UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA E INFORMÁTICA INDUSTRIAL

SIBILLA BATISTA DA LUZ FRANÇA

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE CHIPS DEDICADOS PARA UM NOVO DECODIFICADOR DE CÓDIGOS CORRETORES DE ERROS BASEADO EM CONJUNTOS DE INFORMAÇÃO

TESE

CURITIBA
2013

SIBILLA BATISTA DA LUZ FRANÇA

DESENVOLVIMENTO E IMPLEMENTAÇÃO DE CHIPS DEDICADOS PARA UM NOVO DECODIFICADOR DE CÓDIGOS CORRETORES DE ERROS BASEADO EM CONJUNTOS DE INFORMAÇÃO

Tese de doutorado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de "Doutor em Ciências" - Área de Concentração: Engenharia da Computação.

Orientador: Prof. Dr. Volnei A. Pedroni

CURITIBA

2013

Dados Internacionais de Catalogação na Publicação

F814 França, Sibilla Batista da Luz
Desenvolvimento e implementação de chips dedicados para um novo decodificador de códigos corretores de erros baseado em conjuntos de informação / Sibilla Batista da Luz França. — 2013.
168 p. : il. ; 30 cm

Orientador: Volnei Antonio Pedroni.
Tese (Doutorado) – Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. Curitiba, 2013.
Bibliografia: p. 138-140.

1. Códigos corretores de erros (Teoria da informação). 2. Decodificadores (Eletrônica). 3. Circuitos integrados – Integração em escala muito ampla. 4. Circuito integrado de aplicação específica. 5. Semicondutores complementares de óxido metálico. 6. Arranjos de lógica programável em campo. 7. Engenharia elétrica – Teses I. Pedroni, Volnei Antonio, orient. II. Universidade Tecnológica Federal do Paraná. Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial. III. Título.

CDD (22. ed.) 621.3

Biblioteca Central da UTFPR, Campus Curitiba

UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
Câmpus Curitiba

Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial

Título da Tese Nº. 90

"Desenvolvimento e Implementação de Chips Dedicados Para um Novo Decodificador de Códigos Corretores de Erros Baseado em Conjuntos de Informação".

por

Sibilla Batista da Luz França

Orientador: Prof. Dr. Volnei Antonio Pedroni

Esta tese foi apresentada como requisito parcial à obtenção do grau de DOUTOR EM CIÊNCIAS – Área de Concentração: Engenharia de Computação, pelo Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial – CPGEI – da Universidade Tecnológica Federal do Paraná – UTFPR, às 14h do dia 22 de agosto de 2013. O trabalho foi aprovado pela Banca Examinadora, composta pelos doutores:

Prof. Volnei Antonio Pedroni, Dr.
(Presidente – UTFPR)

Prof Carlos Galup Monto, Dr.
(UFSC)

Prof. Oscar da Costa Gouveia Filho, Dr.
(UFPR)

Prof. Walter Godoy Junior, Dr.
(UTFPR)

Prof. Carlos Raimundo Erig Lima, Dr.
(UTFPR)

Visto da coordenação:

Prof. Ricardo Lüders, Dr.
(Coordenador do CPGEI)

AGRADECIMENTOS

Agradeço primeiramente a Deus, meu amigo verdadeiro.

Ao meu orientador, Prof. Volnei Pedroni, pela orientação, apoio e incentivo que foram fundamentais no desenvolvimento deste trabalho.

Ao Ricardo Jasinski por sua grande participação neste projeto.

Aos Prof. Walter Godoy e Prof. Antônio Gortan pela parceria entre os grupos de Microeletrônica e de Comunicação de Dados.

Ao Prof. David Harris por sua co-orientação durante o doutorado sanduíche.

Ao meu esposo e aos meus pais que sempre me incentivaram.

Ao apoio financeiro da CAPES, pela concessão da bolsa de estudo.

Se Deus é por nós, quem será contra nós? (Romanos 8:31)

RESUMO

FRANÇA, Sibilla B. L. Desenvolvimento e Implementação de Chips Dedicados para um Novo Decodificador de Códigos Corretores de Erros Baseado em Conjuntos de Informação. 2013. 168 f. Tese (Doutorado em Informática Industrial) – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2013.

Códigos corretores de erros estão presentes em quase todos os sistemas modernos de comunicação e armazenamento de dados. Erros durante essas operações são praticamente inevitáveis devido a ruído e interferências nos meios de comunicação e degradação dos meios de armazenamento. Quando um sistema exige alto desempenho, os correspondentes algoritmos (codificador e decodificador) são implementados em hardware. O projeto de pesquisa apresentado nesta tese, um chip dedicado para uma nova família de decodificadores baseados em conjuntos de informação, é parte de um amplo projeto que visa obter um decodificador com desempenho semelhante à decodificação de máxima verossimilhança (MLD), porém com hardware muito mais simples, demonstrando assim que o uso dessa técnica (decodificação por conjuntos de informação), até então proibitiva devido à complexidade do hardware, poderia tornar-se viável. Visando simplificar o hardware, o primeiro passo foi modificar o algoritmo original de Dorsch para reduzir o número de ciclos de clock necessários para decodificar uma mensagem. As principais modificações realizadas foram na redução de Gauss-Jordan e no número de palavras-código candidatas, consideravelmente reduzidas em relação ao algoritmo original de Dorsch. Este algoritmo modificado foi primeiramente implementado utilizando linguagem de descrição de hardware e avaliado em diferentes famílias de FPGAs, onde demonstrou-se o mesmo ser viável, mesmo para grandes códigos. O algoritmo foi implementado posteriormente em um chip dedicado (ASIC), utilizando tecnologia CMOS, a fim de completar a demonstração da viabilidade de sua implementação e uso efetivo.

Palavras-chave: Códigos Corretores de Erros, Decodificador, Conjunto de Informação, VLSI, ASIC, CMOS.

ABSTRACT

FRANÇA, Sibilla B.L. Desenvolvimento e Implementação de Chips Dedicados para um Novo Decodificador de Códigos Corretores de Erros Baseado em Conjuntos de Informação. 2013. 168 f. Tese (Doutorado em Informática Industrial) – Programa de Pós-Graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná. Curitiba, 2013.

Error-correcting codes are present in almost all modern data communications and data storage systems. Errors during these operations are practically inevitable because of noise and interference in communication channels and degradation of storage media. When top-performance is required, the corresponding algorithms (encoder and decoder) are implemented in hardware. The research project presented in this dissertation, a dedicated chip for a new family of decoders based on information sets, is part of a broad project targeting the development of a new decoder capable of achieving near maximum likelihood decoding (MLD) performance, however with a much simpler hardware, thus demonstrating that the use of this technique (decoding based on information sets), previously prohibitive due to the complexity of the hardware, could now be feasible. Aiming to simplify the hardware, the first step was to modify the original Dorsch algorithm to reduce the number of clock cycles needed to decode a message. The main modifications performed were in the Gauss Jordan elimination procedure and in the number of candidate codewords, which was highly reduced with respect to original Dorsch algorithm. This modified algorithm was first implemented using a hardware description language and evaluated in different FPGA families, where the viability was demonstrated. The algorithm was later implemented in a dedicated chip (ASIC) using CMOS technology in order to complete the demonstration of the feasibility of their implementation, and effective use.

Keywords: Error-correcting Codes, Decoder, Information Set, VLSI, ASIC, CMOS.

LISTA DE FIGURAS

Figura 2.1 - Uso de código corretor de erros na transmissão de dados. 19
Figura 2.2 - Modelo de um sistema de comunicação digital. 20
Figura 2.3 - Codificação sistemática de um código de bloco. 21
Figura 2.4 - Codificador RS (255,223). 28
Figura 2.5 - Gráfico de Tanner para um código LDPC (6,2). 29
Figura 2.6 - Decodificação de um código LDPC. 30
Figura 2.7 - Codificador convolucional (2, 1, 2). 31
Figura 2.8 - Diagrama de estados de um codificador convolucional (2, 1, 2). 32
Figura 2.9 - Diagrama de treliça para o codificador convolucional (2, 1, 2). 33
Figura 2.10 - Codificador RSC (2, 1, 2). 33
Figura 2.11 - Codificador turbo. 34
Figura 3.1 - Procedimento de Decodificação de Viterbi. 37
Figura 3.2 - Decisão abrupta x Decisão suave. 38
Figura 4.1 - Algoritmo de decodificação proposto em (GORTAN et al., 2010a). 47
Figura 4.2 - Diagrama de blocos do decodificador utilizando-se k+1 candidatas (GORTAN et al., 2010a). 49
Figura 4.3 - Comparação de desempenho (GORTAN et al., 2010a). 50
Figura 4.4 - Diagrama de blocos do decodificador utilizando m candidatas (GORTAN et al., 2010b). 51
Figura 4.5 - Degradação do desempenho (em dB) em relação ao MLD, em função do número de palavras-código candidatas (GORTAN et al., 2010b). 52
Figura 4.6 - Diagrama de blocos do decodificador com critério de parada (GORTAN, 2011). 56
Figura 4.7 - Número de candidatas que precisam ser inspecionadas antes da atuação do critério de parada (em % do total do número de palavras candidatas) (GORTAN, 2011). 57
Figura 5.1 - Diagrama geral da rede de handshake. 59
Figura 5.2 - Diagrama de blocos do decodificador utilizando-se k+1 candidatas (GORTAN et al., 2010a). 60
Figura 5.3 - Utilização da unidade de *handshake* em um estágio de processamento. 62
Figura 5.4 - Máquina de estados utilizada na unidade de handshake. 63
Figura 5.5 - Diagrama de circuito da unidade de *handshake*. 64
Figura 5.6 - Diagrama de blocos da demodulação/ordenação. 65
Figura 5.7 - Sub-blocos que compõem bloco de demodulação/ordenação de entrada. 65
Figura 5.8 - Implementação em hardware do ordenador de inserção linear. 66
Figura 5.9 - Sub-blocos que compõem o bloco de redução de Gauss-Jordan modificada. 67
Figura 5.10 - Sub-blocos que compõem o bloco de eliminação de Gauss Jordan modificada. 68
Figura 5.11 - Diagrama do sub-bloco RowExchangeNeeded. 69
Figura 5.12 - Diagrama do sub-bloco SelectedRow. 69
Figura 5.13 - Diagrama do sub-bloco rowExchangeImpossible. 70
Figura 5.14 - Diagrama do sub-bloco de troca de linhas. 70
Figura 5.15 - Diagrama do sub-bloco de eliminação de coluna. 71
Figura 5.16 - Diagrama de blocos da matriz G_{r0}. 71
Figura 5.17 - Diagrama de blocos do gerador de mensagens candidatas. 72
Figura 5.18 - Sub-blocos que compõem bloco de geração das palavras candidatas. 72
Figura 5.19 - Diagrama do circuito gerador de mensagens candidatas. 73
Figura 5.20 - Diagrama de blocos do gerador de palavras-código candidatas. 73

Figura 5.21 - Sub-blocos que compõem bloco de geração das palavras-código candidatas. ... 74
Figura 5.22 - Diagrama de circuito do gerador de palavras-código candidatas. 74
Figura 5.23 - Diagrama de blocos do seletor de palavra-código vencedora............................ 75
Figura 5.24 - Sub-blocos que compõem o bloco de seleção da palavra-código vencedora. 76
Figura 5.25 - Diagrama detalhado do seletor da palavra-código vencedora. 76
Figura 6.1 - Entidade *top level* do decodificador. 79
Figura 6.2 - Bloco 1: Modulador/Ordenador. 79
Figura 6.3 - Bloco 2: Redução de Gauss-Jordan modificada. 80
Figura 6.4 - Bloco 3: Geração das mensagens candidatas. 81
Figura 6.5 - Bloco 4: Geração das palavras-código candidatas. 82
Figura 6.6 - Bloco 5: Seleção da candidata vencedora. 82
Figura 7.1 - Configurações escolhidas para as células-padrão e seus respectivos layouts. 88
Figura 7.2 - Layout da unidade de *handshake*. 90
Figura 7.3 - Layout original do bloco de demodulação/ordenação de entrada, composto pelos sub-blocos: latchInput (armazena a entrada analógica); currentAnalogBit (seleciona um símbolo da palavra analógica); currentTag (atribui uma etiqueta para o símbolo selecionado); reliabilityFunction (calcula a confiabilidade do símbolo); tagSorter (ordena os símbolos de acordo com a confiabilidade); e handshake. 92
Figura 7.4 - Layout do bloco de demodulação/ordenação de entrada com os circuitos adicionais de entrada/saída, utilizados devido ao número limitado de pinos disponíveis no chip. 92
Figura 7.5 – Layout final do chip contendo os blocos de demodulação/ordenação de entrada e de geração de mensagens candidatas, fabricado com o objetivo de testar os blocos individualmente. 93
Figura 7.6 - Simulação do esquemático do demodulador/ordenador no Quartus II. 94
Figura 7.7 - Simulação SPICE do demodulador/ordenador sem capacitância parasitas e sem pads. 94
Figura 7.8 - Simulação SPICE do demodulador/ordenador, incluindo capacitância parasitas e pads. 95
Figura 7.9 - Bloco 1 decisão abrupta (osciloscópio). 95
Figura 7.10 - Layout original do bloco de eliminação de Gauss Jordan modificada, composto pelos sub-blocos: s (vetor de confiabilidade); G (matriz geradora); G_r (matriz geradora reduzida ou matriz de trabalho); G_{r0} (matriz geradora reduzida, com todas as colunas não selecionadas substituídas por vetores nulos); EM (matriz resultante após a troca de linhas); EX (matriz resultante após a operação XOR); column (seleção de uma coluna da matriz G, segundo o vetor de confiabilidade), iterationCounter (contador de iterações); selectedRow (linha da matriz G usada na troca de linhas); e handshake 97
Figura 7.11 - Layout do bloco de eliminação de Gauss Jordan modificada com circuitos adicionais de entrada/saída, utilizados devido ao número limitado de pinos disponíveis no chip. 97
Figura 7.12 - Layout final do chip contendo o bloco de eliminação de Gauss Jordan modificada, fabricado com o objetivo de testar o bloco individualmente. 98
Figura 7.13 - Simulação do bloco de redução de Gauss-Jordan modificada. 99
Figura 7.14 - Resultados da simulação do código SPICE para a redução de Gauss-Jordan modificada. 99
Figura 7.15 - Simulação SPICE do bloco de redução de Gauss-Jordan modificada sem capacitâncias parasitas e sem pads. 100
Figura 7.16 - Simulação SPICE do bloco de redução de Gauss-Jordan modificada com capacitâncias parasitas e com pads. 101

Figura 7.17 - Frequência máxima de operação do bloco 2 (osciloscópio). 101
Figura 7.18 - Layout original do bloco de geração de mensagens candidatas, composto pelos sub-blocos: latchR (armazena o sequência demodulada por decisão abrupta); latchG_{r0} (armazena a matriz geradora reduzida, com todas as colunas não selecionadas substituídas por vetores nulos); r × G_{r0} (realiza a multiplicação entre o vetor r e a matriz G_{r0}); xor gates (realiza a operação xor entre a mensagem candidata original, para geração das demais mensagens candidatas); e handshake. 103
Figura 7.19 - Layout do bloco de geração de mensagens candidatas com circuito adicional de entrada/saída. 104
Figura 7.20 - Simulação do esquemático do gerador de mensagens candidatas no Quartus II. 104
Figura 7.21 - Simulação SPICE do bloco de geração de mensagens candidatas sem capacitâncias parasitas e sem pads. 105
Figura 7.22 - Simulação SPICE do bloco de geração de mensagens candidatas com capacitâncias parasitas e com pads. 105
Figura 7.23 - Geração das mensagens candidatas (osciloscópio). 106
Figura 7.24 - Layout original do bloco de geração de palavras-código candidatas, composto pelos sub-blocos: latchUc (armazena a matriz de mensagens candidatas); latchG_r (armazena a matriz geradora reduzida); $U_c \times G_r$ (realiza a multiplicação entre a matriz de mensagens candidatas e a matriz Gr); e handshake. 107
Figura 7.25 - Layout do bloco de geração de palavras-código candidatas com circuitos adicionais de entrada/saída. 108
Figura 7.26 - Layout final do chip contendo o bloco de geração de palavras-código, fabricado com o objetivo de testar o bloco individualmente. 109
Figura 7.27 - Simulação do esquemático do gerador de palavras-código candidatas no Quartus II. 110
Figura 7.28 - Simulação SPICE do bloco de geração de palavras-código candidatas sem capacitâncias parasitas e sem pads. 110
Figura 7.29 - Simulação SPICE do bloco de geração de palavras-código candidatas com capacitâncias parasitas e com pads. 111
Figura 7.30 - Geração de palavras-código candidatas (osciloscópio). 111
Figura 7.31 - Layout original do bloco de geração de palavras-código candidatas, composto pelos sub-blocos: latchX (armazena a palavra analógica); latchC_c (armazena a matriz de palavras-código candidatas); currentCandidateWord (seleciona uma palavra-código candidata); softDistanceCalculation (calcula a distancia suave entre a palavra-código candidata e a palavra analógica); bestCandidateWord (armazena a palavra-código vencedora; e handshake. 113
Figura 7.32 - Layout do bloco de seleção de palavra-código vencedora com os circuitos adicionais de entrada/saída, ... 113
Figura 7.34 - Layout final do chip contendo o bloco de seleção de palavra-código, fabricado com o objetivo de testar o bloco individualmente. 114
Figura 7.35 - Simulação do esquemático do seletor da palavra-código vencedora. 115
Figura 7.36 - Simulação SPICE do seletor da palavra-código vencedora sem capacitâncias parasitas e sem pads. 115
Figura 7.37 - Simulação SPICE do seletor da palavra-código vencedora com capacitâncias parasitas e com pads. 116
Figura 7.38 - Seleção da palavra-código vencedora (osciloscópio). 116
Figura 7.39 - Layout final do chip contendo o decodificador completo, isto é, os cinco blocos interconectados. 117
Figura 7.40-Simulação do decodificador na ferramenta Quartus II. 119

Figura 7.41 - Saída do decodificador: decisão abrupta (sem inclusão de capacitâncias parasitas e pads)....120
Figura 7.42 - Saída do decodificador: decisão suave (sem inclusão de capacitâncias parasitas e pads). 121
Figura 7.43 - Sinais de controle entre os blocos do decodificador....121
Figura 7.44 - Saída do decodificador: decisão suave (osciloscópio)....122
Figura 8.1 - Sequência de operações realizadas na ferramenta RTL Compiler. 125
Figura 8.2 - Script utilizado no RTL Compiler para síntese do decodificador na tecnologia de 180 nm....126
Figura 8.3 - Esquemático completo do decodificador gerado no RTL Compiler. 126
Figura 8.4 - Esquemático do circuito de handshake gerado no RTL Compiler. 126
Figura 8.5 - Sequência de operações realizadas na ferramenta Encounter....127
Figura 8.6 - Design do decodificador gerado pela ferramenta Encounter (da Cadence)....128
Figura 8.7 - Layout final do decodificador gerado pela ferramenta Virtuoso....129
Figura 8.8 - Simulação do decodificador após a síntese no RTL Compile. 129

LISTA DE TABELAS

Tabela 2.1 - Entrada/saída e estados possíveis de um codificador convolucional (2, 1, 2)......32
Tabela 3.1 - Sequências possíveis para um código C (5,2). 36
Tabela 4.1 - Confiabilidade e decodificação abrupta da palavra transmitida. 43
Tabela 4.2 - Palavras candidatas obtidas com o algoritmo de Dorsch. 45
Tabela 4.3 - Distância suave entre as palavras candidatas. 48
Tabela 4.4 - Entrada/saída e estados possíveis de um codificador convolucional (2, 1, 2)......52
Tabela 4.5 - Comparação direta entre os critérios de parada BGW e HO-BGW (GORTAN, 2011). 55
Tabela 6.1 - Resultados da Implementação em FPGA (Stratix III)......83
Tabela 7.1 - Resumo dos chips fabricados no projeto descrito nesta tese. 85
Tabela 7.2 - Seleção dos dados de entrada para o bloco 1. 91
Tabela 7.3 - Seleção dos dados de saída do bloco 1. 91
Tabela 7.4 - Seleção dos dados de entrada do bloco 2. 96
Tabela 7.5 - Seleção dos dados de entrada para o bloco 3. 102
Tabela 7.6 - Seleção dos dados de saída do bloco 3. 103
Tabela 7.7- Seleção dos dados de entrada para o bloco 4. 106
Tabela 7.8- Seleção dos dados de saída do bloco 4. 107
Tabela 7.9 - Seleção dos dados de entrada para o bloco 5. 112
Tabela 7.10 - Definição dos pinos. 118
Tabela 7.11 - Frequência máxima de operação (MHz). 120
Tabela 8.1 - Resumo pós-síntese do projeto do decodificador na tecnologia IBM 7RF. 130
Tabela 8.2 - Resumo pós-síntese do projeto do decodificador na tecnologia IBM 8RF. 130

LISTA DE SIGLAS

ASIC	Application-Specific Integrated Circuit
AWGN	Additive White Noise
CI	Circuito Integrado
CMOS	Complementary Metal Oxide Semicondutor
CSVL	Cascade Voltage Switch Logic
DFF	Flip-Flop D
DL	Latch D
DRC	Design Rule Check
DTG	Dinamic Transmission Gate
ECL	Emitter Coupled Logic
EM	Exchange Row Matrix
FA	Full Adder
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GMD	Generalized Minimum Distance
IS	Information Set
LD	Linearmente Dependente
LDPC	Low-Density parity Check
LI	Linearmente Independente
MLD	Maximum Likelihood Decoder
MOS	Metal Oxide Semicondutor
MOSFET	Metal Oxide Semiconductor Field Effect Transistor
RAM	Random Access Memory
RSC	Recursive Systematic Convolutional
RSIS	Static ratio Intensive
S-CSVL	Static Cascade Voltage Switch Logic
SNR	Sinal to Noise Ratio
SR	Set-Reset
SSTC	Static Single Clocked
STG	Static Transmission Gate
TFF	Flip-Flop T
TG	Transmission Gate
VLSI	Very Large Scale Integration
WM	Working Matrix
XM	Xored Matrix

SUMÁRIO

1 Introdução 16
1.1 Motivações 16
1.2 Objetivos 17
1.3 Contribuições 17
1.4 Estrutura da Tese 17
2 Códigos Corretores de Erros 19
2.1 Introdução 19
2.2 Códigos de Blocos 21
3.2.1 Códigos de Hamming 26
3.2.2 Códigos Reed-Solomon (RS) 26
3.2.3 Códigos Low Parity Density Check (LDPC) 28
2.3 Códigos Convolucionais 30
3.3.1 Códigos Convolucionais Convencionais 31
3.3.2 Códigos Turbo 33
3 Técnicas de Decodificação 35
3.1 Introdução 35
3.2 Decodificação de Máxima Verossimilhança (MLD) 35
3.3 Decodificação de Viterbi 36
3.4 Decodificação com Decisão Abrupta x Decodificação com Decisão Suave 38
3.5 Decodificação por Conjuntos de Informação 39
4 Algoritmos de Decodificação Otimizados para Implementação em Hardware e Arquitetura 41
4.1 Introdução 41
4.2 Algoritmo de Decodificação por Conjuntos de Informação 41
4.3 Algoritmo de Decodificação de Dorsch 42
4.4 Algoritmo de Decodificação Otimizado para Implementação em Hardware 45
4.4.1 Algoritmo Otimizado com k+1 Candidatas 46
4.4.2 Algoritmo Otimizado com m Candidatas 50
4.4.3 Algoritmo Otimizado com Critério de Parada 53
5 Circuitos Propostos Para o Hardware 58
5.1 Introdução 58
5.2 Descrição Geral da Arquitetura 58
5.3 Detalhamento do Circuito de Handshake 62
5.4 Detalhamento do Bloco 1: Demodulador/Ordenador de Entrada 64
5.5 Detalhamento do Bloco 2: Redução de Gauss-Jordan Modificada 67

5.6 Detalhamento do Bloco 3: Gerador de Mensagens Candidatas 71
5.7 Detalhamento do Bloco 4: Gerador de Palavras-Código Candidatas 73
5.8 Detalhamento do Bloco 5: Seletor da Palavra-Código Vencedora 75
6 Implementação do Hardware em FPGA 77
6.1 Introdução 77
6.2 Descrição do Procedimento de Projeto em VHDL/FPGA 77
6.3 Principais Resultados Obtidos em FPGA 83
7 Implementação Manual do Hardware em ASIC e Resultados 84
7.1 Introdução 84
7.2 Construção da Biblioteca de Células-Padrão 84
7.3 Circuito de Handshake 89
7.4 Bloco 1: Demodulador/Ordenador de Entrada 90
8.4.1 Projeto 90
8.4.2 Resultados 93
7.5 Bloco 2: Redução de Gauss-Jordan Modificada 96
8.5.1 Projeto 96
8.5.2 Resultados 98
7.6 Bloco 3: Gerador de Mensagens Candidatas 102
8.6.1 Projeto 102
8.6.2 Resultados 104
7.7 Bloco 4: Gerador de Palavras-Código Candidatas 106
8.7.1 Projeto 106
8.7.2 Resultados 109
7.8 Bloco 5: Seletor da Palavra-Código Vencedora 112
8.8.1 Projeto 112
8.8.2 Resultados 114
7.9 Colocação e Interligação dos Blocos no Frame do Chip 116
8.9.1 Projeto 116
8.9.2 Resultados 119
8 Implementação do Hardware em ASIC Utilizando VHDL e Resultados 123
8.1 Introdução 123
8.2 Design do Chip: RTL-to-GDSII 124
9.2.1 Síntese 124
9.2.2 Place and Route 127
9.2.3 GDSII 128
8.3 RESULTADOS 129
9 Conclusões e Trabalhos Futuros 131

9.1 Conclusões ... 131
9.2 Trabalhos Futuros ... 132
Referências ... 138
ANEXO A ... 141

CAPÍTULO 1

INTRODUÇÃO

1.1 MOTIVAÇÕES

Códigos corretores de erros estão presentes em praticamente todos os sistemas modernos de comunicação e armazenamento de dados. As taxas de transmissão têm aumentado continuamente, chegando ao ponto que não é mais possível implementar todos os algoritmos necessários puramente em software. Devido à maior velocidade de processamento, os algoritmos de codificação e decodificação são comumente implementados em hardware quando o sistema exige elevado desempenho.

A utilização de conjuntos de informação (IS) para correção de erros usando códigos de blocos já é conhecida desde 1962 (PRANGE, 1962). Porém, apenas nos últimos anos a implementação em hardware desse tipo de decodificador tornou-se atraente, devido ao tamanho dos circuitos envolvidos e à elevada quantidade de processamento.

Durante um período de aproximadamente quatro anos, o grupo de pesquisa do qual este projeto faz parte desenvolveu e aprimorou um novo algoritmo de decodificação baseado em conjuntos de informação. Tal decodificador tem desempenho muito próximo ao MLD (Decodificador de Máxima Verossimilhança), porém requer apenas um número reduzido de comparações. Embora seja um algoritmo de decodificação suave (o que aumenta a qualidade do código, porém aumenta também a complexidade dos cálculos), a grande redução do hardware propiciada pelo algoritmo proposto indicava a viabilidade de sua implementação em hardware.

A principal motivação para este trabalho é verificar se a implementação totalmente em hardware (em um circuito integrado dedicado) do novo decodificador corretor de erros baseado em IS é viável, mesmo para códigos grandes, analisando os parâmetros relacionados ao tamanho e desempenho do mesmo.

1.2 OBJETIVOS

O objetivo desta pesquisa é estudar a viabilidade de implementar completamente em hardware decodificadores para códigos corretores de erros baseados em conjuntos de informação, bem como efetuar tais implementações, tanto em FPGAs quanto em chips dedicados. Para tanto, um novo algoritmo foi desenvolvido e aperfeiçoado pelo grupo de pesquisa do qual este projeto faz parte.

Os estudos foram conduzidos em três etapas: (*i*) Estudo, desenvolvimento e implementação em FPGA (utilizando a linguagem para síntese de hardware VHDL); (*ii*) Estudo, desenvolvimento e implementação manual, a nível de transistor, em ASIC; (*iii*) Estudo, desenvolvimento e implementação em ASIC a partir do código VHDL. Os resultados correspondentes possibilitaram obter várias conclusões sobre a viabilidade do novo algoritmo.

1.3 CONTRIBUIÇÕES

Este projeto resultou em várias contribuições relevantes. A primeira foi a comprovação da viabilidade de implementar um decodificador baseado em conjuntos de informação totalmente em hardware, utilizando o algoritmo desenvolvido pelo grupo. A segunda foi a criação e a fabricação de um chip dedicado (ASIC) deste decodificador. A terceira foi o conhecimento adquirido durante o desenvolvimento do projeto, onde cada etapa do desenvolvimento foi detalhadamente estudada. Além disso, foi fundamental o aprendizado do novo conjunto de ferramentas de design, seguindo um fluxo de projeto real, como realizado nas indústrias de chips. Finalmente, uma importante contribuição foi o sucesso da cooperação entre os grupos de Microeletrônica da UTFPR, de Comunicação de Dados da UTFPR e do grupo de Microeletrônica do Harvey Mudd College, dirigido pelo Prof. David Harris.

1.4 ESTRUTURA DA TESE

Este documento está dividido em nove capítulos. O capítulo 2 descreve os principais conceitos relacionados a códigos corretores de erros. O capítulo 3 apresenta técnicas já existentes para decodificação de códigos corretores de erros. O capítulo 4 descreve alguns algoritmos existentes para a decodificação de códigos corretores de erros por conjunto de

informação, além de apresentar um novo algoritmo de decodificação otimizado para a implementação em hardware. O capítulo 5 discute os circuitos propostos para o hardware na implementação do decodificador com k+1 palavras-código candidatas. O capítulo 6 descreve a primeira implementação para o decodificador, em FPGA utilizando a linguagem VHDL. O capítulo 7 apresenta a implementação manual do hardware dedicado em ASIC. O capítulo 8 descreve a implementação automatizada do hardware dedicado em ASIC utilizando VHDL. Finalmente, o capítulo 9 apresenta as principais conclusões e trabalhos futuros. O anexo A apresenta uma breve revisão de circuitos digitais integrados, úteis no desenvolvimento do trabalho.

CAPÍTULO 2

CÓDIGOS CORRETORES DE ERROS

2.1 INTRODUÇÃO

Códigos corretores de erros representam uma técnica introduzida em sistemas digitais para aumentar a confiabilidade nas operações de transmissão e armazenamento de dados. Durante essas operações, podem ocorrer erros, devido principalmente a ruído ou interferências no canal de comunicação ou imperfeições na mídia de armazenamento. Como tais códigos são capazes de detectar e corrigir erros (de acordo com sua capacidade de detecção/correção), eles são amplamente utilizados em sistemas de comunicação via satélite, telefonia digital, redes locais de computadores, discos a laser, controle e automação.

A capacidade de correção de um código deve-se a um processo de codificação no qual são adicionados, aos bits originais de uma mensagem, bits de redundância. Segundo Shannon (Shannon, 1948), se a capacidade máxima do canal (bits/segundo) for respeitada, os erros ocorridos em uma transmissão de dados podem ser reduzidos a qualquer taxa desejada através de codificação e decodificação adequadas (CASTIÑEIRA; FARRELL, 2006) (SWEENEY, 2002).

O uso de um código corretor de erros é ilustrado através de um exemplo na Figura 2.1. Neste exemplo, a mensagem **u**, de 6 bits, é codificada adicionando 3 bits de redundância, resultando em uma palavra-código de 9 bits. Conforme mostrado na Figura, durante a transmissão, a mensagem teve seu terceiro bit corrompido (devido a ruído), resultando, no receptor, a mensagem **c***. Contudo, graças à redundância, a mesma pôde ser reconstruída adequadamente pelo decodificador.

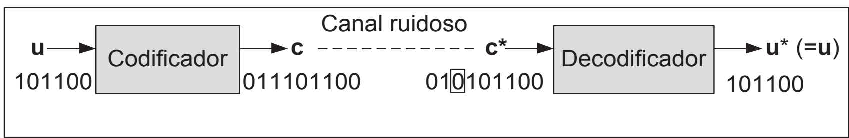


Figura 2.1 - Uso de código corretor de erros na transmissão de dados.

Um diagrama mais detalhado é mostrado na Figura 2.2 (MORELOS-ZARAGOZA, 2002) (CLARK; CAIN, 1981). A *fonte* emite informações que podem ser analógicas ou digitais (se a fonte for analógica, o codificador de fonte deve fazer a conversão analógico-digital). O *codificador de fonte* transforma a sequência recebida em uma sequência binária, chamada *sequência de informação* (**u**). O *codificador de canal* adiciona redundância à sequência **u**, resultando uma sequência codificada, **c**. Na maioria das vezes, a sequência codificada é uma sequência binária, embora algumas aplicações utilizem códigos que não são binários. Os códigos binários são mais fáceis de serem implementados, por outro lado, para determinadas aplicações, como por exemplo ADSL, são usados códigos não binários. O *modulador* transforma todos os símbolos recebidos do codificador de canal em um sinal (com duração de T segundos) que será transmitido através do *canal* de comunicação (suscetível a ruído). Após a transmissão, o *demodulador* processa o sinal recebido e produz uma sequência de saída discreta (*hard decision*) ou analógica (*soft decision*), a qual é utilizada pelo *decodificador de canal* para fazer uma estimativa da palavra originalmente transmitida. A sequência assim decodificada é entregue ao *decodificador de fonte*, que a converte de volta à forma original (se a fonte original era analógica, o decodificador de fonte deve fazer a conversão digital-analógica antes de entregar a mensagem ao *destino* final).

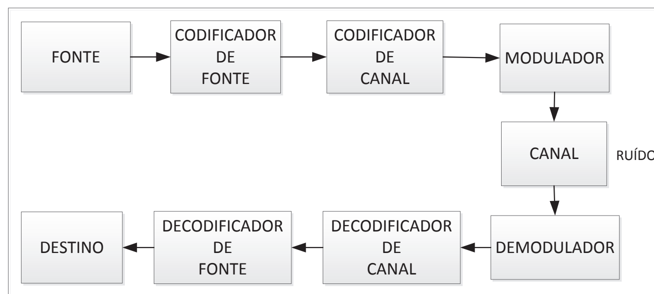


Figura 2.2 - Modelo de um sistema de comunicação digital.

Os códigos corretores de erros podem ser divididos de uma maneira genérica em dois grupos: *códigos de blocos* e *códigos convolucionais*. Cada um deles será examinado nas próximas seções.

Os códigos de blocos foram os primeiros a se destacar. Iniciaram com códigos de Golay (GOLAY, 1949), que tinha capacidade de corrigir alguns erros e o código de Hamming (HAMMING, 1950), capaz de corrigir apenas um erro. Em seguida, surgiram os códigos BCH (HOCQUENGHEM, 1959) (BOSE; RAY-CHAUDHURI, 1960), capazes de corrigir múltiplos erros. Outro código criado na mesma época foi o de Reed-Solomon, para canais não

binários. Em 1962, os códigos LDPC foram introduzidos por Gallager (GALLAGER, 1962), entretanto só se popularizaram em 1996, através de MacKay e Neal (MACKAY; NEAL, 1997), sendo atualmente, os códigos corretores de erros com desempenho mais próximo do limite de Shannon.

Os códigos convolucionais se destacaram um pouco mais tarde do que os códigos de blocos, a partir do algoritmo desenvolvido por Viterbi (VITERBI, 1967), que reduziu enormemente a complexidade de decodificação dos mesmos. Outro código convolucional, introduzido em 1993, foi o código turbo (BERROU; GLAVIEUX; THITIMAJSHIMA, 1993), capaz de apresentar desempenho superior a todos os códigos existentes até aquele momento, inferior atualmente ao código de bloco LDPC.

Os códigos de blocos e convolucionais apresentam diferenças importantes. Por exemplo, no primeiro os dados são subdivididos em blocos de tamanho fixo, ao passo que no segundo o fluxo de dados é contínuo. Outra diferença é o uso de memória, pois os códigos de blocos dependem exclusivamente dos símbolos de informação atuais, portanto sem necessidade de memória, ao passo que os convolucionais dependem não apenas da palavra atual, mas também de dados obtidos a partir de palavras anteriores, exigindo assim o uso de memória.

2.2 CÓDIGOS DE BLOCOS

Nos códigos de blocos, a mensagem é agrupada em blocos de tamanho fixo de k bits. A notação mais utilizada para representar esses códigos é C(n, k), onde n é o número de bits resultante na palavra-código (Figura 2.3 (MORELOS-ZARAGOZA, 2002)). O codificador recebe cada bloco de k bits separadamente (ou seja, um bloco não depende do outro) e o converte em um bloco maior, com $n>k$ bits, formando a palavra-código a ser transmitida. Os bits adicionados à mensagem original, calculados pela própria informação de k bits, são chamados *bits de redundância*.

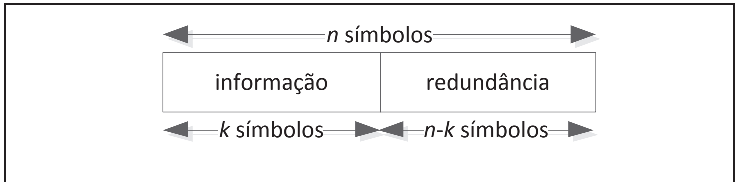


Figura 2.3 - Codificação sistemática de um código de bloco.

A palavra-código pode estar organizada de dois modos diferentes: *sistemático* ou *não sistemático*. No modo sistemático, representado na Figura 2.3, os bits de informação estão separados dos bits de redundância, enquanto no não sistemático os bits de informação e redundância estão misturados.

Um parâmetro muito utilizado para descrever um código de bloco é a sua *taxa*, dada pela Equação 2.1, que indica o nível de redundância do mesmo. Se R for baixo, significa que o código tem um alto grau de redundância, sendo portanto capaz de corrigir um número maior de erros (para um dado código).

$$R = k/n \quad \textbf{(2.1)}$$

Um código de bloco é linear quando qualquer combinação linear de suas palavras-código resulta também em uma palavra pertencente ao código, inclusive a combinação linear de qualquer palavra-código com ela mesma. Assim, todo código linear possui a palavra-código todo zero. Portanto, quando um código é linear, a multiplicação de uma palavra-código por um escalar válido ou a adição de duas palavras-código produz outra palavra pertencente ao código. Sendo assim, a codificação pode ser realizada através de multiplicação matricial, o que, em termos de eletrônica digital, significa que o codificador pode ser construído usado apenas portas XOR, AND e flip-flops (SWEENEY, 2002) (MORELOS-ZARAGOZA, 2002).

Outra definição importante de um código é o seu peso de Hamming. Para uma palavra de código **c**, o peso de Hamming é dado pelo número de posições não nulas na mesma; por exemplo, o peso de 1101000 é 3. O peso mínimo (representado por w) das palavras que compõe um código é o peso de Hamming daquele código (CASTIÑEIRA; FARRELL, 2006).

A distância de Hamming entre duas palavras de mesmo comprimento é o número de posições onde as duas diferem. Por exemplo, a distância de Hamming entre as palavras 1101000 e 0110100 é 4. A distância mínima de Hamming (d_{min}) de um código é a menor distância entre duas palavras pertencentes àquele código. Para um código linear a distância mínima de Hamming é igual ao peso mínimo de Hamming (LIN; COSTELLO, 1983).

A capacidade de correção de um código deve-se ao processo de codificação no qual são adicionados bits de redundância. Esta capacidade está relacionada à distância mínima de Hamming do código. A capacidade de correção (t_c) de um código é dada pela igualdade na

Equação 2.2, e a capacidade de detecção de erros (t_d) (detecta, porém não corrige) é dada pela Equação 2.3.

$$t_c = \frac{\lfloor d_{Hmin} - 1 \rfloor}{2} \quad (2.2)$$

$$t_d = \lfloor d_{Hmin} - 1 \rfloor \quad (2.3)$$

Outro aspecto importante de um código de bloco é a sua matriz geradora. Considere um código de bloco linear binário C(*n, k*). Sendo C um subespaço vetorial *k*-dimensional, C pode ser escrito como $\{\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_{k-1}\}$, tal que qualquer palavra-código **c** ∊ C possa ser representada por uma combinação linear do tipo:

$$\mathbf{c} = u_0\mathbf{c}_0 + u_1\mathbf{c}_1 + \cdots + u_{k-1}\mathbf{c}_{k-1} \quad (2.4)$$

onde $u_i \in \{0, 1\}$, $0 \leq i \leq k{-}1$.

A Equação 2.4 pode ser escrita em termos da matriz geradora (**G**) e o vetor de mensagem, $\mathbf{u} = (u_0, u_1, \ldots, u_{k-1})$:

$$\mathbf{c} = \mathbf{u} \times \mathbf{G} \quad (2.5)$$

onde

$$\mathbf{G} = \begin{pmatrix} \mathbf{c}_0 \\ \mathbf{c}_1 \\ \vdots \\ \mathbf{c}_{k-1} \end{pmatrix} = \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,n-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{k-1,0} & c_{k-1,1} & \cdots & c_{k-1,n-1} \end{pmatrix} \quad (2.6)$$

Portanto, a matriz geradora **G** consiste em uma matriz com *k* colunas linearmente independentes (LI) mais *n-k* colunas linearmente dependentes das primeiras, responsáveis por adicionar redundância à mensagem.

Para obter a codificação na forma sistemática, a matriz geradora deve ter a seguinte forma:

$$\mathbf{G} = [\mathbf{I} \mid \mathbf{P}] \quad (2.7)$$

onde **I** é a matriz identidade de tamanho $k \times k$ e **P** (Equação 2.8) é a matriz de paridade do código, com tamanho $k \times (n - k)$:

$$\mathbf{P} = \begin{pmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,n-1} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{k-1,0} & p_{k-1,1} & \cdots & p_{k-1,n-1} \end{pmatrix} \quad \mathbf{(2.8)}$$

A matriz **G** tem a função de gerar palavras-código pertencentes a C. Por exemplo, considere a matriz geradora de um código C (7, 4) abaixo:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \mathbf{(2.9)}$$

Para codificar a mensagem 1011, é realizada a multiplicação vetorial entre ela e a matriz geradora, resultando a palavra-código 1011010, ou seja:

$$\mathbf{c} = \begin{pmatrix} 1 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \quad \mathbf{(2.10)}$$

Assim como a matriz geradora pode ser utilizada para gerar as palavras de um código, a matriz paridade correspondente, **H**, pode ser utilizada para decodificar as palavras-código. Como C é um espaço vetorial k-dimensional, existe um espaço $(n\text{-}k)$-dimensional dual a C, gerado pelas linhas da matriz paridade, tal que, para toda palavra $\mathbf{c} \in C$, o seguinte ocorre:

$$\mathbf{G} \times \mathbf{H}^{T} = \mathbf{0} \quad \mathbf{(2.11)}$$

Considerando que a matriz **G** pode ser representada como $\mathbf{G} = [\mathbf{I}_k \mid \mathbf{P}]$, a matriz **H** correspondente pode ser obtida a partir de **G** da seguinte maneira:

$$\mathbf{H} = [\mathbf{P}^{T} \mid \mathbf{I}] \quad \mathbf{(2.12)}$$

Como exemplo, considere a matriz geradora do código C(5,3) a seguir:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ \underbrace{0 \quad 0 \quad 1}_{I_k} & & & \underbrace{1 \quad 1}_{P} \end{pmatrix} \tag{2.13}$$

Utilizando a Equação 2.12, obtém-se:

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{2.14}$$

Conforme mencionado, a matriz paridade pode ser utilizada para decodificar as palavras recebidas. Para tal, o primeiro passo consiste em calcular a síndrome (**s**), definida como:

$$\mathbf{s} = \mathbf{c} \times \mathbf{H}^{T} \tag{2.15}$$

Como exemplo, considere um código de bloco C(6,3) com as seguintes matrizes:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \; \mathbf{H} = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.16}$$

Suponhamos que a palavra **c** = 111000 tenha sido transmitida, a qual, devido a ruído no canal, tenha sido recebida como **c*** = 111001. Utilizando a equação 2.15, obtém-se:

$$\mathbf{s} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix} \tag{2.17}$$

Quando todas as posições de **s** são zero, significa que a mensagem recebida não foi corrompida durante a transmissão (ou houve tantos erros, os quais, coincidentemente, levaram a outra palavra-código). Por outro lado, se **s** for diferente de **0**, significa que ocorreu um erro na posição de **H** que coincide com o valor de **s**. Nesse exemplo, como **s** = 001, o que coincide com a última coluna de **H**, o erro está na última posição de **c***, portanto resultando, após a correção, 111000, que é o valor que havia sido originalmente transmitido.

3.2.1 Códigos de Hamming

Uma das classes de códigos de blocos mais conhecidas é aquela dos códigos de Hamming. Para todo inteiro $m \geq 3$, existe um código de Hamming com as seguintes características:

- Número de bits da mensagem original: $k = 2^m - m - 1$
- Número de bits de paridade: $n - k = m$
- Comprimento da palavra após codificação: $n = 2^m - 1$
- Capacidade de correção: $t = 1$ erro

O processo de codificação de um código de Hamming é ditado pela Equação 2.5 (**c** = **u**×**G**), onde **u** é a mensagem original e **G** é a matriz geradora do código. Já o processo de decodificação é baseado na matriz de paridade, **H**, iniciando-se com o cálculo da síndrome (Equação 2.13), conforme descrito acima.

O exemplo a seguir mostra o processo completo de codificação e decodificação de um código de Hamming C(7,4), no qual as matrizes **G** e **H** são dadas por:

$$\mathbf{G} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{pmatrix} \quad \mathbf{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Considere que a palavra a ser transmitida seja **u** = 0111. Portanto, após a codificação, a palavra-código gerada resultante é 0111010. Se, durante a transmissão, essa palavra tiver seu quarto bit invertido, resultará0110010 no receptor. A síndrome resultante será 011, que coincide com a quarta coluna de **H**. Portanto, o quarto bit da palavra recebida deve ser corrigido (invertido), resultando 0111010. Tomando os quatro primeiros bits, temos 0111, que coincide com a mensagem original.

3.2.2 Códigos Reed-Solomon (RS)

Os códigos Reed-Solomon foram criados por Irving S. Reed e Gustave Solomon em 1960 (REED; SOLOMON, 1960). A principal característica desses códigos é que eles visam corrigir blocos de erros ao invés de erros distribuídos aleatoriamente. Eles são utilizados, por exemplo, em CDs e DVDs, pois nesses casos os erros são frequentemente longos (causados

por riscos ou manchas no CD/DVD). Esses códigos também são usados para transmissões de longa distância, como no caso da espaçonave Voyager2.

A operação do código Reed-Solomon é baseada em bloco de símbolos ao invés de blocos de bits. A notação (*n*, *k*) é novamente utilizada, agora indicando que o bloco de saída contém um total de *n* símbolos (ao invés de *n* bits) e o bloco de entrada contém *k* símbolos (ao invés de *k* bits). Por consequência, $m = n - k$ é o número de símbolos de paridade. O total de bits nesses codificadores é, portanto, $k.b$ na entrada e $n.b$ na saída, onde b é o número de bits por símbolo.

Os códigos RS podem ser considerados como uma subclasse dos códigos BCH (Bose, Chaudhuri e Hocquenghem), que é uma classe de códigos de bloco lineares e cíclicos. Os códigos BCH são considerados uma generalização dos códigos de Hamming, pois podem ser projetados para ter qualquer capacidade de correção (*t*). Eles podem ser definidos no campo binário GF (2) ou sobre o campo de Galois GF(q), que é o caso dos códigos Reed-Solomon. Já o fato de serem cíclicos é devido ao fato que qualquer deslocamento cíclico do vetor código em C resulta em outro vetor código que também pertence a C. Por exemplo, considere o código C(4,3) = {0000, 0011, 0101, 0110, 1001, 1010, 1100, 1111}; se a palavra 0110 for deslocada para a direita, obtém-se 0011, que também pertence ao código. Esta propriedade dos códigos cíclicos possibilita a geração dos mesmos através da multiplicação da mensagem (como um polinômio) com o polinômio gerador. Por exemplo, seja o código de Hamming C(7,4) com polinômio gerador **G** = $1 + x + x^3$; a codificação é obtida pela multiplicação do vetor de informação (**u** = 1010 = $1 + x^2$, na sua representação polinomial) pelo polinômio gerador do código, resultando $(1 + x^2) . (1 + x + x^3)$ = 1110010.

A Figura 2.4(PEDRONI, 2008) mostra um codificador RS (255,223). Os parâmetros desse código são:

- Distância mínima de Hamming: d_{min}= 33

- Número de bits em cada símbolo: b= 8

- Número de símbolos na entrada: $k = 2^b - d_{min}$

- Número de símbolos na saída: n = $2^b - 1$

- Número de bits de paridade: $m = n - k$

- Capacidade de correção: $\lfloor (d_{min} - 1)/2 \rfloor$

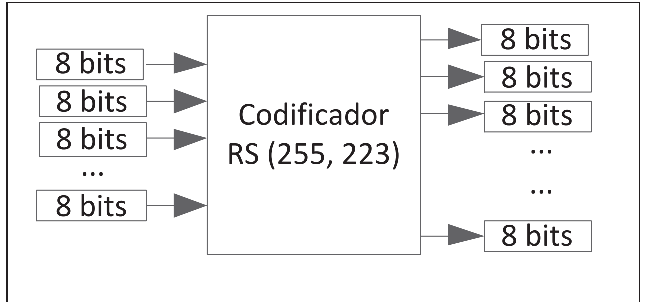


Figura 2.4 - Codificador RS (255,223).

3.2.3 Códigos Low Parity Density Check (LDPC)

Os códigos LDPC foram introduzidos por Gallager em 1962 (GALLAGER, 1962), entretanto só se popularizaram em 1996, através de MacKay e Neal (MACKAY; NEAL, 1997).

Códigos LDPC são códigos de bloco lineares, também representados pelo par (n, k), onde n e k são a quantidade de bits na entrada e na saída do codificador, respectivamente. O número de bits de redundância é portanto $m = n - k$.

Esses códigos são gerados utilizando-se uma matriz de verificação de paridade (**H**) grande (até mesmo com milhares de elementos) e esparsa (poucos '1's e muitos '0's). Após a construção da matriz **H**, que pode ser escrita como $\mathbf{H} = [\mathbf{P}^T \mid \mathbf{I}_{n-k}]$, a matriz geradora **G** é obtida através de $\mathbf{G} = [\mathbf{P} \mid \mathbf{I}_k]$.

Os códigos LDPC podem ser classificados como randômicos ou estruturados, dependendo do método utilizado para a geração da matriz **H**. Normalmente, códigos LDPC randômicos tem desempenho superior aos códigos LDPC estruturados, entretanto os estruturados são menos complexos para decodificar.

Códigos LDPC são geralmente descritos usando gráficos de Tanner. A Figura 2.5 mostra um exemplo de uma matriz **H** e seu respectivo gráfico de Tanner. O gráfico é composto por m nós de teste e n nós de bits. Cada nó de bit representa uma equação de verificação de paridade. Por exemplo, o nó de teste 1 tem conexões com os nós de bits 1, 2 e 3, correspondentes às colunas de valor 1 (da linha 1) de **H**. O mesmo esquema é seguido para os demais nós de teste. O conjunto de testes é equivalente a calcular $\mathbf{c}.\mathbf{H}^T$, que deve ser zero quando a palavra correta é recebida.

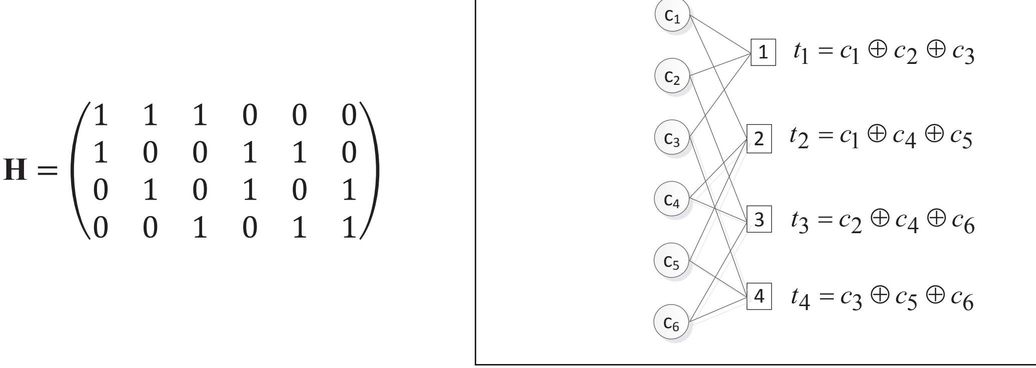


Figura 2.5 - Gráfico de Tanner para um código LDPC (6,2).

Os algoritmos utilizados para decodificar os códigos LDPC são chamados de algoritmos de passagem, pois passam mensagens de um lado para o outro em um grafo de Tanner (TANNER, 1981). Eles são divididos em dois grupos, denominados propagação de crença e soma-produto. Como exemplo, considere novamente o código LDPC que tem a matriz de verificação de paridade apresentada na Figura 2.5. O processo de decodificação ocorre da seguinte maneira: quando uma mensagem é recebida, os testes t1 a t4 são executados; se todos produzirem zero, então a palavra-código recebida é valida e o processo de decodificação finalizado; caso contrário, ocorre a passagem de informação, ou seja, cada nó de teste calcula a informação que deve devolver aos nós de bit. Por exemplo, o nó de teste 1 recebe informações dos nós de bit 1, 2 e 3. A informação que deve retornar ao nó de bit 1 é a soma (ou-exclusivo) da informação dos demais nós de bits que se conectam ao nó de teste 1, ou seja, $t_{11} = c_2 \oplus c_3$. A mesma regra se aplica aos demais nós de bit ligados ao nó de teste 1, ou seja, $t_{12} = c_1 \oplus c_3$ e $t_{13} = c_1 \oplus c_2$. O conjunto completo de informações passadas dos nós de teste para os nós de bit é a seguinte:

- Nó de teste 1 para os nós de bit 1, 2 e 3: $t_{11} = c_2 \oplus c_3$, $t_{12} = c_1 \oplus c_3$, $t_{13} = c_1 \oplus c_2$
- Nó de teste 2 para os nós de bit 1, 4 e 5: $t_{21} = c_4 \oplus c_5$, $t_{24} = c_1 \oplus c_5$, $t_{25} = c_1 \oplus c_4$
- Nó de teste 3 para os nós de bit 2, 4 e 6: $t_{32} = c_4 \oplus c_6$, $t_{34} = c_2 \oplus c_6$, $t_{36} = c_2 \oplus c_4$
- Nó de teste 4 para os nós de bit 3, 5 e 6: $t_{43} = c_5 \oplus c_6$, $t_{45} = c_3 \oplus c_6$, $t_{46} = c_3 \oplus c_5$

Baseado nesses valores recebidos, cada nó de bit atualiza seu valor utilizando uma função de maioria. De acordo com esta função, um bit da palavra-código (nó de bit) será invertido se a maioria dos bits de informação recebidos for diferente do valor atual.

Como exemplo, considere novamente o código da Figura 2.5, e considere que a palavra-código **c** = 110100 tenha sido enviada, com corrupção durante a transmissão, resultando **c*** = 100100 (representada no gráfico de Tanner da figura 2.6(a)). Os testes t1 a t4 são executados, resultando em 1010. Como nem todos os testes obtiveram 0 como resposta, o processo de decodificação deve continuar. As informações que devem retornar aos nós de bit são calculadas e cada nó de bit decide, utilizando a função de maioria, se inverte ou não seu valor corrente. Para o nó de bit 1, por exemplo, calcula-se $c_1 = \text{maioria } (t_{11}, t_{21}, c_1) = \text{maioria } (0,1,1) = 1$, indicando que o nó de bit 1 não precisa inverter seu valor. Já para o nó de bit 2, tem-se $c_2 = \text{maioria } (t_{12}, t_{32}, c_2) = \text{maioria } (1,1,0) = 1$, ou seja, c_2 deve inverter seu valor. A Figura 2.6(b) mostra o gráfico de Tanner após a atualização de todos os nós de bit. Novamente, os testes t1 a t4 são executados, resultando 0000. Portanto, o processo de decodificação está concluído e a palavra-código obtida é 110100, idêntica a **c**.

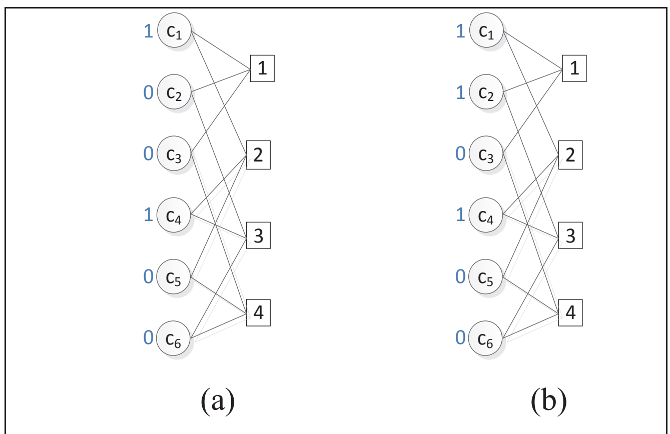


Figura 2.6 - Decodificação de um código LDPC.

2.3 CÓDIGOS CONVOLUCIONAIS

Os códigos convolucionais foram introduzidos por (ELIAS, 1955) e atualmente são utilizados em diversas aplicações. Estes códigos operam serialmente, processando apenas um bit (ou um pequeno grupo de bits) por vez. Os códigos convolucionais utilizam memória, pois as palavras-código geradas dependem tanto da palavra de entrada atual como de palavras anteriores, assim como da estrutura do codificador.

3.3.1 Códigos Convolucionais Convencionais

Um codificador convolucional (n, k, m) toma k bits de informação por vez e produz n bits na saída. O parâmetro m está relacionado à ordem da memória, representando o grau do polinômio de maior grau dentre os polinômios utilizados, ou seja, é o tamanho do maior registrador de deslocamento utilizado para armazenar os bits anteriores do sistema. Outro parâmetro considerado é o *overall constraint lenght* (K), que se refere ao número de entradas anteriores ($u[i$-1], ..., $u[i$-$m]$) que afetam as saídas ($c(0)[i]$, ..., $c(n$-1$)[i]$) no tempo i. K pode ser descrito como a soma de todos os flip-flops nos registradores de deslocamento.

A Figura 2.7apresenta um exemplo de um codificador convolucional (2, 1, 2), sendo K=2. O polinômio gerador $g(x)$ deste codificador é escrito em função dos coeficientes iguais a 1; por exemplo, para $c(0)$ e c(1) temos g_1=1+D+D^2 e g_2=1+D^2, respectivamente, sendo D o atraso de um ciclo de clock (flip-flop) (MORELOS-ZARAGOZA, 2002).

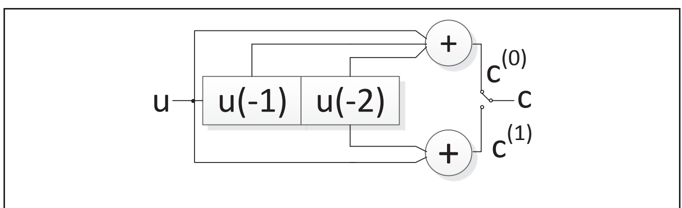


Figura 2.7 - Codificador convolucional (2, 1, 2).

Um codificador convolucional, como o próprio nome sugere, pode ser definido como a convolução entre a resposta do codificador ao impulso a sequência de dados na entrada (CLARK; CAIN, 1981). Os códigos convolucionais são lineares e, por conseguinte, podem ser representados por matrizes. A resposta ao impulso de qualquer sequência de entrada pode ser produzida através da operação de módulo-2 com a devida versão da entrada deslocada, resultando em uma matriz geradora **G.**

Diferentemente dos códigos de blocos, os códigos convolucionais não necessitam que a informação esteja organizada em forma de blocos. Desta maneira, a matriz **G** pode ser considerada uma matriz semi-infinita. A palavra-código correspondente a qualquer sequência de entrada **u** pode ser encontrada através da multiplicação do vetor de entrada por **G**, segundo a Equação 2.5 ($\mathbf{c} = \mathbf{u} \times \mathbf{G}$).

O codificador convolucional pode também ser representado através de uma máquina de estados finita (Figura 2.8), sendo o número de estados necessários dado por 2^K. A Tabela

2.1 apresenta todos os estados possíveis desta máquina, considerando a informação de entrada e seu estado atual (MORELOS-ZARAGOZA, 2002).

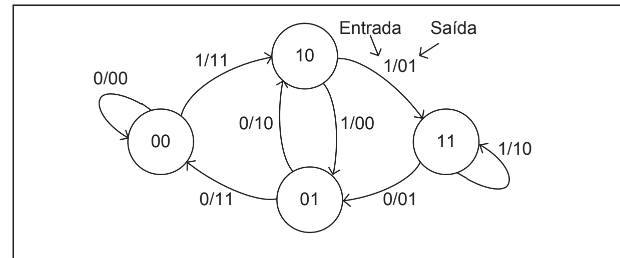


Figura 2.8 - Diagrama de estados de um codificador convolucional (2, 1, 2).

Tabela 2.1 - Entrada/saída e estados possíveis de um codificador convolucional (2, 1, 2).

Estado atual s0[i] s1[i]	Informação (entrada) u[i]	Próximo estado s0[i+1] s1[i-1]	Saída c(0)[i] c(1)[i]
00	0	00	00
00	1	10	11
01	0	00	11
01	1	10	00
10	0	01	10
10	1	11	01
11	0	01	01
11	1	11	10

O diagrama de estados de um codificador convolucional pode ser representado por um diagrama de treliças, onde os estados são conectados (entre os tempos i e i+1) por ramificações, de acordo com a tabela do codificador. A Figura 2.9 apresenta um diagrama de treliça equivalente ao codificador já visto na figura 2.8. Dentro dos círculos estão os valores de cada estado e fora deles os valores de $c(0)$ e $c(1)$. Um parâmetro importante é a distância livre mínima (d_{free}) do código, que tem influência na capacidade de correção de erros, sendo d_{free} igual ao menor peso de Hamming (w) dentre todos os caminhos que partem de A e retornam a A.

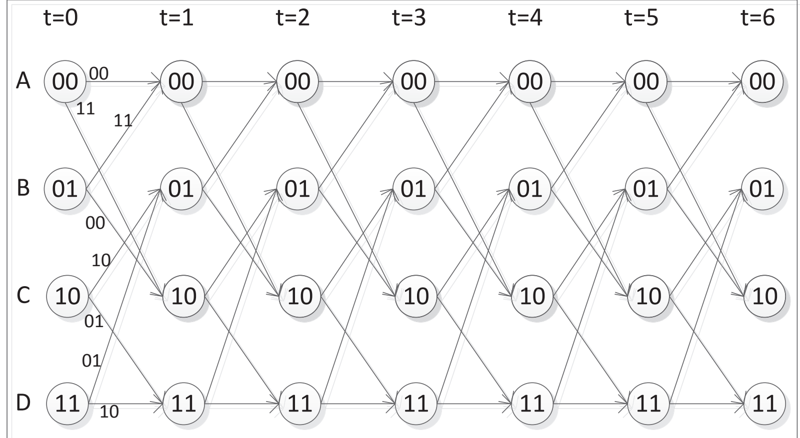


Figura 2.9 - Diagrama de treliça para o codificador convolucional (2, 1, 2).

3.3.2 Códigos Turbo

Berrou, Glavieux e Thitimajshiba introduziram, em 1993, uma nova e aparentemente revolucionária técnica de codificação, chamada *codificação turbo* (BERROU; GLAVIEUX; THITIMAJSHIMA, 1993). Ela consiste essencialmente de uma concatenação paralela de dois códigos binários convolucionais, decodificada por um algoritmo de decodificação iterativo (CASTIÑEIRA; FARRELL, 2006).

Códigos turbo utilizam codificadores convolucionais sistemáticos recursivos (RSCs) em sua construção. A Figura 2.10 mostra o modelo de um codificador RSC (2, 1, 2). Este codificador é semelhante ao mostrado na Figura, porém apresenta uma entrada recursiva (isto é, bits armazenados são realimentados à entrada). Além disso, uma das saídas é a própria entrada (portanto, agora $c(1) = u$). Uma vantagem deste codificador é que ele tende a produzir palavras-código com pesos mais altos que as do codificador da Figura, o que resulta em um desempenho superior em canais com baixa relação sinal ruído (SNR). Outra vantagem é que a decodificação não é exageradamente complexa, levando os mesmos a serem utilizados em várias aplicações, como telefones celulares de terceira geração e comunicação por satélite.

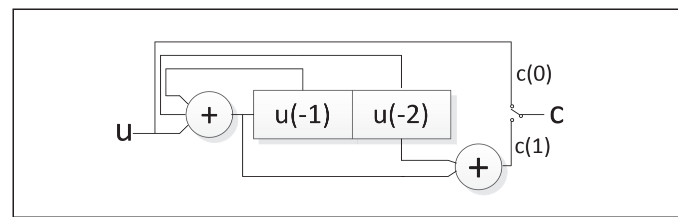


Figura 2.10 - Codificador RSC (2, 1, 2).

Na construção dos códigos turbo são utilizados dois destes codificadores RSC em paralelo, como mostra a Figura 2.11. Existe também um intercalador na entrada do segundo codificador, de modo que ele recebe uma sequência de bits diferentes. Cada codificador tem apenas uma saída, pois a outra saída é a própria entrada. Nesse caso, a sequência de saída é $uc(0)c(1)uc(0)c(1)$ ... , resultando uma taxa global $r = 1/3$.

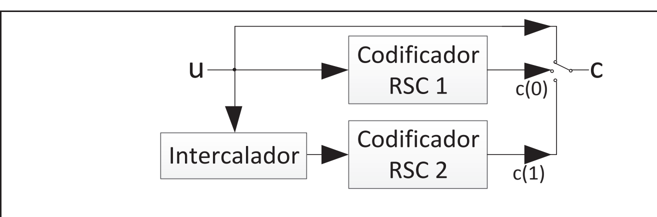


Figura 2.11 - Codificador turbo.

CAPÍTULO 3

TÉCNICAS DE DECODIFICAÇÃO

3.1 INTRODUÇÃO

Nesta seção, são descritas algumas técnicas existentes para decodificação de códigos corretores de erros, tais como: decodificação de máxima verossimilhança, decodificação de Viterbi, decodificação com decisão abrupta e com decisão suave e decodificação por conjuntos de informação.

3.2 DECODIFICAÇÃO DE MÁXIMA VEROSSIMILHANÇA (MLD)

A codificação de máxima verossimilhança é a mais poderosa das técnicas de decodificação. Entretanto, a complexidade de um MLD é bastante alta, pois todas as palavras-código precisam ser analisadas (CLARK; CAIN, 1981) (HOFFMAN et al, 1991).

Esta decodificação baseia-se no fato de que a probabilidade de ocorrer um único erro na sequência transmitida é maior do que a probabilidade de ocorrerem dois, e assim sucessivamente. A probabilidade de um padrão particular de *i* erros ocorrerem é:

$$\boldsymbol{P_e^i(1-P_e)^{n-i}} \quad \textbf{(3.1)}$$

Sendo assim, o algoritmo que toma a sequência recebida como sendo a palavra candidata que tem a menor métrica (menor distância de Hamming, por exemplo) seleciona a palavra que tem maior probabilidade de ser a palavra correta. Um decodificador que implementa esta regra de decodificação obtém palavras com chances mínimas de serem incorretas. Podemos considerar que o MLD é um decodificador ótimo para códigos pequenos, porém sua complexidade (número de candidatas a analisar) cresce exponencialmente com o número de bits.

Como exemplo, considere um código que consiste em quatro palavras-código: 00000, 00111, 11100 e 11011. A Tabela 3.1 inclui as 32 sequências possíveis que poderiam ser recebidas pelo decodificador. As primeiras 24 posições da tabela são compostas por

sequências que tem distância de Hamming igual a 1 (quando comparadas com as 4 palavras aceitas pelo código), enquanto as demais têm distância de Hamming igual a 2. Considerando que neste exemplo a métrica utilizada é a menor distância de Hamming, é mais provável que a palavra transmitida esteja entre as primeiras posições (CLARK; CAIN, 1981).

Tabela 3.1 - Sequências possíveis para um código C (5,2).

00000	11100	00111	11011
10000	01100	10111	01011
01000	10100	01111	10011
00100	11000	00011	11111
00010	11110	00101	11001
00001	11101	00110	11010
10001	01101	10110	01010
10010	01110	10101	01001

3.3 DECODIFICAÇÃO DE VITERBI

O decodificador de Viterbi é o mais comum dos decodificadores para códigos convolucionais. Ele é um decodificador por máxima verossimilhança, porém, graças à sua construção engenhosa, não necessita fazer comparações com todas as sequências possíveis. O procedimento consiste em calcular a distância acumulada entre a sequência recebida até o instante t_i. O cálculo é feito para todos os estados da treliça, e para vários instantes de tempo, para que a menor distância seja encontrada.

A Figura 3.1 apresenta um exemplo de um decodificador de Viterbi. A métrica utilizada no código é a distância mínima de Hamming, para decisão abrupta. Neste exemplo, a palavra **u** = 0110000... é codificada na sequência **c** = 00 11 01 01 11 00 00..., que é transmitida. Devido a ruído no canal, essa mensagem chega corrompida no decodificador, isto é, **c*** = 00 11 00 00 11 00 00.... O procedimento de correção inicia-se a partir do estado A, ao receber os primeiros dois bits (**c*** = 00); a distância de Hamming é calculada para os dois possíveis caminhos e armazenada em uma variável que acumula a distância total até aquele momento (Figura 3.1(a)). O próximo valor recebido é **c*** = 11 e novamente as distâncias são calculadas e a nova métrica acumulada. O algoritmo decide qual caminho escolher com base na menor métrica. O procedimento é repetido até o final da sequência de entrada. O caminho com menor métrica está destacado na Figura 3.1(h). A palavra original é obtida percorrendo o

caminho inverso, observando-se qual dos ramos foi escolhido em cada transição. O diagrama de treliça é estruturado da seguinte maneira: o ramo superior corresponde a uma entrada 0, enquanto o ramo inferior corresponde a uma entrada 1; sendo assim, é possível facilmente recuperar a sequência de entrada, que é 0110000. O algoritmo pode ser acompanhado passo a passo na Figura 3.1 (PEDRONI, 2008).

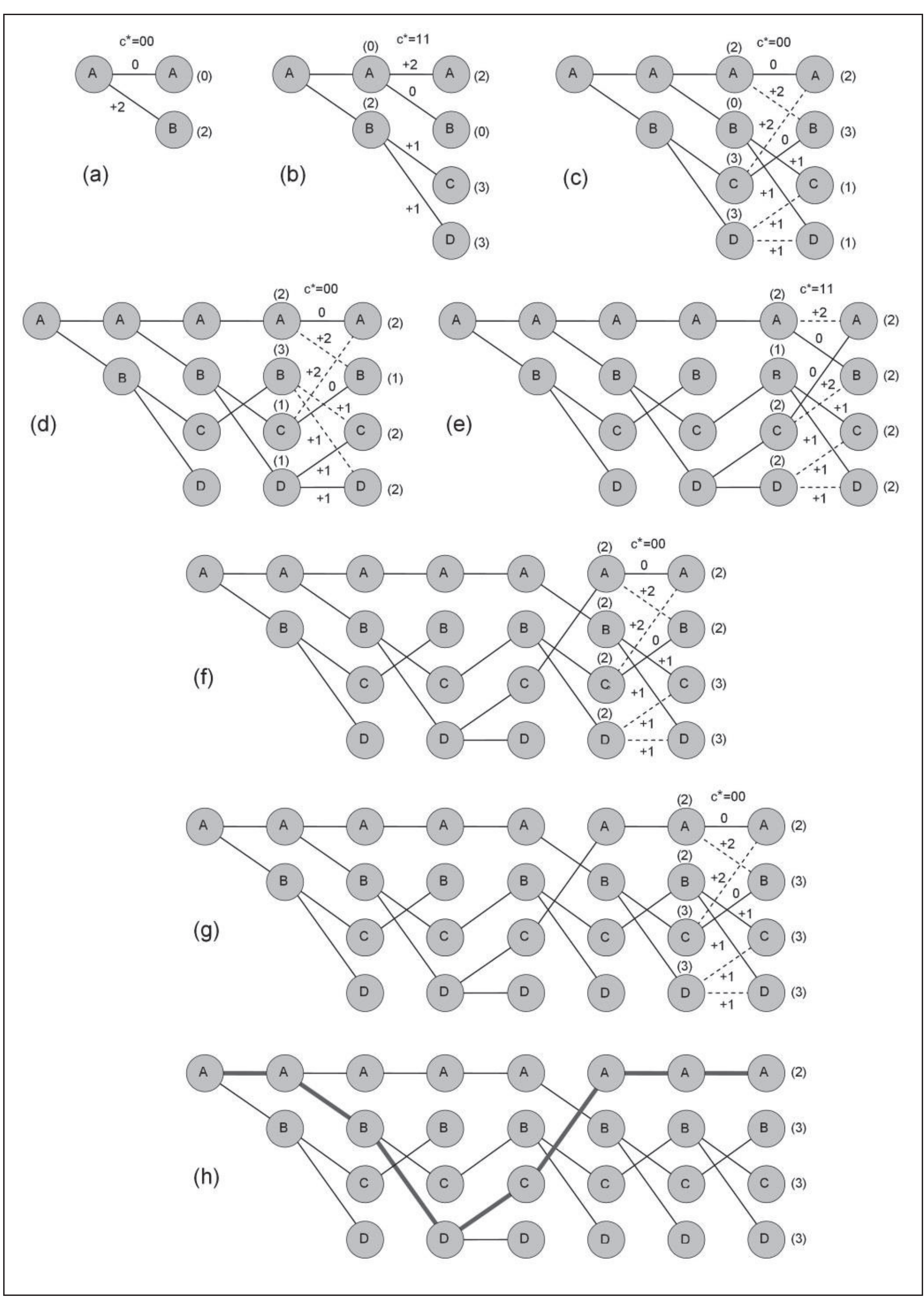


Figura 3.1 - Procedimento de Decodificação de Viterbi.

3.4 DECODIFICAÇÃO COM DECISÃO ABRUPTA X DECODIFICAÇÃO COM DECISÃO SUAVE

Para melhor descrever a diferença entre uma decodificação abrupta e uma decodificação suave, considere um código de Hamming C(7,4) onde a mensagem a ser transmitida é **u** = 0000. Após a codificação, realizada pelo codificador de canal, a palavra-código obtida é **c** = 0000000. Esta palavra é entregue ao modulador (BPSK), que simplesmente a mapeia para a palavra-código a ser transmitida -1 -1 -1 -1 -1 -1 -1, usando as regras 0 $\rightarrow$ -1 e 1 $\rightarrow$ 1. Durante a transmissão, a palavra-código é modificada pelo ruído, resultando a sequência -0,8 0,2 -0,1 -0,6 -0,7 -0,8 -0,5 no receptor.

Na decodificação de decisão abrupta, considera-se apenas se o valor recebido é positivo ou negativo, sendo atribuído o valor 1 para todo valor positivo e -1 para todo valor negativo. Para este exemplo, a sequência resultante seria -1 +1 -1 -1 -1 -1 -1, como mostra a Figura 3.2. Esta sequência é entregue ao demodulador, que a remapeia, seguindo a regra inversa à utilizada no modulador, resultando a palavra demodulada é 0100000.

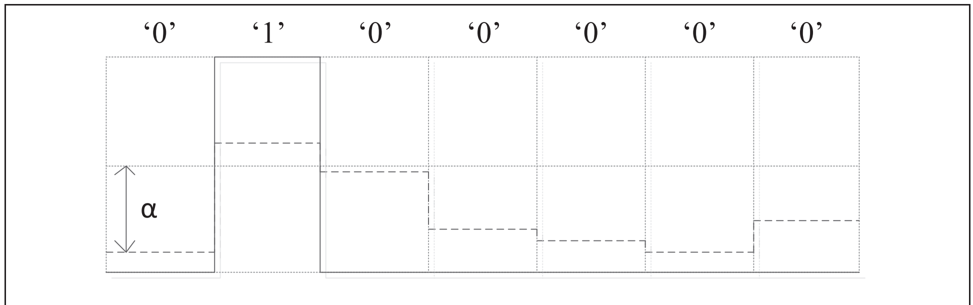


Figura 3.2 - Decisão abrupta x Decisão suave.

Se ao invés da decisão abrupta fosse utilizada a decisão suave neste processo de decodificação, mais uma variável seria considerada: a *confiabilidade*. Nesse caso, considera-se quão negativo ou quão positivo é o valor recebido. Quanto mais os valores analógicos se aproximarem do limite inferior (-1) ou do limite superior (+1), mais confiáveis serão os mesmos. Por exemplo, para a palavra analógica -0,8 0,2 -0,1 -0,6 -0,7 -0,8 -0,5, a sequência de confiabilidade resultante seria 1 6 7 4 3 2 5, sendo que o valor menos confiável da palavra é -0,1 e o mais confiável -0,8. O decodificador recebe as duas variáveis e as utiliza no processo de decodificação. A palavra final (considerada como sendo igual à mensagem enviada) é seleciona entre as candidatas de acordo com a métrica determinada no algoritmo de decodificação (MORELOS-ZARAGOZA, 2002) (GODOY, 1991).

A decodificação por decisão suave é computacionalmente mais complexa que a abrupta, entretanto apresenta um desempenho superior, conseguindo corrigir mais erros com a mesma quantidade de bits de redundância.

3.5 DECODIFICAÇÃO POR CONJUNTOS DE INFORMAÇÃO

Em um código (n, k), um *conjunto de informação* (*Information Set* - IS) é definido como sendo qualquer conjunto de k vetores linearmente independentes (LI) da matriz geradora (CLARK; CAIN, 1981). As n-k posições restantes constituem o conjunto de paridade. Então, se a matriz geradora para o código puder ser escrita na forma canônica, as primeiras k colunas (iniciando da esquerda para a direita) formam um conjunto de informação.

Qualquer outro conjunto de vetores formará um conjunto de informação se for possível, através de operações elementares com as linhas, fazer com que a coluna correspondente da matriz geradora tenha peso unitário. Por exemplo, considere o código de Hamming (7,4) que tem a seguinte matriz geradora:

$$\mathbf{G}_0 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

As primeiras k posições de **G** formam um conjunto de informação que pode ser representado por $I_0 = \{1, 2, 3, 4\}$.

Outra matriz geradora para o mesmo código é obtida adicionando-se a primeira linha à terceira e à quarta, onde a matriz resultante é:

$$\mathbf{G}_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

O resultado dessas operações elementares fizeram com que as colunas 1 e 5 fossem trocadas de lugar e a 6 modificada. Agora as colunas 2, 3 4 e 5 formam o conjunto de informação ($I_1 = \{2, 3, 4, 5\}$).

De modo semelhante, pode ser obtida uma terceira matriz geradora (**G_2**) para o mesmo código C(7,4), onde $I_2 = \{1, 2, 6, 7\}$. Para que essa troca de posições de colunas seja possível, é necessário que as duas colunas tenham o valor 1 na mesma linha.

$$\mathbf{G}_2 = \begin{pmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Como os símbolos contidos no conjunto de informação podem ser especificados independentemente, eles definem unicamente uma palavra-código. Se não existir erros nessas posições, os símbolos restantes na palavra-código transmitida podem ser reconstruídos. Esta propriedade provê a base para todos os algoritmos de decodificação que usam conjuntos de informação. Estes algoritmos serão vistos no capítulo 5.

CAPÍTULO 4

ALGORITMOS DE DECODIFICAÇÃO OTIMIZADOS PARA IMPLEMENTAÇÃO EM HARDWARE E ARQUITETURA

4.1 INTRODUÇÃO

Este capítulo descreve alguns algoritmos existentes para a decodificação de códigos corretores de erros por conjunto de informação (*information-set* - IS). Adicionalmente, um novo algoritmo de decodificação otimizado para a implementação em hardware é apresentado. Este algoritmo é baseado no algoritmo de Dorsch (DORSCH, 1974), porém conta com uma série de modificações que o tornam menos custoso em hardware.

4.2 ALGORITMO DE DECODIFICAÇÃO POR CONJUNTOS DE INFORMAÇÃO

Os algoritmos baseados em conjunto de informação iniciam o processo de decodificação selecionando vários conjuntos de informação. Em seguida, é construída uma palavra-código para cada conjunto, assumindo que os símbolos no conjunto de informação são corretos. Finalmente, cada palavra candidata é comparada com a sequência atual recebida, e a palavra-código que mais se aproximar da palavra desejada (segundo uma métrica pré-determinada) é selecionada. Se a sequência recebida não contém erros no conjunto de informação, este procedimento garante que a palavra-código transmitida faz parte do conjunto de palavras candidatas.

Como exemplo de utilização deste algoritmo considere um código de Hamming C(7,4), cujas matrizes geradoras são G_0, G_1 e G_2 (mostradas na seção 4.5). Suponha que foi transmitida a palavra-código **c** = 1011100, porém a recebida foi **c*** = 1010100 (com o quarto bit invertido). No primeiro passo, os conjuntos de informação são selecionados: $I_0 = \{1, 2, 3, 4\}$, $I_1 = \{2, 3, 4, 5\}$ e $I_2=\{1, 2, 6, 7\}$. O segundo passo é construir uma palavra-código para cada conjunto, utilizando as matrizes geradoras correspondentes:

Para I_0: As posições {1, 2, 3, 4} da palavra **r** são multiplicadas pela matriz $\mathbf{G}_0$.

Palavra candidata 0: 1010001

Para I_1: As posições {1, 2, 4, 5} da palavra **r** são multiplicadas pela matriz $\mathbf{G}_1$.

Palavra candidata 1: 0110100

Para I_2: As posições {1, 2, 6, 7} da palavra **r** são multiplicadas pela matriz $\mathbf{G}_2$.

Palavra candidata 2: 1011100

A métrica utilizada neste exemplo é à distância de Hamming, portanto a palavra-código com menor distância de Hamming será a selecionada:

Palavra candidata 0: $\left.\begin{matrix}1010001\\1010100\end{matrix}\right\} d_H=2$

Palavra candidata 1: $\left.\begin{matrix}0110100\\1010100\end{matrix}\right\} d_H=2$

Palavra candidata 2: $\left.\begin{matrix}1011100\\1010100\end{matrix}\right\} d_H=1$

Logo, a palavra-código escolhida é 1011100.

4.3 ALGORITMO DE DECODIFICAÇÃO DE DORSCH

Devido à necessidade de calcular a distância de Hamming entre a palavra recebida e cada palavra pertencente ao código, os decodificadores por máxima verossimilhança possuem elevada complexidade computacional. Para resolver este problema, diversas abordagens têm sido investigadas. O uso de conjuntos de informação é uma tentativa de reduzir essa complexidade, considerando apenas uma pequena fração do total de palavras candidatas. A utilização do IS na decodificação foi inicialmente proposta por Prange (PRANGE, 1962), e posteriormente estudada por diversos pesquisadores (DORSCH, 1974) (COFFEY, GOODMAN, 1990) (FOSSORIER, LIN, 1995) (FOSSORIER, 2002).

Em 1974, Dorsch criou um algoritmo interessante devido à sua simplicidade de implementação em software. O algoritmo utiliza as informações obtidas na decisão suave para classificar os símbolos recebidos. Neste algoritmo, a matriz geradora ou a matriz de verificação de paridade é reconstruída, de modo que os símbolos menos confiáveis sejam de paridade e os demais o conjunto de informação. Porém, nem todos os símbolos mais confiáveis podem ser utilizados como um conjunto de informação. Neste caso, o IS é ajustado para que um conjunto válido (n-k colunas linearmente independentes na matriz de verificação de paridade) possa ser obtido e utilizado na decodificação por verossimilhança (SWEENEY, 2002).

O exemplo a seguir mostra passo a passo o algoritmo de Dorsch aplicado a um código de Hamming (7, 4, 3), com a seguinte matriz verificação de paridade:

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Considere que (3 4 7 1 0 5 7) é a palavra recebida (já quantizada em três bits). As posições menos confiáveis são a 1ª e a 2ª, níveis 3 e 4, respectivamente, pois são as mais próximas do limiar estabelecido na decisão abrupta, como mostra a Tabela 4.1.

Tabela 4.1 - Confiabilidade e decodificação abrupta da palavra transmitida.

Palavra analógica	**3**	**4**	**7**	**1**	**0**	**5**	**7**
Posição	1	2	3	4	5	6	7
Confiabilidade	0	0	3	2	3	1	3
Decodificação com decisão abrupta	0	1	1	0	0	1	1

No algoritmo de Dorsch as colunas da matriz **H** são rearranjadas de acordo com a confiabilidade da palavra recebida. As colunas de **H** correspondentes às posições menos confiáveis são trocadas pelas colunas correspondentes às posições mais confiáveis (que compõem o IS), portanto as n-k posições de **H** serão compostas pelos símbolos menos confiáveis . O processo inicia-se pela posição 1, que será trocada pela 7. Portanto, a matriz **H** agora é:

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Que, na forma canônica, é equivalente à:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Na sequência, a posição 2 é trocada pela 6. Resultando em:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

A última iteração seria a troca da posição 2 (correspondente ao nível 5 da sequência recebida e que originalmente encontrava-se na posição 6) pela 5, entretanto não é possível escrever esta matriz na forma canônica:

$$\mathbf{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Sendo assim, esse passo é desconsiderado e uma nova tentativa realizada. O próximo nível menos confiável encontra-se na posição 4, portanto, a posição 4 é trocada pela 5:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Agora se tem uma nova palavra analógica ordenada pelo novo conjunto de informação (7 5 7 0 1 4 3). Esta palavra será utilizada no próximo passo, que é a decodificação.

Uma versão do conjunto de informação, utilizando-se decisão abrupta, é extraída, resultando 1110. Esta versão é recodificada, gerando a palavra candidata 1110001. Para calcular a distância entre a palavra recebida e a palavra candidata, a seguinte regra é utilizada:

$$d(x_i, c_i) = \begin{cases} x_i \mid c_i = 0 \\ 2^m - 1 - x_i \mid c_i = 1 \end{cases} \quad \textbf{(4.1)}$$

$$d(\mathbf{x},\mathbf{c}) = \sum_i d(x_i, c_i) \quad \textbf{(4.2)}$$

onde $m = n - k$. Portanto, a distância entre a palavra analógica 7 5 7 0 1 4 3 e a palavra candidata 1110001 é $0 + 2 + 0 + 0 + 1 + 4 + 4 = 11$.

As demais palavras candidatas, nesta versão simplificada do algoritmo de Dorsch (SWEENEY, 2002), são obtidas invertendo-se um bit por vez da versão abrupta do conjunto de informação. Todas devem ser recodificadas e comparadas (utilizando decisão suave) com a palavra recebida. A palavra-código que apresentar menor distância é a palavra escolhida como correta. A Tabela 4.2 mostra todas as palavras candidatas e suas respectivas distâncias da sequência de confiabilidade.

Neste exemplo, a candidata com menor distância é 1110001. Comparando o resultado da decodificação suave e da abrupta (0110011) nota-se que diferem em duas posições.

Tabela 4.2 - Palavras candidatas obtidas com o algoritmo de Dorsch.

Conjunto de informação	Palavra candidata	Distância
1110	1110001	11
1111	1111111	20
1100	1100100	20
0110	0110110	19

No algoritmo original de Dorsch, são utilizadas mais palavras candidatas do que neste exemplo. As candidatas são geradas de forma iterativa tendo por objetivo o MLD.

Uma versão do algoritmo de Dorsch onde se usa um número menor de palavras-código candidatas é discutida a seguir.

4.4 ALGORITMO DE DECODIFICAÇÃO OTIMIZADO PARA IMPLEMENTAÇÃO EM HARDWARE

O decodificador apresentado nessa tese é resultado de um amplo projeto de pesquisa (GORTAN et al., 2010a) (GORTAN et al., 2010b) (GORTAN, 2011) (FRANÇA et al., 2011) (GORTAN et al., 2012), no qual o algoritmo foi gradualmente otimizado para atingir uma maior eficiência na implementação em hardware. Este algoritmo de decodificação por decisão

suave, baseado em conjuntos de informação, tem desempenho muito próximo ao do decodificador de máxima verossimilhança (MLD), porém com um número muito menor de comparações.

O princípio básico da decodificação, comum às três implementações apresentadas, pode ser descrito pelos seguintes passos:

1) A partir da palavra recebida, extrair sua versão demodulada por decisão abrupta (**r**) e a sequência de confiabilidades correspondente (**s**). A sequência de confiabilidade é a palavra analógica ordenada, de modo decrescente, de acordo com a confiabilidade dos símbolos.
2) Baseando-se nos valores de **s**, selecionar os k bits mais confiáveis em **r**, e desprezar os $n - k$ bits restantes.
3) Recodificar os k bits mais confiáveis usando uma nova matriz $\mathbf{G}_n$, derivada da **G** original, porém contendo colunas unitárias nas k posições mais confiáveis.

Uma maneira de se obter a matriz $\mathbf{G_n}$ é através da inversão de uma matriz formada pelas k colunas mais confiáveis de **G**, e então multiplicá-la pela matriz **G** original. Um problema deste procedimento é que nem todos os subconjuntos de k colunas de **G** são LI, e portanto a inversão nem sempre é possível. Nesse caso, um novo conjunto de k símbolos deve ser selecionado, e o processo é reiniciado até que k colunas LI sejam encontradas.

Outra possibilidade, parcialmente baseada em (DORSCH, 1974), mas para a qual um espaço de busca reduzido pode ser demonstrado (GORTAN et al., 2010b), foi selecionada para a implementação do decodificador em hardware. Nessa abordagem, a matriz geradora é manipulada através de operações lineares (redução de Gauss-Jordan modificada) para reduzir as colunas desejadas a vetores unitários.

Como citado anteriormente, o algoritmo foi gradualmente otimizado para atingir uma maior eficiência na implementação em hardware. A seguir cada uma dessas otimizações será descrita em detalhes.

4.4.1 Algoritmo Otimizado com k+1 Candidatas

Na primeira otimização, apenas k+1 palavras candidatas são analisadas para determinar a palavra vencedora (GORTAN et al., 2010a). O funcionamento desse algoritmo é

apresentado a seguir e ilustrado na Figura 4.1, para um código C(7,4, 3) cuja matriz **G** é mostrada na Figura 4.1(a).

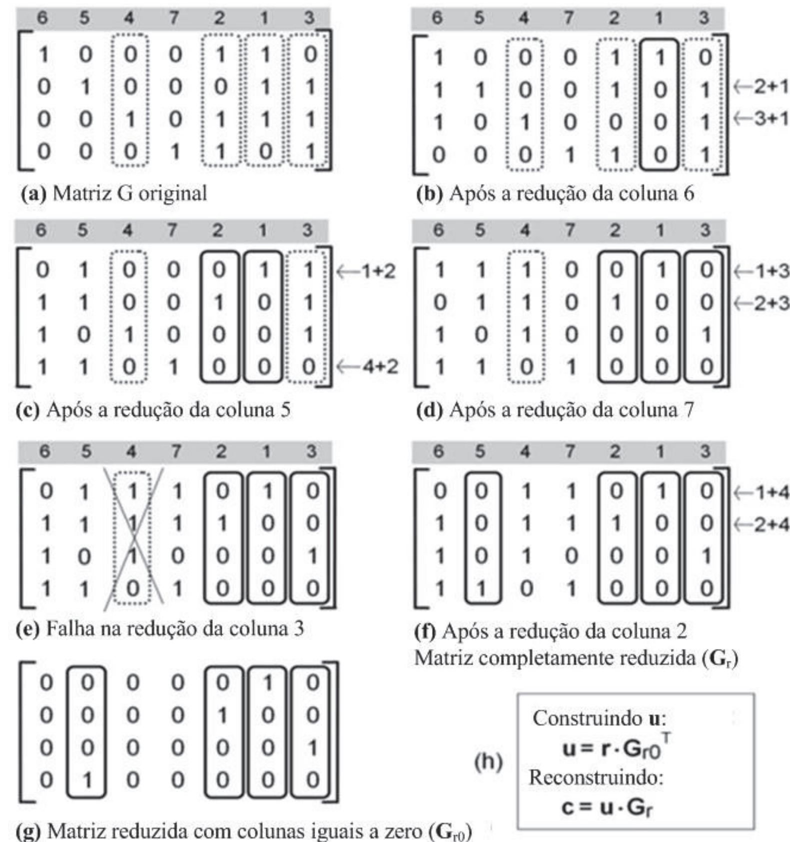


Figura 4.1 - Algoritmo de decodificação proposto em (GORTAN et al., 2010a).

Para este exemplo, consideremos que uma mensagem original **u** = 0100 é codificada em uma palavra-código **c** = 0100011, modulada e transmitida por um canal ruidoso. No decodificador, a sequência recebida (usando codificação em 3 bits para os valores analógicos) é **x** = 2 5 1 3 0 7 6. O algoritmo de decodificação consiste nos seguintes passos:

1) Demodular a palavra recebida por decisão abrupta, obtendo-se a sequência **r** = 0100011, e gerar a sequência de confiabilidades que é igual a **s** = 6 5 7 3 2 1 4. Observe que os valores de confiabilidade associados a cada coluna são mostrados no topo das matrizes da Figura 4.1.
2) Usando transformações lineares, transformar as k colunas mais confiáveis de **G** em vetores unitários. Embora não seja possível garantir que essas k colunas sejam LI, caso seja determinado que o conjunto analisado é LD o processo não precisa ser reiniciado; basta substituir a coluna menos confiável pela próxima coluna indicada pelo vetor de confiabilidades **s**. Este caso é ilustrado na Figura 4.1 (b) a (f). A coluna 6 (a mais confiável) foi reduzida na Figura 4.1 (b), a coluna 5 na Figura 4.1 (c) e a coluna 7 na

Figura 4.1 (d). Porém, na Figura 4.1 (e), não foi possível reduzir a coluna 3 a um vetor unitário, indicando que o conjunto de colunas não é LI. O algoritmo prossegue então para a coluna 2 (a próxima mais confiável), a qual é reduzida com sucesso na Figura 4.1 (f), resultando na nova matriz reduzida $\mathbf{G}_r$.

3) Criar a matriz $\mathbf{G}_{r0}$, a qual é simplesmente $\mathbf{G}_r$ com todas as colunas não selecionadas substituídas por vetores nulos. Esta operação é mostrada na Figura 4.1(g).
4) Multiplicar $\mathbf{r}$ por $\mathbf{G}_{r0}$ para obter uma versão estimada da mensagem original (ou seja, $\mathbf{u}_0 = \mathbf{r}\times\mathbf{G}_{r0}$, como indicado na Figura 4.1(h)).
5) Construir o conjunto de todas as mensagens candidatas, simplesmente invertendo um bit de $\mathbf{u}_0$ para cada candidata. Assim, o número total de mensagens candidatas é $k + 1$, sendo que cada candidata é representada por $\mathbf{u}_i$ ($i = 0$ até k).
6) Recodificar as mensagens candidatas usando $\mathbf{c}_i = \mathbf{u}_i\times\mathbf{G}_r$ para obter um conjunto de palavras-código candidatas.
7) Calcular a distância entre cada candidata $\mathbf{c}_\mathbf{i}$ e a palavra recebida $\mathbf{x}$; a candidata que apresentar a menor distância em relação à $\mathbf{x}$ será escolhida como a saída do decodificador.

Como medida da distância entre as candidatas, pode-se utilizar a distância euclidiana ou alguma métrica simplificada. Em (SWEENEY, 2002) é proposta a métrica apresentada na Equação 5.1 e 5.2, a qual foi selecionada para a implementação em hardware devido a sua simplicidade.

Para o exemplo apresentado, a Tabela 4.3 apresenta todas as palavras candidatas e suas respectivas distâncias. Observando-se essa tabela verifica-se que a palavra $\mathbf{c}_0$apresenta a menor distância, portanto, é a selecionada. Visto que este código é sistemático, as quatro primeiras posições de $\mathbf{c}_0$são referentes à mensagem e os demais são os bits de redundância. Portanto, a mensagem recuperada é 1011, idêntica à enviada.

Tabela 4.3 - Distância suave entre as palavras candidatas.

Mensagem candidata	**Palavra candidata**	**Distância**
$\mathbf{u}_0 = 1011$	$\mathbf{c}_0 = 0100011$	9
$\mathbf{u}_1 = 0011$	$\mathbf{c}_1 = 0111001$	22
$\mathbf{u}_2 = 1111$	$\mathbf{c}_2 = 1111111$	25
$\mathbf{u}_3 = 1001$	$\mathbf{c}_3 = 1110010$	22
$\mathbf{u}_4 = 1010$	$\mathbf{c}_4 = 1001011$	16

A Figura 4.2 apresenta uma possível arquitetura para este decodificador. Este diagrama será discutido detalhadamente no capítulo 6, onde é descrita a arquitetura do hardware utilizada no design do chip dedicado (ASIC) que implementa esta versão do decodificador (SIBILLA et al., 2011).

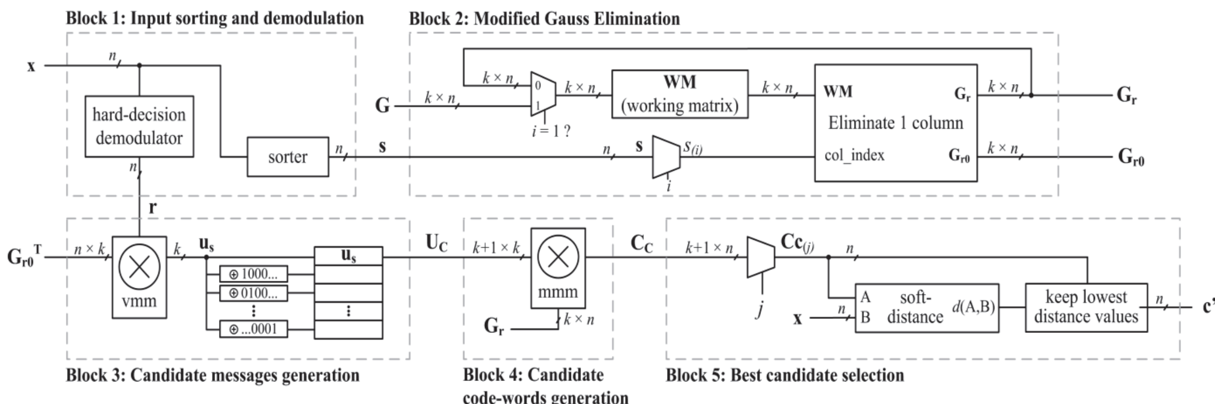


Figura 4.2 - Diagrama de blocos do decodificador utilizando-se *k*+1 candidatas (GORTAN et al., 2010a).

Em (GORTAN et al., 2010a) é demonstrado que neste algoritmo o número de iterações necessárias para que o conjunto de informação seja encontrado nunca é maior que n-$d_{\min}$+1, e que k+1 palavras-código candidatas são suficientes para atingir um desempenho próximo à do MLD. Observando a Figura 4.3, que mostra a desempenho de um código C(48, 24, 12), é possível notar que com 25 (ou seja, k+1) candidatas a perda de desempenho é muito pequena em relação ao MLD (abaixo de 0,1 dB). Como esperado, o desempenho degrada-se com um número menor de palavras candidatas; para o código da Figura 4.3, a perda de desempenho usando 13 candidatas é igual a 0,85 dB.

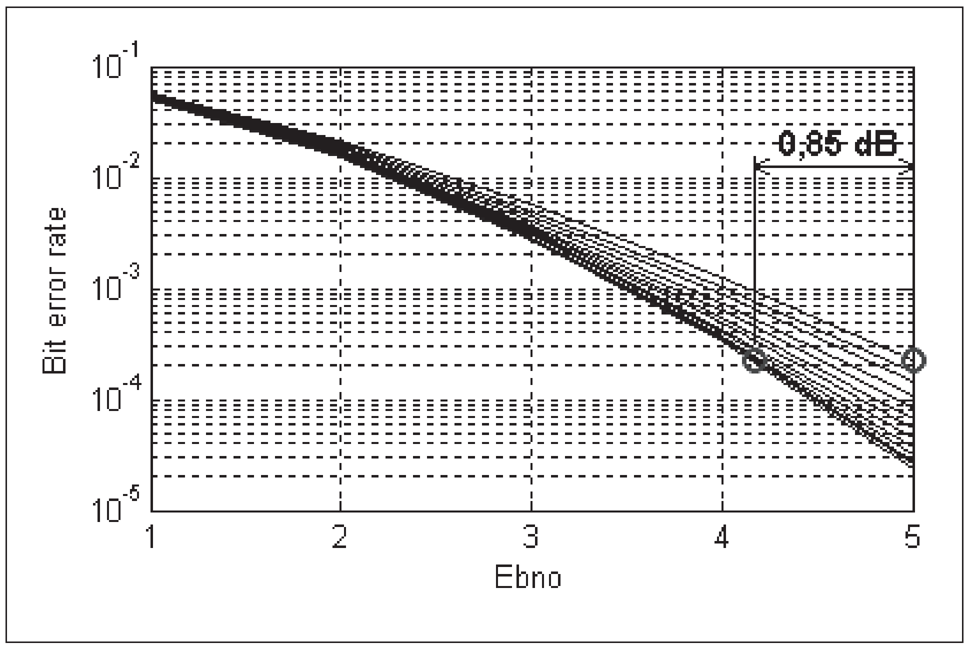


Figura 4.3 - Comparação de desempenho (GORTAN et al., 2010a).

4.4.2 Algoritmo Otimizado com *m* Candidatas

Na segunda otimização é proposto um critério para gerar um conjunto de mensagens candidatas de tamanho arbitrário (com m candidatas) (GORTAN et al., 2010b). Neste algoritmo, ao invés de utilizar um número fixo de mensagens como no anterior (k+1), são utilizadas m candidatas, onde m é determinado baseando-se no desempenho desejado do decodificador (isto é, quão próximo do desempenho do MLD deseja-se estar). É demonstrado que m é consideravelmente menor que 2^k candidatas, número necessário para o MLD.

A estratégia para determinar as candidatas mais prováveis de um dado código de bloco $C(n, k, d)$ consiste em:

- Aleatoriamente, gerar um grande número de palavras-código $\mathbf{c} \in C$, sujeitas ao ruído gaussiano branco aditivo (*Aditiva White Gaussiana Nise* - AWGN).
- Estimar a probabilidade de ocorrência de cada padrão de erro, baseado no número de ocorrências de cada padrão de erro.
- Ordenar os padrões de erro de acordo com sua probabilidade.

A partir disso, um conjunto de padrões de erro (também referido como *bit-flipping patterns*) de um dado código é obtido. Esses padrões são utilizados da seguinte maneira: considere que uma palavra-código $\mathbf{c}$ (possivelmente corrompida) é recebida, e que a partir dela é possível determinar a palavra-código mais provável de ser a correta, sendo $\mathbf{u}_0$ a mensagem candidata de referência a partir da qual $\mathbf{c}$ foi originada. O processo de

decodificação inicia recodificando-se $\mathbf{u}_0$ e calculando-se a distância suave entre a palavra recodificada e a palavra analógica recebida. Então, o próximo padrão de erro mais provável é aplicado a $\mathbf{u}_0$ e a próxima candidata mais provável é obtida, a qual também é recodificada e calculada a distância suave. Este processo continua até que todos os padrões de *bit-flipping* tenham sido aplicados a $\mathbf{u}_0$. Ao final do processo, a candidata com a menor distância suave é selecionada como vencedora.

A Figura 4.4 apresenta o diagrama de blocos desta versão do decodificador. É possível observar que a alteração realizada no algoritmo implica em alterações apenas no bloco 3, onde as mensagens candidatas são geradas.

É importante mencionar que, em geral, as candidatas mais prováveis são as que apresentam apenas um bit invertido, seguidas das candidatas com dois bits invertidos e assim por diante. Entretanto, este não é sempre o caso para todos os padrões. Além disso, é necessário determinar quais entre todas as candidatas com o mesmo número de bits invertidos são mais prováveis. Para tanto, diversas simulações foram necessárias para classificar as candidatas.

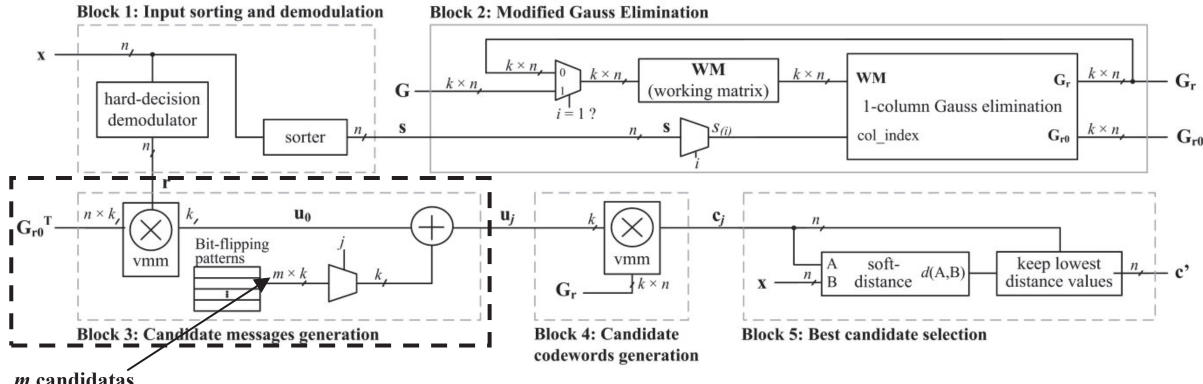


Figura 4.4 - Diagrama de blocos do decodificador utilizando *m* candidatas (GORTAN et al., 2010b).

A consideração mais relevante desta análise é que após testar apenas as candidatas com padrões de erro de um e dois bits invertidos, o decodificador atinge um desempenho muito próximo a do MLD (>99%). Em outras palavras, ao invés de examinar 2^kcandidatas como o MLD faz, apenas poucas palavras foram necessárias para obter quase o mesmo desempenho.

Um exemplo é mostrado na Tabela 4.4 para o código $C(24, 12, 8)$. Observando-se a tabela é possível notar que inspecionando apenas m=25 candidatas é alcançada aproximadamente 98% do desempenho obtido na decodificação MLD (a qual usaria 4096 candidatas). Para m=100 candidatas (que corresponde a apenas 2,44% dos casos), o

desempenho é próximo de 99,8%. Este abordagem que permite que *m* seja um número pequeno é crucial para a viabilidade de implementação em hardware.

Tabela 4.4 - Entrada/saída e estados possíveis de um codificador convolucional (2, 1, 2).

***m* (para*k*=12)**	**Padrão *Bit-flipping***	**Proposto / MLD**
1 (somente $\mathbf{u}_0$)	000000000000	71.89 %
…	…	…
25	000000100010	97.97 %
…	…	…
100	000001100001	99.76 %
…	…	…
$2^{12} = 4096$	111111111111	100 %

Resultados mais detalhados são apresentados na Figura 4.5, onde é mostrada a degradação do desempenho (em dB) para vários códigos em relação ao MLD em função do número de palavras candidatas sob várias condições de ruído.

Esta figura auxilia o projetista a determinar qual é o número de candidatas que precisará utilizar para atingir o desempenho pretendido. Por exemplo, considere que o decodificador pode ter no máximo 0,1 dB de degradação em relação ao MLD. Neste caso se usaria os seguintes números de palavras candidatas:

- Para C(15, 7, 5): 5 candidatas (entre 128 possíveis)
- Para C(24, 12, 8): 12 candidatas (entre 4096 possíveis)
- Para C(48, 12, 12): 100 candidatas (entre $16{,}8\times10^6$possíveis)
- Para C(66, 33, 12): 400 candidatas (entre $8{,}6\times10^9$ possíveis)

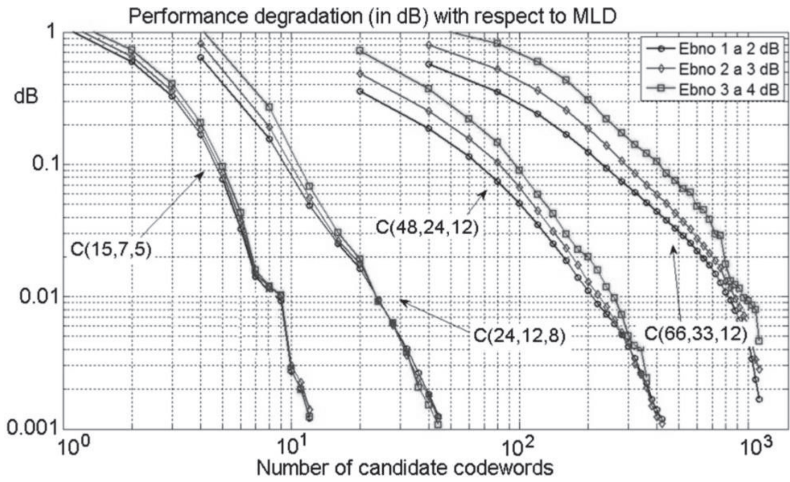


Figura 4.5 - Degradação do desempenho (em dB) em relação ao MLD, em função do número de palavras-código candidatas (GORTAN et al., 2010b).

4.4.3 Algoritmo Otimizado com Critério de Parada

Na decodificação suave, quanto maior o número de candidatas verificadas maior o desempenho do decodificador, aproximando-se do resultado do MLD (considerado um decodificador ótimo). Embora o grande número de candidatas aumente o desempenho do decodificador, acaba tornando o processo de decodificação mais lento. Por essa razão, muitos critérios de aceitação (ou de parada) foram propostos, visando indicar quando uma candidata correta é encontrada, eliminando a necessidade de inspecionar as demais candidatas. Pode-se citar como exemplos de critério de parada o teste de Distância Mínima Generalizada (*Generalized Minimum Distance* - GMD), proposto por Forney (FORNEY, 1966), a regra do cone e mais recentemente os critérios Taipale-Pursley (TAIPALE, PURSLEY, 1991) e o BGW (BARROS et al., 1997).

Um exemplo trivial de critério de parada é verificar a distância euclidiana entre a palavra recebida e uma dada palavra-código. Se a distância for menor que a distância euclidiana mínima do código, a candidata é imediatamente declarada como vencedora. Embora este critério seja fácil de aplicar, é pouco eficiente e não apresenta uma redução significativa no número de operações computacionais realizadas para justificar sua utilização. Existem, entretanto, outros critérios de parada, com diferentes graus de eficiência e demanda computacional.

No critério de parada GMD considera-se que para um dado código de comprimento n e distância mínima de Hamming $d_{H\min}$, dada uma sequência recebida $\mathbf{x}$ e uma palavra-código $\mathbf{c}$, demodulada em BPSK, a seguinte equação deve ser satisfeita:

$$\langle \boldsymbol{x}, \boldsymbol{c} \rangle > \boldsymbol{n} - \boldsymbol{d_{Hmin}} \quad (4.3)$$

onde $\langle x, c \rangle$ representa o produto interno entre $\mathbf{x}$ e $\mathbf{c}$. Embora o critério GMD seja de simples implementação, não é utilizado devido sua baixa eficiência. Para um código C(15, 7, 5), por exemplo, a redução no número de operações computacionais é de apenas 1,77%, para um razão de E_b/N_0 de 5 dB (GORTAN et al., 2012).

A regra do cone é baseada no princípio de que se o ângulo formado entre a palavra-código recebida e a palavra-código candidata for menor que a metade do menor ângulo entre duas palavras-código, então a palavra-código recebida está dentro da região de Voronoi da palavra candidata (a região de Voronoi $V(\boldsymbol{x})$ de um determinado vetor $\boldsymbol{x} \in \boldsymbol{X}$ constitui-se no conjunto de todos os vetores $\boldsymbol{y} \in \boldsymbol{R^n}$ mais próximas de $\boldsymbol{x}$ do que qualquer outro vetor $\boldsymbol{x'} \in$

$\boldsymbol{X}$). O critério é válido para todos os casos onde os módulos de todas as palavras-código forem iguais, como é o caso para modulação BPSK. Analiticamente, o critério do cone afirma que **x** pertence à região de Voronoi de **c** se:

$$\langle \boldsymbol{x}, \boldsymbol{c} \rangle > ||\boldsymbol{x}|| \times \sqrt{\boldsymbol{n} - \boldsymbol{d_{Hmin}}} \quad \textbf{(4.4)}$$

Onde $||\boldsymbol{x}|| = \sqrt{\langle \boldsymbol{x}, \boldsymbol{x} \rangle}$ é o comprimento de **x**. Ao contrário do GMD, a regra do cone apresenta uma redução substancial no número de candidatas avaliadas. Por exemplo, para o código **C**(15, 7, 5), uma redução de 82,2% é verificada com uma razão de E_b/N_0 de 5 dB (GORTAN et al., 2012). Entretanto, enquanto no GMD (Equação 5.3) a distância é sempre comparada com um valor constante, na regra do cone (Equação 5.4) é necessária à comparação com o módulo de **x**, o qual deve ser recalculado para cada nova sequência recebida.

O critério de parada proposto por Taipale e Pursley pode ser descrito da seguinte maneira. Considere um código **C** com distância mínima de Hamming $d_{H\min}$, uma sequência recebida **x** (cujos símbolos $\mathbf{x}_j$ tem confiabilidade β_j, $0 < \beta_j < 1$), e uma palavra candidata **c**. Esta palavra **c** é considerada a melhor candidata se satisfizer a seguinte condição:

$$\sum_{\boldsymbol{j} \in \boldsymbol{S}} \boldsymbol{\beta_j} \leq \sum_{\boldsymbol{j} \in \boldsymbol{T}} \boldsymbol{\beta_j} \quad \textbf{(4.5)}$$

Onde:

- S é o conjunto de índices j tal que os componentes c_j (de **c**) e x_j (de **x**) têm sinais opostos, ou seja, $S = \{j \mid \text{sgn}(c_j) \neq \text{sgn}(x_j), 0 \leq j \leq n\}$. Este conjunto tem cardinalidade $|S|$.
- T é o conjunto de $\delta = d_{H\min} - |S|$ índices no conjunto complementar $\bar{S}$ que tem o menor valor de β_j.

O critério de parada BGW proposto por Barros, Godoy e Wille tem desempenho superior ao GMD e a regra do cone. Este critério apresenta um desempenho equivalente ao Taipale-Pursley (GODOY, WILLE, 2010), entretanto sua implementação em hardware é mais simples. O BWG pode ser descrito da seguinte maneira: considere um código **C** com distância mínima de Hamming $d_{H\min}$, uma sequência recebida **x**, e uma palavra-código candidata **c**. Esta palavra **c** será considerada a melhor candidata se satisfizer a seguinte condição:

$$\sum_{\boldsymbol{j} \in \boldsymbol{Q}} (\boldsymbol{x}\,[+]\boldsymbol{c})_{\boldsymbol{j}} \leq \boldsymbol{0} \quad \textbf{(4.6)}$$

onde Q é o conjunto de $d_{H\min}$ índices j, tal que os j-ésimos componentes de $\mathbf{x}[+]\mathbf{c}_i$ têm os valores mais positivos (ou menos negativos). Na Equação 5.6, a soma híbrida ([+]) (BARROS et al., 1997) entre uma sequência $\mathbf{x} \in \mathbf{R}^n$ e uma palavra-código **c** consiste em inverter o sinal de todos os elementos em **x** para os quais os valores correspondentes em **c** forem iguais a um. Além disso, **x** pertence à região de Voronoi da palavra-código zero se:

$$\sum_{j \in Q} x_j \leq 0 \tag{4.7}$$

Embora o critério de parada possa reduzir o número de operações computacionais, as operações envolvidas consomem uma quantidade significante de ciclos da CPU e/ou uma grande área física (quando implementado em hardware). Sendo assim, uma versão modificada do critério de parada BGW foi proposta (GORTAN, 2011) (GORTAN ET al., 2012), chamada de *BGW orientada para hardware* (*Hardware-Oriented* BGW – HO-BGW) ou BGWG. Esta versão tem o objetivo de tornar o BGW eficiente para implementação em hardware, mantendo basicamente o mesmo desempenho.

A principal vantagem deste algoritmo é que o teste de cada palavra candidata é extremamente simples, e pode ser realizado em um único ciclo de clock. Como a geração de uma candidata também requer apenas um ciclo de clock, o algoritmo não apresenta nenhuma condição de gargalo.

A Tabela 4.5 apresenta uma comparação direta entre os critérios de parada BGW e HO-BGW. Observando-se esta tabela torna-se claro que uma das vantagens do HO-BGW é o fato que o BGW original tem que fazer uma ordenação decrescente da soma híbrida $\mathbf{x'}_n$ para realizar o teste final, enquanto o HO-BGW reutiliza a ordenação de confiabilidade já realizada no inicio do processo de decodificação.

Tabela 4.5 - Comparação direta entre os critérios de parada BGW e HO-BGW (GORTAN, 2011).

BGW	HO-BGW
1) Ordena $\mathbf{x'}_n$ de modo decrescente. 2) Soma os primeiros $d_{H\min}$ valores (os mais positivos). 3) Se a soma é negativa, então **c** é a melhor candidata.	1) Ordena $\mathbf{x'}_n$ de modo decrescente de confiabilidade. 2) Condição C_1: soma dos valores nas posições indicadas por m_1; C_1 é verdadeira se a soma é negativa. 3) Condição C_2: nenhum valor positivo nas posições indicadas por m_2. 4) Se C_1 e C_2 são verdadeiras, então **c** é a melhor candidata.

A condição C_1 é calculada como no BGW original, e é valida apenas em certos casos, indicados por C_2. Em uma implementação em software, os conjuntos S_1 e S_2 podem ser calculados e usados para determinar as condições C_1 e C_2. Entretanto, em hardware é mais eficiente trabalhar com duas máscaras auxiliares, m_1 e m_2. As primeiras k posições de m_1 (mais à esquerda) são copiadas do padrão de *bit-flipping* da candidata atual. As n - k posições (mais à direita) restantes são preenchidas com $d_{H\min}$- m uns. Finalmente, as posições restantes são preenchidas com zeros. A máscara m_2 pode ser derivada a partir de m_1, zerando as k primeiras posições e negando as n - k (mais à direita) de m_1.

Normalmente, nos critérios de parada convencionais o teste de parada é um dos últimos passos no processo de decodificação, sendo um dos estágios mais custosos em termos de operações computacionais. Na maioria das vezes, utilizam operações complexas como somatórios, ordenação e outros processos iterativos, frequentemente implementados como *loops,* sendo necessários vários ciclos de clock para serem executados. O critério de parada HO-BGW elimina essas desvantagens, distribuindo as operações associadas ao critério de parada em todos os estágios de decodificação (correspondentes às linhas em negrito da Figura 4.6).

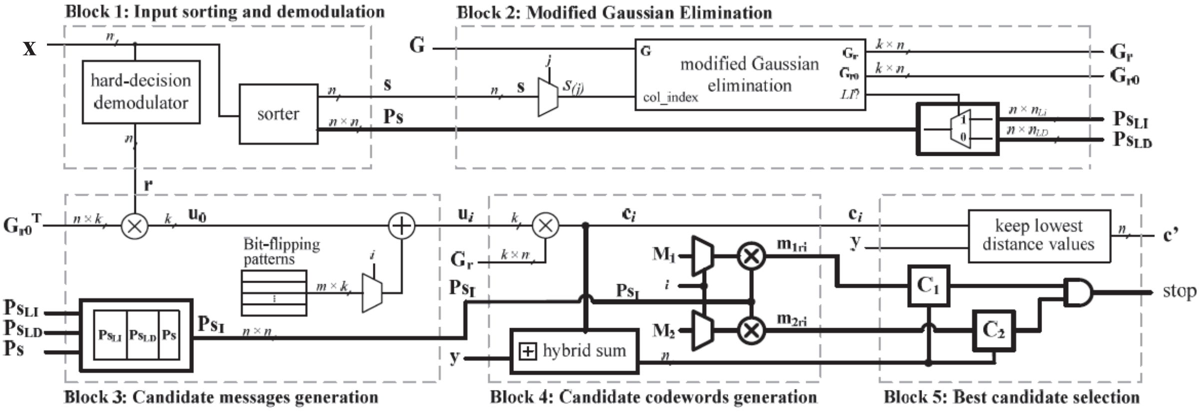


Figura 4.6 - Diagrama de blocos do decodificador com critério de parada (GORTAN, 2011).

A decisão da candidata vencedora no critério HO-BGW é realizada no último passo de decodificação. É um procedimento bastante simples, consistindo apenas em verificar se duas condições de parada (C_1 e C_2, descritas anteriormente) são verdadeiras.

A Figura 4.7 apresenta os resultados de simulações comparando todos os critérios de parada citados anteriormente. Como pode ser visto, o GMD apresenta o pior resultado entre todos os critérios, sendo necessário inspecionar quase todas as candidatas. O critério do cone apresenta um resultado melhor; entretanto, apenas o BGW e o HO-BGW apresentam uma

redução de aproximadamente 90% no número de candidatas analisadas. Para um código pequeno como o **C**(15, 7, 5), ambos apresentam praticamente a mesma eficiência. Entretanto, à medida que o tamanho do código cresce, torna-se mais significativa a diferença entre os dois critérios, sendo o HO-BGW uma versão sub-ótima do BGW.

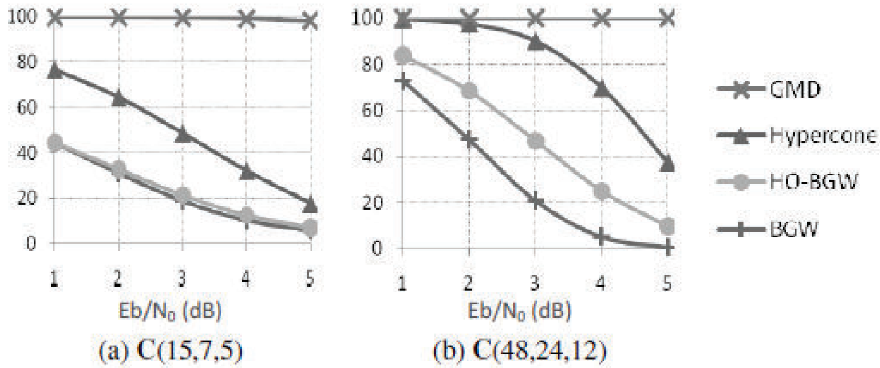


Figura 4.7 - Número de candidatas que precisam ser inspecionadas antes da atuação do critério de parada (em % do total do número de palavras candidatas) (GORTAN, 2011).

CAPÍTULO 5

CIRCUITOS PROPOSTOS PARA O HARDWARE

5.1 INTRODUÇÃO

O chip dedicado (ASIC) desenvolvido como parte deste trabalho de doutorado implementa o algoritmo de decodificação descrito na seção 5.4.1, para um código C(7, 4, 3). Este algoritmo é adequado para a implementação em hardware, pois utiliza um número reduzido e fixo de mensagens candidatas, apenas $k+1$. Nesta escolha, considerou-se a complexidade do projeto e a área de silício disponível para fabricação do chip. Após a seleção do algoritmo, iniciou-se o projeto de uma arquitetura capaz de realizar as operações necessárias em hardware.

5.2 DESCRIÇÃO GERAL DA ARQUITETURA

Um circuito decodificador para códigos de bloco utilizando demodulação suave é uma unidade de processamento digital de sinais, a qual recebe como entrada uma sequência de valores analógicos e fornece em sua saída uma sequência de valores digitais. Devido à natureza sequencial do algoritmo de decodificação, é possível estruturar o circuito decodificador como vários blocos de processamento encadeados em série. Nesse tipo de estrutura, é conveniente utilizar o conceito de *pipelining,* no qual diversas unidades de informação são processadas pelo sistema simultaneamente, porém cada unidade pode estar em diferentes etapas do processamento.

A Figura 5.1 apresenta a *pipeline* do processo de decodificação. Neste processo, o algoritmo é executado em cinco passos distintos. Um diagrama mais detalhado da arquitetura é mostrado na Figura 5.2. Cada um dos cinco blocos mostrados nesta figura implementam um ou mais passos do algoritmo descrito na seção 5.4.1. Estes blocos estão organizados na seguinte sequência:

(1) Demodulação e ordenação da entrada analógica;

(2) Redução de Gauss-Jordan modificada;

(3) Geração das mensagens candidatas;

(4) Geração das palavras-código candidatas;

(5) Seleção da candidata de menor distância.

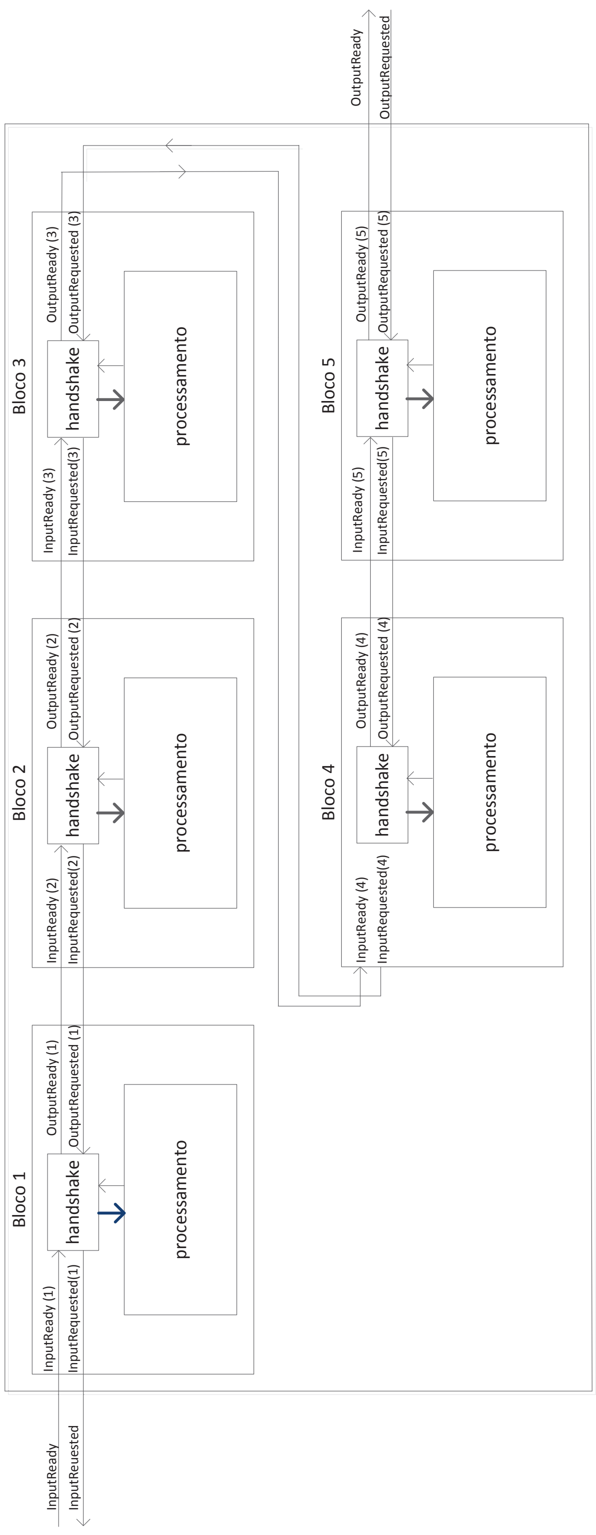


Figura 5.1 - Diagrama geral da rede de handshake.

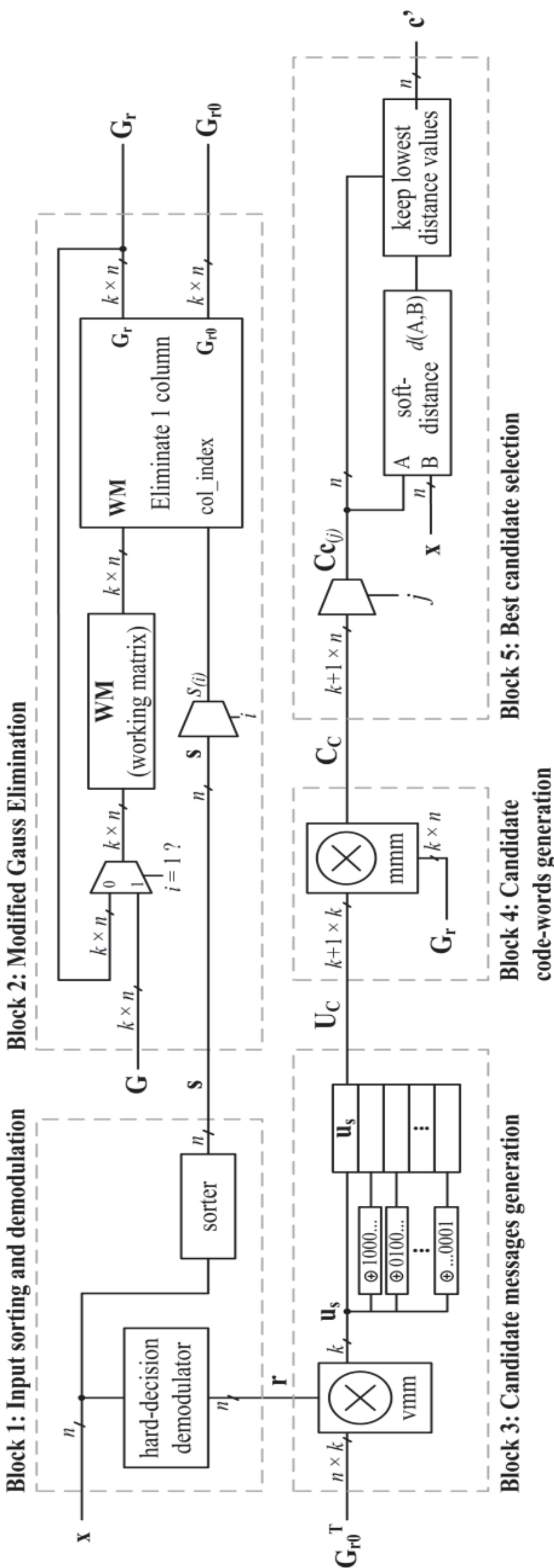


Figura 5.2 - Diagrama de blocos do decodificador utilizando-se *k*+1 candidatas (GORTAN et al., 2010a).

Como visto na descrição do algoritmo (seção 5.4.1), o bloco 1 recebe uma palavra analógica codificada em n bits e produz as sequências **r** (palavra demodulada por decisão abrupta) e **s** (posição dos símbolos em ordem decrescente de confiabilidade). Como cada símbolo de entrada é ordenado no momento em que é registrado, as saídas do bloco 1 estão disponíveis após exatamente n ciclos de clock.

O vetor de confiabilidades **s** é usado como entrada pelo bloco 2, o qual realiza a redução de Gauss-Jordan modificada na matriz geradora **G**. O índice da coluna a ser reduzida é determinado pelo valor de **s** na iteração atual, e o processo de eliminação prossegue até que k colunas LI sejam encontradas. As saídas estão disponíveis após no mínimo k e no máximo n-d_{min}+1 ciclos de clock, como demonstrado em (GORTAN et al. 2010a). As saídas do bloco 2 são a matriz reduzida $\mathbf{G}_r$ e a matriz de transformação $\mathbf{G}_{r0}$, que quando multiplicada pela palavra **r** extrai apenas os bits nas posições do conjunto de informação.

O bloco 3 produz um conjunto de mensagens candidatas, geradas a partir da palavra demodulada **r** e do conjunto de informação selecionado no bloco 2. Primeiramente, a palavra demodulada é multiplicada por $\mathbf{G}_{r0}^{T}$, produzindo a mensagem candidata original $\mathbf{u_0}$. Em seguida, outras k mensagens são geradas, invertendo uma posição de $\mathbf{u}_0$ por vez. Esse processo é totalmente combinacional; portanto, a lista $\mathbf{U_C}$ contendo as k+1 candidatas é gerada em um único ciclo de clock.

Em seguida, o bloco 4 produz a lista de palavras-código candidatas $\mathbf{C_C}$, recodificando as mensagens candidatas através da multiplicação das matrizes $\mathbf{U_C}$ e $\mathbf{G}_r$. Esse processo também é combinacional e ocorre em um único ciclo de clock.

Finalmente, o bloco 5 avalia todas as palavras-código candidatas para selecionar a mais próxima da palavra recebida **x**. A distância entre cada candidata $\mathbf{c_{cj}}$ e a palavra recebida é calculada como descrito nas equações (5.1) e (5.2): para cada símbolo x representado por 3 bits, a distância para um bit ‘0’ será igual a x, e para um bit ‘1’ será igual a 7-x. A distância total entre **x** e $\mathbf{c_{cj}}$ será igual à soma de todas as distâncias dos símbolos individuais. Após calcular a distância para cada candidata, a palavra-código com a menor distância em relação à palavra recebida é selecionada. Cada candidata é avaliada em um ciclo de clock; portanto, o tempo total de processamento para o bloco 5 é de k+1 ciclos de clock.

5.3 DETALHAMENTO DO CIRCUITO DE HANDSHAKE

Cada um dos cinco blocos apresentados na Figura 5.1 instancia uma unidade de *handshake*, responsável por controlar o fluxo dos dados na *pipeline*. Esta unidade sinaliza quando as saídas de um bloco estão prontas, e também quando o bloco está pronto para receber novas entradas. Cada unidade de *handshake* comunica-se com duas outras unidades idênticas, como mostrado anteriormente na Figura 5.1.

O controle realizado pelo *handshake* garante que um bloco nunca irá descartar a sua saída sem que o próximo bloco esteja pronto para recebê-la. Um diagrama demonstrando a utilização da unidade de *handshake* é mostrado na Figura 5.3.

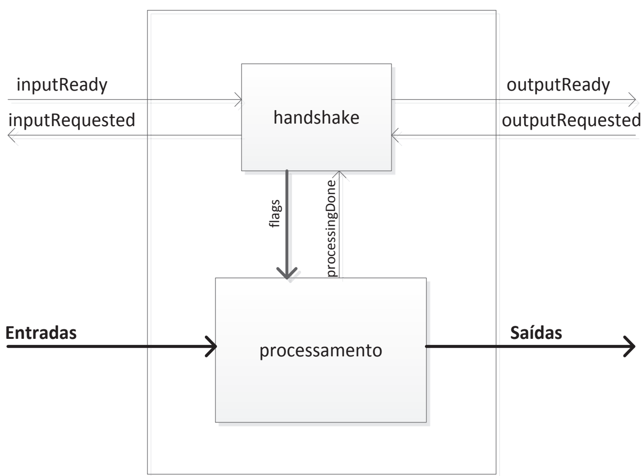


Figura 5.3 - Utilização da unidade de *handshake* em um estágio de processamento.

As entradas do bloco de *handshake* são os sinais *inputReady* (indica quando o bloco está pronto para receber uma nova entrada), *outputRequested* (indica que o próximo bloco está pronto para receber um novo valor) e *processingDone* (indica que o bloco de processamento concluiu suas atividades). As saídas do bloco de *handshake* são os sinais *inputRequested*, *outputReady* e um conjunto de *flags* usado pela unidade de processamento (*awaitingValue*, *firstProcessingCycle*, *lastProcessingCycle* e *outputWillBeConsumed)*. Os sinais *inputRequest* e *outputReady* apresentam nível lógico alto quando o bloco está pronto para receber uma nova entrada e quando uma saída está pronta para ser lida, respectivamente. As demais saídas são *flags* de sinalização: *awaitingValue* é verdadeiro enquanto o processamento não foi iniciado; *firstProcessingCycle* é verdadeiro durante o primeiro ciclo de processamento; *lastProcessingCycle* é verdadeiro durante o último ciclo de processamento; e

outputWillBeConsumed é verdadeiro quando o próximo bloco indicar que consumirá a saída atual.

A unidade de *handshake* contém um contador de iterações que indica o número de ciclos de clock utilizados no processamento. Quando o número de iterações atingir seu valor máximo (diferente para cada bloco do decodificador) ou o sinal de *processingDone* for alto, o sinal de *outputReady* indicará que as saídas do processamento já estão disponíveis para serem utilizadas pelo próximo bloco. Quando a saída for consumida, o bloco estará disponível para receber novas entradas e reiniciar o processamento. A Figura 5.4 apresenta a máquina de estados criada para o contador de iterações.

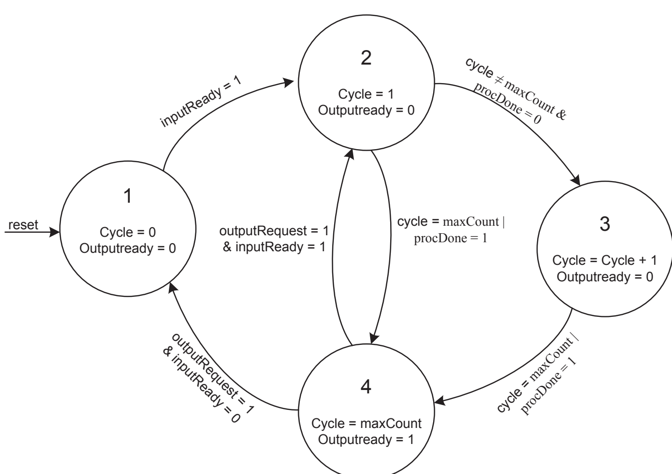


Figura 5.4 - Máquina de estados utilizada na unidade de handshake.

A sequência de estados ocorre da seguinte maneira:

- Quando o sinal de *reset* é acionado, o contador de ciclos é inicializado com o valor zero (estado 1).
- Quando o sinal *inputReady* é alto, ou seja, quando os dados de entrada estão prontos para serem utilizados, o contador de iterações inicia sua contagem, portanto *cycle* recebe o valor inteiro um (estado 2).
- Verifica-se então se o ciclo atual é igual ao número máximo de iterações ou se o sinal de *processingDone* é alto. Se pelo menos uma dessas condições for verdadeira (o que sempre ocorre nos blocos 3 e 4 do decodificador, que fazem seu processamento em apenas um ciclo de clock), o contador de iterações finaliza a contagem, e os dados de saída são disponibilizados (estado 4). Caso contrário, o contador de iterações é incrementado (estado 3).

- A máquina de estados permanece no estado 3 até que o número de iterações atinja seu valor máximo ou *processingDone* seja igual a '1'. Quando ocorrer pelo menos uma dessas condições, a máquina passa para o estado 4.
- A máquina de estados permanece no estado 4 até que o sinal de *outputRequested* seja igual a '1', ou seja, até que o bloco subsequente do decodificador solicite um valor de entrada.
- Quando *outputRequested* for '1', sabe-se que o bloco está pronto para processar uma nova entrada. Neste caso, o valor de *inputReady* determinará o próximo estado. Se *inputReady* for '1', o contador de iterações inicia novamente a contagem (estado 2). Caso contrário, passa para o estado 1, e espera até que novos dados estejam prontos para serem processados.

Um diagrama esquemático da unidade de *handshake* é mostrado na Figura 5.5. A máquina de estados do contador de iterações é construída utilizando-se multiplexadores, que seguem as condições mostradas na Figura 5.4. O contador é atualizado e registrado a cada ciclo de clock. As *flags* de sinalização são obtidas através de circuitos comparadores, e as saídas *inputRequested* e *OutputReady* através de simples portas lógicas AND e OR.

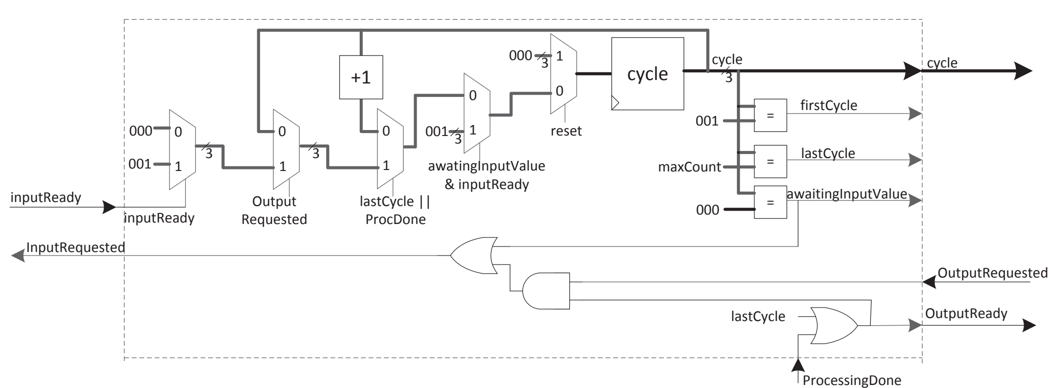


Figura 5.5 - Diagrama de circuito da unidade de *handshake*.

5.4 DETALHAMENTO DO BLOCO 1: DEMODULADOR/ORDENADOR DE ENTRADA

O bloco 1 recebe a palavra de entrada com os valores analógicos codificados em 3 bits e produz as sequências **r** (palavra demodulada por decisão abrupta) e **s** (posição dos

símbolos em ordem decrescente de confiabilidade). Um diagrama de alto nível para o bloco 1 do decodificador é apresentado na Figura 5.6.

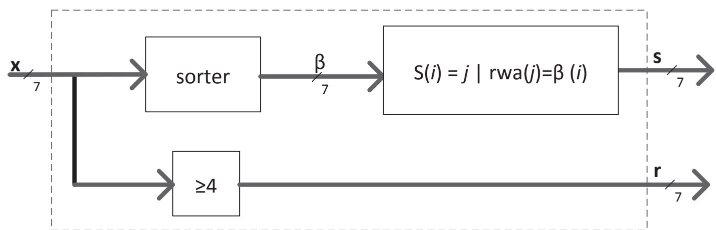


Figura 5.6 - Diagrama de blocos da demodulação/ordenação.

A demodulação e geração da sequência **r** são triviais, bastando observar o bit mais significativo do valor analógico recebido. Para uma codificação em 3 bits, valores maiores ou iguais a 4 são considerados positivos.

O vetor **s** é gerado por um ordenador linear (*linear insertion sorter*). Embora na Figura 5.6 a geração do vetor **s** esteja representada em um sub-bloco separado, na implementação adotada os valores são fornecidos pelo próprio *sorter,* através de *tags* associadas a cada símbolo de entrada.

A Figura 5.7 mostra os sub-blocos que compõem o bloco 1 do decodificador. A sequência de operações realizadas é descrita a seguir.

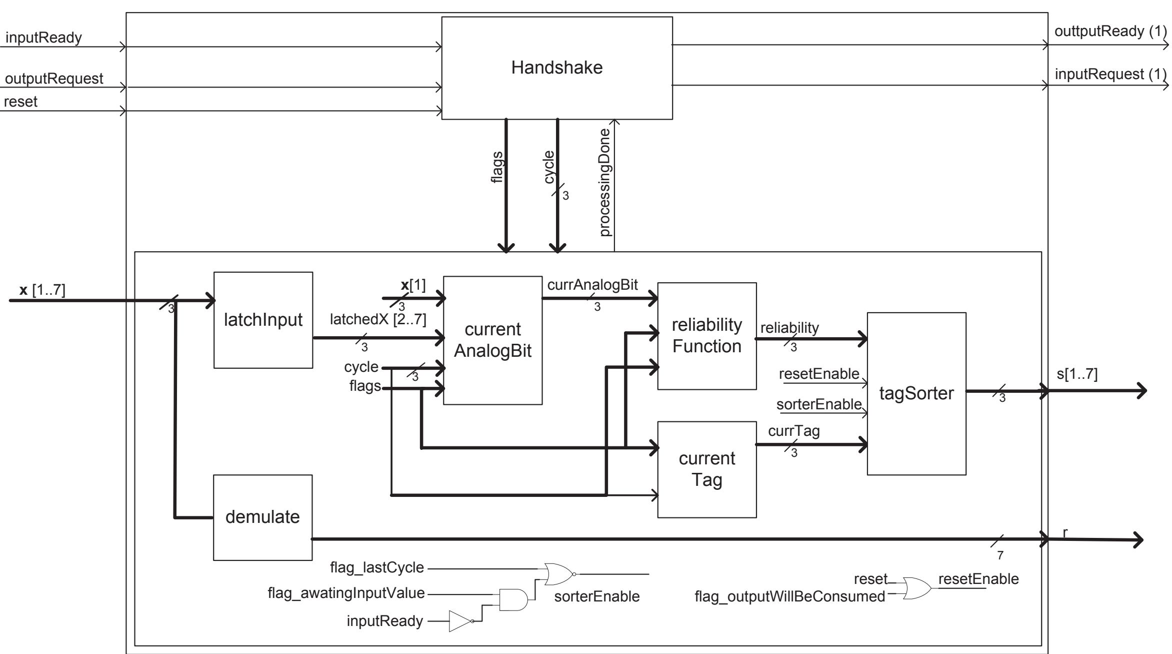


Figura 5.7 - Sub-blocos que compõem bloco de demodulação/ordenação de entrada.

Primeiramente, a sequência analógica recebida é registrada (sub-bloco *latchInput*). Cada um desses símbolos recebe uma *tag* de identificação, numerando os símbolos de 1 a 7 (número de bits da palavra recebida).

A cada ciclo de clock, um dos símbolos da sequência é analisado (sub-bloco *currentAnalogBit*), e sua confiabilidade é calculada. Para obter a confiabilidade do símbolo, utilizou-se a seguinte regra: o bit mais significativo de *currAnalogBit* identifica se o valor é positivo (1) ou negativo (0). Se for positivo, a confiabilidade é igual ao valor correspondente aos dois bits menos significativos. Caso contrário, recebe o valor dos dois bits menos significativos negados.

Os símbolos são ordenados de acordo com sua confiabilidade (sub-bloco *tagSorter*), produzindo o vetor de confiabilidade **s**. Este processo permitirá que o decodificador pondere a relevância de cada símbolo, aumentando o desempenho da decodificação.

O vetor **s** é uma lista de inteiros (3 bits) de 1 a *n*, indicado a posição de cada símbolo em **x** de acordo com sua confiabilidade. O primeiro valor em **s** indica a posição do símbolo mais confiável de **x**. Como os valores analógicos recebidos são normalizados na faixa de -1 a +1, quanto mais próximo o símbolo for de -1 ou +1 mais confiável ele é. Este vetor é gerado por um ordenado de inserção linear. Como os valores analógicos são ordenados realizando-se deslocamentos no ordenador, a saída do bloco 1 está disponível após *n* ciclos de clock.

O ordenador linear utilizado na implementação do bloco 1 é mostrado na Figura 5.8. Nela são vistos os detalhes internos do sub-bloco *tagSorter* (RIBAS et al., 2004) (DONG, WANG, 2009). Neste circuito, os valores de **x** são recebidos serialmente. Para cada símbolo de **x**, verifica-se a posição adequada para sua inserção, ou seja, à esquerda desta posição todos os valores são maiores que **x**, e á direita todos são menores que **x**. Escolhida a posição, todos os valores à direita da posição escolhida são deslocados para a direita. Portanto, a sequência β resultante estará ordenada de modo decrescente. Como a ordenação é uma das primeiras operações realizadas na palavra recebida, apresenta um impacto significante no *throughput* do decodificador.

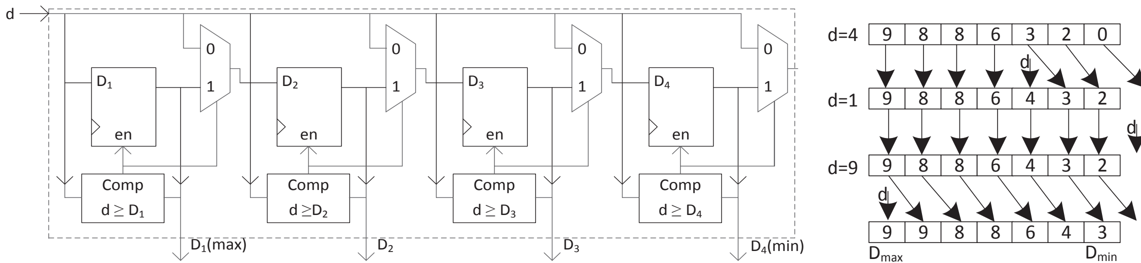


Figura 5.8 - Implementação em hardware do ordenador de inserção linear.

Outra tarefa desempenhada pelo bloco 1 é decodificação com decisão abrupta realizada a partir de **x**, denominada **r**. A operação é simples, e consiste apenas em replicar o bit mais significativo de cada símbolo, ou seja, o bit de sinal. Como mostrado na Figura 5.6 um simples comparador ≥ 4 é suficiente, visto que são utilizados 3 bits (e portanto 8 níveis possíveis) de quantização. Neste caso, valores de **x** na faixa de 0 a 3 são decodificados como 0 e valores na faixa de 4 a 7 são decodificados como 1.

5.5 DETALHAMENTO DO BLOCO 2: REDUÇÃO DE GAUSS-JORDAN MODIFICADA

O bloco 2 do decodificador (Figura 5.9) realiza a redução de Gauss-Jordan modificada. Este processo é iterativo e controlado por dois contadores. O contador de iterações incrementa a cada ciclo do clock, e é utilizado como um índice para a sequência **s** (iniciando pelo elemento mais confiável). O contador de eliminações incrementa quando uma eliminação ocorre com sucesso, ou seja, quando uma coluna LI é encontrada. Quando são obtidas *k* colunas LI, a eliminação é concluída. Os valores obtidos após cada eliminação são mantidos em uma matriz de trabalho (*workingMatrix* – **WM**), atualizada e registrada a cada iteração. No primeiro ciclo de processamento, a matriz de trabalho é inicializada com os valores da matriz **G**.

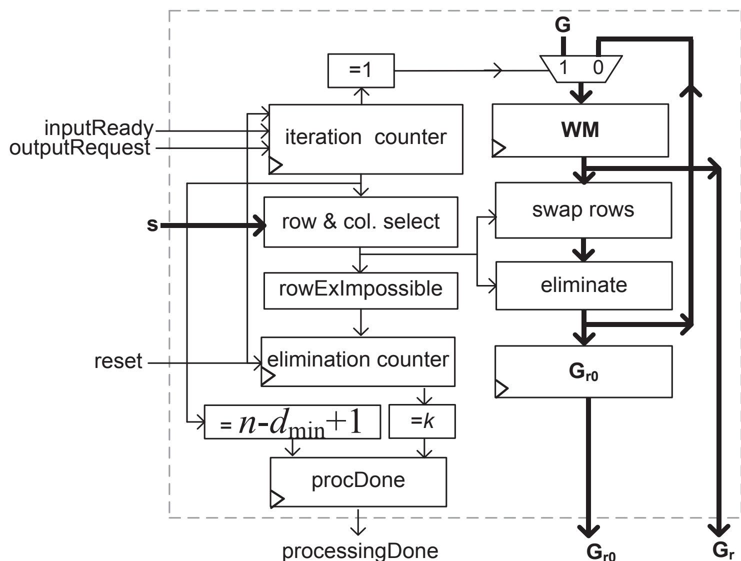


Figura 5.9 - Sub-blocos que compõem o bloco de redução de Gauss-Jordan modificada.

As operações *swap rows* e *eliminate,* que aparecem na figura, são operações elementares aplicáveis a matrizes binárias, puramente combinacionais. Essas operações são: (*i*) troca entre duas linhas para obter-se um pivô na posição desejada, e (*ii*) adição de duas linhas para reduzir uma coluna a um vetor unitário. Após as operações combinacionais, a matriz **WM** registra as atualizações. A matriz $\mathbf{G}_{r0}$ contém‘1’s nas posições dos pivôs e ‘0’s nas demais posições. Esta matriz será utilizada pelo bloco 3 do decodificador para extrair de **r** somente os bits correspondentes ao conjunto de informação. Ao término de todas as iterações, as saídas $\mathbf{G}_r$ (equivalente à matriz **WM**) e $\mathbf{G}_{r0}$ são disponibilizadas para os próximos blocos (blocos 3 e 4 da Figura 5.2).

A Figura 5.10 mostra os sub-blocos que compõem o bloco 2. Dois parâmetros são importantes neste bloco. O primeiro deles é o índice da linha atual (sub-bloco *row*). O processo de eliminação da matriz **G** inicia com o valor *row* = “0001” (apontando para a primeira linha de **G**). A variável *row* é incrementada (deslocada para a esquerda) cada vez que uma eliminação de coluna é concluída com sucesso, até que o processamento seja finalizado (*procesingDone* = ‘1’). O segundo parâmetro importante é o índice da coluna atual (*c*) (sub-bloco *column*), indicado pela sequência de confiabilidade **s**, que a cada iteração determina a coluna a ser reduzida.

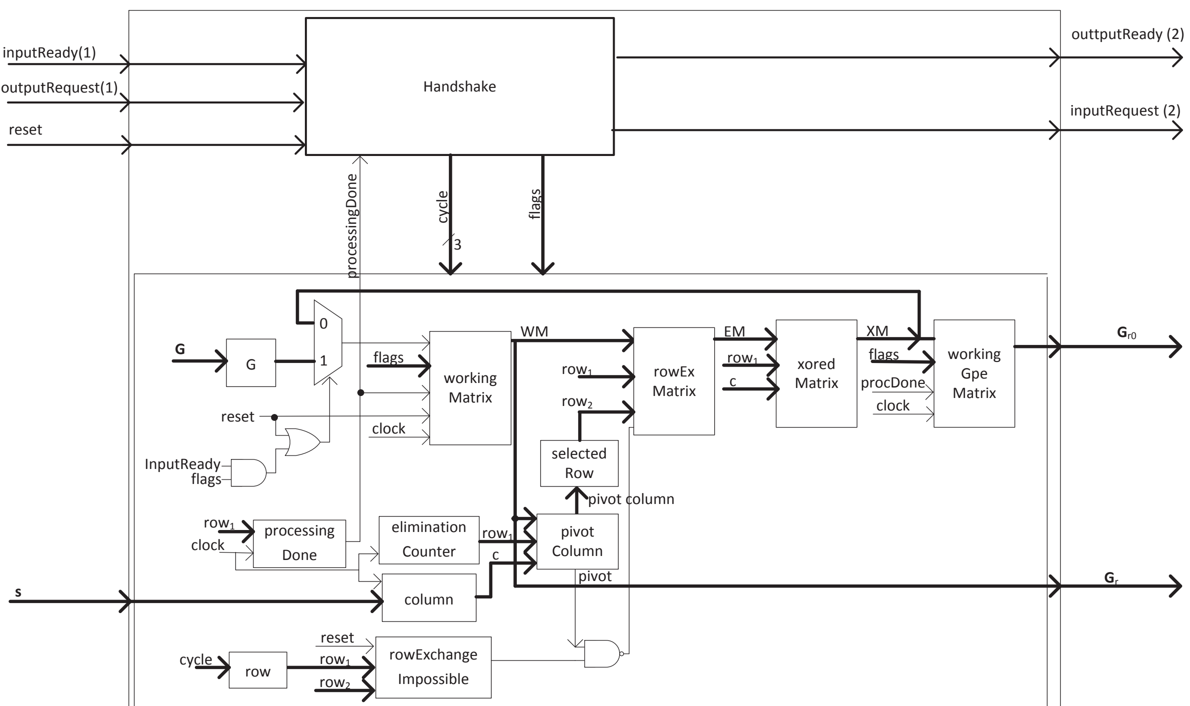


Figura 5.10 - Sub-blocos que compõem o bloco de eliminação de Gauss Jordan modificada.

A primeira operação na redução da **WM** (carregada inicialmente com os valores de **G**) é a seleção do pivô, indicado por *row* e *c.* Se o pivô for '0' então é necessário que a linha seja trocada por outra que com valor '1', na coluna correspondente. A operação de troca de linhas é indicada pelo sinal *rowExchangeNeeded*, como mostra a Figura 5.11. Repare que primeiramente a coluna indicada por **s** é selecionada, e em seguida o pivô é obtido de acordo com a linha atual. Se o pivô é '0' então *rowExchangeNeeded* é '1', sinalizando a necessidade de troca de linhas.

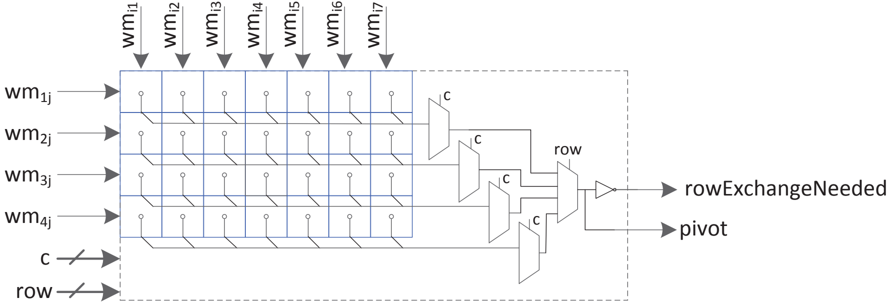


Figura 5.11 - Diagrama do sub-bloco RowExchangeNeeded.

A Figura 5.12 mostra o diagrama do sub-bloco *SelectedRow*, que determina a linha que será trocada pela linha atual (*i*). A primeira tentativa é trocar *i* por *i*+1. Entretanto, a troca só é possível se WM_{i+1j} for igual a '1'. Caso não seja, a próxima linha é verificada, até que um pivô válido seja encontrado.

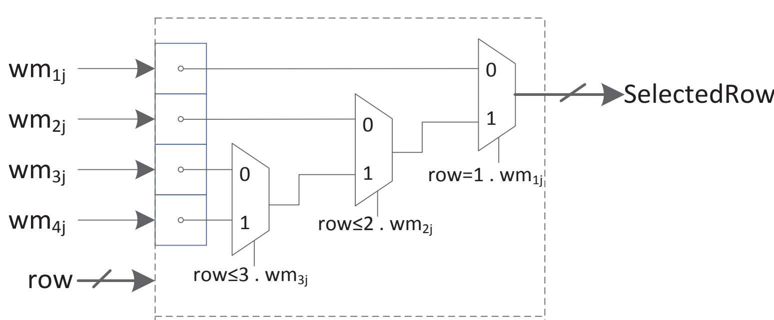


Figura 5.12 - Diagrama do sub-bloco SelectedRow.

É possível que não exista nenhum pivô valido. Esta condição é mostrada na Figura 5.13 (sub-bloco *RowExchangeImpossible*). Neste caso, todas as tentativas de selecionar uma linha, com valor '1' na coluna *j* falham. Se nenhum pivô válido é encontrado, é impossível encontrar um conjunto de informação, isto é, *k* colunas LI.

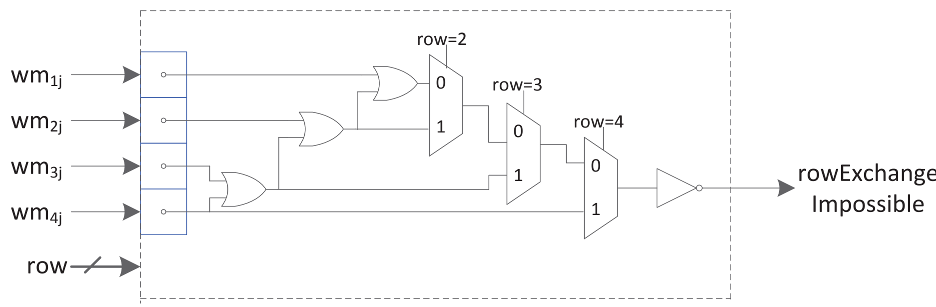


Figura 5.13 - Diagrama do sub-bloco rowExchangeImpossible.

Conhecidas i_1 e i_2 (a linha atual e a linha selecionada para a troca), o próximo passo é efetuar a troca, quando possível. A Figura 5.14 mostra a lógica empregada para a troca de um elemento da nova matriz chamada de *rowExchangedMatrix* (**EM**), uma versão puramente combinacional da **WM**, com linhas trocadas.

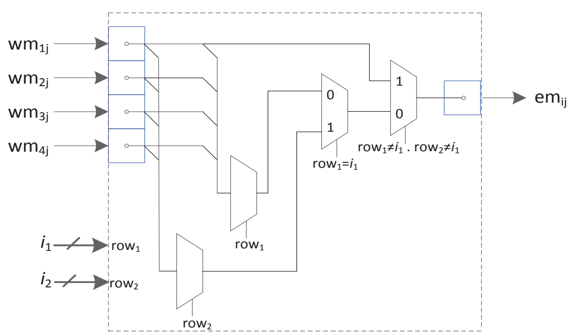


Figura 5.14 - Diagrama do sub-bloco de troca de linhas.

Nesta operação é verificado se a linha atual está envolvida na troca, caso a linha não faça parte da troca, isto é, $row_1 \neq i_1$ e $row_2 \neq i_2$, a posição em_{ij} recebe o mesmo valor que wm_{ij}, ou seja, as linhas não são trocadas. Se $row_1=i_1$, então em_{ij} recebe wm_{row2j}, senão $em_{ij} \leftarrow wm_{row1j}$.

Após a troca de linhas, é necessário eliminar os elementos com valor '1' acima e abaixo do pivô, para transformar a coluna em um vetor de peso unitário. Esta nova matriz, chamada de *xoredMatrix* (**XM**), é uma versão puramente combinacional da **EM**.

A Figura 5.15 mostra o diagrama de blocos para a eliminação de uma coluna da matriz **EM**. O índice *row* está apontando para a linha do pivô. Então para obter a posição xm_{1j}, por exemplo, verifica-se se *row* = '1' (neste caso m_{1j} seria o próprio pivô) ou se $em_{1j} = 0$. Nestes dois casos nenhuma operação é realizada e xm_{1j} recebe em_{1j}. Caso contrário, é feita uma operação XOR entre em_{1j} e o pivô. Após a redução da coluna, a matriz **WM** é atualizada.

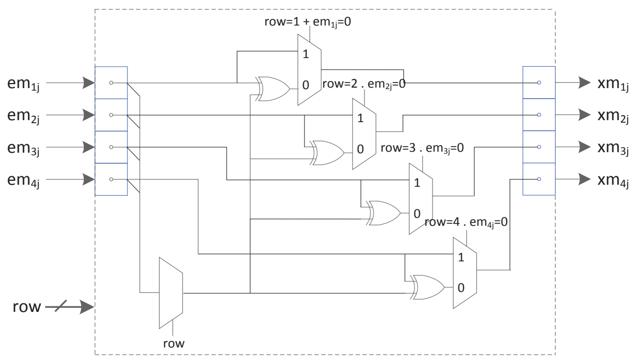


Figura 5.15 - Diagrama do sub-bloco de eliminação de coluna.

O processo de eliminação é executado até que a matriz **WM** esteja completamente escalonada. Essa condição é indicada pelo sinal *processingDone*. O sinal *processingDone* (inicializado com valor '0', pelo reset), tem seu valor alterado quando o índice de linhas apontar para a última linha da matriz **WM** (se não houver nenhuma falha na troca dessa linha).

A versão final da **WM** (completamente escalonada) é a matriz $\mathbf{G}_r$ citada no algoritmo da seção 5.4.1. A matriz $\mathbf{G}_{r0}$ é obtida a partir de **WM**. A Figura 5.16 mostra como um elemento da matriz $\mathbf{G}_{r0}$ é obtido. Inicialmente essa matriz contém apenas zeros. Durante o processo de redução, $\mathbf{G}_{r0}$ recebe '1' nas posições dos pivôs das colunas reduzidas da **WM**.

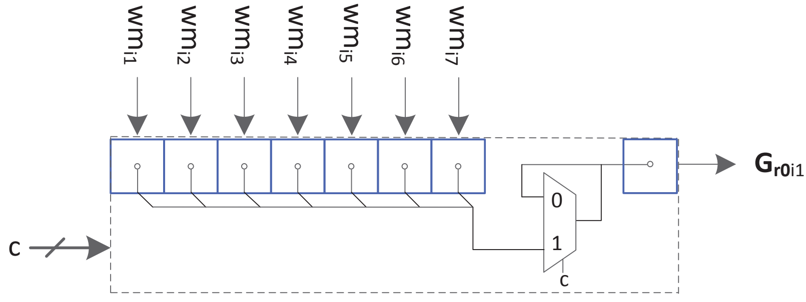


Figura 5.16 - Diagrama de blocos da matriz $\mathbf{G}_{r0}$.

5.6 DETALHAMENTO DO BLOCO 3: GERADOR DE MENSAGENS CANDIDATAS

O bloco 3 do decodificador, apresentado na Figura 5.17, gera um conjunto de mensagens candidatas, que serão posteriormente recodificadas e avaliadas pelos blocos subsequêntes. Primeiramente, a palavra **r** (resultado da decisão abrupta da palavra recebida) é

multiplicada pela matriz $\mathbf{G}_{r0}^{T}$, gerando $\mathbf{u}_0$ (a mensagem candidata original). Em seguida, k mensagens candidatas são geradas, invertendo-se um bit de $\mathbf{u}_0$ por vez.

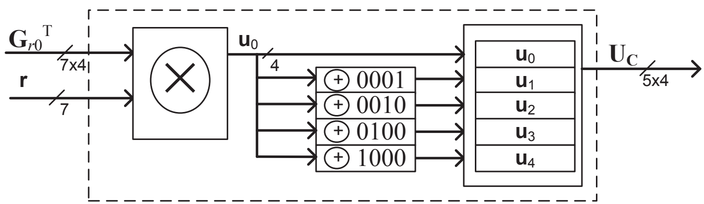


Figura 5.17 - Diagrama de blocos do gerador de mensagens candidatas.

A Figura 5.18 apresenta o diagrama do bloco 3 e seus sub-blocos. Veja que inicialmente **r** e $\mathbf{G}_{r0}$ são registrados, e em seguida a operação de multiplicação é realizada. Finalmente, as demais mensagens candidatas são geradas. As operações de multiplicação e geração das k candidatas são puramente combinacionais.

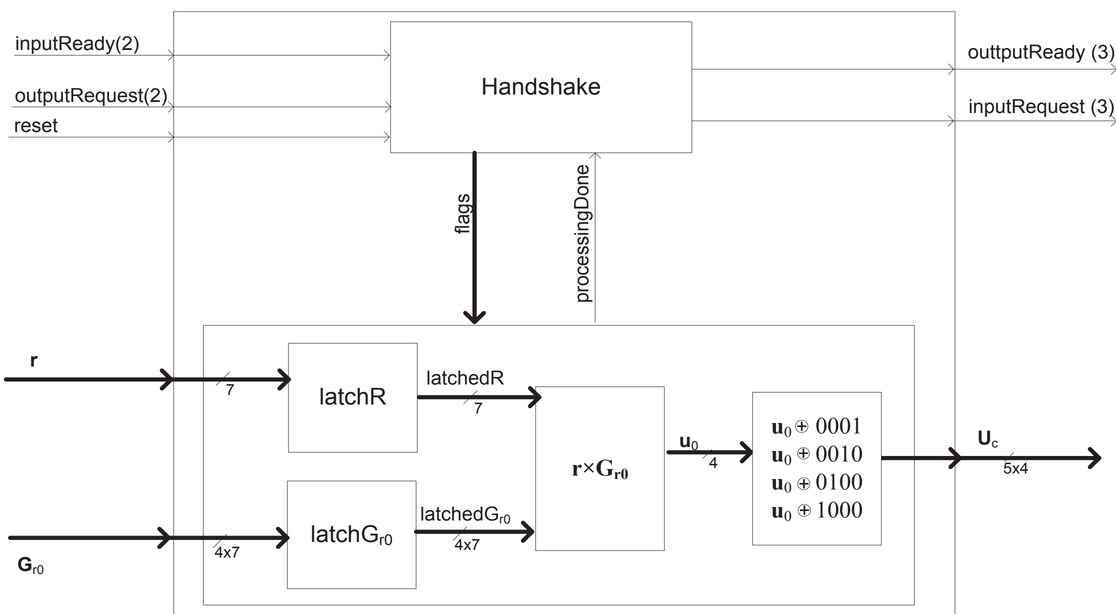


Figura 5.18 - Sub-blocos que compõem bloco de geração das palavras candidatas.

A Figura 5.19 mostra um diagrama detalhado dos sub-blocos de multiplicação e de geração das mensagens candidatas não originais. A parte esquerda da figura, correspondente ao sub-bloco $r\times G_{r0}$, mostra o circuito de geração do primeiro bit de $\mathbf{u}_0$. A parte direita da figura mostra a geração das demais candidatas. A mensagem candidata $\mathbf{u}_0$, resultado da multiplicação entre a palavra **r** e as colunas da matriz $\mathbf{G}_{r0}^{T}$, é obtida utilizando-se portas lógicas AND e XOR. As demais k mensagens candidatas são produzidas invertendo-se um bit

por vez de $\mathbf{u}_0$. Por exemplo, para um código com 4 bits de informação (k=4), o padrão de inversão de bits seria 0001, 0010, 0010, 1000. Em hardware, essa operação é equivalente a fazer a operação XOR bit a bit entre $\mathbf{u}_0$ e o padrão de inversão. No total k+1 candidatas serão avaliadas pelo bloco subsequente, pois a mensagem original, sem *bit-flipping* também é considerada.

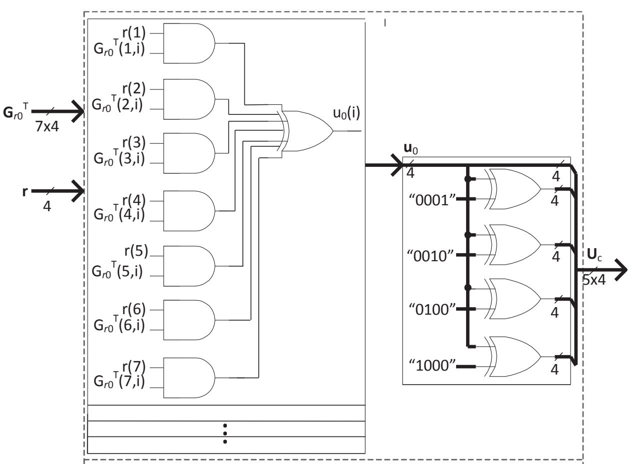


Figura 5.19 - Diagrama do circuito gerador de mensagens candidatas.

5.7 DETALHAMENTO DO BLOCO 4: GERADOR DE PALAVRAS-CÓDIGO CANDIDATAS

O bloco 4 produz uma palavra-código $\mathbf{c}_j$ para cada mensagem candidata $\mathbf{u}_j$ gerada no bloco 3 (Figura 5.20). Esta operação consiste em recodificar cada mensagem usando a matriz geradora reduzida $\mathbf{G}_r$.

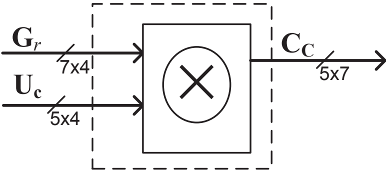


Figura 5.20 - Diagrama de blocos do gerador de palavras-código candidatas.

Assim como no bloco 3, a operação realizada é a multiplicação matricial, desta vez entre $\mathbf{U_c}$ (matriz das mensagens candidatas) e $\mathbf{G}_r$. Esta operação é totalmente combinacional, isto é, a saída está disponível no próximo ciclo de clock.

Os sub-blocos contidos no bloco 4 são mostrados na Figura 5.21. Primeiramente, as matrizes são registradas (sub-blocos latchU_c e latchG_r) e em seguida a multiplicação é efetuada. O circuito de cada uma das células da matriz $\mathbf{C_c}$, correspondente ao sub-bloco $U_c \times G_r$, é apresentado na Figura 5.22. Este circuito é composto de portas lógicas AND e XOR.

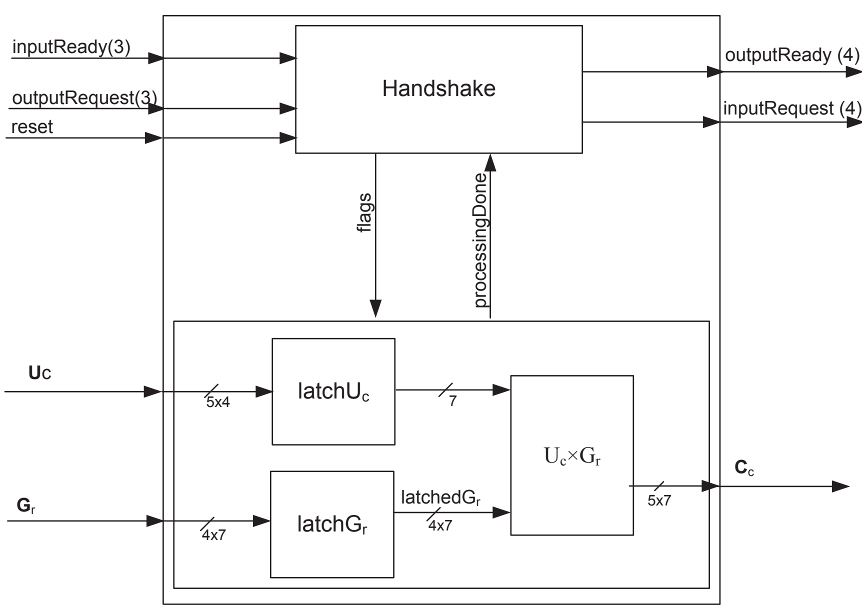


Figura 5.21 - Sub-blocos que compõem bloco de geração das palavras-código candidatas.

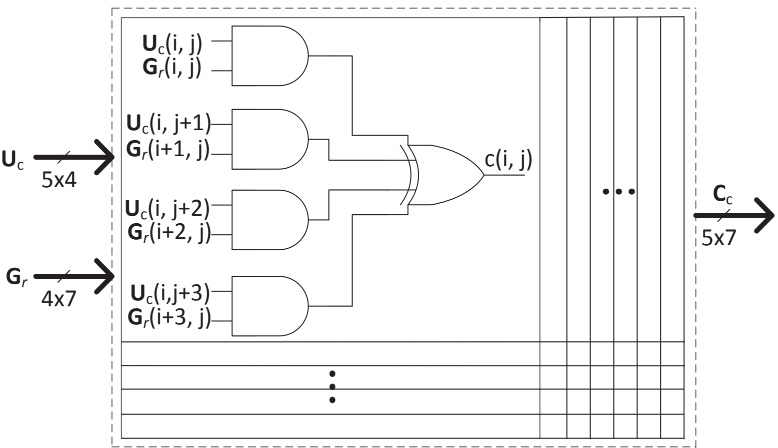


Figura 5.22 - Diagrama de circuito do gerador de palavras-código candidatas.

5.8 DETALHAMENTO DO BLOCO 5: SELETOR DA PALAVRA-CÓDIGO VENCEDORA

O bloco 5 é responsável pela última etapa da decodificação, ou seja, escolher a palavra-código vencedora verificando qual delas mais se assemelha à palavra analógica recebida (Figura 5.23). Esta escolha é baseada na distância suave entre a palavra analógica (**x**) e cada uma das palavras-código candidatas ($\mathbf{C}_{cj}$). A candidata com a menor distância é escolhida como vencedora. Para cada palavra-código, a distância é calculada pelas Equações 5.1 e 5.2. Na Equação 5.1 são calculadas as distâncias individuais entre cada símbolo de **x**, quantizado em três bits, e $\mathbf{c}_j$. Se $\mathbf{c}_j$ tem valor 1, então a distância suave é 7-**x**, senão a distância suave é o próprio valor de **x**. A distância suave total é o somatório das individuais (Equação 5.2). Após calcular esta métrica para cada uma das palavras-código candidatas, a candidata com a menor distância é escolhida como saída do decodificador. Este processo é iterativo e cada palavra candidata é analisada em um ciclo de clock. Portanto, *k*+1 ciclos de clock são utilizados no processamento.

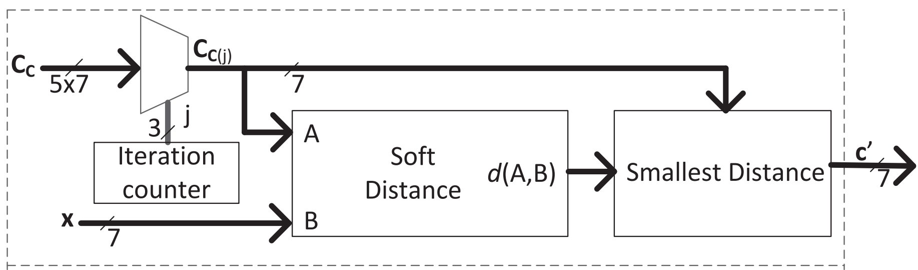


Figura 5.23 - Diagrama de blocos do seletor de palavra-código vencedora.

A Figura 5.24 apresenta os sub-blocos do seletor da palavra-código vencedora. Os sub-blocos *latchX* e *latchC*$_c$ registram a palavra analógica recebida e a matriz de palavras-código candidatas, respectivamente. A cada ciclo de clock, a distância suave entre a palavra analógica (sub-bloco *analogWord*) e uma palavra-código (sub-bloco *currentCandidateWord*) é calculada pelo sub-bloco *softDistanceCalculation*, e o valor de menor distância é atualizado.

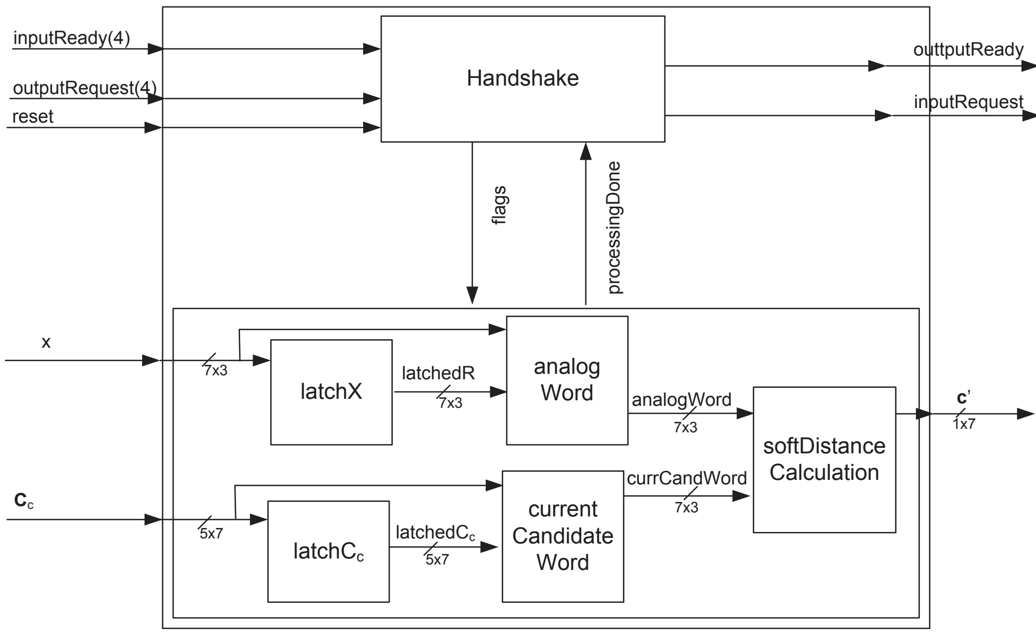


Figura 5.24 - Sub-blocos que compõem o bloco de seleção da palavra-código vencedora.

A Figura 5.25 mostra os detalhes internos do sub-bloco *softDistanceCalculation.* Nesta figura, é possível notar que para cada palavra-código candidata é calculada a distância individual de seus bits em relação à palavra analógica **x**. Em seguida, todas as distâncias individuais são somadas. A distância calculada é comparada com a menor distância já armazenada, e se necessários os registradores contendo o menor valor e a palavra código correspondente são atualizados.

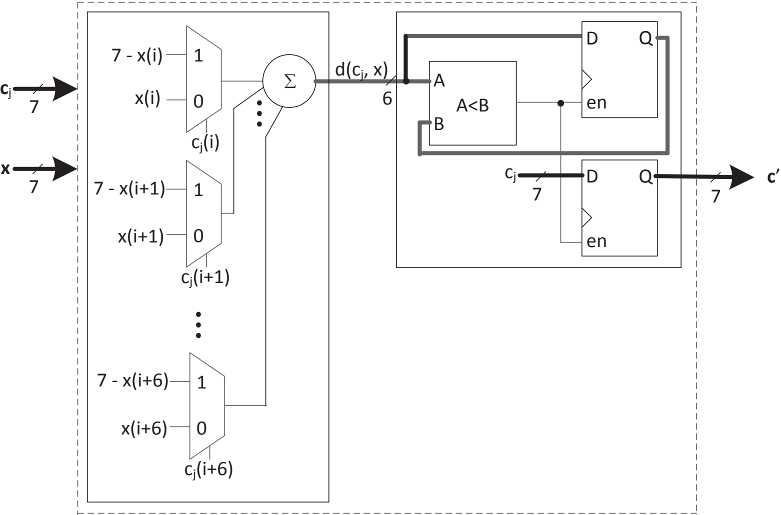


Figura 5.25 - Diagrama detalhado do seletor da palavra-código vencedora.

CAPÍTULO 6

IMPLEMENTAÇÃO DO HARDWARE EM FPGA

6.1 INTRODUÇÃO

Três implementações distintas foram realizadas para o decodificador de k+1 candidatas. A primeira, em FPGA, foi uma implementação preliminar, utilizada como base para as demais implementações. A segunda, foi uma implementação manual do hardware em ASIC. A terceira foi uma implementação automatizada do hardware em ASIC utilizando VHDL.

A implementação preliminar, em FPGA, foi realizada com os seguintes objetivos:

1) Obter uma descrição em VHDL para o decodificador, a qual também foi utilizada para a elaboração de um layout através do pacote de ferramentas da Cadence;
2) Testar exaustivamente (com todos os valores de entrada possíveis) o funcionamento do circuito decodificador, através de *testbenches* em VHDL;
3) Obter uma estimativa da quantidade de recursos lógicos utilizada por cada bloco do decodificador;
4) Obter um valor de referência para a frequência máxima de operação do circuito, quando implementado em FPGAs de baixo custo ou de alto desempenho.

Um resumo dos procedimentos adotados para a implementação em FPGA é descrito a seguir, seguido pelos principais resultados obtidos.

6.2 DESCRIÇÃO DO PROCEDIMENTO DE PROJETO EM VHDL/FPGA

Partindo do diagrama apresentado na Figura 5.2, uma entidade foi criada para cada um dos cinco blocos principais, além da unidade de handshake. Cada entidade foi testada através de um testbench específico, o qual consiste em um arquivo de código responsável por

exercitar as funcionalidades da entidade sob teste. Todas as simulações foram realizadas usando o compilador e simulador ModelSim, da Mentor Graphics.

Todo o código VHDL foi escrito com o cuidado de criar uma implementação totalmente genérica, na qual todos os parâmetros configuráveis podem ser ajustados (por exemplo, a matriz geradora do código (**G**) e o número de bits dos valores analógicos). Assim, foi possível realizar uma análise detalhada da área utilizada pelo circuito, em função do tamanho do código utilizado. Todos os parâmetros configuráveis foram agrupados em um único *package* (chamado *decoder_config*), evitando que seja necessário modificar vários locais no código para se configurar o circuito gerado.

Para realizar operações com matrizes em GF (2), foram criados um tipo específico (*bit_matrix*) e uma biblioteca (*bit_matrix_pkg*) contendo as operações necessárias (e.g., multiplicação, deslocamento, substituição de linhas e colunas, etc.). A biblioteca também foi testada através de *testbenches* desenvolvidos com esta finalidade.

Após a implementação e teste dos blocos individuais, foi criada uma entidade de nível superior, a qual instancia todos os blocos do circuito decodificador. Essa entidade também possui um *testbench* específico, o qual apresenta palavras na entrada do decodificador e compara a saída correspondente com os valores em um arquivo de referência. Esse arquivo é gerado usando o software MATLAB e uma implementação em alto nível do algoritmo do decodificador. É importante ressaltar que o arquivo de referência contém todas as entradas e saídas possíveis para uma implementação usando o código C(7,4) e representação em 3 bits para valores analógicos (um total de 2^{21} combinações), garantindo assim a correção de implementação para esse circuito.

As Figuras 7.1 a 7.6 mostram os circuitos gerados automaticamente pelo compilador Quartus II a partir do código VHDL. Por serem gerados automaticamente, nem todos os itens das figuras são legíveis. Entretanto, o principal objetivo dessas figuras não é a compreensão dos circuitos gerados e sim passar uma noção do tamanho e da complexidade relativa de cada bloco quando implementado em uma FPGA.

A Figura 6.1 mostra o resultado da compilação da entidade de nível superior do decodificador, na ferramenta RTL Viewer (parte do software Quartus II). As Figuras 7.2 a 7.6 mostram a implementação dos blocos 1 a 5, respectivamente[1].

[1] A FPGA utilizada para os testes práticos foi a Cyclone III EP3C16F484, presente no kit DE0.

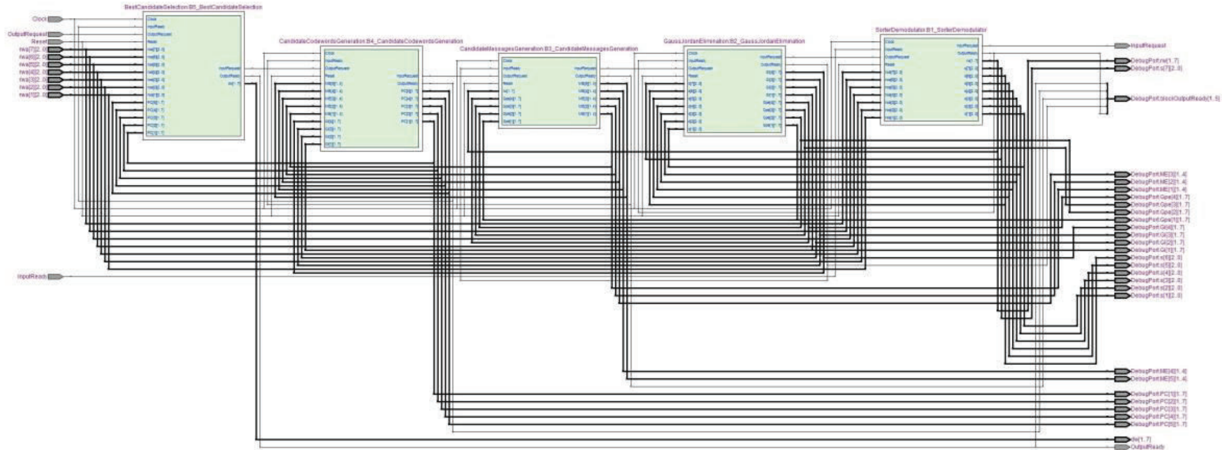


Figura 6.1 - Entidade *top level* do decodificador.

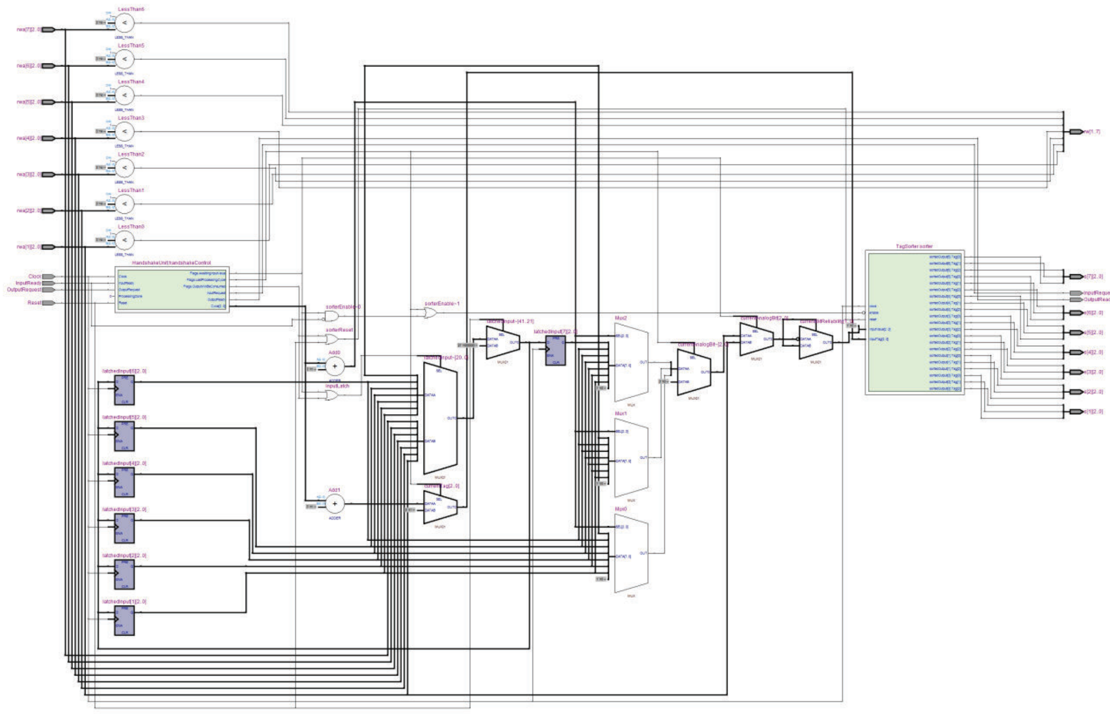


Figura 6.2 - Bloco 1: Modulador/Ordenador.

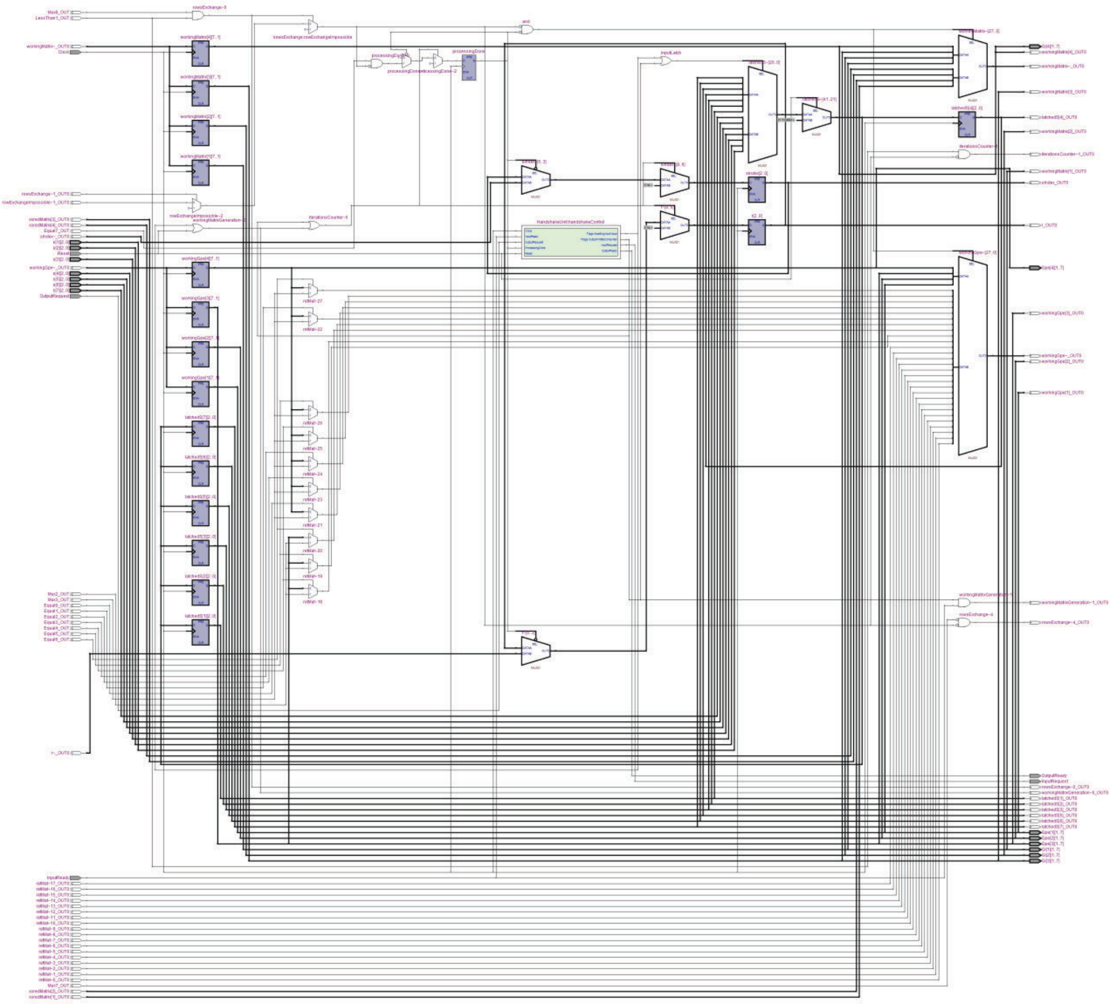


Figura 6.3 - Bloco 2: Redução de Gauss-Jordan modificada.

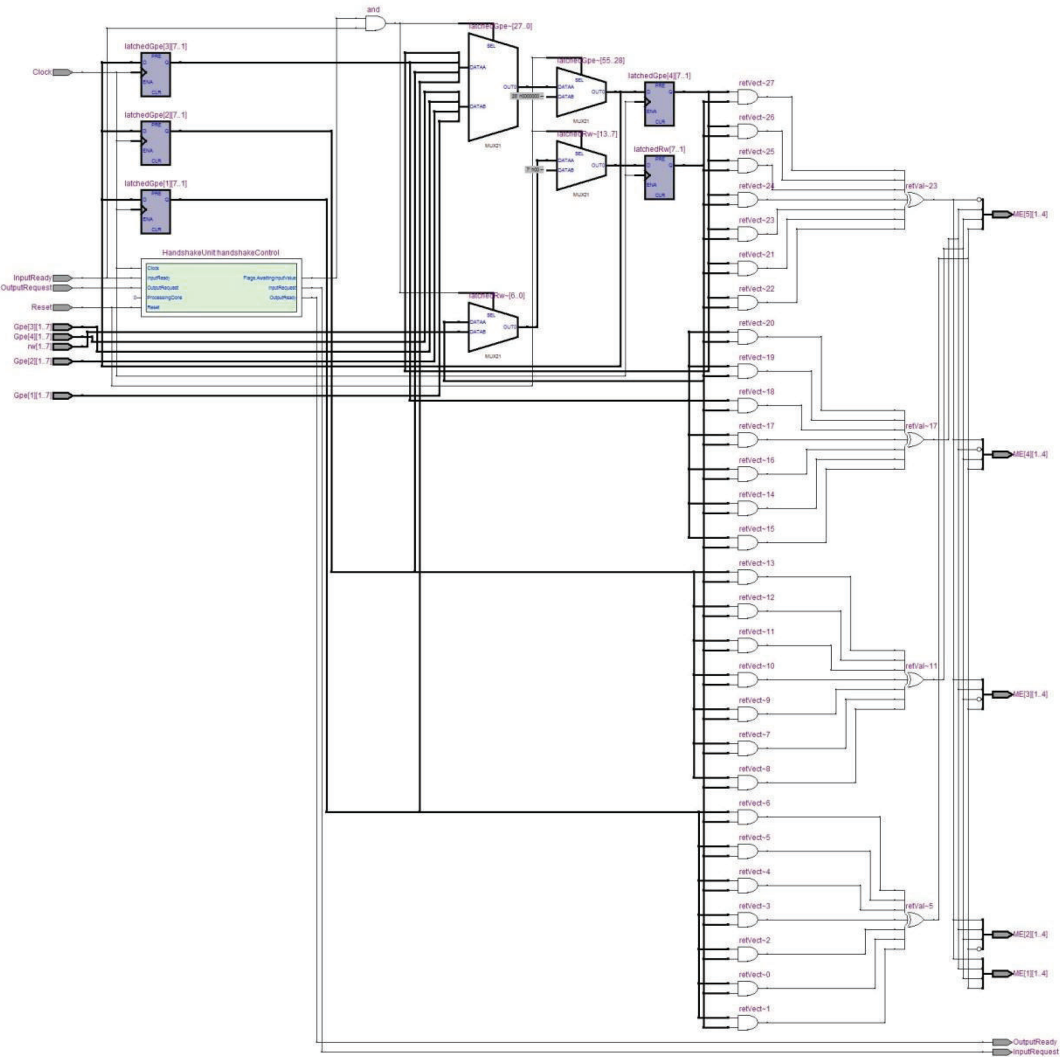


Figura 6.4 - Bloco 3: Geração das mensagens candidatas.

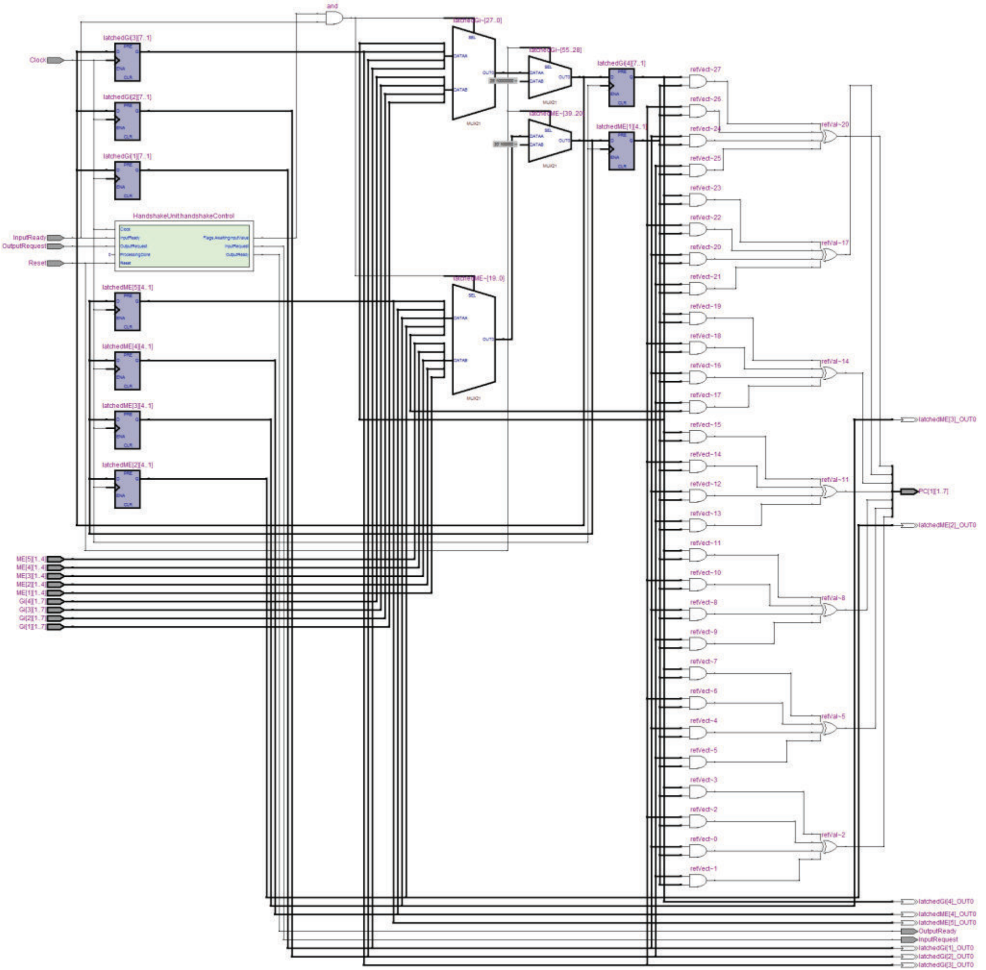


Figura 6.5 - Bloco 4: Geração das palavras-código candidatas.

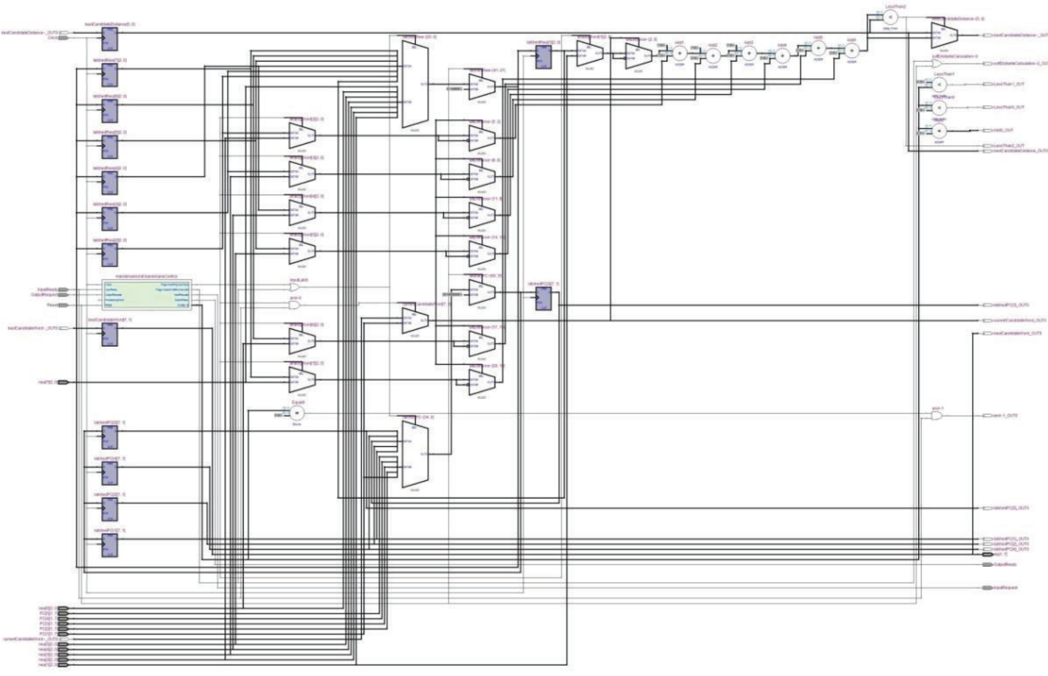


Figura 6.6 - Bloco 5: Seleção da candidata vencedora.

6.3 PRINCIPAIS RESULTADOS OBTIDOS EM FPGA

O código VHDL foi sintetizado para uma FPGA de alto desempenho da família Stratix III (Altera EP3SL70F780C2), para códigos de diversos tamanhos. Os resultados da síntese são apresentados na Tabela 6.1.

Tabela 6.1 - Resultados da Implementação em FPGA (Stratix III)

Código	Flip-flops	ALUTs	f_{MAX} (MHz)	Latência (ciclos)	tbo_{MAX} (ciclos)	Throughput (Mbps)
C(7,4,3)	291	443	159.1	19	5	127.3
C(15,7,5)	838	1,214	111.1	36	11	70.7
C(24,12,8)	1,954	3,273	84.2	56	17	59.4
C(48,24,12)	6,808	10,295	55.5	112	37	36.0
C(66,33,12)	12,387	17,933	50.1	157	55	30.0
C(78,39,14)	16,972	26,639	41.4	185	65	24.8

Como esperado, a quantidade de recursos lógicos é maior para os códigos longos, enquanto a frequência máxima de operação decresce com o aumento de tamanho do circuito. A tabela também mostra os valores de latência (tempo para decodificar a primeira palavra) e o tempo máximo entre valores de saída (tbo_{MAX}) produzidos pelo decodificador. Esses valores de temporização influenciam na velocidade total dos dados fornecidos pelo decodificador (*throughput*).

CAPÍTULO 7

IMPLEMENTAÇÃO MANUAL DO HARDWARE EM ASIC E RESULTADOS

7.1 INTRODUÇÃO

Este capítulo descreve a segunda implementação realizada para o decodificador, uma implementação manual do hardware em ASIC, utilizado um código C(7, 4, 3). São apresentados os detalhes das unidades do decodificador, com layout em nível de transistor. As células utilizadas nesta implementação foram construídas considerando-se aspectos de otimização, como tamanho mínimo e velocidade máxima. Todas as células foram simuladas utilizando a ferramenta PSPICE, onde foram realizados testes de verificação funcional e balanceamento temporal. Nesta implementação, cada um dos cinco blocos do decodificador (Figura 5.2) foi projetado, verificado e fabricado em um chip individual. Finalmente, os blocos foram conectados e o projeto final, contendo o decodificador completo, foi fabricado. A Tabela 7.1 mostra um resumo de todos os chips fabricados nesta implementação.

7.2 CONSTRUÇÃO DA BIBLIOTECA DE CÉLULAS-PADRÃO

Para possibilitar a implementação VLSI descrita neste capítulo, foi desenvolvida uma biblioteca de células-padrão. Esta biblioteca é empregada posteriormente na construção dos blocos do decodificador. Para tal, os passos descritos abaixo foram seguidos.

Passo 1: Escolha da tecnologia

A tecnologia utilizada foi a CMOS On Semi C5N SCM3M_SUBM 0.5μ (5 V de alimentação). A principal motivação na escolha desta tecnologia foi a possibilidade de fabricar o chip através do programa de pesquisa disponibilizado pela MOSIS, sem nenhum custo para a universidade. Anualmente, a MOSIS permite que um projeto de até 16 mm^2 seja fabricado na tecnologia On Semi 0.5 μm ou IBM 8FR 130 nm. A tecnologia de 130 nm não

foi selecionada devido a dificuldades na utilização das células-padrão para a criação do layout. Esses problemas impossibilitaram a fabricação do chip, pois algumas regras de design (DRC) não foram atendidas.

Tabela 7.1 - Resumo dos chips fabricados no projeto descrito nesta tese.

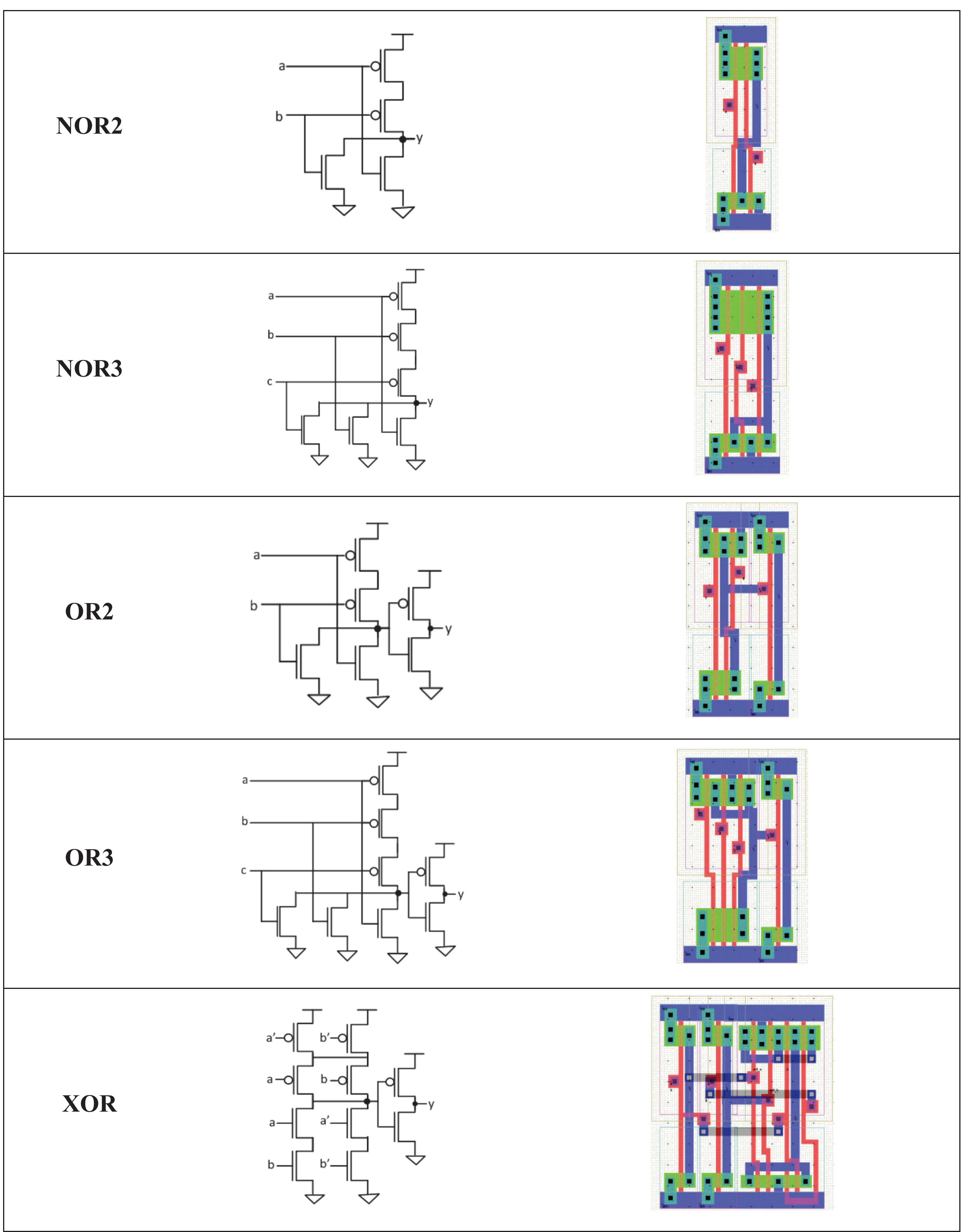
Chip 1	Conteúdo: Bloco 1 - Demodulador/Ordenador de entrada; Bloco 3 - Gerador de mensagens candidatas Tecnologia: CMOS 0.5 μm Fabricante: ON Semi; Processo: C5F	
Chip 2	Conteúdo: Bloco 2 - Redução de Gauss-Jordan Modificada Tecnologia: CMOS 0.5 μm Fabricante: ON Semi; Processo: C5F	
Chip 3	Conteúdo: Bloco 4 - Gerador de palavras-código candidatas Tecnologia: CMOS 0.5 μm Fabricante: ON Semi; Processo: C5F	
Chip 4	Conteúdo: Bloco 5 – Seletor de palavra-código vencedora Tecnologia: CMOS 0.5 μm Fabricante: ON Semi; Processo: C5F	
Chip 5	Conteúdo: Decodificador completo Tecnologia: CMOS 0.5 μm Fabricante: ON Semi; Processo: C5F	

Passo 2: Escolha dos circuitos

Cada circuito foi implementado em baixo nível, utilizando diretamente transistores. Esta foi uma etapa importante, pois se exercitou algumas das decisões que seriam necessárias adiante. Por exemplo, optou-se pela lógica CMOS, por ser estática e robusta a ruídos. Outro exemplo foi a escolha do flip-flop. Conforme visto no capítulo 2, há muitas configurações disponíveis, cada qual com suas vantagens e desvantagens. Neste ponto, optou-se pelo flip-flop TG- C^2MOS (Figura A.24(d), anexo I), pois ele é simples, robusto, compacto e eficiente em termos de energia. Outro exemplo foi o multiplexador. Optou-se pela configuração vista na Figura A.34(c) (anexo I), por ser o circuito mais compacto possível para um multiplexador sem ‘0’s ou ‘1’s pobres (apenas seis transistores), com a desvantagem de não ser buferizado. Já para o comparador, optou-se pela versão da Figura A.32(a) (anexo I).

Em resumo, as configurações escolhidas para as células-padrão constam na Figura 7.1.

Célula	Esquemático	Layout
NOT		
NAND2		
NAND3		
AND2		
AND3		


NOR2
a
b
y
NOR3
a
b
c
y
OR2
a
b
y
OR3
a
b
c
y
XOR
a'
b'
a
b
a
a'
b
b'
y

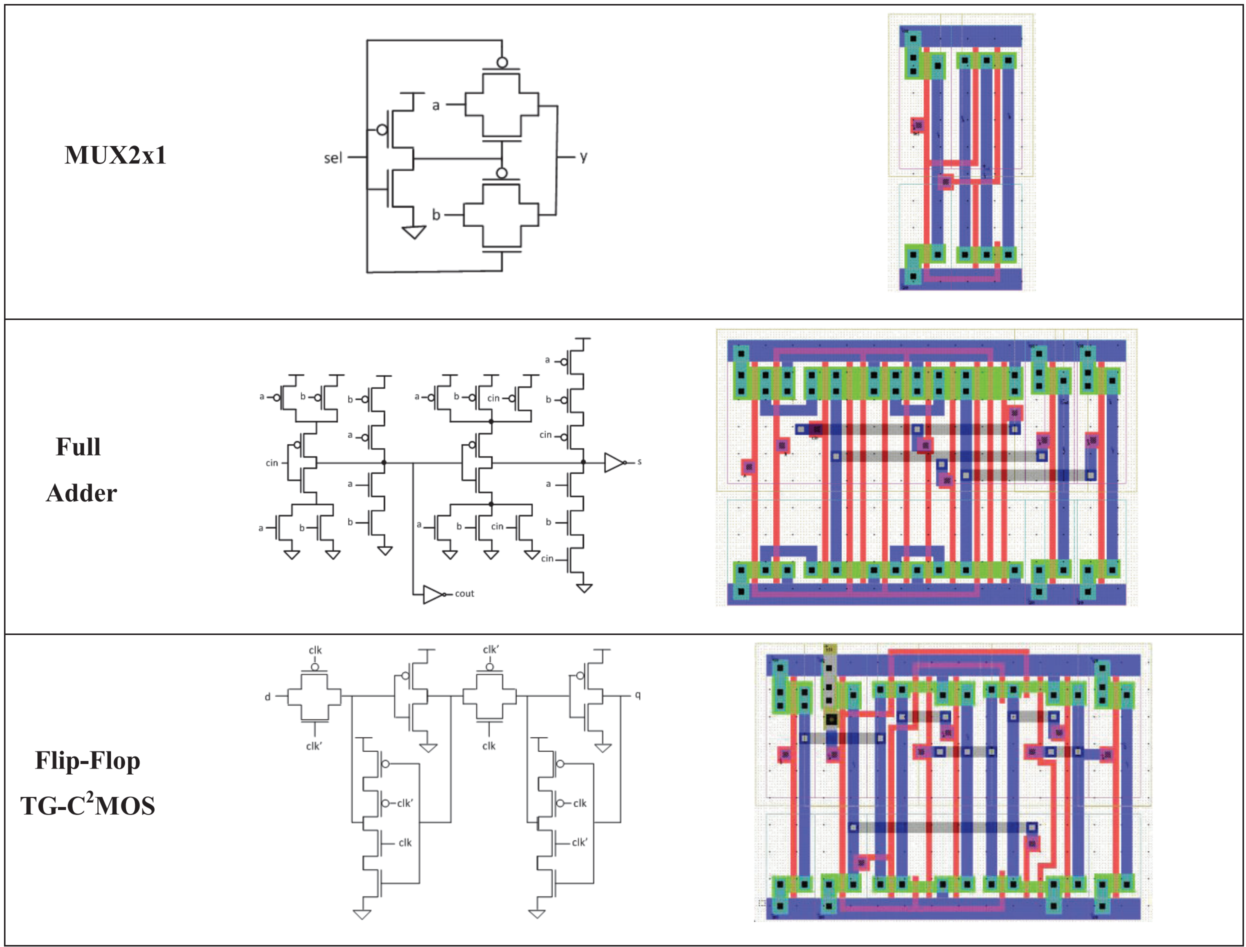


Figura 7.1 - Configurações escolhidas para as células-padrão e seus respectivos layouts.

Passo 3: Código e simulação SPICE

Em seguida, foi escrito o código SPICE para cada um dos circuitos escolhidos, seguido de simulações com o PSPICE, a fim de verificar o funcionamento adequado dos circuitos e também o balanceamento da resposta temporal. Com isso, obteve-se uma biblioteca de códigos SPICE.

Passo 4: Layout e DRC (Design Rule Check)

O próximo passo foi a criação do layout e o roteamento de cada um dos circuitos listados. Obviamente, a ferramenta de DRC foi aplicada continuamente durante a execução dos layouts. Assim como feito para o código SPICE, uma biblioteca de células-padrão de layout foi construída (Figura 7.1). A ferramenta utilizada foi o Ledit, da Tanner.

Passo 5: Layout versus Schematic (LVS)

O último passo constou da verificação de todos os layouts versus esquemáticos, utilizando a ferramenta LVS. De cada layout foi extraído um código SPICE equivalente, o qual foi comparado àquele escrito no passo 1, a fim de garantir a inexistência de erros. A ferramenta utilizada foi o LVS da Tanner.

7.3 CIRCUITO DE HANDSHAKE

O bloco de *handshake*, apresentado na seção 6.3, é responsável pelo controle entre os blocos da decodificação. Ele garante que um bloco nunca forneça sua saída sem que o próximo bloco esteja pronto para recebê-la, e vice-versa.

O primeiro passo na implementação da unidade de *handshake* (Figura 5.5) foi a criação dos circuitos esquemáticos. Esta tarefa foi realizada utilizando-se blocos esquemáticos na ferramenta Quartus II (diferente da primeira implementação, descrita no capítulo 7, que utilizou linguagem de descrição de hardware). Estes circuitos esquemáticos foram utilizados posteriormente para a criação dos circuitos em nível mais baixo, isto é, em nível de transistor.

Os esquemáticos em nível de transistor são necessários para que o circuito seja implementado em linguagem SPICE e também para a criação do layout. Os resultados das simulações do código SPICE permitem a análise detalhada do circuito, sendo possível verificar tensão, corrente, tempo de subida/descida, etc., em cada ponto de interesse.

Após as simulações realizadas na ferramenta PSPICE, o layout foi criado utilizando a ferramenta Ledit da Tanner. A Figura 7.2 mostra o layout da unidade de *handshake*. No design, são utilizadas as células da biblioteca padrão (Figura 7.1). Simulações SPICE pós-layout, incluindo capacitâncias parasitas, também foram executadas.

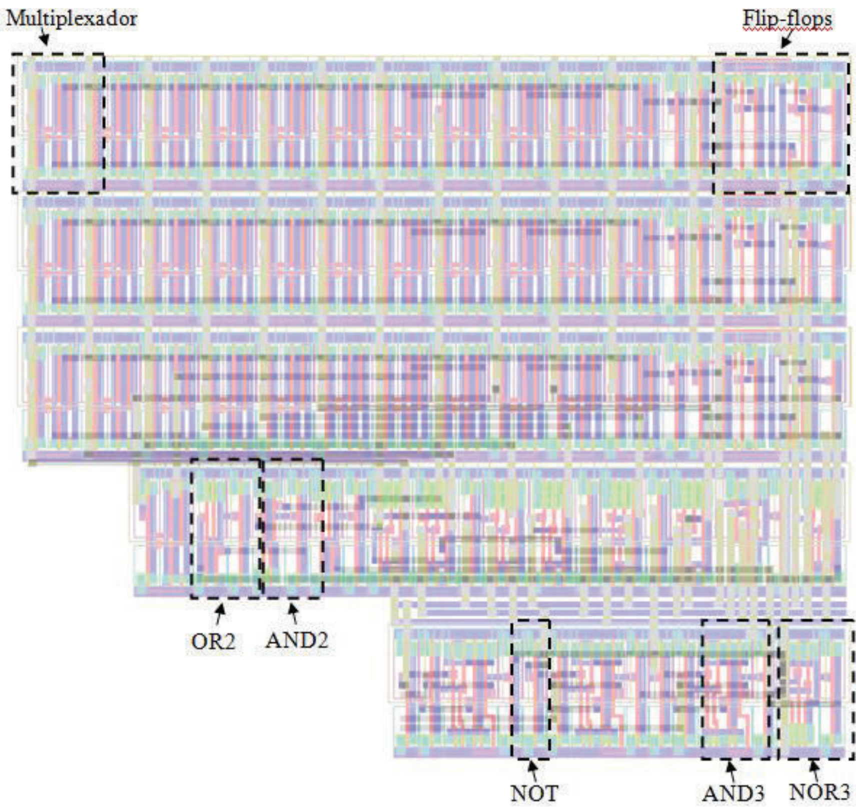


Figura 7.2 - Layout da unidade de *handshake*.

7.4 BLOCO 1: DEMODULADOR/ORDENADOR DE ENTRADA

7.4.1 Projeto

O bloco 1 recebe uma palavra analógica e extrai dela uma versão digitalizada (**r**), além de realizar o processamento inicial necessário para os blocos subsequentes, como a criação do vetor de confiabilidade (**s**).

Assim como realizado para o bloco de *handshake*, o primeiro passo na implementação do bloco 1 foi realizado na ferramenta Quartus II, onde os circuitos foram criados e simulados utilizando-se diagramas esquemáticos. A partir desses circuitos, foram criados os esquemáticos em nível de transistor, usados na criação o código SPICE. Após as simulações do código SPICE e os ajustes para o correto funcionamento do bloco, foi feito o layout do bloco.

Para analisar em detalhes cada um dos cinco blocos do decodificador (Figura 5.2), antes de incluí-los em um mesmo projeto, cada bloco foi projetado e fabricado individualmente. Para isso, foi necessária a inserção de circuitos adicionais para a entrada e saída de dados, devido à quantidade limitada de pinos no chip.

No bloco 1, por exemplo, para que todas as entradas e saídas fossem paralelas, seriam necessários aproximadamente 60 pinos, porém apenas 40 estavam disponíveis devido ao encapsulamento DIP40 utilizado. Uma opção seria a entrada serial dos dados; entretanto, alguns blocos têm matrizes como entradas, o que tornaria o processo de inserção lento e trabalhoso. A estratégia adotada foi incluir quatro valores diferentes para os dados de entrada, selecionados através de pinos de entrada no chip (Tabela 7.2). Para que as saídas pudessem ser verificadas com o osciloscópio, elas foram multiplexadas, de acordo com a seleção dos pinos sel_out (3 bits) (Tabela 7.3).

Tabela 7.2 - Seleção dos dados de entrada para o bloco 1.

Entradas	Valor de x	Saídas esperadas
sel_in = 0 0	**x** = 2 5 1 3 0 7 6	**r** = 0 1 0 0 0 1 1, **s** = 6 5 7 3 2 1 4
sel_in = 0 1	**x** = 2 1 4 0 7 6 3	**r** = 0 0 1 0 1 1 0, **s** = 5 4 6 2 1 7 3
sel_in = 1 0	**x** = 5 2 0 3 4 6 1	**r** = 1 0 0 0 1 1 0, **s** = 3 7 6 2 1 5 4
sel_in = 1 1	**x** = 0 1 2 3 5 6 7	**r** = 0 0 0 0 1 1 1, **s** = 7 1 6 2 5 3 4

Tabela 7.3 - Seleção dos dados de saída do bloco 1.

Entradas	Saídas
sel_out = 0 0 0	r1, s1
sel_out = 0 0 1	r2, s2
sel_out = 0 1 0	r3, s3
sel_out = 0 1 1	r4, s4
sel_out = 1 0 0	r5, s5
sel_out = 1 0 1	r6, s6
sel_out = 1 1 0	r7, s7
sel_out = 1 1 1	r7, s7

A Figura 7.3 mostra o layout do bloco 1 original, enquanto a Figura 7.4 mostra a inclusão dos circuitos adicionais de entrada e saída (registradores e multiplexadores). A adição destes circuitos aumenta a área de silício utilizada na fabricação do chip; entretanto, são utilizados apenas no layout individual dos blocos. A versão final do decodificador contém apenas os blocos originais.

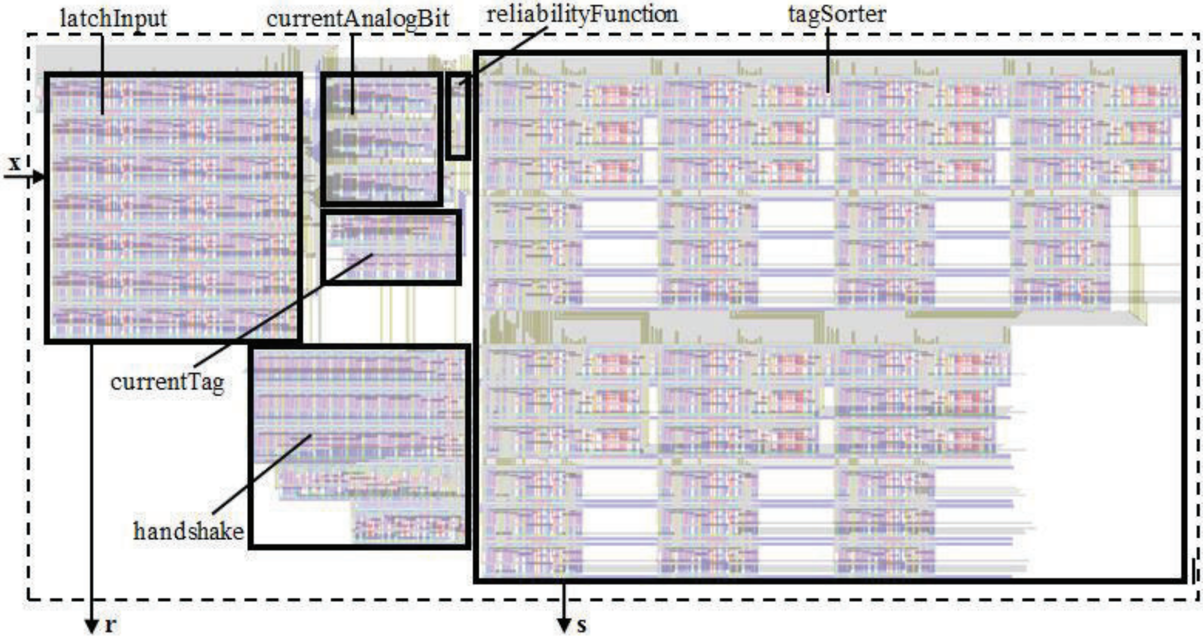


Figura 7.3 - Layout original do bloco de demodulação/ordenação de entrada, composto pelos sub-blocos: latchInput (armazena a entrada analógica); currentAnalogBit (seleciona um símbolo da palavra analógica); currentTag (atribui uma etiqueta para o símbolo selecionado); reliabilityFunction (calcula a confiabilidade do símbolo); tagSorter (ordena os símbolos de acordo com a confiabilidade); e handshake.

O layout completo do bloco 1, em seu padframe, é mostrado na Figura 7.5. Neste chip foram colocados os blocos 1 e 3 do decodificador, pois a área utilizada por esses blocos permitia fabricá-los no mesmo o chip. Entretanto, os blocos não têm comunicação entre si e os pinos operam de modo independente, para que um bloco não influencie o outro.

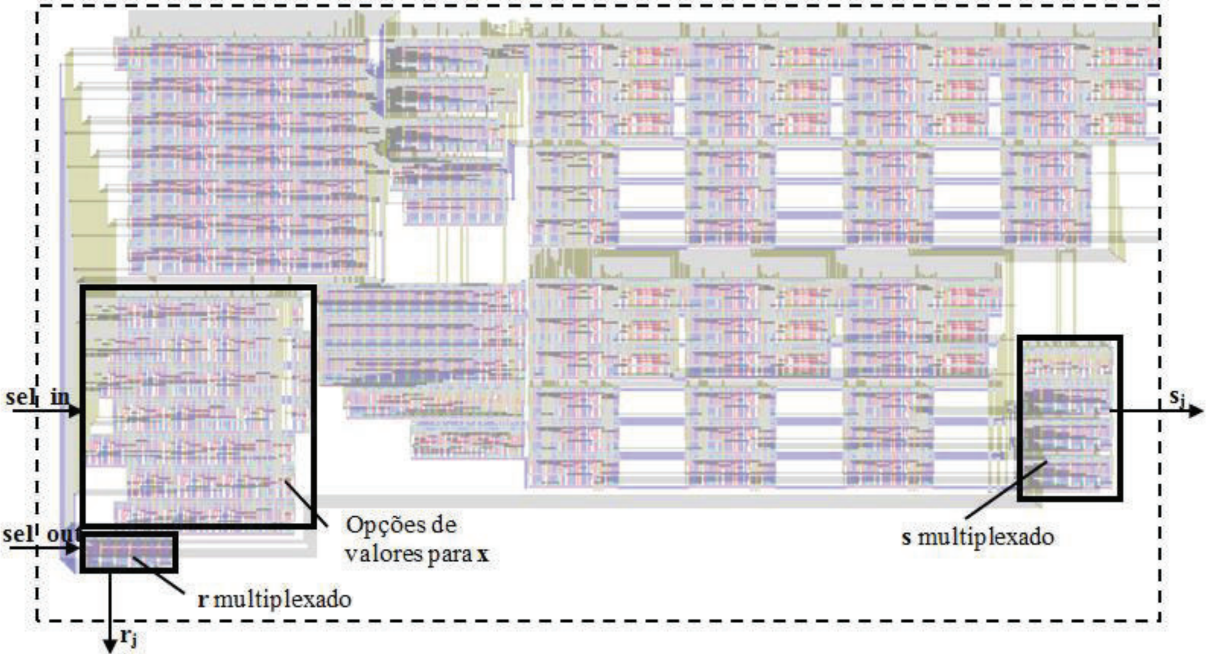


Figura 7.4 - Layout do bloco de demodulação/ordenação de entrada com os circuitos adicionais de entrada/saída, utilizados devido ao número limitado de pinos disponíveis no chip.

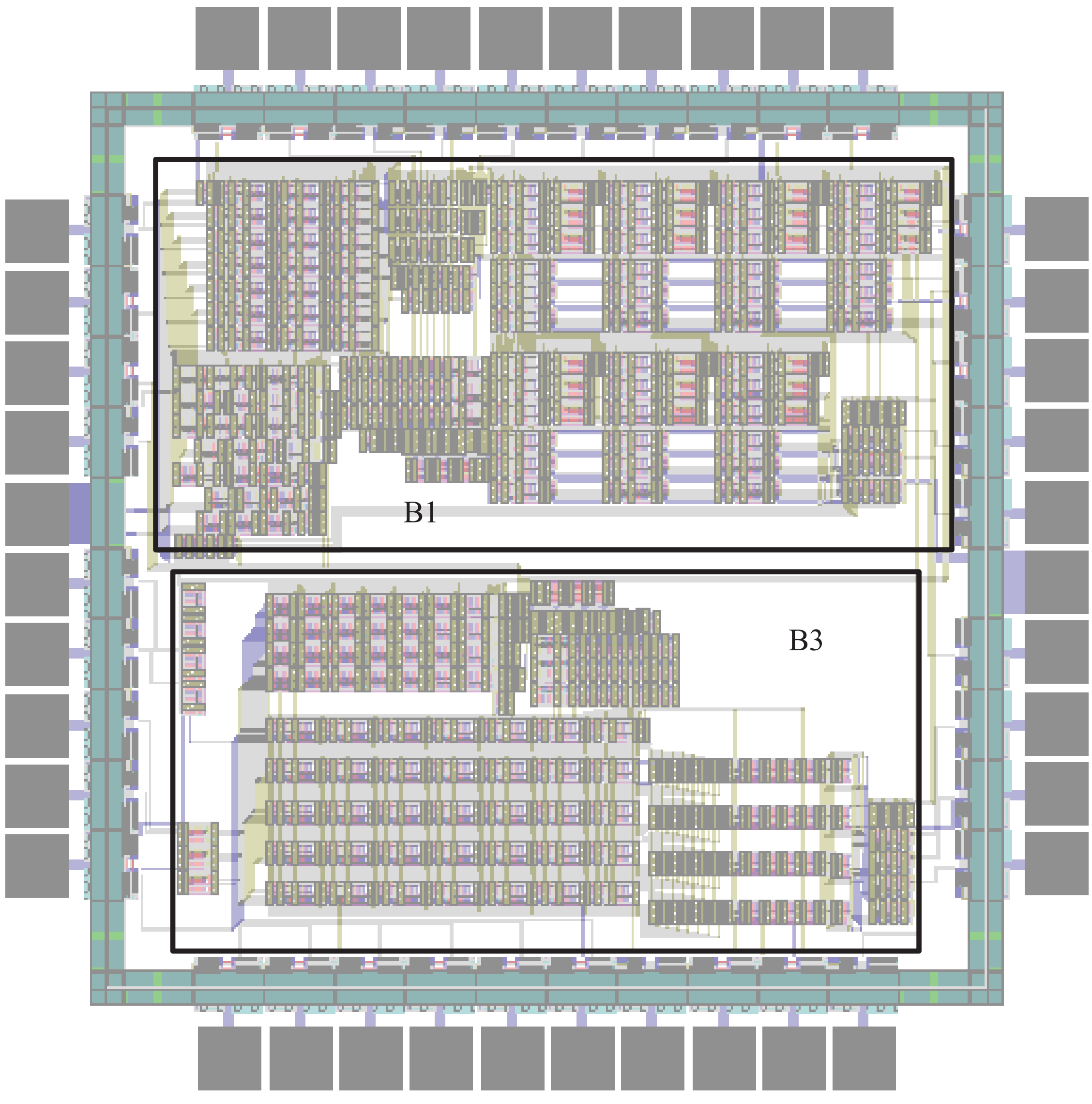


Figura 7.5 – Layout final do chip contendo os blocos de demodulação/ordenação de entrada e de geração de mensagens candidatas, fabricado com o objetivo de testar os blocos individualmente.

7.4.2 Resultados

A Figura 7.6 mostra a simulação do esquemático criado com diagramas de bloco na ferramenta Quartus II. É possível verificar que após os dados de entrada se tornarem prontos para uso (*inputReady* = 1), são necessários sete ciclos de clock (*n*) para que **s** (vetor de confiabilidade) se torne disponível (*outputReady* = ‘1’).

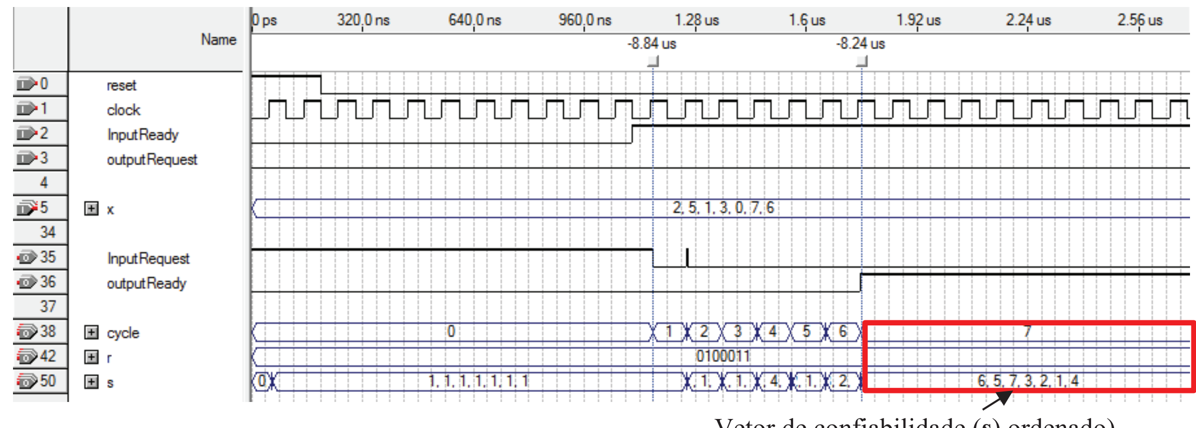


Figura 7.6 - Simulação do esquemático do demodulador/ordenador no Quartus II.

O resultado da implementação SPICE deste bloco é mostrada nas Figuras 8.7, 8.8 e 8.9. A Figura 7.7 apresenta a simulação SPICE para o bloco 1 antes da criação do layout, sem inclusão de pads e capacitâncias parasitas. Nesta primeira implementação, o objetivo é verificar a funcionalidade do bloco. Nesta figura é mostrado o resultado da decisão abrupta (r1 a r7), que para **x** = 2 5 1 3 0 7 6 é "0100011", e os sinais s_out, referentes à primeira posição do vetor de confiabilidade (6, 5, 7, 3, 2, 1, 4).

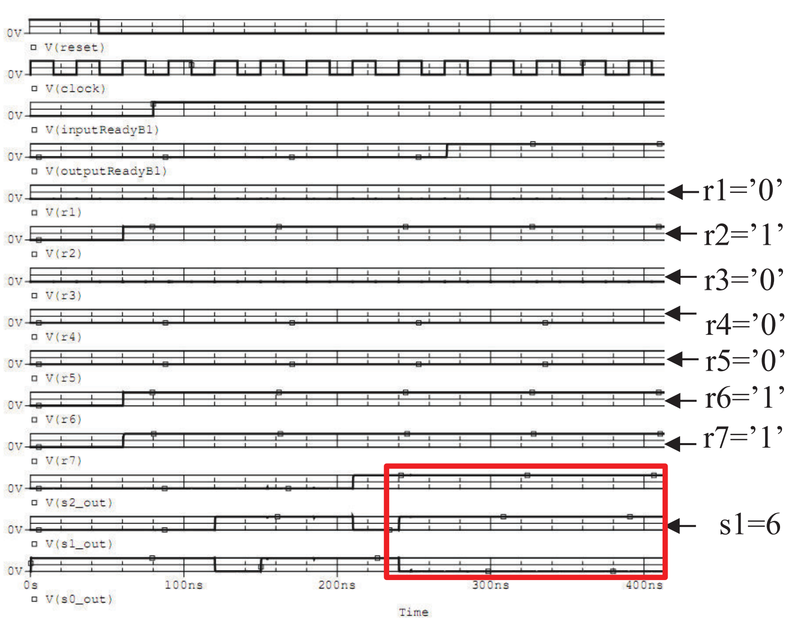


Figura 7.7 - Simulação SPICE do demodulador/ordenador sem capacitância parasitas e sem pads.

A Figura 7.8 mostra o resultado da implementação do mesmo bloco. Entretanto, nesta simulação foram incluídas as capacitâncias parasitas associadas ao layout do bloco 1 e pads. Por esse motivo, é possível notar as curvas de carga e descarga mais acentuadas nas transições dos sinais.

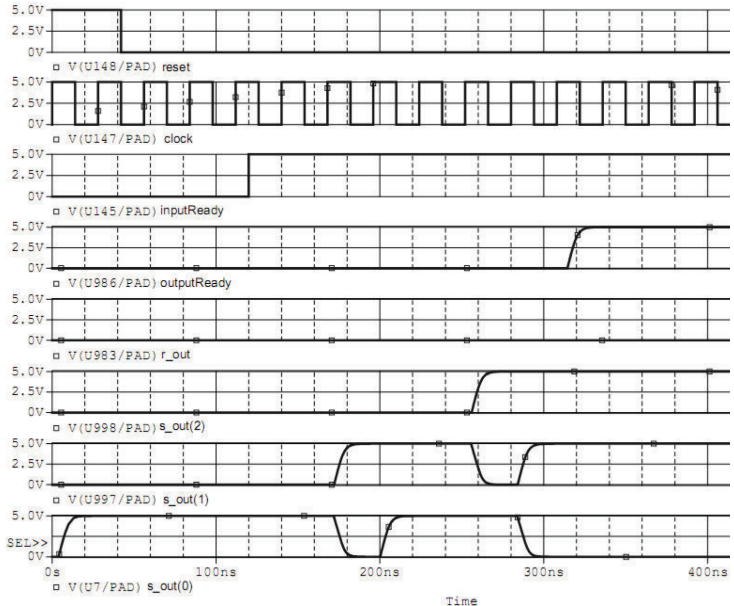


Figura 7.8 - Simulação SPICE do demodulador/ordenador, incluindo capacitância parasitas e pads.

A Figura 7.9 mostra os sinais, capturados com o osciloscópio, do chip fabricado com o layout da Figura 7.4. Na primeira linha, vemos o sinal de clock (amarelo) e de *inputReady* (verde). Os sinais D8-D14 são o resultado da decisão abrupta para o mesmo valor de **x** das simulações SPICE (**r** = 0100011).

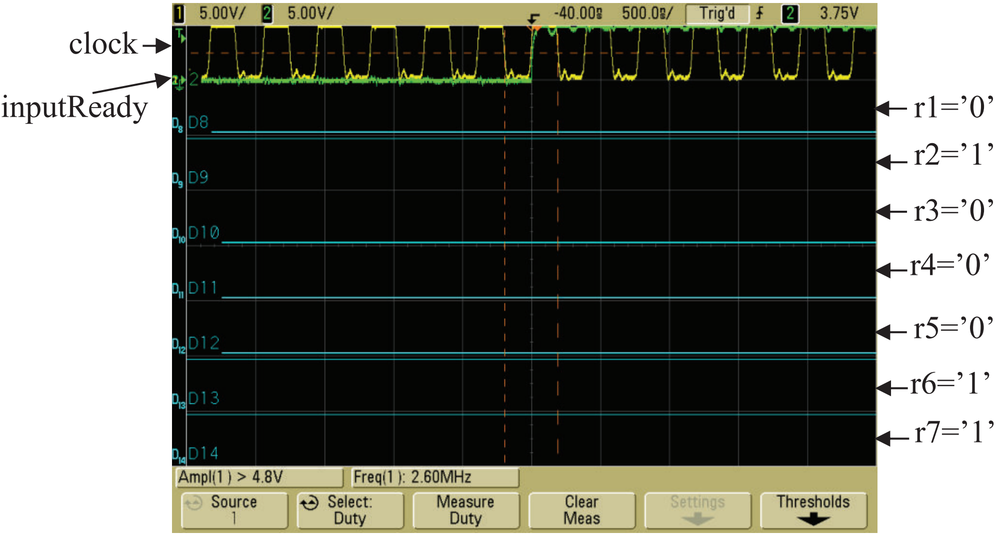


Figura 7.9 - Bloco 1 decisão abrupta (osciloscópio).

7.5 BLOCO 2: REDUÇÃO DE GAUSS-JORDAN MODIFICADA

7.5.1 Projeto

O bloco 2, referente à redução de Gauss-Jordan modificada, é o bloco mais complexo deste projeto. Na implementação em FPGA, corresponde a aproximadamente a metade da área de silício necessária para a construção do decodificador completo. Este bloco recebe o vetor de confiabilidade **s** (gerado pelo bloco 1) e a matriz geradora **G**, e produz as matrizes $\mathbf{G}_r$ (**G** reduzida) e $\mathbf{G}_{r0}$ (**G** reduzida transformada), vistas na seção 5.4.1.

Um chip contendo apenas este bloco foi fabricado. Para isso, foi necessário que circuitos adicionais fossem incluídos no bloco para a entrada e saída dos dados. Foram armazenados quatro valores diferentes para **s**, selecionados através de pinos de entrada no chip (Tabela 7.4). Antes da criação do layout deste bloco, foram realizadas simulações dos esquemáticos e do código SPICE.

Tabela 7.4 - Seleção dos dados de entrada do bloco 2.

Entradas	Valor atribuído à s	Saídas esperadas
sel_in = 0 0	**s** = 6 5 7 3 2 1 4	$\mathbf{G}_r$= 0011010, 101100, 1010001, 10010000
		$\mathbf{G}_{r0}$= 0000010, 0000100, 0000001, 0100000
sel_in = 0 1	**x** = 5 4 6 2 1 7 3	$\mathbf{G}_r$= 0110100, 0111001, 0100011, 1010001
		$\mathbf{G}_{r0}$= 0000100, 0001000, 0000010, 1000000
sel_in = 1 0	**x** = 3 7 6 2 1 5 4	$\mathbf{G}_r$= 1011100, 0001101, 1000110, 1101000
		$\mathbf{G}_{r0}$= 0010000, 0000001, 0000010, 0100000
sel_in = 1 1	**x** = 7 1 6 2 5 3 4	$\mathbf{G}_r$= 0001101, 1011100, 0011010, 0110100
		$\mathbf{G}_{r0}$= 0000001, 1000000, 0000010, 0100000

A Figura 7.10 mostra o layout original do bloco 2, enquanto a Figura 7.11 mostra também os circuitos de entrada e saída. A diferença entre os dois layouts é destacada na Figura 7.11. No circuito original, existe apenas uma matriz de flips-flops que armazena a sequência de confiabilidade entregue pelo bloco 1. No segundo layout quatro valores diferentes para **s** são armazenados, como visto na Tabela 7.4. A versão final do layout do bloco 2, no padframe do chip, consta na Figura 7.12.

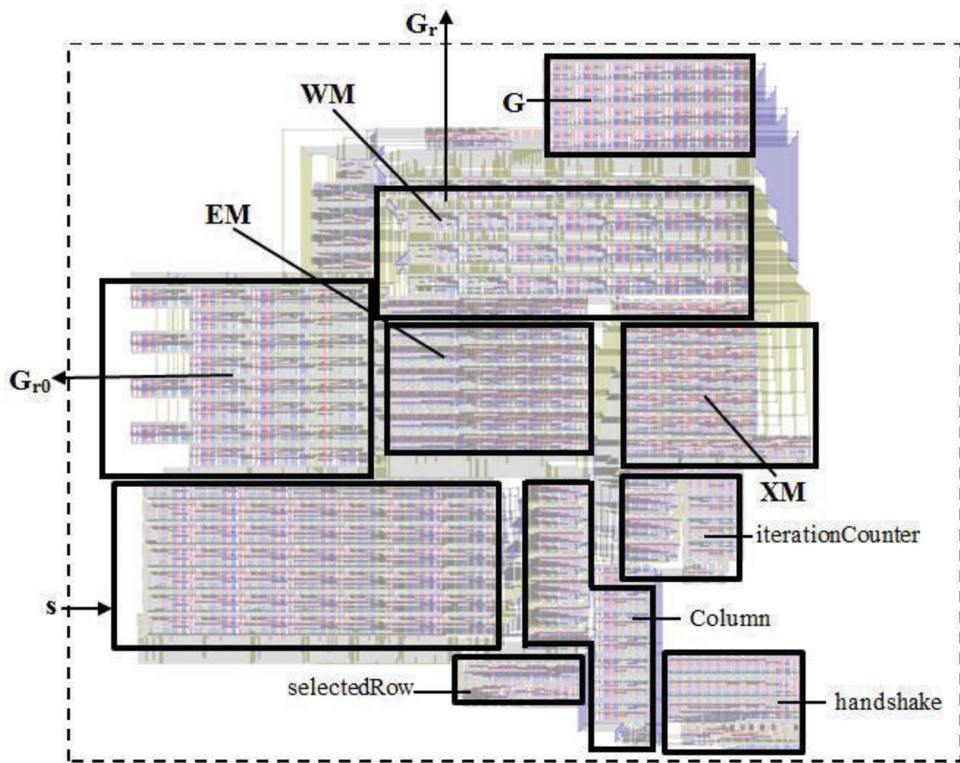


Figura 7.10 - Layout original do bloco de eliminação de Gauss Jordan modificada, composto pelos sub-blocos: s (vetor de confiabilidade); G (matriz geradora); G_r (matriz geradora reduzida ou matriz de trabalho); G_{r0} (matriz geradora reduzida, com todas as colunas não selecionadas substituídas por vetores nulos); EM (matriz resultante após a troca de linhas); EX (matriz resultante após a operação XOR); column (seleção de uma coluna da matriz G, segundo o vetor de confiabilidade), iterationCounter (contador de iterações); selectedRow (linha da matriz G usada na troca de linhas); e handshake .

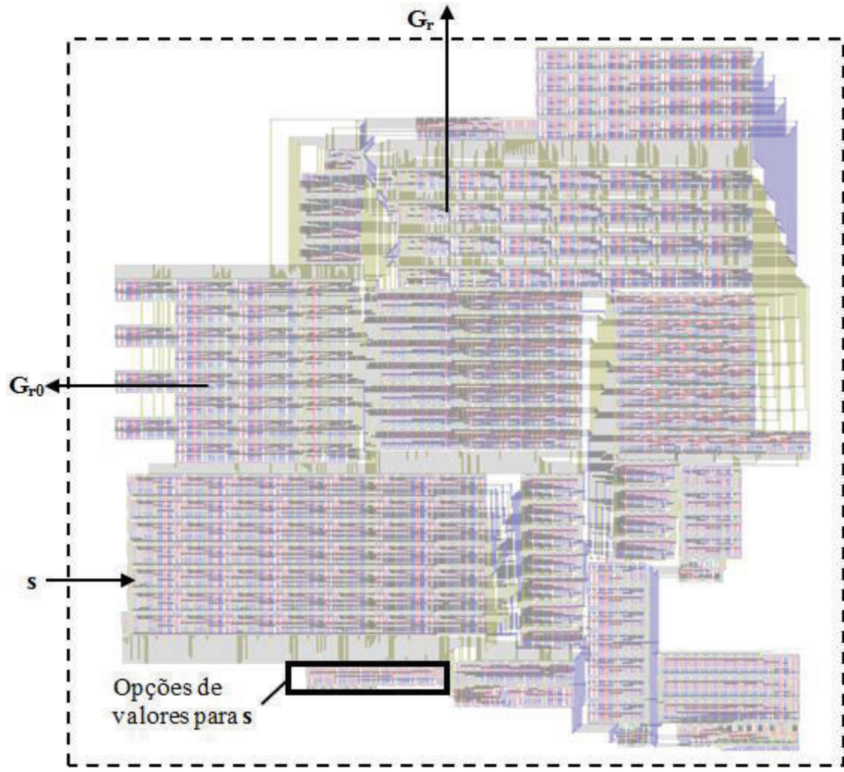


Figura 7.11 - Layout do bloco de eliminação de Gauss Jordan modificada com circuitos adicionais de entrada/saída, utilizados devido ao número limitado de pinos disponíveis no chip.

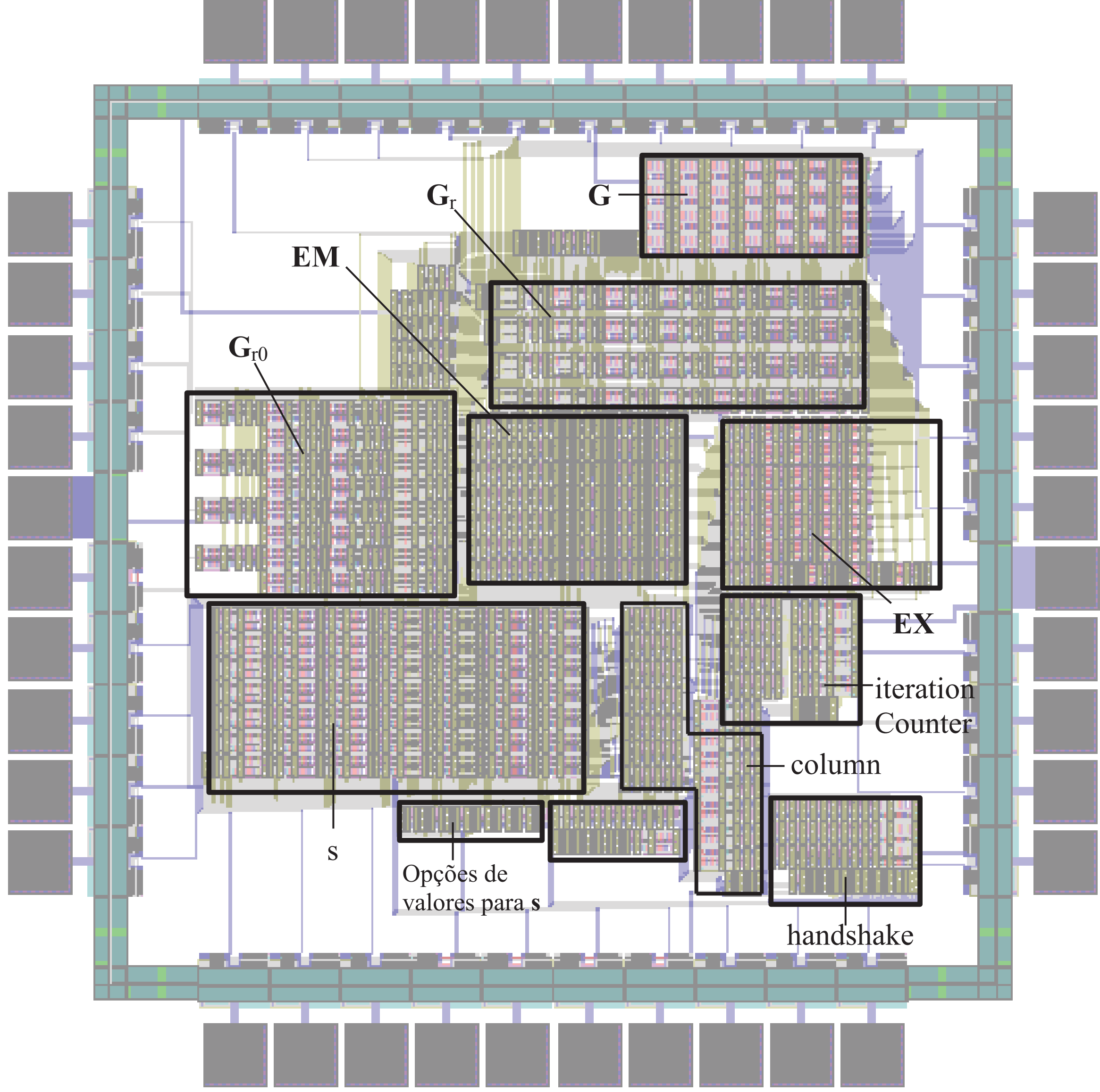


Figura 7.12 - Layout final do chip contendo o bloco de eliminação de Gauss Jordan modificada, fabricado com o objetivo de testar o bloco individualmente.

7.5.2 Resultados

Diversas simulações foram realizadas para verificar o funcionamento do bloco de redução de Gauss-Jordan modificada, utilizando a ferramenta Quartus II. A Figura 7.13 mostra uma dessas simulações, cuja matriz geradora é a mesma da Figura 4.1(a). Observe que a impossibilidade de eliminar a coluna 3 (Figura 4.1(e)) ocorre no ciclo 4 da simulação. Em seguida, a coluna 2 é tomada como segunda tentativa para obter um conjunto de informação. O conjunto de informação é encontrado dentro do número máximo de iterações, ou seja, $n-d_{Hmin}+1$ ciclos de clock, como citado no final da seção 5.4.1.

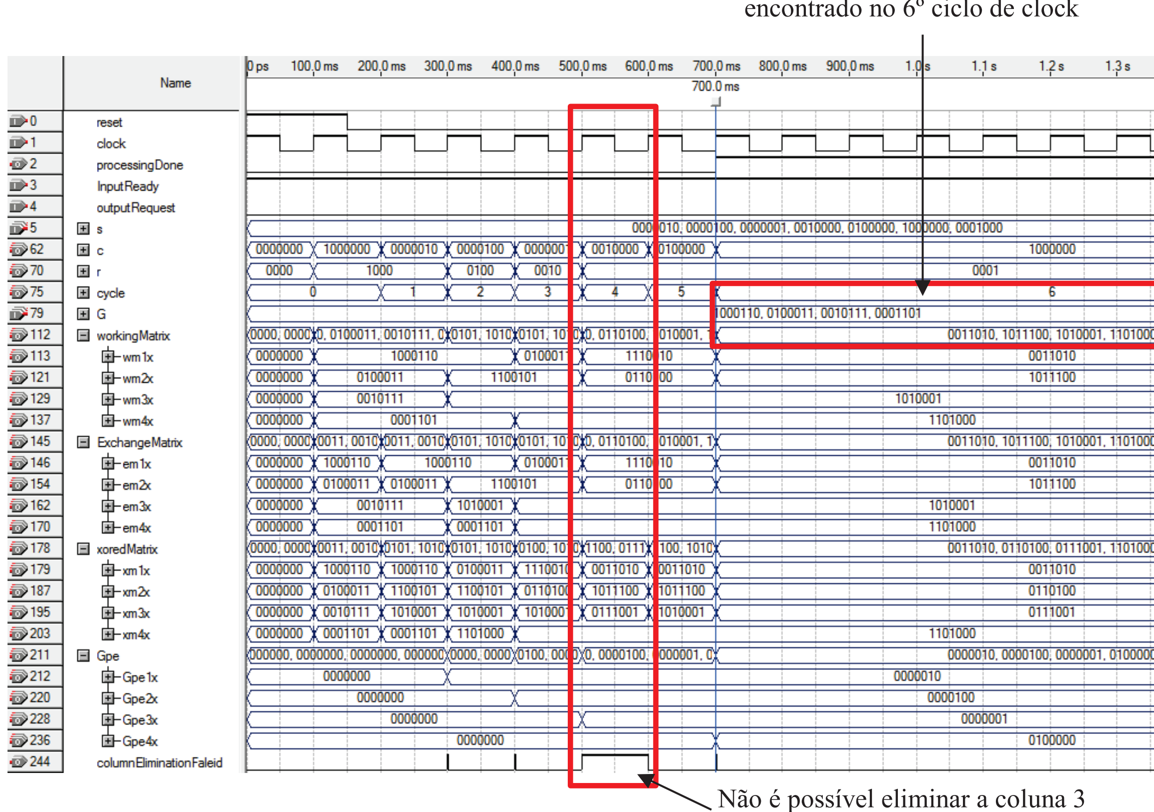


Figura 7.13 - Simulação do bloco de redução de Gauss-Jordan modificada.

Simulações do código SPICE foram realizadas para verificar o funcionamento dos circuitos com transistores (Figura 7.14).

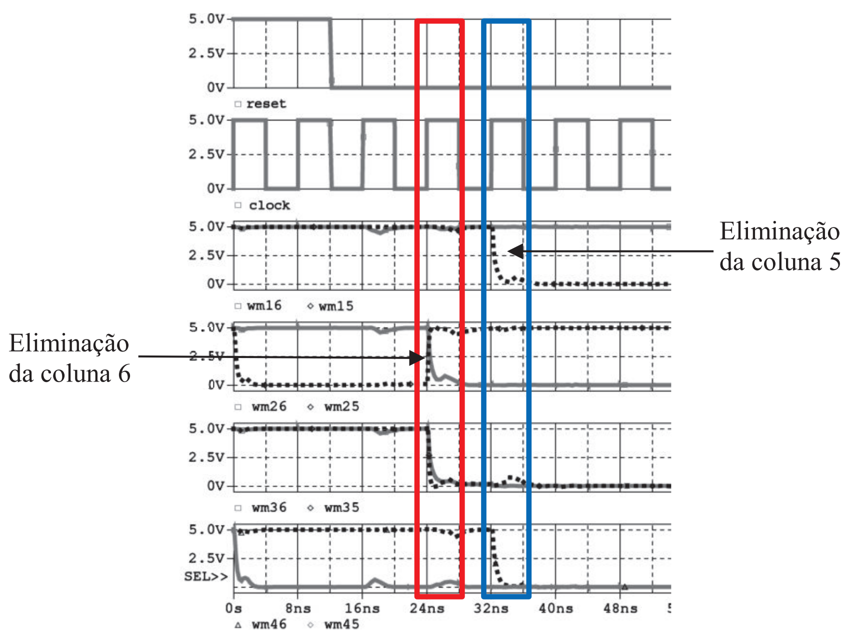


Figura 7.14 - Resultados da simulação do código SPICE para a redução de Gauss-Jordan modificada.

Os sinais wm_{15} a wm_{45} e wm_{16} a wm_{46}, mostrados nesta figura, referem-se às colunas 5 (linha pontilhada) e 6 (linha sólida) da **WM**. Seus valores originais eram 1011 e 1110, respectivamente, como mostrado na Figura 4.1(a). Após a eliminação da coluna 6, os novos valores são 1101 e 1000, correspondentes aos valores da Figura 4.1(b). Em seguida, a coluna 5 é eliminada, resultando em 0100 e 1000. O fato de cada coluna ter se tornado um vetor unitário indica que a eliminação ocorreu como esperado. Além disso, é possível perceber que essas colunas permanecem estáveis pelo resto da simulação, o que também está de acordo com o comportamento esperado.

A Figura 7.15 mostra o processo completo de redução da primeira linha da matriz de trabalho. Comparando essa figura com a Figura 4.1, que mostra passo a passo da redução da matriz, vemos que a cada evento do clock os sinais wm11 a wm17 se alteram até assumir o valor "0011010", referente ao passo da Figura 4.1(f). A simulação da Figura 7.16, diferente da Figura 7.15, inclui as capacitâncias parasitas e os pads. Por esse motivo, nota-se subidas e descidas mais lentas nos sinais.

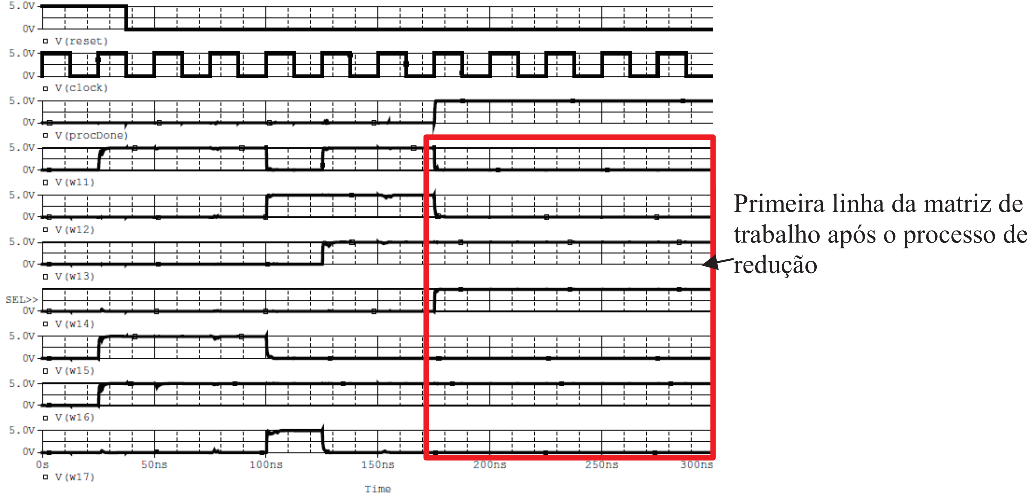


Figura 7.15 - Simulação SPICE do bloco de redução de Gauss-Jordan modificada sem capacitâncias parasitas e sem pads.

Como mencionado, o bloco 2 é muito importante no processo de decodificação. É o bloco do decodificador que ocupa maior área de silício e contém mais circuitos combinacionais e sequenciais, além de utilizar mais ciclos de clock no processo (n-d_{Hmin}+1, no pior caso). Para determinar a frequência máxima de operação deste bloco, foram feitos testes de simulação e de bancada, variando-se o clock até que os limites de operação fossem encontrados. A frequência máxima obtida em simulação, considerando as capacitâncias parasitas, foi de 38 MHz (tecnologia 0.5 μm). Este teste também foi realizado diretamente no

chip, com o osciloscópio (Figura 7.17). O valor medido na prática, 33 MHz, está bastante próximo do valor encontrado na simulação. Na Figura 7.17 observa-se o sinal de clock e o primeiro elemento da matriz de trabalho (wm11). A partir de 33 MHz, o sinal é bastante distorcido e os níveis lógicos não são mais claramente definidos. A frequência máxima é delimitada principalmente pelos circuitos combinacionais do bloco. Além disso, os pads de entrada e saída não operam em altas frequências.

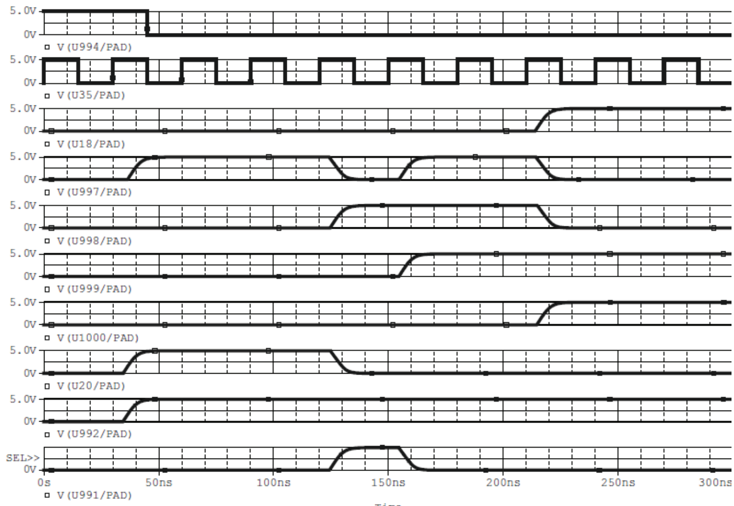


Figura 7.16 - Simulação SPICE do bloco de redução de Gauss-Jordan modificada com capacitâncias parasitas e com pads.

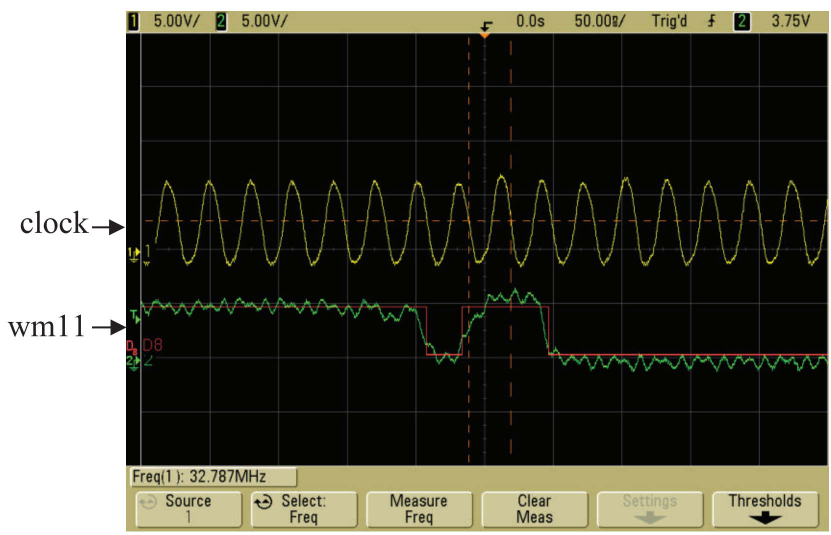


Figura 7.17 - Frequência máxima de operação do bloco 2 (osciloscópio).

O mesmo código SPICE, incluindo as capacitâncias parasitas e os pads, foi simulado para as tecnologias de 180 nm (IBM 7RF) e 130 nm (IBM 8RF). As frequências máximas

foram 67 MHz e 77 MHZ, respectivamente. Entretanto, esses resultados poderiam ser melhores se os transistores utilizassem seus tamanhos mínimos, o que não é o caso das simulações realizadas, já que essas tecnologias não são escaláveis (onde apenas o valor de lambda é alterado). Considerando os resultados obtidos na FPGA da família Stratix III (tecnologia de 65 nm), que foram de aproximadamente 159 MHz, nota-se que o chip fabricado apresenta um desempenho inferior, pois utiliza uma tecnologia mais antiga. Entretanto, os resultados obtidos em simulação mostram que à medida que tecnologias mais atuais são utilizadas, a frequência máxima de operação aumenta. Infelizmente, os parâmetros SPICE para a tecnologia de 65 nm não estão disponíveis, o que impediu uma comparação direta.

7.6 BLOCO 3: GERADOR DE MENSAGENS CANDIDATAS

7.6.1 Projeto

O bloco 3 é responsável por gerar k+1 mensagens candidatas, as quais são geradas em um único ciclo de clock. Na implementação deste bloco também foram utilizados circuitos auxiliares para entrada e saída de dados, assim como nos demais blocos do decodificador. Neste caso, quatro valores diferentes para Gr0 foram armazenados. A seleção do valor utilizado é feita através de dois pinos de entradas (sel_in[1..0]). A sequência r (palavra decodificada por decisão abrupta) é inserida de modo paralelo. Os valores possíveis para Gr0 e saídas esperadas estão listados na

Tabela 7.5. Devido ao número limitado de pinos de entrada e saída no chip, não seria possível disponibilizar todas as mensagens candidatas simultaneamente. Por esse motivo, são disponibilizadas uma por vez, selecionadas através de três pinos de entrada (sel_out[2..0]), como mostra a Tabela 7.6.

Tabela 7.5 - Seleção dos dados de entrada para o bloco 3.

Entradas	Valor atribuído a $\mathbf{G}_{r0}$	Saídas esperadas
sel_in = 00	$\mathbf{G}_{r0}$ = 0000010, 0000100, 0000001, 0100000	$\mathbf{U_c}$ = 1011, 0011, 1111, 1001, 1010
sel_in= 0 1	$\mathbf{G}_{r0}$ = 0000100, 0001000, 0000010, 1000000	$\mathbf{U_c}$ = 1010, 0010, 1110, 1000, 1011
sel_in= 1 0	$\mathbf{G}_{r0}$ = 0010000, 0000001, 0000010, 0100000	$\mathbf{U_c}$ = 0010, 1010, 0110, 0000, 0011
sel_in= 1 1	$\mathbf{G}_{r0}$ = 0000001, 1000000, 0000010, 0100000	$\mathbf{U_c}$ = 1010, 0010, 1110, 1000, 1011

Tabela 7.6 - Seleção dos dados de saída do bloco 3.

Entradas	Saídas
sel_out = 0 0 0	$\mathbf{u}_0$
sel_out = 0 0 1	$\mathbf{u}_1$
sel_out = 0 1 0	$\mathbf{u}_2$
sel_out = 0 1 1	$\mathbf{u}_3$
sel_out = 1 0 0	$\mathbf{u}_4$
sel_out = 1 0 1	$\mathbf{u}_4$
sel_out = 1 1 0	$\mathbf{u}_4$
sel_out = 1 1 1	$\mathbf{u}_4$

A Figura 7.18 mostra o layout original do bloco 3. Este é o bloco do decodificador que ocupa a menor área de silício. A Figura 7.19 apresenta o layout do bloco gerador de mensagens candidatas, que inclui os circuitos de entrada e saída. A versão final do layout do bloco 3, no padframe do chip, é mostrada na Figura 7.5, a qual inclui também o bloco 1 do decodificador.

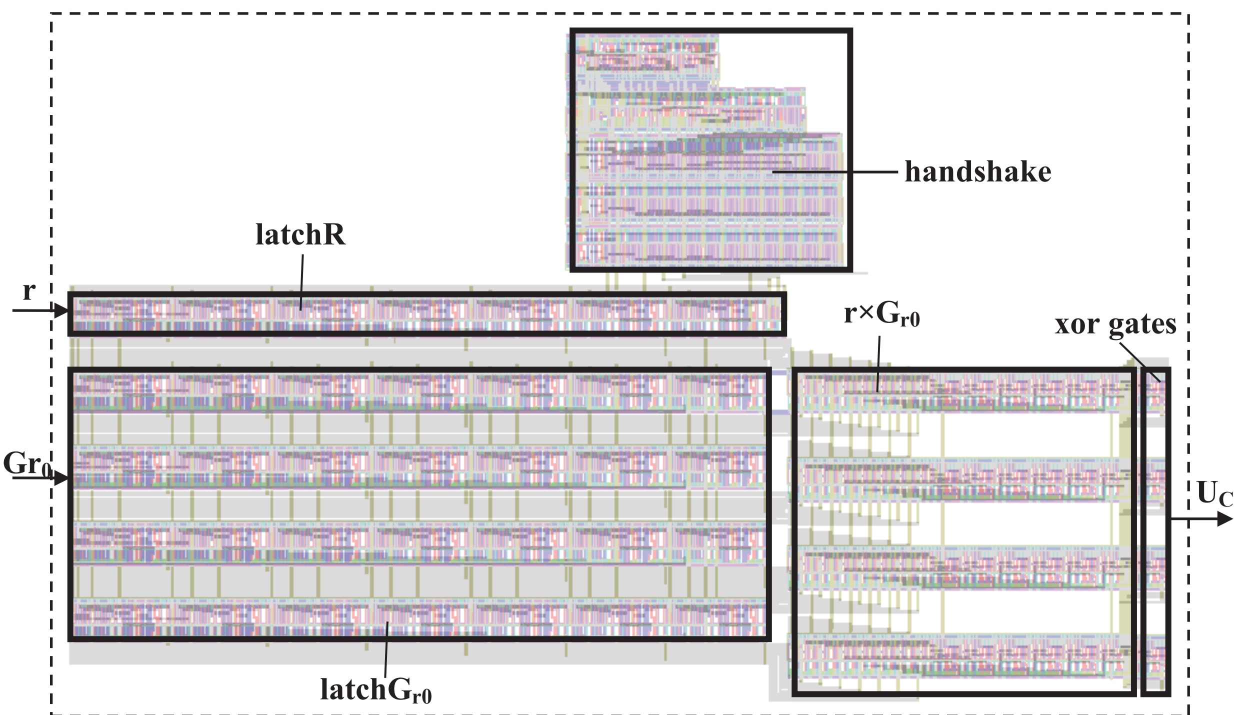


Figura 7.18 - Layout original do bloco de geração de mensagens candidatas, composto pelos sub-blocos: latchR (armazena o sequência demodulada por decisão abrupta); latchG_{r0} (armazena a matriz geradora reduzida, com todas as colunas não selecionadas substituídas por vetores nulos); r × G_{r0} (realiza a multiplicação entre o vetor r e a matriz G_{r0}); xor gates (realiza a operação xor entre a mensagem candidata original, para geração das demais mensagens candidatas); e handshake.

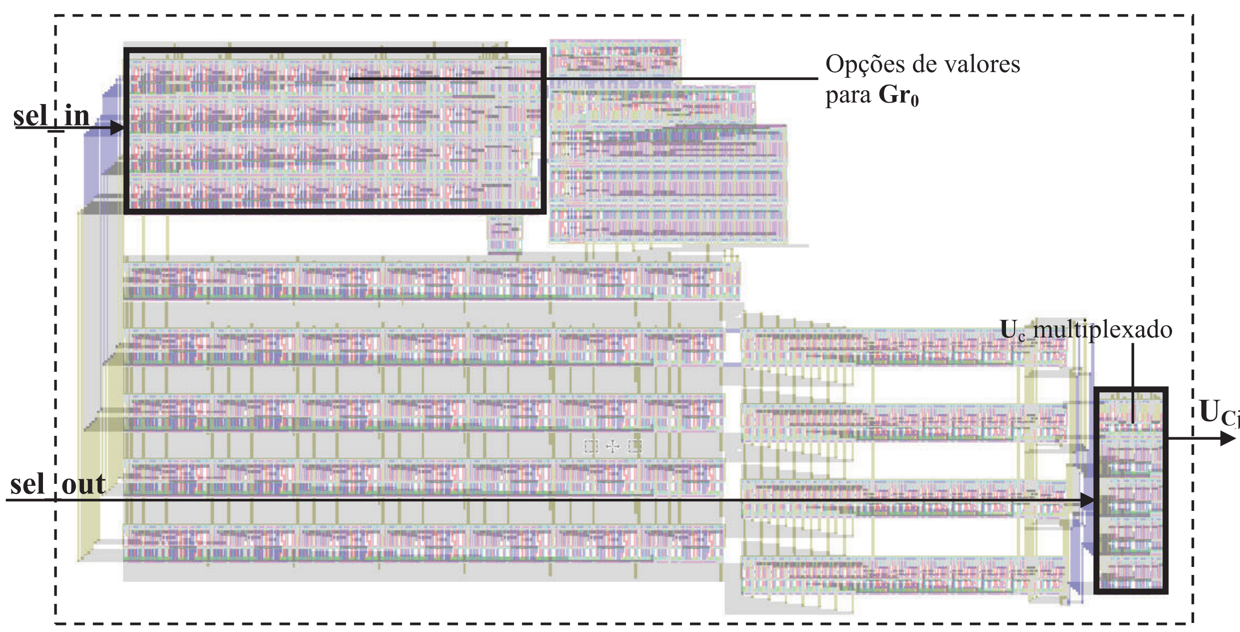


Figura 7.19 - Layout do bloco de geração de mensagens candidatas com circuito adicional de entrada/saída.

7.6.2 Resultados

A Figura 7.20 mostra a simulação do circuito implementado na ferramenta Quartus II. Após os dados de entrada se tornarem prontos para uso (*inputReady* = 1), é necessário apenas um ciclo de clock para que as *k*+1 mensagens sejam geradas.

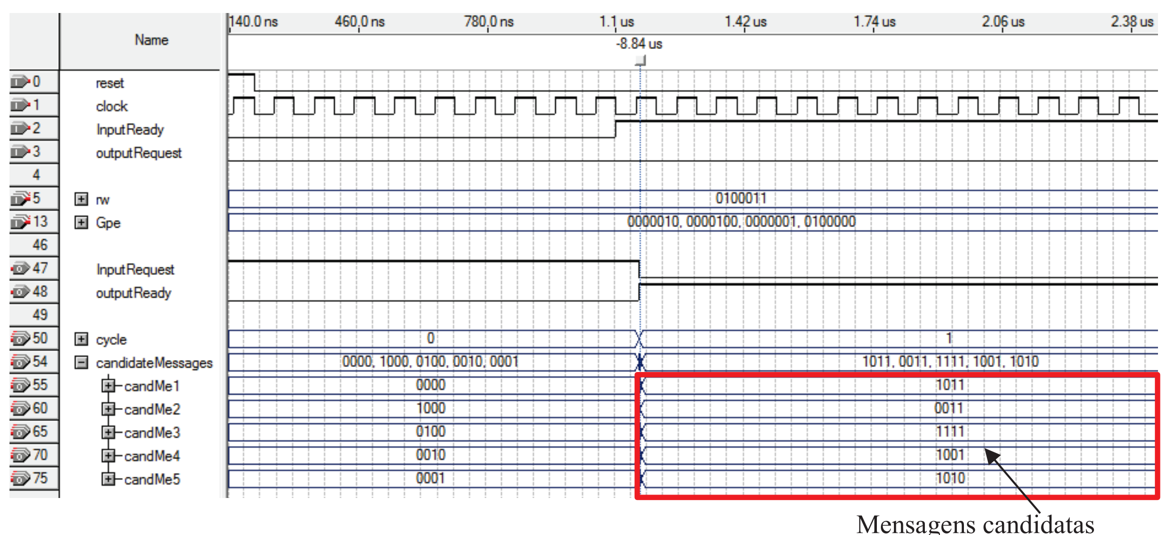


Figura 7.20 - Simulação do esquemático do gerador de mensagens candidatas no Quartus II.

A Figura 7.21 apresenta a simulação SPICE para o bloco 3, antes da criação do layout, portanto sem inclusão de pads e capacitâncias parasitas. Nesta primeira implementação, o

objetivo é verificar a funcionalidade do bloco. Na simulação, vemos a mensagem candidata original ($\mathbf{u}_0$) e alguns sinais de controle. A Figura 7.22 mostra a simulação SPICE pós-layout, incluindo as capacitâncias parasitas e os pads.

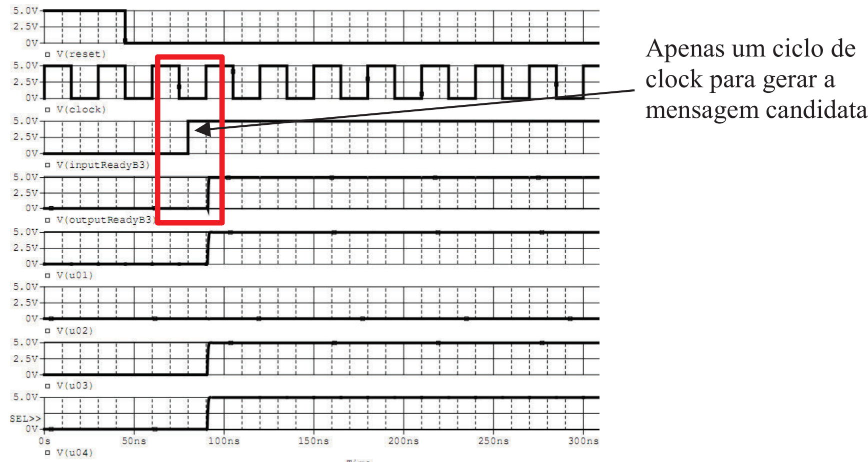


Figura 7.21 - Simulação SPICE do bloco de geração de mensagens candidatas sem capacitâncias parasitas e sem pads.

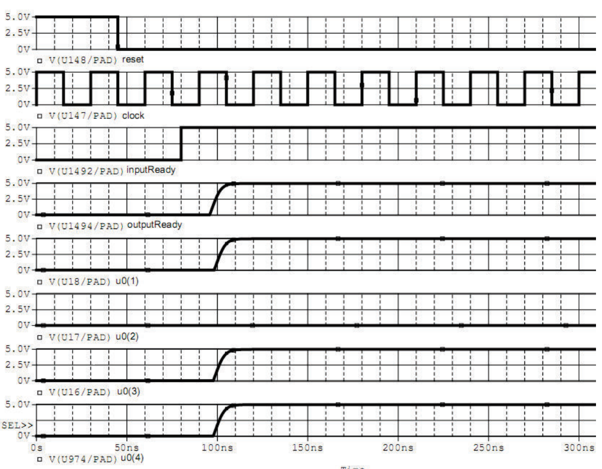


Figura 7.22 - Simulação SPICE do bloco de geração de mensagens candidatas com capacitâncias parasitas e com pads.

Finalmente, a Figura 7.23 mostra a medição dos sinais no chip fabricado, capturados com o osciloscópio. O primeiro sinal (em rosa) é o clock. Em seguida, em amarelo, o sinal de entrada *inputReady*. O sinal *outputReady* (em verde) passa para nível lógico alto no primeiro clock após *inputReady* = 1 . Os últimos quatro sinais referem-se à mensagem candidata $\mathbf{u}_0$ ($\mathbf{u}_{0=}$1011 para sel_in = 00).

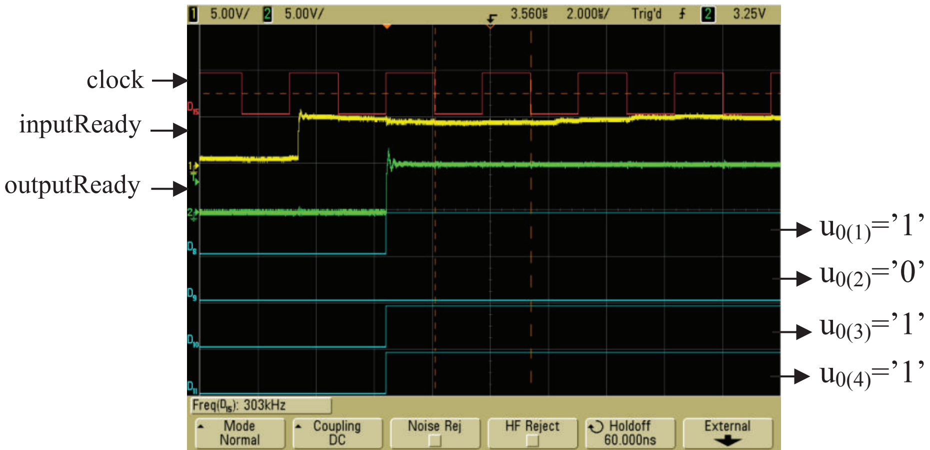


Figura 7.23 - Geração das mensagens candidatas (osciloscópio).

7.7 BLOCO 4: GERADOR DE PALAVRAS-CÓDIGO CANDIDATAS

7.7.1 Projeto

O bloco 4 do decodificador gera o conjunto das palavras-código candidatas, através da multiplicação entre as k+1 mensagens candidatas e a matriz $\mathbf{G}_r$.

Assim como realizado nos blocos anteriores, este bloco foi primeiramente implementado e simulado em alto e baixo nível, para em seguida realizar o projeto de layout.

Para a análise funcional do chip exclusivo para o bloco 4, foram utilizados quatro valores diferentes para as entradas $\mathbf{G}_r$ e $\mathbf{U}_c$. A Tabela 7.7 apresenta os possíveis valores de entrada do bloco, bem como as saídas esperadas para essas entradas. A saída da matriz $\mathbf{C}_c$ pode ser vista uma palavra por vez, utilizando-se três pinos de entrada do chip, que selecionam qual linha de $\mathbf{C}_c$ será disponibilizada nos pinos de saída (Tabela 7.8).

Tabela 7.7- Seleção dos dados de entrada para o bloco 4.

Entradas	Valor atribuído a $\mathbf{G}_r$ e $\mathbf{U}_c$	Saídas esperadas
sel_in = 0 0	$\mathbf{G}_r$ = 0011010, 1011100, 1010001, 1001000 $\mathbf{U}_c$ = 1011, 0011, 1111, 1001, 1010	$\mathbf{C}_c$ = 0100011, 0111001, 1111111, 1110010, 1001011
sel_in = 0 1	$\mathbf{G}_r$ =0110100, 0111001, 0100011, 1010001 $\mathbf{U}_c$ = 1010, 0010, 1110, 1000, 1011	$\mathbf{C}_c$ = 0010111, 0100011, 0101110, 0110100, 1000110
sel_in = 1 0	$\mathbf{G}_r$ = 1011100, 0001101, 1000110, 1101000 $\mathbf{U}_c$ = 0010, 1010, 0110, 0000, 0011	$\mathbf{C}_c$ = 1000110, 0011010, 1001011, 0000000, 0101110
sel_in = 1 1	$\mathbf{G}_r$ = 0001101, 1011100, 0011010, 0110100 $\mathbf{U}_c$ = 1010, 0010, 1110, 1000, 1011	$\mathbf{C}_c$ = 0010111, 0011010, 1001011, 0001101, 0100011

Tabela 7.8- Seleção dos dados de saída do bloco 4.

Entradas	Saídas
sel_out = 0 0 0	$\mathbf{c}_0$
sel_out = 0 0 1	$\mathbf{c}_1$
sel_out = 0 1 0	$\mathbf{c}_2$
sel_out = 0 1 1	$\mathbf{c}_3$
sel_out = 1 0 0	$\mathbf{c}_4$
sel_out = 1 0 1	$\mathbf{c}_4$
sel_out = 1 1 0	$\mathbf{c}_4$
sel_out = 1 1 1	$\mathbf{c}_4$

A Figura 7.24 mostra o layout original do bloco 3. Pode-se observar os sub-blocos responsáveis por registrar as entradas (latchU_c e latchG_r), o sub-bloco que realiza a multiplicação entre as entradas, e o *handshake* .

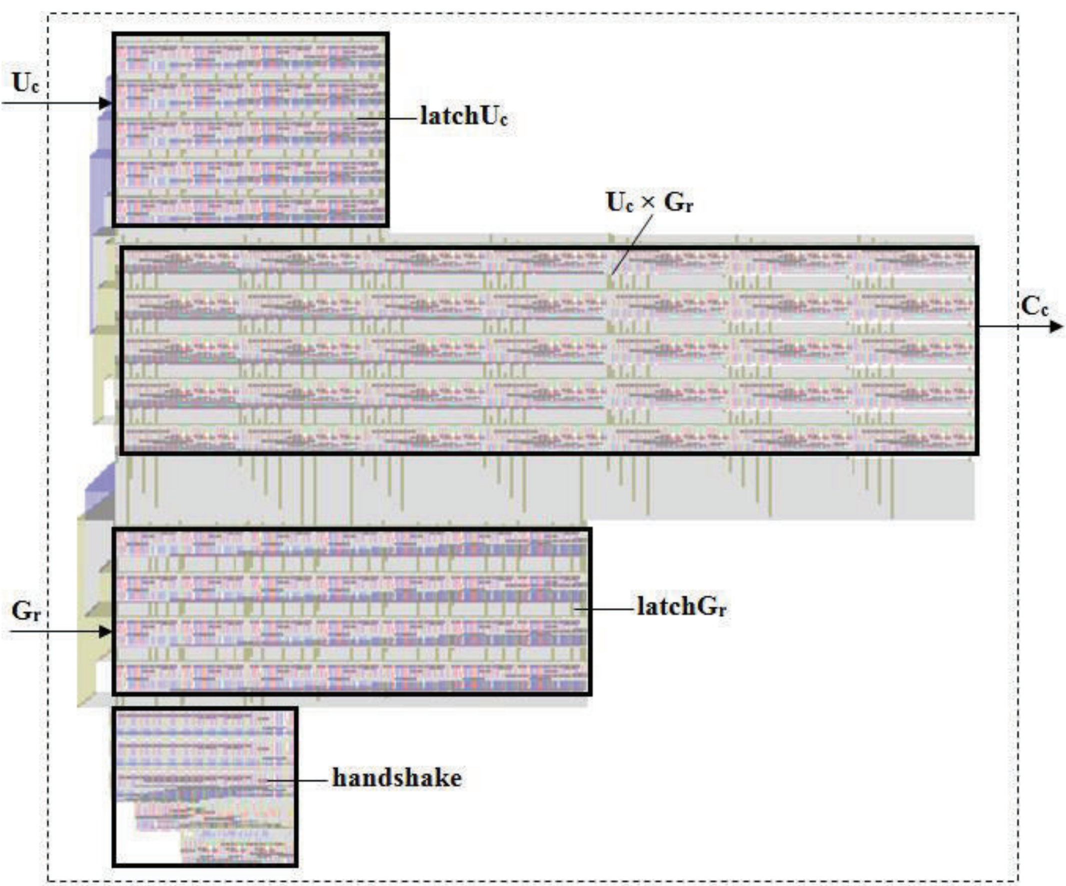


Figura 7.24 - Layout original do bloco de geração de palavras-código candidatas, composto pelos sub-blocos: latchUc (armazena a matriz de mensagens candidatas); latchG_r (armazena a matriz geradora reduzida); $U_c \times G_r$ (realiza a multiplicação entre a matriz de mensagens candidatas e a matriz Gr); e handshake.

Na Figura 7.25 é incluído o circuito adicional para a entrada/saída dos dados. Note que nesta versão parte da funcionalidade do bloco 3 foi repetida no bloco 4: a mensagem candidata original $\mathbf{u}_0$ é recebida como entrada e as demais mensagens são geradas usando-se portas XOR, o que originalmente é função do bloco 3. Visando analisar passo a passo o processo interno, o conteúdo da matriz de mensagens candidatas e da matriz $\mathbf{G}_r$ também foram conectadas a pinos de saída do chip. A versão final do layout do bloco 4, no padframe do chip, é mostrada na Figura 7.26.

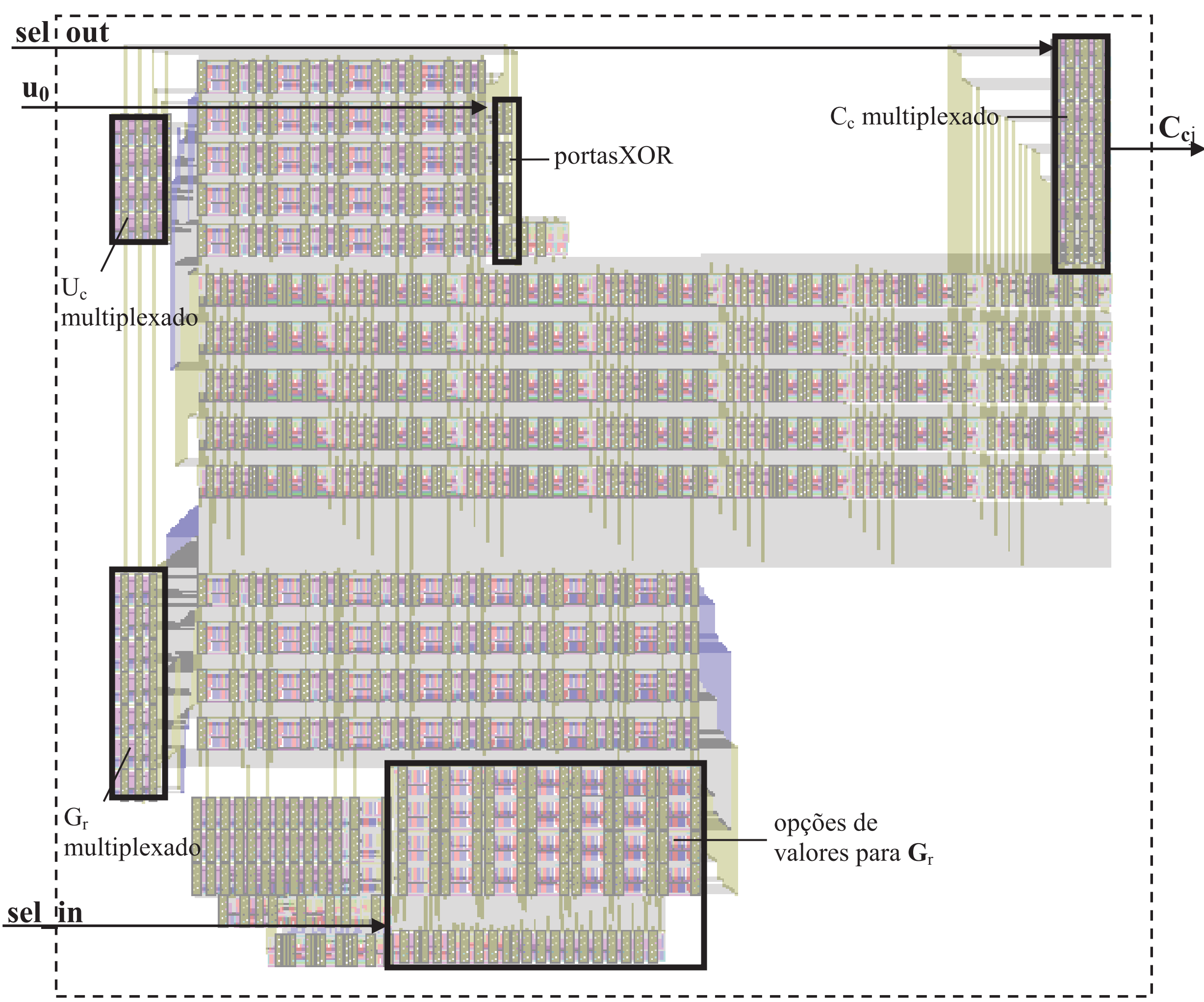


Figura 7.25 - Layout do bloco de geração de palavras-código candidatas com circuitos adicionais de entrada/saída.

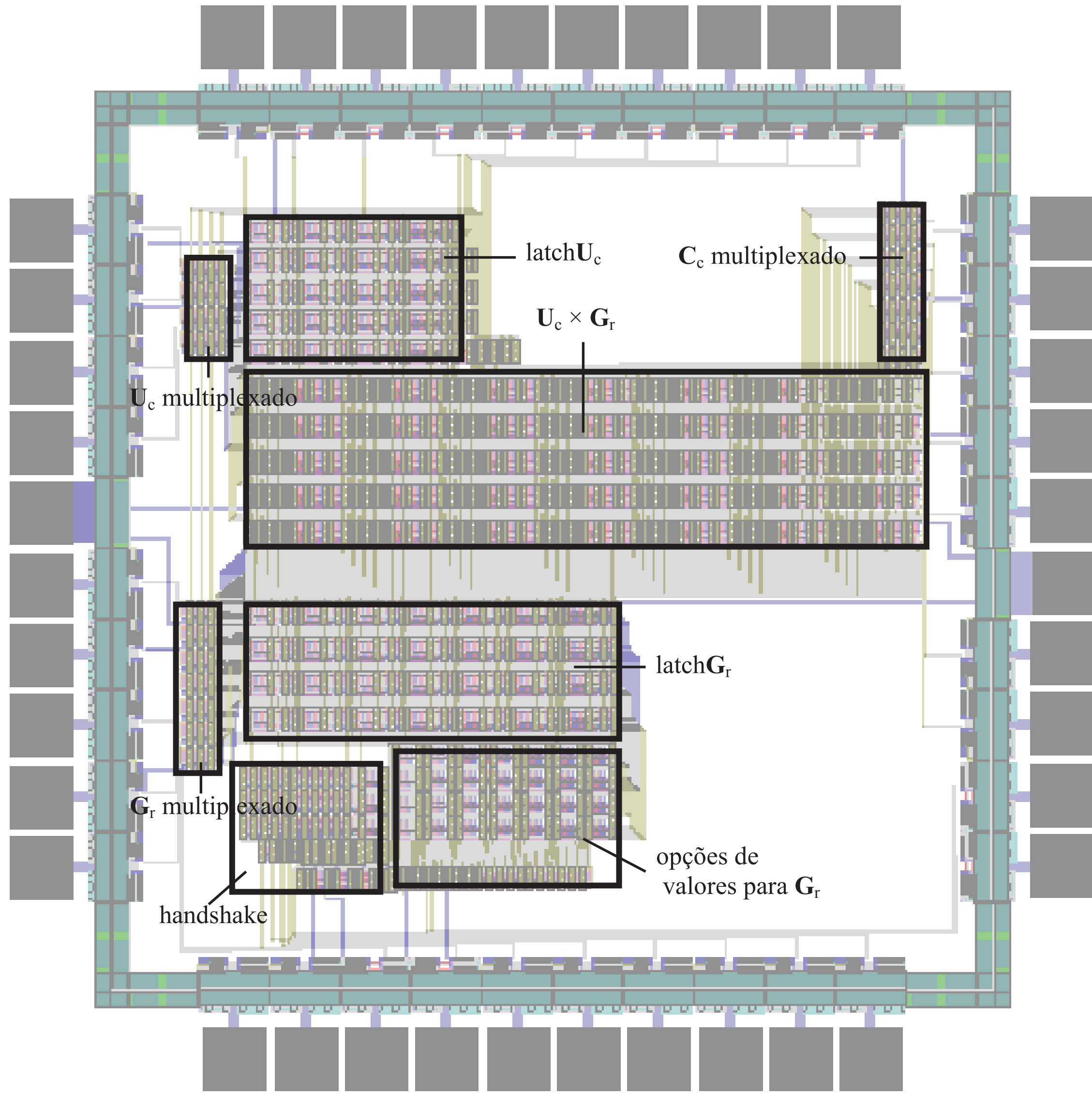


Figura 7.26 - Layout final do chip contendo o bloco de geração de palavras-código, fabricado com o objetivo de testar o bloco individualmente.

7.7.2 Resultados

A Figura 7.27 refere-se à simulação do esquemático criado com diagrama de blocos. Observe que assim que os dados de entrada se tornam disponíveis (*inputReady*='1') é necessário apenas um ciclo de clock para a saída estar pronta (*outputReady*='1') .

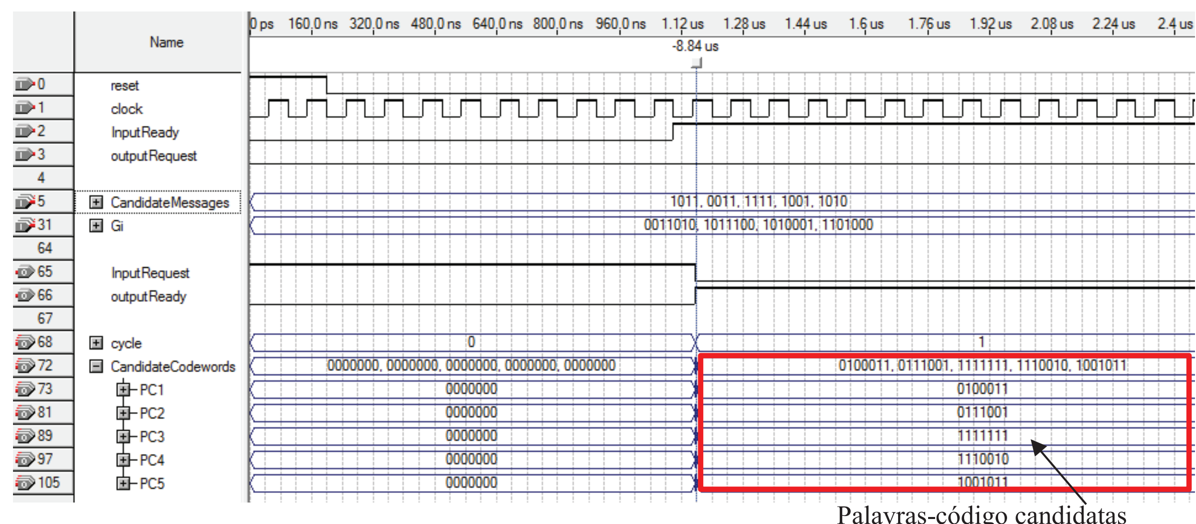


Figura 7.27 - Simulação do esquemático do gerador de palavras-código candidatas no Quartus II.

O resultado da implementação do código SPICE deste bloco é visto na Figura 7.28 e Figura 7.29. A simulação da Figura 7.28 foi realizada antes da criação do layout do chip. Nesta figura vê-se a recodificação da mensagem candidata 1011, que resulta na palavra-código candidata 0100011. A Figura 7.29 mostra a simulação pós-layout, com as capacitâncias parasitas e os pads do chip inclusos.

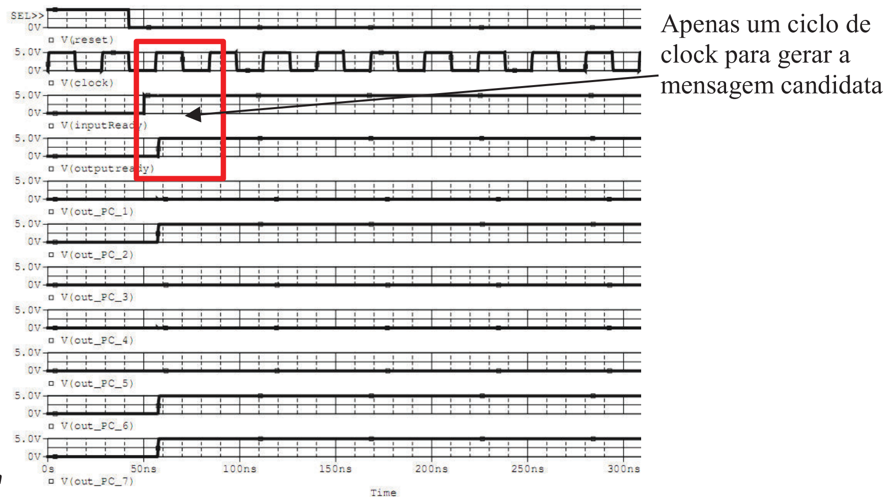


Figura 7.28 - Simulação SPICE do bloco de geração de palavras-código candidatas sem capacitâncias parasitas e sem pads.

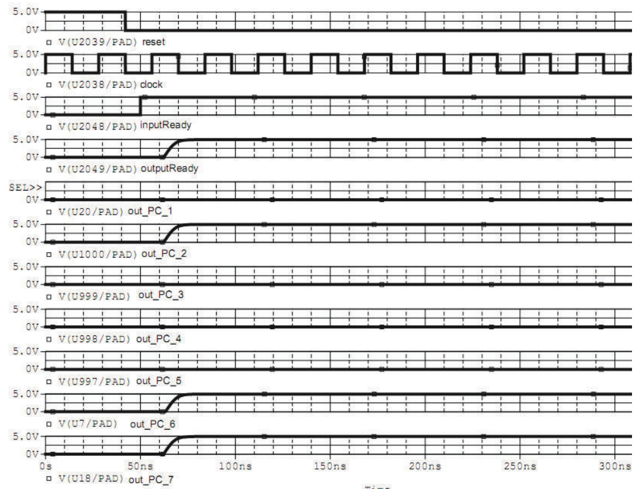


Figura 7.29 - Simulação SPICE do bloco de geração de palavras-código candidatas com capacitâncias parasitas e com pads.

Finalmente, após a fabricação do chip, os testes práticos foram realizados. A Figura 7.30 mostra os sinais referentes ao clock (rosa), *inputReady* (amarelo), *outputReady* (verde) e palavra-código de valor 0100011, capturados com o osciloscópio.

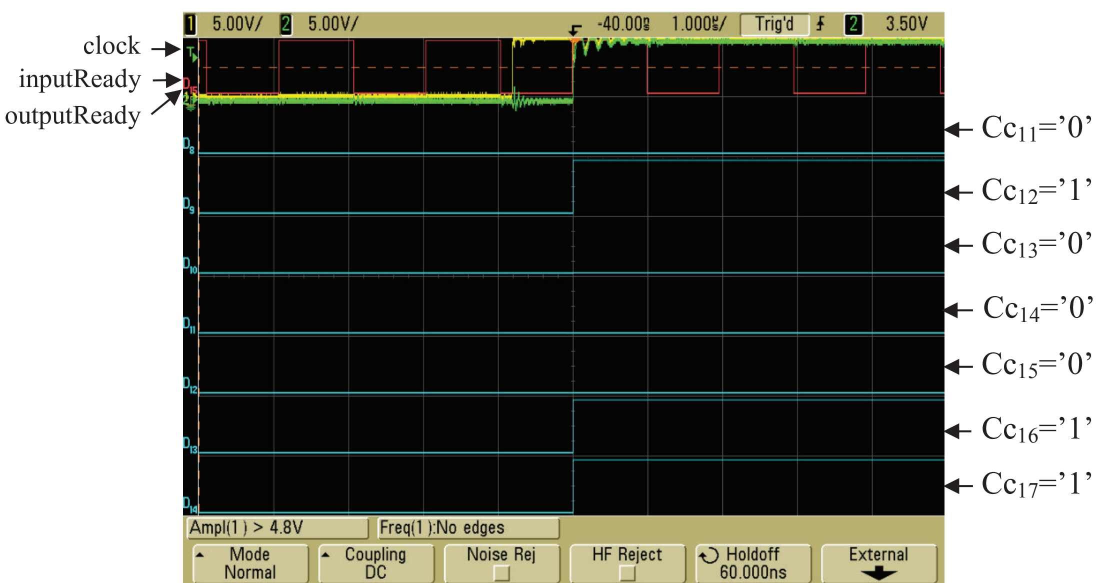


Figura 7.30 - Geração de palavras-código candidatas (osciloscópio).

7.8 BLOCO 5: SELETOR DA PALAVRA-CÓDIGO VENCEDORA

7.8.1 Projeto

O bloco 5 do decodificador avalia as k+1 palavras-código candidatas com o objetivo de selecionar a candidata mais próxima da palavra recebida, ou seja, a palavra-código com a menor distância suave em relação a **x**.

Nas simulações efetuadas no Quartus II e no PSPICE, foram executados testes exaustivos variando-se as entradas. Entretanto, para os testes práticos realizados no chip, foram utilizados apenas quatro valores diferentes para **x** e para a matriz $\mathbf{C_c}$, selecionados através de dois pinos de entrada do chip, conforme mostra a Tabela 7.9. Os resultados obtidos para cada entrada aplicada são comparados com os valores esperados, contidos na última coluna desta tabela.

Tabela 7.9 - Seleção dos dados de entrada para o bloco 5.

Entradas	Valor atribuído a x e $\mathbf{C_c}$	Saídas esperadas
sel_in = 0 0	**x** = 2 5 1 3 0 7 6	**c'** = 0100011
	$\mathbf{C_c}$ = 0100011, 0111001, 1111111, 1110010, 1001011	
sel_in = 0 1	**x** = 2 1 4 0 7 6 3	**c'** = 0010111
	$\mathbf{C_c}$ = 0010111, 0100011, 0101110, 0110100, 1000110	
sel_in = 1 0	**x** = 5 2 0 3 4 6 1	**c'** = 1000110
	$\mathbf{C_c}$ = 1000110, 0011010, 1001011, 0000000, 0101110	
sel_in = 1 1	**x** = 0 1 2 3 5 6 7	**c'** = 0010111
	$\mathbf{C_c}$ = 0010111, 0011010, 1001011, 0001101, 0100011	

A Figura 7.31 apresenta o layout original do bloco 5, seguindo o diagrama apresentado na Figura 5.24. A Figura 7.32 mostra o layout fabricado para verificação individual do seletor da palavra-código vencedora, incluindo os circuitos adicionais de entrada e saída. Na versão final do decodificador, apenas os blocos originais (sem circuitos auxiliares) são utilizados, o que reduz a área de silício necessária. A versão final do layout do bloco 5, no padframe do chip, é mostrada na Figura 7.33.

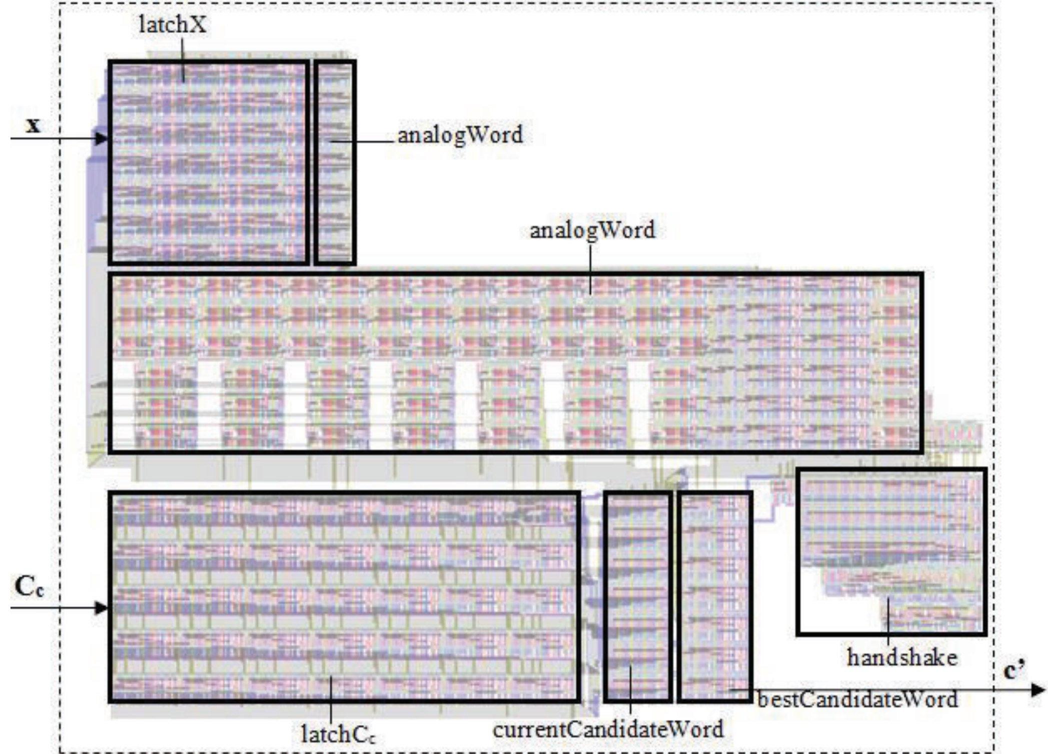


Figura 7.31 - Layout original do bloco de geração de palavras-código candidatas, composto pelos sub-blocos: latchX (armazena a palavra analógica); latchC_c (armazena a matriz de palavras-código candidatas); currentCandidateWord (seleciona uma palavra-código candidata); softDistanceCalculation (calcula a distancia suave entre a palavra-código candidata e a palavra analógica); bestCandidateWord (armazena a palavra-código vencedora; e handshake.

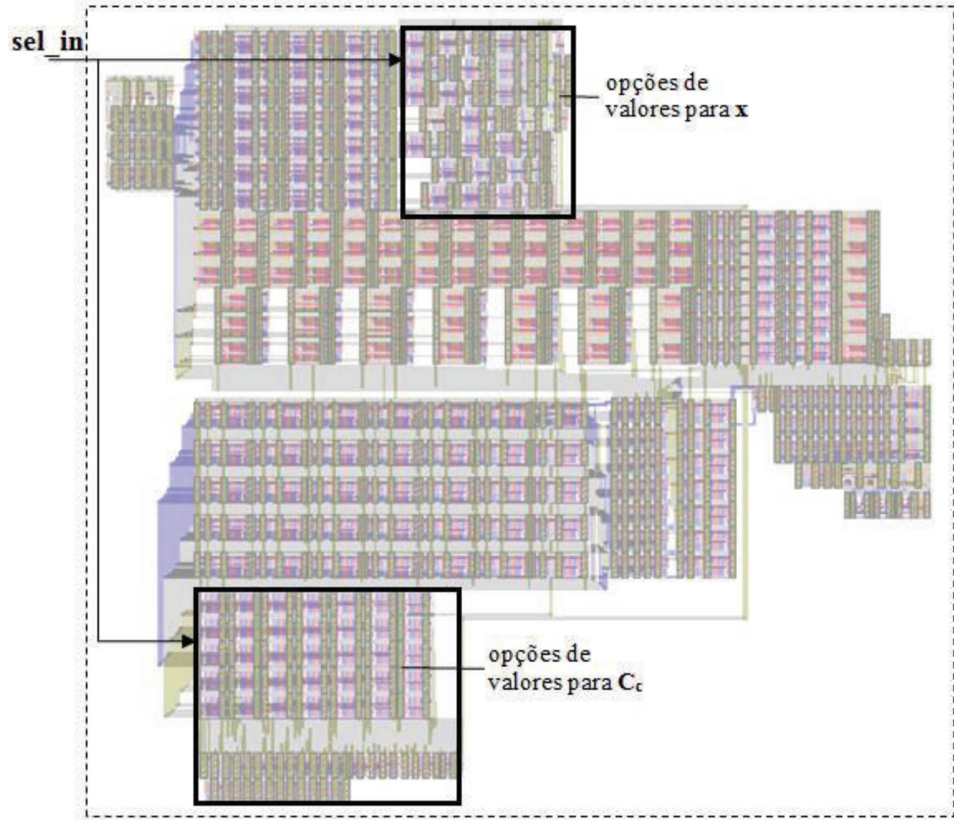


Figura 7.32 - Layout do bloco de seleção de palavra-código vencedora com os circuitos adicionais de entrada/saída,

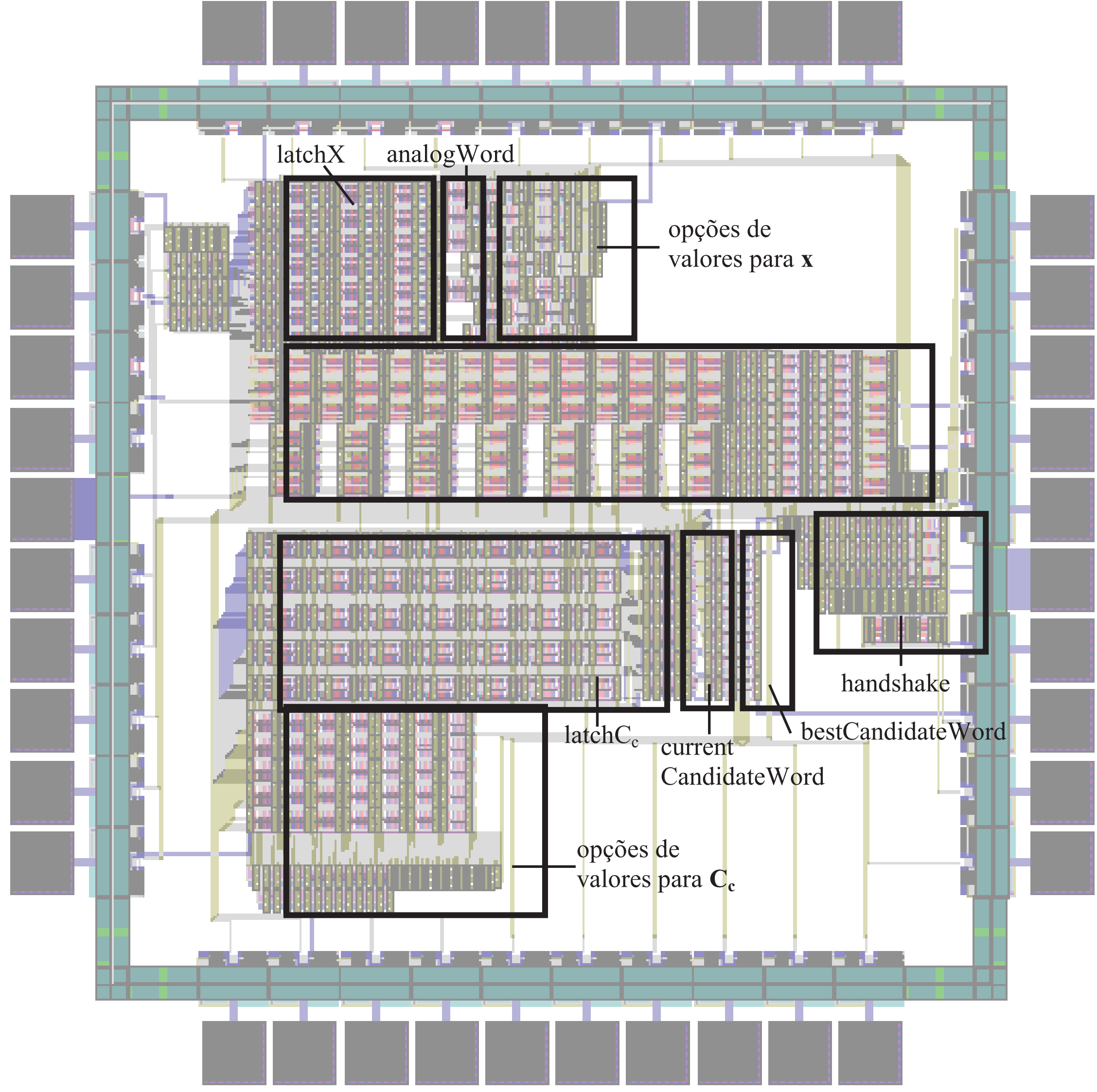


Figura 7.33 - Layout final do chip contendo o bloco de seleção de palavra-código, fabricado com o objetivo de testar o bloco individualmente.

7.8.2 Resultados

A simulação do esquemático do bloco 5 é mostrada na Figura 7.34. Note que o processamento completo ocorre em cinco ciclos de clock. Para **x** = 2 5 1 3 0 7 6, mostrado nesta simulação, a primeira palavra-código analisada é a melhor candidata.

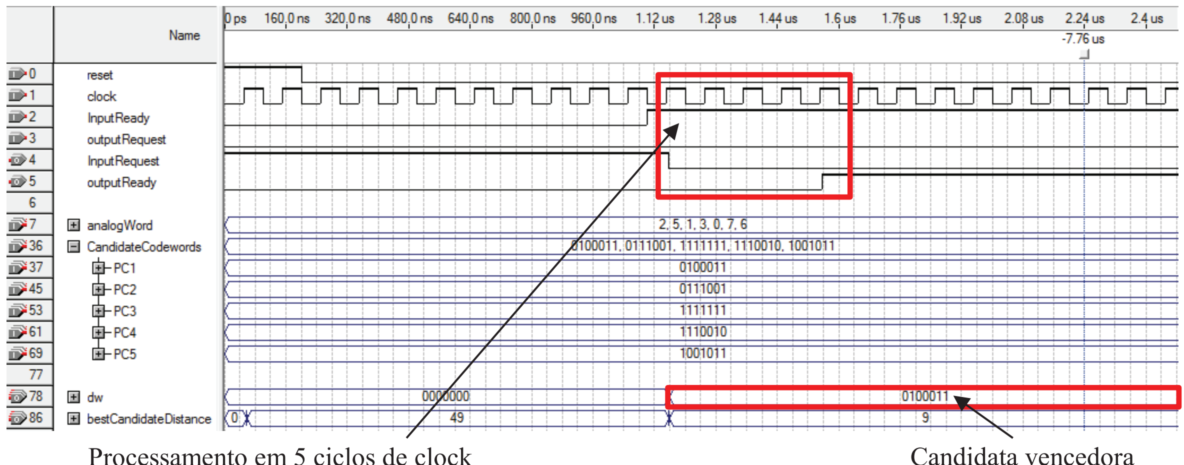


Figura 7.34 - Simulação do esquemático do seletor da palavra-código vencedora.

O mesmo esquemático, agora em nível de transistores, foi implementado em linguagem SPICE. O resultado da simulação pré-layout é mostrado na Figura 7.35. A simulação pós-layout do circuito completo, incluindo os adicionais de entrada/saída de dados, é visto na Figura 7.36.

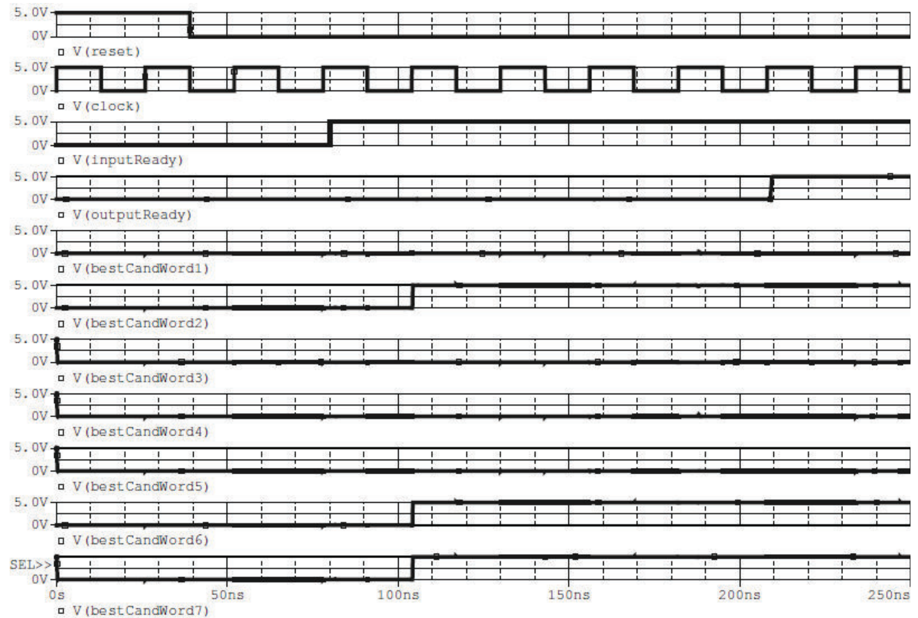


Figura 7.35 - Simulação SPICE do seletor da palavra-código vencedora sem capacitâncias parasitas e sem pads.

Finalmente, os resultados dos testes realizados após a fabricação do chip são apresentados na Figura 7.37. Os sinais digitais em azul mostram a saída do decodificador, ou seja, a palavra-código vencedora, que neste caso é 0100011. Os demais sinais presentes na figura são: clock (rosa), *inputReady* (amarelo) e *outputReady* (verde).

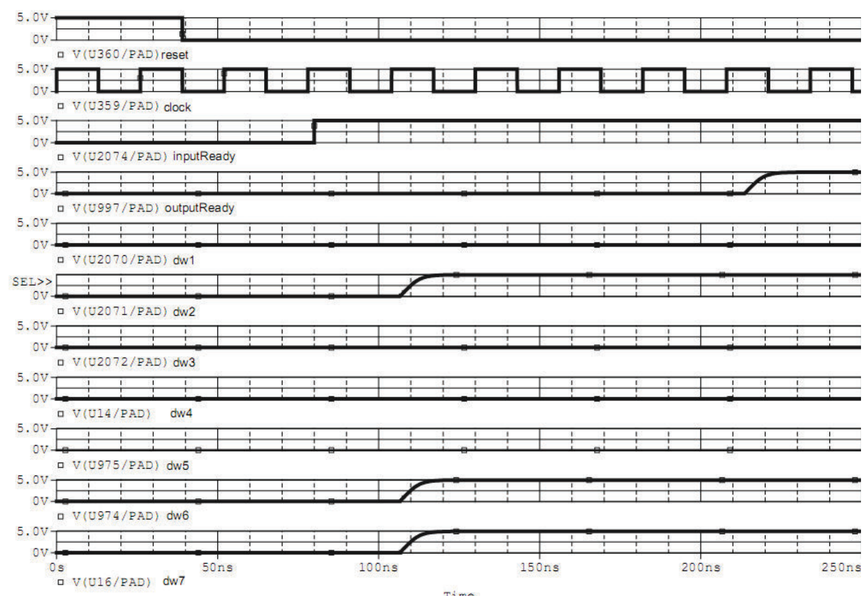


Figura 7.36 - Simulação SPICE do seletor da palavra-código vencedora com capacitâncias parasitas e com pads.

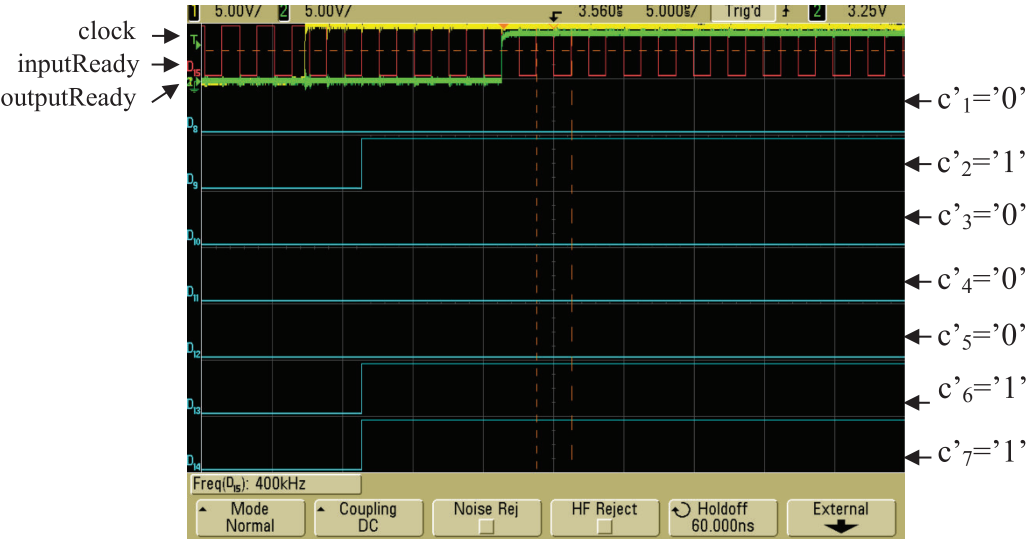


Figura 7.37 - Seleção da palavra-código vencedora (osciloscópio).

7.9 COLOCAÇÃO E INTERLIGAÇÃO DOS BLOCOS NO FRAME DO CHIP

7.9.1 Projeto

Após a validação de cada um dos blocos, através de simulações pré e pós-layout e testes práticos, o projeto completo do decodificador foi implementado em um único chip (Figura 7.38).

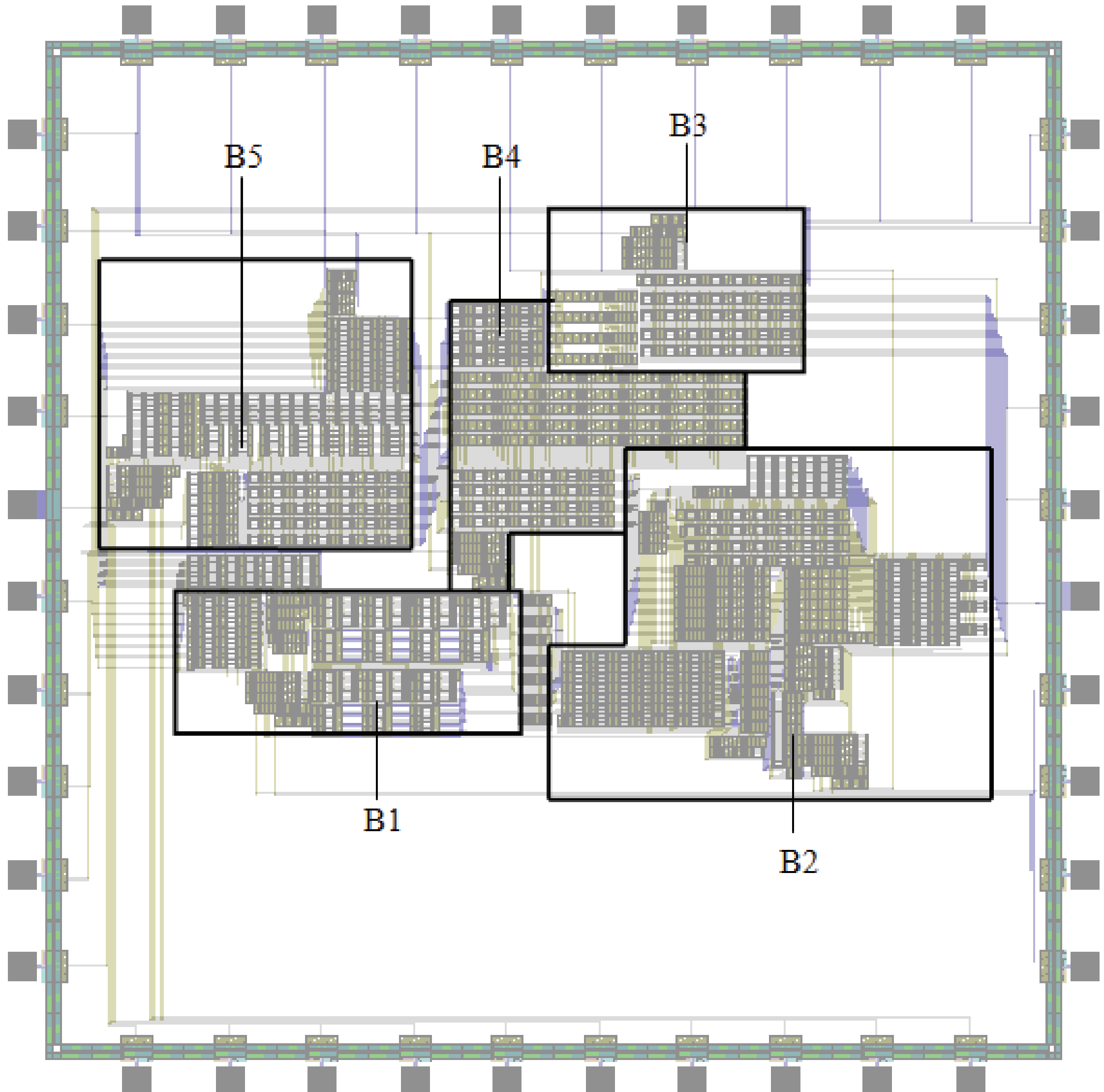


Figura 7.38 - Layout final do chip contendo o decodificador completo, isto é, os cinco blocos interconectados.

Neste chip, a palavra analógica é recebida pelo decodificador serialmente. Para isso, foram utilizados três *shift-registers* de sete bits. Assim, um por vez, os símbolos de **x** são recebidos, registrados, e os valores deslocados uma posição para a direita, até que **x** tenha sido completamente armazenado. O sinal de relógio utilizado nos *shift-registers* corresponde a uma entrada externa do decodificador, e é distinto do clock utilizado no processo de decodificação. Após o registro da palavra analógica, o sinal *inputReady* assume o valor ‘1’ e a decodificação é iniciada.

A criação do layout completo do decodificador foi uma tarefa bastante trabalhosa, já que a ferramenta utilizada (Ledit da Tanner) não possui muitos recursos, o que torna o design de inteira responsabilidade do projetista. A área de silício utilizada no layout dos cinco blocos foi 2.7mm×1.6mm. Entretanto, a área total do chip, incluindo o padframe, foi de

3.2mm×3.2mm. A tecnologia do chip fabricado é SCN3M_SUBM 0.5µm. O encapsulamento utilizado foi o DIP40. A lista completa dos pinos do chip consta na Tabela 7.10. Entre as saídas disponibilizadas pelo decodificador estão o resultado da decisão abrupta, o da decisão suave, e os sinais de *handshake* que controlam a sequência de operação dos blocos. Além disso, alguns pinos foram dedicados à verificação do valor de entrada registrado, permitindo visualizar o valor contido nos *shift-registers*.

Tabela 7.10 - Definição dos pinos.

Nome	**Número**	**Tipo**	**Descrição**
Reset	1	Entrada	Reset do decodificador. Ativo em nível lógico 1
Clock	2	Entrada	Sinal de relógio que sincroniza o decodificador
r1	3	Saída	Palavra demodulada por decisão abrupta, bit 1
r2	4	Saída	Palavra demodulada por decisão abrupta, bit 2
r3	5	Saída	Palavra demodulada por decisão abrupta, bit 3
r4	6	Saída	Palavra demodulada por decisão abrupta, bit 4
r5	7	Saída	Palavra demodulada por decisão abrupta, bit 5
r6	8	Saída	Palavra demodulada por decisão abrupta, bit 6
r7	9	Saída	Palavra demodulada por decisão abrupta, bit 7
inputReadyB3	10	Saída	*InputReady* do bloco 3
outputRequestB3	11	Saída	*outputRequest* do bloco 3
inputReadyB4	12	Saída	*InputReady* do bloco 4
outputRequestB4	13	Saída	*outputRequest* do bloco 4
out_x(1)	14	Saída	Valor de **x** armazenado, bit 1 (MSB)
out_x(2)	15	Saída	Valor de **x** armazenado, bit 2
out_x(3)	16	Saída	Valor de **x** armazenado, bit 3 (LSB)
sel_out(1)	17	Entrada	Seleção do símbolo de **x** armazenado, bit 1 (MSB)
sel_out(2)	18	Entrada	Seleção do símbolo de **x** armazenado, bit 2
sel_out(3)	19	Entrada	Seleção do símbolo de **x** armazenado, bit 3 (LSB)
GND	20		Ground digital
in_x(1)	21	Entrada	Palavra analógica digitalizada, bit 1 (MSB)
in_x(2)	22	Entrada	Palavra analógica digitalizada, bit 2
in_x(3)	23	Entrada	Palavra analógica digitalizada, bit 3 (LSB)
clock_shift	24	Entrada	Sinal de relógio para o *shift-register* de entrada

c'(1)	25	Saída	Palavra decodificada, bit 1
c'(2)	26	Saída	Palavra decodificada, bit 2
c'(3)	27	Saída	Palavra decodificada, bit 3
c'(4)	28	Saída	Palavra decodificada, bit 4
c'(5)	29	Saída	Palavra decodificada, bit 5
c'(6)	30	Saída	Palavra decodificada, bit 6
c'(7)	31	Saída	Palavra decodificada, bit 7
outputReadyB5	32	Saída	*outputReady* do bloco 5
inputRequestB5	33	Saída	*inputRequest* do bloco 5
inputReadyB5	34	Saída	*inputReady* do bloco 5
outputRequestB5	35	Saída	*outputRequest* do bloco 5
inputReadyB1	36	Entrada	*inputReady* do bloco 1
outputRequestB1	37	Entrada	*outputRequest* do bloco 1
inputReadyB2	38	Saída	*inputReady* do bloco 2
outputRequestB2	39	Saída	*outputRequest* do bloco 2
VDD	40		Alimentação digital

7.9.2 Resultados

Seguindo os mesmos passos executados para os blocos individuais, inicialmente o circuito esquemático do decodificador foi implementado e simulado (Figura 7.39). Note que após o sinal indicando que as entradas estão prontas (*inputReady*='1'), são necessários vinte ciclos de clock para a decodificação. Na figura, **r** representa o resultado da decisão abrupta da decodificação e **c'** a decisão suave (saída do decodificador).

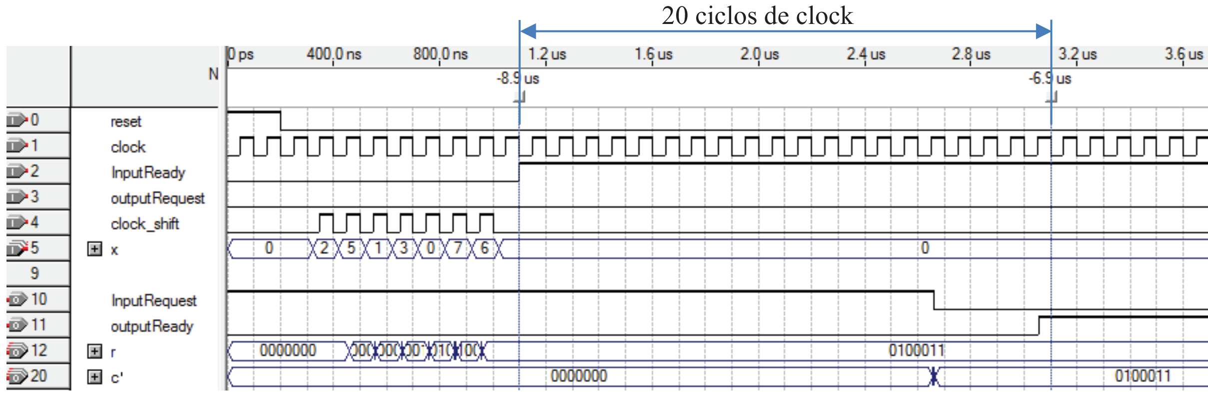


Figura 7.39-Simulação do decodificador na ferramenta Quartus II.

O passo seguinte foi a implementação do código SPICE, com todos os blocos já existentes conectados, como representado na Figura 5.1.

Simulações pós-layout, realizadas na ferramenta PSPICE, são mostradas nas Figuras 8.41 e 8.42. A frequência máxima de operação nesta tecnologia é de aproximadamente 21 MHz, como mostra a Tabela 7.11. À medida que tecnologias mais atuais são utilizadas, maior é a frequência de operação obtida. Simulações realizadas utilizando parâmetros da tecnologia IBM 7RF (180 nm) indicam uma frequência máxima de 74 MHz. Para a tecnologia IBM 8RF (130 nm), a frequência aumenta um pouco mais, chegando a 100 MHz. A Tabela 7.11 também apresenta a frequência máxima de operação individual para cada um dos blocos do decodificador.

Tabela 7.11 - Frequência máxima de operação (MHz).

	Bloco 1	Bloco 2	Bloco 3	Bloco 4	Bloco 5	Decodificador Completo
0.5 µm	39	40	3300	67	100	21
0.18 µm	87	133	1100	133	333	74
0.13 µm	154	200	1200	333	500	100

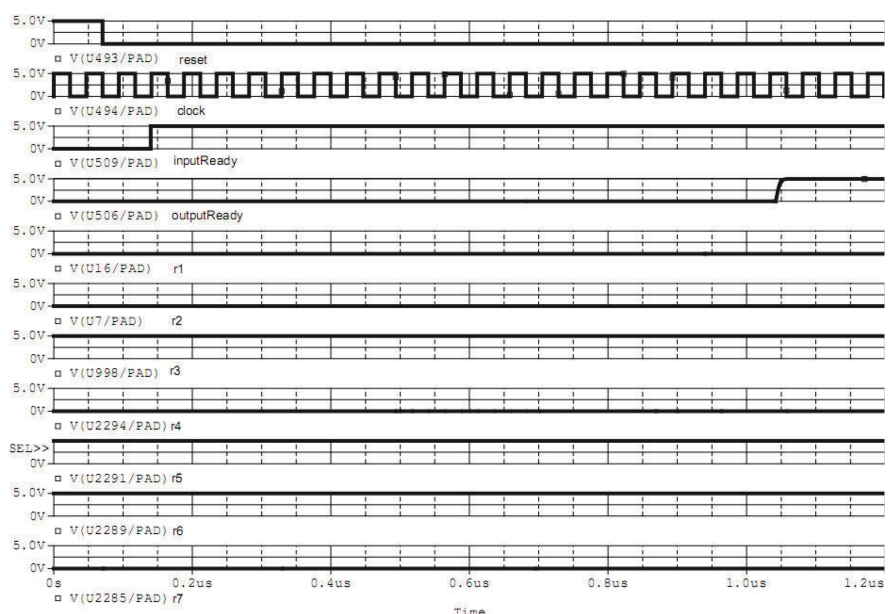


Figura 7.40 - Saída do decodificador: decisão abrupta (sem inclusão de capacitâncias parasitas e pads).

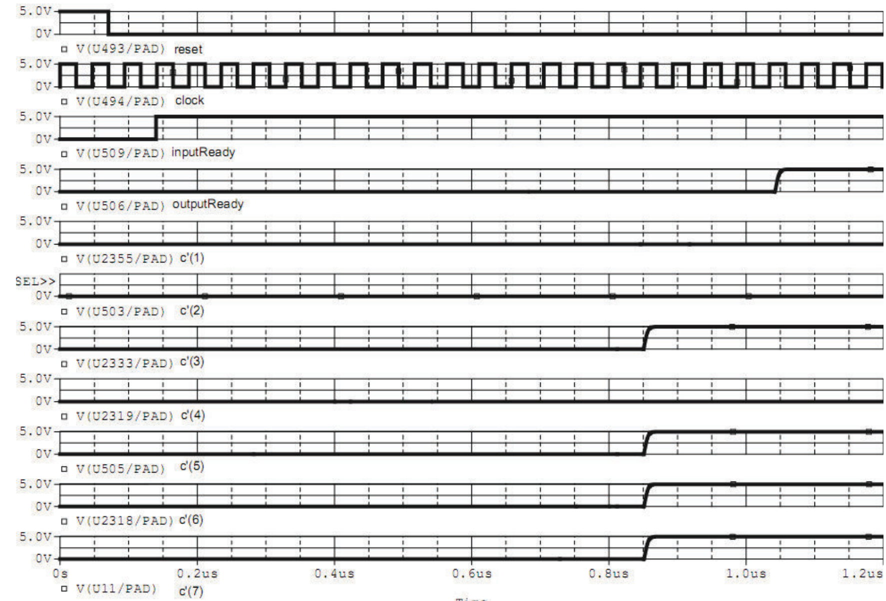


Figura 7.41 - Saída do decodificador: decisão suave (sem inclusão de capacitâncias parasitas e pads).

A Figura 7.42 mostra os sinais de controle entre os blocos do controlador capturados através de um osciloscópio durante o teste prático do chip. Os primeiros sinais (parte superior da figura) são o clock e o *inputReady* do decodificador. Em seguida, vê-se os sinais de *outputReady* de cada um dos cinco blocos. A Figura 7.43 apresenta o resultado da decodificação com decisão suave para uma palavra analógica “1111111” (mensagem original, antes da codificação, igual a “1111”). Como o código está organizado de maneira sistemática, os primeiros *k* bits correspondem à informação e os demais *n-k* bits à redundância. Portanto, a mensagem recuperada pelo decodificador é “1111”.

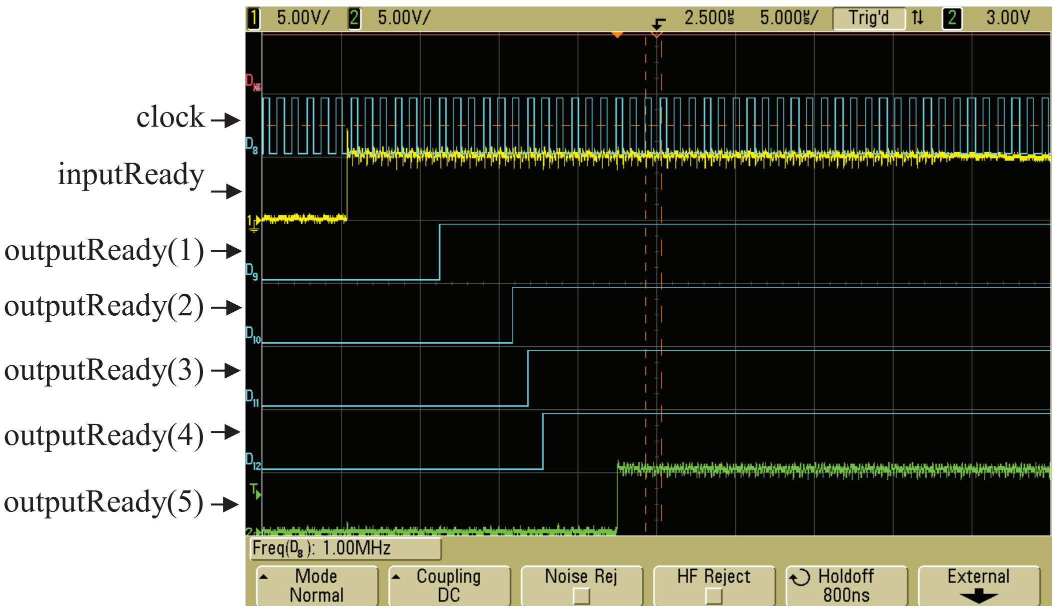


Figura 7.42 - Sinais de controle entre os blocos do decodificador.

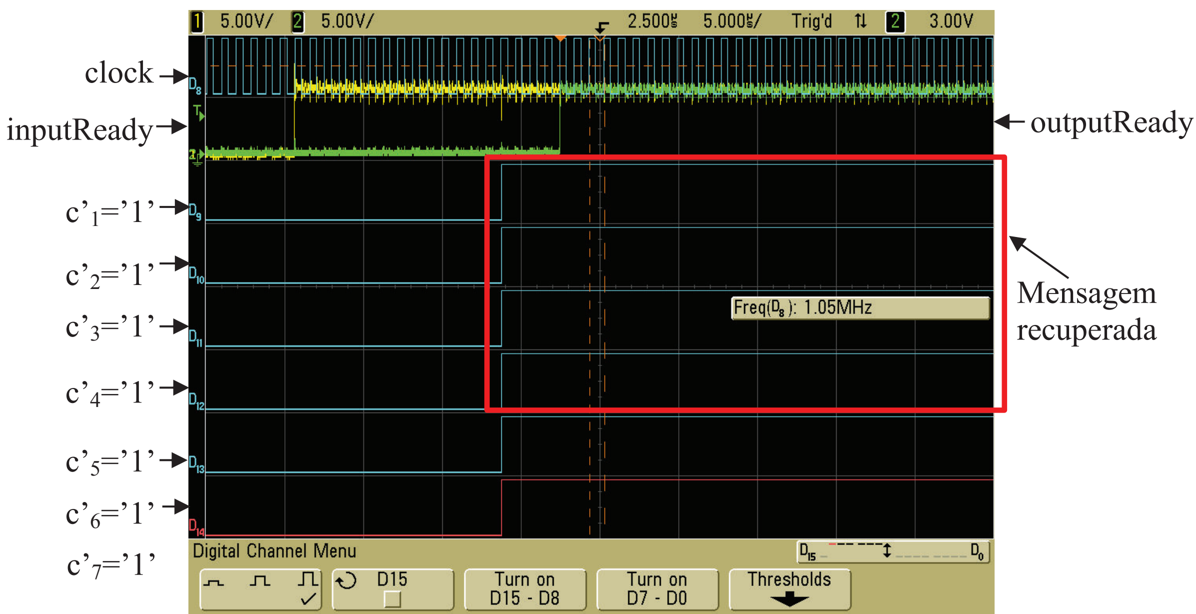


Figura 7.43 - Saída do decodificador: decisão suave (osciloscópio).

CAPÍTULO 8

IMPLEMENTAÇÃO DO HARDWARE EM ASIC UTILIZANDO VHDL E RESULTADOS

8.1 INTRODUÇÃO

Este capítulo descreve a terceira implementação realizada para o decodificador, utilizando ferramentas de layout e roteamento automatizadas (da Cadence), a partir do código VHDL.

A migração das ferramentas da Tanner para as da Cadence para o design do decodificador foi motivada por várias razões. A principal foi o interesse em implementar o decodificador seguindo um fluxo de projeto real, similar àquele realizado na indústria. Para isso foi necessário aprender a trabalhar com ferramentas poderosas de padrão industrial que são utilizadas para o desenvolvimento de circuitos integrados dedicados de forma mais automatizada, onde o código VHDL é utilizado na criação do layout. As ferramentas da Cadence são complexas e demandam experiência para sua utilização. Até então, essas ferramentas nunca tinham sido utilizadas na UTFPR, por esse motivo, fez-se necessário um treinamento intensivo no Harvey Mudd College, realizado durante o programa de doutorado "sanduíche" da Capes, de janeiro a maio de 2012.

Além da motivação em aprender o fluxo real de desenvolvimento de circuitos integrados, o uso dessas ferramentas foi motivado pela possibilidade de projetar o decodificador em tecnologias mais atuais do que 0.5 μm, o que seria inviável com as ferramentas da Tanner por não haver pads disponíveis para utilização nas tecnologias de 180 nm ou 130 nm da IBM (cedidos à UTFPR pela MOSIS). Além disso, muitos arquivos das tecnologias IBM não têm seu conteúdo divulgado, especialmente na tecnologia de 130 nm, impossibilitando a construção dos pads na ferramenta Ledit da Tanner. Finalmente, o interesse em utilizar códigos mais longos, também motivou tal migração.

Na implementação manual (0.5 μm) foi utilizado um código C(7, 4, 3), pois um código maior requereria uma área de silício maior do que aquela disponível. A utilização de uma tecnologia mais recente permite que códigos maiores sejam usados. Na versão implementada com as ferramentas da Cadence, o código utilizado foi o Golay estendido C(24,12,8).

8.2 DESIGN DO CHIP: RTL-TO-GDSII

Os softwares da Cadence constituem poderosas ferramentas de padrão industrial. Entre as ferramentas utilizadas no desenvolvimento do circuito integrado estão o RTL Compiler, para síntese, o Encounter, para *place and route*, e o Virtuoso, para *full-custom layout*.

8.2.1 Síntese

O processo de design de um chip envolve várias etapas. A primeira consiste na implementação do projeto em linguagem de descrição de hardware. Esta foi a primeira versão criada do decodificador, implementado em VHDL e testado em uma FPGA Stratix III, utilizando-se a ferramenta Quartus II.

O próximo passo envolve a síntese do código VHDL utilizando uma ferramenta específica para projeto de chips. Neste processo, chamado de síntese *Logic/RTL*, o código VHDL é otimizado e convertido em uma lista de portas lógicas (*Gate Level Netlist*). A ferramenta utilizada para esta tarefa é o RTL Compiler. Essa ferramenta usa como entrada o arquivo de descrição de hardware RTL (*Register Transfer Level*) e uma biblioteca de células-padrão, produzindo como saída uma *netlist* em nível de portas lógicas. Restrições de tempo, área e consumo também podem ser consideradas, visando encontrar a melhor implementação possível no nível de portas lógicas.

A Figura 8.1 mostra a sequência de operações executadas no RTL Compiler, descritas através de um scrip (.tcl), mostrado na Figura 8.2. Primeiramente, são definidos parâmetros, como a biblioteca utilizada, lista dos arquivos VHDL, *toplevel* das entidades, linguagem de descrição de hardware utilizada, definições de clock, etc. Em seguida, os arquivos VHDL são lidos e verificados, buscando possíveis erros. Em seguida, o código é sintetizado e os esquemáticos são criados.

A Figura 8.3 mostra o resultado da síntese do código VHDL do decodificador. Esta figura apresenta a entidade *toplevel*, onde os cinco blocos são instanciados. Como exemplo, a Figura 8.4 mostra o esquemático gerado para o circuito de *handshake* de um dos blocos.

Após a síntese, a *netlist* é escrita (em Verilog) em um arquivo de saída. As restrições de tempo do circuito utilizadas também são colocadas em um arquivo de saída (.sdc), sendo utilizadas posteriormente no *place and route*. Finalmente, os relatórios de área, tempo e consumo são gerados.

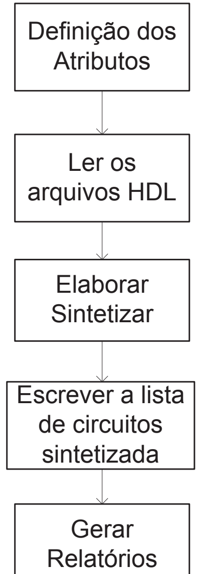


Figura 8.1 - Sequência de operações realizadas na ferramenta RTL Compiler.

```
###############################################################
# Script for Cadence RTL Compiler synthesys
# Sibilla Franca, 24-10-2012
##############################################################

# Set the search paths to libraries
set_attribute lib_search_path {./}
set_attribute library {PnomV180T025_STD_CELL_7RF.lib}
set_attribute information_level 6

setmyFiles
[listBooleanVectorMatrixArithmetic.vhdDecoderConfig.vhdConvert.vhdDecoderTypes.vhdHandsh
akeUnit.vhdSorterConfig.vhdTagSorter.vhd B1_SorterDemodulator.vhd
B2_GaussJordanElimination.vhd B3_CandidateMessagesGeneration.vhd
B4_CandidateCodewordsGeneration.vhd B5_BestCandidateSelection.vhd InfosetDecoder.vhd]
set basename InfosetDecoder

set myClk Clock                     ;# The name of your clock
set myPeriod_ps 10000              ;# desired clock period (in ps) (sets speed goal)
set myInDelay_ps 250              ;# delay from clock to inputs valid
set myOutDelay_ps 250             ;# delay from clock to output valid
set runname RTL

set myInputBuf INVERT_D          ;# name of cell driving the inputs
set myLoadLibrary PnomV180T025_STD_CELL_7RF         ;# name of library the cell comes
from
set myLoadPin A            ;# name of pin that the outputs drive

# Analyze and Elaborate the HDL files
read_hdl -vhdl ${myFiles}
elaborate ${basename}

#ApplyConstraintsandgenerateclocks
set clock [define_clock -period ${myPeriod_ps} -name ${myClk} [clock_ports]]
dc::set_driving_cell  -library $myLoadLibrary -lib_cell $myInputBuf  [all_inputs]
external_delay -input $myInDelay_ps -clock ${myClk} [find / -port ports_in/*]
external_delay -output $myOutDelay_ps -clock ${myClk} [find / -port ports_out/*]
dc::set_load -pin_load 0.004648 [all_outputs]

# sets transition to default values for Synopsys SDC format,
# fall/rise 400ps
dc::set_clock_transition .4 $myClk

#check that the design is OK so far
check_design -unresolved
```

```
report timing -lint

# Synthesize the design to the target library
synthesize -to_mapped

# Write out thereports
report timing >  ${basename}_${runname}_timing.rep
report gates >  ${basename}_${runname}_cell.rep
report power >  ${basename}_${runname}_power.rep

# Write out the structural Verilog and sdc files
write_hdl -mapped > ${basename}_${runname}.v
write_sdc> ${basename}_${runname}.sdc
```

Figura 8.2 - Script utilizado no RTL Compiler para síntese do decodificador na tecnologia de 180 nm.

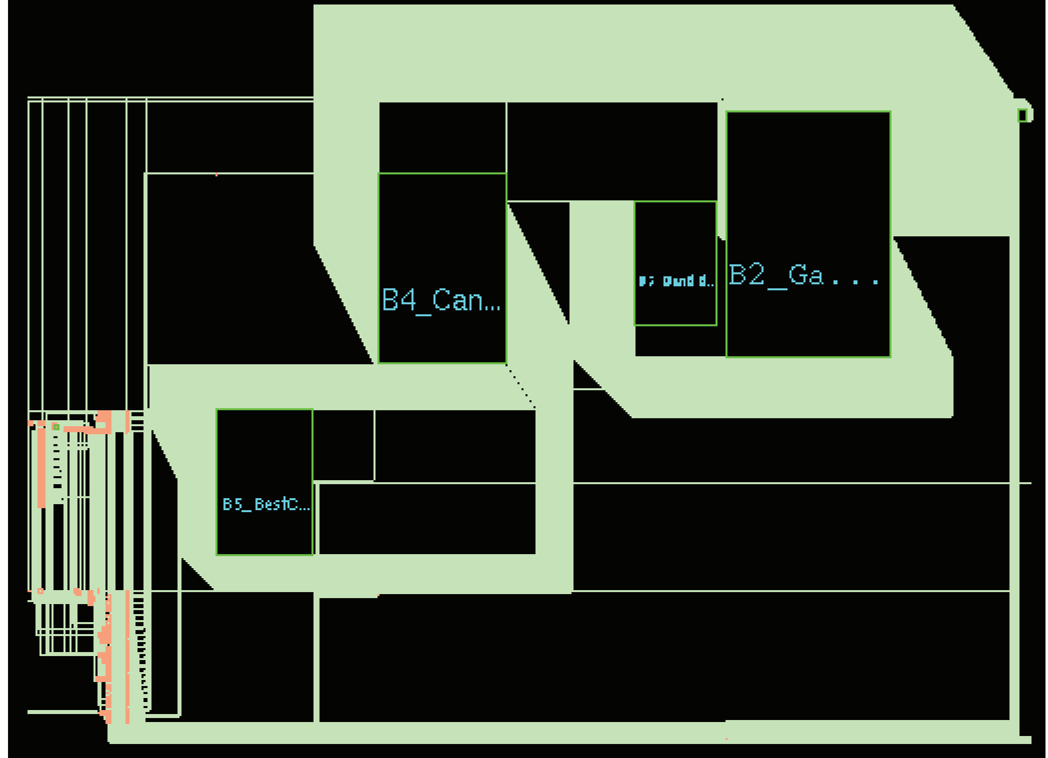


Figura 8.3 - Esquemático completo do decodificador gerado no RTL Compiler.

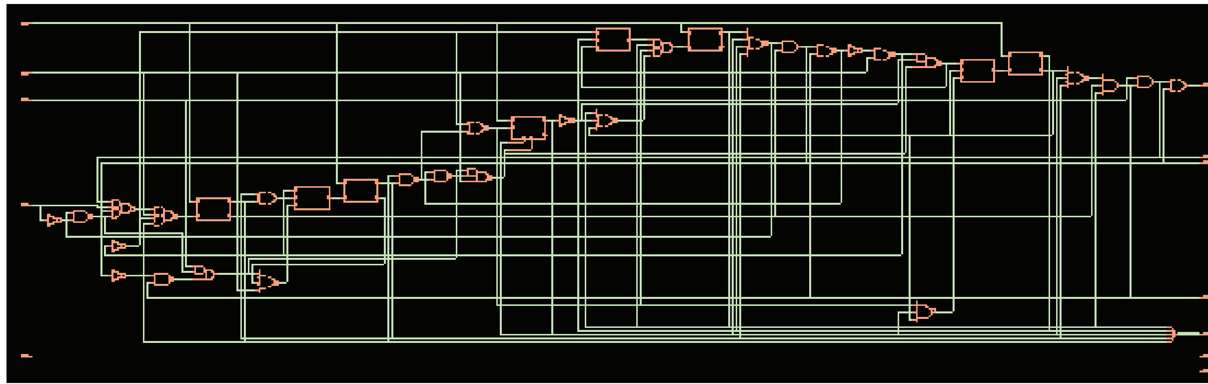


Figura 8.4 - Esquemático do circuito de handshake gerado no RTL Compiler.

8.2.2 Place and Route

No passo seguinte, a ferramenta utilizada é o Encounter (da Cadence), a qual realiza o *place and route* e a inserção do *padframe*. Para tal, esta ferramenta requer como entrada os arquivos gerados na etapa anterior, as especificações de tecnologia e a biblioteca de células. Além disso, são necessárias as especificações referentes aos pads e o arquivo que relaciona os sinais de entrada/saída do decodificador, ou seja, *netlist* com os pads. A Figura 8.5 mostra a sequência de operações realizadas por esta ferramenta.

Nesta etapa são definidos parâmetros como área, margem entre o *padframe* e o *core*, sinais de alimentação, espessura dos anéis de alimentação, etc. Após a configuração desses parâmetros, as células-padrão são inseridas e o roteamento é executado.

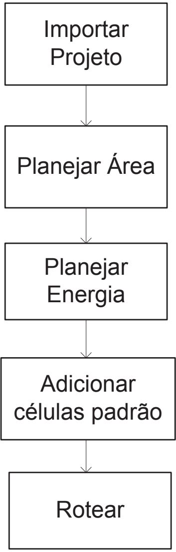


Figura 8.5 - Sequência de operações realizadas na ferramenta Encounter.

O design resultante, na tecnologia IBM 7RF (180 nm), é apresentado na Figura 8.6. Note que os cantos também são adicionados, além de alguns retângulos de preenchimento para que o anel de alimentação não seja interrompido.

O principal problema encontrado neste design está relacionado aos pads de alimentação. Diferentemente de outras tecnologias, nesta existem apenas pads de V_{DD} e GND programáveis, definidos como pinos de saída. Na tecnologia IBM 8RF (130 nm) ocorreram problemas com as células padrão *arm* disponibilizadas pelo fabricante, originando diversos erros no layout. A documentação disponível é bastante limitada. Inúmeras tentativas para solucionar esse problema foram realizadas; entretanto, nenhuma resolveu satisfatoriamente o problema.

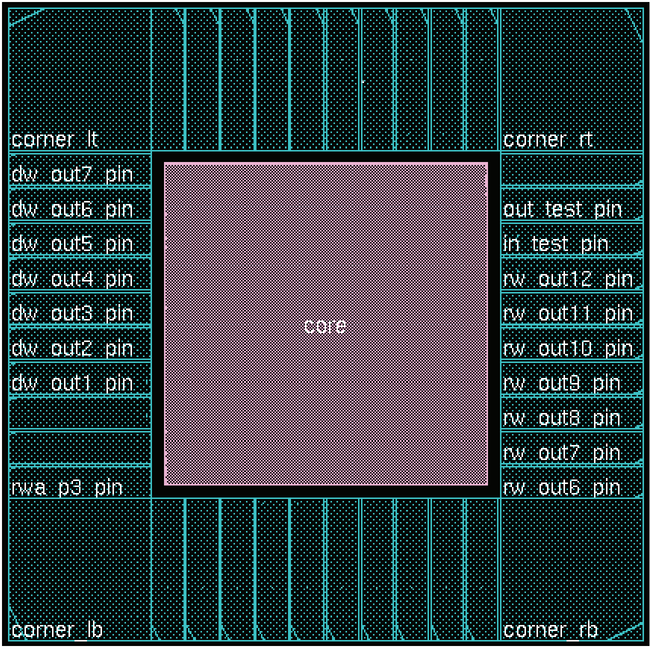


Figura 8.6 - Design do decodificador gerado pela ferramenta Encounter (da Cadence).

8.2.3 GDSII

O último passo no processo de design é realizado na ferramenta Virtuoso (da Cadence). Nela é inserido o *guardring,* que delimita a área do layout, e gerado o GDSII, arquivo enviado para fabricação (Figura 8.7). Devido aos problemas mencionados anteriormente, provavelmente o chip apresentaria problemas após a fabricação. Sendo assim, optou-se por fabricar o decodificador apenas na tecnologia 0.5 µm, visto que a cota de fabricação provida pela MOSIS consiste em apenas dois projetos em 0.5 ou 0.18 µm por semestre e um projeto de 0.5 ou 0.13 µm por ano.

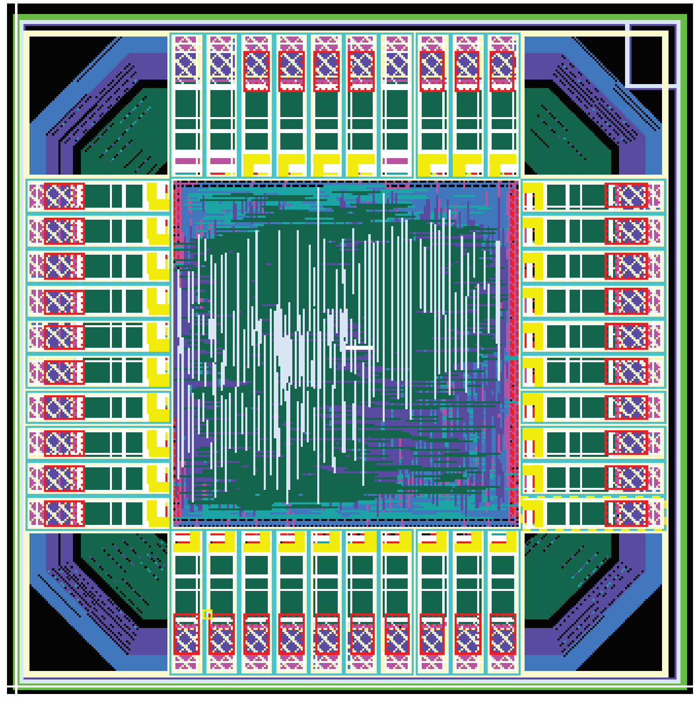


Figura 8.7 - Layout final do decodificador gerado pela ferramenta Virtuoso.

8.3 RESULTADOS

Além das simulações exaustivas realizadas previamente no projeto do decodificador, através de *testbenches,* foram executadas simulações após a síntese do RTL Compiler. Estas simulações pós-síntese têm o objetivo de verificar se a combinação das portas lógicas e registradores gerados com base na biblioteca da tecnologia apresentam as saídas esperadas.

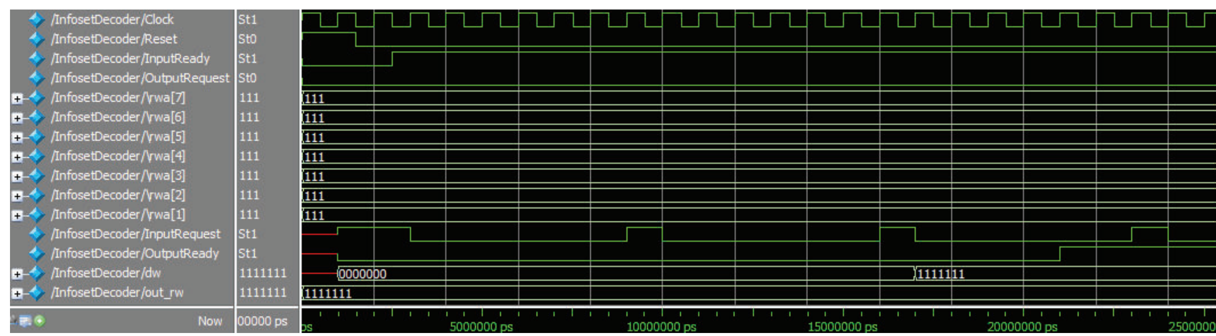


Figura 8.8 - Simulação do decodificador após a síntese no RTL Compile.

As Tabelas 9.1 e 9.2 mostram um resumo dos projetos do decodificador nas tecnologias IBM 7RF (180 nm) e 8RF (130 nm), após a síntese no RTL Compiler. Como era de se esperar, o número de células lógicas (e consequentemente a área empregada) aumenta

significativamente à medida que o tamanho do código processado pelo decodificador aumenta.

Tabela 8.1 - Resumo pós-síntese do projeto do decodificador na tecnologia IBM 7RF.

	C(7, 4, 3)	C(15, 7, 5)	C(24,12, 8)	(48, 24, 12)
Número de células lógicas	1494	5466	15889	89922
Área das células lógicas (μm^2)	43634	153365	440968	2237949
Potência de fuga (μW)	0,5	1,82	5,79	28,20
Potência dinâmica (μW)	7509,67	37034,63	151826,06	812000,31
Potência Total (μW)	7510,17	37036,46	151831,86	812028,51

Tabela 8.2 - Resumo pós-síntese do projeto do decodificador na tecnologia IBM 8RF.

	C(7, 4)	C(15, 7)	C(24,12)	(48, 24)
Número de células lógicas	1347	5152	15560	86025
Área das células lógicas (μm^2)	16991	59901	170770	838312
Potência de fuga (nW)	0,22	0,78	2,32	9,46
Potência dinâmica (nW)	1208,35	4059,27	13559,38	66212,21
Potência Total (nW)	1208,57	4060,05	13561,71	66221,67

Apesar de não ter sido possível fabricar o chip aqui projetado, a execução das tarefas descritas neste capítulo foi de suma importância, por vários motivos. Em primeiro lugar, foi possível aprender um conjunto de ferramentas que são de fato utilizadas na indústria de dispositivos semicondutores integrados, as quais são complexas e exigiram um grande esforço, desde sua correta instalação, passando pelo aprendizado e, finalmente, pela utilização efetiva das mesmas. Em segundo lugar, foi possível desenvolver as simulações pertinentes e comparar com resultados obtidos em simulações anteriores. Finalmente, essas tarefas levaram ao desenvolvimento da etapa “sanduíche” do doutorado junto ao grupo do Prof. David M. Harris (HMC, Estados Unidos), o qual atua intensamente nesta área e é autor de um dos principais livros textos sobre o assunto, sendo este também usuário das ferramentas da Cadence, permitindo assim entender e aprender detalhes importantes sobre todas as fases desse tipo de projeto.

CAPÍTULO 9

CONCLUSÕES E TRABALHOS FUTUROS

9.1 CONCLUSÕES

O decodificador apresentado nessa tese é resultado de um amplo projeto de pesquisa, desenvolvido através de uma parceria entre o Grupo de Microeletrônica e o Grupo de Comunicação de Dados da Universidade Tecnológica Federal do Paraná. O algoritmo de decodificação, baseado em conjuntos de informação, foi criado pelo grupo e tem desempenho muito próximo ao MLD, com a vantagem adicional de usar apenas um número reduzido de comparações. Assim, embora seja um algoritmo de decodificação suave (o que aumenta a qualidade do código, porém aumenta também a complexidade dos cálculos), as otimizações realizadas no algoritmo permitiram que sua implementação se tornasse viável em hardware.

Este projeto envolveu todas as etapas do desenvolvimento de um decodificador, desde a elaboração do algoritmo até sua implementação em hardware. Três implementações distintas foram realizadas para o decodificador por conjuntos de informação. A primeira, em FPGA, foi uma implementação preliminar, utilizada como base para as demais implementações. A segunda, uma implementação manual do hardware em ASIC. A terceira, uma implementação automatizada do hardware em ASIC utilizando VHDL.

No desenvolvimento dessa tese foram fabricados cinco chips dedicados, na tecnologia de 0.5 μm. Os quatro primeiros chips continham blocos individuais do decodificador, operando de maneira independente. O último chip contém o decodificador completo. A área deste último chip foi de 3.2mm×3.2mm, e sua frequência máxima de operação foi de 21 MHz. Embora o valor da frequência não pareça tão elevado, as simulações realizadas mostram que à medida que tecnologias mais atuais são utilizadas, é possível atingir frequências de operação mais altas. Por exemplo, as simulações utilizando parâmetros da tecnologia de 180 nm indicam uma frequência máxima de 74 MHz. Para a tecnologia de 130 nm, a frequência chega a 100 MHz.

Além dos chips fabricados na tecnologia de 0.5 μm, implementados de maneira manual, outros dois chips dedicados foram projetados, um na tecnologia de 180 nm (IBM 7RF) e outro na tecnologia de 130 nm (IBM 8RF), ambos utilizando as ferramentas da

Cadence. O uso dessas ferramentas foi motivado pelo interesse em implementar o decodificador seguindo um fluxo de projeto real, similar àquele realizado na indústria. Para isso foi necessário aprender a trabalhar com ferramentas poderosas de padrão industrial, utilizadas para o desenvolvimento de circuitos integrados dedicados de forma mais automatizada, a partir do código VHDL criado na primeira implementação do decodificador. As ferramentas da Cadence são complexas e exigem experiência para sua utilização, por esse motivo, um treinamento intensivo no Harvey Mudd College foi realizado durante o programa de doutorado "sanduíche" da Capes.

Este projeto resultou em várias contribuições relevantes. A primeira foi a comprovação da viabilidade de implementar um decodificador baseado em conjuntos de informação totalmente em hardware, utilizando o algoritmo desenvolvido pelo grupo. A segunda foi a criação e a fabricação de um chip dedicado (ASIC) deste decodificador. A terceira foi o conhecimento adquirido durante o desenvolvimento do projeto, onde cada etapa do desenvolvimento foi detalhadamente estudada. Além disso, foi fundamental o aprendizado do novo conjunto de ferramentas mencionado acima, seguindo um fluxo de projeto real, como realizado nas indústrias de chips. Finalmente, uma importante contribuição foi o sucesso da cooperação entre os grupos de Microeletrônica da UTFPR, de Comunicação de Dados da UTFPR e do grupo de Microeletrônica do Harvey Mudd College, dirigido pelo Prof. David Harris.

9.2 TRABALHOS FUTUROS

Com o desenvolvimento desta tese surgiram várias oportunidades para trabalhos futuros. Um dos possíveis projetos seria o estudo e a criação de um chip dedicado utilizando o algoritmo apresentado na seção 5.4.2, no qual um número arbitrário de mensagens candidatas é utilizado, permitindo atingir um desempenho tão próximo ao MLD quanto se queira. Outro possível trabalho futuro seria adicionar ao decodificador um critério de parada, visando reduzir o número de candidatas analisadas na determinação da palavra-código vencedora (como descrito na seção 5.4.3). No entanto, para esta implementação ser viável, a utilização de tecnologias mais atuais no design do chip será imprescindível.

REFERÊNCIAS

BERROU, C.; GLAVIEUX, A.; THITIMAJSHIMA, P. *"Near Shannon Limit Error-Correcting Coding and Decoding Turbo-Codes"*. Proc. IEEE Int. Conf. Communications, pp.1064-1070, maio 1993.

BOSE, R. C.; RAY-CHAUDHURI, D. K., "On a Class of Error Correcting Binary Group Codes*".Information and Control.* v. 3, pp. 68-79, 1960.

CASTIÑEIRA, M. J.; FARRELL, P. G. *Essentials of Error-Control Coding*. Chichester: J. Wiley, 2006.

CLARK, G. C. Jr.; CAIN, J. B. *Error-Correction Coding for Digital Communications*.New York : Plenum Press, 1981.

DONG, S.; WANG, X.*"A novel High-Speed parallel scheme for data sorting algorithm based on FPGA"*. Int. Congress on Image and Signal Processing (CISP'09), 2009.

DORSCH, B. G. "A Decoding Algorithm for Binary Block Codes and J-ary Output Channels," *IEEE Trans. on Information Theory*, vol. IT-20, pp. 391-394, May 1974.

ELIAS, P. "Coding for Noisy Channels". *IRE Conv.Rec.*, v. 3, pt. 4, pp. 37-46, 1955.

FOSSORIER, M.; LIN, S. "Soft-decision Decoding of Linear Block Codes Based on Order Statistics," *IEEE Trans. on Information Theory*, vol. 41, No. 5, pp. 13791396, Sep. 1995.

FRANÇA, S. B. L.; JASINSKI, R. P.; PEDRONI, V. A. *"An Efficient VLSI Implementation for the Soft Information-Set Decoding Algorithm"*.Proc. 11th Microelectronics Students Forum (SFORUM 2011). João Pessoa, 2011.

GOLAY, M. J. E. *"Notes on Digital Coding"*. Proc. IEEE 37 (1949), 657.

GODOY, W. Jr. *Esquemas de Modulação Codificada com Códigos de Bloco*. Editora CefetPr, 1991.

BARROS, J. D.; GODOY, W. Jr.; WILLE, E. C. G. "A new approach to the information set decoding algorithm". *Computer Comunications*, vol 20, pp. 302-308, 1997.

GODOY, W. Jr.; WILLE, E. C. G, *"Adaptative decoding of binary linear block codes using information sets and erasures"*.CTRQ, Atenas, 2010

GORTAN, A.; GODOY, W. Jr.; JASINSKI, R. P.; PEDRONI, V. A. *"Hardware-Friendly Implementation of Soft Information Set Decoders"*.IEEE International Telecommunications Symposium (ITS 2010), Manaus, 2010.

GORTAN, A.; JASINSKI, R.P.; GODOY, W.; PEDRONI, V.A.*"Achieving near-MLD performance with soft information-set decoders implemented in FPGAs"*. IEEE Asia Pacific

Conference on Circuits and Systems (APCCAS), pp.312-315, Dec. 2010

GORTAN, A. *Otimização de Algoritmos de Decodificação de Códigos de Bloco por Conjuntos de Informação Visando sua Implementação em Hardware*. 2011.185 f. Dissertação (Mestrado em Engenharia Elétrica). Universidade Tecnológica Federal do Paraná, Curitiba, 2011.

GORTAN, A ; JASINSKI, R. P.; GODOY W JR.; PEDRONI, A. V. *"A Very Efficient, Hardware-Oriented Acceptance Criterion for Soft Information-Set Decoders"*.35th IEEE Sarnoff Symposium. 2012.

HAMMING, R. W. "Error Detecting and Error Correting Codes". *Bell Syst. Tech. J.*, v. 29, pp. 147-160, abr. 1950.

HOCQUENGHEM, A. "CodesCorrecteursd'Erreus". *Cliffres*, v. 2, pp. 147-165, 1959.

LIN, S.; COSTELLO, J. J. *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs: Prentice-Hall, 1983.

HOFFMAN, D. G et al. *Coding Theory: The Essentials*. New York: Marcel Dekker Inc., 1991.

MACKAY, D. J. C.; NEAL, R. M. "Near Shannon Limit Performance of Low Density Parity Check Codes".*Electronic Letters*, vol. 33, no. 6, Mar. 1997.

MORELOS-ZARAGOZA, R. H. *The Art of Error Correcting Coding*.Chichester: J. Wiley, 2002.

PEDRONI, V.A. *Digital Electronics and Design with VHDL*. Boston: Elsevier Morgan Kaufmann Publishers, 2008.

PRANGE, E. "The use of information sets in decoding cyclic codes". *IRE Transactions on Information Theory*, Vol. IT-8, pp. 5-9, Sep. 1962.

RABAEY, J. A.; CHANDRAKASAN, D. N. *Digital Integrated Circuits: A Design Perspective*. 2. ed. Upper Saddle River-NJ: Prentice Hall, 2003.

REED, I.; SOLOMON, G. Polynomial codes over certain finite fields, *SIAM Journal of Applied Mathematics*, v. 8, p. 300-304, 1960.

RIBAS, L.; CASTELLS, D.; CARRABINA, J. *"A linear sorter core based on programmable register file".Conference* on Design of Circuits and Integrated Systems (DCIS'04), 2004.

SHANNON, C. "A Mathematical Theory of Communication". *Bell Syst. Tech. J.*, v. 27, pp. 379-423, 623-656, 1948.

SWEENEY, P.*Error Control Coding: From Theory to Pratice*. Chichester: J. Wiley, 2002.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. *Sistemas Digitais: Princípios e Aplicações*. 10. ed. São Paulo: Pearson Prentice Hall, 2007.

TANNER, L. M. “A Recursive Approach to Low Complexity Codes”.*IEEE Trans. Inf. Theory*,vol. 27, no. 5, pp. 533–547, 1981.

VITERBI, A. J. “Error bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm”. *IEEE Transactions on Information Theory*, v. 13, pp. 260-269, Abr. 1967.

WESTE, N. H. E.; HARRIS, D. M. CMOS VLSI Design: A Circuits and Systems Perspective. 4. ed. Boston: Addison Wesley, 2011.

ANEXO A

CIRCUITOS INTEGRADOS

A.1. INTRODUÇÃO

Circuitos integrados (CIs) digitais modernos são normalmente construídos com transistores do tipo MOSFET (*Metal Oxide Semiconductor Field Effect Transistor*), ou simplesmente MOS. Eles foram escolhidos porque, além de poderem ser utilizados como chaves com baixo consumo de energia, são pequenos e facilmente integráveis, permitindo a criação de milhões de portas em um único dispositivo (RABAEY; CHANDRAKASAN, 2003).

O primeiro flip-flop em circuito integrado foi criado em 1958 por Jack Kilby, construído com apenas dois transistores bipolares. Mais tarde, em 1963, os transistores MOS passaram a ser utilizados, marcando uma grande evolução na área da microeletrônica. Atualmente, os circuitos integrados contêm bilhões de transistores, graças à miniaturização dos mesmos e à evolução do processo de fabricação. Esses avanços resultaram em chips com menor consumo de energia, mais rápidos e mais baratos (WESTE; HARRIS, 2011). Na área digital, apenas em algumas aplicações bem específicas os transistores MOS são substituídos pelos transistores bipolares, como lógica ECL e BiCMOS.

Este capítulo tem por objetivo rever brevemente os principais circuitos lógicos que foram utilizados ou que foram considerados como candidatos nas implementações descritas nos capítulos adiante.

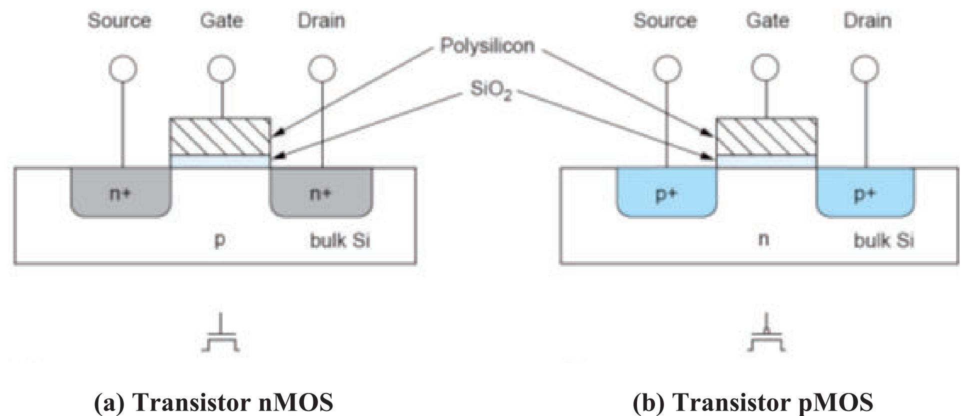
A.2. TRANSISTOR MOS

O silício (Si) é o material semicondutor utilizado na maioria dos circuitos integrados. Porém, para que ele possa ser utilizado, passa por um processo de dopagem, onde outros elementos são adicionados à sua estrutura cristalina. Como o silício possui quatro elétrons na camada de valência (grupo IV), os dopantes podem pertencer ao grupo III ou V da tabela periódica. Utilizando-se um dopante do grupo III (como boro, alumínio, gálio ou índio), têm-se como resultado três ligações covalentes com os átomos de Si e uma lacuna (ausência de

elétron) livre. Esse semicondutor é denominado do tipo p, pois suas cargas livres predominantes (lacunas) são positivas. Para um dopante do grupo V (como arsênio, fósforo ou antimônio) têm-se quatro ligações covalentes e um elétron livre. Como as cargas livres predominantes (elétrons) nesse caso são negativas, esse semicondutor é dito ser do tipo n.

Uma estrutura MOS é criada sobrepondo-se ao semicondutor diversas camadas de condutores e isolantes. Sua construção envolve vários processos físico-químicos, como oxidação do silício, inserção de dopantes e deposição de metal para criação das ligações e dos contatos. Os transistores são construídos em um cristal de silício praticamente perfeito (chamado de wafer ou substrato) e podem ser de canal n (nMOS) ou de canal p (pMOS).

Um diagrama simplificado do transistor nMOS é mostrado na **Figura A.1**(a). Ele é construído sobre um substrato do tipo p, e possui regiões adjacentes à porta (*gate*, G) do tipo n+ (semicondutor do tipo n super dopado), chamadas de fonte (*source*, S) e dreno (*drain*, D). O transistor pMOS (**Figura A.1**(b)) é exatamente o oposto, possuindo duas regiões p+ (semicondutor do tipo p super dopado) adjacentes à porta, sobre um substrato do tipo n.



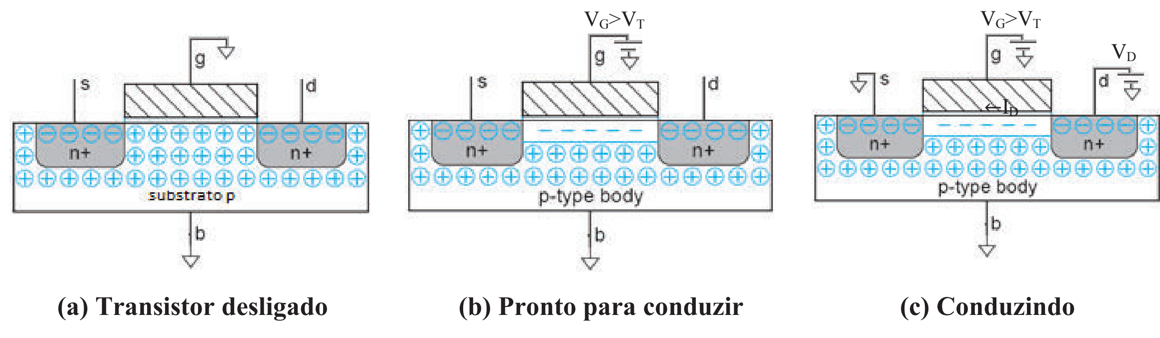
(a) Transistor nMOS **(b) Transistor pMOS**

Figura A.1 - Seção transversal dos transistores nMOS e pMOS.

A.3. OPERAÇÃO DO TRANSISTOR MOS

Um transistor MOS pode ser visto como uma chave controlada eletricamente. Quando a tensão aplicada à porta é maior do que a tensão de limiar (*threshold*, V_T) do transistor, abre-se um canal entre fonte e dreno. A tensão V_T típica para transistores nMOS está entre 0,4 V e 0,7 V, enquanto para pMOS está entre -0,5 V e -0,9 V.

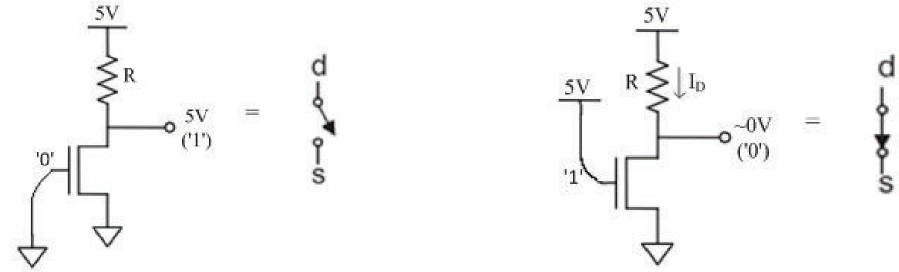
A **Figura A.2** ilustra o funcionamento de um transistor nMOS. Quando nenhum campo elétrico externo é aplicado (**Figura A.2**(a)), uma região p separa as regiões n+ que formam a fonte e o dreno, impedindo a circulação de corrente (transistor cortado, chave aberta). Quando uma tensão superior à tensão de limiar é aplicada à porta (**Figura A.2**(b)), forma-se uma região n próxima à superfície do semicondutor (chamada canal), conectando as regiões n+ que formam a fonte e o dreno, deixando assim o transistor pronto para conduzir. Portanto, se agora uma tensão é aplicada entre dreno e fonte (**Figura A.2**(c)), corrente fluirá através do referido canal (transistor conduzindo, chave fechada). Essa operação está resumida na **Figura A.3**.



(a) Transistor desligado **(b) Pronto para conduzir** **(c) Conduzindo**

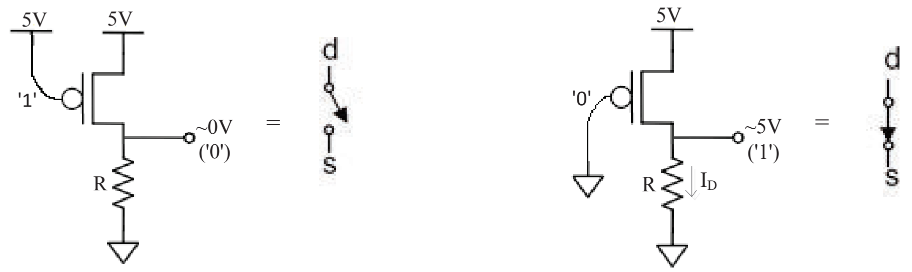
Figura A.2 - Funcionamento do transistor nMOS.

No caso do transistor pMOS, o substrato é ligado a V_{DD} em vez de GND, dispensando o uso de uma fonte negativa, pois qualquer tensão menor do que V_{DD} será então uma tensão negativa para a porta. Como resultado, a operação do transistor pMOS é inversa àquela do transistor ao nMOS, ou seja, ele opera como uma chave aberta quando a tensão da porta é alta e como uma chave fechada quando a tensão é baixa. Tal operação está resumida na **Figura A.4**.



(a) Transistor cortado (chave aberta) **(b) Transistor conduzindo (chave fechada)**

Figura A.3 - Operação do transistor nMOS com chave.



(a) Transistor cortado (chave aberta) **(b) Transistor conduzindo (chave fechada)**

Figura A.4 - Operação do transistor pMOS com chave.

O transistor MOS tem três modos principais de operação. Ele opera na *região de corte* quando a tensão entre porta e fonte é menor do que a tensão de limiar, resultando em fluxo de corrente desprezível, ou seja:

$$\mathbf{V_{GS} < V_T \rightarrow I_D = 0} \quad \textbf{(A.1)}$$

Quando a tensão entre porta e fonte é maior do que a tensão de limiar, o transistor entra em modo de condução efetiva. É a tensão entre dreno e fonte que determina em qual das outras duas regiões o transistor está operando. Se V_{DS} é alto, o transistor opera na *região de saturação*, onde sua corrente é praticamente independente de V_{DS}, ou seja:

$$\mathbf{V_{GS} \geq V_T \text{ e } V_{DS} \geq V_{GS} - V_T \rightarrow I_D = \left(\frac{\beta}{2}\right)(V_{GS} - V_T)^2} \quad \textbf{(A.2)}$$

Entretanto, se V_{DS} é baixo, o transistor opera na *região linear* ou *triodo*, sendo sua corrente expressa por:

$$\mathbf{V_{GS} \geq V_T \text{ e } V_{DS} < \mathit{V}_{GS} - V_T \rightarrow I_D = \beta\left[(V_{GS} - V_T)V_{DS} - \frac{{V_{DS}}^2}{2}\right]} \quad \textbf{(A.3)}$$

O valor de β utilizado nas equações acima é medido em A/V^2, dado pela equação A.4.

$$\mathbf{\beta = \mu C_{ox}\frac{W}{L}} \quad \textbf{(A.4)}$$

O símbolo μ representa a mobilidade das cargas majoritárias no canal, que nos transistores nMOS são elétrons e nos pMOS são lacunas. C_{ox} é a capacitância do óxido da

porta por unidade de área, e pode ser determinada pela equação A.5, sendo a permissividade do SiO_2 dada por $\varepsilon_{ox} = 3.9\varepsilon_o$, onde$\varepsilon_o = 8{,}85 \times 10^{-14}$ F/cm. W e L são a largura e o comprimento do canal do transistor, respectivamente. O valor resultante para β do transistor pMOS é aproximadamente um terço do valor de um transistor nMOS do mesmo tamanho pois a mobilidade das lacunas é inferior à dos elétrons.

$$\mathbf{C_{ox}} = \frac{\varepsilon_{ox}}{t_{ox}} \qquad \textbf{(A.5)}$$

A **Figura A.5** apresenta as características I-V típicas de um transistor nMOS, onde pode-se observar as três regiões anteriormente citadas. A primeira, região de corte, encontra-se abaixo da curva para $V_{GS} = V_T$, na qual a corrente de dreno é praticamente nula. Após V_{GS} atingir a tensão de limiar, o transistor opera na região de saturação. Finalmente, com a diminuição de V_{DS}, chega-se à curva de contração, à esquerda da qual encontra-se a região linear ou triodo.

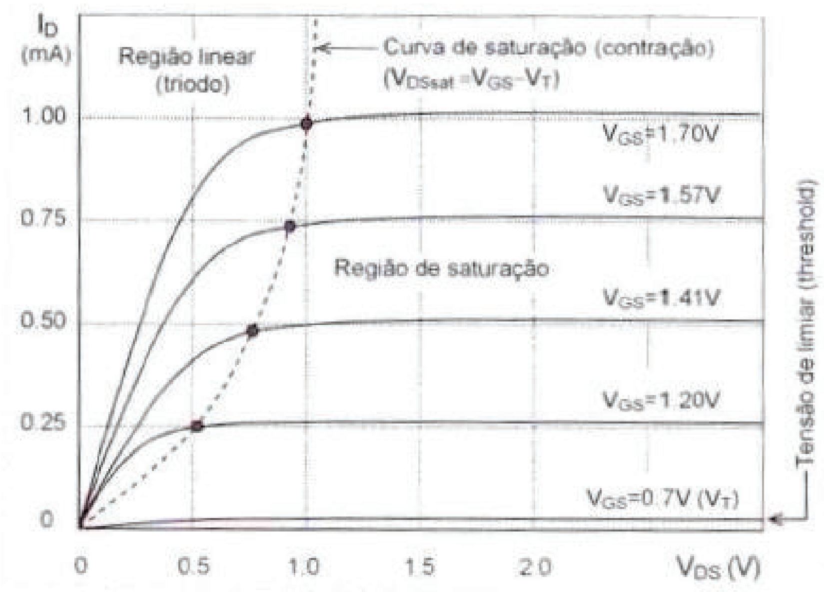


Figura A.5 - Características I-V típica de um transistor nMOS.

A curva que separa as regiões de saturação e linear é chamada *curva de saturação*, dada pela seguinte equação:

$$\mathbf{V_{DS\,sat} = V_{GS} - V_T} \qquad \textbf{(A.6)}$$

O consumo de energia é um dos pontos mais importantes em circuitos digitais modernos. O consumo total de um circuito é obtido pela soma das potências estática e dinâmica. A potência estática é a energia consumida enquanto o circuito permanece no mesmo estado, enquanto que a potência dinâmica é a energia consumida quando o circuito muda de estado.

O circuito da **Figura A.3** é um exemplo que consome energia estática, pois durante todo o tempo em que V_{GS} é alta uma corrente I_D flui de V_{DD} a GND através do resistor e do transistor, consumindo, portanto, $P_{estática} = V_{DD}.I_D$. Na lógica CMOS (*Complementary* MOS), que será vista na seção 2.4.1, praticamente não existe consumo estático porque são utilizados dois transistores complementares, um pMOS e um nMOS, sendo que um deles sempre estará cortado. Entretanto, nas tecnologias CMOS mais novas, nas quais os transistores têm tamanhos abaixo de 65 nm, a corrente de fuga tem crescido, fazendo com que esse consumo estático não seja mais desprezível, quando utilizada uma tensão de *threshold* baixa.

A energia dinâmica é consumida por todos os tipos de circuitos digitais. Ela é dada pela soma da potência de curto e potência capacitiva. Para exemplificar cada uma destas potências, é apresentada na **Figura A.6** a versão CMOS para os inversores mostrados nas Figuras 2.3 e 2.4. A potência de curto corresponde ao momento em que um transistor está sendo desligado, porém o outro está ainda parcialmente ligado, de forma que, por um breve instante, ambos estão conduzindo, fazendo com que exista uma corrente passando através deles (**Figura A.6** (b)). A potência capacitiva corresponde à energia consumida para carregar o capacitor (**Figura A.6** (c)) quando ocorre uma mudança de estado (quando *x* passa de ‘1’ para ‘0’, neste caso).

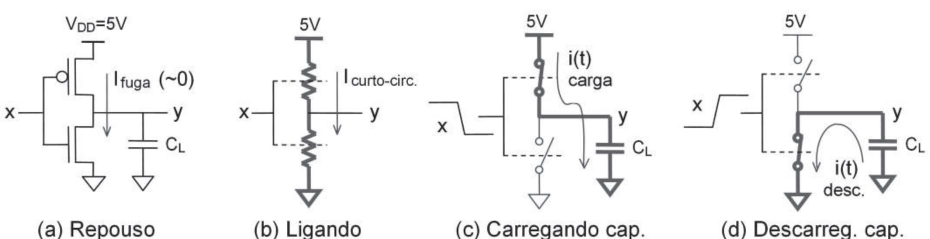


Figura A.6 - Consumo de energia no inversor CMOS.

A.4. ARQUITETURAS MOS ESTÁTICAS

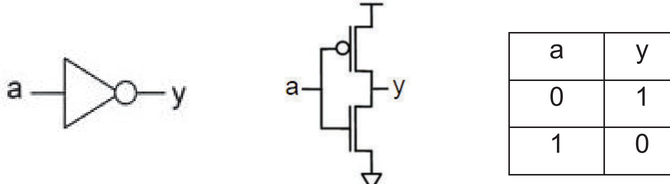
Existem dois tipos de arquiteturas MOS: estáticas e dinâmicas. As arquiteturas MOS dinâmicas são semelhantes às estáticas, porém são controladas por um sinal de clock para ligar e desligar. Arquiteturas dinâmicas não consumem potência estática, porém são sensíveis a ruído durante a fase de avaliação e consomem maior potência dinâmica.

Arquiteturas estáticas são vastamente utilizadas em circuitos integrados, especialmente a lógica CMOS. Outros exemplos de arquiteturas estáticas são a lógica pseudo-nMOS e a lógica com portas de transmissão, todas brevemente descritas abaixo.

A.4.1. Lógica CMOS

A arquitetura lógica CMOS é popular na fabricação de circuitos integrados devido ao seu baixíssimo consumo de potência estática. O único consumo estático deve-se às correntes de fuga, que são significativas somente nas tecnologias mais novas (a partir de 65 nm).

A lógica CMOS utiliza transistores pMOS e nMOS de modo complementar, ou seja, para cada pMOS existe um nMOS. A porta lógica mais simples é o inversor, o qual utiliza apenas um transistor de cada tipo. Esta lógica dispensa o uso de resistores, diferentemente, portanto dos modelos de inversores mostrados nas Figuras 2.3 e 2.4. Desta maneira, a área de silício empregada é menor, pois a construção de resistores ocupa um espaço significativo. A **Figura A.7** apresenta o símbolo, a tabela verdade e o circuito CMOS de um inversor. Outros exemplos de circuitos CMOS serão apresentados na seção 2.6.



a	y
0	1
1	0

Figura A.7 - Inversor CMOS.

A.4.2. Lógica Pseudo-nMOS

A lógica pseudo-nMOS utiliza apenas a parte nMOS do circuito CMOS correspondente, sendo todos os transistores pMOS substituídos por apenas um transistor pMOS, que está sempre em condição de condução, pois sua porta está permanentemente conectada a GND. Esse transistor age como um elemento de *pull-up* fraco. A **Figura A.8** mostra uma porta XOR implementada utilizando lógica pseudo-nMOS.

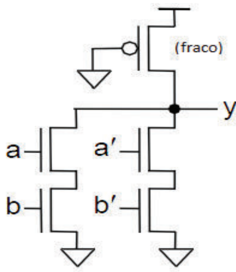


Figura A.8 - Porta XOR construída com lógica pseudo-nMOS.

O espaço ocupado é menor se comparado com circuitos CMOS correspondentes. Em contrapartida, o consumo de potência estática é maior, pois o transistor pMOS está sempre em condição de condução. Adicionalmente, na lógica pseudo-nMOS as transições podem ser lentas; por exemplo, a transição de '0' para '1' pode ser demorada visto que o transistor pMOS é fraco. Além disso, a competição entre o transistor pMOS e os transistores nMOS resulta em um zero pobre na saída (a tensão de saída jamais atingirá 0 V).

A.4.3. Lógica com Portas de Transmissão

Uma porta de passagem (*Transmission Gate* - TG) consiste em uma chave CMOS, ou seja, um transistor pMOS e um nMOS em paralelo, como mostra a **Figura A.9**. Esta chave seleciona se bloqueia ou se repassa para a saída o sinal aplicado à sua entrada. Se a tensão aplicada em *sw* for alta, os transistores nMOS e pMOS conduzirão e repassarão a tensão de entrada (x) para a saída (y). Por outro lado, quando *sw* = '0', os dois transistores estão cortados, deixando y desconectado de x. Esta lógica é ocasionalmente utilizada para implementar circuitos (XOR e outros) de pequeno tamanho, porém também com velocidade relativamente baixa.

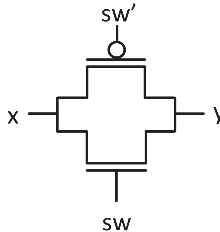


Figura A.9 - Porta de transmissão (TG).

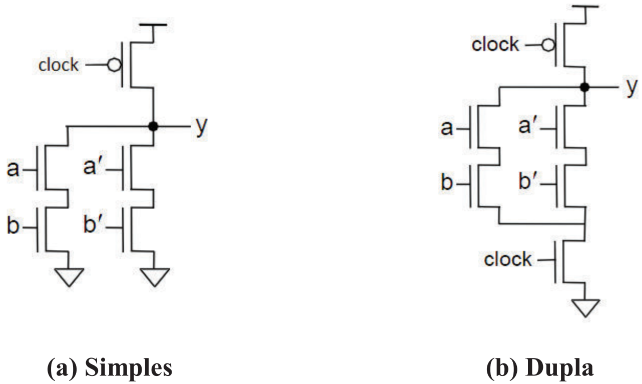
A.5. ARQUITETURAS MOS DINÂMICAS

A operação dos circuitos que utilizam arquitetura dinâmica é dividida em duas partes: fase de pré-carga e fase de avaliação. A fase de pré-carga ocorre quando o clock é baixo (ou alto), inicializando *y*, enquanto que a fase de avaliação ocorre quando clock é alto (ou baixo), produzindo o valor de saída. As principais arquiteturas MOS dinâmicas estão a lógica dinâmica simples, a lógica dominó e a lógica C^2MOS.

A.5.1. Lógica Dinâmica Simples

A lógica dinâmica simples assemelha-se à pseudo-nMOS, pois todos os transistores pMOS são substituídos por apenas um transistor. A diferença é que agora o transistor pMOS é forte e é controlado pelo clock (**Figura A.10**) ao invés de estar permanentemente em estado de condução. Como consequências, o consumo de energia estática torna-se quase desprezível e a transição de '0' para '1' na saída é mais rápida. A desvantagem é obviamente o maior consumo de potência dinâmica.

A **Figura A.10**(a) apresenta uma porta XOR construída com lógica dinâmica. Quando o clock é '0', *y* é pré-carregado com '1' pois o transistor pMOS está conduzindo. Quando o clock passa para '1', a combinação das entradas define o valor de *y*, ou seja, se *a* e *b* forem diferentes, *y* continuará alto; caso contrário, *y* será reduzido a '0'. A **Figura A.10**(b) mostra uma configuração mais rápida e de menor consumo que a configuração da **Figura A.10**(a) devido ao duplo controle exercido pelo clock. Evidentemente, o consumo de energia do circuito de clock será maior.



(a) Simples **(b) Dupla**

Figura A.10 - Porta XOR construída com lógica dinâmica.

A.5.2. Lógica Dominó

A lógica dominó é semelhante à lógica dinâmica simples, porém possui um inversor estático adicionado à saída (Figura 2.11). Durante o período de pré-carga (*clock* = '0'), a saída do circuito é carregada a '1', resultando '0' na saída do inversor. No período de avaliação (*clock* = '1'), a saída é condicionalmente descarregada, assim como a saída do inversor. Esta lógica resolve o problema da monotonicidade existente na lógica dinâmica simples, pois garante que a saída do circuito continuará baixa ou então passará de baixa para alta, mas nunca de alta para baixa.

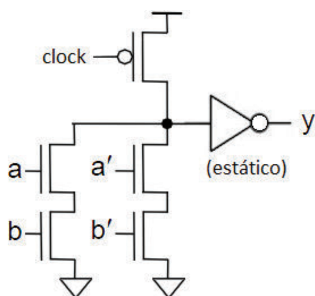


Figura A.11 - Porta XOR construída com lógica dominó.

A.5.3. Lógica C2MOS

Na lógica C^2MOS, um inversor, controlado pelo clock, é inserido antes do nó de saída, propiciando um estado de alta impedância. Um exemplo é mostrado na **Figura A.12**, o qual consiste em um latch tipo D. Quando *clk* = '1', os dois transistores controlados pelo clock estão conduzindo, portanto o valor de entrada (*d*) é repassado para a saída (*q*) (de forma

invertida). Quando *clk* = ‘0’, esses mesmos transistores estão cortados, fazendo com que o nó de saída fique em estado flutuante, mantendo temporariamente o valor anterior.

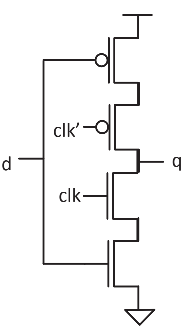


Figura A.12 - Latch tipo D construído com lógica C2MOS.

A.6. PORTAS LÓGICAS FUNDAMENTAIS CMOS

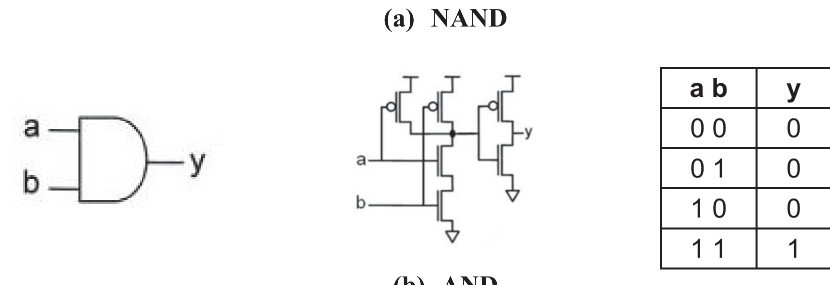
Esta seção e aquelas que se seguem apresentam diagramas dos principais circuitos lógicos empregados nas implementações abordadas nos capítulos seguintes, todos utilizando a lógica CMOS.

A.6.1. NAND e AND

A **Figura A.13**(a) mostra o símbolo de uma porta NAND, seu circuito CMOS e sua tabela verdade. Ela consiste em dois transistores nMOS em série, entre *y* e GND, e dois transistores pMOS em paralelo, entre *y* e V_{DD}. Se pelo menos uma das entradas (*a* ou *b*) for ‘0’, um ou os dois nMOS estarão cortados, interrompendo o caminho entre *y* e GND. Porém, nessas condições, pelo menos um dos transistores pMOS estará conduzindo, resultando *y* = ‘1’. Caso *a* e *b* sejam iguais a ‘1’, os dois transistores nMOS estarão conduzindo e os dois transistores pMOS estarão cortados, resultando *y* = ‘0’. A **Figura A.13**(b) refere-se a uma porta AND, que consiste em uma porta NAND seguida por um inversor.

a b	y
0 0	1
0 1	1
1 0	1
1 1	0

(a) NAND



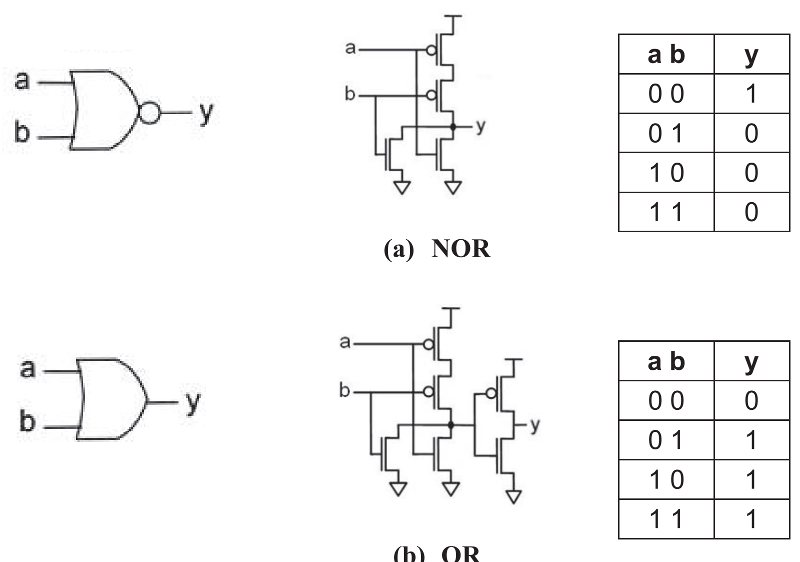
a b	y
0 0	0
0 1	0
1 0	0
1 1	1

(b) AND

Figura A.13 - Portas NAND e AND.

A.6.2. NOR e OR

A **Figura A.14**(a) apresenta o símbolo de uma porta NOR, seu circuito CMOS e sua tabela verdade. Ela é construída utilizando-se dois transistores nMOS em paralelo, entre *y* e GND, e dois transistores pMOS estão em série, entre *y* e V_{DD}. Se pelo menos uma de suas entradas for ‘1’, *y* será ‘0’. A **Figura A.14**(b) refere-se a uma porta OR, que consiste em uma porta NOR seguida por um inversor.



a b	y
0 0	1
0 1	0
1 0	0
1 1	0

(a) NOR

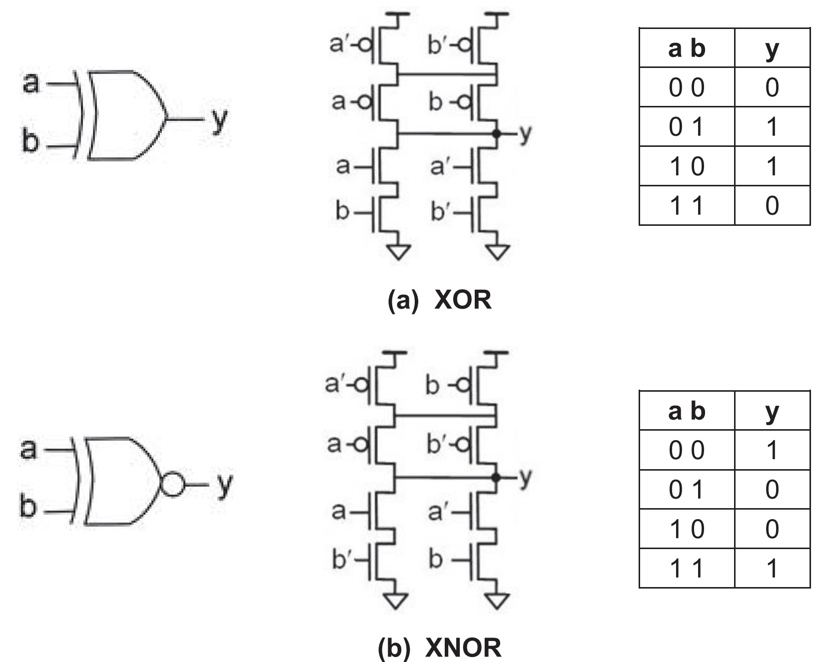
a b	y
0 0	0
0 1	1
1 0	1
1 1	1

(b) OR

Figura A.14 - Portas NOR e OR.

A.6.3. XOR e XNOR

A **Figura A.15**(a) apresenta o símbolo da porta XOR, seu circuito CMOS e sua tabela verdade. A saída de uma porta XOR é ‘1’ quando o número de entradas altas é ímpar. A sua contraparte é a porta XNOR, mostrada na **Figura A.15**(b).



a b	y
0 0	0
0 1	1
1 0	1
1 1	0

(a) XOR

a b	y
0 0	1
0 1	0
1 0	0
1 1	1

(b) XNOR

Figura A.15 - Portas XOR e XNOR.

A.7. LATCHES

Latches são circuitos registradores sensíveis ao nível do sinal de controle (clock) (com exceção dos latches não clocados), ao passo que flip-flops (seção 2.8) são circuitos registradores sensíveis à borda do sinal de controle. Tais circuitos são utilizados para construir circuitos sequenciais e registradores de uso geral.

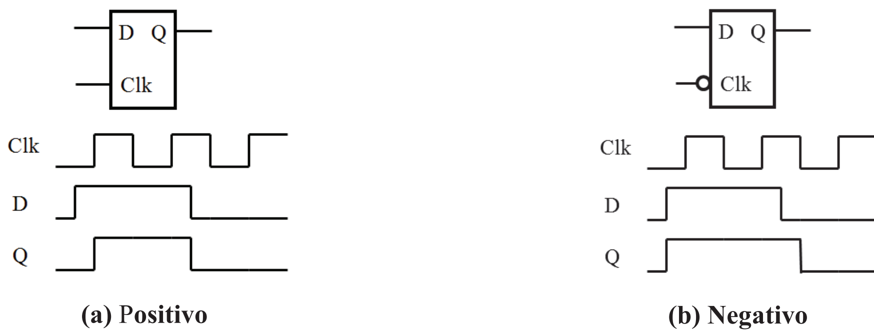
Os latches podem ser divididos nos tipos D (*data*) e SR (*set-reset*). O primeiro é muito mais utilizado do que o segundo, por isso somente ele será descrito neste capítulo.

A.7.1. Latches Tipo D Estáticos

O latch D (DL) é estático quando ele é capaz de manter o valor nele armazenado por um tempo indefinidamente longo.

A operação do DL está resumida na **Figura A.16**. Na **Figura A.16**(a), o latch é transparente (entrada copiada à saída) enquanto o clock está alto, por isso é dito ser um latch

de nível positivo. Sua contraparte é mostrada na **Figura A.16**(b), que é transparente enquanto o clock está baixo (latch negativo).



(a) Positivo **(b) Negativo**

Figura A.16 - Latch tipo D de nível positivo e negativo.

Alguns parâmetros relacionados com o tempo são importante para o correto funcionamento de um latch D. O primeiro é o tempo de propagação de *clk* a *q* (t_{pCQ}), que é o tempo máximo necessário para que um valor que já estava presente em *d* chegue a *q* quando o latch torna-se transparente (**Figura A.17**). Outro parâmetro é o tempo de propagação de *d* para *q* (t_{pDQ}), que é o tempo máximo necessário para que uma mudança em *d* chegue a *q* enquanto o latch estiver transparente. Outros dois parâmetros são os tempos em que a entrada deve se manter estável antes e depois do latch tornar-se opaco, chamados *tempo de setup* e *tempo de hold*, respectivamente (também ilustrados na **Figura A.17**).

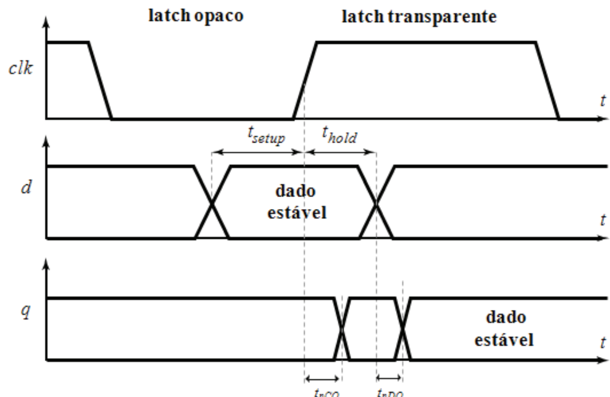
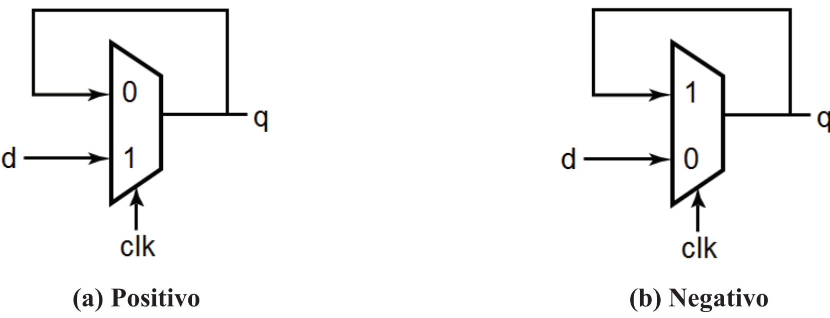


Figura A.17 - Principais parâmetros de tempo do latch D.

Os DLs estáticos podem ser divididos em três grupos: latches baseados em multiplexador, latches do tipo RAM e latches de modo corrente.

DLs estáticos baseados em multiplexador

Uma maneira simples de construir um latch é utilizando um multiplexador. A **Figura A.18** mostra dois DLs deste tipo, sendo o primeiro de nível positivo e o segundo de nível negativo (RABAEY; CHANDRAKASAN, 2003).



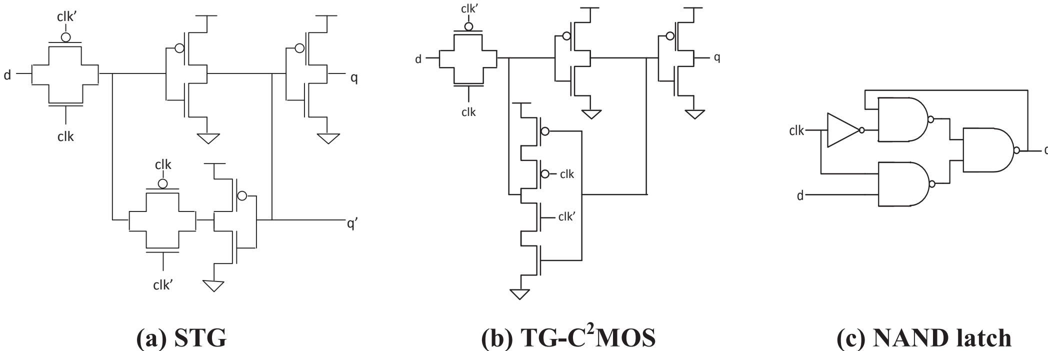
(a) Positivo **(b) Negativo**

Figura A.18 - Latches D baseados em multiplexador.

A **Figura A.19**(a) mostra a implementação CMOS de um latch STG (*Static Transmission Gate*), baseado em multiplexador, o qual é construído com portas de transmissão (TGs). Quando *clk* = '1', a chave de entrada está fechada (os transistores pMOS e nMOS estão conduzindo), fazendo com que o sinal *d* seja repassado à saída, resultando *q* = *d*. Quando *clk* = '0', a chave da entrada está aberta e a chave do laço fechada, portanto o anel formado pelos dois inversores retém o valor anterior de *d*.

Outra implementação possível para latches baseados em multiplexador é o latch TG-C^2MOS, mostrado na **Figura A.19**(b). Ele é semelhante ao anterior, com a diferença de que, ao invés de utilizar uma chave TG e um inversor no elo de retorno, uma porta C^2MOS é empregada. Isso reduz um pouco o tamanho do circuito sem prejudicar significativamente sua velocidade.

A última implementação apresentada para latches baseados em multiplexador (**Figura A.19** (c)) utiliza portas NAND convencionais. Essa implementação não é recomendada em geral por demandar uma área maior.



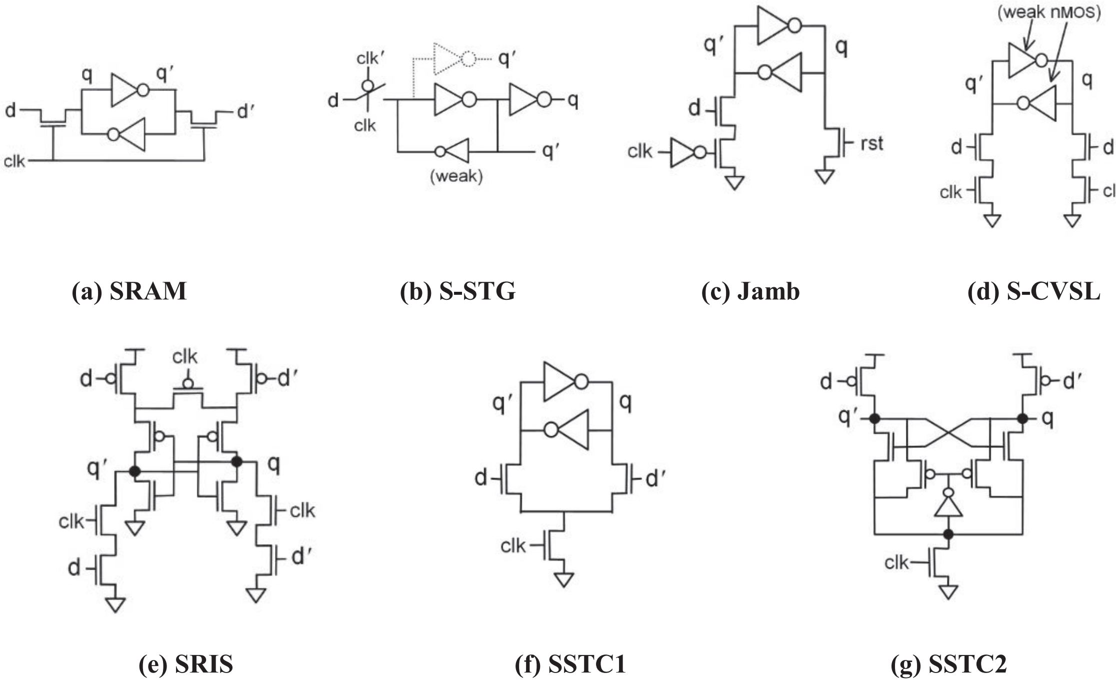
(a) STG **(b) TG-C²MOS** **(c) NAND latch**

Figura A.19 - Latches D estáticos baseados em multiplexador.

DLs estáticos tipo RAM

Os latches estáticos tipo RAM se assemelham aos baseados em multiplexador, porém com uma diferença no elo de retorno: no caso dos baseados em multiplexador, existe uma chave que ora está aberta, ora está fechada, ao passo que nos latches tipo RAM esse elo está permanentemente fechado. Isso reduz o tamanho do hardware, porém a operação de escrever se torna um pouco mais difícil devido à contenção que ocorre entre o sinal de entrada e o sinal armazenado (quando seus valores são diferentes).

Alguns latches positivos com arquitetura RAM são mostrados na **Figura A.20** (PEDRONI, 2008). O primeiro é o circuito clássico de seis transistores usado para implementar cada bit de uma memória SRAM convencional. Uma versão simplificada do latch SGT (denominado S-STG) é vista na **Figura A.20**(b), o qual é semelhante ao STG, porém sem chave no elo de retorno, produzindo um circuito mais compacto, embora mais lento. O circuito da **Figura A.20**(c) é conhecido como *jamb latch*, normalmente empregado em flip-flops para circuitos sincronizadores. O latch da **Figura A.20** (d) utiliza lógica CSVL (*Cascade Voltage Switch Logic*), resultando novamente um circuito compacto, o qual emprega apenas uma fase do clock. A **Figura A.20** (e) mostra um latch SRIS (*Static Ratio Insensitive*), que é uma versão derivada do latch S-CVSL, porém insensível ao tamanho dos transistores do anel (devido ao acréscimo de três transistores pMOS na parte superior do circuito). O latch da **Figura A.20** (f), denominado SSTC1 (*Static Single Transistor Clocked 1*), é semelhante ao S-CVSL, porém com só um transistor clocado, reduzindo a carga do circuito de clock. Finalmente, uma versão um pouco mais complexa mas mais rápida, chamada de SSTC2, é apresentada na **Figura A.20**(g).



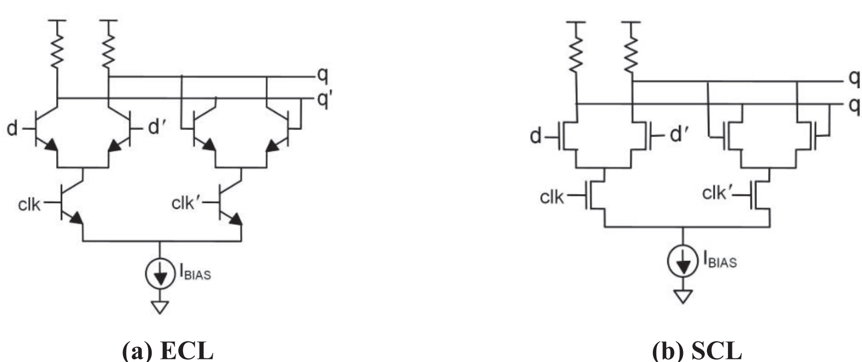
(a) SRAM **(b) S-STG** **(c) Jamb** **(d) S-CVSL**

(e) SRIS **(f) SSTC1** **(g) SSTC2**

Figura A.20 - Latches D estáticos tipo RAM.

DLs estáticos de modo corrente

Os latches de modo corrente são os mais rápidos, porém apresentam maior consumo de energia. Uma implementação deste tipo de latch, utilizando a lógica ECL (*Emitter Coupled Logic*), é mostrada na **Figura A.21**(a). Conforme pode-se observar, ele não têm elo de retorno como os circuitos anteriores, e utiliza transistores bipolares ao invés de MOSFETs, os quais estão acoplados pelo emissor. Uma implementação equivalente, porém com menor consumo de energia, mas também menor velocidade é mostrada na **Figura A.21**(b), utilizando transistores nMOS.



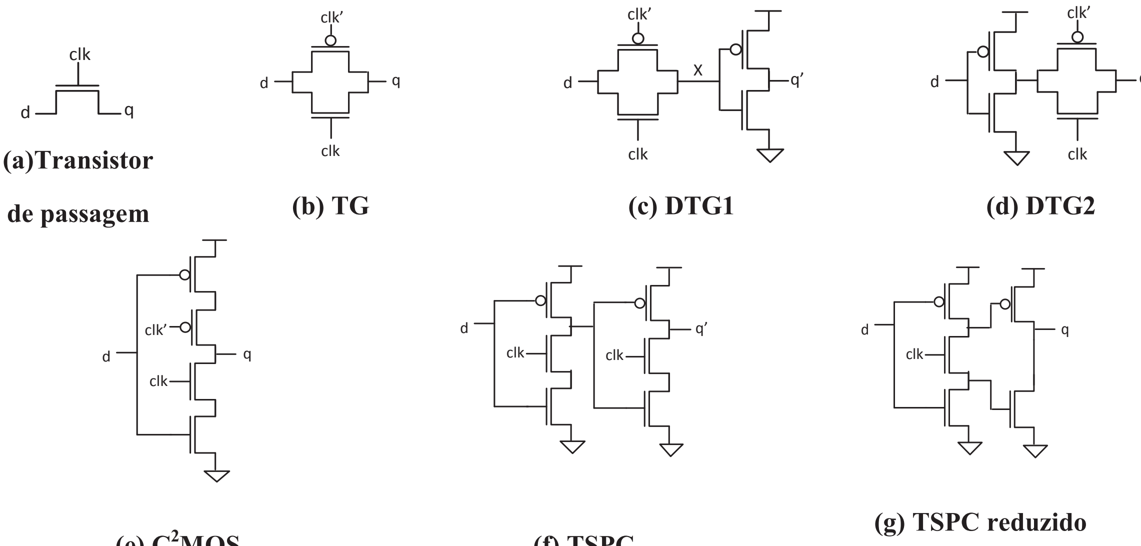
(a) ECL **(b) SCL**

Figura A.21 - Latches D estáticos de modo corrente.

A.7.2. Latches Tipo D Dinâmicos

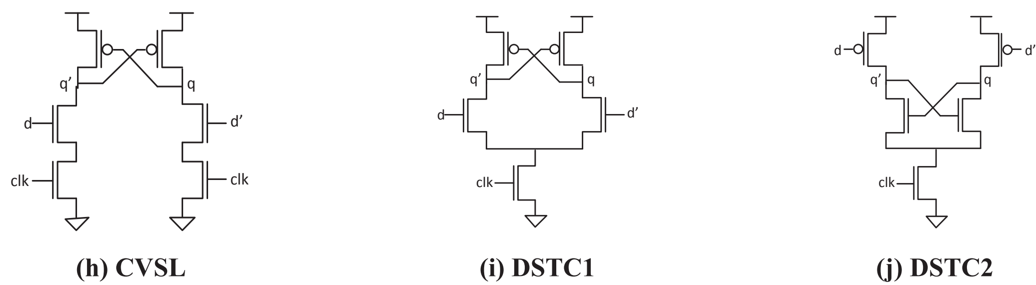
Os latches dinâmicos precisam ser atualizados periodicamente para que possam manter o valor neles armazenado. Isso é necessário porque o valor guardado depende da carga armazenada em capacitores muito pequenos (parasitas). Tal atualização demanda potência adicional e não permite que o circuito funcione com frequências de clock muito baixas. Porém, latches dinâmicos geralmente utilizam menos área de silício e são mais rápidos do que os latches estáticos.

Uma série de latches dinâmicos é mostrada na **Figura A.22** (PEDRONI, 2008). O primeiro deles é o menor latch possível, pois consiste em apenas um transistor (chave de passagem), o qual causa o sinal de entrada a ser armazenado no capacitor parasita do nó de saída. O latch da **Figura A.22**(b) é semelhante ao anterior, porém utiliza uma chave CMOS completa (*Transmission Gate* - TG), eliminando o problema do '1' pobre existente no latch anterior, porém agora as duas fases do clock são necessárias. O latch DTG1 (*Dynamic* TG-*based* 1), mostrado na **Figura A.22**(c), tem uma porta inversora adicionada à saída do TG, fazendo com que o nó *x* fique isolado dos ruídos provenientes do nó de saída. Já na **Figura A.22**(d), a porta inversora é colocada antes do TG, isolando a entrada. Diversas outras arquiteturas são mostradas nas **Figura A.22**(e)-(j).



(a)Transistor de passagem **(b) TG** **(c) DTG1** **(d) DTG2**

(e) C²MOS **(f) TSPC** **(g) TSPC reduzido**



(h) CVSL **(i) DSTC1** **(j) DSTC2**

Figura A.22 - Latches D dinâmicos. - Latches D dinâmicos.

A.8. FLIP-FLOPS

Flip-flops também são circuitos registradores, porém, ao contrário dos latches, são transparentes apenas no momento de transição do clock (isto é, na borda positiva ou negativa deste). Se a saída é atualizada (recebe uma cópia da entrada) na transição positiva do clock, ele é dito ser um flip-flop de borda positiva; caso contrário, é um flip-flop de borda negativa.

A.9. TIPOS DE FLIP-FLOPS

Os flip-flops podem ser divididos em quatro tipos: D (*Data*), T (*Toggle*), SR (*Set-Reset*) e JK. O flip-flop tipo D é o mais utilizado, pois além de ser um flip-flop de uso geral, todos os demais podem ser construídos a partir dele.

A.8.1.Flip-Flop Tipo D

Flip-flops tipo D (DFFs) são construídos a partir latches tipo D, utilizando uma de duas técnicas: mestre-escravo ou baseada em pulso

Flip-flops D mestre-escravo

A abordagem mestre-escravo para construir flip-flops é ilustrada na **Figura A.23**. Ela consiste em utilizar-se dois latches tipo D ligados em série, cada um transparente em uma das fases do clock. Como nesse caso o primeiro é transparente com clock negativo e o segundo é transparente com clock positivo, um DFF de borda positiva resulta.

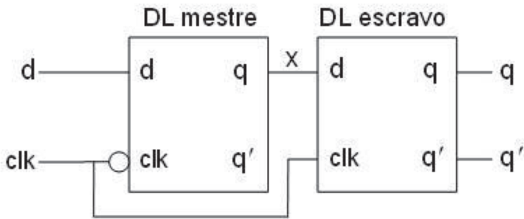
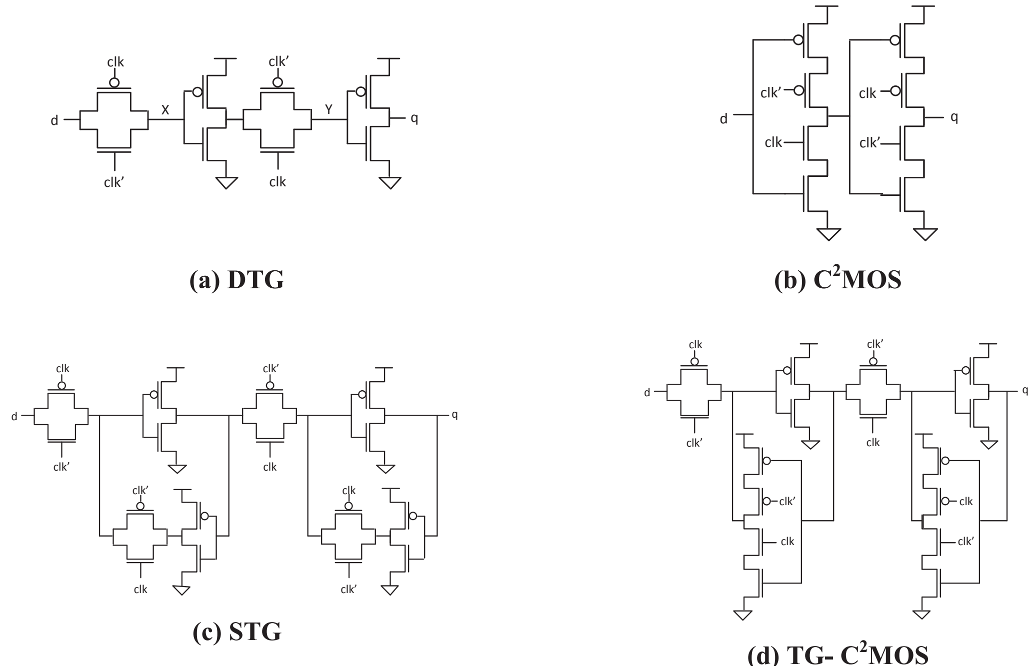


Figura A.23 - Técnica mestre-escravo para construção de flip-flops tipo D.

Algumas implementações de DFFs baseadas na técnica mestre-escravo constam na **Figura A.24** (PEDRONI, 2008). O dois primeiros utilizam os latches dinâmicos das Figuras 2.22(c) e 2.22(e), respectivamente, enquanto que o terceiro e o quarto utilizam os latches estáticos das Figuras 2.19(a) e 2.19(b), respectivamente.



(a) DTG **(b) C^{2}MOS**

(c) STG **(d) TG- C^{2}MOS**

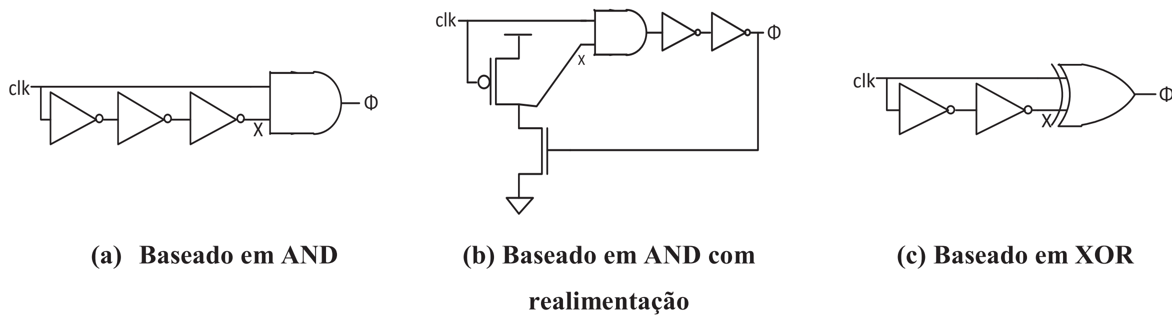
Figura A.24 - Flip-flops tipo D mestre-escravo (todos de borda positiva).

Flip-flops D baseados em pulsos

Outra abordagem utilizada na construção de flip-flops é baseada em pulso curto (derivado do clock). Esse sinal pulsado (Φ) é usado como clock do latch, fazendo com que ele fique transparente durante apenas um breve instante de tempo, comportando-se assim como se fosse um flip-flop. Isso reduz o tamanho dos circuitos, porém o projeto no tocante ao comportamento no tempo é muito mais crítico. Praticamente todos os DFFs de

microprocessadores comerciais operando na faixa de multi-GHz são construídos utilizando essa técnica.

Diversos estreitadores de pulsos são mostrados na **Figura A.25**. O primeiro consiste em uma porta AND que recebe o clock numa das entradas e uma versão invertida e com retardo do clock na outra, resultando um breve pulso na saída sempre que o clock muda de ‘0’ para ‘1’. O segundo também é baseado em porta AND, porém com elo de realimentação, dando mais segurança à saída. O último opera com porta XOR, resultando assim pulsos estreitos com o dobro da frequência do clock na saída.



(a) Baseado em AND **(b) Baseado em AND com realimentação** **(c) Baseado em XOR**

Figura A.25 - Gerador de pulso estreito.

A.8.2.Flip-flops D com Reset e Enable

Muitos circuitos requerem um sinal de *reset* para que seu estado inicial seja conhecido. Tal comando pode ser assíncrono ou síncrono. O primeiro, ilustrado na **Figura A.26**(a), força a saída a ‘0’ imediatamente, enquanto que o segundo (também chamado de *clear*) força a saída para ‘0’ na próxima borda (positiva) do clock.

(a) Assíncrono **(b) Síncrono.**

Figura A.26 - Flip-flop tipo D com reset assíncrono e síncrono.

Alguns circuitos requerem também que os flip-flops possuam um sinal de *enable.* Tal sinal pode ser obtido utilizando-se um multiplexador, como na **Figura A.27**. Se *ena* = ‘1’, *d* é

disponibilizado na entrada do DFF; caso contrário, o valor atualmente armazenado no DFF é aplicado à sua entrada.

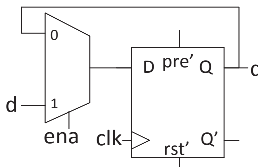
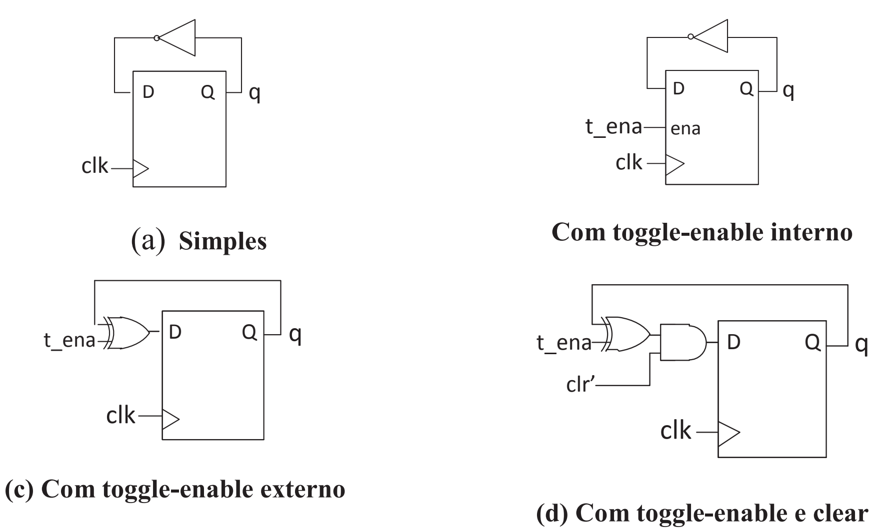


Figura A.27 - Flip-flop tipo D com enable.

A.8.3. Flip-Flop Tipo T

O flip-flop tipo T (*Toggle Flip-Flop* - TFF) é um circuito baseado em alternância, ou seja, muda sua saída de ‘0’ para ‘1’ ou de ‘1’ para ‘0’ toda vez que ocorre uma borda (positiva ou negativa, dependendo se é um flip-flop de borda positiva ou negativa) do clock.

É possível construir um TFF com um DFF através da simples conexão de uma versão invertida da sua saída de volta à entrada, conforme mostra a **Figura A.28**(a). Em certas aplicações, é necessário um sinal de *toggle enable* (*t_ena*), como na **Figura A.28**(b), o qual habilita a operação do TFF somente quando é ‘1’. A **Figura A.28**(c) mostra uma opção para adicionar este sinal de enable. Já no caso da **Figura A.28**(d), além do sinal de *togue embale*, um sinal de *colear* (reset síncrono) foi também adicionado ao TFF.



(a) **Simples**

Com toggle-enable interno

(c) Com toggle-enable externo

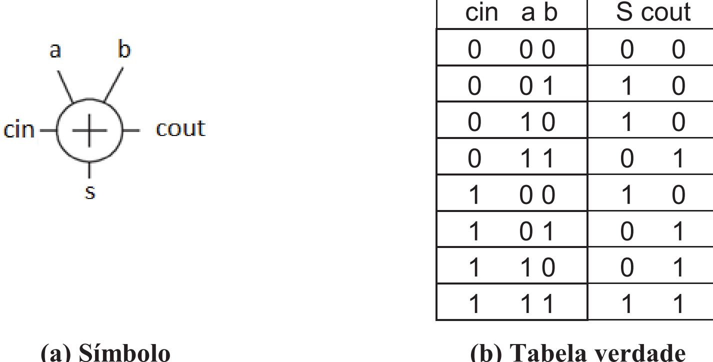
(d) Com toggle-enable e clear

Figura A.28 - Flip-flops tipo T.

Os contadores síncronos são construídos com TFFs. Contudo, como o TFF pode ser obtido facilmente a partir do DFF, somente este último é necessário. A propósito, o DFF é o único tipo de flip-flop fabricado em FPGAs.

A.10. COMPARADORES

Os comparadores são circuitos lógicos combinacionais que fazem a comparação entre duas quantidades binárias. Em geral, eles são construídos utilizando-se células somadoras básicas completas de um bit, conhecidas como *full adder* (FA). Um símbolo e a tabela verdade do FA são mostrados na **Figura A.29**.



cin a b	S cout
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

(a) Símbolo **(b) Tabela verdade**

Figura A.29 - Somador completo de um bit (full adder).

Uma implementação CMOS para o FA é mostrada na **Figura A.30**. *a* e *b* são as entradas a serem somadas e *cin* é o bit de vem-um (*carry-in*), também considerado na soma. As saídas são a soma *s* e o bit de vai-um *cout* (*carry-out*), dados pelas seguintes expressões:

$$\mathbf{s = a \oplus b \oplus cin} \quad \textbf{(2.7)}$$

$$\mathbf{cout = a.b + a.cin + b.cin} \quad \textbf{(2.8)}$$

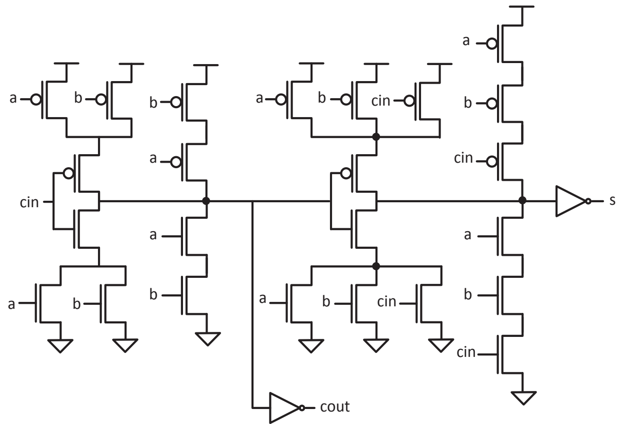


Figura A.30 - Circuito CMOS do full adder.

Um comparador de magnitude é mostrado na **Figura A.31**, o qual determina qual das entradas é maior. Para determinar se as entradas A e B são iguais, a equação $\mathrm{B} - \mathrm{A} = \mathrm{B} + \overline{\mathrm{A}}$ é calculada. Se o *carry-out* for ‘0’, então $\mathrm{A} \leq \mathrm{B}$; caso contrário, $\mathrm{A} > B$. Um detector de zero indica quando os dois valores são iguais (WESTE; HARRIS, 2011) (TOCCI, R; WIDMER; MOSS, 2007).

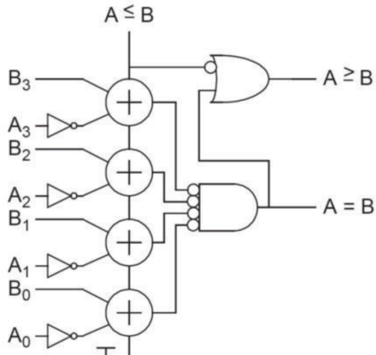
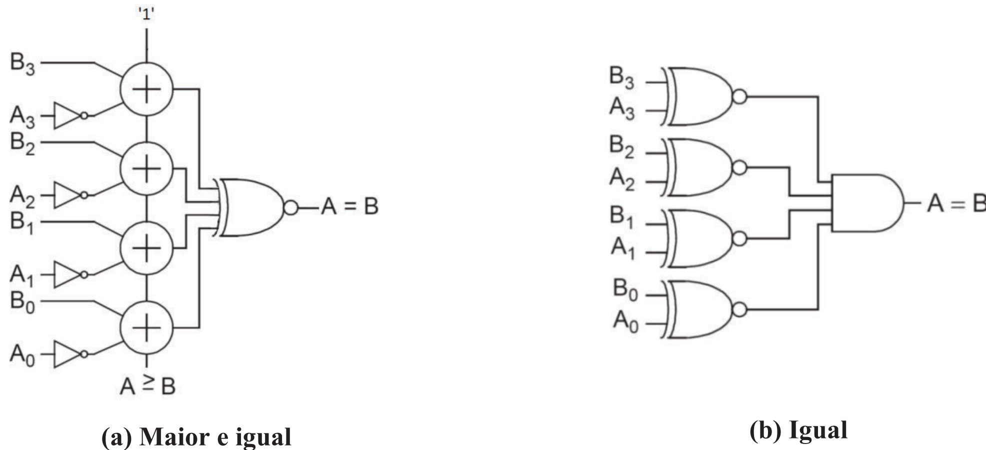


Figura A.31 - Comparador de magnitude de quatro bits sem sinal.

Outras configurações de comparadores, novamente baseados no somador FA, são apresentadas na **Figura A.32**. Na **Figura A.32** (a) são realizadas as comparações $A \geq B$ e $A = B$. Se o objetivo for apenas determinar se os valores das entradas são iguais, o circuito da **Figura A.32** (b) pode ser utilizado, o qual é mais simples e mais rápido (WESTE; HARRIS, 2011).



(a) Maior e igual **(b) Igual**

Figura A.32 - Comparador maior e igual e comparador igual.

Comparar sinais em complemento de dois é um pouco mais complicado, pois existe a possibilidade de *overflow* quando uma subtração de números com sinais diferentes é realizada. A **Figura A.33**mostra o circuito de um comparador com sinal (PEDRONI, 2008), com valores negativos representados utilizando complemento de dois.

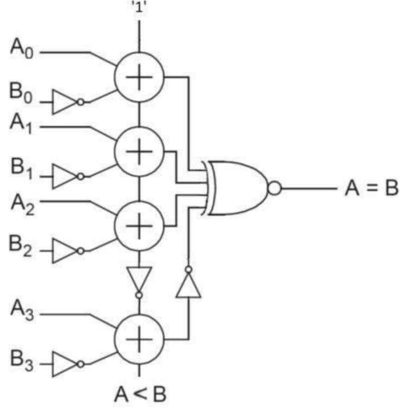


Figura A.33 - Comparador com sinal.

A.11. MULTIPLEXADORES

Multiplexadores são circuitos combinacionais muito utilizados, pois permitem o roteamento de dados e manipulação de estruturas. Um multiplexador escolhe, dependendo do valor do sinal de seleção, qual das entradas deve ser repassada para a saída. Portanto, sua função lógica (para duas entradas, *a* e *b*, e sinal de seleção *sel*) é dada por:

$$y = sel'.a + sel.b \quad (2.9)$$

O símbolo do multiplexador é mostrado na **Figura A.34**(a). Existem várias maneiras de construí-lo, sendo duas delas ilustradas nas **Figura A.34** (b)-(c). A primeira é baseada em portas NAND, enquanto que na segunda são utilizadas chaves do tipo *transmission gate* (TG).

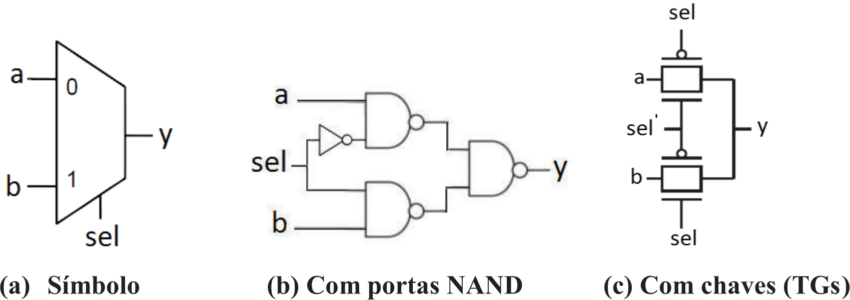


Figura A.34 - Construção de multiplexadores.

Multiplexadores maiores podem ser construídos utilizando-se outros menores. Por exemplo, é possível obter-se um multiplexador com entradas e saída de quatro bits (**Figura A.35**) utilizando-se quatro multiplexadores menores (com duas entradas de um bit cada) em paralelo. O sinal de seleção utilizado é o mesmo para todos os muxes.

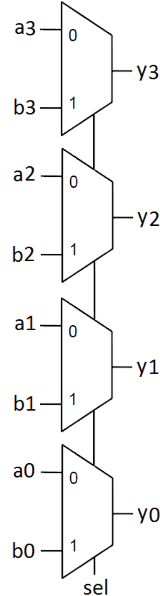
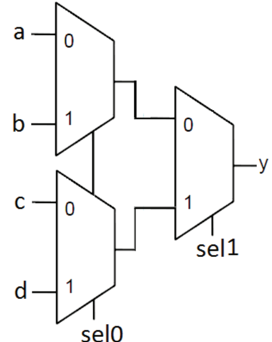


Figura A.35 - Multiplexador 2x3 construído com muxes menores.

Multiplexadores com mais do que duas entradas também podem ser construídos combinando muxes menores. A **Figura A.36** apresenta um exemplo de um multiplexador de quatro entradas de um bit cada. Como o circuito possui quatro entradas, o sinal de seleção necessita de dois bits, como mostra sua tabela verdade, também incluída na figura.



sel0 sel1	y
0 0	a
1 0	b
0 1	c
1 1	d

Figura A.36 - Multiplexador 4x1 construído com muxes menores.