j*, que corresponde neste caso a percorrer todas as colunas como se irá constatar.

Uma vez que *m* (índice de linha) não varia, é necessário verificar se *j* é diferente de *n*. Em caso afirmativo estamos perante células diferentes da mesma linha e poder-se-á proceder com a verificação, averiguando então se a célula (m,n) é igual à célula (m,j) (Figura 4.8). Consoante o resultado desta comparação é retornado o valor *true* ou *false*.

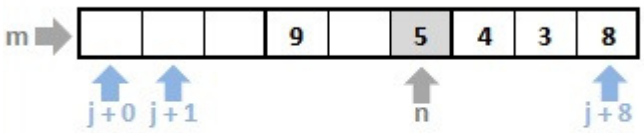


Figura 4.8 – Exemplo ilustrativo da verificação por linha

Finalmente, para a terceira fase, que corresponde à validação por colunas, basta apenas efetuar uma ligeira alteração da função correspondente à fase anterior, tal como se pode observar na Figura 4.9.

```
FOR j = 0 : 8
  IF j ≠ m
     IF célula (m,n) == célula (j,n)
        RETURN false;

RETURN true;
```

Figura 4.9 – Pseudocódigo da implementação em *software* da função *Validate*, para a validação por coluna

Pela Figura 4.9, pode-se observar que esta função é muito idêntica à anterior. Como tal, esta começa por incrementar uma variável auxiliar *j*, que corresponde, neste caso, a percorrer todas as linhas.

Uma vez que *n* (índice de coluna) não varia, é necessário verificar se *j* é diferente de *m*. Em caso afirmativo estamos perante células diferentes da mesma coluna e poder-se-á proceder com a verificação, averiguando se a célula (m,n) é igual à célula (j,n) (Figura 4.10), sendo que o valor retornado pela função varia consoante o resultado desta comparação.

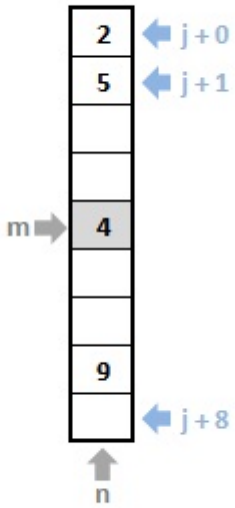


Figura 4.10 – Exemplo ilustrativo da verificação por coluna

As três fases apresentadas, em conjunto constituem a função *Validate* na sua totalidade. Seja a função *matrixchk(m,n)* correspondente à implementação da primeira fase, a função *rowchk(m,n)* correspondente à segunda fase e a função *columnchk(m,n)* correspondente à terceira fase, e seja ainda *c* o valor booleano retornado por cada uma destas funções, tem-se então que a função *Validate* é dada pelo pseudocódigo da Figura 4.11.

Assim, dada uma célula (m,n) é possível verificar se esta se encontra preenchida de acordo com as regras do puzzle Sudoku. Para isso, são avaliadas todas as outras células correspondentes à linha, coluna e caixa onde a célula (m,n) se encontra.

De notar que esta função apenas compara uma célula de uma dada posição (m,n) com todas as restantes células da sua caixa, linha e coluna.

```
c = matrixchk(m,n)
IF c == false
  RETURN false;

c = rowchk(m,n)
IF c == false
  RETURN false;

c = columnchk(m,n)
IF c == false
  RETURN false;

RETURN true
```

Figura 4.11 – Pseudocódigo da implementação em *software* da função *Validate*

4.1.4 Lista de possibilidades ou candidatos

Para implementar esta função, recorrer-se-á a uma função previamente desenvolvida: a função *Validate* (secção 4.1.3). Relembrando, quando fornecida a posição de uma célula, é retornado um valor *true* ou *false*, caso esta célula se encontre ou não preenchida de acordo com as regras do puzzle, respetivamente.

Assim, a sua implementação é muito simples. Basta para isso percorrer todas as células do puzzle e "preencher" cada uma das células vazias com números de 1 a 9, onde, para cada número, se "chama" a rotina *Validate*. Consoante o resultado desta rotina, determina-se se o número é passível ou não de ser colocado na respetiva célula. Se for, é então adicionado à lista de possibilidades (ver secção 3.2.2). Após verificar todos os números para uma dada célula, esta é colocada no seu estado inicial, isto é, vazia, uma vez que não se sabe qual dos candidatos será o que conduzirá à solução final. Este processo pode ser observado no pseudocódigo da Figura 4.12.

```
FOR m = 0 : 8
  FOR n = 0 : 8
     count = 0;
     IF célula (m,n) == 0
        FOR número = 1 : 9
           célula (m,n) = número;
           IF Validate (m,n) == true
              Adiciona número à lista de possibilidades na posição (m,n);
        célula (m,n) = 0;
```

Figura 4.12 – Pseudocódigo da implementação em *software* da função geradora da lista de possibilidades

Cada vez que uma célula é preenchida, bem ou mal, deve-se efetuar a atualização da lista de possibilidades. Isto permite:

- Retirar possibilidades associadas às células que foram entretanto preenchidas;
- Atualizar as possibilidades das outras células de acordo com as células já preenchidas e de acordo com as regras do puzzle.

É, portanto, necessária uma rotina de atualização, uma vez que a função desenvolvida apenas trata de gerar a lista a partir do seu estado inicial. Esta rotina de atualização pode ser expressa pelo pseudocódigo apresentado na Figura 4.13.

```
FOR m = 0 : 8
  FOR n = 0 : 8
     FOR i = 0 : 8
        IF célula (m,n) == 0
           IF possibilidade célula (m,n,i) ≠ 0
              célula (m,n) = possibilidade célula (m,n,i);
              IF Validate (m,n) == true
                 célula (m,n) = 0;
              ELSE
                 possibilidade célula (m,n,i) = 0;
                 shift das possibilidades restantes;
                 célula (m,n) = 0;
        ELSE
           possibilidade célula (m,n,i) = 0;
```

Figura 4.13 – Pseudocódigo da implementação em *software* da função responsável por atualizar a lista de possibilidades

Tal como ilustrado na Figura 4.13, esta implementação é bastante simples. Nela, incrementam-se a linha, coluna e uma variável auxiliar responsável por selecionar o

candidato a ser lido para cada célula. Assim, se uma célula se encontra vazia, é necessário verificar se as suas possibilidades ainda se encontram válidas, e, para isso, a célula é preenchida com as possibilidades já existentes e, recorrendo à rotina *Validate*, averigua-se se estas continuam como candidatos para a célula ou não. Caso deixem de ser candidatos, são então removidos da lista de possibilidades e é feito o *shift* dos candidatos seguintes (secção 3.2.2).

Se a célula não estiver vazia, é necessário preencher a sua lista a zeros, uma vez que esta poderia estar vazia num ciclo anterior e, como tal, os seus candidatos terão que deixar de existir.

Na Figura 4.14 pode-se observar o exemplo da lista de possibilidade de 10 células.

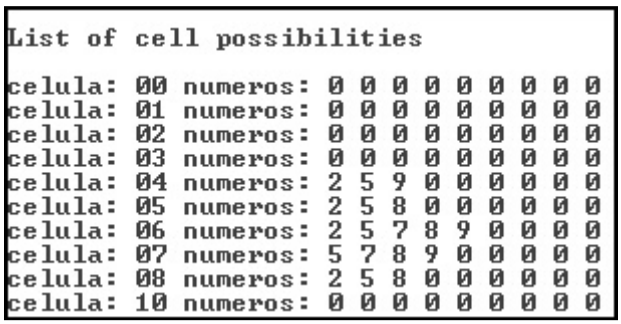


Figura 4.14 – Exemplo ilustrativo de uma lista de possibilidades resultante da implementação em *software* para 10 células

4.1.5 Mínimos

Uma vez existindo correspondência direta entre as linhas e colunas da lista de possibilidades e do mapa de mínimos (secção 3.2.4), torna-se bastante simples a implementação desta rotina em *software*. A primeira fase da implementação é responsável por gerar o mapa de mínimos do puzzle e a segunda fase por devolver a posição da célula que contém o mínimo número de candidatos.

De acordo com o pseudocódigo da Figura 4.15, é usada uma variável auxiliar para definir o índice do candidato de cada célula que se pretende verificar se existe ou não. Assim, são percorridas todas as células, averiguando quais as células que possuem dois candidatos apenas (candidato $i = 1$ preenchido e candidato $i + 1$ vazio), sendo estas preenchidas com o valor do contador *count* que corresponderá ao seu índice de ordem

crescente. Após verificar todas as células para a situação de 2 candidatos apenas, a variável *i* é incrementada e passa-se a avaliar a existência de um terceiro candidato e ausência de um quarto, e assim sucessivamente.

```
count = 1;

FOR i = 1 : 7
 FOR m = 0 : 8
    FOR n = 0 : 8
       IF possibilidade célula (m,n,i) ≠ 0
          IF possibilidade célula (m,n,i+1) == 0
             mapa_mínimos(m,n) = count++;
```

Figura 4.15 – Pseudocódigo da implementação em *software* da função geradora do mapa de mínimos

Após gerar o mapa de mínimos, basta percorrer agora todas as posições do mesmo e devolver a posição da célula que contém índice 1, tal como demonstrado no pseudocódigo da Figura 4.16. Esta célula é a que contém menor número de candidatos passíveis de ser colocados.

```
FOR m = 0 : 8
 FOR n = 0 : 8
    IF mapa_mínimos(m,n) == 1
       Devolve (m,n);
```

Figura 4.16 – Pseudocódigo da função responsável por devolver a posição da célula de mínimo

4.1.6 Filled

Para implementar esta função, basta percorrer cada uma das células do puzzle e verificar que nenhuma destas se encontra vazia, tal como demonstrado no pseudocódigo da Figura 4.17.

```
FOR j = 0 : 8
  FOR k = 0 : 8
     IF célula (j,k) == 0
        RETURN false;

RETURN true;
```

Figura 4.17 – Pseudocódigo da implementação em *software* da função *Filled*

De acordo com a Figura 4.17, são então percorridas todas as células do puzzle. Se uma dada célula se encontrar vazia, é então retornado o valor *false*, indicando que a condição de puzzle totalmente preenchido é falsa. Caso contrário, após verificar todas as células, se nenhuma estiver vazia, significa isto que estão todas preenchidas e, portanto, é retornado o valor *true* sinalizando o preenchimento de todas as células do puzzle.

4.1.7 Not Empty

A implementação desta função é toda ela muito idêntica à da função *Filled*. Como tal, são percorridas todas as células do puzzle até encontrar a primeira célula preenchida. O mesmo pode ser descrito através do pseudocódigo da Figura 4.18.

```
FOR j = 0 : 8
   FOR k = 0 : 8
      IF célula (j, k) ≠ 0
         RETURN true;

RETURN false;
```

Figura 4.18 – Pseudocódigo da implementação em *software* da função *Not Empty*

De acordo com o pseudocódigo apresentado na Figura 4.18, são então percorridas todas as células do puzzle. Se uma dada célula se encontrar preenchida, é retornado o valor *true*, sinalizando que o puzzle contém células preenchidas; caso contrário, o puzzle encontra-se vazio e é retornado o valor *false*.

Esta função é de extrema importância na fase de tentativa e erro, permitindo averiguar se um índice dado como ocupado se encontra efetivamente preenchido.

4.2 Sistema global

De seguida, serão apresentadas as duas abordagens para resolução de puzzles Sudoku, sem e com tentativa e erro. Estes dois sistemas variam, quer no tipo quer na quantidade de recursos utilizados.

4.2.1 Simples

Este é o primeiro sistema capaz de resolver puzzles Sudoku. Uma vez que não faz uso do método de tentativa e erro, a sua implementação torna-se bastante simples pois a maioria das funções mencionadas na secção 4.1 não serão ainda usadas nesta fase. Assim, o seu fluxograma pode ser dado pelo representado na Figura 4.19.

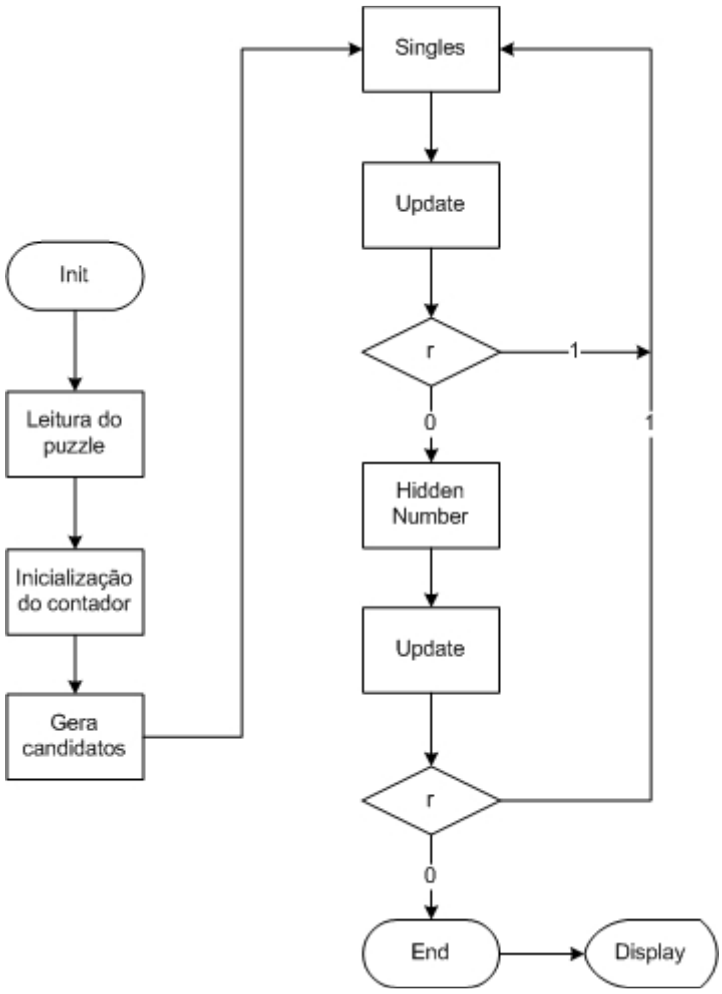


Figura 4.19 – Fluxograma do sistema solucionador simples de puzzles Sudoku

Com o *software* implementado, de acordo com este fluxograma, é possível resolver os puzzles mais simples, isto é, que não necessitem de uma fase de tentativa e erro e possam ser corretamente resolvidos recorrendo apenas aos métodos *Singles* e *Hidden Number*.

Como se pode observar pela Figura 4.19, tem-se assim as seguintes etapas para este sistema:

1. *Init* – o sistema começa por efetuar a inicialização do *array* bidimensional que é o puzzle. Esta inicialização é feita colocando todas as suas posições a zero;
2. *Leitura do puzzle* – é efetuada a leitura dos valores do puzzle, sendo que o valor das células é carregado para o *array* bidimensional já inicializado;
3. *Inicialização do contador* – é iniciado o processo de contagem de tempo que só será interrompido com a chegada ao estado *End*;
4. *Gera candidatos* – neste estado, é invocada a rotina responsável por gerar candidatos para cada uma das células (secção 4.1.4);
5. *Singles* – neste estado, é invocada a rotina responsável por identificar candidatos únicos e dá-se preenchimento das respetivas células (secção 4.1.1);
6. *Update* – após o término da rotina *Singles*, é necessário atualizar a lista de candidatos para cada célula, uma vez que, se foram identificados candidatos únicos, existem novas células preenchidas, o que gera uma lista de candidatos diferente;
7. *r* – neste ponto é avaliado o valor de *r*. Este corresponde ao valor retornado pela rotina *Singles* (ver secção 4.1.1). Se o valor deste for '1', isto é, *true*, significa que existem novas células preenchidas e, como tal, poderão existir novos candidatos únicos e, por isso, é necessário voltar a invocar a rotina *Singles*. O mesmo já não acontece caso o valor retornado seja '0', isto é, *false*, que corresponde à situação contrária, e, por conseguinte, o sistema avança por forma a invocar a rotina seguinte;
8. *Hidden Number* – neste estado, é invocada a rotina responsável pela identificação de candidatos únicos não diretos e dá-se o preenchimento das respetivas células (secção 4.1.2);
9. *Update* - após o término da rotina *Hidden Number*, é necessário atualizar a lista de candidatos para cada célula, uma vez que, se foram identificados candidatos únicos

não diretos, existem novas células preenchidas, o que gera uma lista de candidatos diferente;

10. *r* – neste ponto é novamente avaliado o valor de r que, desta vez, corresponde ao valor retornado pela rotina *Hidden Number*;
11. *End* – é parado o processo de contagem do contador, sendo armazenado o tempo de execução deste sistema que corresponde ao tempo da resolução de um puzzle. Em simultâneo, são apresentados o resultado da contagem do tempo de resolução e solução do puzzle (Figura 4.20).

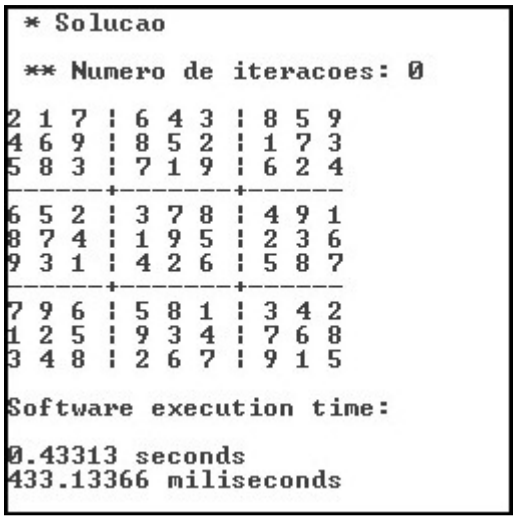


Figura 4.20 – Exemplo de resolução de puzzle com solucionador simples

4.2.2 Tentativa e erro

Com o objetivo de resolver puzzles mais complexos, surge então a implementação do método de tentativa e erro. A sua implementação é mais complexa e exige recorrer às ferramentas apresentadas na secção 4.1; contudo esta faz também uso da implementação do método simples, tal como pode ser observado no fluxograma da Figura 4.21.

Apesar de mais complexo, com o *software* implementado de acordo com este fluxograma é possível resolver qualquer tipo de puzzle, bastando para isso que exista memória suficiente (índices) para efetuar cópias do puzzle, tantas quanto necessárias. Com os computadores da atualidade, a falta de memória não será um problema, mas o mesmo já não ocorrerá no que diz respeito à *FPGA*, tal como será abordado na secção 5.2.2.

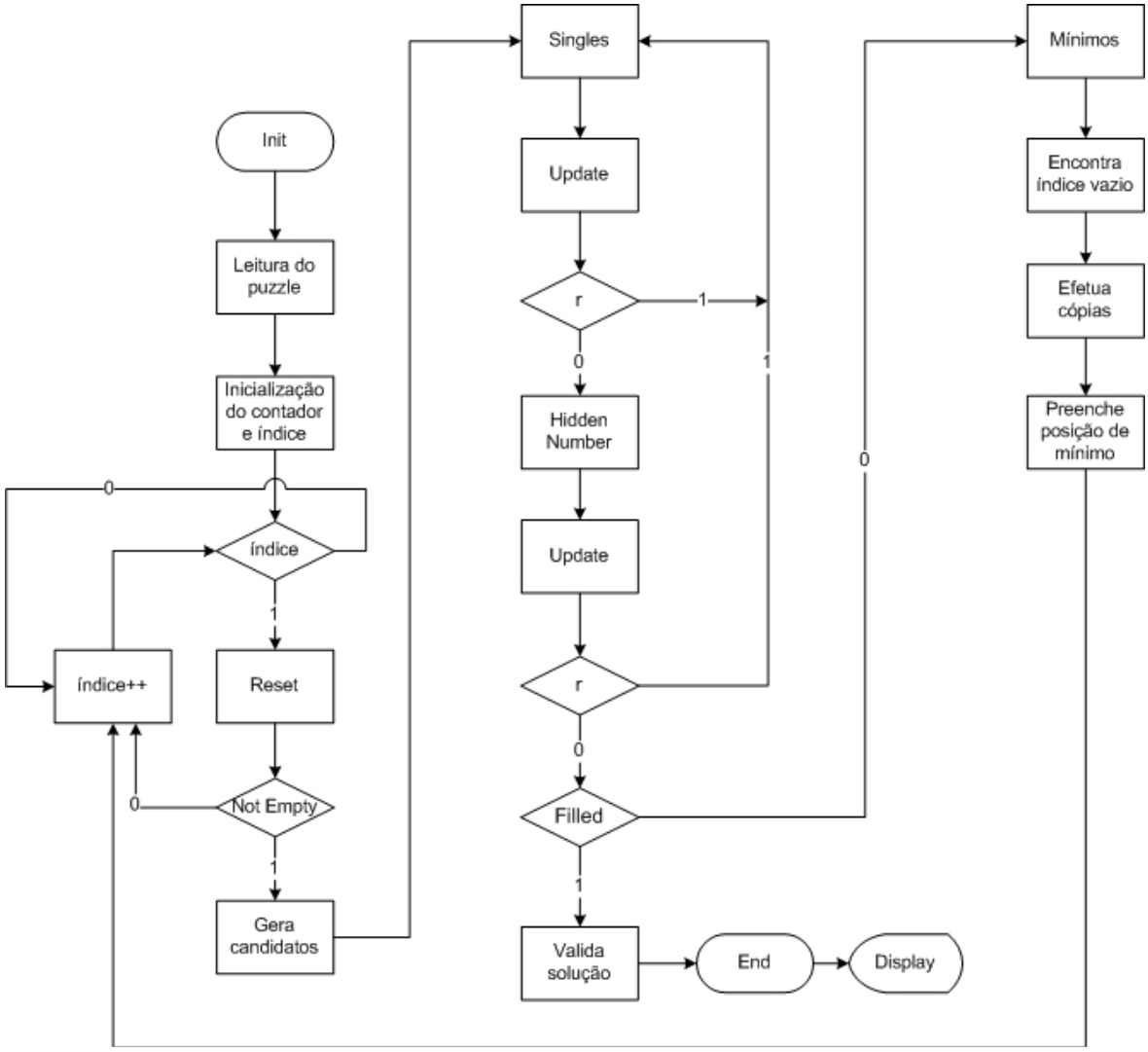


Figura 4.21 – Fluxograma do sistema solucionador com tentativa e erro de puzzles Sudoku

Como se pode observar pela Figura 4.21, têm-se assim as seguintes etapas para este sistema:

1. *Init* – o sistema começa por efetuar a inicialização do *array* bidimensional, que é o puzzle, assim como do *array* tridimensional, que contém a lista de possibilidades de todas as células. Esta inicialização é feita preenchendo cada uma das posições com o valor zero;
2. *Leitura do puzzle* – é efetuada a leitura dos valores do puzzle, sendo que o valor das células é carregado para o *array* bidimensional já inicializado;

3. *Inicialização do contador e índice* – é iniciado o processo de contagem que só será interrompido com a chegada ao estado *End*. Após isto, é preenchido o primeiro índice do mapa de índices, sinalizando que o primeiro puzzle encontra-se preenchido;
4. *Índice* – neste ponto, é averiguado se o índice atual se encontra preenchido. Se preenchido, pode-se continuar com o processo de resolução, caso contrário os índices continuam a ser incrementados;
5. *Reset* – todas as células da lista de possibilidades e as do mapa de mínimos são preenchidas com o valor zero, por forma a evitar interferências com ocorrências anteriores;
6. *Not Empty* – neste ponto, é invocada a rotina *Not Empty*, responsável por averiguar se, efetivamente, o puzzle correspondente ao índice atual possui células preenchidas. Se este não possuir células preenchidas, é necessário avançar para o próximo índice;
7. Nesta fase, é executado o mesmo algoritmo do solucionador simples, isto é, desde o gerar dos candidatos até ao fim da rotina de *Hidden Number* e respetiva avaliação do seu valor retornado;
8. *Filled* – aqui é invocada a rotina *Filled*, responsável por averiguar se o puzzle já se encontra preenchido; em caso afirmativo, é averiguada a validade da solução e, finalmente, termina a execução do processo;
9. *Mínimos* – neste estado, é invocada a rotina *Mínimos*, responsável por gerar o mapa de mínimos do puzzle e devolver a posição da célula com menor número de candidatos;
10. *Encontra índice vazio* – neste ponto, são percorridas todas as posições do mapa de índices até encontrar a primeira correspondente a um índice vazio, índice a partir do qual será efetuado o número de cópias correspondente ao número de candidatos existentes na célula de mínimo;
11. *Efetua cópias* – neste estado, são efetuadas tantas cópias quantas as possibilidades existentes na célula de mínimo. Estas cópias serão efetuadas para índices a partir do primeiro índice vazio, e, como tal, estes novos índices ocupados serão também sinalizados como ocupados;

12. *Preenche posição de mínimo* – a cada uma das cópias efetuadas será atribuído um candidato da célula de mínimo, preenchido na correspondente célula;
13. *índice++* – o valor do índice atual é incrementado, correspondendo isto a avançar para o puzzle seguinte e o processo é retomado no estado *índice*.
14. *End* – é parado o processo de contagem do contador, sendo armazenado o tempo de execução deste sistema, que corresponde ao tempo de resolução de um puzzle. Em simultâneo, é feito o *display* do resultado da contagem do tempo de resolução, assim como também é apresentada a solução do puzzle (Figura 4.22).

```
 * Solucao

 ** Numero de iteracoes: 9

8 9 4 ¦ 7 6 1 ¦ 2 3 5
5 2 3 ¦ 4 9 8 ¦ 6 1 7
7 1 6 ¦ 2 5 3 ¦ 9 4 8
------+-------+------
9 4 8 ¦ 6 7 2 ¦ 3 5 1
3 6 5 ¦ 1 8 4 ¦ 7 9 2
1 7 2 ¦ 9 3 5 ¦ 8 6 4
------+-------+------
4 5 7 ¦ 3 2 6 ¦ 1 8 9
6 8 9 ¦ 5 1 7 ¦ 4 2 3
2 3 1 ¦ 8 4 9 ¦ 5 7 6

Software execution time:

3.64062 seconds
3640.62093 miliseconds
```

Figura 4.22 - Exemplo de resolução de puzzle com solucionador com tentativa e erro

De notar que ao contrário do que ocorre na resolução do puzzle com o solucionador simples (ver Figura 4.20), neste caso, como se pode observar na Figura 4.22, o número de iterações é diferente de zero. Este valor corresponde ao número do índice do puzzle onde reside a solução, isto é, ao número de cópias efetuadas, ou ainda ao número de suposições total que foi necessário para resolver o puzzle. No caso do solucionador simples, isto não ocorre, uma vez que não existe fase de tentativa e erro.

Note-se, também, que para este solucionador recorreu-se a uma árvore de pesquisa do tipo *BFS* (*Breadth-First Search*). Considerando o primeiro índice, puzzle original, como o nó raiz, são assim testados todos os seus nós filhos, e, consequentemente, filhos dos seus nós descendentes, pela ordem apresentada na Figura 4.23. No que concerne à implementação em *software*, esta corresponde ao incremento do valor do índice atual, tal como descrito no estado *13* já mencionado.

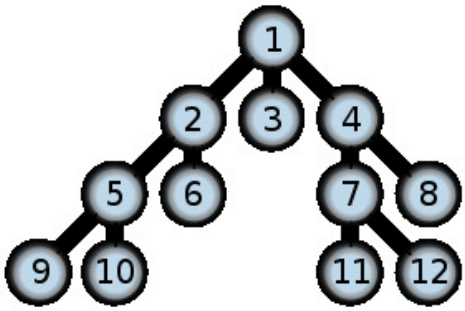


Figura 4.23 – Ordem pela qual os índices são percorridos [41]

4.3 Conclusões

Neste capítulo foram detalhadas as implementações em *software* dos métodos lógicos abordados na secção 2.3.2. A descrição destas baseou-se na apresentação de pseudocódigos juntamente com uma breve explicação dos mesmos. Além destas, foram também introduzidas duas implementações do solucionador: simples e com tentativa e erro.

O solucionador simples recorre apenas a métodos heurísticos para resolução de puzzles, também estes simples e que sejam passíveis de ser totalmente preenchidos apenas com estes. Por sua vez, o solucionador com tentativa e erro, além de englobar os mesmos métodos heurísticos do solucionador simples, possui uma fase adicional capaz de resolver puzzles mais complexos, isto é, que não sejam passíveis de ser resolvidos recorrendo apenas aos métodos heurísticos. Esta fase adicional, de tentativa e erro, é responsável por supor quanto ao valor de algumas células, e testar, assim, as várias possibilidades. Para tal, optou-se pela implementação de uma árvore de pesquisa do tipo *BFS*.

As implementações supracitadas, descritas nos fluxogramas da Figura 4.19 e Figura 4.21 descrevem o modo como os métodos interagem, constituindo, assim, o solucionador de puzzles.

5 Métodos lógicos de resolução - hardware

Neste capítulo, serão descritos os processos implementados em *hardware* dos métodos lógicos de resolução abordados na secção 2.3.2. Estes processos serão implementados através de máquinas de estados finitos, onde o número de estados varia consoante a complexidade de cada rotina a ser implementada.

Uma vez que estes correspondem às funções implementadas em *software*, abordadas no capítulo 4, à parte dos pormenores da sua implementação, as novidades serão apenas introduzidas na secção 5.2, com a descrição dos solucionadores.

5.1.1 Singles

Apesar da implementação em *software* ser de fácil compreensão e concretização, no que toca à implementação em *hardware*, surgem algumas dificuldades.

Em primeiro lugar, a função *Singles* dará lugar a uma máquina de estados finitos, cujos estados se encontram descritos na Figura 5.1. Além disso, cada estado é constituído por um vasto conjunto de operações, por forma a garantir o funcionamento deste método. A interação com este processo é feita através do interface apresentado na Figura 5.2.

De seguida, será então feita a descrição de cada um dos estados constituintes desta máquina, tendo-se, assim, uma máquina constituída por 6 estados, em que a cada um dos estados corresponde o seguinte conjunto de operações:

- **idle** – Este é o estado inicial, responsável por efetuar o *reset* das *flags* de fim do processo de averiguação de existência de candidatos únicos (*found_singles*), assim como também colocar o barramento de endereços de memória (*addr*) na sua posição inicial, isto é, a zeros. O sistema mantém-se no estado *idle* até o sinal *enable* ser ativado e retorna a este somente quando este sinal for desativado, esteja o sistema em que estado estiver.

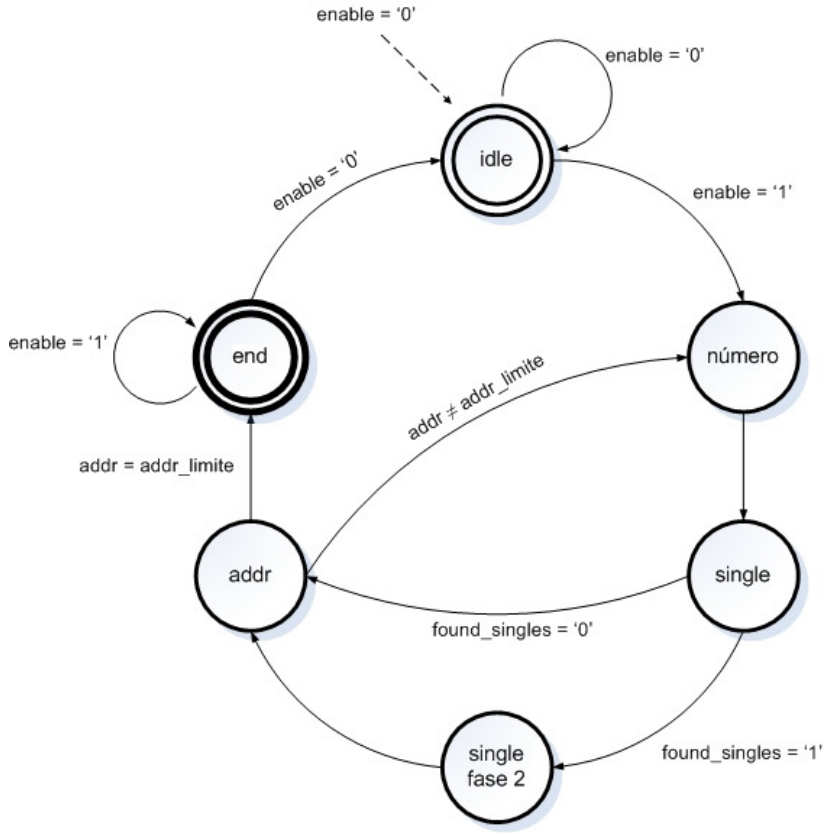


Figura 5.1 – Máquina de estados finitos responsável pela implementação do processo *Singles*.

- **<u>número</u>** – Este estado surge após o sinal *enable* estar ativo. Neste, é efetuada a leitura do primeiro e segundo valores da lista de possibilidades (de acordo com a secção 3.2.2, estes correspondem aos 8 bits mais significativos do barramento de dados da lista) para a posição *addr*, que, imediatamente após o sinal *enable* ser ativo, corresponde à primeira célula do puzzle. Estes dois valores são armazenados e serão processados no estado seguinte, *single*.

- **<u>single</u>** – Neste estado, são avaliados os valores da primeira e segunda possibilidades da lista, lidos no estado anterior. Atendendo ao modo como as possibilidades são armazenadas em memória, já mencionado na secção 3.2.2, se o segundo valor for zero e o primeiro diferente de zero, estamos perante uma situação de candidato único. Se for este o caso, a *flag* responsável pela sinalização da descoberta de um candidato único,

found_singles, é ativa e salta-se então para o estado *single_fase2*. Caso contrário, a *flag* mantém-se inativa e a máquina salta para o estado *addr*.

- **singles_fase2** – Este estado surge no seguimento do estado *single*, após a descoberta de um candidato único. Aqui é apenas efetuado o preenchimento da posição de memória atual (*addr*) com o único candidato possível.

  Importa referir que, neste estado, ao contrário do que acontece em *software*, o valor do candidato não é removido da lista de possibilidades. Esta remoção é efetuada pela rotina *Update* (secção 4.1.4) após o término da rotina *Singles*.

- **addr** – Neste ponto, é efetuado o incremento da posição de memória (*addr*), isto é, a célula do puzzle a ser lida. Após isto, salta-se novamente para o estado *número* por forma a efetuar todo este processo novamente. Este é interrompido quando é atingida a posição *addr_limite*.

- **end** – Sendo este o último estado, é apenas necessário ativar a *flag* de término do processo *Singles* (*eos*) e garantir que o barramento de endereços responsável por determinar a posição da célula do puzzle (*addr*) é colocado na sua posição inicial, isto é, feito o seu *reset*. Tal como anteriormente, o sistema mantém-se neste estado até que o sinal *enable* seja desativado.

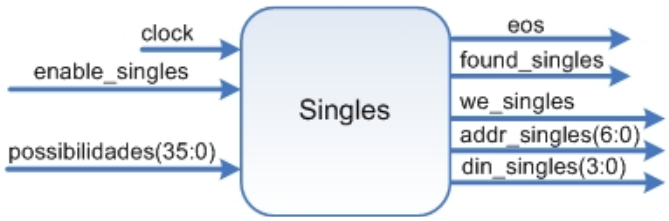


Figura 5.2 – Interface do processo *Singles*

5.1.2 Hidden Number

Da mesma forma que a implementação em *software*, a implementação em *hardware* pode ser dividida em três fases. Estas fases são, então, a verificação por linhas, colunas e caixas, tal e qual como anteriormente.

A máquina de estados correspondente à primeira fase, verificação por linhas, encontra-se descrita na Figura 5.3.

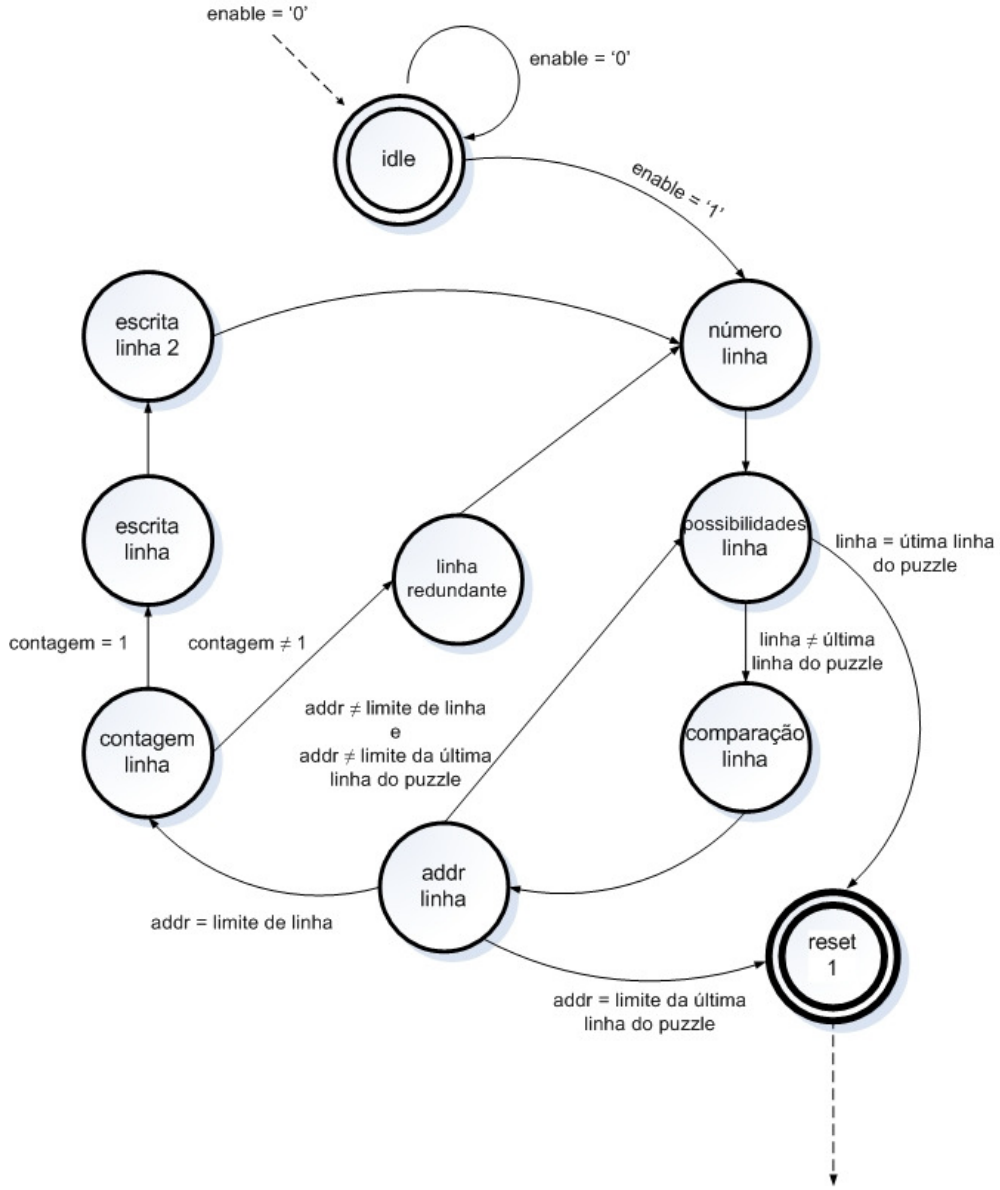


Figura 5.3 – Máquina de estados finitos responsável pela implementação da primeira fase do processo *Hidden Number*

Sendo esta a primeira fase, esta começa no estado *idle*, e termina no estado *reset 1*. Este último faz a ponte de ligação para o início da fase seguinte.

Esta fase é então composta por 10 estados, sendo que a cada um dos estados corresponde o seguinte conjunto de operações:

- **idle** – Este é o estado inicial, quer desta fase, quer da máquina de estados *Hidden Number*. Neste estado, é efetuado o *reset* de várias *flags*, contadores e ainda de alguns sinais de controlo. As *flags* de fim de processo de averiguação de candidatos únicos não diretos (*eoh*) assim como sua descoberta (*found_hidden*) são colocadas no seu valor inicial, isto é, a zeros; os contadores do número de ocorrências por linha, coluna e caixa e do número de caixas já verificadas são igualmente colocados a zero; os sinais de controlo existentes e que necessitam de ser colocados no seu valor inicial são os identificadores de linha, coluna e caixa, aos quais é atribuído o valor 0001_2 (note-se que estes variam entre a posição 1 e 9, ou seja, 0001_2 e 1001_2), e ainda os sinais responsáveis pelo número a ser verificado e barramento de endereços de memória (*addr*) que são colocados a zero.

  O sistema mantém-se neste estado até o sinal *enable* ser ativado, fazendo o sistema avançar para o estado *número linha*, só voltando ao estado *idle* quando o sinal *enable* for desativado.

- **número linha** – Neste estado, não só é definido o número que se irá avaliar, como também a linha onde se irá procurar por candidatos únicos não diretos. Esta definição é feita da seguinte forma: o número vai sendo incrementado desde o seu valor inicial – 1 – até ao seu valor final – 9. Chegado a este valor, o número volta à sua posição inicial e a linha é então incrementada (pois consiste num contador), isto é, após verificar todos os números para uma dada linha, efetua-se o mesmo processo para a linha seguinte.

- **possibilidades linha** – Neste estado, é lido o barramento de dados da lista de possibilidades para a posição *addr*. Por outras palavras, são lidos todos os candidatos possíveis para a célula dada pela posição *addr*. No entanto, esta leitura só é efetuada caso não tenha sido atingido o limite de linhas do puzzle.

  Assim sendo, quando o valor resultante do contador de linhas atingir o limite, o processo de procura por candidatos únicos não diretos por linhas terminou, pelo que o próximo estado será o estado *reset 1*. Caso contrário, os candidatos são lidos como já mencionado, e o sistema avança para o estado *comparação linha*.

- **comparação linha** – Neste estado, os candidatos resultantes da leitura efetuada no estado anterior são comparados com o valor do número que se encontra a ser verificado no momento (definido no estado *número linha*). Se estes forem iguais, é então incrementado o contador do número de ocorrências e o valor da posição *addr* (endereço) é guardado num sinal auxiliar.

  Esta salvaguarda do valor da posição *addr* permite, independentemente do resultado da contagem de ocorrências ser unitário ou não, saber a última posição em que o incremento ocorreu. Isto é vantajoso apenas para o caso em que a contagem foi unitária, uma vez que, quando se for efetuar a escrita do candidato na célula, já se sabe a posição desta, não havendo necessidade de a voltar a procurar.

- **addr linha** – Neste estado, é efetuado o incremento da posição de memória (*addr*) para a qual se pretende efetuar a leitura dos seus candidatos. É então neste ponto que se efetua a mudança de célula do puzzle que se encontra a ser lida. Além disto, se o endereço atual corresponder ao último endereço de uma das 9 linhas, então, uma vez atingido o final da linha, o sistema avança para o estado *contagem linha*, por forma a processar o resultado da contagem na presente linha. Se o endereço atual não corresponder ao último endereço de uma das 9 linhas, então é verificado se este corresponde ao limite de linhas do puzzle e, em caso afirmativo, a verificação termina com o avanço do sistema para o estado *reset 1*, senão o sistema volta ao estado *possibilidades linha*, para avaliar as possibilidades de outra célula. Desta forma garante-se que, uma vez atingida a última posição de uma dada linha, é processado o resultado da contagem de um dado número nessa mesma linha. Caso contrário, é avaliada a próxima célula dessa mesma linha. Excetua-se o caso em que foi atingido o limite de linhas do puzzle, que leva ao término desta fase, uma vez que não existem mais linhas para serem avaliadas.

- **contagem linha** – Este estado é responsável por ler o resultado da contagem do número de ocorrências para uma dada linha. Se o resultado da contagem for então unitário, estamos perante uma situação de candidato único não direto, pelo que deve ser sinalizada a sua descoberta, isto é, a *flag found_hidden* é ativada, é feito *reset* ao contador de ocorrências, e o endereço, salvaguardado previamente no estado

comparação linha, é agora carregado para o barramento de endereços de posição, uma vez que este corresponde à célula que possui o candidato. De seguida, o sistema avança para o estado *escrita linha*. No entanto, o endereço tem que ser carregado para o barramento de endereços neste estado, por forma a que, quando se efetuar a escrita (sinal *we* ativo), já o barramento tem o valor do endereço atualizado.

Caso o resultado da contagem não seja unitário, é feito também o *reset* do valor do contador de ocorrências e o sistema salta para o estado *linha redundante*, que será explicado mais à frente.

Além disto, este estado tem uma outra função de extrema importância. Se o fim de uma linha tiver sido atingido e estivermos perante o número 9, significa isto que o próximo passo é recomeçar o processo no início da linha seguinte no número 1. Então, perante esta situação, é avaliada qual a linha atual e salvaguardado o endereço de início da linha seguinte. Caso contrário, é apenas avaliada a linha atual e carregado o seu endereço de início. No entanto, excetua-se o caso da última linha que, perante o número 9, não necessita ser carregado nenhum endereço de início de linha, uma vez que esta fase terminou. Desta forma, aquando de um novo incremento é atingido o limite de linhas, o sistema salta para o estado *reset 1*.

Este passo é extremamente importante, pois após este estado, o sistema salta para o estado de *escrita* ou de *linha redundante*, onde este endereço salvaguardado é carregado para o barramento de endereços de posição de memória. Desta forma, quando o sistema chegar ao estado *número linha*, já tem o endereço correto da próxima posição a ser averiguada. Não há necessidade de, neste estado, definir o próximo número a ser averiguado, uma vez que o estado *número linha* já o faz.

Como nota adicional para clarificar um pouco mais esta definição dos endereços, é de referir que o estado *addr linha* apenas verifica se o endereço corresponde ao fim de uma linha e, se for o caso, é então necessário efetuar a contagem do número de ocorrências nessa mesma linha, ao passo que, no estado *contagem linha*, é definido o próximo endereço, a partir do qual deve recomeçar o processo. Não nos devemos esquecer que este varia consoante tenhamos verificado todos os números para uma linha ou não; por outras palavras, se todos os 9 números já tiverem sido verificados para uma linha, existe a necessidade de avançar para a linha seguinte, caso contrário, existe necessidade de voltar ao início da linha atual. Existe, portanto, uma gestão de

atribuição de endereços ao barramento que varia consoante se esteja a escrever dados em memória, ou, simplesmente, a efetuar a verificação das ocorrências de candidatos únicos não diretos.

- **linha redundante** – Este estado consiste apenas num ciclo de espera por atualização de um só sinal. Como mencionado no estado anterior, existe a salvaguarda do próximo endereço a ser atribuído ao barramento. No entanto, este não é logo atribuído, dado que a atribuição do endereço e a salvaguarda ocorrem ao mesmo tempo, isto é, no mesmo flanco do *clock*. Assim, o endereço salvaguardado seria o correto, mas o atribuído seria o seu valor anterior.

  Existe então a necessidade de separar este processo em duas fases. A primeira é feita no estado anterior, onde o endereço é salvaguardado, tendo este tempo de atualizar o seu valor, e a segunda fase neste estado, onde o endereço é então atribuído ao barramento de endereços.

  Desta forma, quando o sistema chegar novamente ao estado *número linha*, já o barramento está atualizado com o endereço correto para recomeçar a verificação.

- **escrita linha** – Após o barramento de endereços ter sido atualizado com a posição correta para efetuar a escrita, resta apenas atribuir o número a ser escrito (que é o número atual com o qual a verificação por linhas foi feita) ao barramento de dados assim como também ativar a *flag* de *write enable* (*we*) da memória.

  De notar que, neste ponto, o endereço carregado para o barramento de endereços foi feito no estado *comparação linha*, e, como tal, é o endereço correspondente à posição da célula que será escrita. No entanto, existe também a necessidade de, após a escrita, antes de saltar novamente para o estado *número linha*, carregar o endereço salvaguardado no estado *contagem linha* para o barramento de endereços de posição. Este carregamento é feito no estado seguinte, uma vez que, neste estado, é necessário manter no barramento o endereço da posição a ser escrita.

- **escrita linha 2** – Tal como mencionado, este estado apenas carrega o barramento de endereços de posição de memória com o endereço salvaguardado no estado *contagem*

linha, que corresponde à próxima posição a ser avaliada. De seguida, o sistema avança novamente para o estado *número linha*.

- **reset 1** – Este é o estado final desta primeira fase e que serve de ponte para a fase seguinte. Uma vez que o estado *idle* corresponde ao início da máquina de estados finitos na sua totalidade, o mesmo pode ser equiparado com o estado *idle* da segunda fase, pois, tal como este, é efetuado o *reset* de alguns sinais de controlo e contadores. É então feito o *reset* ao contador de ocorrências e ao número, colocando-os a zero, assim como também ao barramento de endereços de memória (*addr*) e endereços salvaguardados nos estados anteriores. Com este estado, o sistema é colocado nas condições iniciais, que é imperativo para o começo da próxima e de qualquer uma das fases. Excetua-se a *flag* de descoberta de candidatos únicos não diretos (*found_hidden*), à qual não é feito nenhum *reset* até ao final do processo na sua totalidade.

Terminada a primeira fase, verificação por linhas, começa então a segunda fase que consiste na verificação por colunas. As diferenças são muito poucas comparativamente com a fase anterior, sendo que a grande diferença reside no incremento da posição *addr*, e na definição dos endereços de começo e fim de coluna.

A máquina de estados finitos correspondente a esta fase encontra-se descrita na Figura 5.4.

De acordo com esta, excetuando-se o estado *reset 1*, esta fase é composta por 9 estados. Estes apesar de muito idênticos aos da fase anterior, apresentam algumas diferenças. A cada estado correspondem as seguintes operações:

- **número coluna** – Neste estado, é definido o número que se irá avaliar e também a coluna onde se irão procurar candidatos únicos não diretos. Esta definição é feita exatamente da mesma maneira que o estado *número linha* da fase anterior, ou seja, o número parte do seu valor inicial e vai sendo incrementado até ao valor final – 9. Neste ponto, o número volta ao seu valor inicial e a coluna é incrementada para o seu valor seguinte.

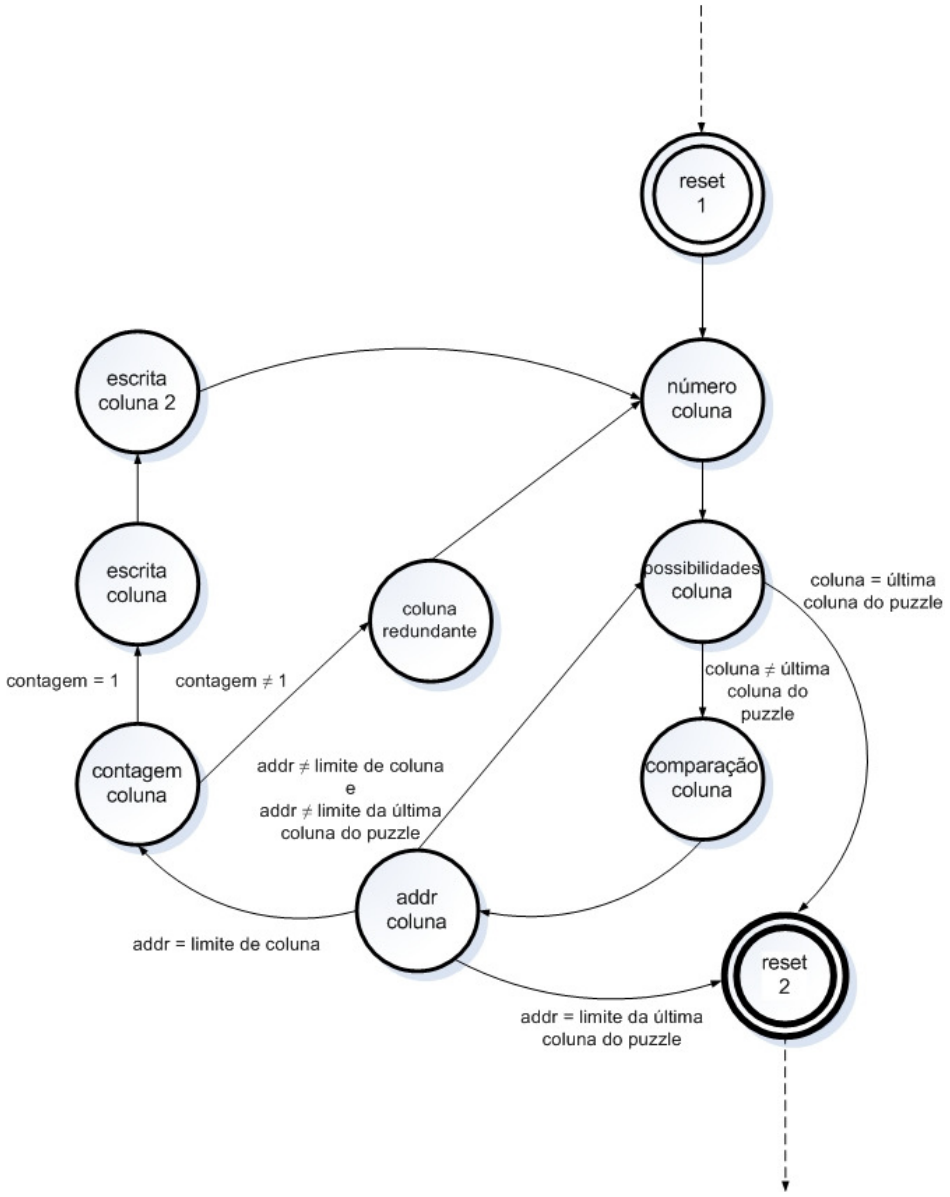


Figura 5.4 – Máquina de estados finitos responsável pela implementação da segunda fase do processo *Hidden Number*

- **possibilidades coluna** – Neste estado, é lido o barramento de dados da lista de possibilidades para a posição *addr*, isto é, são lidos todos os candidatos possíveis para a célula correspondente a esta posição, como já mencionado. Tal como anteriormente, esta leitura só é efetuada caso não tenho sido atingido o limite de colunas, que também é imposto pela décima coluna do puzzle. Se este tiver sido atingido, então não existem mais colunas a serem verificadas e o próximo estado é o estado *reset 2*.

- **<u>comparação coluna</u>** – Neste estado, os candidatos resultantes da leitura efetuada no estado anterior são comparados com o valor do número que se encontra a ser verificado no momento (definido no estado número coluna). Se estes forem iguais, é então incrementado o contador do número de ocorrências e o valor da posição *addr* é guardado num sinal auxiliar. Esta salvaguarda é feita pelo mesmo motivo que a salvaguarda no estado *comparação linha* da fase anterior.

- **<u>addr coluna</u>** – Neste estado, é efetuado o incremento da posição de memória (*addr*) para a qual se pretende efetuar a leitura dos seus candidatos. Este incremento, ao contrário do que ocorre no estado *addr linha* que é unitário, nesta fase, é de 9 números, ou seja, 9 posições de memória. Isto ocorre pois a posição correspondente a uma dada célula de uma coluna, quando incrementada de 9 posições, corresponde à posição da célula seguinte da mesma coluna (Figura 5.5).

	1	2	3	4	5	6	7	8	9
A						2			7
B				9		5	4	3	8
C	8		9				9		
D		7					3		2
E				1		+0	+1	+2	+3
F	+4	+5	+6	+7	+8	+9		9	
G							6		5
H	1	2	6	8		9			
I	4			3					

Figura 5.5 – Exemplo ilustrativo da soma de 9 unidades para a verificação por coluna

Além disto, se o endereço atual corresponder ao último endereço de cada uma das 9 colunas, isto é, atingida a última posição de cada coluna, deve então ser avaliado o resultado da contagem e, portanto, o sistema avança para o estado *contagem coluna*. Caso contrário, continuam a ser avaliadas as possibilidades de cada posição da coluna, ou seja, o sistema salta para o estado *possibilidades coluna*, até que se atinja o limite de colunas, e, nesse caso, não existem mais colunas a ser avaliadas e, como tal, esta fase termina avançando para o estado *reset 2*.

- **contagem coluna** – Tal como no caso do estado *contagem linha*, este estado é também de extrema importância pois o bom funcionamento do processo depende dele. Este é, por conseguinte, responsável por ler o resultado da contagem do número de ocorrências para uma dada coluna e averiguar se o resultado é então unitário. Se assim for, a *flag found_hidden*, se ainda não estiver ativa, é ativada neste ponto e é feito o *reset* ao contador de ocorrências ao mesmo tempo que o endereço, salvaguardado previamente no estado *comparação coluna*, é carregado para o barramento de endereços de posição e o sistema avança para o estado *escrita coluna*; senão, é feito também o *reset* ao contador de ocorrências e o sistema salta para o estado *coluna redundante*.

  Da mesma forma que no estado *contagem linha*, o endereço tem que ser carregado para o barramento de endereços neste estado, de modo a que, quando se efetuar a escrita, já o barramento tem o valor do endereço atualizado, isto é, já se encontra na posição correta onde se deverá escrever o valor do candidato.

  Além destas operações, é também averiguado se a posição atual da célula é a última posição da coluna. Se assim for, existe a necessidade de definir o próximo endereço de princípio de coluna. Este pode ser definido como o início da coluna atual ou o início da coluna seguinte, caso o número seja inferior ou igual a 9 respetivamente. Para a situação em que a posição corresponde à última posição da última coluna e número 9 atualmente a ser verificado, não existe necessidade de definir nenhum próximo endereço, pois o sistema terminou a verificação por colunas.

  A definição do próximo endereço de princípio de coluna corresponde apenas à salvaguarda desse mesmo endereço, que será carregado para o barramento de posição de memória num estado seguinte.

- **coluna redundante** – Este estado, tal como o linha redundante, consiste apenas num ciclo de espera por atualização de um sinal apenas. Este sinal é o endereço salvaguardado no estado anterior que é neste estado atribuído ao barramento. A razão de existir este estado redundante é a mesma mencionada no estado *linha redundante*.

- **escrita coluna** – Neste estado é então efetuada a escrita do valor do candidato único não direto para a posição de memória salvaguardada no estado *comparação coluna*.

Assim, o candidato a ser escrito em memória é carregado para o barramento de dados, ao mesmo tempo que o sinal de *write enable* (*we*) é ativo.

De notar que, como anteriormente, o endereço atribuído ao barramento de endereços corresponde à posição da célula que será preenchida, apesar de existir a necessidade de, após a escrita, antes de saltar novamente para o estado *número coluna*, carregar o endereço salvaguardado para o barramento. Esta ação é feita no estado seguinte, uma vez que, neste estado é necessário manter o barramento de endereços com a posição da célula a ser preenchida, pois só assim se garante o preenchimento da posição correta.

- **escrita coluna 2** – No seguimento do estado anterior, este estado é apenas responsável por carregar, para o barramento de endereços, o endereço salvaguardado que corresponde à próxima posição a ser avaliada, e, como tal, o sistema avança para o estado *número coluna*.

- **reset 2** – Este estado é exatamente igual ao estado *reset 1*. No entanto, este marca o fim da segunda fase deste processo, atuando como ponte entre a verificação por colunas e a verificação por caixas.

Após a segunda fase, verificação por colunas, começa então a terceira e última fase, que consiste na verificação por caixas. Conceptualmente, esta fase é muito idêntica às anteriores, a sua implementação, porém, difere bastante.

O diagrama de estados correspondente a esta fase encontra-se representado na Figura 5.6. De acordo com este, esta fase é composta 9 estados, excetuando-se o estado *reset 2*, já contabilizado na fase anterior.

Para a implementação desta fase, é necessário recorrer ao mapa de caixas, apresentado na secção 3.2.3, por forma a identificar as células constituintes de cada uma das caixas. Assim, é possível através de um identificador (*idCaixa*), saber se uma dada célula corresponde à caixa em que se pretende averiguar a presença de candidatos únicos não diretos ou se pertence a uma outra caixa. Este identificador permite então associar as células às diferentes caixas constituintes do puzzle.

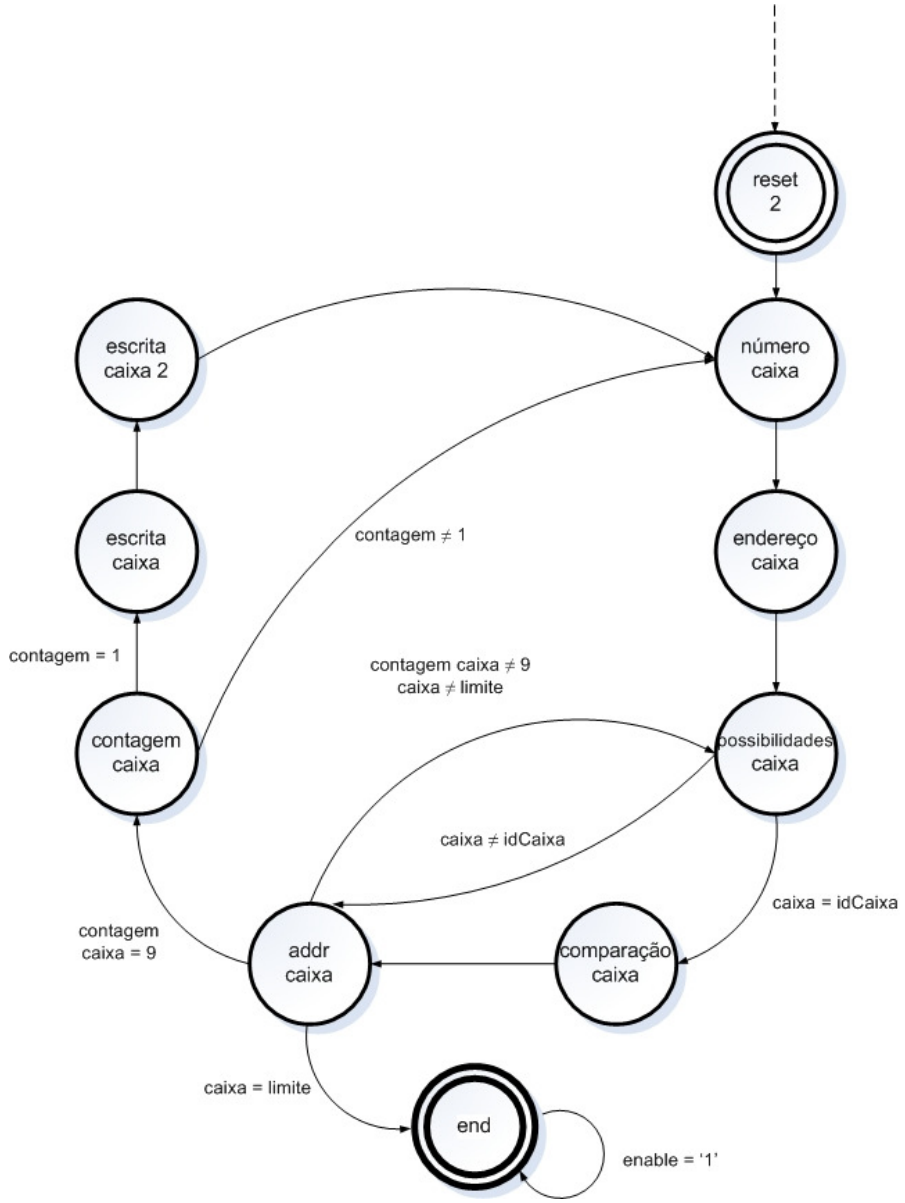


Figura 5.6 – Máquina de estados finitos responsável pela implementação da terceira fase do processo *Hidden Number*

Uma outra ferramenta necessária será um contador que contabilize o número de células já verificadas por caixa. É sabido que cada caixa é constituída por 9 células, ou seja, após verificar 9 células duma dada caixa, não existe a necessidade de procurar mais células da mesma caixa, pois estas não existem. Este contador é necessário dada a natureza da implementação desta terceira fase.

Seguidamente, são apresentadas as operações constituintes de cada estado.

- **número caixa** – Este estado é responsável por definir o número que se irá avaliar, como também a caixa onde se irá procurar por candidatos únicos não diretos. Esta definição é feita da seguinte forma: o número vai sendo incrementado até ao seu valor final – 9. Uma vez atingido o seu valor final, este é colocado novamente no seu valor inicial – 1 – e é feito o incremento da caixa que se pretende avaliar de seguida, ou seja, após verificar todos os números para uma dada caixa, avança-se para a caixa seguinte.

- **endereço caixa** – Neste estado, é feito o ajuste da primeira posição da caixa definida no estado anterior, isto é, consoante a caixa que se pretende avaliar, é então atribuído, ao barramento de endereços (*addr*), o endereço correspondente à posição da primeira célula da respetiva caixa. Este estado encontra-se separado do anterior, uma vez que existe necessidade de uma prévia atualização do sinal responsável por definir a caixa a ser avaliada.

- **possibilidades caixa** – Neste estado, é feita a leitura dos valores do barramento de dados da lista de possibilidades para a posição *addr* já definida. No entanto, esta leitura só é efetuada se o identificador desta mesma caixa (na verdade da posição *addr*) corresponder à caixa que se pretende avaliar, isto é, só será efetuada a leitura dos valores se a célula pertencer à caixa que se pretende avaliar. Se tal ocorrer, além da leitura, é incrementado o contador do número de células já verificadas para a caixa (estado *contagem caixa*, Figura 5.6) e o sistema avança para o estado *comparação caixa*. Se o identificador da caixa para a posição *addr* não corresponder à caixa que se pretende avaliar, então o sistema avança para o estado *addr caixa*, que será responsável por mudar a posição *addr* das células.

- **comparação caixa** – Neste estado, tal como no das fases anteriores, os candidatos resultantes da leitura efetuada no estado anterior são comparados com o valor do número previamente definido no estado *número caixa*. Se estes forem iguais, é então incrementado o contador do número de ocorrências e o valor da posição da célula (*addr*) é guardado num sinal auxiliar.

- **addr caixa** – Neste estado, é efetuado o incremento da posição de memória *addr*. Este incremento é unitário e, por isso, é necessário o mapa de caixas. O que ocorre nesta fase é o seguinte: as células são todas percorridas uma a uma e apenas as pertencentes à caixa que se deseja verificar é que são submetidas a leitura dos seus candidatos. A implementação foi assim feita uma vez que cada caixa possui 3 células em 3 linhas diferentes e 3 células em 3 colunas diferentes, e seria bastante complicado, de outra forma, definir o próximo endereço a ser avaliado.

  Além desta operação, é verificado se já foram avaliadas todas as 9 células constituintes da caixa e, em caso afirmativo, o sistema avança para o estado *contagem caixa*, para processar o resultado da contagem do número de ocorrências. Caso contrário, se a caixa a ser avaliada corresponder ao limite (*idCaixa* = 10), significa que o processo de verificação por caixas terminou e, portanto, o sistema avança para o estado *end*.

  Se nenhuma destas duas situações ocorrerem, isto é, se não tiverem sido ainda avaliadas todas as células de uma caixa e não tiverem sido processadas todas as caixas do puzzle, uma vez incrementado o endereço, o sistema regressa ao estado *possibilidades caixa*.

- **contagem caixa** – Este estado é responsável por ler o resultado da contagem do número de ocorrências para uma dada caixa. Se o resultado da contagem for unitário, estamos perante uma situação de candidato único não direto, pelo que deve ser sinalizada a sua descoberta, isto é, a *flag found_hidden* é ativada, é feito *reset* ao contador de ocorrências. Em simultâneo, é salvaguardada a posição atual do barramento de endereços *addr*, é atribuído, a este, o endereço salvaguardado no estado *comparação caixa* e o sistema avança para o estado *escrita caixa*. Caso o resultado da contagem não seja unitário, é feito o *reset* do contador de ocorrências e o sistema volta ao estado *número caixa*.

  Além destas operações, é também feito o *reset* do contador do número de células já verificadas por caixa, por forma a que se possam verificar novamente as mesmas 9 células, procurando agora por outro candidato.

  De notar que neste ponto não é necessário recorrer a um ciclo redundante como acontecia anteriormente. Isto deve-se ao facto de o estado *endereço caixa* se encarregar

de definir a primeira posição da caixa pela qual se deve começar a verificação. Tal não acontecia anteriormente, uma vez que o estado *contagem linha/coluna redundante* era necessário para atribuir essa mesma posição ao barramento de endereços, de modo a que, quando atingido o estado *número linha/coluna*, bastasse apenas definir o número a ser verificado e efetuar a leitura das possibilidades. Simplificando, o estado redundante é aqui substituído pelo estado *addr caixa*.

- **escrita caixa** – Após o barramento de endereços ter sido atualizado com o endereço da posição da célula correta para efetuar a escrita, resta apenas atribuir ao barramento de dados o número a ser escrito em memória, assim como ativar a *flag* de *write enable* (*we*) da memória.

- **escrita caixa 2** – Neste estado, é recuperado o endereço salvaguardado no estado *contagem caixa*, que corresponde à posição *addr* em que se encontrava o sistema nesse ponto, ou seja, após alterar a posição *addr* para a posição da célula a ser escrita, volta-se a reposicionar o sistema na célula em que se encontrava antes da escrita.

  Esta recuperação do endereço salvaguardado não é de todo necessária, uma vez que, após o sistema saltar para o estado *número caixa* e seguidamente para o estado *endereço caixa*, é novamente definida a posição inicial da caixa, a partir da qual será iniciada nova verificação. No entanto, na primeira versão da implementação desta máquina de estados, esta possuía o ciclo redundante já referido, uma vez que existia a necessidade de recuperar o endereço. Por forma a manter a boa prática de recuperar a posição do barramento de endereços após a sua alteração, manteve-se este estado.

- **end** – Este estado consiste não só no último estado desta fase como também no último estado da máquina de estados finitos *Hidden Number*. Como tal, é necessário ativar a *flag* de término do processo *Hidden Number* (*eoh*) e garantir que o barramento de endereços responsável por determinar a posição da célula do puzzle (*addr*) é colocado na sua posição inicial, isto é, é efetuado o *reset* da posição *addr*. O sistema mantém-se neste estado até que o sinal *enable* seja desativado.

Apresentadas cada uma das três fases deste processo, as mesmas em conjunto constituem a máquina de estados finitos *Hidden Number* na sua totalidade que pode ser simplificada pelo diagrama de estados da Figura 5.7, onde a cada fase corresponde, respetivamente, a máquina de estados finitos já apresentada. No total, esta máquina é constituída por 28 estados, e o seu interface é feito de acordo com a Figura 5.8.

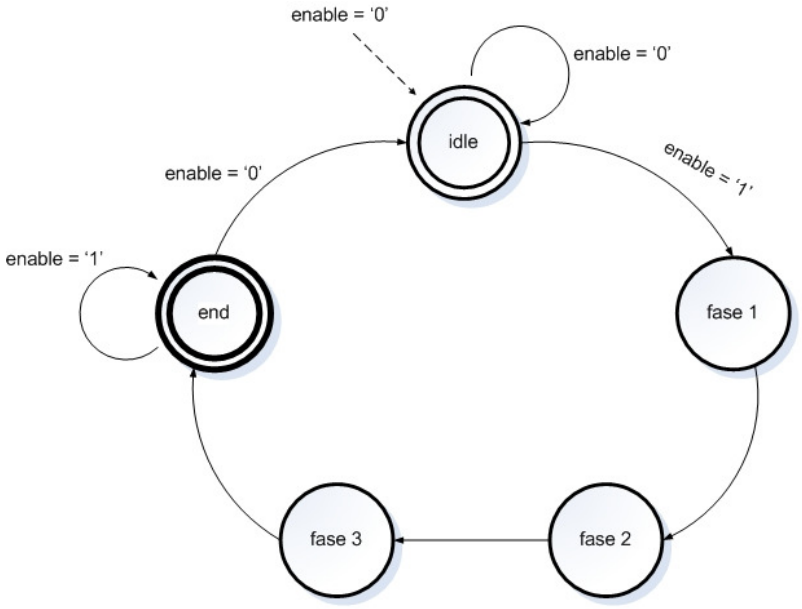


Figura 5.7 – Máquina de estados finitos do processo *Hidden Number* na sua totalidade

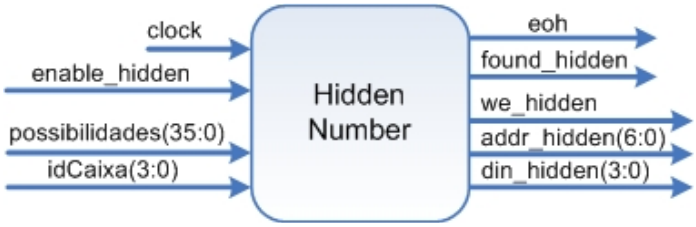


Figura 5.8 – Interface do processo *Hidden Number*

5.1.3 Validate

A função *Validate* dará agora lugar a uma máquina de estados finitos constituída por 10 estados, tal como representado na Figura 5.9.

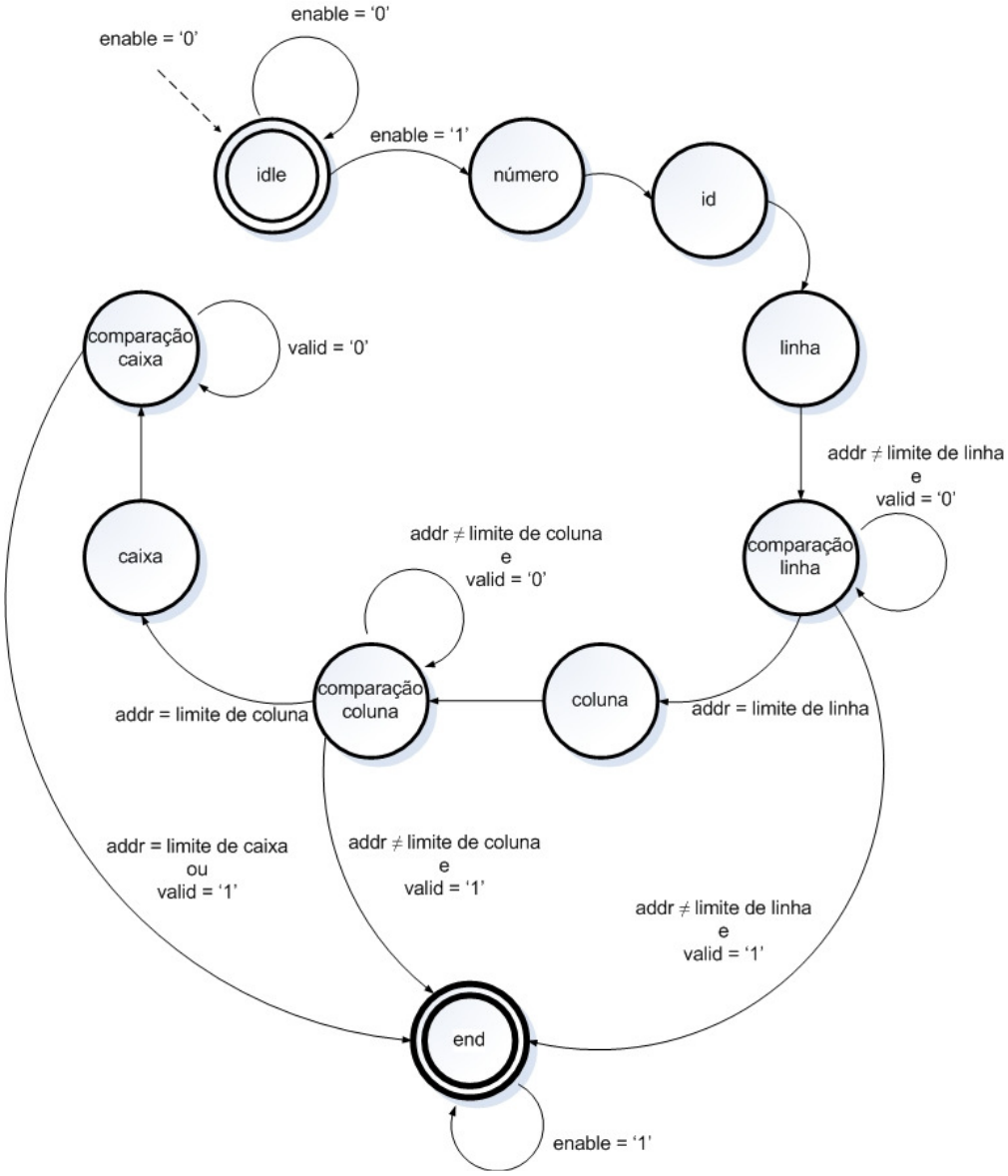


Figura 5.9 – Máquina de estados finitos responsável pela implementação do processo *Validate*

Para a implementação desta máquina, o bloco responsável pela mesma necessitará de interagir com o mapa de colunas e de caixas, já apresentados na secção 3.2.3, por forma a identificar as células constituintes de cada uma das colunas e caixas, através de um identificador. Esta interação é feita de acordo com o interface apresentado na Figura 5.11.

De seguida, será então feita a descrição de cada um dos estados constituintes desta máquina.

- **idle** – Este é o estado inicial, responsável por efetuar o *reset* das *flags* de fim de processo de validação (*eov*) e de número válido (*valid*), isto é, colocar o seu valor a zero. A *flag valid* tem como valor inicial zero e mantém-se assim até ao final do processo caso o número seja válido, isto é, quando colocado numa célula específica, esta fique preenchida de acordo com as regras do puzzle; caso contrário, esta *flag* é ativada, isto é, colocada no nível alto, indicando que o número não é válido. É importante referir que a *flag* é ativada quando o número não é válido, uma vez que os sinais por omissão se encontram a zero e, sendo assim, torna-se fulcral sinalizar uma operação que obrigue o processo a terminar. Uma vez que essa operação corresponde ao momento em que se constata que o número não é válido, é neste momento que deve ser feita a sinalização, invertendo o valor da *flag*, passando esta de '0' a '1'. Esta definição pode ser facilmente compreendida se pensarmos que a *flag* se chama "*not valid*", ao invés de "*valid*", uma vez que é ativada apenas quando o número não é válido.

  O sistema mantém-se neste estado até o sinal *enable* ser ativado, fazendo o sistema avançar para o estado *número*. De notar que o sistema só volta a este estado quando o sinal *enable* for desativado, esteja o sistema em que estado estiver. É importante referir que, quando o sinal *enable* for desativado, independentemente do estado atual do sistema, este deve retornar a este estado *idle*, para que as condições iniciais desta máquina sejam repostas. Esta prática evitará problemas relacionados com sinais que necessitam de assumir um valor inicial cada vez que esta máquina é solicitada para que esta funcione corretamente.

- **número** – Este estado é responsável por efetuar a leitura do número, assim com também da posição deste. Tal como acontecia na implementação em *software*, em que a posição da célula era passada como parâmetro de entrada da função, aqui esta também é passada como parâmetro de entrada, bem como também o número a ser verificado para a dada posição. Assim, neste estado, é lido um valor de 7 bits que corresponde ao endereço de memória da posição que se quer validar, assim como também é lido um valor de 4 bits, sendo este o número que se deseja validar para a mesma posição (Figura 5.11).

De notar que em *software*, a célula já teria sido previamente preenchida aquando da sua validação, ao passo que, na implementação em *hardware*, a célula tem que se encontrar necessariamente livre para se poder testar o número desejado e averiguar se este é um número válido para a respetiva célula, cuidado este deixado a cabo de uma entidade externa.

No entanto, as implementações são idênticas uma vez que é esta entidade externa que define o número a ser validado, isto é, em *software* esta entidade preenche a célula desejada com o número desejado e efetua a validação, fazendo variar o número, se assim for o caso; em *hardware*, a entidade externa fornece o valor do número desejado como parâmetro de entrada, colocando-o no barramento de dados propositado para o caso. Em termos de *hardware*, esta implementação apresenta a vantagem de não ser necessário para cada posição da linha, coluna ou caixa, verificar se é a mesma que estamos a validar ou não, evitando assim possíveis atrasos na leitura dos valores que possam comprometer o desempenho desta máquina.

- **id** – Neste estado, recorre-se aos mapas de coluna e de caixa por forma a identificar qual a coluna e a caixa a que a posição da célula definida no ponto anterior corresponde. Estes valores dos identificadores de coluna e de caixa são então armazenados para, mais tarde, poderem ser lidos, uma vez que esta identificação será útil, posteriormente, na definição da posição de início de verificação, tal como será descrito mais à frente. O processo de identificação de linha é bem mais fácil pelo que não necessita de mapa de linhas algum.

  Importa também referir que este estado tem necessariamente que estar separado do estado *número*, uma vez que só após a leitura da posição da célula, efetuada no estado anterior, é que esta posição é carregada para o barramento de endereços de memória, o mesmo que é partilhado com os mapas de coluna e de caixa. É, portanto, necessário existir um ciclo de espera, de modo a que os valores dos identificadores lidos sejam os corretos.

- **linha** – Neste estado, consoante o valor da posição da célula (*addr*), é determinada a linha em que a célula se encontra, definindo assim a posição de início de linha e de

limite. Estes dois valores serão usados como posição de início de validação da linha e posição de fim de validação da linha.

- **comparação linha** – Neste estado, é feita a leitura do valor da célula atual, endereçada pelo valor *addr*, que é comparado com o número lido no estado *número*. Se igual, é ativada a *flag valid*, responsável por indicar se o número é ou não válido quando colocado em determinada célula. No entanto, esta comparação só é feita se esta *flag valid* se encontrar previamente desativada, pois, caso contrário, o número já não é válido e não será necessário proceder com a verificação, terminando assim no estado *end*. Sendo o número válido (*valid* desativada), é também incrementada a posição *addr*, e o sistema salta novamente para o mesmo estado *comparação linha*, por forma a verificar a célula seguinte.

  Quando for atingido o limite da linha, já definido no estado anterior, significa que foram verificadas todas as células da linha e, então, pode-se avançar para a verificação da coluna, saltando para o estado *coluna*.

- **coluna** – Neste estado, consoante o identificador de coluna salvaguardado no estado *id*, é determinado o endereço de início de coluna e o de limite. Estes dois valores serão usados como a posição de início de validação da coluna e posição de fim de validação da coluna.

- **comparação coluna** – Neste estado, é feita a leitura do valor da célula atual, endereçada pelo valor *addr*, que é comparado com o número lido no estado *número*. Este estado é cópia integral do estado *comparação linha*, tendo este as mesmas operações. A principal diferença reside no facto de o incremento do valor *addr* não ser unitário, mas sim de 9 unidades, por forma a verificar a célula seguinte da mesma coluna (tal pode ser observado na Figura 5.5).

  Atingido o limite da coluna, já definido no estado *coluna*, todas as células da coluna foram verificadas e o sistema avança para o estado *caixa*.

- **caixa** – Neste ponto, é definido o começo da caixa, isto é, a primeira posição da caixa. Esta definição é feita com base no identificador de caixa salvaguardado no estado *id*.

Além deste, também são definidos três limites distintos, uma vez que a caixa será tratada como três linhas distintas e, como tal, é necessário definir limite para cada uma destas três linhas.

- **<u>comparação caixa</u>** – Neste estado, é feita a leitura do valor da célula atual. Através do estado anterior, é possível determinar a primeira posição da célula desta caixa, no entanto, as posições seguintes serão determinadas com base nos três limites impostos no estado anterior. Assim, se a *flag valid* se encontrar desativada, é averiguado se a posição *addr* atual corresponde ao primeiro limite, e, em caso afirmativo, o *addr* é incrementado de 7 unidades, por forma a ler a próxima célula da caixa. O mesmo ocorre se a posição *addr* corresponder ao segundo limite (ver exemplo da Figura 5.10). Caso não corresponda a nenhum destes dois limites, então a posição *addr* sofre um incremento unitário, ao mesmo tempo que é lido o valor do número residente na célula atual e comparado com o número lido no estado *número*. Se igual, é ativada a *flag valid*, responsável por indicar se o número é ou não válido, senão a *flag* mantém-se inativa.

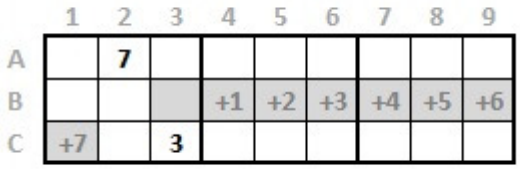


Figura 5.10 – Exemplo ilustrativo da soma de 7 unidades para a verificação por caixa

A verificação da caixa mantém-se até ser atingido o último limite, terceiro limite, não havendo mais células para verificar ou caso o número deixe de ser válido.

- **<u>end</u>** – Sendo este o último estado, este é responsável por ativar a *flag* de término do processo de *Validate* (*eov*) e garantir que o barramento de endereços responsável por determinar a posição da célula do puzzle (*addr*) é colocado na sua posição inicial, isto é, feito o seu *reset*. O sistema mantém-se neste estado até que o sinal *enable* seja desativado.

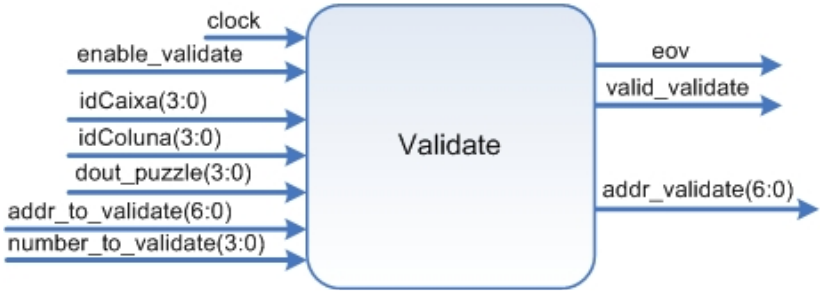


Figura 5.11 – Interface do processo *Validate*

5.1.4 Lista de possibilidades ou candidatos

Em termos de *hardware*, como se sabe, a implementação será bastante diferente, uma vez que a função dará agora lugar a uma máquina de estados finitos. O diagrama de estados constituintes desta máquina pode ser observado na Figura 5.12.

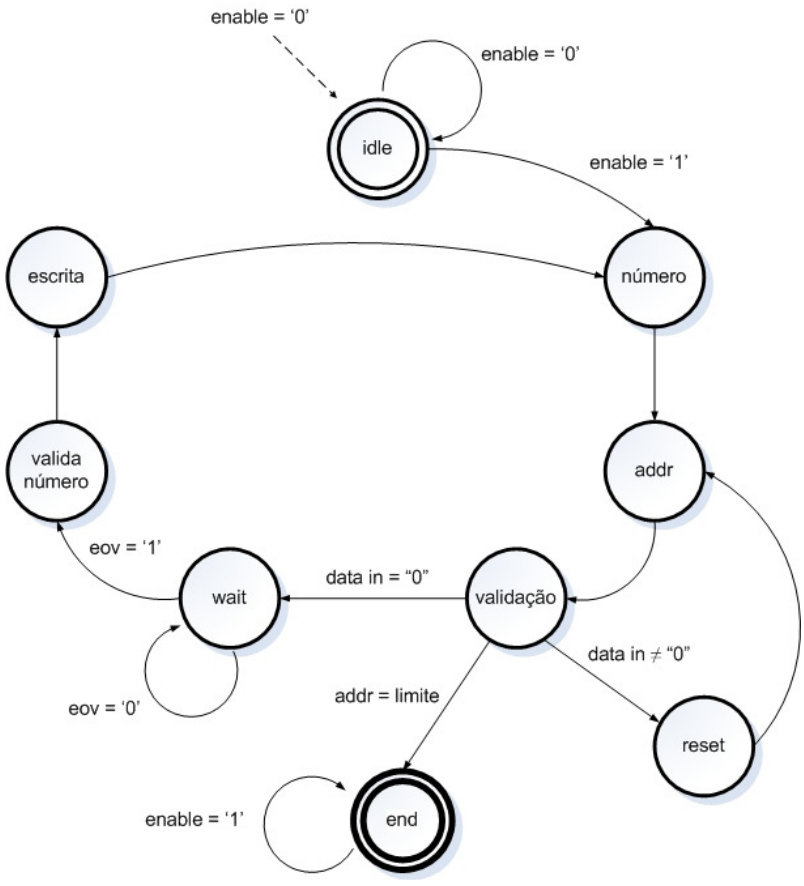


Figura 5.12 – Máquina de estados finitos responsável pela implementação do processo gerador da lista de possibilidades

Da Figura 5.12, constata-se que esta máquina é constituída por 9 estados. Cada um destes estados está associado um conjunto de operações que garantem o bom funcionamento deste processo. De seguida são descritas estas operações.

- **idle** – Este é o estado inicial responsável por efetuar o *reset* das *flags* de fim de processo de criação da lista de possibilidades (*eopl – end of possibilities list*) e de uma *flag* auxiliar usada para sinalizar o fim de criação da lista de possibilidades para cada célula individual. É também feito o *reset* do número, isto é, definido como zero.

  O sistema mantém-se neste estado até o sinal *enable* ser ativado, fazendo o sistema avançar para o estado *número*. De notar que o sistema só volta a este estado quando o sinal *enable* for desativado, esteja o sistema em que estado estiver.

- **número** – Este estado surge após o sinal *enable* ser ativado. É neste estado que é definido o número a ser validado assim como também a *flag* auxiliar mencionada no estado anterior. Tem-se assim que, se o número atual for o número 9, significa isto que já foram averiguados todos os números para a célula atual e então é necessário reposicionar o valor do número em 1, por forma a avaliar a célula seguinte. A *flag* auxiliar é então ativada, sinalizando que a célula atual acabou a sua verificação e que se pode avançar para a célula seguinte.

  Caso o número não seja o 9, é então feito o incremento deste por forma a avaliar o número seguinte e a *flag* auxiliar é imposta a zero.

- **addr** – Neste estado, é definida a célula a validar. Esta definição é feita tendo por base o sinal da *flag* do estado anterior. Se a *flag* tiver sido ativa, significa que já foram validados todos os 9 números para a célula atual, pelo que se pode avançar para a célula seguinte. Se for este o caso, é então feito o *reset* da *flag* e a posição *addr* é incrementada de uma unidade. Ao mesmo tempo, é feito o *reset* do sinal de possibilidades, isto é, as possibilidades para nova célula são zero, uma vez que ainda não se procedeu à validação.

  Se a *flag* não tiver sido ativa, não existe necessidade de efetuar nenhuma operação neste estado e este funciona como um estado de "passagem", avançando o sistema para o estado *validação*.

- **validação** – Este estado é responsável por acionar o processo de validação de um número para uma dada casa. Esta validação é feita recorrendo à máquina *Validate* já mencionada na secção 5.1.3.

  Em primeiro lugar, é verificado se a posição atual não corresponde ao limite do puzzle. Se atingida a posição limite, o sistema termina avançando para o estado *end*. Caso não tenha sido atingida esta posição, é necessário verificar se estamos perante uma célula vazia ou não. Se estivermos perante uma célula vazia, isto é, se *data in* for zero (valor lido do barramento de dados do puzzle de memória, Figura 5.14) é então acionado o processo de *Validate*, ativando o sinal *enable* do mesmo, e o sistema salta para o estado *wait*. Caso a célula esteja preenchida, isto é, *data in* diferente de zero, a sua lista de possibilidades é então colocada a zero, o que significa que não existem possibilidades e o sistema avança para o estado *reset*.

- **reset** – Este estado serve apenas para incrementar o endereço da posição *addr* antes de retornar ao estado *addr*. O nome deste estado deve-se ao facto de surgir no seguimento do *reset* das possibilidades de uma célula.

- **wait** – Este estado surge como um estado redundante de espera por fim do processo *Validate*. Uma vez iniciado o processo no estado *validação*, o sistema fica neste estado em espera até que a *flag* de sinalização de fim de processo *Validate* seja ativada, sendo que, nesta altura, o sistema avança para o estado *valida número*.

- **valida número** – Após o término do processo de *Validate*, é então necessário proceder ao preenchimento da lista de possibilidades. Assim sendo, se o resultado do processo *Validate* for *valid = '0'*, significa que o número é válido (ver estado *idle* da secção 5.1.3) e, portanto, este é posto na lista de possibilidades através de um sinal auxiliar responsável por armazenar os vários candidatos por ordem. Este sinal corresponde à entrada do barramento de dados da lista de possibilidades.

  De acordo com a estrutura de dados para a lista de possibilidades (secção 3.2.2), os números terão que ser preenchidos por ordem crescente. Uma vez que eles já são

verificados por ordem crescente (do número 1 ao 9), basta então averiguar qual a primeira casa livre para preencher com o próximo candidato, tal como demonstrado na Figura 5.13.

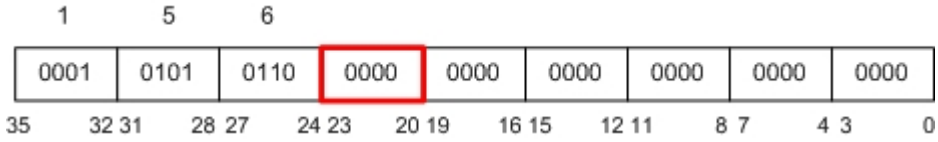


Figura 5.13 – Exemplo ilustrativo do preenchimento do sinal auxiliar de possibilidades

A primeira casa livre encontrada é então preenchida com o número do candidato. Este sinal auxiliar vai sendo gerado à medida que os números vão sendo validados, sendo que, chegado ao último número, isto é, ao número 9, para uma dada célula, este sinal fica pronto a ser escrito no barramento de dados da lista de possibilidades para a posição *addr*.

Se o número não for válido, o sinal auxiliar de possibilidades mantém-se no seu valor atual, visto não ter sofrido nenhuma alteração desde o estado *addr*.

- **escrita** – Após o barramento de endereços ter sido atualizado com a posição correta para efetuar a escrita do valor das possibilidades, é carregado, para o barramento de dados da lista de possibilidades, o sinal auxiliar que contém as possibilidades para a célula atual, assim como também é ativa a *flag* de escrita da memória da lista de possibilidades. Desta forma, os candidatos, até então passíveis de ser colocados na célula da posição *addr*, ficam associados à mesma.

- **end** – Este é o último estado e, como tal, cabe a este sinalizar o fim do processo de criação da lista de possibilidades. Esta sinalização é feita através da ativação da *flag eopl*. Em simultâneo, é feito o *reset* do barramento de endereços, de modo a garantir que a posição *addr* é colocada na sua posição inicial. O sistema mantém-se neste estado até que o sinal *enable* seja desativado.

Como se pode facilmente deduzir, esta implementação é ligeiramente diferente do *software*. Enquanto em *software* foi criada uma rotina responsável por criar a lista de

possibilidades e uma para a atualizar, esta rotina apenas cria a lista de possibilidades. No entanto, por forma a garantir o bom preenchimento e atualização foi necessário partir de dois pressupostos:

1. A lista associada a todas as células preenchidas tem forçosamente que ser preenchida a zeros. Isto permite que células previamente livres e com listas de possibilidades, após terem sido preenchidas, a sua lista deixe de conter números.
2. Aquando da atualização dos candidatos para uma célula vazia, é feito o *reset* da sua lista de possibilidades e avaliados os candidatos um-a-um. Note-se que, na implementação em *software* da atualização da lista, um candidato que deixasse de o ser era removido da lista e feito o *shift* de todos os candidatos seguintes (ver secção 4.1.4). Este pressuposto evita esta complicação na implementação em *hardware*.

Com estes pressupostos, esta máquina não só trata de criar a lista de possibilidades como também de a atualizar. O seu interface é apresentado na Figura 5.14.

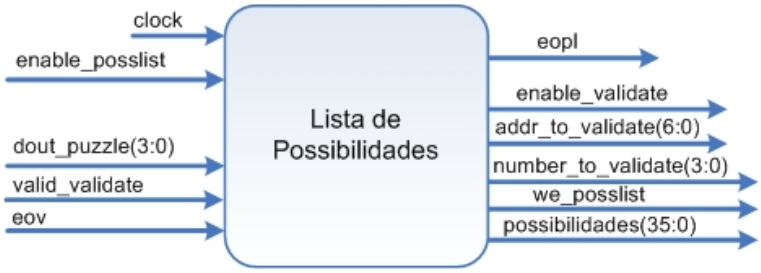


Figura 5.14 – Interface do processo *Lista de Possibilidades*

5.1.5 Mínimos

Ao contrário da implementação em *software*, a implementação da função responsável por gerar o mapa de mínimos torna-se mais complexa no que diz respeito à sua implementação em *hardware*. Em primeiro lugar, esta corresponderá a uma máquina de estados finitos, responsável por gerar o mapa de mínimos e devolver o endereço da célula que contém o menor número de possibilidades, tal como se pode observar no diagrama da Figura 5.15.

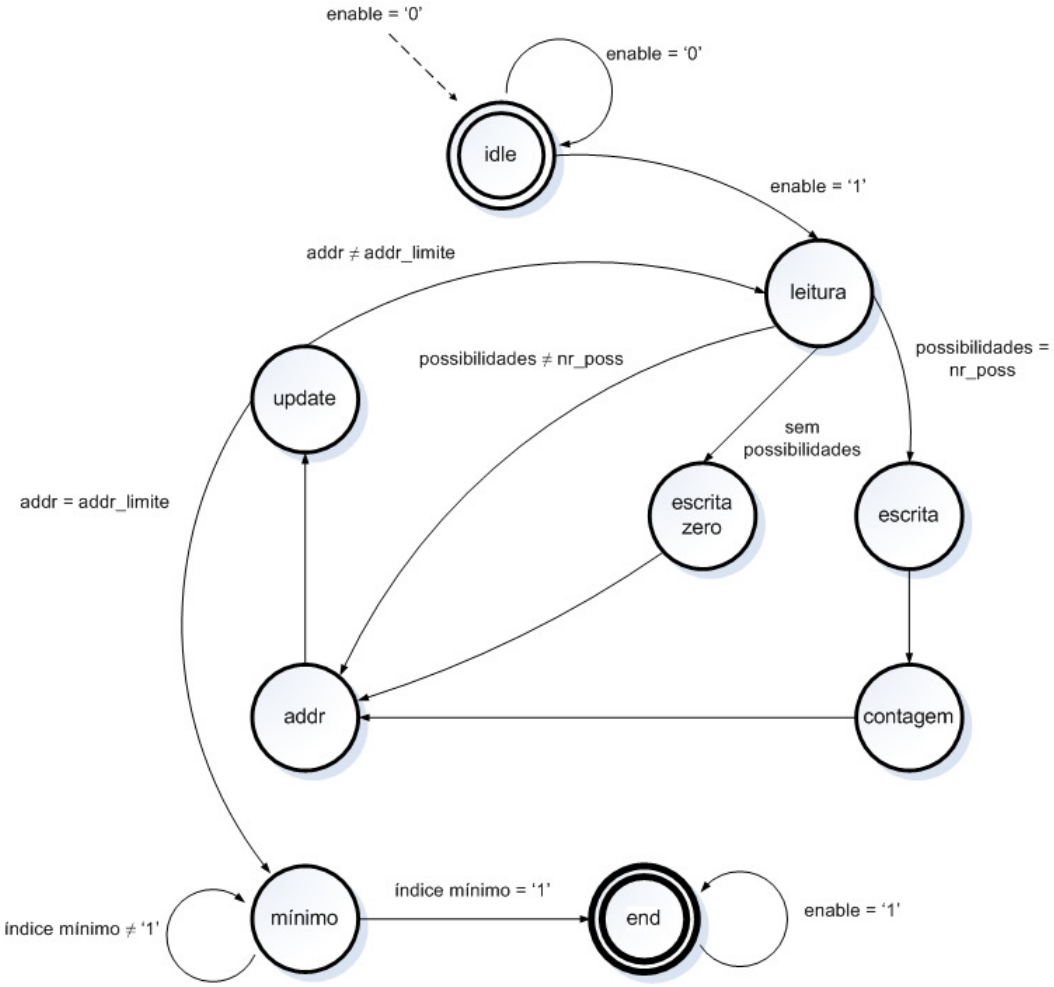


Figura 5.15 – Máquina de estados finitos do processo *Mínimos*

Por forma a simplificar a explicação deste processo, este irá ser dividido em 2 fases. A primeira fase responsável por gerar o mapa de mínimos é constituída apenas por 7 estados, que se estendem desde o estado *idle* até ao estado *update*. Os últimos 2 estados fazem parte da segunda fase, a qual determina a posição do mínimo. Na sua totalidade, este processo é então constituído por 9 estados que, conjuntamente, constituem então a máquina de estados finitos de *Mínimos*, cujo interface é apresentado na Figura 5.17.

A cada um destes estados corresponde o seguinte conjunto de operações:

- **idle** – Este é o estado inicial e, como tal, responsável pela inicialização de algumas das *flags* e sinais de controlo. É assim feito o *reset* da *flag* de fim do processo de geração do mapa (*eom*), assim como também do barramento de endereços e da posição correspondente à célula que contém o mínimo.

Neste estado, são também inicializados o sinal auxiliar, *nr_poss* (Figura 5.16), responsável por declarar o número atual de candidatos pelos quais se deve verificar em cada célula pelo que o seu valor inicial será de 2, pois nesta fase não existirão menos de 2 candidatos por célula como já referido; e o contador, responsável por atribuir o valor dos índices às células, pelo que o seu valor inicial é de 1, como também já referido.

O estado do sistema mantém-se em *idle* até que o sinal *enable* seja ativo e só retorna a este estado apenas e somente quando este sinal for desativado, esteja o sistema em que estado estiver.

- **leitura** – Neste estado, como sabido, o mínimo de possibilidades para cada célula é de 2 números, uma vez que o processo *Singles* resolveu os casos de possibilidades inferiores. Assim, baseando-se no valor do sinal auxiliar *nr_poss* (que se encontra inicializado com o valor 2) serão procuradas as células que possuam apenas este número de possibilidades na sua lista.

  Esta verificação é feita de acordo com a estrutura do barramento de dados da lista de possibilidades. Desta forma, para avaliar, por exemplo, a existência de apenas 2 candidatos basta ler os bits 31 a 28 e 27 a 24 e verificar se o primeiro conjunto é diferente de zero e o segundo igual a zero, tal como demonstrado na Figura 5.16.

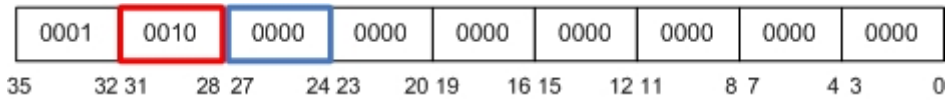


Figura 5.16 – Exemplo ilustrativo da verificação de dois candidatos para uma dada célula

Para validar a existência de 3 candidatos, basta ler uma posição para a direita, isto é, os bits 27 a 24 e 23 a 20, verificando se somente o segundo conjunto é zero. O processo é o mesmo para todos os outros números de candidatos que se deseje verificar.

Após esta verificação, o valor do contador (índice do mapa) é então carregado para o barramento de dados da memória do mapa e o sistema avança para o estado *escrita*, caso a célula contenha o número de candidatos que se encontra a ser validado no momento, ou, em caso contrário, avança para o estado *addr*, por forma a avançar para a célula seguinte. No entanto, se a célula se encontrar preenchida, é necessário preencher o mapa de mínimos com zeros para a posição atual. Esta escrita é feita no estado

escrita zero e, antes de avançar para este estado, é carregado o valor zero no barramento de dados de entrada do mapa de mínimos.

É estritamente necessário forçar que as células preenchidas contenham, no seu mapa de mínimos, o valor "a zeros", uma vez que, de cada vez que se necessita criar um mapa de mínimos, não existam conflitos com valores anteriormente escritos para a mesma célula.

- **escrita zero** – Uma vez que este estado surge após o barramento de dados do mapa de mínimos já ter sido previamente "carregado" com o valor zero e o barramento de endereços se encontrar já com o valor da posição da célula atual, é somente necessário ativar o sinal de escrita na memória responsável pelo mapa. Após isto, o sistema avança para o estado *addr*, por forma a avançar para a célula seguinte.

- **escrita** – Este estado é bastante idêntico ao estado *escrita zero*. Neste, é apenas necessário ativar o sinal de escrita da memória responsável pelo mapa, uma vez que o valor do contador já foi previamente carregado para o barramento de dados do mapa e, portanto, está pronto a ser escrito. Após isto, o sistema avança para o estado *contagem*, por forma a atualizar o valor do contador para ser preenchido na próxima célula com o mínimo seguinte.

- **contagem** – Neste estado, é então efetuada a atualização do valor da contagem. Assim, é apenas incrementado o valor do contador de uma unidade.

- **addr** – Este estado efetua o controlo da posição que se encontra a ser lida. Aqui é então incrementado o valor do endereço *addr*, por forma a avançar para a célula seguinte e proceder-se então à verificação do número de candidatos.

- **update** – Sendo este o último estado desta primeira fase, neste é feita a verificação se, efetivamente, se atingiu o limite do puzzle, averiguando se o endereço da posição atual corresponde ao endereço da posição da última célula do puzzle (*addr_limite*) e se o valor do sinal auxiliar *nr_poss* se encontra com o valor 1001_2, isto é, o último número de candidatos testado foi 9. Se tal for o caso, é então feito o *reset* ao barramento de

endereços *addr* e o sistema avança para a segunda fase, onde procurará pela célula que contém então o menor número de candidatos. Caso contrário, o barramento de endereços *addr* sofre o *reset* e o sinal auxiliar *nr_poss* é então incrementado, uma vez não se terem ainda verificado todos os números possíveis de candidatos para as células; em simultâneo, o sistema avança para o estado *leitura*, repetindo assim o processo.

- **mínimo** – Sendo este o estado responsável por "devolver" o endereço da posição onde se encontra o mínimo deste puzzle, são então percorridas todas as células do mapa de mínimos através do incremento do endereço de posição *addr* até ser encontrada a célula que contém o valor '1', isto é, a célula preenchida com o primeiro índice.

- **end** – Sendo este o último estado, aqui é apenas sinalizado o fim do processo de geração do mapa de mínimos e de descoberta do mínimo através da ativação da *flag eom*. Tal como nas outras máquinas de estado, o sistema mantém-se neste estado até que o sinal *enable* seja desativado.

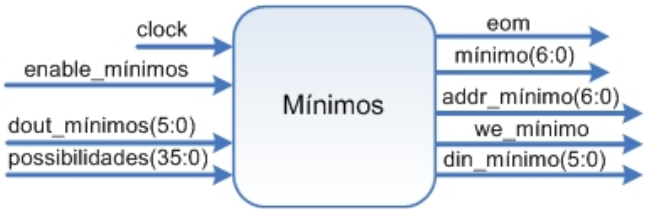


Figura 5.17 – Interface do processo *Mínimos*

5.1.6 Filled

A implementação em *hardware* para este processo (interface na Figura 5.19) corresponde a uma máquina de 4 estados apenas. Estes podem ser observados no diagrama da Figura 5.18.

Esta máquina é assim constituída por apenas 4 estados, correspondendo a cada um dos estados o seguinte conjunto de operações:

- **idle** – Neste estado, é efetuado o *reset* das *flags* de fim do processo *Filled* (*eof*) e da *flag* de sinalização de puzzle preenchido (*filled*). Em simultâneo, é também

reposicionado o barramento de posição de memória (*addr*) para a posição inicial, isto é, a primeira célula do puzzle.

O sistema mantém-se neste estado até que o sinal *enable* seja desativado. Assim, o sistema só volta a este estado quando este sinal for desativado.

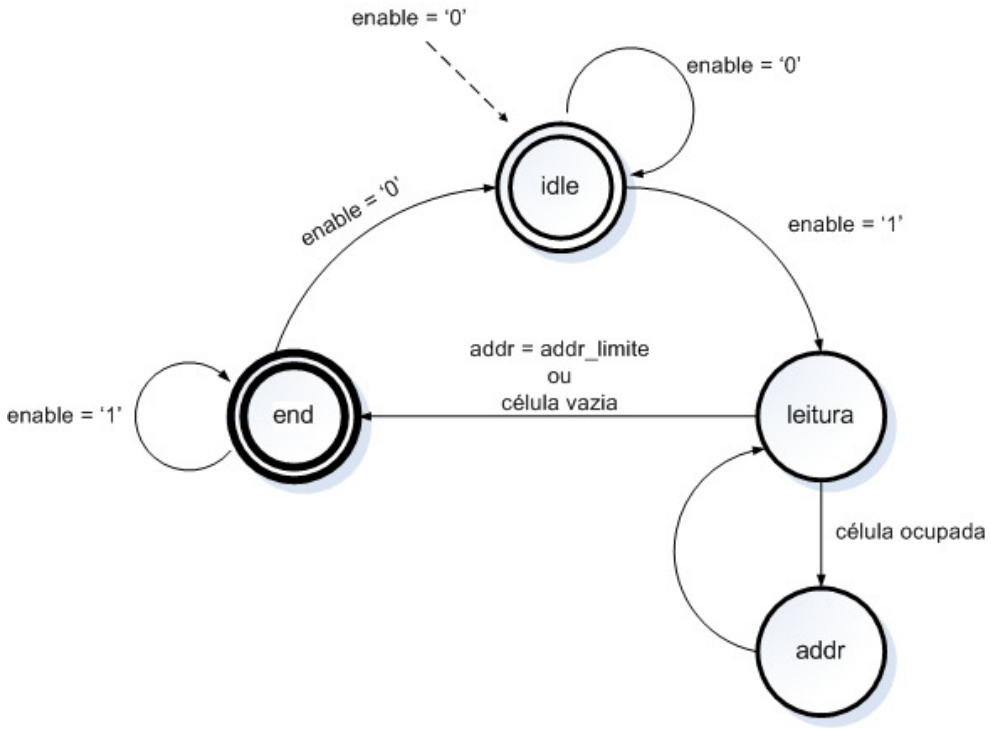


Figura 5.18 – Máquina de estados finitos do processo *Filled*

- **leitura** – Neste estado, é feita a leitura do valor da célula atual, definida pelo valor *addr*. Em primeiro lugar, é avaliado se a posição atual corresponde à posição final (*addr_limite*), sendo que, se for este o caso, o processo termina. Caso contrário, é então lido o barramento de dados da memória Puzzle e o resultado da leitura origina uma de duas situações:

    1. Se a célula atual estiver vazia, isto é, o resultado da leitura for zero, a *flag filled* mantém-se a zero e o sistema termina;

2. Se a célula atual estiver preenchida (resultado diferente de zero), a *flag filled* é ativa e o sistema avança para o estado *addr* por forma a avançar para a célula seguinte.

De notar que, se o puzzle estiver completamente preenchido, a *flag filled* é ativada na primeira célula e manter-se-á assim até ao final do processo. O mesmo não ocorre se for encontrada uma célula vazia, que forçará esta *flag* a ser desativada e a terminar o sistema.

- **addr** – Neste estado é apenas definida a célula a validar através do incremento da posição *addr*.

- **end** – Neste último estado, é apenas sinalizado o fim do processo através da ativação da *flag* usada para esse propósito (*eof*). Tal como acontecia anteriormente, o sistema mantém-se neste estado até que o sinal *enable* seja desativado.

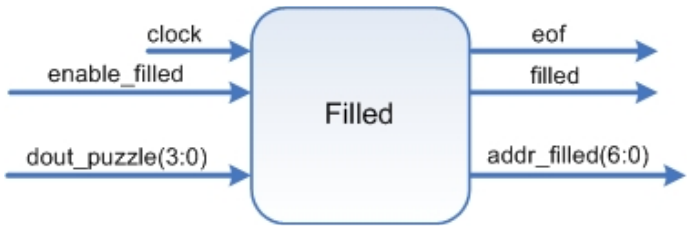


Figura 5.19 – Interface do processo *Filled*

5.1.7 Not Empty

A implementação deste processo é, na prática, muito idêntica à implementação do processo *Filled*. A principal diferença reside no estado de leitura dos valores das células, onde são tomadas decisões diferentes das tomadas no processo supracitado.

Desta forma, observe-se a Figura 5.18. Ao contrário do que ocorre no estado *leitura*, este (*Not Empty*) apenas avança para o estado *addr* se o resultado da leitura da célula for nulo, isto é, célula vazia. Do mesmo modo, só avança do estado leitura para o estado *end* se o resultado da leitura for não nulo, isto é, célula preenchida. As *flags* envolvidas neste

processo são a de sinalização de fim de processo *eone* (*end of not empty*) e a de sinalização de puzzle não vazio *notEmpty* (Figura 5.20).

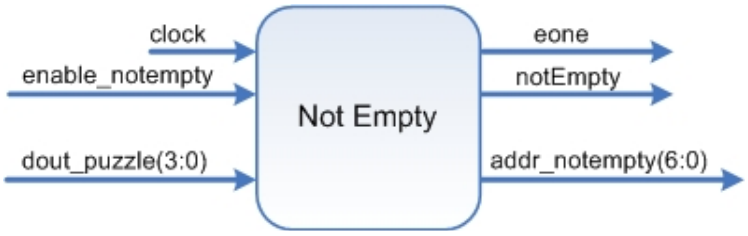


Figura 5.20 – Interface do processo *Not Empty*

5.1.8 Check Solution

A implementação em *hardware* desta função é algo muito parecida com a implementação do processo *Validate*, já apresentada na secção 5.1.3. Na implementação do processo *Validate*, este recebia um número e uma posição *addr*, averiguando se o número era passível de ser colocado nessa mesma posição. Neste novo processo, a diferença reside no facto de, para cada posição, se ler o número já preenchido, uma vez que se parte do pressuposto que o puzzle já se encontra totalmente preenchido, e averiguar se esse número seria passível de ser colocado nessa célula, caso esta se encontrasse vazia. No fundo, é basicamente idêntica à *Validate* com a particularidade de variar o número e posição *addr* e de não verificar a própria posição (posição do número atual).

O diagrama de estados constituinte desta máquina pode ser dividido em 3 fases: a fase de leitura do número e verificação por linha; a verificação por coluna e, finalmente, a verificação por caixa. Na Figura 5.21 pode-se observar o diagrama de estados correspondente à primeira fase. Esta primeira fase, constituída por 7 estados, corresponde à validação dos números por linha. A cada um dos estados descritos na Figura 5.21, corresponde o seguinte conjunto de operações:

- **idle** – Este é o estado inicial e, como tal, o seu comportamento é igual a qualquer outro estado *idle* de qualquer outra máquina. Aqui é feito o *reset* de apenas duas *flags*: fim do processo de verificação de solução correta (*eocs - end of check solution*) e *flag* de sinalização de solução correta (*allright*).

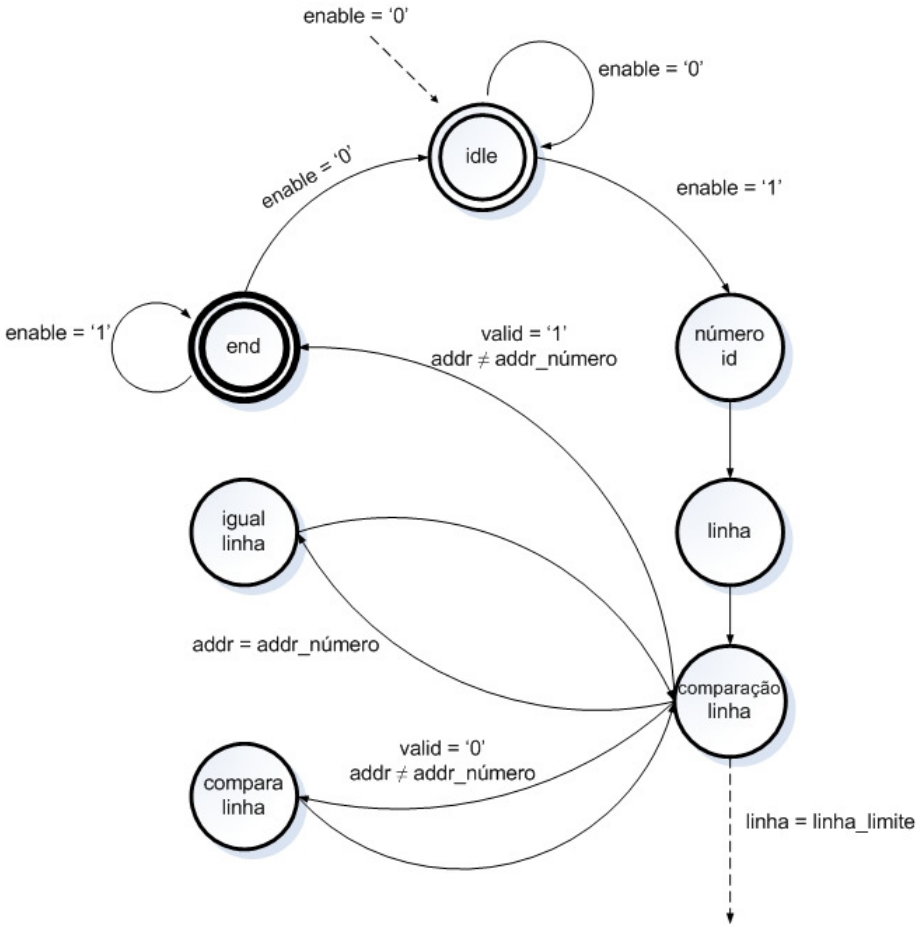


Figura 5.21 – Máquina de estados finitos correspondente à primeira fase do processo *Check Solution*

- **número id** – Neste estado, é feita a leitura do número da posição atual definida pela posição *addr_número*. Aqui é então armazenado o valor do número existente nessa célula, assim como também são salvaguardados os identificadores de coluna e de caixa.

- **linha** – Através do valor do *addr_número*, é, neste estado, definido o valor da posição *addr* do início da linha, isto é, se o *addr_número* corresponder à posição de uma célula da primeira linha, o valor da posição *addr* é então definido como o endereço da primeira célula. Aqui é também definido o limite de cada linha.

- **comparação linha** – Constitui um estado intermédio da comparação dos valores da linha com o do número salvaguardado. Em primeiro lugar, é averiguado se a posição

atual não corresponde à posição do limite da linha. Se sim, então a linha já foi verificada e pode-se avançar para a comparação coluna. Caso contrário, uma vez na linha, é necessário também verificar se a posição atual não corresponde à mesma posição do número que se quer validar, isto é, se a posição *addr* é igual à posição *addr_número*. Se sim, o sistema salta para o estado *igual linha*, se não, o sistema avança para a segunda fase deste estado, o estado *compara linha*, caso o sinal de *valid* (sinalização de solução correta) continue a zero (tal como o sinal *valid* da máquina *Validate*, este é invertido). Caso este não seja zero, significa que a solução já não é válida e não será necessário continuar com o processo, terminando-o.

- **igual linha** – Este estado apenas trata de efetuar o incremento da posição *addr*. Uma vez na verificação linha, este incremento é unitário. Após isto, o sistema regressa ao estado *comparação linha*.

- **compara linha** – Este estado constitui a segunda fase da comparação dos valores da linha. Se o número da posição atual for igual ao número salvaguardado no estado *número id*, o sinal auxiliar *valid* é então ativado, sinalizando que a solução não é válida; caso contrário, este sinal mantém-se inativo. Em simultâneo, a posição *addr* sofre o incremento de uma unidade e o sistema volta ao estado *comparação linha*.

  De notar que esta verificação por linhas encontra-se dividida em dois estados: o *comparação linha* e o *compara linha*. O primeiro é responsável pela definição das posições, ao passo que o segundo, pela leitura do valor a ser comparado.

Após o sistema terminar a verificação da linha e atingir a sua posição limite, este avança para a segunda fase, que corresponde à verificação da coluna. A implementação desta segunda fase é bastante idêntica e obedece ao diagrama de estados apresentado na Figura 5.22. As operações correspondentes a cada um destes 5 estados são:

- **coluna** – Este estado surge após o término da primeira fase. Através do valor do identificador da coluna salvaguardado no estado *número id*, é carregado no barramento de endereços de posição *addr* a posição inicial da respetiva coluna. É, aqui também, definido o limite da coluna.

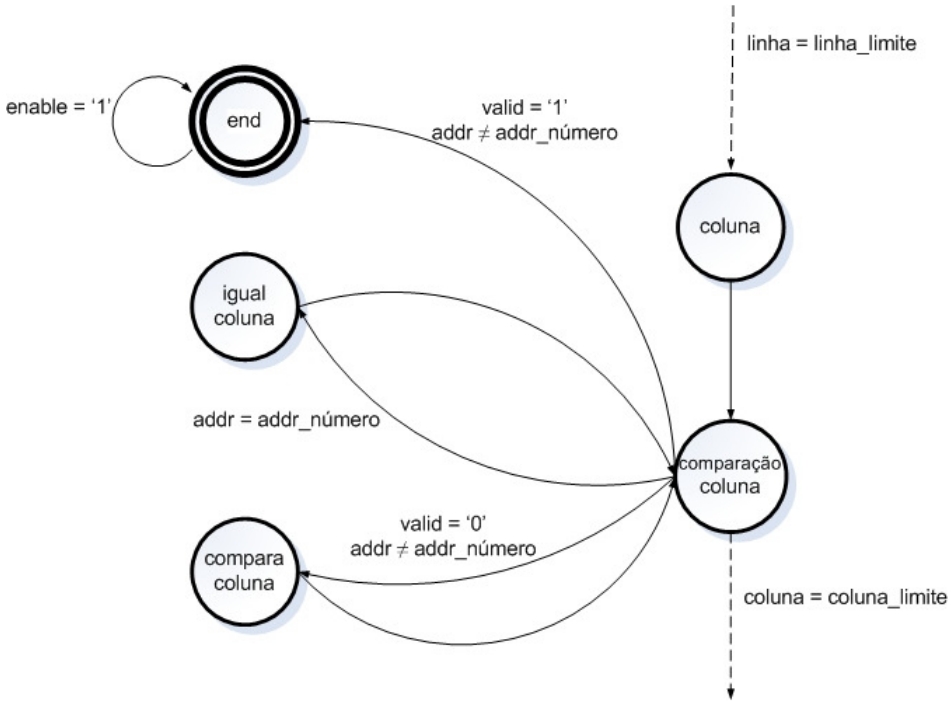


Figura 5.22 – Máquina de estados finitos correspondente à segunda fase do processo *Check Solution*

- **comparação coluna** – Este estado, exatamente igual ao *comparação linha*, constitui um estado intermédio da comparação dos valores da coluna com o do número salvaguardado.

- **igual coluna** – Este estado apenas trata de efetuar o incremento da posição *addr*. Como estamos perante a verificação de uma coluna, este incremento é de 9 unidades por forma a avançar para a próxima célula da mesma coluna. Após isto, o sistema regressa ao estado *comparação coluna*.

- **compara coluna** – Este estado, semelhante ao que ocorria anteriormente, constitui a segunda fase da comparação dos valores da coluna. Se o número da posição atual for igual ao número salvaguardado no estado *número id*, então o sinal *valid* é ativado, indicando que a solução não é válida. Caso contrário, este sinal mantém-se inativo. Em simultâneo, e uma vez que estamos perante a verificação da coluna, a posição *addr* sofre o incremento de 9 unidades.

Após a verificação da coluna ter terminado, é tempo de avançar para a terceira fase, que corresponde à verificação da caixa. A implementação é um pouco diferente das duas fases anteriores, como se pode notar pela observação do diagrama de estados da Figura 5.23.

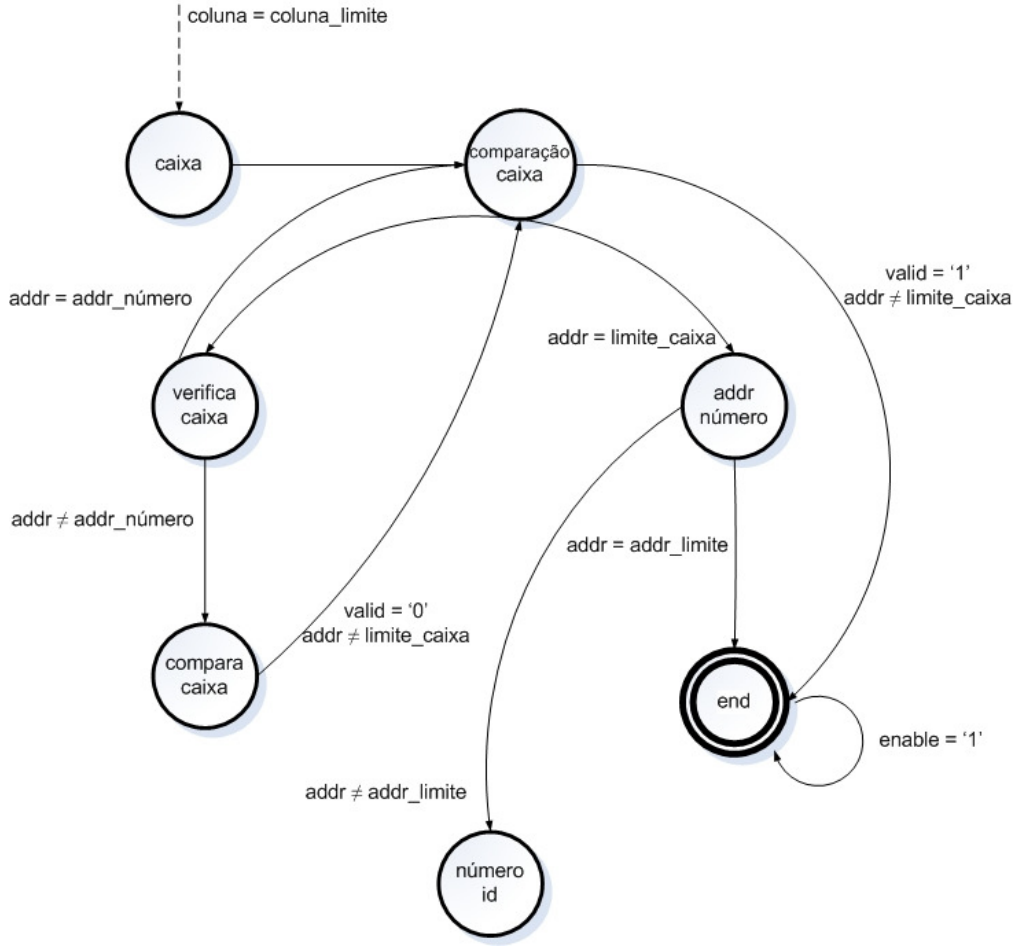


Figura 5.23 – Máquina de estados finitos correspondente à terceira fase do processo *Check Solution*

As operações correspondentes a cada um destes 7 estados são:

- **caixa** – Após a verificação da coluna, é necessário redefinir o ponto de começo e limite para a verificação da caixa. Essa definição é feita neste estado, tendo por base o identificador salvaguardado no estado *número id*. São então definidos a posição *addr*, assim como também os 3 limites necessários à caixa, uma vez que esta é tratada como 3 linhas diferentes.

- **comparação caixa** – Neste estado, é averiguado se o limite atingido é o da última posição da caixa. Em caso afirmativo, chegou-se ao limite da mesma e a verificação para o número definido no estado *número id* acabou, havendo necessidade de avançar para o número seguinte, isto é, incrementar o valor da posição *addr_número* e avançando para o estado *addr_número*.

  Se este não for o caso, se o sinal *valid* se encontrar inativo, o sistema avança parra o estado *verifica caixa*, por forma a verificar os valores da caixa, senão para o estado *end*, uma vez que a solução apresentada já não é válida.

- **verifica caixa** – Este estado é apenas um estado intermédio responsável por verificar a posição atual da caixa. É, então, em primeira instância, verificado se a posição atual (*addr*) é igual à posição *addr_número*. Se sim, estamos então na mesma célula que desejamos verificar se é válida ou não, e, por conseguinte, é necessário avançar para a célula seguinte, uma vez que não se pode efetuar a verificação desta. Por forma a avançar para a célula seguinte são verificadas 3 situações:

  1. Célula atual corresponde ao limite da primeira linha da caixa e, portanto, a posição terá que ser incrementada de 7 valores para avançar para a primeira célula da linha seguinte da caixa;
  2. Célula atual corresponde ao limite da segunda linha da caixa e, portanto, a posição terá que ser incrementada de 7 valores, tal como anteriormente;
  3. Se não estivermos perante nenhum destes limites basta então incrementar a posição de uma unidade para avançar para a célula seguinte, isto é, a célula na mesma caixa e na mesma linha que surge de seguida.

  Após isto, o sistema regressa ao estado *comparação caixa*. Todavia, se a célula atual não for a mesma definida pela posição *addr_número*, poder-se-á proceder à verificação, avançando assim para o estado *compara caixa*.

- **compara caixa** – Este estado é semelhante aos estados *compara linha* e *compara coluna*. Aqui é verificado se o número da posição atual é igual ao número

salvaguardado no estado *número id*, e em caso afirmativo, a *flag valid* é então ativa, sinalizando solução não válida. Se este não for igual, então a *flag* mantém-se inativa e o processo continua.

Em simultâneo, é também definida a posição *addr* da próxima célula a ser verificada, mas, uma vez que estamos perante a verificação da caixa, este incremento pode ser feito de 3 formas:

1. Se a posição da célula atual corresponder ao limite da primeira linha da caixa, o incremento é de 7 unidades, por forma a avançar para a primeira célula da linha seguinte;
2. Se a posição da célula atual corresponder ao limite da segunda linha da caixa, o incremento é de 7 unidades, pelo mesmo motivo do ponto 1;
3. Se a posição não corresponder a nenhum dos 2 limites, o incremento é então unitário, uma vez que é apenas necessário avançar para a célula seguinte que se encontra na mesma linha.

Após isto o sistema regressa ao estado *comparação caixa*.

- **addr número** – Este estado é responsável pela atualização da posição do número a ser lido. Deste modo, se a posição atual corresponder ao limite do puzzle, uma vez que este estado só existe para a verificação da caixa, significa que o fim foi atingido e, portanto, termina-se o processo. Caso contrário, o valor da posição *addr_número* definido no estado *comparação caixa* é agora carregado no barramento de endereços e o sistema volta ao estado *número id*, continuando assim o processo de verificação.

- **end** – Sendo este o último estado do processo *Check Solution*, este é responsável por efetuar o *reset* da posição *addr* e da posição *addr_número*, assim como também sinalizar o fim do processo de verificação da solução, ativando a *flag eocs*. Tal como em qualquer estado *end*, o sistema mantém-se neste estado até que seja desativado o sinal *enable*.

Estas três fases constituem o processo responsável por validar a solução do puzzle encontrada. Na totalidade, o processo *Check Solution* corresponde a uma máquina de estados finitos composta por 17 estados, sendo o seu interface feito de acordo com a Figura 5.24.

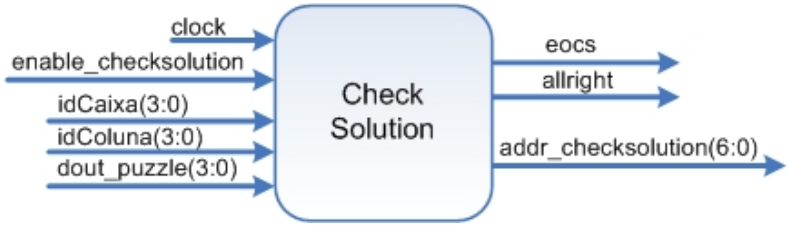


Figura 5.24 – Interface do processo *Check Solution*

5.1.9 Contador de ciclos

Este processo consiste num simples contador de ciclos de relógio (*clock*). É através deste contador que será determinado o tempo de execução do processo solucionador de puzzles. A Figura 5.25 apresenta o exemplo de funcionamento do contador.

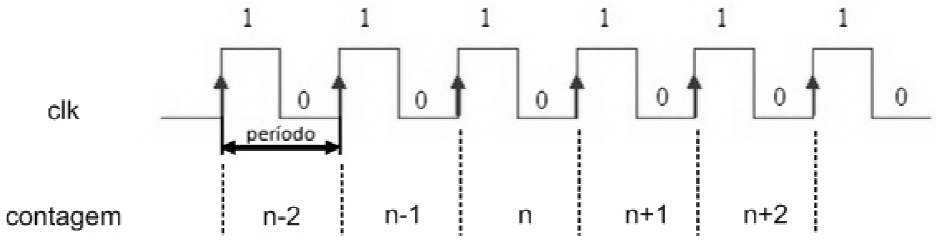


Figura 5.25 – Exemplo de funcionamento de um contador de ciclos de relógio

Tomando como exemplo a frequência de relógio da placa de prototipagem (secção 6.1), cada ciclo de relógio corresponde a:

$$período = \frac{1}{f} = \frac{1}{50MHz} = 20ns$$

E assim, o tempo total de execução após N ciclos de relógio vem dado por:

$$T_{execução} = N * período = N * 20ns$$

A implementação em *hardware* deste contador é bastante simples. Foi usado um contador de 32 bits, que permite obter uma gama de contagem bastante boa, mesmo para puzzles mais exigentes.

$$T_{máximo} = 2^{32} * período = 4294967296 * 20ns \cong 86s$$

Uma vez que, nesta fase, ainda não se sabe o tempo total necessário para a resolução dos puzzles na placa, pelo que uma gama de 86 segundos é um bom ponto de começo. Desta forma, a implementação deste contador pode ser descrita pelo diagrama de estados da Figura 5.26.

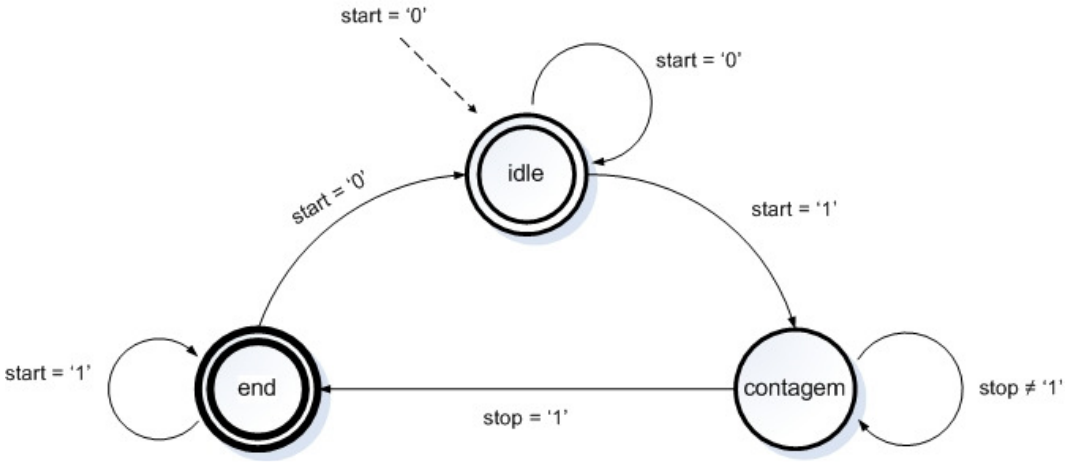


Figura 5.26 – Máquina de estados finitos do processo contador de ciclos

Esta máquina possui apenas 3 estados e a cada um deles corresponde um número mínimo de operações, tal como descrito de seguida:

- **idle** – Estado inicial responsável por efetuar o *reset* da *flag* sinalizadora do fim do processo de contagem (*eoc – end of count*). O sistema mantém-se neste estado até que o sinal *start* seja ativado.

- **contagem** – Após o sinal *start* ser ativado, inicia-se o processo de contagem, que só é interrompido quando o sinal *stop* for ativo, fazendo terminar o contador.

- **end** – Este estado serve apenas para garantir que o contador se encontra efetivamente parado, mantendo-se neste estado até que o sinal *start* volte a ser inativo. Desta forma, o processo só pode voltar a ser inicializado após o sinal *stop* ter sido ativado e o sinal *start* desativado, como medida de precaução. Sendo este o estado final, é sinalizado o fim do processo de contagem através da ativação da *flag eoc*.

Por fim, o interface deste processo é feito de acordo com a Figura 5.27.

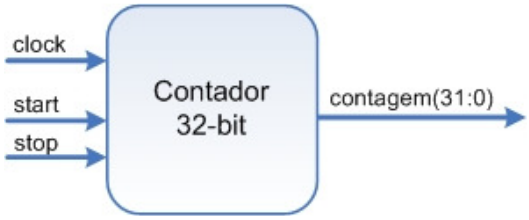


Figura 5.27 – Interface do processo contador de ciclos

5.2 Sistema global

Nesta secção, será descrita a forma como os processos descritos na secção anterior interagem entre si, formando o sistema solucionador de puzzles.

A implementação do solucionador será feita de três formas: simples, com tentativa e erro e com tentativa e erro com processamento paralelo. Estas três implementações diferem entre si quer no modo como os processos descritos anteriormente interagem, quer nos recursos necessários a cada uma.

Na Figura 5.28 é apresentado um diagrama de blocos genérico do sistema global.

A *Unidade de Controlo* será então responsável por gerir o acesso aos blocos de *Block RAMs* e *Processos*, sejam estes de controlo, dados ou endereços, e implementar o solucionador. Para cada uma das três formas de implementação do solucionador, será apresentada uma *Unidade de Controlo* diferente. De notar que, relativamente ao diagrama de blocos apresentado na Figura 3.7, foi adicionado um bloco extra, o bloco *Processos*, que consiste numa abstração de todos os processos abordados anteriormente.

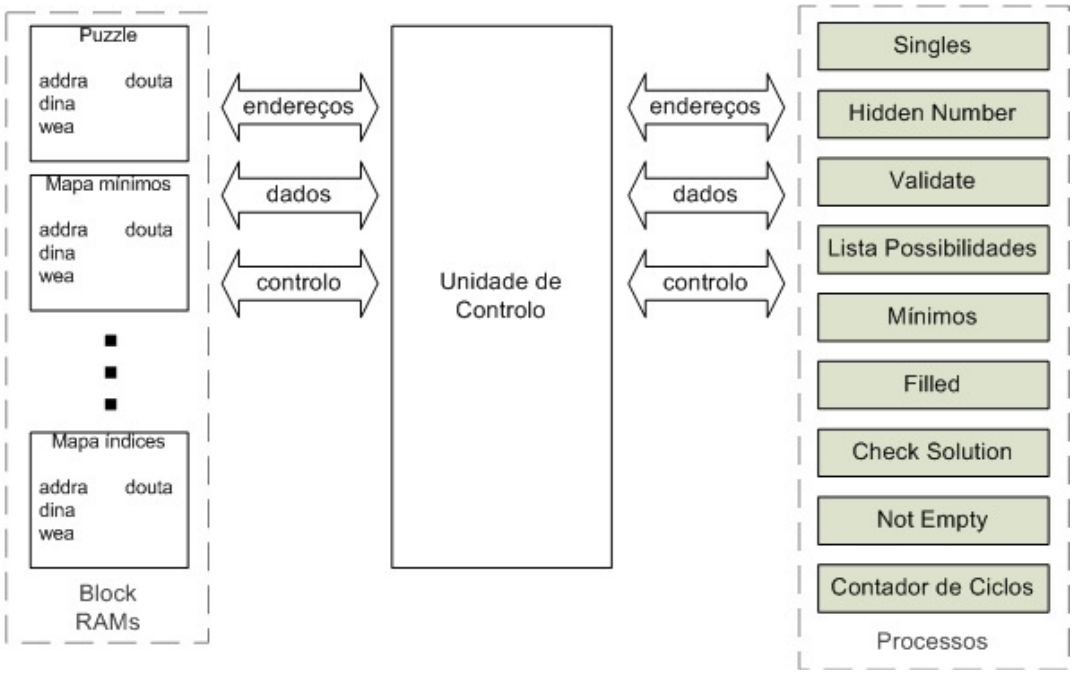


Figura 5.28 – Diagrama de blocos simplificado do sistema global

5.2.1 Simples

Nesta parte, é abordada a implementação do solucionador simples. Este consiste no ponto de partida para chegar às implementações mais complexas abordadas nas secções seguintes.

Este solucionador será capaz de resolver os puzzles mais simples, isto é, puzzles que não necessitem de recorrer ao método de tentativa e erro. Desta forma, é possível criar um algoritmo que interligue os vários processos já implementados, gerindo o seu acesso às *Block RAMs*.

Por observação do fluxograma da Figura 5.29, constata-se que esta implementação é constituída por um número reduzido de processos, sendo que as máquinas de estados implementadas correspondentes a estes processos são controladas através dos seus sinais *enable*. A gestão destes sinais é feita de acordo com o fluxograma apresentado.

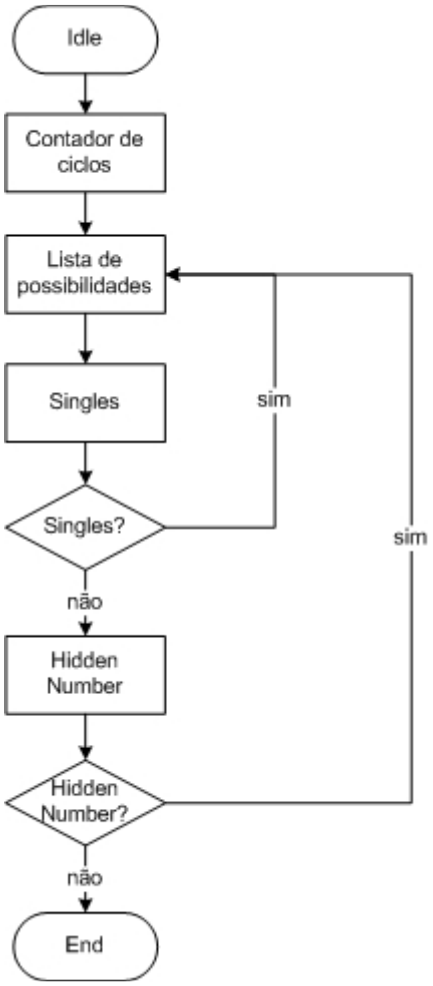


Figura 5.29 – Fluxograma da implementação em *hardware* do sistema solucionador simples de puzzles Sudoku

Por forma a compreender melhor a função de cada estado, é de seguida apresentada uma breve descrição:

1. *Idle* – este é o estado de *stand-by* deste solucionador. Neste estado, o sistema torna-se transparente, no que diz respeito ao acesso às memórias (*Block RAMs*), isto é, o acesso quer ao barramento de endereços, quer ao de dados e de controlo, é atribuído a uma entidade externa (ex.: Botões existentes na placa de prototipagem usada [40]), até que o sinal *enable* do solucionador seja ativado. Neste instante, o acesso às memórias passa a ser atribuído ao processo ativo no momento, determinado a partir do seu respetivo *enable*. Assim sendo, neste estado, os sinais *enable* de todas as máquinas encontram-se inativos;

2. *Contador de ciclos* – após o sinal *enable* do solucionador ser ativo, inicia-se o processo de contagem do número de ciclos de *clock*, ativando o sinal *start* do contador de 32 bit;
3. *Lista de possibilidades* – neste estado, é ativado o sinal *enable* do processo *Lista de Possibilidades*, pelo que os barramentos de acesso às memórias serão agora atribuídos a este processo, sendo apenas libertados quando este processo terminar e o seu respetivo sinal *enable* for desativado. De notar que esta máquina, ela própria gere o sinal *enable* do processo *Validate* (ver secção 5.1.4), pelo que, neste estado, estas duas máquinas irão competir pelo acesso às memórias;
4. *Singles* – neste estado, é ativado o sinal *enable* do processo *Singles*, sendo então averiguada a existência de candidatos únicos. Após o término do processo, o seu sinal *enable* é desativado, retirando-lhe o acesso às memórias.
5. *Singles*? – o resultado da *flag found_singles* dita o próximo estado do sistema, isto é, se forem encontrados candidatos únicos é necessário voltar a gerar a lista de possibilidades, caso contrário pode-se avançar para o estado *Hidden Number*;
6. *Hidden Number* – neste estado, é ativado o sinal *enable* do processo *Hidden Number*, sendo averiguada a ocorrência de candidatos únicos não diretos. Após o término do processo, o seu sinal *enable* é desativado, retirando-lhe o acesso às memórias.
7. *Hidden Number*? – o resultado da *flag found_hidden* ditará o próximo estado do sistema, sendo que se forem encontrados candidatos não diretos é novamente gerada a lista de possibilidades, caso contrário o processo termina;
8. *End* – este é o estado final e, como tal, é sinalizado o fim do processo solucionador do puzzle.

É de salientar que o sistema somente chega ao estado *End* quando não existirem mais candidatos únicos nem candidatos únicos não diretos e, portanto, só os puzzles passíveis de serem resolvidos com apenas estas duas funções chegarão ao fim do processo completamente preenchidos.

De notar, também, que a implementação deste solucionador não faz uso de todos os processos abordados anteriormente.

Na Figura 5.30 é apresentado o diagrama de blocos da *Unidade de Controlo* deste solucionador.

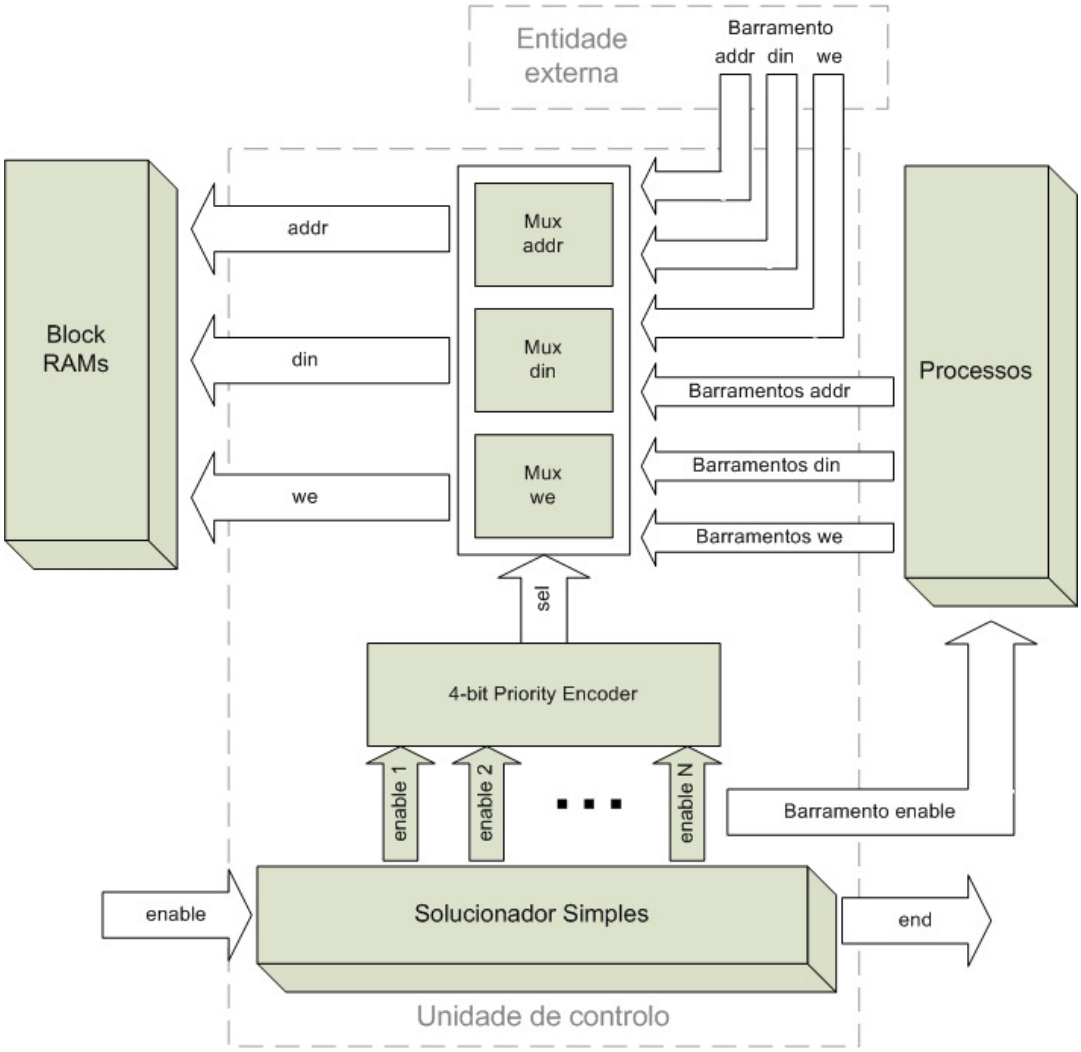
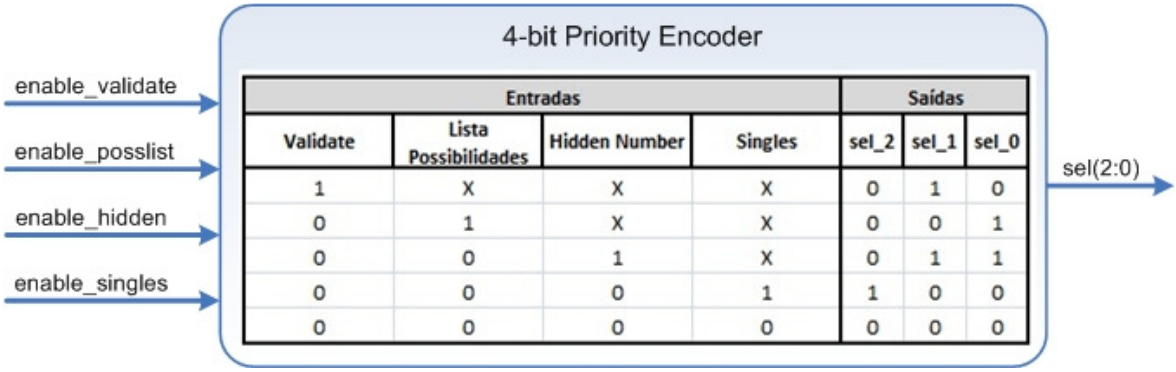


Figura 5.30 – Diagrama de blocos da *Unidade de Controlo* do solucionador simples

A gestão do acesso às memórias, ou seja, o acesso ao barramento de endereços, dados e controlo (sinais *we*) das *Block RAMs* por parte dos processos, é feita através dos seus sinais *enable*. No entanto, relembrando o processo *Lista de Possibilidades*, quando este está ativo (sinal *enable* ativo), é responsável por ativar e desativar sucessivamente o processo *Validate*, pelo que existirá um dado instante onde dois sinais *enable* estarão ativos. Uma vez que as *Block RAMs* são do tipo *Single Port RAM* (ver capítulo 3), o seu barramento não poderá ser controlado por mais que uma entidade em simultâneo. Existe, assim, a necessidade de atribuir prioridades aos vários processos envolvidos. A melhor forma de implementar uma gestão de acessos com prioridade é através de um codificador

de prioridade, sendo que este, consoante os sinais *enable*, gera um sinal de seleção que será usado como selecionador dos vários *multiplexers*, onde os barramentos dos vários processos estarão ligados na entrada, e a saída ligará às várias memórias, tal como demonstrado na Figura 5.30.

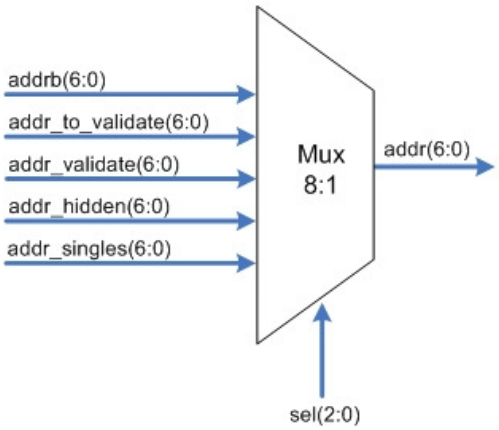
Na Figura 5.31, pode ser observada a tabela de verdade do codificador de prioridade. De notar que a primeira entrada, *Validate*, é a entrada de maior prioridade, a seguir a *Lista de Possibilidades* e outras. Este codificador é apenas necessário para evitar colisão no barramento entre estes dois processos, no entanto, é aproveitado para gerar o sinal de seleção para todos os outros, pelo que a prioridade destes últimos já não é fulcral para o bom funcionamento do solucionador.



Entradas				Saídas		
Validate	Lista Possibilidades	Hidden Number	Singles	sel_2	sel_1	sel_0
1	X	X	X	0	1	0
0	1	X	X	0	0	1
0	0	1	X	0	1	1
0	0	0	1	1	0	0
0	0	0	0	0	0	0

Figura 5.31 – Interface do codificador de prioridade do Solucionador Simples e respetiva tabela de verdade

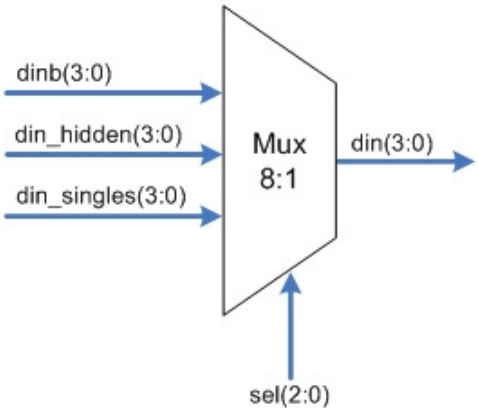
Relativamente aos *multiplexers* implementados, o *multiplexer* responsável pelo barramento de endereços gerirá o acesso ao mesmo barramento de endereços de todas as memórias envolvidas neste solucionador (Puzzle, Lista de Possibilidades, Mapa de colunas e de caixas) e atribuirá o acesso aos 4 processos intervenientes ou a uma entidade externa, consoante o sinal de seleção (*sel*), tal como se pode observar na Figura 5.32.



Entradas			Saídas	Entidade
sel_2	sel_1	sel_0	addr	
0	0	0	addrb	Externa
0	0	1	addr_to_validate	Lista Possibilidades
0	1	0	addr_validate	Validate
0	1	1	addr_hidden	Hidden Number
1	0	0	addr_singles	Singles
1	0	1	addrb	Externa
1	1	0	addrb	Externa
1	1	1	addrb	Externa

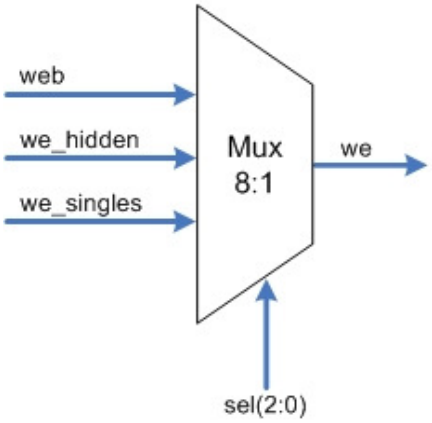
Figura 5.32 – Interface do *multiplexer* de endereços do solucionador simples e respetiva tabela de verdade

Quanto aos outros dois *multiplexers*, são estes os responsáveis por atribuir o acesso ao barramento de dados (Figura 5.33) e sinal *write enable* (Figura 5.34) do *Puzzle*, pelo que este acesso será disputado apenas entre o processo *Hidden Number* e o processo *Singles*. Nesta fase, apenas estes necessitarão de aceder a essa memória, uma vez que o processo *Lista de Possibilidades* terá ligação direta com o barramento da memória responsável por armazenar os candidatos, que é a única memória em que este processo necessitará de escrever dados.



Entradas			Saídas	Entidade
sel_2	sel_1	sel_0	din	
0	0	0	dinb	Externa
0	0	1	dinb	Externa
0	1	0	dinb	Externa
0	1	1	din_hidden	Hidden Number
1	0	0	din_singles	Singles
1	0	1	dinb	Externa
1	1	0	dinb	Externa
1	1	1	dinb	Externa

Figura 5.33 – Interface do *multiplexer* de dados do solucionador simples e respetiva tabela de verdade



Entradas			Saídas	Entidade
sel_2	sel_1	sel_0	din	
0	0	0	web	Externa
0	0	1	web	Externa
0	1	0	web	Externa
0	1	1	we_hidden	Hidden Number
1	0	0	we_singles	Singles
1	0	1	web	Externa
1	1	0	web	Externa
1	1	1	web	Externa

Figura 5.34 – Interface do *multiplexer* de controlo do solucionador simples e respetiva tabela de verdade

De notar que o sinal de seleção é o mesmo produzido pelo codificador de prioridade. No entanto, os *multiplexers* apresentados na Figura 5.33 e Figura 5.34 apenas reagem quando o processo *Hidden Number* ou *Singles* se encontram ativos, sendo transparente em todos os outros casos.

5.2.2 Tentativa e erro

Nesta parte, será abordada a implementação do solucionador com tentativa e erro. Tendo por base o solucionador simples, este sofrerá algumas alterações, quer no número de processos envolvidos, quer no fluxograma da sua implementação e na *Unidade de Controlo*.

Com a fase de tentativa e erro adicionada ao solucionador, este será capaz de resolver qualquer puzzle, desde que exista memória suficiente, sendo esta determinada através do número de *Block RAMs* disponíveis na *FPGA*. Para esta fase, tal como anteriormente, recorreu-se a uma árvore de pesquisa do tipo *BFS*, já descrita na Figura 4.23.

O fluxograma da Figura 5.35 traduz o conjunto de estados deste solucionador. Ao contrário da implementação simples, esta é constituída por um número mais elevado de processos, sendo que os sinais *enable* de cada processo são ativados/desativados de acordo com este fluxograma. Por forma a compreender melhor a função de cada estado, é, seguidamente, apresentada uma breve descrição:

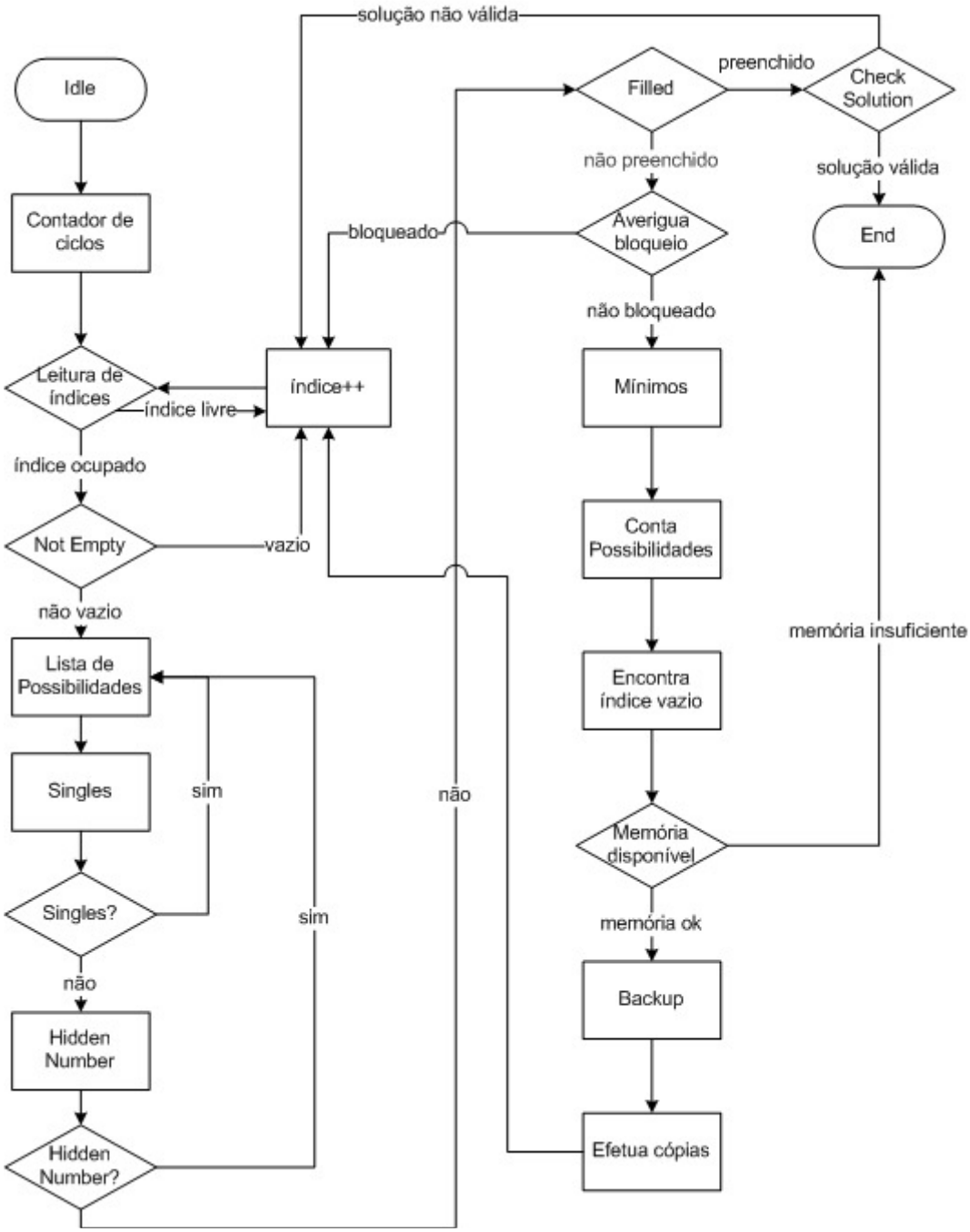


Figura 5.35 – Fluxograma da implementação em *hardware* do sistema solucionador com tentativa e erro de puzzles Sudoku

1. *Idle* – este é o estado de *stand-by* deste solucionador. Neste estado o sistema torna-se transparente no que diz respeito ao acesso às memórias, tal como no

solucionador anterior. O sinal *enable* de todos os processos intervenientes neste solucionador encontram-se inativos enquanto o sistema permanecer neste estado;

2. *Contador de ciclos* – após o sinal *enable* do solucionador ser ativo, inicia-se o processo de contagem do número de ciclos de *clock* ativando o sinal *start* do contador de 32 bits;
3. *Leitura de índices* – estado responsável por efetuar a leitura do índice atual, averiguando se este se encontra ocupado e pronto para proceder à execução do resto do algoritmo ou se existe necessidade de encontrar outro índice;
4. *Not Empty* – neste estado, é ativado o sinal *enable* do processo *Not Empty* por forma a que este averigue se o puzzle contém células preenchidas;
5. *Lista de possibilidades* – neste estado, é ativado o sinal *enable* do processo *Lista de Possibilidades*, pelo que os barramentos de acesso às memórias estão agora entregues a este processo. De notar que, tal como anteriormente, esta máquina gere o sinal *enable* do processo *Validate*;
6. *Singles* – neste estado, é ativado o sinal *enable* do processo *Singles*, sendo então averiguada a existência de candidatos únicos;
7. *Singles*? – aqui é avaliada a *flag found_singles*, resultante do processo *Singles*. O sistema avança para o estado *Hidden Number* se não tiverem ocorrido *Singles*, caso contrário é atualizada a *Lista de Possibilidades*;
8. *Hidden Number* – neste estado, é ativado o sinal *enable* do processo *Hidden Number*, sendo averiguada a ocorrência de candidatos únicos não diretos;
9. *Hidden Number*? – aqui é avaliada a *flag found_hidden*, resultante do processo *Hidden Number*. O sistema avança para o estado *Filled* se não tiverem ocorrido *Hidden Numbers*, caso contrário é atualizada a *Lista de Possibilidades*;
10. *Filled* – neste estado, é ativado o sinal *enable* do processo *Filled*. Caso o puzzle já se encontre preenchido é necessário verificar a veracidade da solução;
11. *Check Solution* – aqui é ativado o sinal *enable* do processo *Check Solution*, sendo então averiguada a veracidade da solução. Se a solução não for correta, é necessário avançar para o índice seguinte e continuar a execução do algoritmo;
12. *Averigua bloqueio* – caso o puzzle não se encontre preenchido, é necessário averiguar se este não se encontra bloqueado, isto é, se o algoritmo de tentativa e erro não produziu células livres que não possuem qualquer candidato. Claro está

que este estado só é útil depois da primeira suposição feita quanto ao valor para uma determinada célula;

13. *Mínimos* – neste estado, é ativado o sinal *enable* do processo *Mínimos*, sendo criado o mapa de mínimos e devolvido o endereço da posição de mínimo;
14. *Conta Possibilidades* – através da posição de mínimo obtida no estado anterior, contam-se o número de possibilidades existentes na célula referenciada por essa posição;
15. *Encontra índice vazio* – neste estado, é procurado o primeiro índice vazio, a partir do qual serão feitas tantas cópias do puzzle quanto o número de possibilidades da posição de mínimo;
16. *Memória disponível* – neste estado, é necessário averiguar se existe memória para guardar este novo índice, pelo que, se não existir, as cópias não poderão ser feitas e o solucionador terminará por falta de memória;
17. *Backup* – neste estado, é feita uma cópia do puzzle do índice atual para o puzzle auxiliar;
18. *Efetua cópias* – neste estado, o puzzle guardado no puzzle auxiliar é copiado para o primeiro índice vazio já determinado e para os índices seguintes, sendo então feito o número de cópias correspondente ao número de possibilidades existentes na posição de mínimo, onde cada uma destas cópias será preenchida com uma das possibilidades dessa mesma posição;
19. Após efetuar as cópias, dá-se o incremento do índice para o índice seguinte, e o algoritmo repete-se;
20. *End* – sendo este o estado final, é ativada a *flag end* responsável por sinalizar o fim do processo solucionador.

Para a implementação deste algoritmo, é necessário proceder a algumas alterações face ao algoritmo implementado no solucionador simples:

- A memória Puzzle (*Block RAM*) deixará de ser um puzzle apenas e passará a constituir um bloco de memórias com sinal *enable*, desmultiplexadas através do valor do índice atual (Figura 5.36);

- Adição do mapa de índices, contendo a informação das memórias livres/ocupadas assim como limite de índices possíveis para a execução do algoritmo;
- Reestruturação dos *multiplexers* de endereços e codificador de prioridade por forma a interagir com mais processos;
- Introdução de um *multiplexer* adicional para a disputa de acesso aos barramentos entre o bloco *Entidade externa* e o bloco *Solucionador* (Figura 5.36), uma vez que este bloco, responsável por gerir todos os outros processos, necessita também de aceder ao bloco de memórias e, como tal, também disputará o acesso aos barramentos. Desta forma, quando o barramento não tiver sido entregue a nenhum processo do bloco *Processos*, estará entregue ao *Solucionador*, a menos que este não se encontre ativo.

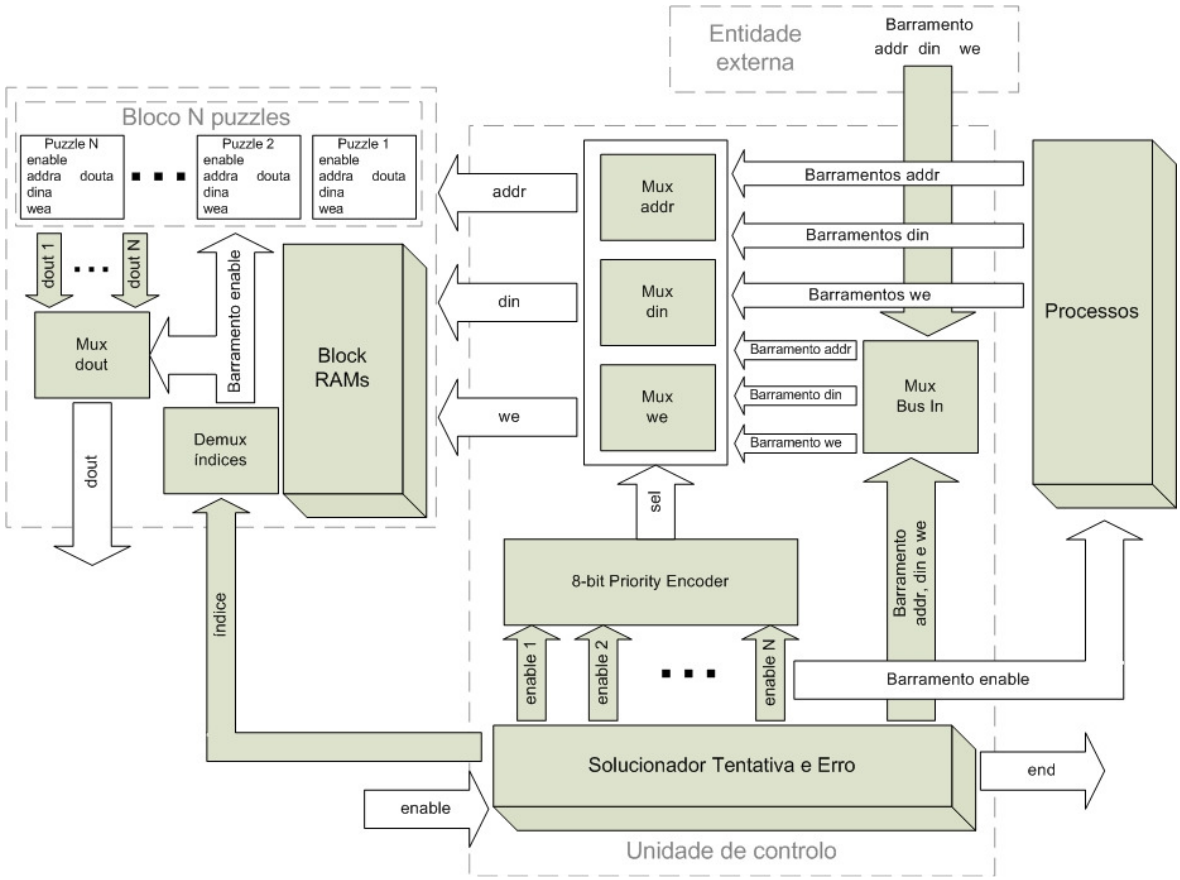


Figura 5.36 – Diagrama de blocos da *Unidade de Controlo* e memória do solucionador com tentativa e erro

Neste novo diagrama de blocos (Figura 5.36), tem-se então que a memória Puzzle deu agora lugar a um bloco de N memórias. O acesso individual a cada uma destas memórias é controlado através do valor do índice atual. Este índice será desmultiplexado no bloco *Demux índices*, que ativará a respetiva memória através do seu sinal *enable*, sendo que este, por sua vez, servirá de selecionador para multiplexar os dados de saída das memórias.

Desta forma, do ponto de vista da *Unidade de Controlo*, o Puzzle continua a comporta-se como um só puzzle, isto é, através da multiplexagem do valor do índice atual consegue-se obter uma abstração para o bloco de N memórias, comportando-se este como um puzzle apenas.

Nesta implementação, existirão tantos índices quanto o número de *Block RAMs* disponíveis na *FPGA*. De acordo com a secção 6.1, tem-se que:

Número total de *Block RAMs* disponíveis na *FPGA*: 28

Número de *Block RAMs* usado: 6
(Puzzle Auxiliar + Lista de Possibilidades + Mapa Colunas + Mapa Caixas + Mapa Mínimos + Mapa Índices)

Número de *Block RAMs* disponível: $28 - 6 = 22$

O número de *Block RAMs* disponível corresponde ao número máximo de puzzles passíveis de ser usados nesta implementação, assim como também ao limite de índices. Assim sendo, além da *Block RAM* usada para armazenar o puzzle original, esta implementação é capaz de resolver puzzles até um máximo de 21 suposições quanto ao valor das células, sendo que o mapa de índices conterá a informação acerca dos puzzles livres ao longo do processo.

A seleção da memória que se encontra ativa é feita através do desmultiplexador de índices. Este receberá na entrada um sinal capaz de selecionar cada um dos 22 puzzles (5 bits) e consoante o índice selecionado ativará o respetivo sinal *enable* (Figura 5.37 e Figura 5.38).

Entrada					Saídas					
índice_4	índice_3	índice_2	índice_1	índice_0	enable_p21	...	enable_p3	enable_p2	enable_p1	enable_p0
0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	0	0		1	0	0
0	0	0	1	1	0	0	1	0	0	0
		...						...		
1	0	1	0	1	1	0	0	0	0	0

Figura 5.37 – Tabela de verdade do bloco desmultiplexador de índices

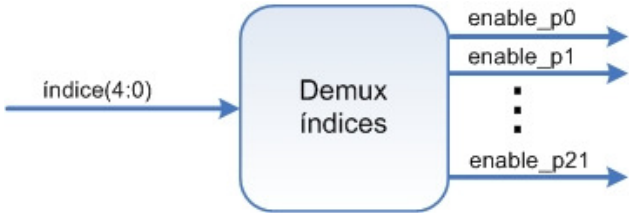
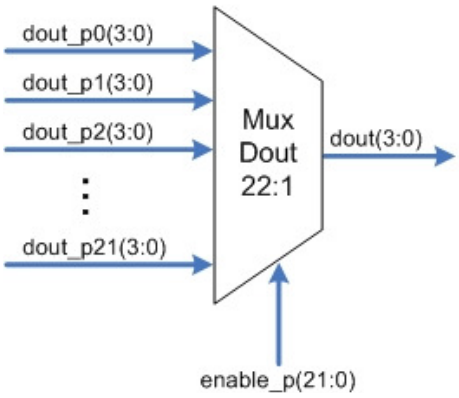


Figura 5.38 – Interface do desmultiplexador de índices

Do mesmo modo, consoante o sinal *enable* de cada uma das memórias, é selecionado o respetivo barramento de saída de dados das memórias. Assim, o barramento dos sinais *enable* atua como sinal de seleção do *multiplexer Mux Dout*, gerindo, desta forma, a saída de dados do bloco de 22 puzzles (Figura 5.39).

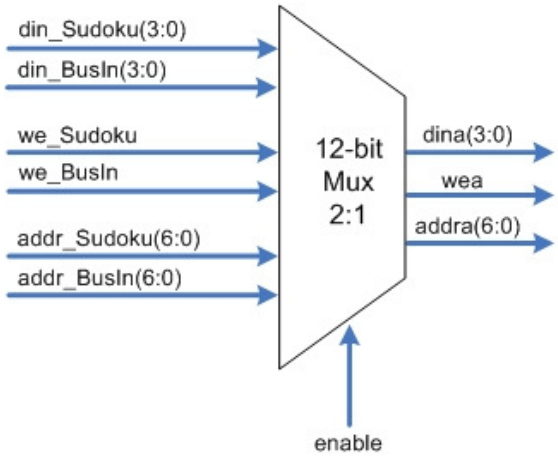


Entradas					Saídas
enable_p21	...	enable_p2	enable_p1	enable_p0	
0	0	0	0	1	dout_p0
0	0	0	1	0	dout_p1
0	0	1	0	0	dout_p2
		...			...
1	0	0	0	0	dout_p20
0	0	0	0	0	dout_p21

Figura 5.39 – Interface do *multiplexer Mux Dout* do bloco de memórias e respetiva tabela de verdade

A gestão do acesso ao barramento de endereços, dados e controlo (sinais *we*) das memórias por parte dos processos é feita através dos seus sinais *enable* (Figura 5.40). Uma vez que o solucionador necessita ele próprio (enquanto sistema global) também de aceder às memórias, este disputará o acesso aos barramentos juntamente com a *Entidade externa* através do *multiplexer Mux Bus In*. Desta forma, quando o solucionador se encontrar ativo,

este encontrar-se-á na posse dos barramentos de acesso, pelo que a *Entidade externa* somente acederá aos mesmos quando o solucionador não se encontrar ativo.

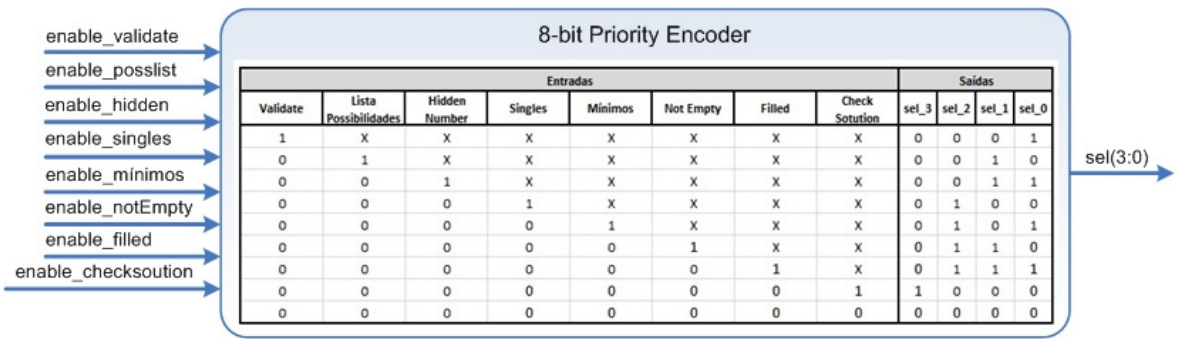


Entrada seleção	Saída		
enable	dina	wea	addra
0	Entidade externa		
1	Sudoku		

Figura 5.40 – Interface do *multiplexer Mux Bus In* e respetiva tabela de verdade

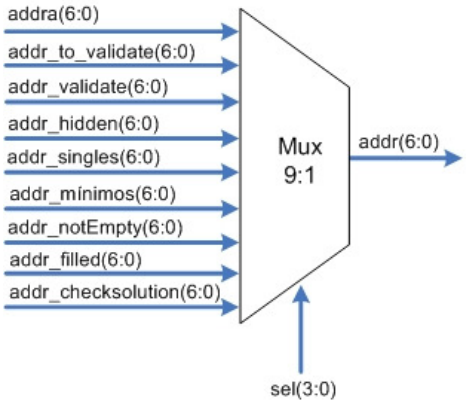
A saída resultante do *multiplexer Mux Bus In* (Figura 5.40) encontrar-se-á ligada aos restantes 3 *multiplexers* e disputará ela também o acesso aos barramentos com os restantes processos. Semelhantemente ao que ocorre na implementação do solucionador simples, quando nenhum processo se encontra ativo (o acesso era atribuído à *Entidade externa*), o acesso ao barramento é então entregue ao *Mux Bus In*. Deste modo, o barramento encontrar-se-á entregue ao solucionador para que leve a cabo as suas operações, a menos que este não se encontre ativo e, como tal, poderá a *Entidade externa* controlar o acesso (Figura 5.42).

Havendo um número maior de processos intervenientes nesta implementação é necessário redimensionar estes 3 *multiplexers*, assim como também o codificador de prioridade (Figura 5.41).



Entradas								Saídas			
Validate	Lista Possibilidades	Hidden Number	Singles	Mínimos	Not Empty	Filled	Check Sotution	sel_3	sel_2	sel_1	sel_0
1	X	X	X	X	X	X	X	0	0	0	1
0	1	X	X	X	X	X	X	0	0	1	0
0	0	1	X	X	X	X	X	0	0	1	1
0	0	0	1	X	X	X	X	0	1	0	0
0	0	0	0	1	X	X	X	0	1	0	1
0	0	0	0	0	1	X	X	0	1	1	0
0	0	0	0	0	0	1	X	0	1	1	1
0	0	0	0	0	0	0	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

Figura 5.41 – Interface do codificador prioridade do solucionador com tentativa e erro e respetiva tabela de verdade



Entradas				Saídas	Entidade
sel_3	sel_2	sel_1	sel_0	addr	
0	0	0	0	addra	Externa/Solucionador
0	0	0	1	addr_validate	Validate
0	0	1	0	addr_to_validate	Lista Possibilidades
0	0	1	1	addr_hidden	Hidden Number
0	1	0	0	addr_singles	Singles
0	1	0	1	addr_mínimos	Mínimos
0	1	1	0	addr_notEmpty	Not Empty
0	1	1	1	addr_filled	Filled
1	0	0	0	addr_CheckSolution	Check Solution
others				addra	Externa/Solucionador

Figura 5.42 – Interface do *multiplexer* de endereços do solucionador com tentativa e erro e sua respetiva tabela de verdade

Os *multiplexers din* e *we* são os mesmos da implementação do solucionador simples, uma vez que estes barramentos são apenas disputados entre o processo *Singles*, *Hidden Number* ou *Solucionador/Entidade externa*. Assim, estes poderão ser observados na Figura 5.33 e Figura 5.34, tendo em conta que as entradas *dinb* e *web* deverão ser substituídas por *dina* e *wea*, provenientes do *multiplexer Mux Bus In*.

5.2.3 Tentativa e erro com processamento paralelo

Neste capítulo, será abordada a implementação do solucionador com tentativa e erro implementado com processamento paralelo. O objetivo desta será conseguir melhorar o desempenho do Solucionador Tentativa e Erro já implementado.

A base de funcionamento desta fase é a mesma apresentada na secção anterior, isto é, baseando-se numa árvore de pesquisa do tipo *BFS* (Figura 4.23), mas sofrendo algumas alterações, de modo a obter vários solucionadores a funcionar em simultâneo, sendo que o número total de solucionadores será determinado pelo número de *Block RAMs* disponível na *FPGA*.

Como primeira abordagem, ter-se-ão vários solucionadores responsáveis por executar os processos já abordados sem que estes possuam tentativa e erro. Esta fase, tentativa e erro, será entregue a uma nova entidade capaz de gerir os vários solucionadores, quer no que diz respeito ao seu funcionamento como ao seu conteúdo.

Para esta abordagem, é então necessário proceder a um conjunto de alterações no solucionador face à implementação da secção anterior:

- Um solucionador, ao contrário do Solucionador com Tentativa e Erro, possuirá uma só *Block RAM* para armazenar dados relativos ao puzzle, uma para *Lista de Possibilidades* e uma por cada mapa;
- O solucionador deixa de possuir mapa de índices assim como gestor de índices;
- Uma vez que o solucionador não possuirá tentativa e erro, cabe a este averiguar se o puzzle se encontra corretamente preenchido ou gerar o seu mapa de mínimos indicando a posição em que se encontra o mínimo;
- O processo *Not Empty* deixa de ser necessário, uma vez que o solucionador só será ativo após receber os dados do puzzle-pai.

O fluxograma da Figura 5.43 traduz o conjunto de estados de cada solucionador. De seguida é feita uma breve descrição de cada estado:

1. *Idle* – este é o estado *stand-by* do solucionador. Durante este estado, o solucionador encontra-se em espera, aguardando que a nova *Unidade de Controlo* o inicie. Desta forma, este encontra-se no seu estado transparente, pelo que o acesso à memória Puzzle encontra-se atribuído à *Unidade de Controlo*, que é uma unidade externa a este solucionador;
2. *Lista de Possibilidades* – neste estado, é ativado o sinal *enable* do processo *Lista de Possibilidades*, que se encontrará na posse dos barramentos;
3. *Singles* – neste estado, é ativado o sinal *enable* do processo *Singles*;
4. *Singles*? – aqui é avaliado o preenchimento de *Singles* por parte do processo anterior. Se este tiver ocorrido, é então necessário atualizar a *Lista de Possibilidades*;
5. *Hidden Number* – neste estado, é ativado o sinal *enable* do processo *Hidden Number*;
6. *Hidden Number*? – aqui é avaliado o preenchimento de *Hidden Number* por parte do processo anterior. Se este tiver ocorrido, é então necessário atualizar a *Lista de Possibilidades*;

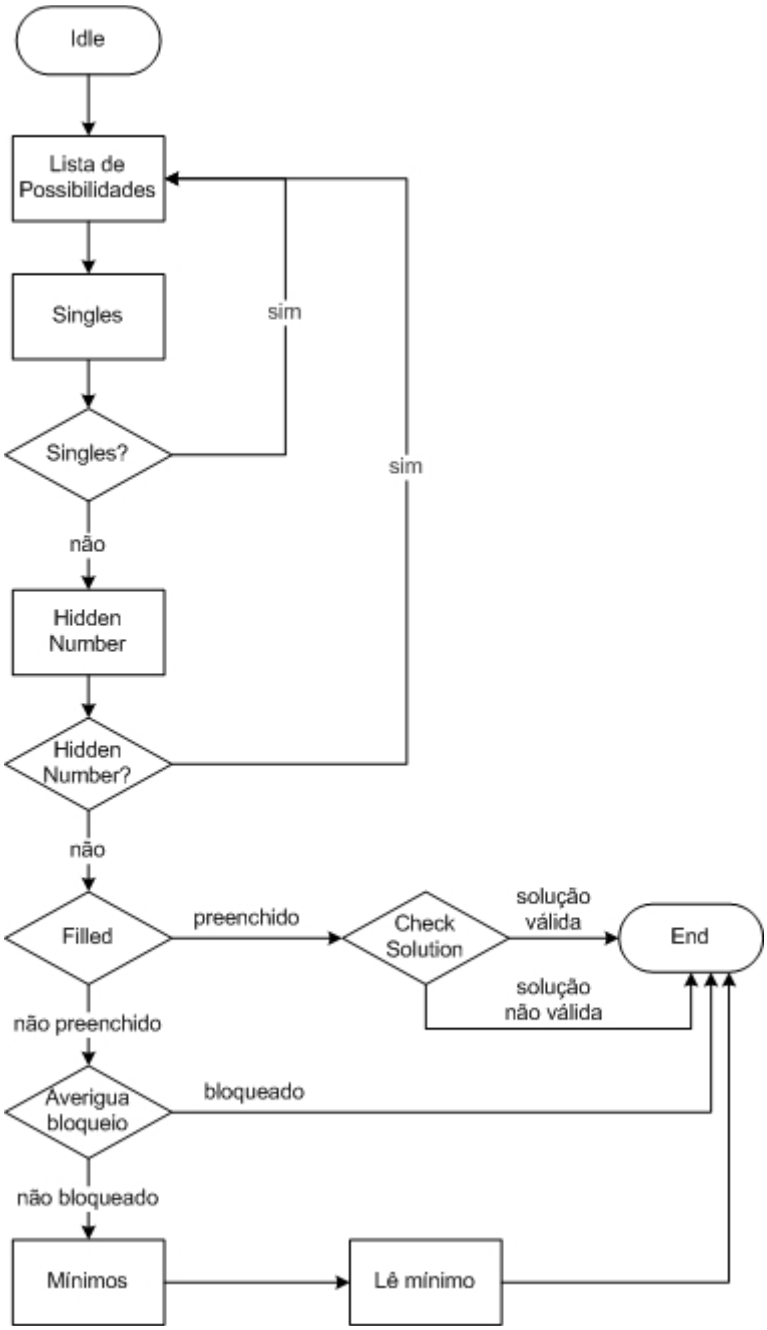


Figura 5.43 – Fluxograma da implementação em *hardware* do solucionador com tentativa e erro com processamento paralelo

7. *Filled* – aqui é ativado o sinal *enable* do processo *Filled*. Caso o puzzle esteja completamente preenchido, é necessário verificar se a solução constitui uma solução válida, uma vez que atrasos temporais facilmente poderão produzir soluções erradas;
8. *Check Solution* – aqui é ativado o sinal *enable* do processo *Check Solution*, sendo averiguada a veracidade da solução. Ao contrário do que acontecia anteriormente, seja a solução válida ou não, o processo é terminado, variando apenas no valor da *flag solved* (*flag* sinalizadora de solução correta);
9. *Averigua bloqueio* – caso o puzzle não se encontre totalmente preenchido, é necessário averiguar se este não se encontra bloqueado, uma vez que a possibilidade arriscada poderá ser a errada e, como tal, origina casas livres sem possibilidades;
10. *Mínimos* – neste estado, é ativado o sinal *enable* do processo *Mínimos*;
11. *Lê mínimo* – neste estado, o valor das possibilidades da posição de mínimo, obtida no estado anterior, é colocado no barramento de saída deste solucionador para ser lido pela *Unidade de Controlo*;
12. *End* – sendo este o estado final, é ativada a *flag end* responsável por sinalizar o fim deste solucionador.

Na Figura 5.44, podem notar-se semelhanças do bloco deste solucionador com o da implementação com tentativa e erro; no entanto, o número de processos envolvidos neste, assim como *Block RAMs*, é menor, pelo que foram especificados no diagrama, para evidenciar esta diferença. Os *multiplexers* assim como o codificador prioridade são iguais aos anteriores, tendo sido apenas removidas as entradas do processo *Not Empty* que não fará parte desta implementação.

Este diagrama fará parte de um bloco genérico que constituirá um solucionador dos N solucionadores possíveis (Figura 5.45) para esta implementação.

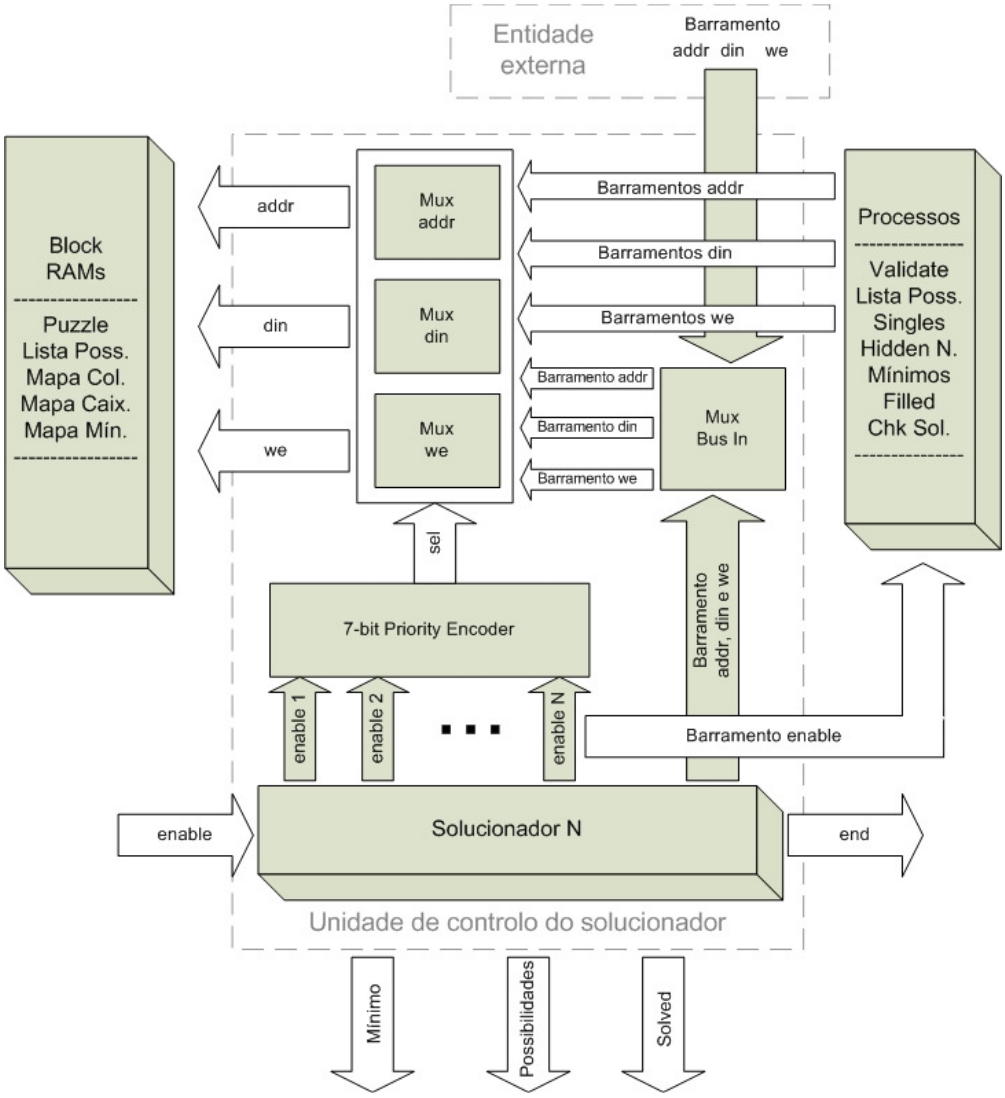


Figura 5.44 – Diagrama de blocos da *Unidade de Controlo* de um solucionador N sem tentativa e erro

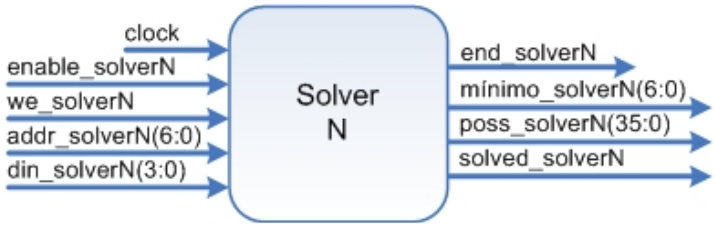


Figura 5.45 – Interface de um solucionador N

Após a implementação do solucionador, é necessário projetar a *Unidade de Controlo* do sistema geral (Figura 5.44), de modo a interagir com vários solucionadores e implementar a fase de tentativa e erro.

Assim, cada solucionador terá a si associado um índice, gerido pelo mapa de índices, sendo que o primeiro solucionador dos N possíveis conterá o puzzle a ser resolvido. A fase de tentativa e erro surge quando um solucionador não conseguir resolver o puzzle, indicando a posição de mínimo e as possibilidades da respetiva posição (Figura 5.45), sendo efetuadas tantas cópias quanto o número de possibilidades para índices livres, isto é, solucionadores que ainda não estejam a ser utilizados. Tendo esta afirmação como princípio base para a *Unidade de Controlo* é assim possível obter o fluxograma da Figura 5.46.

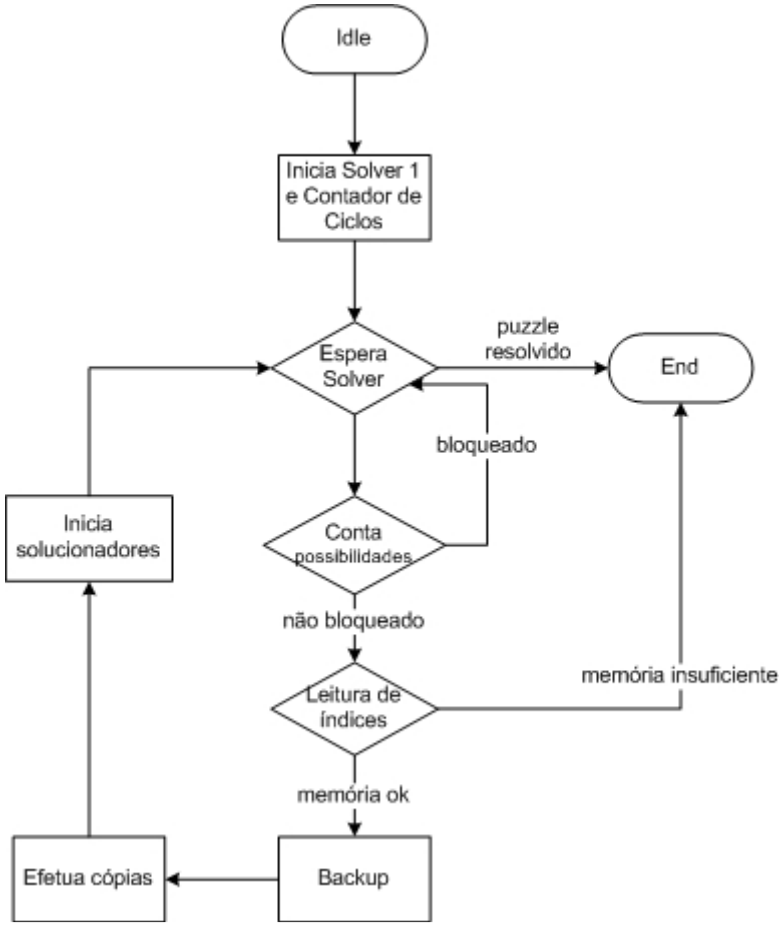


Figura 5.46 – Fluxograma da implementação em *hardware* da *Unidade de Controlo* do sistema com processamento paralelo

O fluxograma da Figura 5.46 traduz o conjunto de estados da *Unidade de Controlo*. De seguida é descrito o conjunto de operações de cada estado:

1. *Idle* – este é o estado inicial, pelo que durante este, o solucionador encontra-se transparente. Os solucionadores encontram-se todos desativados, no entanto o primeiro índice já se encontra preenchido, sinalizando que o primeiro solucionador possui um puzzle pronto a ser resolvido assim que seja ativo o sinal *enable* do solucionador;
2. *Inicia Solver 1 e Contador de Ciclos* – neste estado, é então iniciado o processo de resolução do primeiro puzzle, sendo ativado o sinal *enable* do Solver 1 (Figura 5.45) e dando início ao processo de contagem de ciclos;
3. *Espera Solver* – neste estado, é aguardado o término de cada um dos *N* solucionadores existentes. Solucionadores que possuam índice de ordem menor são atendidos em primeira instância, isto é, uma vez que os solucionadores são iniciados por ordem, também serão atendidos por ordem, por forma a manter uma organização do mapa de índices. Neste ponto é também avaliada a *flag solved* do solucionador que terminou, sendo que se esta *flag* se encontrar ativa, o puzzle está resolvido e pode-se dar o processo de resolução como terminado; caso contrário, é necessário iniciar a fase de tentativa e erro;
4. *Conta possibilidades* – este estado consiste num estado 2 em 1. Nele, é contado o número de possibilidades da posição do mínimo devolvida pelo solucionador. Este número de possibilidades não só ditará o número de cópias a ser efetuadas como também se o puzzle se encontra bloqueado ou não. Se a posição de mínimo corresponder a uma célula vazia e sem possibilidades, então o puzzle encontra-se bloqueado e, como tal, é terminado o solucionador em causa e retorna-se ao estado *Espera Solver*, por forma a aguardar pelo término de outro solucionador;
5. *Leitura de índices* – neste estado, é procurado o primeiro índice vazio que corresponde ao índice do primeiro solucionador disponível para onde serão feitas as cópias. Em simultâneo, é averiguado se existe memória disponível para efetuar a cópia do puzzle. Numa situação em que não exista memória suficiente para todas as cópias, são feitas tantas cópias quanto o número de índices livres;
6. *Backup* – neste estado, é efetuada uma cópia do puzzle do índice atual para o puzzle auxiliar;
7. *Efetua cópias* – neste estado, o puzzle guardado no puzzle auxiliar é então copiado para o primeiro índice vazio já determinado e índices seguintes consoante o número

de cópias necessárias. Em simultâneo, o mapa de índices é preenchido com as informações relativas a estes índices, isto é, são sinalizados como ocupados;

8. *Inicia solucionadores* – os solucionadores para onde foram efetuadas as cópias são inicializados através da ativação do seu sinal *enable* e o sistema retorna ao estado *Espera Solver*, por forma a aguardar pelo término de um próximo solucionador;
9. *End* – sendo este o estado final, é ativada a *flag end* responsável por sinalizar o fim do solucionador, sendo também atribuído um valor a uma *flag* auxiliar *memory not enough* responsável por indicar se houve ou não falta de memória durante o processo.

De acordo com fluxograma apresentado, note-se que a *Block RAM* do *Puzzle Auxiliar* e *Mapa de Índices* é externa aos solucionadores. Isto ocorre pelo facto da fase de tentativa e erro estar implementada fora de cada solucionador.

Assim, o número de máximo de solucionadores corresponderá ao número de *Block RAMs* disponíveis na *FPGA* que, de acordo com a secção 6.1, pode-se escrever:

Número total de *Block RAMs* disponíveis na *FPGA*: 28

Número de *Block RAMs* por *Solucionador*: 5
(Puzzle + Lista de Possibilidades + Mapa de Colunas + Mapa Caixas + Mapa Mínimos)

Número de *Block RAMs* da *Unidade de Controlo*: 2
(Puzzle Auxiliar + Mapa de Índices)

Número de *Block RAMs* disponível para Solucionadores: $28 - 2 = 26$

Número máximo de *Solucionadores*: $\lfloor 26 \div 5 \rfloor = 5$

Podendo implementar 5 Solucionadores, significa isto que, para além do Solucionador 1, onde estará contido o puzzle original, é possível ter 4 Solucionadores a funcionar em simultâneo. No entanto, o número de suposições, quanto ao valor de uma dada célula (tentativa e erro), fica reduzido apenas a 4 suposições, valor este muito reduzido.

A Figura 5.47 apresenta o diagrama de blocos da *Unidade de Controlo* deste sistema solucionador.

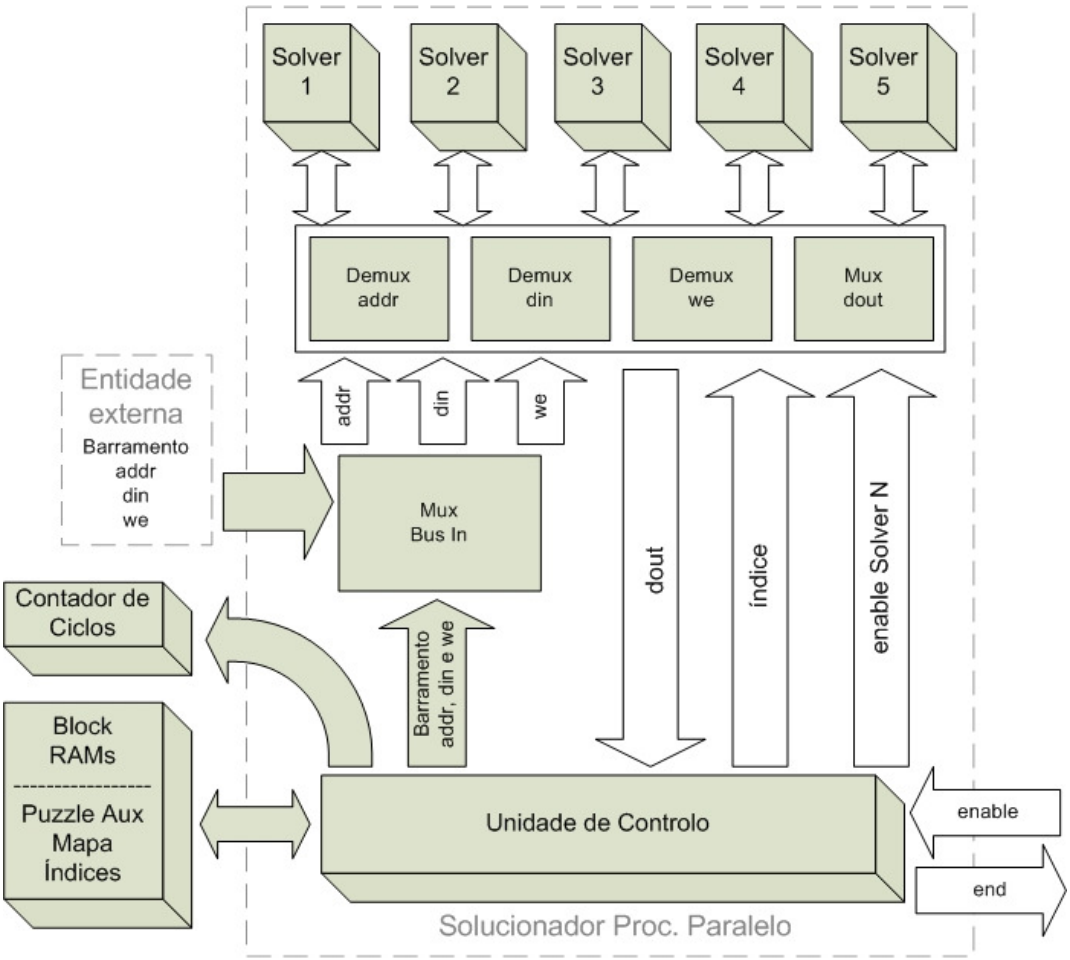
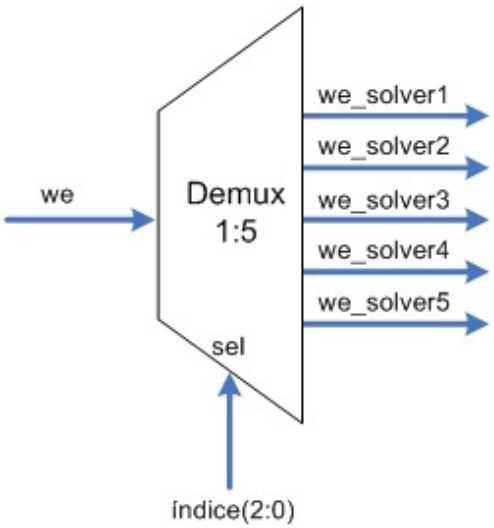


Figura 5.47 – Diagrama de blocos da *Unidade de Controlo* do sistema solucionador com processamento paralelo

A gestão do acesso ao barramento dos solucionadores é feita através dos *Demux* representados na Figura 5.47. Estes são responsáveis por receber os dados do barramento da *Unidade de Controlo* ou *Entidade externa* e redirecioná-los para um dos cinco solucionadores, consoante o valor do índice, que atuará como sinal de seleção. Do mesmo modo, consoante o valor deste, é selecionada a saída de um dos 5 solucionadores através do *multiplexer Mux dout*. A implementação destes é feita de acordo com os interfaces e tabelas de verdade apresentadas a seguir.

De notar que não existe índice 000_2, uma vez que o puzzle original já estará contido no solucionador 1 que corresponde ao índice 1 (001_2). A tabela de verdade dos outros 2

demultiplexers é exatamente igual a esta (Figura 5.48), mudando apenas os sinais de entrada e saída, de acordo com o seu funcionamento (endereços ou dados de entrada), pelo que não há necessidade de apresentar o seu esquema.



Entradas			Saídas
sel_2	sel_1	sel_0	
0	0	1	we_solver1
0	1	0	we_solver2
0	1	1	we_solver3
1	0	0	we_solver4
1	0	1	we_solver5

Figura 5.48 – Interface do *Demux write enable* (*we*) e respetiva tabela de verdade

O *multiplexer Mux dout* é um simples multiplexador das saídas dos solucionadores, atuando de acordo com o seu sinal de seleção *índice*. Assim sendo, também não é necessário apresentar aqui o seu esquema.

Por fim, o *multiplexer Mux Bus In*, tal como anteriormente, é responsável por selecionar a entidade responsável por aceder aos solucionadores, quer seja esta *Externa* ou U*nidade de Controlo*. Como tal, o seu interface e tabela de verdade são os mesmos da Figura 5.40.

5.3 Conclusões

Neste capítulo foram descritas as implementações em *hardware* dos métodos lógicos apresentados na secção 2.3.2. A descrição destas baseou-se na apresentação e explicação das respetivas máquinas de estados.

Foram também descritas as implementações dos três solucionadores: Simples, Tentativa e Erro e Tentativa e Erro com Processamento Paralelo. A implementação destes

foi feita tendo por base os métodos apresentados na secção 2.3.2, e adicionalmente um conjunto de operações, descritas no fluxograma da Figura 5.29, Figura 5.35 e Figura 5.46.

Além destas, também foi descrita a *Unidade de Controlo*, no que concerne à gestão de acesso às memórias por parte dos vários processos, originando, assim, os vários solucionadores. Esta gestão foi feita recorrendo ao uso de *multiplexers*, *demultiplexers* e codificadores prioridade.

6 Resultados

Neste capítulo, serão apresentados os resultados das várias implementações mencionadas. Estes consistirão nos tempos de processamento necessários para resolver os diversos puzzles e recursos usados na *FPGA* por cada uma das três abordagens. É aqui também feita uma breve descrição da placa de prototipagem usada para efeitos de teste.

De notar que uma vez que não foi implementada a comunicação entre o *PC* e a Placa para transferência de dados, os testes na *FPGA* foram feitos inicializando a *Block RAM Puzzle* com os valores das células do puzzle já preenchidas, e no *PC* foi feito o mesmo com o *Array* do *Puzzle*. De notar, também, que os resultados em *hardware* foram obtidos em número de ciclos de relógio, pelo que, o tempo de execução é calculado com base na frequência máxima de operação (f_{max}) para cada solucionador. No entanto, os solucionadores simples e tentativa e erro com processamento paralelo, foram testados usando a frequência do cristal da placa, 50 *MHz* (secção 6.1); já para o solucionador tentativa e erro, dada a sua frequência máxima de operação ser inferior, este foi testado usando uma frequência de 25 *MHz*, através de um divisor de frequência de relógio. No que concerne aos resultados em *software*, estes foram obtidos usando uma *CPU Intel Core 2 Duo*, com frequência de operação 1.87 *GHz*.

Os puzzles usados (consultar Apêndice A) para os testes são os mesmos em cada abordagem, quer para a implementação em *software* quer para a de *hardware*.

6.1 Placa de prototipagem

Para execução destes solucionadores recorreu-se a uma placa de prototipagem da *Digilent*: a placa *Nexys 2*. Apesar de muitas das funcionalidades não terem sido usadas (Figura 6.1), esta constitui uma boa placa, reunindo um conjunto de características bastante razoável, tendo em conta o seu preço no mercado.

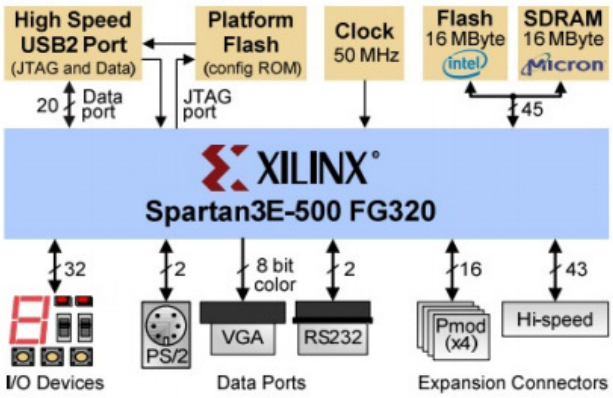


Figura 6.1 – Diagrama de blocos da placa *Nexys 2* [40]

Da Figura 6.1, destacam-se os dispositivos I/O, como botões, *switches* e *leds,* que desempenharam um papel importante no teste das várias máquinas implementadas, e a *FPGA Spartan 3E-1200 FG320*, sendo esta igual à representada na figura, mas de maior capacidade. Esta última, apesar de fazer parte da gama *low-cost* da *Xilinx,* reúne as condições necessárias para a implementação e teste deste projeto. As suas características podem ser observadas na Tabela 6.1.

Tabela 6.1 – Recursos da *FPGA Spartan 3E-1200 FG320* [40]

FPGA	Portas	CLBs	Slices	LUTs / Flip-flops	RAM16	RAM Distribuída (bits)
XC3S1200E	1200K	2 168	8 672	17 344	8 672	138 752

6.2 Sistema solucionador simples

Para as implementações simples, foram usados puzzles simples [42], todos eles passíveis de ser resolvidos com este método. Os resultados obtidos para o tempo de processamento podem ser observados na Tabela 6.2.

Tabela 6.2 – Tabela dos resultados do tempo de processamento para a implementação do solucionador simples, f_{max} = 55.6 *MHz*

Puzzle	Software (ms)	Hardware (ms)
Easy Puzzle 1	0.92604	1.425
Easy Puzzle 2	0.58681	0.794931
Easy Puzzle 3	0.76751	1.20079
Easy Puzzle 4	0.98223	1.67786
Medium Puzzle 1	2.04131	3.91976
Medium Puzzle 2	3.75556	1.56605
Medium Puzzle 3	1.73127	2.02714
Medium Puzzle 4	2.87173	3.36064

Estes primeiros resultados mostram, que à medida que a complexidade do puzzle aumenta, o tempo necessário para resolver o puzzle também aumenta, tal como esperado. Isto pode ser facilmente observado na tabela, comparando os resultados dos puzzles com prefixo *Easy* e prefixo *Medium*.

Além disto, é evidente a discrepância existente nos tempos do *Software* comparativamente aos do *Hardware* (Figura 6.2), sendo o *Hardware* mais lento, tal como também seria de esperar, dadas as características da *FPGA* e frequência de relógio máxima permitida, 55.6 *MHz*. Em média, para resolver o mesmo puzzle, o *Hardware* demora 35% mais tempo que o *Software* (Figura 6.3).

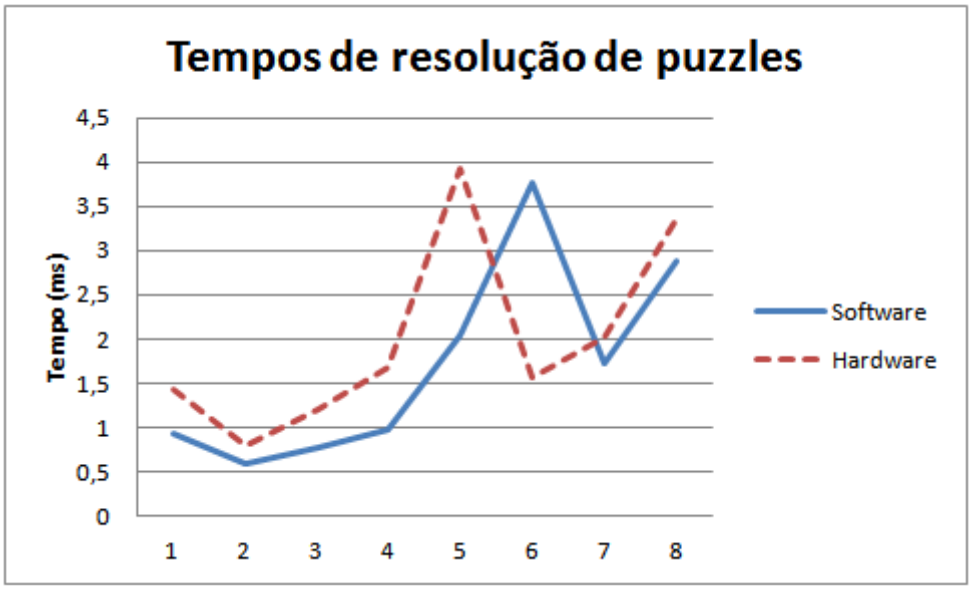


Figura 6.2 – Gráfico dos tempos de resolução de puzzles para o solucionador simples

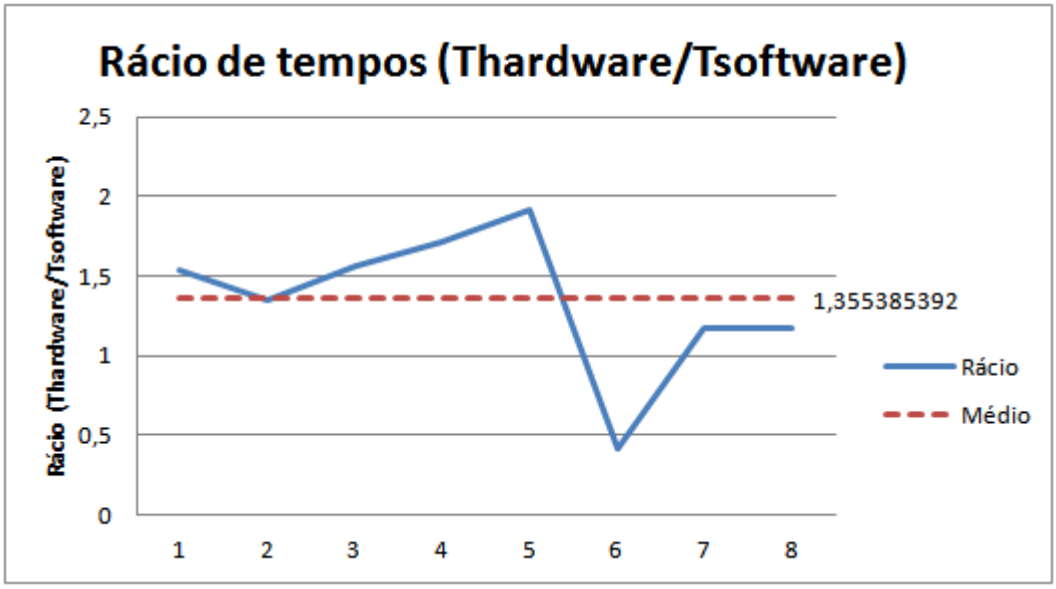


Figura 6.3 – Gráfico do rácio de tempos de resolução de puzzles para o solucionador simples

Com estes primeiros resultados (Tabela 6.2, Tabela 6.3), é possível ter uma noção do comportamento da *FPGA* face ao *PC* e, assim, retirar ilações mais sólidas para as implementações seguintes.

Tabela 6.3 – Utilização de recursos da *FPGA* para o solucionador simples

Recursos	Usados	Disponíveis	Utilização
Slices	529	8 672	6%
LUTs	974	17 344	5%
Block RAMs	4	28	14%

6.3 Sistema solucionador com tentativa e erro

Para as implementações com tentativa e erro, foram usados puzzles [42-44] cujo número de tentativas necessárias para atingir a solução final fosse variável, mas que fosse comportado pela memória disponível. Por outras palavras, foram usados puzzles de graus de dificuldade diferente, mas que o número de tentativas possa ser suportado pela memória disponível na *FPGA*.

Deste modo, os resultados para os tempos de processamento obtidos para esta implementação podem ser observados na Tabela 6.4.

Tabela 6.4 – Tabela dos resultados do tempo de processamento para a implementação do solucionador com tentativa e erro, f_{max} = 43.4 *MHz*

Puzzle	Iterações	Software (ms)	Hardware (ms)
Hard Puzzle 5	4	5.62525	14.1647
Hard Puzzle 9	4	6.9422095	20.6003
Hard Puzzle 12	2	4.3436325	8.1463
Expert Puzzle 1	3	3.9939015	6.67655
Expert Puzzle 2	2	4.778578	22.0562
Extreme Puzzle 5	1	3.3893245	11.0064
Evil Puzzle 1	1	5.533839	22.1446
Evil Puzzle 2	3	5.3998865	10.5393
Evil Puzzle 3	1	2.915443	6.25332
Extra Chall 13	1	4.918234	18.8609
Evil Puzzle 8	5	5.8148165	12.8761
Evil Puzzle 17	1	4.1525255	6.21172
Evil Puzzle 5	2	5.223146	9.2453
Hard Puzzle 10	3	3.3637205	7.34415
Expert Puzzle 10	1	2.7478945	15.2092
Expert Puzzle 10a	4	5.874304	8.20003
Hard Puzzle 10a	8	4.8594695	11.024
Evil Puzzle 3a	4	5.277493	10.5094
Evil Puzzle 5a	7	11.522501	24.5721
Evil Puzzle 17a	7	10.333517	18.1601

Observando os resultados da Tabela 6.4, deu-se um claro aumento dos tempos de resolução face à implementação simples. Tal é devido ao conjunto de processos extra envolvidos nesta implementação, assim como ao número de iterações exigidas. Estas correspondem ao número de nós na árvore de pesquisa (Figura 4.23) que vai sendo construída durante o processo de solução.

O aumento do número de processos envolvidos torna-se mais crítico do lado do *Hardware*, uma vez que existem mais operações a ser executadas a um ritmo (43.4 *MHz*) inferior ao do *Software*. Esta causa leva a que a discrepância nos tempos de resolução entre as duas plataformas aumente (Figura 6.4), pelo que em média, o *Hardware* demora cerca de duas vezes mais tempo a resolver o mesmo puzzle (Figura 6.5).

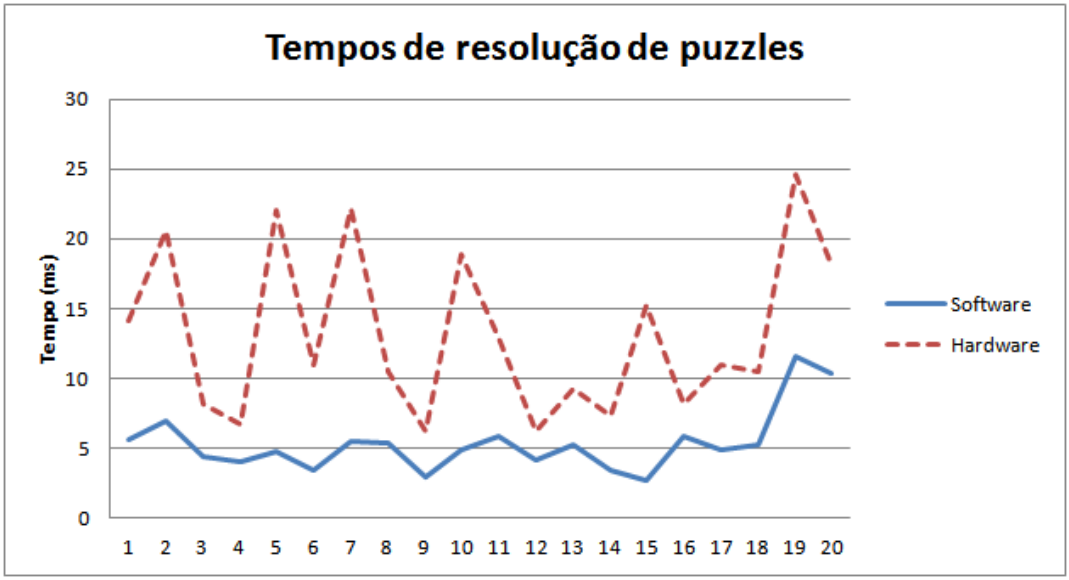


Figura 6.4 – Gráfico dos tempos de resolução de puzzles para o solucionador com tentativa e erro

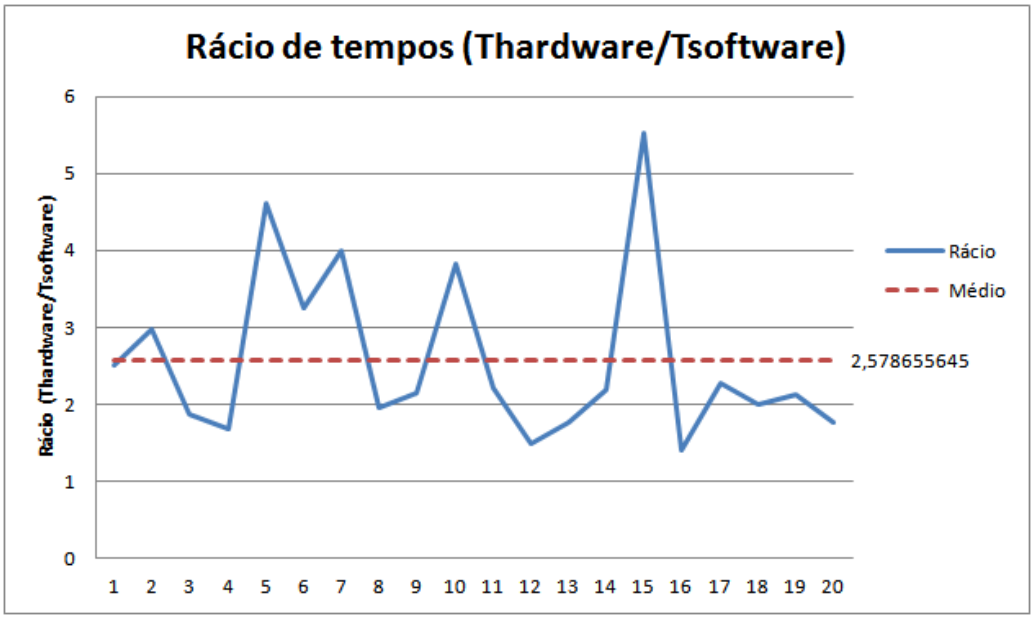


Figura 6.5 – Gráfico do rácio de tempos de resolução de puzzles para o solucionador com tentativa e erro

Não obstante a relação existente entre os tempos de resolução (ver Figura 6.5), os resultados obtidos em *Hardware* são bastante satisfatórios, dada não só a frequência de funcionamento da placa, mas também o facto de se estar a usar uma *FPGA low-cost*.

Por observação da Tabela 6.5, é possível constatar um claro aumento dos recursos face aos recursos usados na implementação do solucionador simples. Este aumento deve-se

ao aumento dos processos envolvidos neste. De notar que, apesar de esta implementação permitir resolver puzzles com número máximo de 21 suposições, através do uso da totalidade das *Block RAMs* disponíveis (ver secção 5.2.2), para efeitos de teste, tal não foi feito e, por isso, a utilização destas foi de apenas 67% que foi suficiente para os testes efetuados.

Tabela 6.5 – Utilização de recursos da *FPGA* para o solucionador com tentativa e erro

Recursos	Usados	Disponíveis	Utilização
Slices	961	8 672	11%
LUTs	1 757	17 344	10%
Block RAMs	19	28	67%

Apesar do grande aumento de recursos, estes constituem apenas um pequeno percentual da capacidade da *FPGA*, excetuando-se as *Block RAMs*. De notar que o aumento temporal na resolução dos puzzles, apesar de evidente, não constitui um mau resultado, tendo em conta o arsenal de puzzles passíveis de ser resolvidos com a implementação com tentativa e erro.

6.4 Tentativa e erro versus tentativa e erro com processamento paralelo

Uma vez que o algoritmo de tentativa e erro com processamento paralelo não é capaz de resolver puzzles que necessitem de um grande número de iterações para ser resolvidos devido aos recursos usados, para esta fase foram apenas usados puzzles que não necessitem de mais do que 4 iterações, isto é, não sejam feitas mais de 4 suposições relativamente ao valor de uma dada célula (de acordo com a secção 5.2.3). Os resultados obtidos encontram-se sintetizados na Tabela 6.6.

Tabela 6.6 – Tabela dos resultados do tempo de processamento para a implementação do solucionador com processamento paralelo e comparação com o solucionador sem processamento paralelo, $f_{max} = 51.7\ MHz$

Puzzle	Iterações	Software (ms)	Hardware Sem Proc. Paralelo (ms)	Hardware Com Proc. Paralelo (ms)
Hard Puzzle 5	4	5.62525	14.1647	5.65
Hard Puzzle 9	4	6.9422095	20.6003	5.42
Hard Puzzle 12	2	4.3436325	8.1463	4.91
Expert Puzzle 1	3	3.9939015	6.67655	4.14
Expert Puzzle 2	2	4.778578	22.0562	4.98
Extreme Puzzle 5	1	3.3893245	11.0064	5.14
Evil Puzzle 1	1	5.533839	22.1446	9.57
Evil Puzzle 2	3	5.3998865	10.5393	7.22
Extra Chall 13	1	4.918234	18.8609	7.25
Evil Puzzle 17	1	4.1525255	6.21172	5.22
Evil Puzzle 5	2	5.223146	9.2453	5.63
Hard Puzzle 10	3	3.3637205	7.34415	5.08
Expert Puzzle 10	1	2.7478945	15.2092	3.96

Os resultados temporais para esta fase são bastante satisfatórios. Como se pode observar na Figura 6.6, deu-se uma melhoria significativa nos tempos da implementação com processamento paralelo relativamente à implementação sem processamento paralelo.

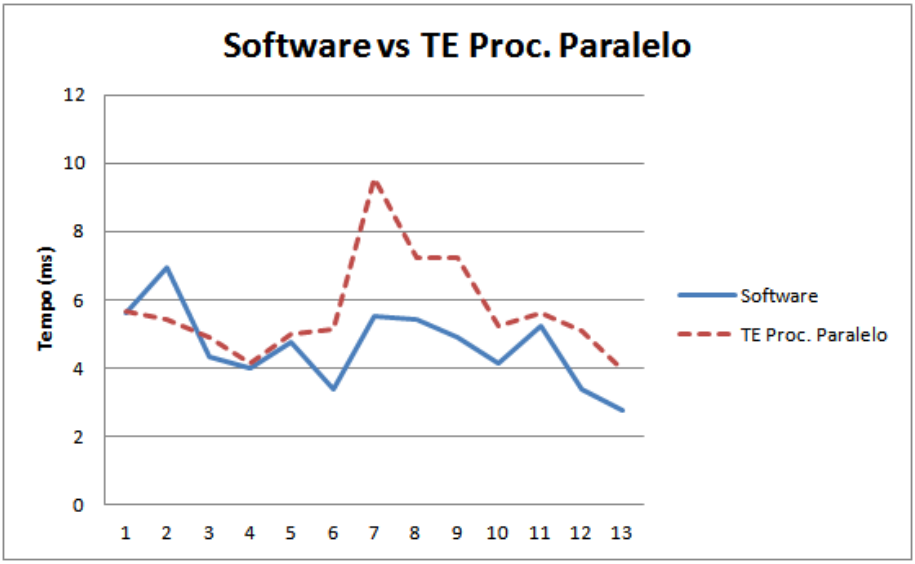


Figura 6.6 – Gráfico dos tempos de resolução de puzzles para o solucionar com tentativa e erro implementado com processamento paralelo

Comparativamente aos resultados obtidos em *software*, estes são bastante próximos, não existindo uma discrepância significativa como anteriormente (ver Figura 6.7). De notar que o puzzle *Hard Puzzle 9*, assinalado a azul, obteve um resultado bastante bom, tendo superado o tempo de resolução em *software*.

Com estes resultados, é possível provar o desempenho que é conseguido com uma *FPGA*.

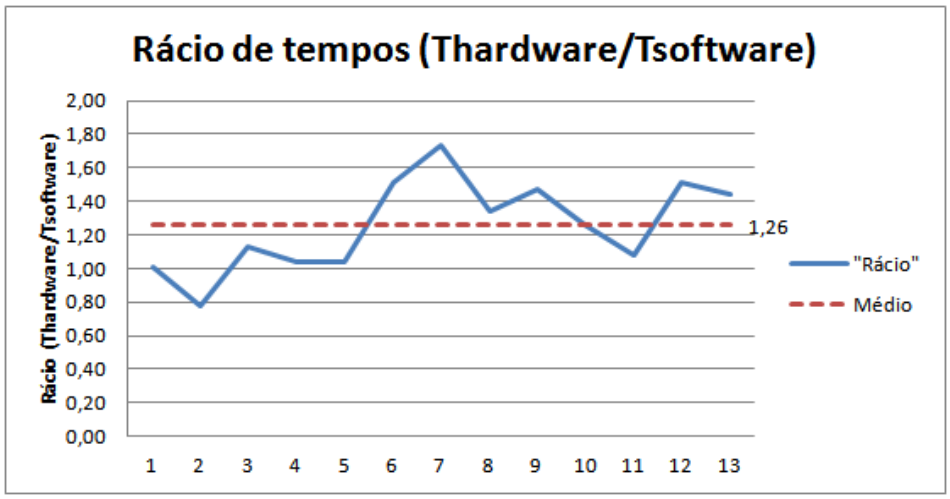


Figura 6.7 – Gráfico do rácio de tempos de resolução de puzzles para o solucionar com tentativa e erro implementado com processamento paralelo

Tal como esperado, esta implementação é a que requer um maior número de recursos usados (Tabela 6.7). Isto deve-se ao facto de serem usados vários solucionadores em paralelo, e, por conseguinte, usados vários processos em simultâneo.

Nesta fase é importante comparar os resultados entre estas duas implementações (Tabela 6.8), averiguando até que ponto o processamento paralelo será vantajoso para este tipo de aplicação.

Tabela 6.7 – Utilização de recursos da *FPGA* para o solucionador com tentativa e erro implementado com processamento paralelo

Recursos	Usados	Disponíveis	Utilização
Slices	4 297	8 672	49%
LUTs	7 578	17 344	43%
Block RAMs	27	28	96%

Tabela 6.8 – Tabela síntese dos recursos usados pelos solucionadores Tentativa e Erro e Processamento Paralelo

	Tentativa e Erro	Tentativa e Erro com Processamento Paralelo
Block RAMs	100%	96%
Nº de iterações	21	4
Slices	11%	49%
LUTs	10%	43%
Rácio Médio	2.578	1.260

Assim sendo, o primeiro aspeto a ser comparado é o número de *Block RAMs* usado face ao número de iterações que cada implementação consegue comportar. Ambas as implementações fazem uso intensivo destas memórias, contudo o número de iterações associado a cada uma delas varia bastante, tal como se pode observar na Tabela 6.8.

Deste ponto de vista, a solução Tentativa e Erro constitui não só uma melhor solução, como uma solução mais capaz, tendo capacidade para resolver puzzles de graus de complexidade superiores.

O segundo aspeto comparativo é o número de *Slices* e *LUTs* disponíveis que poderão ser usadas para otimizar a implementação e, porventura, poupar *Block RAMs* que serão usadas para um número de iterações maior. Tendo em conta este aspeto, apesar da implementação Tentativa e Erro usar um número menor de recursos, ambas as implementações têm bastantes recursos disponíveis, pelo que teria que ser feito um estudo que relacionasse o número de *Slices* e *LUTs* a serem usados com o número de *Block RAMs* poupado. Assim, sob este ponto de vista, ambas as implementações têm a capacidade de esbanjar recursos, por forma a libertar algumas das *Block RAMs* usadas, ou otimizar o seu uso.

Como conclusão, na implementação com processamento paralelo, existe uma grande utilização de *Block RAMs* e de recursos, assim como também se sacrifica o número de iterações a troco de um aumento de desempenho.

6.5 Comparação com outros solucionadores

Como se pode constatar, todos os três solucionadores implementados apenas resolvem puzzles de ordem N = 3. Desde modo, tomar-se-ão os puzzles 3a e 3b,

disponibilizados como instância de teste na *FPT'09* [7], por forma a comparar os resultados obtidos com os solucionadores mencionados na secção 2.4.

Tomando por base a Tabela 2.5, os resultados relativos a estes puzzles podem ser sintetizados na seguinte Tabela 6.9.

Tabela 6.9 – Comparação de resultados obtidos para puzzles ordem N = 3

Solucionador	3a (ms)	3b (ms)
Simples	4.15	-
Tentativa e Erro	9.1	-
TE Proc. Paralelo	6.09	-
TU Delft	20.821	12.460
Processador específico	8.600	9.762
Solucionador SSAS	-	-
ECP	0.257	3.178

Uma vez que o puzzle *3a* é passível de ser resolvido recorrendo apenas a métodos heurísticos, este pode ser solucionado com todos os três solucionadores implementados. Como se pode observar na Tabela 6.9, tanto o solucionador *Simples*, como o solucionador *Tentativa e Erro* implementado com *Processamento Paralelo* apresentam bons resultados, tendo obtido, respetivamente, o segundo e terceiro resultado mais rápido. Já o solucionador *Tentativa e Erro*, tal como se tem vindo a verificar, este é mais lento. Não obstante o seu resultado, este é satisfatório, tendo-se aproximado do resultado do solucionador *Processador específico*, cuja implementação é bastante idêntica.

Relativamente ao puzzle *3b*, este necessita de 57 iterações (Figura 6.8), valor que excede a capacidade de resolução dos solucionadores implementados. Desta forma, não foi possível obter o tempo de execução do mesmo.

```
 * Solucao

 ** Numero de iteracoes: 57

3 2 5 ¦ 4 7 8 ¦ 9 6 1
1 4 6 ¦ 2 3 9 ¦ 5 7 8
8 7 9 ¦ 1 5 6 ¦ 2 3 4
------+-------+------
9 1 2 ¦ 7 4 3 ¦ 8 5 6
6 3 4 ¦ 5 8 2 ¦ 1 9 7
5 8 7 ¦ 6 9 1 ¦ 4 2 3
------+-------+------
4 5 1 ¦ 9 6 7 ¦ 3 8 2
2 6 3 ¦ 8 1 5 ¦ 7 4 9
7 9 8 ¦ 3 2 4 ¦ 6 1 5

Software execution time:

0.06392 seconds
63.92152 miliseconds
```

Figura 6.8 – Número de iterações necessárias para solucionar o puzzle 3b

Por fim, importa referir que os resultados obtidos, ainda que apenas para o puzzle 3a, são bastante satisfatórios quando comparados com os outros solucionadores, que para o efeito usaram *FPGAs* mais avançadas (secção 2.4). Na Figura 6.9 podem ser comparados os recursos utilizados por cada um dos solucionadores, tendo os solucionadores mais complexos (*SSAS* e *Processador Específico*) utilizado um maior número de recursos, tal como esperado, dada a natureza da sua implementação.

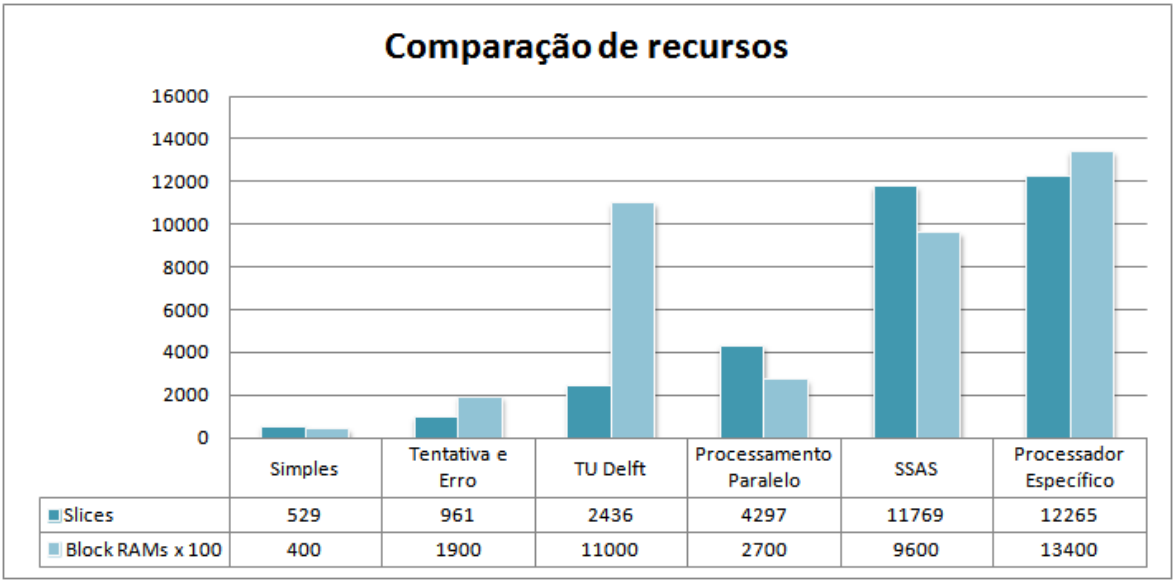


Figura 6.9 – Comparação de recursos usados pelos vários solucionadores

6.6 Conclusões

Neste capítulo foram apresentados os resultados obtidos para cada um dos três solucionadores implementados em *hardware*, tendo estes sido comparados com os obtidos em *software*. Desta comparação, concluiu-se que a implementação com processamento paralelo é a que mais se aproxima da execução em *software*, evidenciando o desempenho da *FPGA*.

Foram, também, comparados os resultados obtidos entre os solucionadores implementados e os mencionados na secção 2.4, para o puzzle *3a*. Apesar da incapacidade de resolver puzzles de ordem superior, assim como o número de iterações limitado, os três solucionadores obtiveram resultados bastante satisfatórios, quando comparados com solucionadores implementados com *FPGAs* mais avançadas.

7 Conclusão

Este capítulo constitui a finalização da dissertação, sendo aqui apresentadas as principais conclusões da realização deste projeto e objetivos atingidos. Serão aqui também apresentadas algumas propostas para trabalho futuro, assim como algumas publicações.

7.1 Trabalho realizado

Na primeira parte deste trabalho, foi realizado um estudo acerca do conjunto de métodos lógicos usados na resolução de puzzles Sudoku. Foram implementados, com sucesso, alguns destes métodos, assim como ferramentas de auxílio a estes. Do ponto de vista do *Software*, não sucederam quaisquer complicações na conceção destes, no entanto, o mesmo já não ocorreu em *Hardware*, existindo algumas dificuldades em criar máquinas de estados finitos simples e de fácil perceção. Além disto, foram registadas várias situações onde os atrasos físicos comprometiam o bom funcionamento destas, tendo sido necessário alterar os estados destas máquinas, mudando operações de estados e perdendo, assim, algum desempenho com a introdução de estados redundantes. Por este motivo, estas máquinas não se encontram totalmente otimizadas.

Na segunda parte deste trabalho, foram implementados os três solucionadores: Simples, Tentativa e Erro, Tentativa e Erro com Processamento Paralelo, fazendo uso dos métodos lógicos já implementados. Para gerir estes processos, foi necessário criar uma unidade de gestão de acesso aos barramentos de dados, controlo e endereços das memórias, a *Unidade de Controlo*. Esta gere o acesso aos barramentos como também a ordem pela qual os processos são executados, constituindo esta o solucionador.

Tal como anteriormente, em *Hardware* surgiram situações indesejadas devido aos atrasos, sendo necessário alterar novamente algumas das *FSM* (*Finite State-Machines*) implementadas. No entanto, todos os três solucionadores foram implementados e testados com sucesso.

A implementação destes três solucionadores demonstrou as capacidades da *FPGA*, quer em termos de recursos quer em termos de desempenho, tendo os melhores resultados sido obtidos para a abordagem com processamento paralelo. Não obstante, com a implementação do processamento paralelo conseguiu-se provar que com uma *FPGA low-*

-cost é possível atingir um grande desempenho, que, em outras aplicações computacionalmente exigentes pode-se revelar uma mais-valia, especialmente com o uso de *FPGAs* mais avançadas.

Por fim, conclui-se que os objetivos propostos para este trabalho foram atingidos.

7.2 Trabalho futuro

Por forma a melhorar este projeto, são apresentadas as seguintes propostas para trabalho futuro:

- Otimização das máquinas de estados finitos de cada processo implementado por forma a reduzir o tempo de execução dos mesmos;
- Otimização da implementação, quer em *software*, quer em *hardware*, das funções *Singles* e *Hidden Number*, nomeadamente no que concerne à atualização do número de candidatos de cada célula e aos ciclos de contagem de números, respetivamente;
- Otimização das implementações em *software* dos solucionadores, mais concretamente a troca do estado *Lista de Possibilidades* com a decisão *r*;
- Desenvolvimento de um sistema de gestão de acesso aos mapas (coluna, caixa) para o processamento paralelo por forma a que estes se encontrem na *Unidade de Controlo* e não em cada solucionador, poupando desta forma *Block RAMs* que serão úteis para a implementação de mais solucionadores. Esta gestão poderá ser conseguida recorrendo a um *FIFO (First In, First Out*), atendendo os solucionadores por ordem de pedido de acesso;
- Otimização da memória da *FPGA*, usando, por exemplo, *Dual Port RAMs* por forma a armazenar mais que um puzzle por *Block RAM*;
- Implementação da comunicação *USB* (*Universal Serial Bus*) para transferência de dados entre a placa e o computador;
- Estudo da possibilidade de implementação dos processos para resolução de puzzles de ordem superior a 3, isto é, matrizes de dimensão superior a 9x9.
- Aumento do número de iterações suportado pelo solucionador tentativa e erro com processamento paralelo, permitindo, deste modo, resolver puzzles mais complexos

(ex.: 3b da *FPT'09*) e aumentar o seu desempenho. A conclusão mais importante é que o paralelismo deve ser aplicado mais agressivamente.

7.3 Publicações

Como resultado deste trabalho, foram escritos dois artigos.

O primeiro, submetido numa conferência internacional, aborda as implementações dos três solucionadores, fazendo uma breve descrição de cada um dos algoritmos e sua arquitetura. Por fim, são apresentados os resultados, sendo estes comparados com os da Conferência Internacional *FPT'09*, assim como também as principais conclusões deste trabalho.

O segundo, foi aceite para publicação na revista Electrónica e Telecomunicações do Departamento de Electrónica, Telecomunicações e Informática e Universidade de Aveiro. Este, tal como o primeiro, aborda também a implementação dos algoritmos e arquiteturas dos solucionadores, apresentando por fim os seus resultados.

Os artigos:

- I. Skliarova, T. Vallejo, V. Sklyarov, "FPGA-based Implementation of Sudoku Solvers", submitted to the 16th IEEE Mediterranean Electrotechnical Conference, March 25-28, 2012, Medina, Tunisia.

- T. Vallejo, "Sudolu em FPGA", aceite para a revista Electrónica e Telecomunicações, Departamento de Electrónica, Telecomunicações e Informática, Universidade de Aveiro.

Apêndice A

Lista de puzzles utilizados para efeitos de teste

Easy Puzzle 1

2	1	7	6	4	3	8	5	9
4	6	9	8	5	2	1	7	3
5	8	3	7	1	9	6	2	4
6	5	2	3	7	8	4	9	1
8	7	4	1	9	5	2	3	6
9	3	1	4	2	6	5	8	7
7	9	6	5	8	1	3	4	2
1	2	5	9	3	4	7	6	8
3	4	8	2	6	7	9	1	5

Easy Puzzle 2

4	6	7	8	2	5	1	3	9
5	1	3	9	7	4	8	2	6
8	9	2	6	3	1	4	7	5
1	2	8	7	6	3	9	5	4
9	5	6	4	8	2	7	1	3
3	7	4	5	1	9	2	6	8
7	3	5	1	9	8	6	4	2
6	4	9	2	5	7	3	8	1
2	8	1	3	4	6	5	9	7

Easy Puzzle 3

6	1	9	4	5	8	2	3	7
7	8	5	2	3	6	1	9	4
2	3	4	9	1	7	5	8	6
8	4	7	6	9	2	3	1	5
3	9	1	7	4	5	8	6	2
5	6	2	1	8	3	7	4	9
9	5	6	3	7	1	4	2	8
1	2	8	5	6	4	9	7	3
4	7	3	8	2	9	6	5	1

Easy Puzzle 4

8	5	6	4	3	7	9	2	1
4	9	2	1	6	8	7	5	3
3	1	7	5	2	9	8	4	6
5	6	8	9	1	4	3	7	2
2	4	9	7	8	3	6	1	5
7	3	1	6	5	2	4	9	8
6	2	4	3	9	1	5	8	7
9	8	3	2	7	5	1	6	4
1	7	5	8	4	6	2	3	9

Medium Puzzle 1

1	9	2	3	5	6	8	7	4
4	5	8	1	7	2	3	6	9
3	6	7	8	4	9	2	1	5
2	3	6	7	9	5	1	4	8
7	4	1	6	2	8	9	5	3
5	8	9	4	3	1	6	2	7
6	7	4	9	1	3	5	8	2
9	1	5	2	8	4	7	3	6
8	2	3	5	6	7	4	9	1

Medium Puzzle 2

3	9	6	1	4	7	8	2	5
2	5	1	8	3	6	9	7	4
4	7	8	5	2	9	3	6	1
1	3	7	6	9	2	5	4	8
6	8	9	4	5	1	2	3	7
5	4	2	3	7	8	6	1	9
9	2	4	7	8	3	1	5	6
7	6	3	9	1	5	4	8	2
8	1	5	2	6	4	7	9	3

Medium Puzzle 3

6	2	9	5	3	4	1	7	8
5	3	4	7	1	8	2	9	6
1	8	7	2	9	6	3	4	5
8	5	3	4	6	7	9	1	2
7	1	2	8	5	9	6	3	4
9	4	6	3	2	1	8	5	7
3	7	8	1	4	2	5	6	9
2	9	1	6	7	5	4	8	3
4	6	5	9	8	3	7	2	1

Medium Puzzle 4

8	4	5	3	1	2	6	7	9
2	7	3	9	4	6	5	1	8
1	9	6	5	7	8	4	2	3
5	3	7	8	6	1	9	4	2
6	8	4	7	2	9	1	3	5
9	2	1	4	3	5	7	8	6
3	6	8	1	9	4	2	5	7
4	5	2	6	8	7	3	9	1
7	1	9	2	5	3	8	6	4

Hard Puzzle 5

7	8	2	6	1	4	5	9	3
3	6	1	5	2	9	4	8	7
5	9	4	8	3	7	2	1	6
2	1	3	7	6	8	9	5	4
9	7	8	4	5	1	6	3	2
4	5	6	2	9	3	1	7	8
1	4	9	3	8	2	7	6	5
6	3	7	1	4	5	8	2	9
8	2	5	9	7	6	3	4	1

Hard Puzzle 9

5	7	4	1	3	6	2	9	8
3	6	2	9	8	7	5	1	4
1	9	8	5	2	4	7	6	3
9	8	1	7	6	5	4	3	2
4	3	7	8	1	2	9	5	6
2	5	6	4	9	3	1	8	7
8	4	5	6	7	1	3	2	9
7	2	9	3	5	8	6	4	1
6	1	3	2	4	9	8	7	5

Hard Puzzle 12

9	3	2	1	8	7	4	6	5
1	8	4	3	6	5	2	9	7
7	6	5	4	9	2	8	3	1
8	7	6	2	5	4	3	1	9
5	9	3	8	1	6	7	2	4
2	4	1	9	7	3	6	5	8
6	1	9	7	3	8	5	4	2
4	5	8	6	2	1	9	7	3
3	2	7	5	4	9	1	8	6

Expert Puzzle 1

3	8	6	2	1	7	5	9	4
5	2	9	8	6	4	7	3	1
7	4	1	3	5	9	8	6	2
1	7	3	4	2	8	6	5	9
6	9	2	1	7	5	4	8	3
4	5	8	9	3	6	2	1	7
2	3	5	7	8	1	9	4	6
9	6	7	5	4	3	1	2	8
8	1	4	6	9	2	3	7	5

Expert Puzzle 2

7	6	9	3	5	2	4	1	8
4	5	8	1	9	7	6	3	2
3	2	1	8	6	4	7	9	5
6	7	5	2	8	9	1	4	3
2	1	4	7	3	6	5	8	9
9	8	3	5	4	1	2	7	6
1	3	2	9	7	5	8	6	4
8	4	7	6	2	3	9	5	1
5	9	6	4	1	8	3	2	7

Extreme Puzzle 5

5	4	3	2	1	9	8	7	6
1	8	2	7	6	5	9	4	3
9	7	6	8	4	3	2	1	5
8	9	7	6	3	4	1	5	2
2	5	1	9	8	7	6	3	4
6	3	4	5	2	1	7	9	8
7	2	9	4	5	8	3	6	1
4	1	8	3	9	6	5	2	7
3	6	5	1	7	2	4	8	9

Evil Puzzle 1

3	8	5	7	1	2	9	4	6
4	6	7	8	9	3	1	5	2
9	2	1	6	4	5	8	3	7
1	3	2	9	5	4	6	7	8
8	4	9	2	6	7	3	1	5
5	7	6	3	8	1	2	9	4
7	9	3	4	2	6	5	8	1
2	1	8	5	7	9	4	6	3
6	5	4	1	3	8	7	2	9

Evil Puzzle 2

2	3	4	6	7	1	8	5	9
5	6	7	2	8	9	1	3	4
9	1	8	4	5	3	7	6	2
3	4	9	5	6	8	2	7	1
7	5	6	1	4	2	3	9	8
8	2	1	9	3	7	6	4	5
1	8	5	7	9	6	4	2	3
4	7	3	8	2	5	9	1	6
6	9	2	3	1	4	5	8	7

Evil Puzzle 3

8	9	4	7	6	1	2	3	5
5	2	3	9	4	8	6	1	7
7	1	6	3	5	2	4	9	8
9	4	2	6	8	7	3	5	1
3	8	5	1	9	4	7	2	6
1	6	7	2	3	5	9	8	4
6	7	8	5	2	3	1	4	9
2	5	9	4	1	6	8	7	3
4	3	1	8	7	9	5	6	2

Extra Chall 13

3	9	2	6	8	4	5	1	7
8	4	7	2	5	1	9	6	3
6	1	5	7	3	9	2	4	8
5	6	9	1	7	8	4	3	2
4	2	1	9	6	3	8	7	5
7	8	3	5	4	2	1	9	6
9	7	6	8	1	5	3	2	4
1	3	8	4	2	6	7	5	9
2	5	4	3	9	7	6	8	1

Evil Puzzle 8

5	2	9	6	3	1	4	8	7
6	7	1	9	4	8	3	5	2
8	4	3	5	2	7	6	1	9
9	8	4	3	1	2	7	6	5
2	6	5	8	7	9	1	4	3
1	3	7	4	5	6	2	9	8
7	5	2	1	9	4	8	3	6
4	9	6	2	8	3	5	7	1
3	1	8	7	6	5	9	2	4

Evil Puzzle 17

1	4	3	6	5	8	7	9	2
2	7	6	3	9	1	5	8	4
9	8	5	7	4	2	1	6	3
4	5	2	8	6	3	9	1	7
7	6	8	2	1	9	4	3	5
3	9	1	5	7	4	8	2	6
5	1	4	9	3	6	2	7	8
8	3	7	1	2	5	6	4	9
6	2	9	4	8	7	3	5	1

Evil Puzzle 5

3	1	2	4	6	9	8	5	7
9	8	5	3	7	1	4	6	2
4	6	7	5	2	8	9	3	1
7	5	6	9	4	3	1	2	8
2	3	9	8	1	7	6	4	5
1	4	8	2	5	6	7	9	3
5	2	1	7	9	4	3	8	6
6	9	3	1	8	2	5	7	4
8	7	4	6	3	5	2	1	9

Hard Puzzle 10

5	**3**	7	**4**	2	1	8	9	6
6	**9**	4	**8**	**3**	7	2	1	5
2	**1**	**8**	9	6	5	7	4	3
1	**6**	**9**	3	**8**	4	5	2	**7**
8	**2**	5	1	**7**	9	3	**6**	4
7	4	3	6	**5**	2	**1**	**8**	**9**
4	8	6	7	1	3	**9**	**5**	2
9	7	2	5	**4**	**8**	6	**3**	**1**
3	5	1	2	9	**6**	4	**7**	**8**

Expert Puzzle 10

6	**3**	**8**	2	1	9	**5**	7	4
5	**2**	4	6	8	**7**	**3**	**1**	9
1	9	7	**5**	4	3	2	8	**6**
8	**1**	3	7	5	**6**	9	4	2
2	5	9	**8**	3	**4**	7	6	**1**
7	4	6	**1**	9	2	8	**5**	3
9	8	2	4	7	**1**	6	3	5
4	**6**	**5**	**3**	2	8	1	**9**	7
3	7	**1**	9	6	5	**4**	**2**	**8**

Expert Puzzle 10 a

6	**3**	**8**	9	1	2	**5**	4	7
5	**2**	4	6	8	**7**	**3**	**1**	9
1	9	7	**5**	4	3	2	8	**6**
8	**1**	3	7	5	**6**	9	2	4
2	5	9	**8**	3	**4**	6	7	**1**
7	4	6	**1**	2	9	8	**5**	3
9	8	2	4	6	**1**	7	3	5
4	**6**	**5**	**3**	7	8	1	**9**	2
3	7	**1**	2	9	5	**4**	6	8

Hard Puzzle 10 a

2	**3**	7	**4**	6	1	8	9	5
6	**9**	4	**8**	**3**	5	7	1	2
5	**1**	**8**	9	2	7	6	4	3
1	**6**	**9**	3	**8**	4	5	2	**7**
8	**2**	5	1	**7**	9	3	**6**	4
7	4	3	6	**5**	2	**1**	**8**	**9**
4	8	2	7	1	3	**9**	**5**	6
9	7	6	5	**4**	**8**	2	**3**	**1**
3	5	1	2	9	**6**	4	**7**	**8**

Evil Puzzle 3 a

8	**9**	3	7	4	**1**	**2**	5	6
5	**2**	4	3	6	8	9	1	7
7	1	**6**	9	**5**	2	8	4	3
9	4	2	**6**	**8**	7	5	3	1
3	6	**5**	**1**	9	**4**	**7**	2	8
1	7	8	2	**3**	**5**	6	9	**4**
6	5	7	4	**2**	3	**1**	8	**9**
4	8	9	5	1	6	3	**7**	2
2	**3**	**1**	**8**	7	9	4	**6**	5

Evil Puzzle 5 a

1	3	**2**	**4**	**6**	9	8	5	7
9	**8**	**5**	1	7	3	4	2	6
4	6	7	5	**2**	8	**9**	3	1
7	2	8	3	4	6	1	9	5
3	1	**9**	**8**	5	**7**	**6**	4	2
5	**4**	6	9	1	2	7	8	**3**
2	5	**1**	7	**9**	4	3	6	8
6	9	3	2	8	1	**5**	**7**	**4**
8	7	4	6	**3**	**5**	**2**	1	9

Evil Puzzle 17 a

4	1	**3**	**6**	9	8	2	7	5
2	**7**	**6**	5	1	3	8	9	4
9	8	5	7	**4**	2	**1**	6	3
1	**5**	2	3	6	4	9	8	**7**
7	6	**8**	**2**	5	**9**	**4**	3	1
3	4	1	8	7	1	5	**2**	6
5	2	**4**	9	**3**	6	7	1	8
8	3	7	1	2	5	**6**	**4**	**9**
6	9	1	4	**8**	**7**	**3**	5	2

3a

6	9	4	1	7	8	2	5	**3**
8	2	3	**4**	**5**	**6**	**1**	7	9
1	**5**	7	2	3	9	4	6	8
7	**1**	**5**	**9**	6	2	**3**	8	4
3	4	8	7	**1**	5	6	9	2
2	**6**	9	3	**8**	4	**5**	1	**7**
5	8	**2**	6	4	7	9	3	1
9	3	6	8	2	**1**	**7**	**4**	5
4	**7**	1	5	**9**	3	8	2	6

3b

3	2	5	**4**	7	8	9	6	**1**
1	**4**	6	**2**	3	9	**5**	7	8
8	**7**	**9**	1	5	6	2	**3**	4
9	1	2	7	4	3	8	5	6
6	3	4	**5**	**8**	2	**1**	9	**7**
5	**8**	7	**6**	9	1	4	2	3
4	**5**	1	9	6	**7**	**3**	8	2
2	**6**	3	8	1	5	7	**4**	9
7	9	**8**	3	2	**4**	6	1	5

Bibliografia

[1] Cardoso, F.A.C.M., Fernandes, M.A.C., (2007). *FPGA e Fluxo de Projecto*. Faculdade de Engenharia Eléctrica de Computação UNICAMP.

[2] Gent, U. (2011). *ParaFPGA2011*. Consultado a 17 Out, 2011: http://parafpga.elis.ugent.be//.

[3] Skliarov, V., Skliarova, I., Sudnitson, A., (2011). *FPGA-based Systems in Information and Communication*. Proceedings of the 5th International Conference on Application of Information and Communication Technologies - AICT'2011, Baku, Azerbaijan. pp. 551-555.

[4] Wikipedia. (2011). *Lei de Moore*. Consultado a 22 Out, 2011: http://pt.wikipedia.org/wiki/Lei_de_Moore.

[5] Hasan, S., Yakovlev, A., Boussakta, S., (2010). *Performance efficient FPGA implementation of parallel 2-D MRI image filtering algorithms using Xilinx system generator* Sch. of Electr., Electron. & Comput. Eng., Univ. of Newcastle, UK

[6] Ahmed, S., (2009). *Parallel Computing Using FPGA*. pp. 1-2.

[7] 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.

[8] Bok, K.v.d., Taouil, M., Sourdis, I., Afratis, P. (2009). *The TU Delft Sudoku Solver on FPGA*. International Conference on Field-Programmable Technology 2009.

[9] Wolf, W. (2004). *FPGA-Based System Design*. Georgia Tech University: Prentice Hall. Chapter 3.

[10] Maxfield, C. (2004). *The Design Warrior's Guide to FPGA*. Xilinx: ELSEVIER SCIENCE & TECHNOLOGY.

[11] Biran, D., (2008). *The evolution of FPGAs, from prototype to production*. OpenSystems Publishing.

[12] Wikipedia. (2011). *Circuit logique programmable*. Consultado a 20 Out, 2011: http://fr.wikipedia.org/wiki/Circuit_logique_programmable.

[13] Wikipedia. *Field programmable gate array*. Consultado a 20 Out, 2011: http://en.wikipedia.org/wiki/Field-programmable_gate_array.

[14] Wikipedia. *FPGA Transistor Count*. Consultado a 1 Nov, 2011: http://en.wikipedia.org/wiki/Transistor_count#FPGA.

[15] Santarini, M. (2010) *Celebrating Customer Innovation.* XCell Journal.

[16] Maxfield, C. (2011) *Xilinx tips world's highest capacity FPGA*. EETimes.

[17] Jr, J.A., (2010). *State of art in FPGA technology*. São Carlos.

[18] Xilinx. *FPGA Comparison Table*. Consultado a 21 Out, 2011: http://www.xilinx.com/products/silicon-devices/fpga/index.htm.

[19] Bolsens, I., (2010). *Parallel programming in FPGA*. Xilinx.

[20] Almeida, M.J.d.S. *Métodos e ferramentas para reconfiguração de FPGAs remotamente*. Aveiro: *Departamento de Electrónica, Telecomunicações e Informática*, 2008. Dissertação de mestrado.

[21] Wikipedia. *Sudoku*. Consultado a 14 Set, 2011: http://pt.wikipedia.org/wiki/Sudoku.

[22] Loy, J. (2002). *Number Place*. Consultado a 15 Set, 2011: http://www.jimloy.com/puzz/place.htm.

[23] Wikipedia. (2010). *Solving Technique*. Consultado a 15 Set, 2011: http://sudopedia.org/wiki/Solving_Technique.

[24] Johnson, A. *Solving Sudoku*. Consultado a 17 Out, 2010: http://www.angusj.com/sudoku/hints.php.

[25] Stephens, P. (2007). *How to solve sudoku puzzles*. Consultado a 14 Nov, 2010: http://www.paulspages.co.uk/sudokuxp/howtosolve/.

[26] Sudopedia. (2010). *Hidden Single*. Consultado a 21 Out, 2010: http://oubk.com/Techniques/Hidden-Single-Technique.html.

[27] Oubk.com. *Hidden Single Technique*. Consultado a 21 Out, 2010: http://oubk.com/Techniques/Hidden-Single-Technique.html.

[28] Blog, S.D.s. (2005). *A wrong Sudoku*. Consultado a 16 Set, 2011: http://www.shanghaidaily.com/editor/article.asp?id=129.

[29] Technologies, A. *Sudoku Solutions: 9 by 9 Sudoku Solver*. Consultado a 17 Out, 2010: http://www.sudoku-solutions.com/.

[30] Asano, S., Maruyama, T., Yamaguchi, Y., (2009). *Performance comparison of FPGA, GPU and CPU in image processing*. Field Programmable Logic and Applications, 2009. FPL 2009.

[31] Jiang, J., Mirian, V., Chow, P., Tang, K.P. (2009). *Matrix Multiplication based on Scalable Macro-Pipelined FPGA Accelerator Architecture*. Reconfigurable Computing and FPGAs, 2009. ReConFig '09.

[32] FPT'09. (2009). *Sudoku Solver on FPGA Design Competition at FPT'09*. Consultado a 25 Out, 2011: http://fpt09.cse.unsw.edu.au/competition.html.

[33] González, C., Olivito, J., Resano, J., (2009). *An Initial Specific Processor for Sudoku Solving*. International Conference on Field-Programmable Technology 2009.

[34] Wikipedia. *Depth-First Search*. Consultado a 25 Out, 2011: http://en.wikipedia.org/wiki/Depth-first_search.

[35] Malakonakis, P., Smerdis, M., Sotiriades, E., Dollas, A. (2009). *An FPGA-Based Sudoku Solver based on Simulated Annealing Methods*. International Conference on Field-Programmable Technology 2009.

[36] Wikipedia. *Simulated annealing*. Consultado a 27 Out, 2011: http://en.wikipedia.org/wiki/Simulated_annealing#The_basic_iteration.

[37] ScienceZero. *Linear Feedback Shift Refisters - LFSR*. Consultado a 27 Out, 2011: http://sciencezero.4hv.org/science/lfsr.htm.

[38] Wikipedia. *Finite field*. Consultado a 27 Out, 2011: http://en.wikipedia.org/wiki/Finite_field.

[39] Dittrich, M., Preuber, T.B., Spallek, R.G., (2009). *Solving Sudokus through an Incidence Matrix on an FPGA*. International Conference on Field-Programmable Technology 2009.

[40] Xilinx. (2009). *Spartan-3E FPGA Family Datasheet*. Disponível em: http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

[41] Wikipedia. *Breadth-first search*. Consultado a 30 Out, 2011: http://en.wikipedia.org/wiki/Breadth-first_search.

[42] Greenspan, G. *Web Sudoku*. Consultado a 17 Out, 2010: http://www.websudoku.com/.

[43] Feenstra, M. *Collection of printable sudokus*. Consultado a 1 Set, 2011: http://www.sudoku.ws/.

[44] About.com. *Extra-Challenging Sudoku Puzzles*. Consultado a 3 Nov, 2011: http://puzzles.about.com/library/sudoku/blprsudokux04.htm.