Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto de Circuitos Digitais Sequenciais por Algoritmos Baseados em Programação Genética Cartesiana em FPGA

Vitor Coimbra de Oliveira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientador
Prof. Dr. Jacir Luiz Bordim

Coorientador
Prof. Dr. Marcus Vinicius Lamar

Brasília
2018

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Pós-graduação em Informática

Coordenador: Prof. Dr. Bruno Machiavello

Banca examinadora composta por:

Prof. Dr. Jacir Luiz Bordim (Orientador) — CIC/UnB
Prof. Dr. Ricardo Pezzuol Jacobi — CIC/UnB
Prof. Dr. Renato Perez Ribas — Inf/UFRGS

CIP — Catalogação Internacional na Publicação

Oliveira, Vitor Coimbra de.

Projeto de Circuitos Digitais Sequenciais por Algoritmos Baseados em Programação Genética Cartesiana em FPGA / Vitor Coimbra de Oliveira. Brasília : UnB, 2018.
137 p. : il. ; 29,5 cm.

Dissertação (Mestrado) — Universidade de Brasília, Brasília, 2018.

1. algoritmos genéticos, 2. fpga, 3. circuitos digitais

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto de Circuitos Digitais Sequenciais por Algoritmos Baseados em Programação Genética Cartesiana em FPGA

Vitor Coimbra de Oliveira

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Prof. Dr. Jacir Luiz Bordim (Orientador)
CIC/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
CIC/UnB

Prof. Dr. Renato Perez Ribas
Inf/UFRGS

Prof. Dr. Bruno Machiavello
Coordenador do Programa de Pós-graduação em Informática

Brasília, 30 de julho de 2018

Dedicatória

Dedico esse trabalho a meus pais, Oswaldo e Marianita.

Agradecimentos

Agradeço ao meus orientadores por permitirem que esse projeto pudesse ser feito e aos alunos Gabriel e Douglas pela ajuda prestada.

Resumo

Técnicas de projeto de circuitos digitais atualmente se baseiam principalmente em métodos *top-down*, que utilizam um conjunto de regras e restrições para auxiliar a construção do projeto. Por conta disso, ainda há um espaço desconhecido de soluções para vários problemas. Algoritmos genéticos, por outro lado, constroem soluções utilizando uma metodologia *bottom-up*, e provaram-se úteis para problemas de alta complexidade e de otimização. Este trabalho propõe uma nova abordagem para o projeto de circuitos sequenciais utilizando algoritmos genéticos para explorar soluções fora do espaço alcançado atualmente pelo estado da arte. Trabalhos recentes têm um foco grande em evoluir apenas a parte combinacional dos circuitos sequenciais, ou seja, suas funções de transição e saída. Neste projeto, armazenamento e funcionalidade são ambos levados em conta, permitindo que a evolução use dos dois para alcançar seu objetivo. Os experimentos realizados nos circuitos básicos assíncronos, em ordem crescente de complexidade, *latches* SR, D, *XOR*, JK, D multiplexada, de duas portas e BILBO, e também nos circuitos síncronos *flip-flop* D e paridade-2, mostram que é possível encontrar soluções inovadoras, algumas com características como melhor utilização de espaço, para esses tipos de circuito.

Palavras-chave: algoritmos genéticos, fpga, circuitos digitais

Abstract

Current digital circuit design techniques are based on top-down methods, which depend on a set of rules and restrictions made to help the design process. Because of that, there is still an unknown space of solutions for many problems. Genetic algorithms, on the other hand, build solutions by using a bottom-up methodology and have proven themselves useful for high complexity and optimization problems. This work proposes a new approach to the design of sequential circuits by using genetic algorithms to explore solutions outside the design space currently reached by the state of the art. Recent works focus mainly on evolving the combinational part of a sequential circuit, that is, its transition and output functions. In this project, both the mechanism used for storing and its functionality are taken into account, allowing the genetic algorithm to manipulate both in its search. The experiments done on the basic asynchronous circuits, in increasing complexity, SR, D, XOR, JK, multiplexed D, two port and BILBO latches, and on the synchronous circuits D flip-flop and 2 bit parity circuits show that it is possible to find novel solutions, some with improvements such as better space usage, for these kinds of circuits.

Keywords: genetic algorithms, fpga, evolved circuits, sequential digital circuits

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ASIC Circuito Integrado de Aplicação Específica (do Inglês *Application-Specific Integrated Circuit*). 22

BIST *Built-in Self-test*. 68

CGP Programação Genética Cartesiana (do Inglês *Cartesian Genetic Programming*). 31, 34, 38, 42, 49

CPU Unidade Central de Processamento (do Inglês *Central Processing Unit*). 49, 50

DNA Ácido Desoxirribonucleico. 1

FPGA *Field-Programmable Gate Array*. 1–4, 22–25, 34, 37, 39, 49, 50, 52, 54, 55, 57–59, 62, 63, 68, 74, 77–82, 84, 85, 87, 88

FPL *Field-Programmable Logic*. 24

HDL Linguagem de Descrição de Hardware (do Inglês *Hardware Description Language*). 22

HMC-CGP *Hybrid Multi-Chromosome Cartesian Genetic Programming*. 39

I/O Entrada / Saída (do Inglês *Input / Output*). 22, 50

LE *Logic Element*. 23, 24

LUT *Look-up Table*. 23

MC-CGP *Multi-Chromosome Cartesian Genetic Programming*. 38, 39

NSGA-II *Non-dominated Sorting Genetic Algorithm II*. 38

SSH *Secure Shell*. 49

tpd tempo de atraso de propagação (do Inglês *propagation delay time*). 9

VRC Circuito Reconfigurável Virtual (do Inglês *Virtual Reconfigurable Circuit*). 50, 55, 57, 78–80, 82, 85

Capítulo 1

Introdução

Tradicionalmente, os seres humanos projetam sistemas de alta complexidade, tais como prédios, computadores e carros, utilizando um conjunto complexo de regras que visam cumprir uma série de requerimentos. Por natureza, esse processo é feito de uma sistemática *top-down*, ou seja, se parte de uma especificação abstrata do problema e segue-se através da construção de sistemas menores e mais específicos. Esta sistemática se apresenta em forte contraste com o mecanismo que resultou na concepção da imensa diversidade de seres vivos hoje encontrados. Os seres vivos são a consequência de um conjunto de instruções simples, codificadas por DNA, que constroem componentes mais complexos. Um organismo é criado após inúmeras reações químicas causadas pelo DNA. Organismos complexos são então formados através de um processo evolutivo. Segundo Darwin [2], uma das bases da evolução é o mecanismo de Seleção Natural. Este mecanismo é baseado em uma população composta de diversos indivíduos, cada um com características próprias definidas pelos genes. A sobrevivência de um indivíduo e sua capacidade de transferir seus genes para seus descendentes são diretamente relacionados à sua adaptação em relação ao ambiente em que se encontra.

Com o advento da computação, diversas técnicas para o projeto de componentes digitais surgiram para solucionar diferentes problemas nos últimos anos.

Dentre elas, surgiram os chamados dispositivos reconfiguráveis, cuja principal característica é a sua capacidade de ser configurado para a realização de tarefas específicas. Alguns desses dispositivos podem ter partes reconfiguradas enquanto outras estão executando suas funções. Um tipo especial de circuito reconfigurável chamado *Field-Programmable Gate Array* (FPGA) é utilizado neste trabalho.

Formado essencialmente por uma grade de células lógicas, cada qual interconectada com suas vizinhas e cujo objetivo é controlar se dois fios estão eletricamente ligados ou não, torna-se possível a implementação de um vasto conjunto de funções Booleanas [3].

Paralelamente, outra maneira de solução de problemas, chamada de *Programação Genética*, baseada em algoritmos genéticos, já vem há muito tempo sendo utilizada para problemas relacionados a *software*. Algoritmos genéticos se baseiam fortemente nos princípios da seleção natural para resolução de problemas [4]. Os indivíduos de uma população representam as possíveis soluções e são codificados em uma sequência de *bits* chamada *bitstream*, atuando com o papel de cromossomos, descrevendo essa solução. Novas populações são formadas e testadas utilizando a chamada função de *fitness*, em que cada indivíduo é avaliado e associado a um *score* baseado no quão perto do resultado desejado ele chegou. Esse *score* determina suas chances de se reproduzirem e passarem seus cromossomos para as populações seguintes [4]. Dessa maneira, à medida que novas iterações são feitas, indivíduos pouco adaptados tendem a não sobreviver e os mais adaptados a se reproduzir. Ao longo do tempo, devido aos mecanismos de *crossover* e mutação geralmente encontrados em tais algoritmos, pequenas modificações podem ser observadas, resultando em novas soluções e possivelmente gerando também um aumento no *fitness* da população e, consequentemente, uma aproximação maior à solução adequada.

1.1 Motivação

A motivação principal deste trabalho é a união dos dois campos apresentados e possivelmente a exploração de um espaço de soluções ainda não bem explorado. As perguntas principais que guiam este trabalho são: "é possível utilizar algoritmos genéticos para projetar circuitos digitais sequenciais intrinsecamente?" e "esses projetos possuem alguma característica destoante de projetos humanos, sendo, possivelmente mais eficientes?". Assim, tendo em vista que a natureza é capaz de evoluir sistemas orgânicos como os seres humanos, esperamos, por métodos inspirados, obter resultados inovadores que possamos estudar e avançar o conhecimento do projeto de circuitos digitais.

1.2 Objetivos

Este trabalho possui como objetivo principal o estudo e proposição de técnicas que possibilitem a evolução genética intrínseca para o projeto de circuitos digitais sequenciais em FPGA.

Como objetivos específicos, buscamos desenvolver uma técnica de evolução intrínseca capaz de ser executada inteiramente no FPGA *Cyclone V*, aproveitando-se de seu processador *ARM* embarcado. Nessa técnica, planejamos também desenvolver avaliações capazes de darem suporte a esse tipo de evolução e sua eficácia será verificada por meio de testes envolvendo circuitos assíncronos e síncronos básicos.

1.3 Justificativa

Esperamos, com essa proposição, contribuir com o estado da arte do projeto de circuitos digitais sequenciais, explorar e estudar desenhos que ainda não foram alcançados no espaço de soluções.

Este trabalho está organizado da seguinte forma: no capítulo 2 é apresentada a revisão bibliográfica sobre os conceitos de álgebra booleana, circuitos digitais, FPGAs e algoritmos genéticos e suas abordagens, bem como a apresentação do estado-da-arte em projetos de circuitos sequenciais. O Capítulo 3 detalha as técnicas e procedimentos propostos para a evolução intrínseca de circuitos digitais sequenciais, seguido de detalhes da arquitetura utilizada. No Capítulo 4, a metodologia proposta é aplicada a alguns experimentos escolhidos e apresentados em ordem de complexidade, indicando características interessantes em cada, além de análises sobre o desempenho e considerações finais. O Capítulo 5 apresenta as conclusões finais, apontado também possíveis direções futuras para trabalhos nesta linha.

Capítulo 2

Fundamentação Teórica

Este capítulo apresentará conceitos básicos utilizados neste trabalho. Primeiramente, serão mostrados os fundamentos da Álgebra Booleana, juntamente com técnicas de otimização de expressões lógicas. Uma breve explicação sobre os dispositivos FPGAs é dada, em seguida é feita uma revisão sobre a base teórica de algoritmos genéticos e, ao final, uma breve revisão sobre o estado da arte da evolução de circuitos.

2.1 Álgebra Booleana

Inicialmente proposta por George Boole [5], a área hoje denominada Álgebra Booleana trata de problemas relacionados à lógica moderna. Essa lógica também forma a base para computação em sistemas modernos. Qualquer algoritmo ou circuito eletrônico digital pode ser representado por um sistema de equações booleanas [6].

Nesta álgebra, variáveis assumem apenas um conjunto finito de valores. Mais precisamente, apenas dois valores distintos são representados: $\{0, 1\}$. Operações são realizadas sobre essas variáveis utilizando operadores unários, que possuem uma variável como entrada, e binários, que possuem duas.

2.1.1 Operadores Básicos

As funções básicas e principais que compõem todas as outras possíveis são denominadas: *AND*, *OR* e *NOT*, formando então um conjunto completo de conectivos.

Operador *AND*

O resultado da operação *AND* pode ser definido como 1 apenas se ambas as entradas são 1 e 0 senão. Essa operação normalmente é representada utilizando o símbolo $\cdot$ e sua tabela verdade é mostrada na Figura 2.1.

Operador *OR*

O operador *OR* tem como sua saída 1 se quaisquer das entradas for 1, senão é definida como 0. Sua tabela verdade pode ser escrita conforme mostra a tabela na Figura 2.2. Associa-se a este operador o símbolo +.

Operador *NOT*

A operação *NOT* é a simples inversão do valor passado. Portanto, se sua entrada for 1, a saída é 0 e vice-versa como mostra a tabela na Figura 2.3. Se uma variável é denominada A, então sua inversão é definida como $\overline{A}$.

2.1.2 Composição de Funções

Funções em Álgebra Booleana são criadas a partir de composições utilizando os três operadores básicos apresentados interespaçados pelas constantes 0, 1 ou variáveis. Um exemplo de função seria $f(A, B) = A \cdot \overline{B} + \overline{A} \cdot B$ cujo resultado se limita ao conjunto $\{0, 1\}$. Por consequência, uma característica favorável é conseguir representar essas funções a partir de tabelas verdade, como mostra a Tabela 2.1 para a função descrita. Essa função também é conhecida como o operador *XOR* (*eXclusive OR*) binário.

Tabela 2.1: Tabela verdade da função $f(A, B) = A \cdot \overline{B} + \overline{A} \cdot B$

A	B	$\overline{A}$	$\overline{B}$	$A \cdot \overline{B}$	$\overline{A} \cdot B$	$A \cdot \overline{B} + \overline{A} \cdot B$
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	1	0	1
1	1	0	0	0	0	0

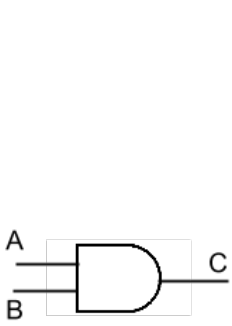
2.1.3 Circuitos Combinacionais

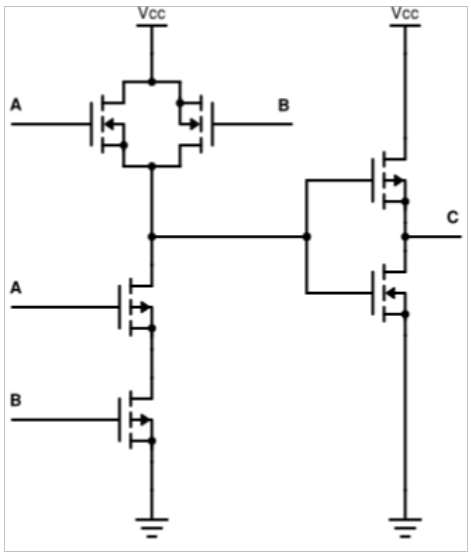
Outra maneira de representar funções booleanas é por meio de portas lógicas. Atualmente, circuitos digitais combinacionais, ou seja, aqueles cujas saídas dependem exclusivamente de suas entradas, são abstraídos dessa forma.

Representação física

Fisicamente, portas lógicas são implementadas utilizando-se elementos básicos de circuitos elétricos, como resistores, diodos e transistores [7]. Os níveis lógicos são mapeados para um intervalo de voltagem, onde um extremo assume o nível lógico 1 e o oposto, 0. Um exemplo desse intervalo seria $5V$ ser considerado o valor lógico 1 e $0V$, 0.

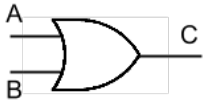
Utilizando as variáveis A e B para entradas e C para saída, as Figuras 2.1 a 2.7 mostram os símbolos das portas lógicas, uma possível construção física utilizando transistores MOS e suas respectivas tabelas verdade.

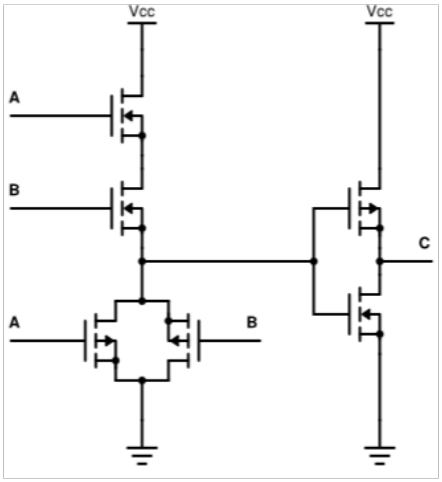




A	B	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

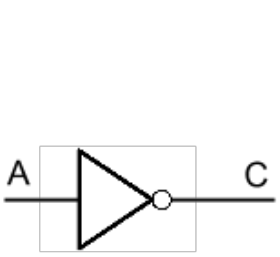
Figura 2.1: Porta lógica *AND*, circuito e tabela verdade.

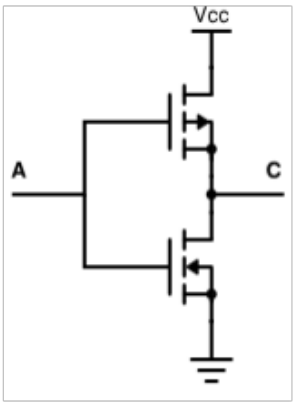




A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

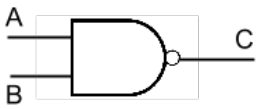
Figura 2.2: Porta lógica *OR*, circuito e tabela verdade.

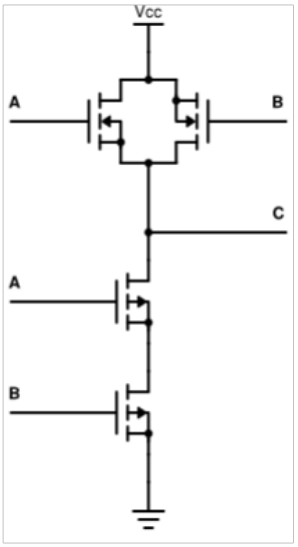




A	$\overline{A}$
0	1
1	0

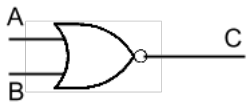
Figura 2.3: Porta lógica *NOT*, circuito e tabela verdade.

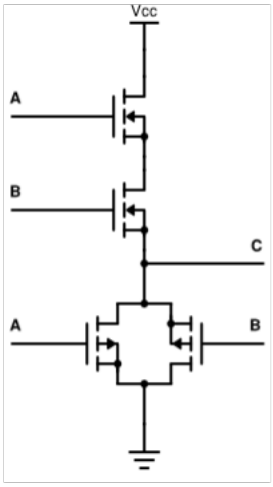




A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

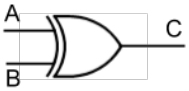
Figura 2.4: Porta lógica *NAND*, circuito e tabela verdade.

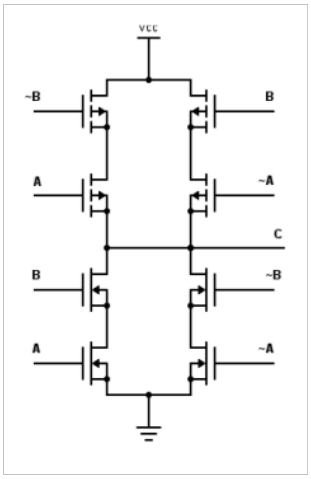




A	B	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

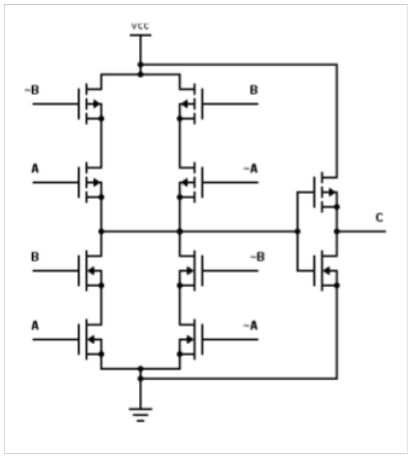
Figura 2.5: Porta lógica *NOR*, circuito e tabela verdade.





A	B	$\overline{A \oplus B}$
0	0	0
0	1	1
1	0	1
1	1	0

Figura 2.6: Porta lógica *XOR*, circuito e tabela verdade.



A	B	$\overline{A \oplus B}$
0	0	1
0	1	0
1	0	0
1	1	1

Figura 2.7: Porta lógica *XNOR*, circuito e tabela verdade.

Observa-se que, dependendo da porta lógica, a sua implementação pode requerer diferentes números de transistores, sendo que a porta *NOT* a mais simples, e as portas *XOR* e *XNOR* as mais complexas.

Tempo de propagação

Enquanto expressões booleanas são suficientes para expressar a funcionalidade de circuitos digitais combinacionais, uma característica importante para projetos mais complexos é o tempo de atraso de propagação (do Inglês *propagation delay time*) (tpd). Ele pode ser definido como o tempo necessário para que, após uma mudança na entrada do circuito, a saída atinja 50% de seu valor máximo, ou seja se torne válido, em 0 ou 1 [7].

Como exemplo, uma mudança na entrada A do circuito da Figura 2.8 terá seu resultado refletido na saída R somente após a propagação do sinal estável em cada uma das portas lógicas *NOT*, resultando em um tempo de propagação total de:

$$tpd_{total} = tpd_{N_1} + tpd_{N_2} + tpd_{N_3} + tpd_{N_4}$$

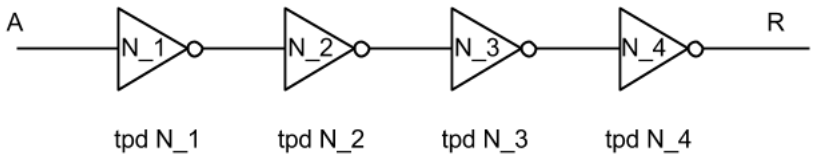


Figura 2.8: Circuito combinacional exemplificando tempo de propagação.

Deste modo, quanto mais camadas de portas lógicas são necessárias para a tarefa desejada, maior será o atraso para a obtenção da resposta, limitando assim a máxima frequência de utilização do circuito.

De acordo com [8], fios possuem aproximadamente um atraso de $1ns$ para cada $15cm$ de comprimento, enquanto uma porta lógica pode variar de cerca de $10ns$ até à casa de picossegundos, dependendo da tecnologia utilizada.

2.1.4 Autossuficiência

Autossuficiência para o caso de portas lógicas significa que é possível formar qualquer função booleana com uma combinação delas. Para isto, basta mostrar que é possível que sejam geradas as portas *AND*, *OR* e *NOT*, que compõem um dos conjuntos completos de operações, a partir de uma única porta lógica. Se isso for verdade, então, já que se pode representar todas as funções booleanas com apenas essas 3 funções [5], a porta lógica em questão também o faz. Esse tipo de característica também é conhecido como porta lógica universal.

É o caso da função *NAND*, funcionalmente idêntica a $f(A, B) = \overline{A \cdot B}$. Sua tabela verdade pode ser vista na tabela da Figura 2.4. O operador *NOT* é reproduzido a partir da função de apenas *NANDs* como $f(A) = \overline{A \cdot A}$. A função *OR* pode ser definida como $g(A, B) = \overline{\overline{A \cdot A} \cdot \overline{B \cdot B}}$. Para *AND*, define-se $h(A, B) = \overline{\overline{A \cdot B} \cdot \overline{A \cdot B}}$.

Outro exemplo é o da função *NOR* (tabela verdade na Figura 2.5). Sua representação em funções booleanas se dá por uma porta *OR* seguida de uma *NOT*, $f(A, B) = \overline{A + B}$. Para mostrar que *NOR* também possui essa completude funcional, sua representação para portas *OR* se dá como $\overline{\overline{A + B} + \overline{A + B}}$. Para portas *AND*, $\overline{\overline{A + A} + \overline{B + B}}$ e, *NOT*, $\overline{A + A}$.

Portas *NAND* e *NOR* são desejáveis não só por serem funcionalmente completas, mas também por possuírem a menor configuração CMOS quando comparadas a outras portas lógicas. Como consequência, elas são utilizadas no conjunto das portas básicas em muitas famílias de *IC*s de lógica digital [9].

2.1.5 Forma Canônica

A criação de funções booleanas se mostra ser bastante flexível e diversificada. Porém, nem todas são únicas. É possível que duas expressões lógicas distintas produzam a mesma saída, como por exemplo $f(A, B) = A \cdot B$ e $g(A, B) = \overline{\overline{A} + \overline{B}}$.

Neste caso, há uma infinidade de funções booleanas para uma dada tabela verdade. Torna-se necessário uma padronização visando apenas uma função booleana possível para cada configuração de tabela verdade.

Isto pode ser alcançado utilizando-se de formas canônicas de expressões lógicas. Há duas formas principais de para expressões canônicas, conhecidas como a forma soma de produtos e a forma produto de somas [7]. A forma de utilização mais comum é o formato de soma de produtos, também chamado de soma de *min-termos*.

Para um circuito com entradas $A_1, A_2, \ldots, A_n$, um min-termo é um produto em que cada variável de entrada ou seu complemento aparecem apenas uma vez [5]. Para um circuito com entradas A, B, C e D, um mini-termo possível é $A \cdot B \cdot \overline{C} \cdot D$, enquanto que $\overline{A} \cdot \overline{B} \cdot C$ não pode ser considerado um. Para a tabela verdade vista na Tabela 2.2, só há a função canônica $f(A, B, C) = A \cdot B \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C}$ que a representa.

Tabela 2.2: Tabela verdade com mini-termos da função canônica $f(A, B, C) = A \cdot B \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot \overline{C}$

A	B	C	$f(A, B, C)$	Mini-termo
0	0	0	1	$\overline{A} \cdot \overline{B} \cdot \overline{C}$
0	0	1	0	
0	1	0	0	
0	1	1	0	
1	0	0	0	
1	0	1	0	
1	1	0	1	$A \cdot B \cdot \overline{C}$
1	1	1	0	

2.1.6 Minimizando Funções Booleanas

Muitas vezes, quando se projeta um circuito combinacional, é desejável obter sua função mínima, ou seja aquela cujo número de operações é o menor possível, para que seja utilizado apenas o necessário, economizando recursos. Para isto existem técnicas de minimização como *Mapas de Karnaugh*, *Método de Quine-McCluskey*, dentre outros [10].

Para otimizações de implementação, outras técnicas têm surgido, principalmente envolvendo conceitos de programação genética que serão discutidas mais adiante neste trabalho. Esta seção se limita somente à minimização lógica.

Mapas de *Karnaugh*

Mapas de *Karnaugh* são uma maneira de representar uma tabela verdade de tal modo que ela exponha características redundantes de um circuito. Eles se utilizam da forma canônica para suas construções. Suas células são compostas de valores 0 ou 1, sendo cada 1 um mini-termo desejado da função. Como exemplo, a função $f(A, B, C) = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$

tem uma tabela verdade como mostra a tabela 2.3 e sua representação em mapa de *Karnaugh* se dá como mostra a tabela 2.4.

Tabela 2.3: Tabela verdade com mini-termos da função $f(A, B, C) = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$

A	B	C	$f(A, B, C)$	Mini-termo
0	0	0	0	
0	0	1	1	$\overline{A} \cdot \overline{B} \cdot C$
0	1	0	0	
0	1	1	1	$\overline{A} \cdot B \cdot C$
1	0	0	0	
1	0	1	1	$A \cdot \overline{B} \cdot C$
1	1	0	0	
1	1	1	0	

Tabela 2.4: Mapa de *Karnaugh* de $f(A, B, C) = \overline{A} \cdot B \cdot C + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot C$

$f(A, B, C)$:

B \ A,C	00	01	11	10
0	0 (0)	1 (1)	1 (5)	0 (4)
1	0 (2)	1 (3)	0 (7)	0 (6)

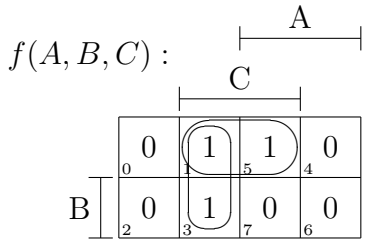
A Tabela 2.5 exibe células que podem ser otimizadas da Tabela 2.4. Isso ocorre pois o mapa de *Karnaugh* foi projetado para que células adjacentes difiram somente em um valor da entrada. Dessa forma, é possível retirar termos desnecessários da expressão lógica, se utilizando do fato que $A + \overline{A} = 1$ e $B + \overline{B} = 1$.

Uma das otimizações marcadas na Tabela 2.5 que pode ser feita é sobre os termos $\overline{A} \cdot B \cdot C$ e $\overline{A} \cdot \overline{B} \cdot C$. Em ambos, a única variável que se altera é B. Por causa disso, é possível juntá-los, resultando no termo $\overline{A} \cdot C$. O mesmo ocorre para $\overline{A} \cdot \overline{B} \cdot C$ e $A \cdot \overline{B} \cdot C$, ao retirar a variável A de ambos, o resultado é $\overline{B} \cdot C$.

No final, após todas as otimizações terem sido feitas, somam-se todos os termos gerados. No exemplo da Tabela 2.5, seu circuito mínimo é $\overline{A} \cdot C + \overline{B} \cdot C$.

A principal desvantagem de se usar mapas de *Karnaugh* é que o algoritmo funciona bem para até 6 variáveis de entrada, tornando-se impraticável para valores maiores. O número excessivo de células dificulta uma seleção razoável de saídas adjacentes [9].

Tabela 2.5: Mapa de *Karnaugh* exibindo células passíveis de otimização.



Método de *Quine-McCluskey*

Uma função booleana também pode ser expressa de uma forma numérica. Cada mini-termo é representado por um número, sendo ele a conversão binária de seus dígitos para decimal. Por exemplo, o mini-termo de quatro variáveis $\overline{A} \cdot B \cdot C \cdot D$ é expressado como 0111 em base binária, resultando no número 7 na base decimal.

Uma função booleana, portanto, também pode ser expressa como um somatório de mini-termos em sua representação decimal. Usando como exemplo a função da Tabela 2.3, temos que ela pode ser expressa como $m_1 + m_3 + m_5$, também sendo representada como $\sum m(1, 3, 5)$.

O método de *Quine-McCluskey*, também conhecido como método dos implicantes primos, visa amenizar a dificuldade vista para o método de *Karnaugh* para otimização de funções com mais de seis variáveis. Ele consiste de duas partes: a primeira é achar todos os termos implicantes primos, ou seja, aqueles que são candidatos para inclusão no resultado final simplificado. A segunda é escolher, dentre esse termos, os que dão a expressão com o menor número de implicantes primários [9].

Para determinar quais termos são primos implicantes, cada mini-termo é comparado com todos os outros. Se eles diferem em apenas uma variável, ela é removida e um novo termo é formado. Isso se repete até que não seja mais possível combinar mais termos.

A Tabela 2.6 demonstra isso para a função da Tabela 2.3.

Tabela 2.6: Tabela de implicantes primos para a função $\sum m(1, 3, 5)$.

Número de 1s	Mini-termo	Binário	Implicante (Tamanho 2)
1	m_1	001	$m(1, 3)0 - 1$ $m(1, 5) - 01$
2	m_3 m_5	011 101	

O passo seguinte monta o gráfico dos implicantes primos, como pode ser visto na Tabela 2.7.

Tabela 2.7: Gráfico de implicantes primos para a função $\sum m(1,3,5)$. Construído após o primeiro passo do algoritmo de *Quine-McCluskey*.

	1	3	5	Implicante	Expressão
$m(1,3)$	X	X		$0-1$	$\overline{A}\cdot C$
$m(1,5)$	X		X	-01	$\overline{B}\cdot C$

Ao final, todos os termos encontrados são somados. Para este exemplo, como visto na Tabela 2.7, o resultado é $\overline{A}\cdot C + \overline{B}\cdot C$, chegando então na mesma expressão encontrada pelo método de *Karnaugh*.

2.2 Lógica Sequencial

Até agora, a saída dos circuitos vistos na seção anterior dependiam apenas de suas entradas. Circuitos sequenciais utilizam-se tanto de circuitos combinacionais quanto de circuitos de retroalimentação. Isso possibilita a aplicação de uma abstração chamada *estado*. Dessa forma, torna-se muito comum a utilização e produção de chamadas *máquinas de estado*. A Figura 2.9 mostra esse tipo de abordagem.

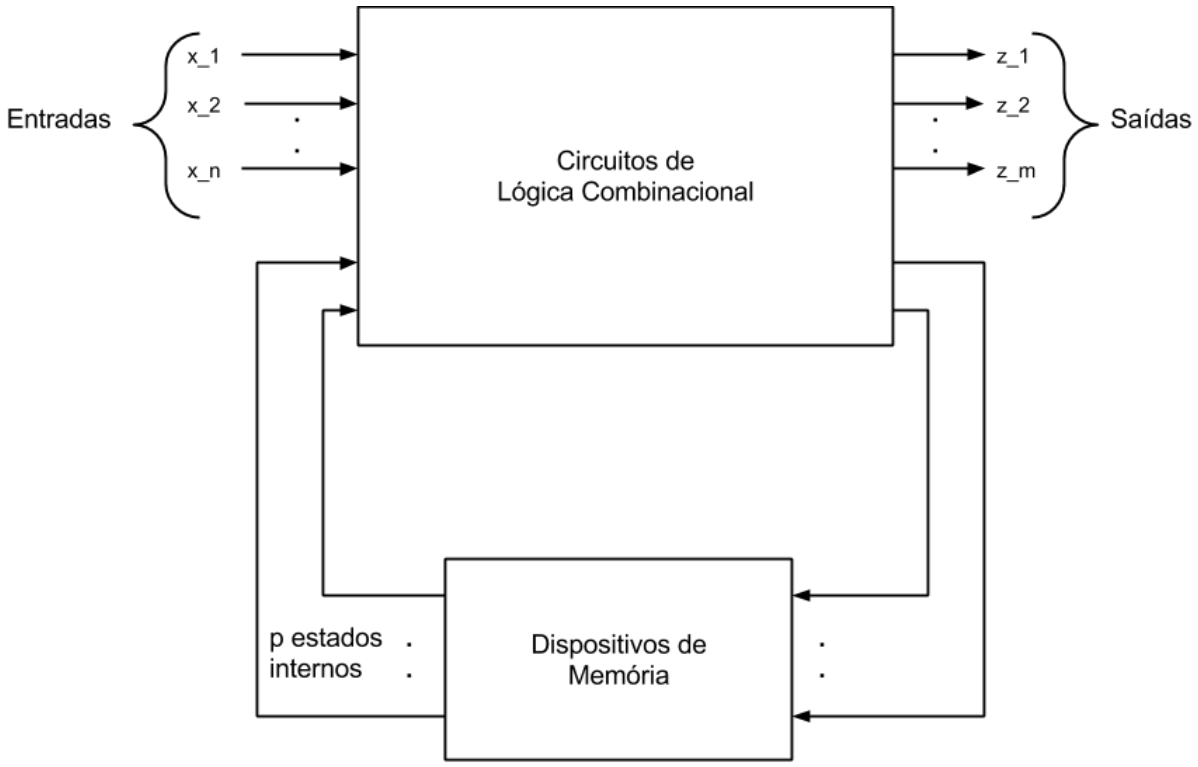


Figura 2.9: Estrutura conceitual de uma máquina de estados.

Um exemplo comum de circuito sequencial é o contador. Um contador é utilizado para a numeração de tíquetes de uma fila bancária, por exemplo.

2.2.1 Elementos Básicos

Para que circuitos consigam manter estados, uma estrutura básica denominada *flip-flop* é utilizada. Nesta seção, será mencionado apenas o *flip-flop* tipo D, porém outros tipo podem ser encontrados.

O *flip-flop* é um elemento capaz de guardar o valor de 1 *bit* [11]. Um circuito retroalimentado, também conhecido como um *loop combinacional*, pode ser usado para alcançar este objetivo. A Figura 2.10 apresenta um pequeno circuito realimentado composto por duas portas *NOT*.

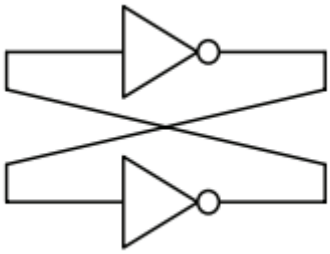


Figura 2.10: Estrutura realimentada estável independente de entradas (Fonte: [11]).

Independentemente do estado inicial do circuito, observa-se que o circuito permanece estável, isto é, sua saída não se altera. Como esse circuito é estável nos dois níveis lógicos de saída chama-se de *loop* combinacional bi-estável, ou simplesmente estável neste trabalho. A desvantagem desse circuito é a impossibilidade de alteração do valor armazenado, já que é completamente auto-contido.

Um exemplo de um circuito cuja saída não é estável, ou seja, trata-se de um *loop* combinacional instável, é mostrado na Figura 2.11, onde sua saída oscila entre 0 e 1 de acordo com as características construtivas físicas dos transistores que compõem a porta *NOT*.

A porta *NAND*, apresentada na seção 2.1.4, atua como um simples inversor se uma de suas saídas for mantida em 1. Dessa maneira, é possível replicar a estrutura da Figura 2.10 utilizando portas *NAND* como mostra a Figura 2.12.

Alterando os valores de uma das entradas, é possível alterar a saída das portas lógicas, com o circuito se estabilizando novamente no momento em que as duas entradas retornarem para 1.

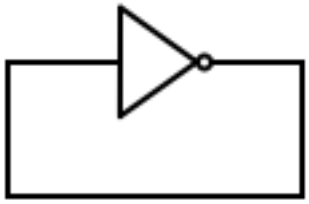


Figura 2.11: Estrutura realimentada instável, com saída oscilante.

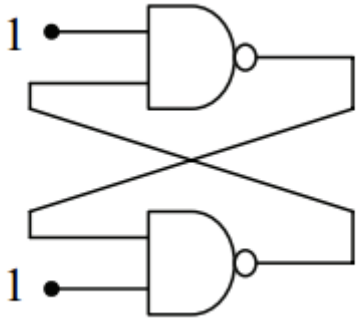


Figura 2.12: Estrutura de retroalimentação utilizando portas *NAND* (Fonte: [11]).

Caso a entrada superior receba o rótulo S e a inferior R, correspondente às saídas Q e $\overline{Q}$, o elemento é chamado de *latch SR*, e é utilizado na maior parte de circuitos que necessitam armazenar dados. Seu diagrama pode ser visto na Figura 2.13.

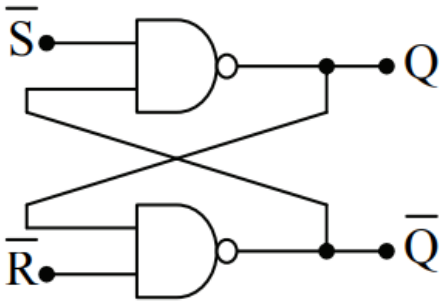


Figura 2.13: Diagrama de um *latch SR* (Fonte: [11]).

Latch **D**

Um possível módulo que pode ser formado pelo *latch RS* é o *latch* tipo D, que tem a interface definida na Figura 2.14, onde a saída Q é o valor armazenado no circuito, E é o

valor que determina se o valor D é armazenado no *latch*.

A entrada E funciona como uma porta. Se estiver no nível 1 (aberta), qualquer alteração no valor D causa uma alteração na saída do circuito. Se estiver em 0 (fechada), o valor da saída se mantém estável mesmo que o valor D se altere. Seu comportamento completo pode ser visto na Tabela 2.8.

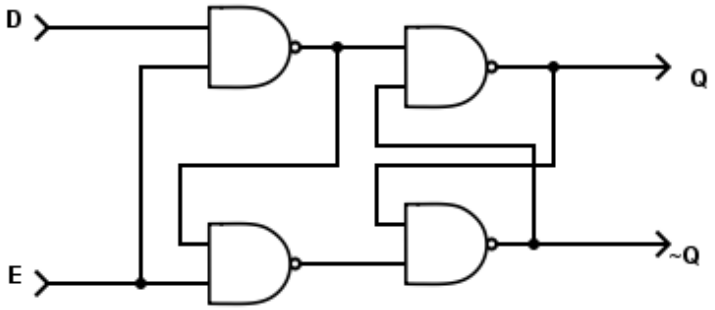


Figura 2.14: Diagrama de um *latch D*.

Tabela 2.8: Tabela verdade da estrutura *latch* D.

Entradas		Saídas	
E	D	Q	$\overline{Q}$
0	X	Q	$\overline{Q}$
1	0	0	1
1	1	1	0

Flip-flop **D**

Latches são normalmente categorizadas como sensíveis a nível, ou seja, se seu nível de *enable* estiver em alto, todas as alterações de D são vísiveis na saída do *latch*. *Flip-flops*, diferentemente, só causam alterações na saída nas transições $0 \rightarrow 1$ ou $1 \rightarrow 0$ do *enable*, também chamado de *clock* no caso de *flip-flops*. A essas duas categorias dão-se os nomes de *positive edge triggered* e *negative edge triggered* [11].

A estrutura interna de um *flip-flop* D de *negative edge*, ou seja, com mudança de saída apenas durante a transição $1 \rightarrow 0$, pode ser vista na Figura 2.15.

Este tipo de *flip-flop*, em especial, é chamado de *master-slave*. Com o circuito montado dessa maneira, sua tabela verdade pode ser vista na Tabela 2.9.

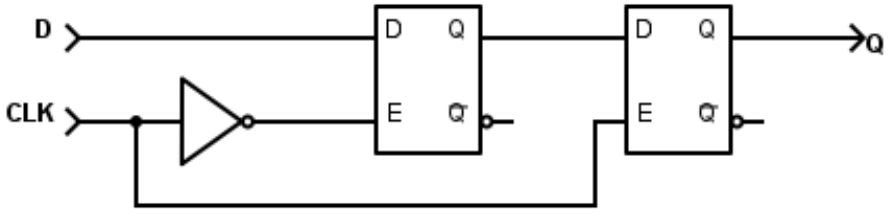


Figura 2.15: Diagrama de um *flip-flop D*.

Tabela 2.9: Tabela verdade do *flip-flop* tipo D.

Entradas		Saídas	
Clock	D	Q	$\overline{Q}$
$0 \rightarrow 1$	0	0	1
$0 \rightarrow 1$	1	1	0
0	X	Q	$\overline{Q}$
1	X	Q	$\overline{Q}$

2.2.2 Máquina de Estados Finitos

A abstração e modelagem de problemas que se utilizam de circuitos sequenciais normalmente se dá por meio das chamadas máquinas de estados.

Uma máquina de estados pode ser definida como uma quíntupla $M = (Y, X, Z, \alpha, \beta)$. Y é o conjunto de todos os estados existentes na máquina. X é o conjunto de entradas para a máquina. Z, o conjunto de todos os estados de saída possíveis da máquina. α é a função $\alpha : (Y \times X) \rightarrow Y$, que define o próximo estado com base no estado atual e entrada atual. A função β define a saída atual da máquina, sua definição formal depende da classificação da máquina como Moore ou Mealy [12].

Máquina de Moore

Para máquinas de Moore, a função β é definida como:

$$\beta : Y \rightarrow Z,$$

ou seja, a saída atual do circuito depende apenas de seu estado atual.

Máquina de Mealy

No caso de máquinas de Mealy, a saída atual da máquina depende tanto do estado atual quanto da entrada atual. Ou seja, sua função β é da forma

$$\beta : (Y \times X) \rightarrow Z.$$

Comparação

Máquinas de estados são frequentemente representadas utilizando grafos especiais chamados Diagramas de Transições e Estados [12]. Um exemplo de uma máquina de Mealy com uma entrada e duas saídas pode ser visto na Figura 2.16, onde cada nó representa um estado (com seu rótulo) e os números nas arestas indicam as entradas e as saídas associadas. Sua tabela de transições associada está descrita na Tabela 2.10.

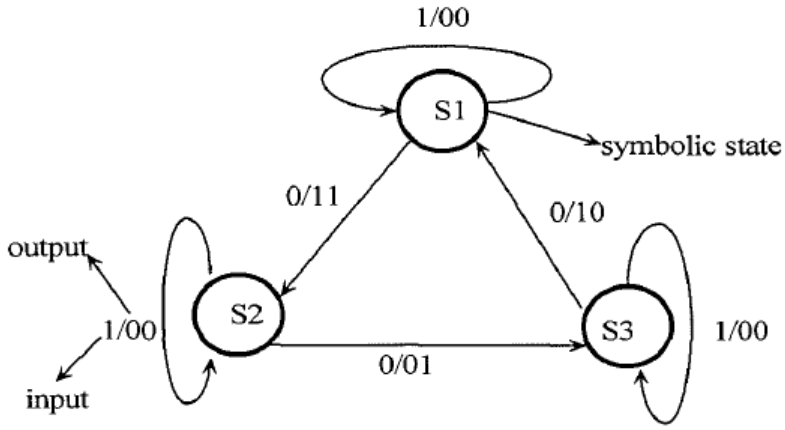


Figura 2.16: Exemplo de Diagrama de Transições e Estados (Fonte: [12]).

Tabela 2.10: Exemplo de tabela de transição de estados.

Entrada	Estado Atual	Próx. Estado	Saída
0	S_1	S_2	11
0	S_2	S_3	01
0	S_3	S_1	10
1	S_1	S_1	00
1	S_2	S_2	00
1	S_3	S_3	00

A mesma máquina implementada utilizando o paradigma de Moore teria um diagrama de transições e estados como mostra a Figura 2.17, com uma tabela de transições descrita na Tabela 2.11.

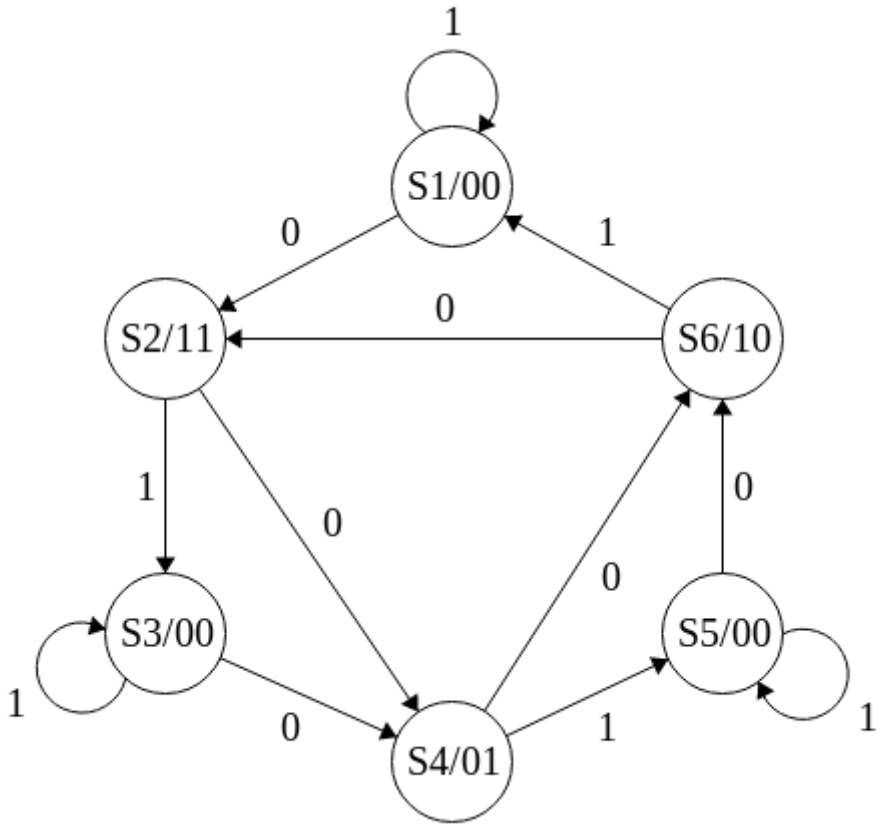


Figura 2.17: Exemplo de Diagrama de Transições e Estados utilizando o paradigma de Moore.

Tabela 2.11: Tabela de transições de estados para o exemplo utilizando o paradigma de Moore.

Estado Atual/Saída	Prox. Estado	
	0	1
$S_1/00$	S_2	S_1
$S_2/11$	S_4	S_3
$S_3/00$	S_4	S_3
$S_4/01$	S_6	S_5
$S_5/00$	S_6	S_5
$S_6/10$	S_2	S_1

Em geral, máquinas de Moore possuem uma saída mais simples de ser entendida, já que depende apenas de um fator. Além disso, são mais seguras pois se a entrada for assíncrona com o *clock* da máquina, então ocorrem condições de corrida [8]. Em contrapartida, máquinas de Mealy são significativamente mais compactas, possuindo menos estados em geral.

2.2.3 Processo de *Design*

O processo de projeto de um circuito sequencial, assim como a abordagem tradicional para outros problemas, é dividir a especificação em sub-etapas [12]. Cada uma deve ter, idealmente, pouca ou nenhuma relação com as outras. O objetivo é conseguir, a partir da especificação de um problema, desenvolver uma máquina de estados finitos que o resolva. Um esquema de como isso é normalmente feito está na Figura 2.18.

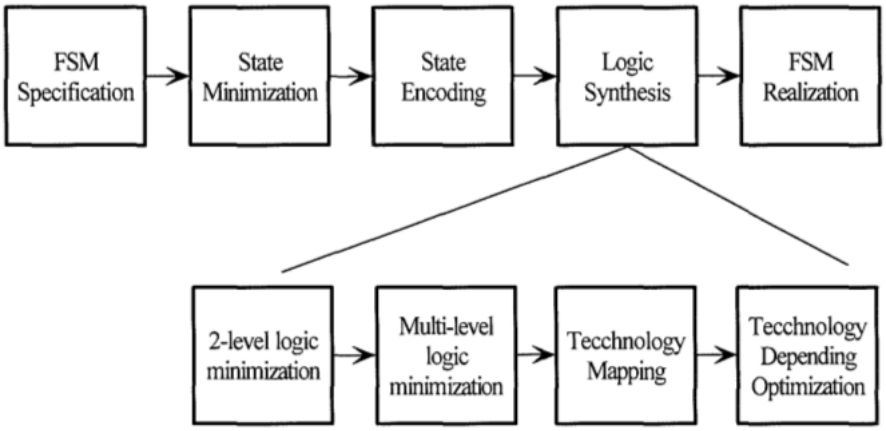


Figura 2.18: Processo de projeto de um circuito sequencial (Fonte: [12]).

A razão disso vem do fato de que o problema de sintetizar um circuito sequencial é um problema considerado difícil. Características tais como grande quantidade de possíveis implementações possíveis para um dado problema, além de outros fatores como área que o circuito ocupa e seu gasto de energia para a determinação de uma boa solução [12].

Especificação da Máquina

Este passo consiste na configuração inicial desenvolvida pelos projetistas do circuito. Normalmente contém o diagrama de estados e transições com uma tabela de transições de estados.

Minimização de Estados

Muitas vezes, a máquina inicialmente gerada não possui um número ótimo de estados. Um método clássico para realizar isso é incorporar *don't cares*, ou seja, valores que em um dado instante não são importantes, na tabela de transições e, consequentemente, no diagrama de transições e estados.

Em contrapartida, outras abordagens de minimização focam não só no número de estados mas na complexidade lógica da minimização ótima, porém nem sempre é desejável se ter o número mínimo de estados [13].

Codificação de Estados

Em implementações de máquinas de estados, os estados são representados por *strings* de *bits*. O problema que estuda a relação de mapeamento entre estados e *strings* de *bits* que possui o menor custo é chamado de Problema da Codificação de Estados [14].

Para essa etapa, diversos autores já tentaram diferentes abordagens [12]. Em geral, são utilizadas heurísticas para solucionar esse tipo de problema. Dentre elas, algoritmos genéticos foram usados com sucesso [14], porém o problema ainda é considerado difícil.

Síntese do Circuito

Atualmente, existe uma vasta gama de ferramentas dedicadas a síntese a partir de uma descrição da máquina. As formas mais comuns de se descrever circuitos sequenciais são pelas linguagens denominadas Linguagem de Descrição de Hardware (do Inglês *Hardware Description Language*) (HDL), como *VHDL* e *Verilog* [12].

As máquinas são simuladas e testadas em dispositivos específicos denominados reconfiguráveis. Com os objetivos alcançados, é feito um Circuito Integrado de Aplicação Específica (do Inglês *Application-Specific Integrated Circuit*) (ASIC) a partir da descrição resultante.

A seguinte seção trata de um dos tipos mais utilizados de dispositivos reconfiguráveis, extremamente útil para o desenvolvimento de circuitos não só sequenciais, mas de forma geral.

2.3 Dispositivo Reconfigurável - FPGA

Um *Field-Programmable Gate Array* (FPGA) é um dispositivo lógico que consiste de um arranjo bidimensional de chamadas células lógicas e chaves programáveis. Cada célula lógica pode ser programada para exercer uma função booleana, cujo valor é disponibilizado para células vizinhas através dessas chaves [15]. Células especiais, principalmente as que estão na borda do arranjo, se conectam a portas destinadas a canais de I/O.

Em um dado momento, a configuração do FPGA é determinada por uma memória *on-chip*, que pode ser modificada por *software*. Apesar da configuração do circuito não se modificar, as mudanças realizadas a essa memória fazem com que as ligações entre

células lógicas e suas configurações se modifiquem, portanto, exercendo a função de um novo circuito [16], como mostra a Figura 2.19.

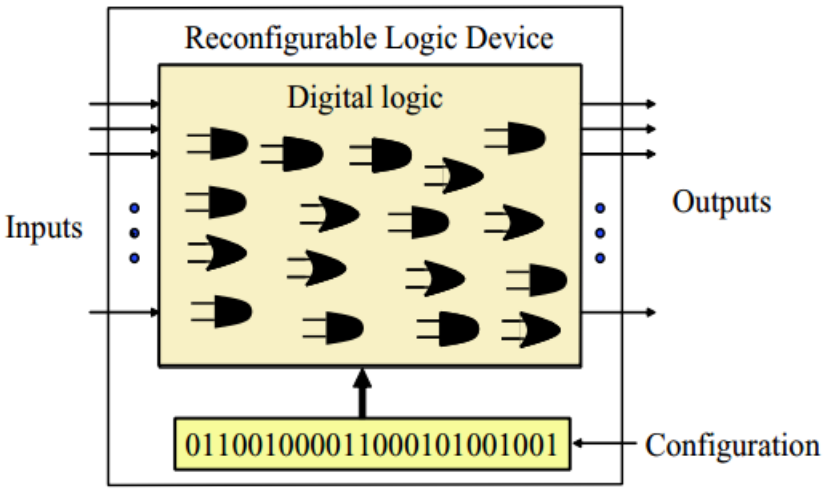


Figura 2.19: Configuração de memória mapeando para um circuito em um dispositivo reconfigurável (Fonte: [17]).

Cada célula lógica tipicamente consiste de uma *Look-up Table* (LUT) configurável de 3 ou 4 entradas, ou seja uma tabela verdade que descreve uma função booleana, e um *flip-flop* tipo-D. Por exemplo, a célula das famílias de FPGAs *Cyclone II*, *Cyclone IV* e *Cyclone V* podem ser vistas, respectivamente, nas Figuras 2.20 a 2.22.

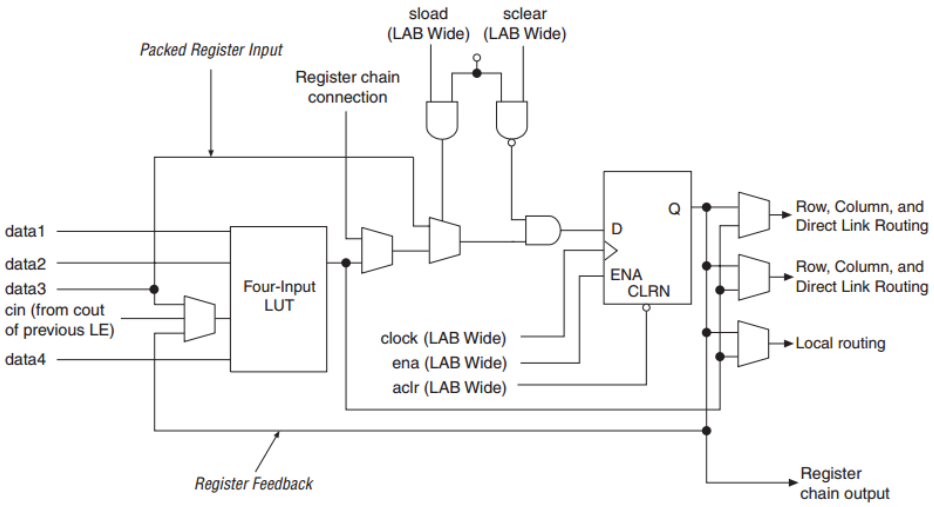


Figura 2.20: *Logic Element* (LE) da família de FPGAs *Cyclone II* (Fonte: [18]).

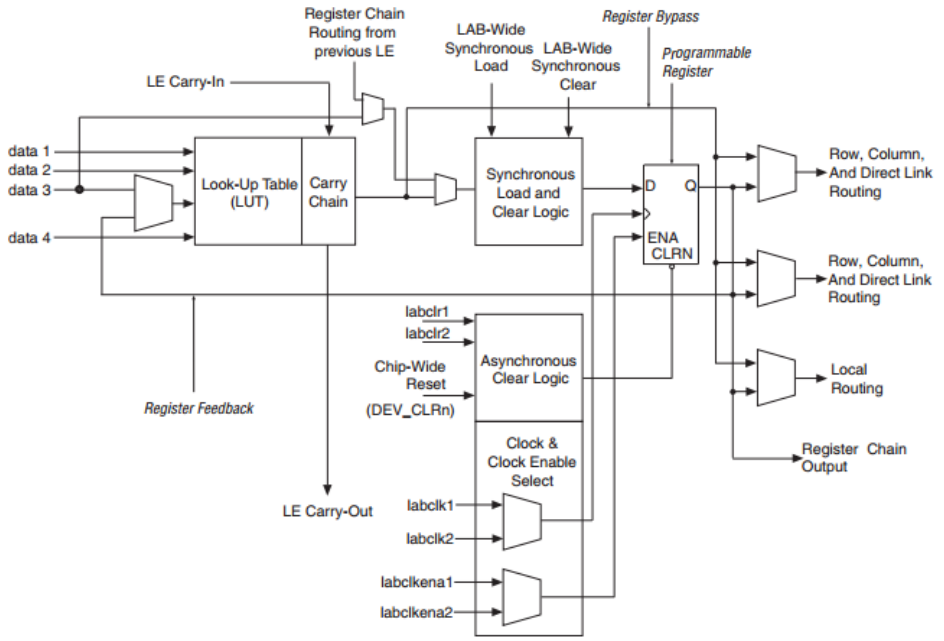


Figura 2.21: *Logic Element* (LE) da família de FPGAs *Cyclone IV* (Fonte: [19]).

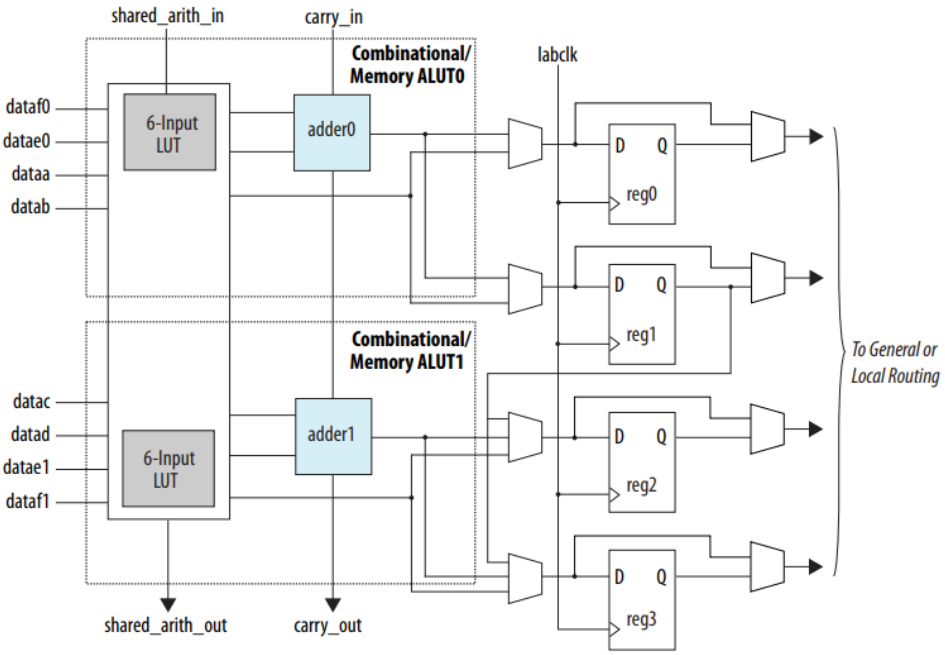


Figura 2.22: *Logic Element* (LE) da família de FPGAs *Cyclone V* (Fonte: [20]).

FPGAs pertencem a uma classe de dispositivos chamada *Field-Programmable Logic* (FPL) [21], pois o processo de programação pode ser feito várias vezes após a produção do dispositivo.

O Figura 2.23 mostra conceitualmente como o arranjo de células é feito.

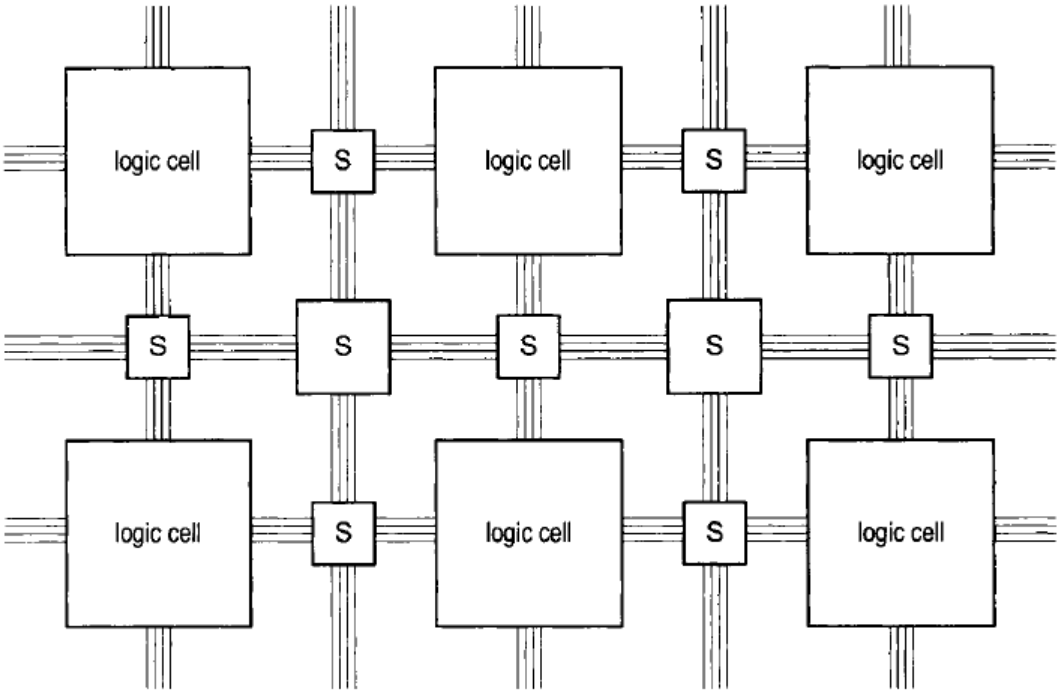


Figura 2.23: Arranjo conceitual de elementos lógicos e chaves programáveis (Fonte: [15]).

Existe uma variedade grande de diferentes tipos de dispositivos reconfiguráveis. Para este trabalho, o foco será apenas em FPGAs por suas facilidades de reconfiguração apresentadas.

Processo de *Design*

Empresas fabricantes de FPGAs tipicamente oferecem *softwares* que facilitam o desenvolvimento em suas plataformas. Em geral, esses programas têm a responsabilidade de auxiliar a etapa de compilação (exemplo na Figura 2.24) da descrição de um circuito para o *bitstream* de configuração específico de algum FPGA, passando pelas fases de análise, síntese e montagem do projeto. Para os chips do fabricante *Intel*, esse *software* de desenvolvimento se chama *Quartus Prime*.

Após a compilação do circuito, esses programas costumam oferecer também mecanismos de *download* das descrições geradas para o *chip*, configurando-o com a funcionalidade desejada.

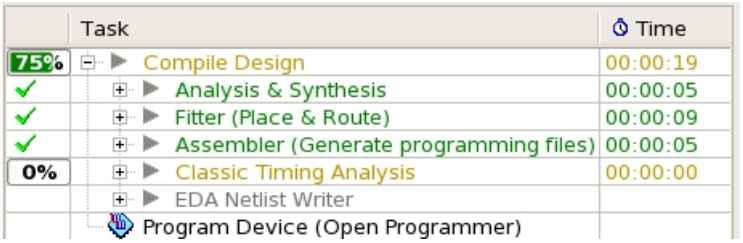


Figura 2.24: Etapas típicas de compilação para *bitstream* alvo.

2.4 Algoritmos Genéticos

As características das espécies são determinadas não por intervenções divinas como se acreditava na época anterior à publicação desse trabalho, mas sim por um mecanismo denominado *seleção natural* [2].

Indivíduos mais adaptados de uma população têm maiores chances de se reproduzirem. Como consequência disso, seus genes têm uma probabilidade maior de serem passados para as gerações seguintes. Devido a isso, a cada nova geração, os genes da população tendem a se convergir para aquele que está mais adaptado ao ambiente.

Pode se ver que, aplicado em escalas de milhões de anos, esse mecanismo de seleção natural exibe características bastante interessantes nas espécies. Há muito se busca uma maneira de imitar esse tipo de comportamento e aplicá-lo a outros tipos de problemas, dando origem à área de Algoritmos Genéticos.

Conceito inventado por John Holland na década de 60 [4], seu objetivo era estudar o fenômeno de adaptação que ocorre na natureza visando desenvolver mecanismos similares para sistemas computacionais.

Antes de aplicar uma abordagem genética a um problema, é preciso torná-lo apto a isso. Um dos primeiros requisitos é conseguir descrevê-lo com base em todos os parâmetros necessários, ou seja, suas possíveis soluções. Estas são denominadas os indivíduos da população. Cada indivíduo é representado por um cromossomo, ou seja, um conjunto de genes, que formam os parâmetros mencionados. Por sua vez, genes são representados por um conjunto de "alelos" possíveis, que no contexto computacional pode ser descrito como os dois valores válidos para um *bit*: 0 e 1. Dessa forma é possível resumir um indivíduo em um arranjo de 0s e 1s.

2.4.1 Operadores Genéticos

Nesta seção, serão introduzidos operações comumente utilizadas em algoritmos genéticos.

Reprodução ou *Crossover*

Esta operação busca imitar o fenômeno da recombinação genética. O *crossover* de um ponto é aquele em que um ponto de separação do cromossomo é escolhido aleatoriamente e dois novos indivíduos são gerados a partir da permuta da informação genética entre os dois cromossomos selecionados. A Figura 2.25 ilustra essa operação.

Entre outros tipos de *crossover*, há aquele também em que dois pontos são selecionados e um segmento genético é trocado entre os cromossomos pais, gerando seus filhos, conhecido como o *crossover* de dois pontos.

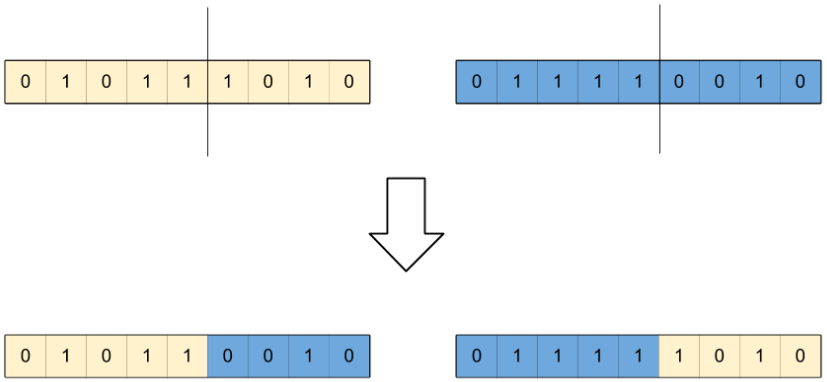


Figura 2.25: Exemplo de *crossover* entre dois cromossomos.

Mutação

A mutação visa simular a situação em que o cromossomo é alterado por erros durante a divisão celular. Essas mudanças são essenciais para o surgimento de novas características.

No caso de algoritmos genéticos, essa mudança se representa por uma probabilidade p_m. Em geral, a mutação atua na construção de novos indivíduos. Ou seja, em sua criação, há uma probabilidade de p_m de haver uma mutação. Mutações costumam alterar um ou mais alelos aleatórios como mostra a Tabela 2.12.

Tabela 2.12: Exemplo de operação de mutação em um cromossomo.

Cromossomo Inicial	0110010111001011
Cromossomo Mutante	0110010111011011

Função de *fitness*

As funções de *fitness* têm como entrada um cromossomo e saída um valor de avaliação, dependendo do quão perto ele chega da solução, ou seja, atuam definindo aqueles que

estão mais ou menos aptos a ter seus genes passados adiante. A função de *fitness* é considerada um dos pontos mais sensíveis de um algoritmo genético.

Uma vez definidos os operadores básicos, um algoritmo genético pode ser definido na série de passos apresentada na Figura 2.26.

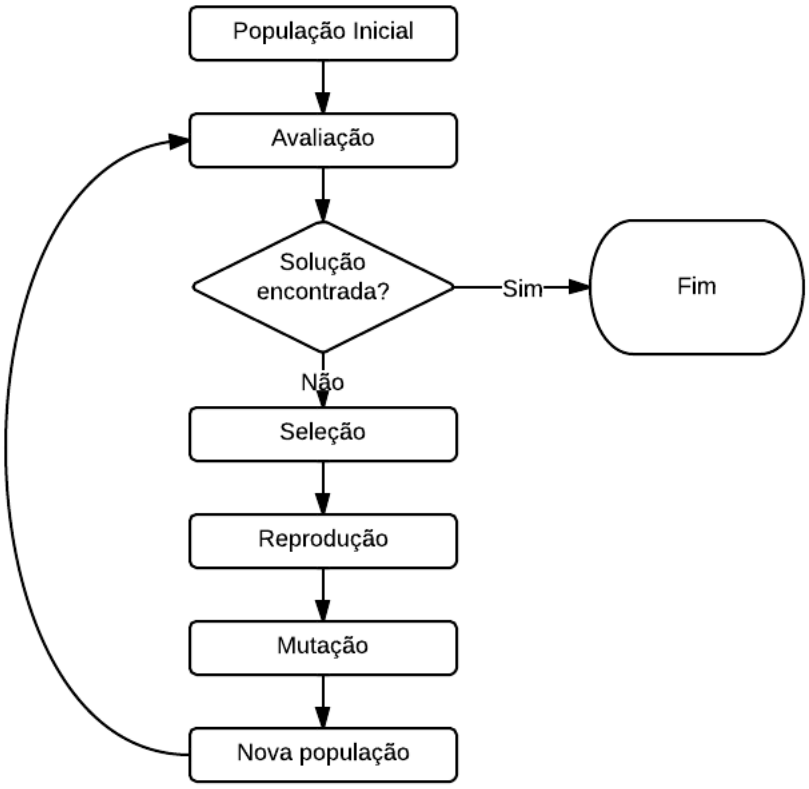


Figura 2.26: Passos gerais para um algoritmo genético.

2.4.2 Operação de Seleção

Essa operação consiste em selecionar cromossomos mais adaptados para a reprodução. Quanto mais adaptado, maior é sua chance de ser escolhido, mas não há garantia de que isso ocorra. Dentre os diversos métodos de seleção, os mais populares são as estratégias de seleção por roleta ou por torneio descritos a seguir.

Seleção por Roleta

A seleção por roleta se assemelha bastante a uma roleta real de cassino, de onde seu nome é derivado. Em geral, nela, indivíduos que possuem *fitness* maiores possuem uma maior probabilidade de serem escolhidos, porém não se garante a escolha. A probabilidade é

distribuída relativamente para cada indivíduo, ou seja, a soma de todas as probabilidades deve ser 1, e deve ser proporcional ao *fitness* de cada um.

Dessa maneira, a forma de se calcular o *fitness relativo* do n-ésimo indivíduo é dada por

$$f_r(x_n) = \begin{cases} \frac{f(x_n)}{\sum_{i=1}^{N} f(x_i)}, & \text{se } n = 1 \\ f_r(x_{n-1}) + \frac{f(x_n)}{\sum_{i=1}^{N} f(x_i)}, & \text{senão} \end{cases} \tag{2.1}$$

onde N é o tamanho da população.

Após o cálculo dos *fitness* relativos, gera-se um número aleatório $0 \leq r < 1$, escolhendo aquele indivíduo em que r pertence ao intervalo entre um *fitness* relativo e outro. Uma representação visual deste processo pode ser visto na Figura 2.27.

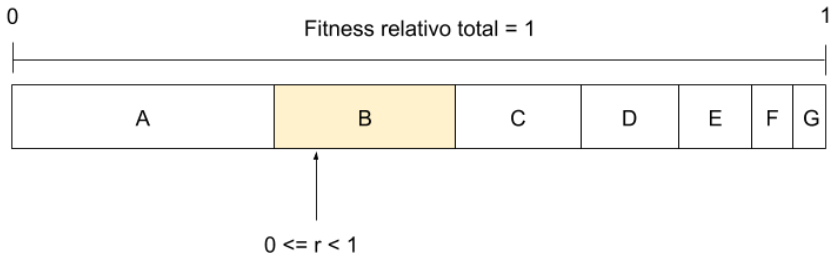


Figura 2.27: Exemplo da seleção por roleta.

Na Figura 2.27, o individuo A possui o maior *fitness*, tendo portanto uma maior probabilidade de ser selecionado. Por outro lado, o indivíduo G é o menos adaptado, possuindo a menor probabilidade de seleção.

Seleção por Torneio

A seleção por torneio busca simular o fato de que uma população inteira, salvo poucas exceções, nunca está competindo contra si como um todo, mas sim por pequenas competições localizadas.

Algumas vantagens de implementações desse tipo de seleção são: a facilidade de escrita, eficiência do ponto de vista de paralelismo e providenciam um maior controle sobre a *pressão da seleção* [22]. Essa pressão é definida como o grau em que indivíduos mais adaptados são favorecidos. Trabalhos anteriores [22] também citam que o sucesso de um algoritmo genético depende, em maior parte, da pressão da seleção. Se a pressão for muito baixa, a taxa de convergência será desnecessariamente baixa. Se for muito alta, é grande a chance de que a convergência ocorra em uma solução não-ideal.

Dada uma população P, e um tamanho de torneio k, a seleção por torneio seleciona um conjunto aleatório de k indivíduos de P para competirem entre si, e, para cada torneio,

escolhendo aquele mais adaptado. Seu pseudo-código correspondente pode ser visto no algoritmo 1.

Algoritmo 1 Algoritmo para seleção de um indivíduo por torneio.

```
1: procedure TORNEIO(P, k)
2:     for i ← 1, k do
3:         competidor ← P[random()]
4:         if competidor.fitness > melhor.fitness then
5:             melhor ← competidor
6:         end if
7:     end for
8:     return melhor
9: end procedure
```

A escolha do melhor indivíduo de uma população é realizada pela simples escolha do indivíduo com maior *fitness* em torneio de tamanho k.

Apesar do tamanho ideal do torneio variar de acordo com o problema [23], normalmente são escolhidos valores não muito altos para k. De fato, muitas vezes $k = 2$ é o valor utilizado, chamado de torneio binário. Observe que para $k = 1$, a seleção por torneio se reduz a uma seleção aleatória.

Estratégia evolucionária $(1 + \lambda)$

O algoritmo $(1 + \lambda)$ para geração e seleção de uma nova população vem sendo utilizado pela comunidade científica [24][25] como escolha de estratégia evolutiva por ser eficiente e de implementação simples [26]. Seu nome vem do fato de haver somente 1 indivíduo, gerando λ outros indivíduos por mutações. Seu algoritmo está descrito no algoritmo 2.

Algoritmo 2 Algoritmo que descreve a estratégia $(1 + \lambda)$.

```
1: procedure ONEPLUSLAMBDA(B)
2:     C ← λ mutações de B
3:     M = {c|c ∈ C, c.fitness ≥ B.fitness}
4:     if M ≠ ∅ then
5:         B ← max(M)
6:     end if
7:     return B
8: end procedure
```

Uma visualização gráfica pode ser vista na Figura 2.28, onde $\lambda = 4$. Nela, o indivíduo 1, que gera suas λ mutações, está sempre no topo da gerações. Na geração 0, como nenhum indivíduo gerado a partir do 1 possui um *fitness* igual ou maior a ele, ele é passado como o 1 da seguinte geração. Na geração 1, um dos indivíduos gerados possui um *fitness* igual a seu gerador, tornando-o indivíduo 1 da geração 2. Esse processo se repete até que o objetivo desejado seja alcançado [27].

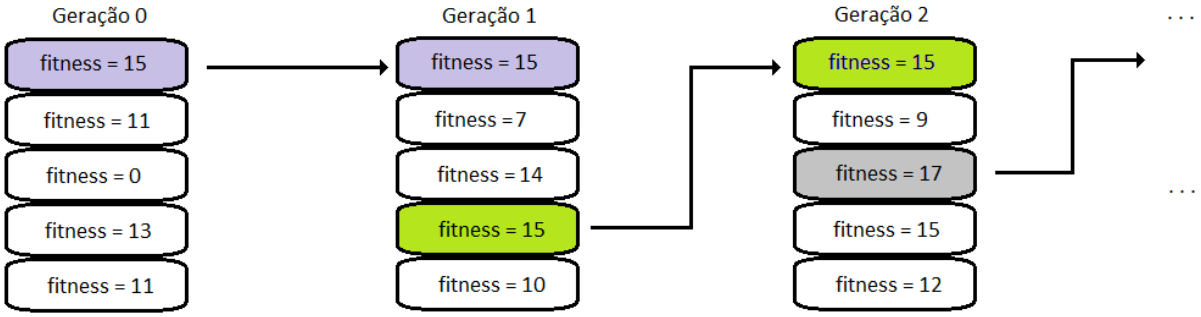


Figura 2.28: Exemplo de execução do algoritmo $(1 + \lambda)$.

A principal diferença entre esse algoritmo e os descritos anteriormente é que seu operador principal é a mutação, não havendo operações de *crossovers*. Todos os λ indivíduos são gerados a partir de mutações do individuo pai. De acordo com [26], tipicamente $\lambda = 4$.

O algoritmo $(1 + \lambda)$ é naturalmente elitista. O indivíduo com maior *fitness* sempre é preservado para gerações seguintes.

2.5 Programação Genética Cartesiana

A Programação Genética Cartesiana (do Inglês *Cartesian Genetic Programming*) (CGP), tem como característica principal a linearização de um cromossomo para representar uma rede, utilizando-se números inteiros. O termo *cartesiano* vem do fato do método consistir em uma grade bidimensional de nós [28].

O cromossomo é dividido em duas seções distintas: nós e saídas. Cada gene da seção de nós identifica dois componentes básicos de um nó:

1. A operação realizada pelo nó;
2. As entradas usadas para o cálculo.

A seção de saídas indica a que nó cada saída deve se ligar para produzir seu resultado.

A descrição que o cromossomo representa é conhecida como *genótipo*. O programa resultante da decodificação desse genótipo é chamado de fenótipo. Para CGP, enquanto o genótipo possui tamanho fixo, descrevendo completamente a grade, o fenótipo efetivo varia com o número de nós utilizados desde zero até todos os disponíveis no genótipo [29].

2.5.1 Forma Geral

Se n_i é o número de entradas necessárias para o programa e n_o o número de saídas, um sistema genético cartesiano de $N = n_r \times n_c$ nós, cada qual com r entradas, tem seu endereçamento de cromossomos da seguinte maneira:

$$F_0 C_{0,0} \cdots C_{0,r-1} F_1 C_{1,0} \cdots C_{1,r-1} \cdots F_{N-1} C_{n_c * n_r, 0} \cdots C_{n_c * n_r, r-1} O_0 \cdots O_{n_o - 1}$$

onde n_r indica o número de linhas, n_c o número de colunas, F_k é a função que o nó de número k exerce, C é o endereço de onde provém o valor de entrada e os valores O_l indicam de qual nó a saída l retira seu valor.

Um parâmetro adicional l indica o número de colunas anteriores que a coluna j pode usar para valores de entradas de seus nós, limitando sua escolha de colunas para $[max(0, j-l), \cdots, (j-1)]$. Dessa forma, não são permitidas retroalimentações, gerando uma topologia com apenas ligações *feedforward*. Uma visualização gráfica da forma geral pode ser vista na Figura 2.29.

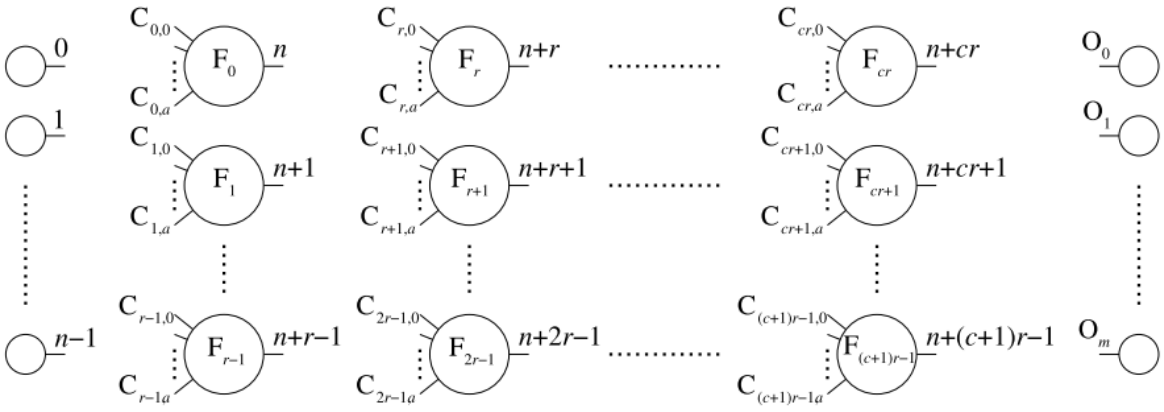


Figura 2.29: Forma geral da estrutura de um sistema genético cartesiano (Fonte: [29]).

O exemplo da Figura 2.30 mostra um sistema com $n_i = 6$, $n_o = 2$, $n_r = 2$, $n_c = 3$, $r = 2$ e $l = 2$. É possível observar a tradução que ocorre entre a descrição do genótipo para o fenótipo. No fenótipo, é instanciado um grafo de elementos lógicos, com suas ligações e funções definidas pelo genótipo. As saídas do circuito constituem os dois últimos elementos do genótipo.

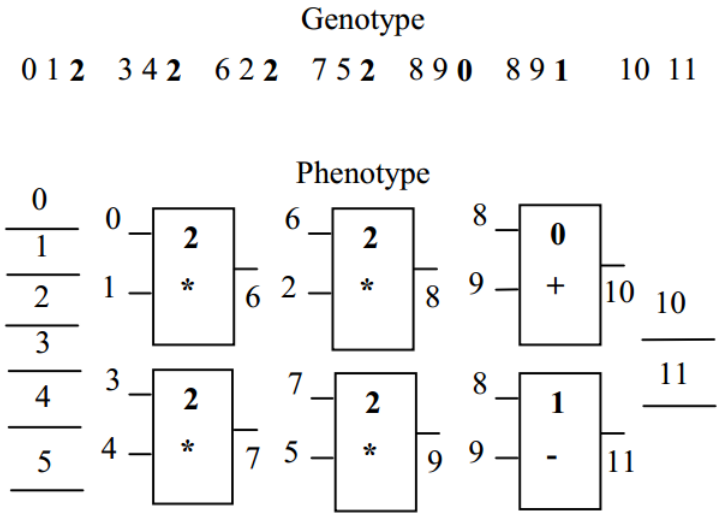


Figura 2.30: Exemplo de sistema cartesiano genético. Evidencia relação entre genótipo e fenótipo (Fonte: [28]).

2.5.2 Restrição de Valores

Para a variável F, que indica a função do nó, seja n_f o número de funções possíveis do sistema, então

$$0 \leq F < n_f.$$

Para as entradas do nó $C_{i,j}$, deve se respeitar o valor l de níveis atingíveis. Portanto,

$$n_i + (j - l)n_r \leq C_{i,j} \leq n_i + jn_r$$

para $j \geq l$ e

$$0 \leq C_{i,j} \leq n_i + jn_r$$

para $j < l$. Para os valores de saída O, segue-se a seguinte restrição:

$$0 \leq O < n_i + N.$$

Tipicamente, conforme apresentado na seção 2.5.1, sistemas genéticos cartesianos não possuem ciclos. Para que o objetivo de liberação de restrições seja melhor atendido, não será usada a variável l que indica o número de colunas anteriores atingíveis. Para este trabalho colunas poderão usar quaisquer outras como possíveis valores. Dessa forma, torna-se possível a retroalimentação, estrutura imprescindível em circuitos capazes de armazenar dados, tais como *latches* e *flip-flops*.

2.5.3 Neutralidade

CGP representa o genótipo como uma lista de tamanho fixo de números inteiros que indicam as propriedades de cada nó em um grafo. O fenótipo resultante, porém, é resultado da aplicação das ligações desses nós no genótipo. Portanto, o fenótipo possui um tamanho variável, havendo então genes inativos, ou seja, que não possuem efeito algum na avaliação de seu *fitness*. À existência de genes inativos dá-se o nome de *neutralidade*.

O fenômeno da neutralidade permite que o cromossomo se mova pelo espaço de soluções sem afetar sua avaliação, e já foi minuciosamente estudado [25][30][31]. Os estudos realizados chegaram à conclusão que a neutralidade é um efeito extremamente benéfico à eficiência do processo evolutivo.

2.6 Hardware Evolutivo

Nesta seção, as duas áreas distintas descritas nas seções 2.3 e 2.4 serão unidas em uma nova, recentemente fundada, chamada *Hardware Evolutivo*.

Na década de 90, com o surgimento de dispositivos como FPGA, pesquisadores começaram a aplicar técnicas evolutivas em conjunto com a reconfiguração disponível. Isso permitiu que também fossem utilizadas para a própria configuração dos dispositivos. Um dos primeiros trabalhos, realizado por Koza [32], envolveu a evolução extrínseca de componentes eletrônicos básicos como filtros e amplificadores. Higuchi e sua equipe [33][34] foram um dos primeiros trabalharem com a aplicação da evolução a *hardware* real, com o objetivo de usá-la na compressão de imagens, filtros analógicos, ajuste de tempos de *clock*, dentre outros.

Atualmente, a indústria de produção de circuitos eletrônicos atinge uma vasta gama de aplicações. Dessa forma, há uma crescente demanda de circuitos cada vez mais complexos. Técnicas como as apresentadas na seção 2.1.6 são úteis para o projeto de circuitos combinacionais. Por outro lado, o projeto de circuitos sequenciais, que representam uma grande parte das aplicações, pouco tiram proveito [12]. Consequentemente, inovações em técnicas de projeto são pesquisadas diariamente. Dentre elas, apresentam-se os chamados sistemas bio-inspirados [35].

Hardware Evolutivo pode ser definido como um circuito ou sistema capaz de modificar sua arquitetura e, portanto, seu comportamento dinamicamente e de forma independente por meio de interações com seu ambiente [12].

2.6.1 Aplicações

A área de *hardware evolutivo* busca resolver problemas de maneira equivalente àquela descrita na seção 2.4. Ou seja, através da inspiração em mecanismos da natureza, busca-se obter um *hardware* mais flexível.

Tolerância a Falhas

Nenhum sistema se mantém estático. Falhas, internas ou externas (interferência eletromagnética, por exemplo), são eventualmente introduzidas ao longo do tempo, podendo em algum momento, em conjunto com outros fatores, surgir como erros. Esse problema é agravado quando posto em paralelo com a necessidade de sistemas cada vez mais complexos. Falhas podem, então, surgir em diversas partes, tornando impraticável a verificação total e constante do circuito.

Dessa maneira, no projeto de um sistema, pode-se utilizar da abordagem evolutiva para embutir falhas no processo da seleção, como mudanças de temperatura para melhor se adaptar a diferentes ambientes [16].

Em um estudo [36], um oscilador foi evoluído, mostrando certa viabilidade para esse tipo de método utilizando uma simples função de *fitness* que levava em conta apenas a saída do circuito. Após um indivíduo atingir um nível aceitável, integrou-se um novo fator para a seleção. Uma falha era introduzida, um elemento lógico aleatório tinha sua saída travada para 1 ou 0. Uma queda súbita no *fitness* médio foi observada, porém ao longo das gerações o nível anterior se restabeleceu. O resultado foi um circuito robusto, com certa redundância e boa resistência a falhas.

Soluções Inovadoras

Uma das principais motivações para se utilizar algoritmos genéticos para desenvolver circuitos vem do fato de não conhecermos todo o espaço de soluções possíveis. De fato, *design* de circuitos lida com um conjunto infinito de possíveis soluções, principalmente para aqueles que podem ser classificados como sequenciais, como a seção 2.2 descreve.

Projetistas frequentemente se utilizam de abstrações para achar soluções de problemas. Por exemplo, no *design* de processadores de propósito geral, a menor abstração utilizada tipicamente se mantém no nível de portas lógicas. Portas lógicas, porém, são formadas por um conjunto de transistores, que por sua vez, possuem sua própria abstração física. Seria completamente impraticável o projeto de algo assim levando em conta os absolutos mínimos detalhes.

Por não lidar com essas abstrações impostas para a conveniência de projetistas, *hardware* evolutivo possui um espaço de soluções ainda não completamente explorado a sua disposição. A Figura 2.31 ilustra a afirmação.

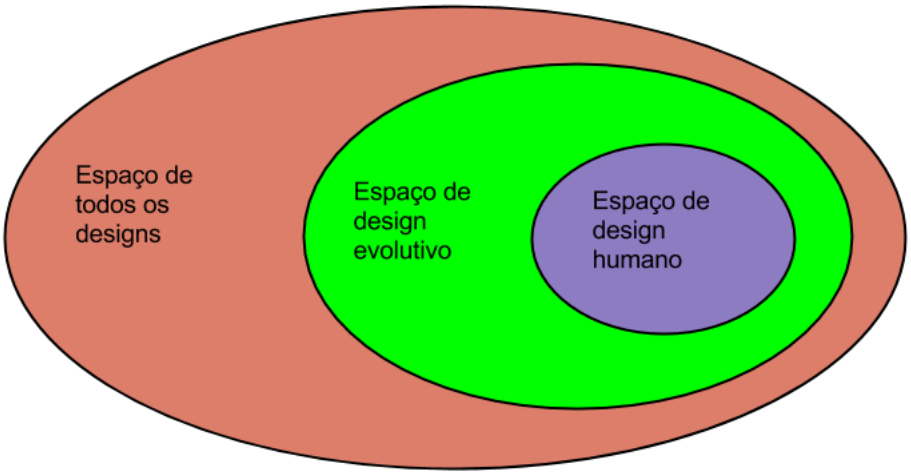


Figura 2.31: Espaço de soluções e alcances das abordagens evolutiva e humana.

Um detalhe deve ser destacado, porém. Enquanto a remoção de restrições pode ressaltar soluções antes não vistas, é possível também que não se chegue a ela em um tempo prático ou que nem se chegue nela [12]. É importante encontrar um equilíbrio entre liberdade e restrição para que uma solução eficiente seja encontrada.

2.6.2 Formas de Evolução

Para *hardware* evolutivo, existem duas abordagens principais para o projeto do circuito final.

Extrínseca

Na abordagem extrínseca, o fenótipo dos circuitos é avaliado por simulação em *software*, com o resultado final, então, sendo implementado em um circuito real. Como simular todas as características do mundo real é computacionalmente muito caro, simuladores de circuitos, tais como o *SPICE*, geralmente optam por se utilizarem de modelos de abstrações com características bem definidas para evitar cálculos mais detalhados [16].

Intrínseca

Na abordagem intrínseca, a avaliação dos indivíduos da população é feita diretamente no ambiente físico determinado, normalmente um dispositivo FPGA. Dessa forma, o circuito correspondente a cada indivíduo é sintetizado e avaliado no ambiente real onde será aplicado. Esta abordagem é também chamada de evolução *on-line* [37].

2.6.3 Limitações em *Hardware* Evolutivo

Como todas as áreas de conhecimento humano, *hardware* evolutivo também possui seus desafios e problemas principais. Eles podem ser resumidos na seguinte lista [38]:

1. Escalabilidade: A escalabilidade de *hardware* evolutivo é reconhecido como um problema importante por pesquisadores. Pela nossa revisão, não conseguimos encontrar trabalhos que utilizem técnicas evolutivas aplicadas a *hardware* em larga escala. Um dos fatores relacionados a isso é que a representação de um cromossomo de evolução sem restrições, como aquelas realizadas neste trabalho, possui uma complexidade de cerca de $O(n^2)$, com n sendo o número de células lógicas. Assim, quanto mais células lógicas são utilizadas, uma representação quadraticamente maior é necessária para representar seu cromossomo e, por consequência, o espaço de soluções cresce exponencialmente, dificultando o processo de evolução. Uma das questões mais comumente feitas é, se é normal executar experimentos envolvendo apenas dezenas de portas lógicas por dias, o quão viável isso é para milhares ou dezenas de milhares de células lógicas?

2. *Fitness* e verificação / testes: Um problema essencial que surge no projeto evolucionário de circuitos digitais é se certificar de que ele é correto. Se uma função de *fitness* é definida tal que ela é máxima somente quando o circuito é totalmente correto, então isso não é um problema. Pela natureza exponencial relacionada à quantidade de entradas de um circuito e suas possíveis combinações, rapidamente isso se torna um problema porém. Evoluções sem restrições apresentam ainda mais problemas pelo fato do circuito evoluído se utilizar de todas as características físicas de seu ambiente de avaliação, dificultando sua aplicação funcional para outros.

3. Terminação da evolução: Como uma consequência da dificuldade de definição para um algoritmo genético, saber quando parar a evolução e determinar um indivíduo como correto também se torna um problema. Um detalhe que não se mostra claro em pesquisas de *hardware* evolutivo é se seus resultados foram encontrados em apenas uma execução do algoritmo genético ou se várias foram necessárias. Qual a média

de execuções necessária para se encontrar uma solução correta? Quando se deve parar?

4. Manutenção: Circuitos evoluídos frequentemente apresentam organizações extremamente complexas, portanto, sendo de difícil entendimento. Dessa maneira, são tratados como caixas-pretas sem a possibilidade de alterações internas, o que não é a situação ideal para projetos reais em larga escala.

Outros estudos mais recentes [39][40] também confirmam que o maior desafio que persiste para a área de *Hardware Evolutivo* é a escalabilidade e que não houve muito progresso para solucioná-lo nos últimos 20 anos.

Uma outra grande limitação da área vem da falta de trabalhos teóricos em relação às técnicas evolutivas utilizadas, que poderiam auxiliar o desenvolvimento de algo que seja mais facilmente escalável [39]. Um estudo [41] analisou a complexidade Kolmogorov de circuitos tolerantes a falhas pela aplicação da complexidade Lempel-Ziv deles. Essa análise envolve a compressibilidade dos *bitstrings* que representam esses circuitos e a comparação deles com o espectro de complexidade relacionado (*bitstrings* simples a complexos). Seu resultado sugere que esses circuitos ocupam apenas uma pequena parte do espaço total e que é possível ter benefícios ao enviesar buscas para essa parte do espaço de soluções.

2.7 Estado da Arte

Essa seção discutirá e revisará a literatura atual sobre a evolução de circuitos digitais, tanto combinacionais quanto sequenciais.

2.7.1 Evolução de circuitos digitais combinacionais

As seções anteriores do texto já fizeram referências a diversos estudos envolvendo a evolução de circuitos combinacionais. Alguns serão revistos aqui.

O estudo de Miller [28] foi um dos primeiros a demonstrar a eficácia da técnica Programação Genética Cartesiana (do Inglês *Cartesian Genetic Programming*) (CGP), mostrando sua viabilidade para a evolução de circuitos combinacionais. Posteriormente, vários estudos propuseram variações dessa técnica para resolver diferentes problemas.

Dentre eles, Walker [25] propõe utilizar uma variação, denominada por ele de *Multi-Chromosome Cartesian Genetic Programming* (MC-CGP), para que soluções conseguissem ser obtidas de maneira eficiente. Outros estudos [42] comprovaram o ganho de eficiência relatado.

No campo de *design* e otimização, o estudo de Hilder *et al.* [24] propõe o uso do algoritmo *Non-dominated Sorting Genetic Algorithm II* (NSGA-II) [43] para circuitos

que utilizam a organização cartesiana. Os autores chegaram a soluções extremamente otimizadas de circuitos somadores, multiplicadores e decodificadores de 7 segmentos.

Ainda nesse mesmo campo, outro estudo [44] propõe uma técnica chamada *Hybrid Multi-Chromosome Cartesian Genetic Programming* (HMC-CGP) para obter rapidamente soluções para os circuitos, adaptando a utilização da técnica MC-CGP para este fim, seguida de uma fase de otimização dessas soluções, permitindo a interação lógica das diferentes soluções. A técnica proposta foi aplicada aos circuitos combinacionais somadores (*full-adder*) de 1 e 2 *bits*, multiplicador de 2 *bits* e decodificador de 7 segmentos.

2.7.2 Evolução de circuitos digitais sequenciais

Em comparação aos estudos de circuitos combinacionais, há relativamente pouca exploração no espaço de circuitos sequenciais. Dentre os existentes, e um dos primeiros a serem realizados, está o estudo de Thompson [16]. Nele, observa-se a evolução de um circuito capaz de discriminar entre frequências de 1kHz e 10kHz de entrada. O processo foi feito em um ambiente de FPGA, sem o auxílio de entradas externas (como *clocks*) além da necessária para a realização do experimento. O resultado relatado é um circuito que se aproveita de propriedades que não são levadas em consideração no projeto convencional de circuitos (conjecturadas como a interferência eletromagnética das células do FPGA).

Dois estudos [45][46] utilizam uma abordagem que se aproveita do fato de circuitos sequenciais conceitualmente se apresentarem como duas funções matemáticas: $s : S \times I \rightarrow S$ e $o : S \times I \rightarrow O$. s é a função de transição de estados, o, a de saída do circuito, e os conjuntos S, I e O representam os possíveis valores do estado, da entrada e da saída respectivamente. A Figura 2.32 apresenta essa organização, com o bloco *Next state logic* representando a função s e o bloco *Output logic*, a função o. Dessa maneira, os autores possibilitaram o uso da evolução tratando ambas as funções como circuitos combinacionais separados, sem precisar manipular o armazenamento inerente a circuitos sequenciais. Assim, é possível a aplicação das técnicas já discutidas para circuitos combinacionais, reduzindo o problema da evolução sequencial para combinacional. Essa abordagem permitiu a evolução de dois circuitos detectores de sequência. Uma consequência importante dela, porém, é que essa redução necessariamente representa uma restrição no espaço de soluções, potencialmente excluindo circuitos que possuam características interessantes.

Neste capítulo, foram apresentados todos os conceitos básicos necessários para o entendimento restante do trabalho, assim como uma discussão sobre as pesquisas mais recentes quanto ao estado-da-arte de evolução para circuitos sequenciais. Com isso, podemos então introduzir a proposta principal deste trabalho.

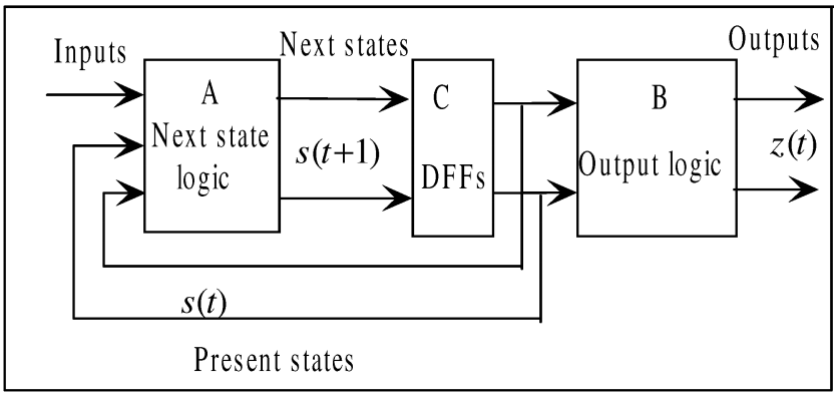


Figura 2.32: Componentes de um circuito sequencial (Fonte: [46]).

Capítulo 3

Metodologia Proposta

Este trabalho propõe uma nova metodologia para a exploração do espaço de projeto de circuitos digitais sequenciais por meio de algoritmos genéticos. A característica principal em comparação ao estado da arte atual se baseia em realizar uma evolução intrínseca em que funcionalidade e armazenamento são ambos levados em consideração, permitindo uma busca maior e mais fina dentro do espaço de soluções.

Este capítulo está organizado nas seguintes seções: a primeira seção introduz mais detalhadamente o problema a ser resolvido, a segunda seção descreve brevemente a evolução intrínseca, suas dificuldades e porque é necessária, a seguinte descreve sobre o projeto da função de avaliação, suas características, problemas possíveis e soluções, e a última seção apresenta a implementação da metodologia proposta.

3.1 Definição do Problema

A aplicação de algoritmos genéticos à evolução de circuitos digitais combinacionais é uma área que já recebeu bastante atenção. O estudo dessa aplicação para o projeto de circuitos sequenciais, porém, ainda é um pouco esparso. A maior parte dos estudos reduz o problema de encontrar circuitos sequenciais para o de encontrar os circuitos combinacionais que formam as funções de transição e saída que compõem um circuito sequencial.

Neste trabalho buscamos projetar circuitos sequenciais pela definição de uma sequência de entradas e saídas que descreva seu comportamento desejado, como ilustra a Figura 3.1 para a *latch SR*. Dessa maneira, o circuito produzido é aquele que satisfaz as sequências, sendo completamente livre de outras restrições que possam afetar sua estrutura, como a redução comentada na Seção 2.7.2 impõe.

Em comparação aos métodos clássicos de projeto, a principal vantagem dessa abordagem é a possibilidade de encontrar soluções alternativas e inovadoras. Como ocorre na maior parte das aplicações envolvendo algoritmos genéticos, no entanto, não há garantia

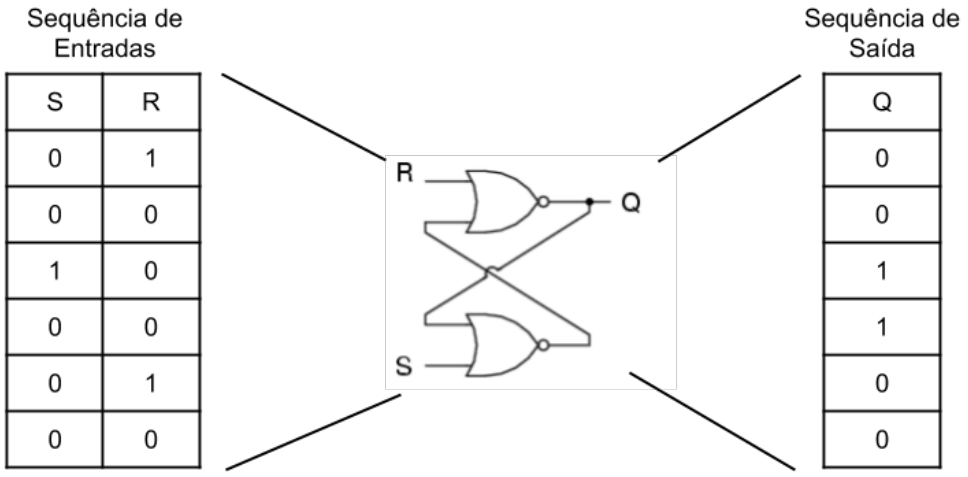


Figura 3.1: Sequência de entradas e saídas da *latch SR*.

de que as soluções serão mais eficientes, ou mesmo que soluções serão encontradas em um tempo viável de execução da busca.

Por conta da abordagem escolhida, não há maneira fácil de realizá-la por uma evolução extrínseca, pois circuitos sequenciais possuem características chamadas de *loops combinacionais*, de difícil simulação por depender muito da tecnologia utilizada para a construção desses circuitos. Dessa maneira, torna-se necessária a evolução intrínseca.

3.2 Evolução Intrínseca

Para que haja uma evolução intrínseca, é necessário que o processo ocorra dentro do ambiente de execução do circuito. As principais dificuldades desse tipo de abordagem podem ser resumidas em uma evolução mais complexa e possivelmente menos eficiente, pela necessidade de comunicação entre partes distintas (ambiente de execução e ambiente de avaliação). Avaliações paralelas intrínsecas também podem ser mais difíceis ou até impossíveis dependendo do ambiente necessário.

O algoritmo genético utilizado é o $(1 + \lambda)$ (descrito na seção 2.4.2), com indivíduos compostos de um arranjo similar àquele visto na técnica CGP [28]. Cada célula desse arranjo assume uma das seguintes funções lógicas: *AND*, *OR*, *NOT*, *XOR*, *NAND*, *NOR* e *XNOR*. A principal diferença da abordagem proposta é a remoção da limitação de que células devem apenas usar colunas anteriores como entradas. Por consequência, as células estão livres para se conectar a qualquer outra, incluindo a si mesmas. Essa abordagem está em contraste direto com àquelas detalhadas nos estudos de Soleimani *et al.* [45] e Belgasem [46] por explicitamente incluir o mecanismo de armazenamento no processo evolutivo.

3.3 Avaliação dos Indivíduos

Sendo a parte mais sensível do processo evolutivo, a função de *fitness* necessita de um detalhamento minucioso. Neste projeto, deve-se garantir o comportamento correto dos indivíduos, ou seja, que os circuitos por eles representados respondam adequadamente a uma dada sequência de entrada.

De maneira geral, a avaliação de um indivíduo, descrito por C, se dá por uma dupla de sequências: uma de entradas $(I_1, I_2, \cdots, I_N)$, onde cada I tem o formato $(i_1, i_2, \cdots, i_T)$, $i_j \in \{0, 1\}$ e outra de saídas esperadas $(S_1, S_2, \cdots, S_N)$ onde cada S tem o formato $(s_1, s_2, \cdots, s_U)$, $s_k \in \{0, 1\}$, onde T é o número de *bits* da entrada do indivíduo e U o número de *bits* da saída. Seja a tupla $a = (a_1, a_2, \cdots, a_U)$ uma amostra da saída, obtida pela aplicação de $sim(C, I)$, que é a função responsável por aplicar a entrada I no indivíduo C. Para a L-ésima entrada I_L, então, definimos sua sequência de amostras (a uma frequência f_a) $A_L = (a_1, a_2, \cdots, a_M)$ de tamanho M. Aplicado a todos os elementos da sequência de entradas, isso nos dá a sequência de todas as amostras $A = (A_1, A_2, \cdots, A_N)$. Essa amostragem é um componente necessário pois circuitos que possuem *loops combinacionais* têm a possibilidade de apresentar um comportamento oscilatório em suas saídas. Assim, fazendo a amostragem, minimiza-se a possibilidade de oscilações serem detectadas como comportamento correto.

A função de análise dessas amostras, que consiste em calcular os erros acumulados das amostras em relação às saídas esperadas, está descrita na Equação 3.1

$$samp(a, C, i, s) = \begin{cases} 0, & \text{se } a \geq M \\ samp(a+1, C, i, s), & \text{se } a < D \\ 1, & \text{se } (\sum_{u=1}^{U} s_u \oplus sim(C, i)_u) > 0 \\ samp(a+1, C, i, s), & \text{senão.} \end{cases} \tag{3.1}$$

que determina que a amostragem está correta somente se todos os valores corresponderem ao valor esperado na saída. Caso contrário, a amostragem é determinada como incorreta e a ela é atribuída o valor 1. Essa função é, então, utilizada no cálculo da análise de todas as amostras e determina o número de erros de um cromossomo como a soma de todas essas análises, como mostra a Equação 3.2

$$seq(C, I, S) = \sum_{n=1}^{N} samp(0, C, I_n, S_n). \tag{3.2}$$

Assim, um indivíduo C é considerado correto se $seq(C, I, S)$ resultar em 0, ou seja, todas as amostras corresponderem aos valores esperados na saída do circuito avaliado.

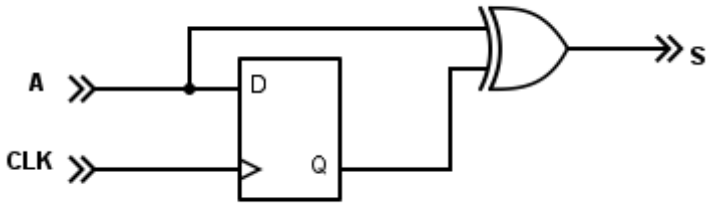


Figura 3.2: Circuito sequencial de *XOR* entre a entrada atual e anterior.

Para acomodar tempos de propagação de sinais e seus transientes, as primeiras $D < M$ amostras são descartadas, ou seja, não são contabilizadas na análise de amostras feita pela função da Equação 3.1.

Um outro possível problema é a oscilação em transições, ou seja, é possível que a saída do indivíduo dependa de alguma oscilação interna, tornando-a estável, mas incorreta apenas ao trocar sua entrada. Se, por chance, a análise identificar esses valores oscilatórios como aqueles iguais aos da sequência de saída, esse indivíduo, apresentando esse comportamento incorreto, será considerado como correto. Assim, para tratar esse problema, a função de análise de amostras é chamada, no máximo, N_R vezes até que a amostragem indique um comportamento incorreto ou o número de tentativas seja esgotado, quando o indivíduo é, então, considerado correto, conforme descrito na função da Equação 3.3.

$$retry(r, C, I, S) = \begin{cases} 0, & \text{se } r \geq N_R \\ seq(C, I, S), & \text{se } seq(C, I, S) > 0 \\ retry(r+1, C, I, S), & \text{senão.} \end{cases} \tag{3.3}$$

A função de *fitness* total, descrita pela Equação 3.4, é a chamada da função presente na Equação 3.3 com o valor inicial de repetição 0.

$$fit(C, I, S) = retry(0, C, I, S) \tag{3.4}$$

Como exemplo, para o circuito visto na Figura 3.2, os parâmetros de entrada e saída do circuito são, respectivamente, $T = 2$, $U = 1$. Assim, para codificar seu comportamento, que realiza a operação *XOR* entre a entrada atual e a entrada anterior, definimos suas sequências de entrada e saída na Tabela 3.1, de tamanho $N = 22$.

Com $M = 3$ e $D = 0$, podemos obter de uma amostragem hipotética a seguinte sequência de amostras $A = ((0,0,0), (0,0,0), (1,1,1), (0,0,0), (1,1,0), (1,1,1), (0,0,0), (1,1,1), (0,0,0), (0,0,0), (0,0,0), (1,1,1), (0,0,1), (1,1,1), (0,0,0), (0,0,0), (1,1,1), (0,0,0), (0,0,0), (1,1,1), (1,1,1), (0,0,0))$. Dessa maneira, somando-se os erros em

Tabela 3.1: Tabela com sequências de entrada e saída descrevendo o comportamento do circuito da Figura 3.2

A Clk	Q
00	X
01	0
11	1
01	0
11	1
10	1
00	0
10	1
00	0
01	0
00	0
10	1
11	0
01	1
11	0
10	0
00	1
10	0
11	0
01	1
00	1
01	0

comparação à sequência de saídas esperadas, como mostra a equação 3.1, obtemos o valor de *fitness* total de 2.

É importante ressaltar que uma desvantagem dessa forma de codificação é o possível viés que o circuito resultante pode ter para a sequência, respondendo corretamente para apenas ela. Dessa maneira, o circuito se comporta corretamente, mas apenas para os casos avaliados. Em outras palavras, é possível não conseguir cobrir todo o espaço de combinações de entradas e respostas do circuito.

3.4 Mutação do Indivíduo

No contexto do algoritmo de seleção $(1 + \lambda)$, a mutação é uma operação vetorial que busca, a partir de um ponto no espaço de soluções, gerar outro aleatoriamente que seja potencialmente mais próximo à resposta desejada. Assim, é importante detalhar como esse processo é realizado em nossa proposta.

O cromossomo do indivíduo é composto de duas partes: a configuração de suas células lógicas internas e a configuração de suas saídas. Seja $F = \{AND, OR, NOT, XOR, NAND, NOR, XNOR\}$ o conjunto de funções lógicas possíveis, então cada célula lógica pode ser descrita pela tripla (in_0, in_1, f), onde $in_i \in [0, n_e+T)$ é um valor inteiro representando uma referência a uma entrada do circuito ou saída de uma célula lógica, e $f \in F$, sua função lógica associada, totalizando então 3 elementos que podem sofrer mutação. Cada saída $s_i \in [0, n_e + T)$ pode também sofrer mutação e ter seu valor alterado.

Uma operação de mutação para nosso indivíduo consiste em escolher aleatoriamente um de seus elementos mutáveis e alterar seu valor para outro valor semanticamente correto. Seja $rand()$ a função que gera um valor aleatório no intervalo $[0, 1)$, então definimos a geração do novo valor v', utilizado para se referenciar a uma entrada ou saída de célula lógica, como mostra a Equação 3.5, onde n_e é o número total de células lógicas do indivíduo. Esta operação é utilizada tanto para alterar o valor de uma entrada de uma célula quanto o da saída do indivíduo.

$$v'(C) = \lfloor (n_e(C) + T(C) - 1) \times rand() \rceil \tag{3.5}$$

A geração de uma função aleatória f' para uma célula pode ser vista na Equação 3.6.

$$f' = F_{\lfloor 7 \times rand() \rceil} \tag{3.6}$$

Podemos calcular o número de elementos expostos a mutação n_{el} como mostra a Equação 3.7.

$$n_{el}(C) = n_e(C) \times 3 + U(C) \tag{3.7}$$

Assim, a mutação de um indivíduo é descrita no Algoritmo 3. Sua execução consiste em primeiro detectar se a mutação será feita em uma célula lógica ou em um valor de saída. Caso seja uma saída, altera-se seu valor como mostra a Equação 3.5. Senão, para a mutação de uma célula lógica, calcula-se qual deve ser alterada e que elemento dela. Caso seja uma entrada, ela é alterada da mesma maneira como mostra a Equação 3.5. Caso seja uma função, a alteração segue a Equação 3.6.

Algoritmo 3 Aplicação da operação de mutação em um cromossomo C.

```
1:  procedure MUTACAO(C)
2:      r ← ⌊rand() × (n_el(C) − 1)⌉
3:      if r ≥ (3 × n_e(C) − 1) then                ▷ Mutação de valor de saída
4:          r ← r − (3 × n_e(C) − 1)
5:          C.s_r ← v'(C)
6:      else                                        ▷ Mutação de célula lógica
7:          ind ← r/3
8:          elem ← r mod 3
9:          if elem = 2 then
10:             C.cel_ind.f ← f'
11:         else
12:             C.cel_ind.in_elem ← v'(C)
13:         end if
14:     end if
15:     return C
16: end procedure
```

Utilizando-se de uma porcentagem de mutação $p_m \in [0, 1]$, calculamos o número de operações de mutação a serem feitas em um indivíduo como mostra a Equação 3.8. Dessa maneira, à medida que o número de elementos lógicos de um indivíduo aumenta, mais operações de mutação são realizadas para gerar um filho.

$$n_m(C) = \lceil n_{el}(C) \times p_m \rceil \tag{3.8}$$

O algoritmo que calcula, então, a mutação usada para gerar novos filhos para a seleção $(1 + \lambda)$ pode ser visto no Algoritmo 4, envolvendo a aplicação repetida da operação de mutação de acordo com o número de operações que deve ser realizado calculado pela Equação 3.8.

Algoritmo 4 Aplicação da operação de mutação para o cromossomo C.

```
1: procedure MutacaoPorcentagem(C)
2:     C' ← C
3:     for i ← 1, n_m(C) do
4:         C' ← Mutacao(C')
5:     end for
6:     return C'
7: end procedure
```

3.5 Implementação

O ambiente no qual os experimentos foram realizados está no dispositivo *DE1-SoC*, cuja organização está mostrada na Figura 3.3.

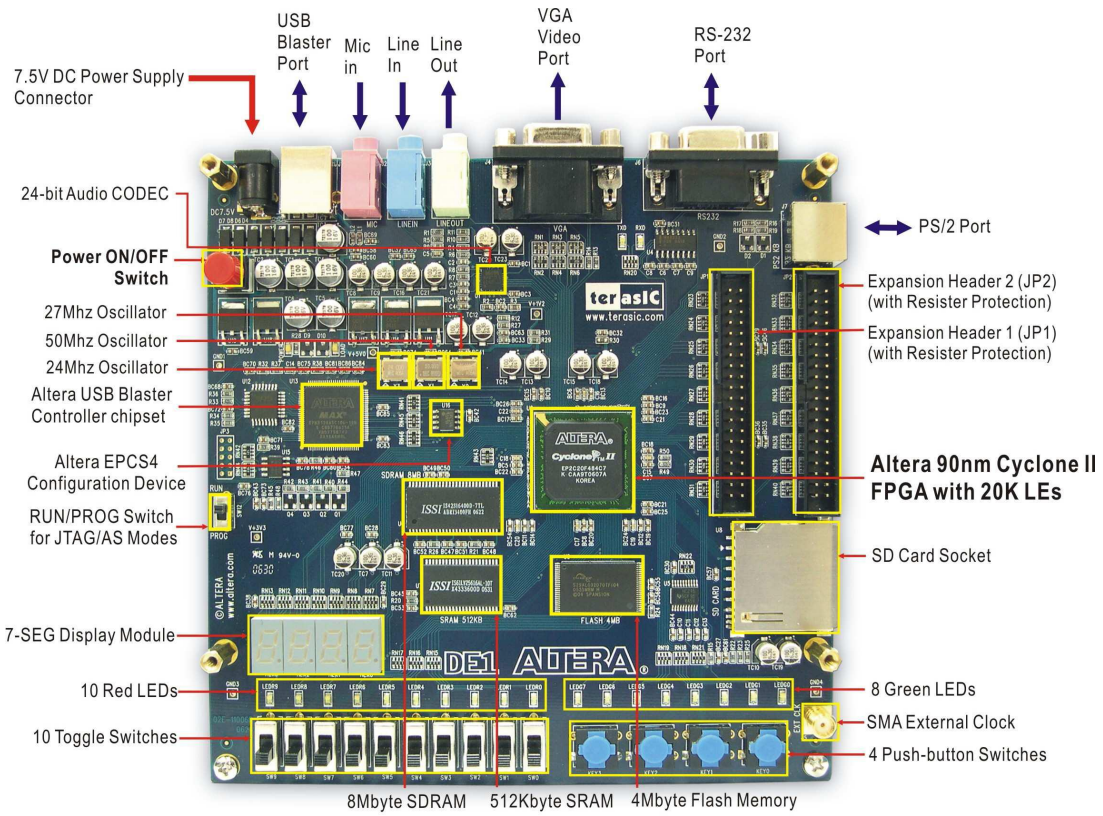


Figura 3.3: Placa de desenvolvimento *DE1-SoC* [1].

Por possuir um processador *ARM Cortex A9* de 800 MHz, é possível carregar um sistema operacional *Yocto Linux* e utilizá-lo como um computador remoto (por *Secure Shell* (SSH)). Além da execução do sistema operacional, o dispositivo permite a configuração e operação de circuitos digitais em seu FPGA *Cyclone V*, possibilitando também

a comunicação entre ambos. Dessa maneira, é possível realizar o algoritmo por completo em apenas um dispositivo, dispensando a utilização de comunicação externa (como *RS-232*). Isso traz vantagens como eficiência e simplicidade da comunicação, apontados como características importantes para a viabilidade desse tipo de aplicação [44].

Como, no kit *DE1-SoC*, é disponibilizado ao FPGA *Cyclone V* um *clock* máximo de 50 MHz (sem necessidade de uso de PLLs), as amostragens são realizadas a essa frequência ($f_a = 50 \times 10^6$Hz).

O algoritmo total está organizado em duas partes: no algoritmo genético sendo executado pela CPU *ARM* e no circuito responsável por realizar o cálculo do *fitness* de um indivíduo. O algoritmo genético é responsável pela execução e manuseio da lógica de gerações, mutações e envio de cromossomos para serem avaliados pelo circuito. O circuito, então, recebe descrições codificadas dos indivíduos e realiza o procedimento de cálculo do número de erros, dadas as sequências de entradas e saídas desejadas. Essa arquitetura está apresentada, em alto nível, na Figura 3.4.

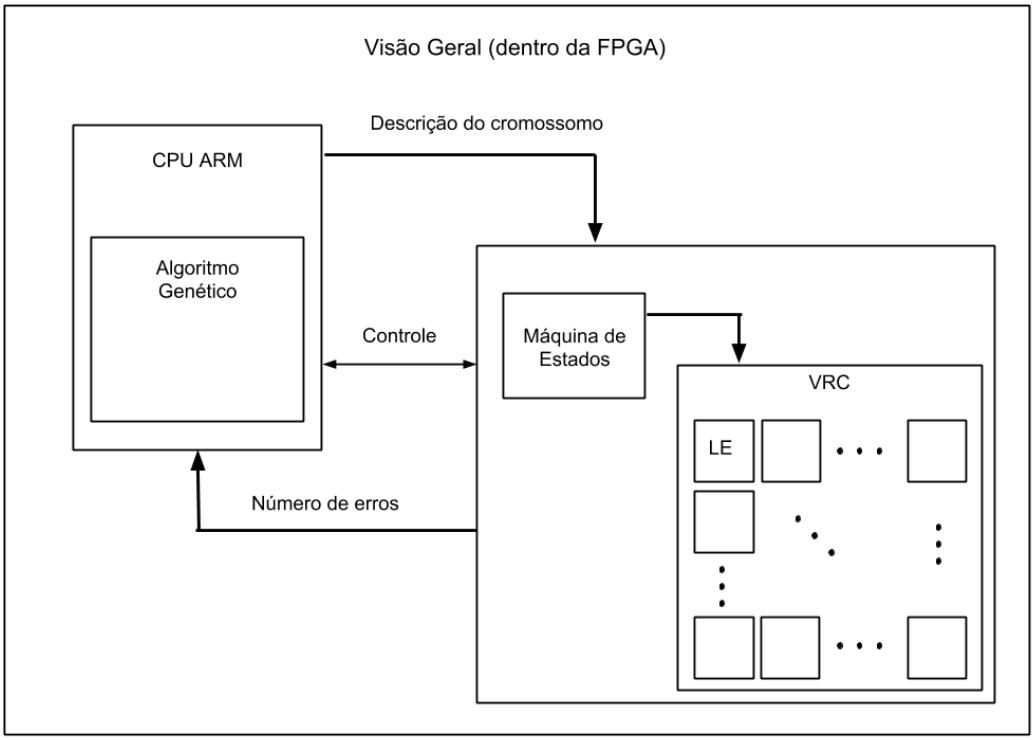


Figura 3.4: Visão geral da organização do algoritmo pela perspectiva do FPGA.

A definição de um sistema CGP envolve uma série de condições para que seja determinado como tal. Especificamente, ao permitirmos que células se conectem a quaisquer outras, isso foge da condição de que, em um arranjo bidimensional, células só utilizam

como entradas aquelas que estiverem em colunas anteriores a elas. Na técnica utilizada neste trabalho, não há a necessidade de organizar as células lógicas nesse arranjo bidimensional, visto que não há restrições relacionadas a essa organização.

Por simplicidade, escolhemos o número de entradas das células lógicas como 2, facilitando o projeto dos circuitos e seu entendimento. Não há restrições quanto ao número de entradas das células lógicas, mas o tamanho do cromossomo é, também, proporcional a esse número de entradas, aumentando consideravelmente para um número alto. Assim, tendo as células lógicas com 2 entradas permite também que o tamanho do cromossomo relacionado a esse parâmetro seja o menor possível.

Para minimizar o número de células lógicas presentes nas soluções, o algoritmo é executado incrementalmente. Isso significa que, se uma resposta correta não for encontrada em até um número máximo de gerações, o número de células disponível para evolução é incrementado em 1 e o algoritmo é reiniciado.

A implementação do conjunto de células e suas interconexões se dá pelo circuito básico visto na Figura 3.5, que utiliza multiplexadores envolvendo todos os outros *inputs* de todas as outras células para a seleção de suas entradas e um multiplexador de sua função para determinar sua saída, podendo executar uma das seguintes funcionalidades: *AND*, *OR*, *XOR*, *NOT*, *NAND*, *NOR*, *XNOR* e *BUFFER*. Os valores que configuram a seleção de cada multiplexador são considerados como a configuração da célula.

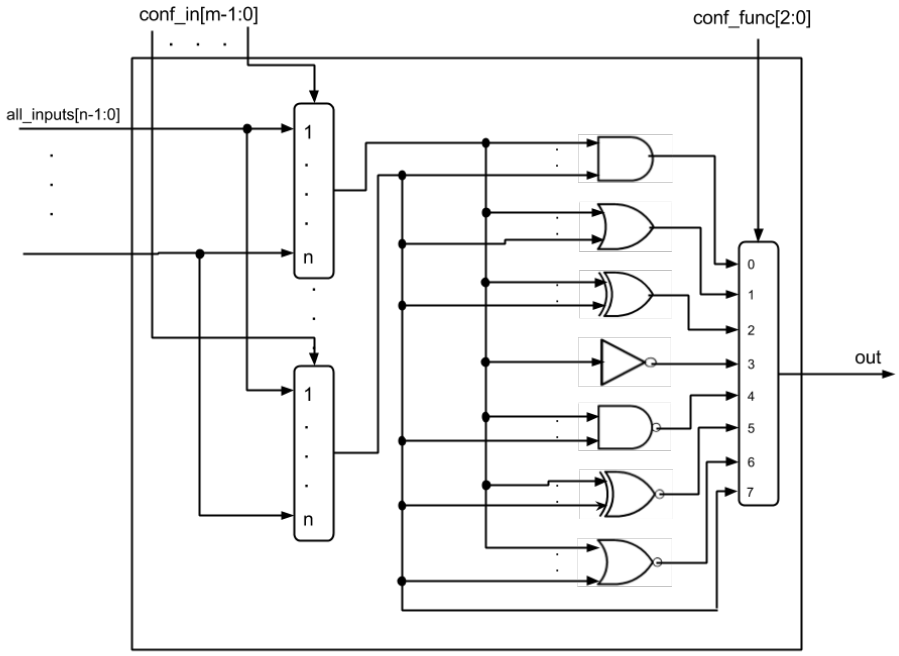


Figura 3.5: Célula Lógica básica.

O circuito reconfigurável final é descrito por várias dessas células totalmente interconectadas, juntamente com multiplexadores que selecionam as saídas descritas. A descrição total de um cromossomo, portanto, é dada pela configuração de todas as células lógicas básicas em conjunto com a configuração desses multiplexadores de saída. Essa organização pode ser vista na Figura 3.6. A esse tipo de circuito se dá o nome de Circuito Reconfigurável Virtual (do Inglês *Virtual Reconfigurable Circuit*) (VRC) [47].

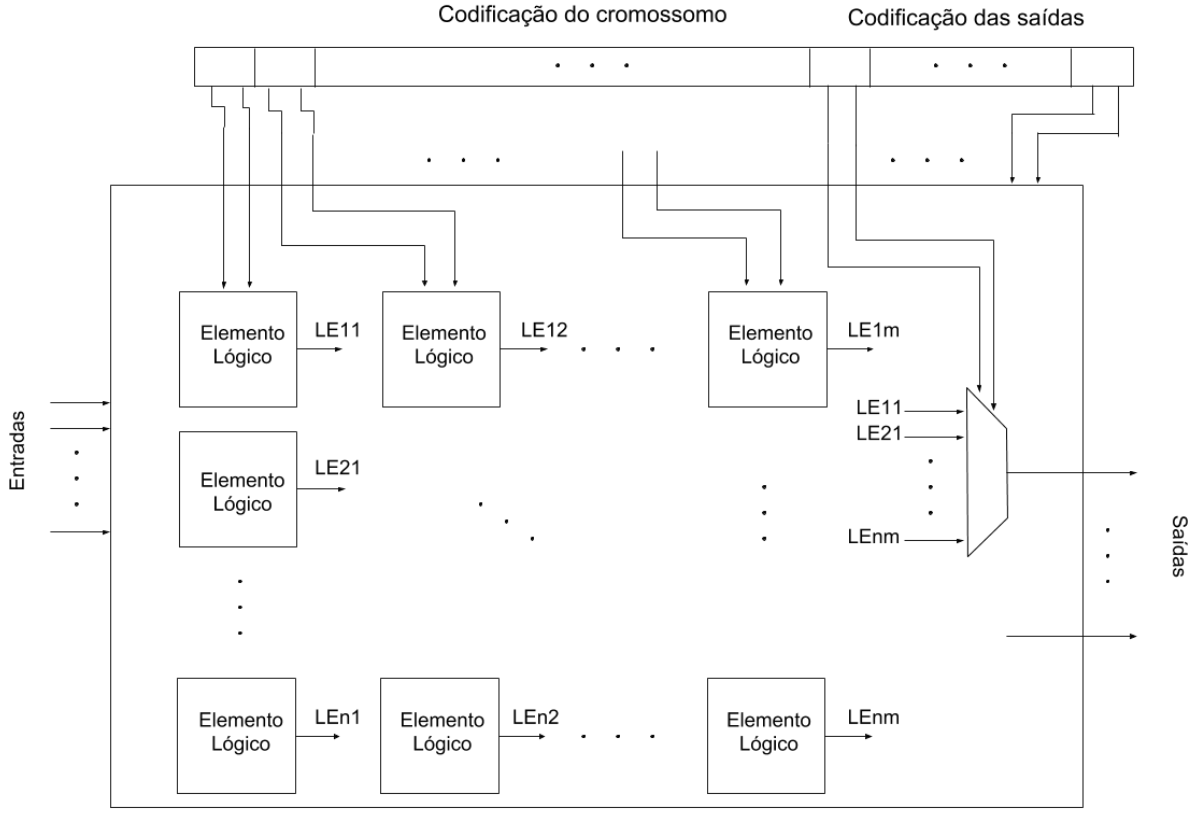


Figura 3.6: Organização do circuito que implementa a execução de um indivíduo e sua relação com a codificação deste.

A passagem de descrições de indivíduos da CPU para o circuito se dá por meio de portas I/O paralelas, que controlam uma máquina de estados responsável por iniciar e avisar sobre o término do processo de cálculo do número de erros. Essa máquina de estados está descrita na Figura 3.7.

O programa em execução na CPU é responsável pelos sinais de *StartProcessing*, usado para sinalizar para o circuito o início do cálculo junto com o indivíduo codificado, e *DoneProcessingFeedback*, que indica para o circuito que o programa registrou seu valor retornado. O circuito, por sua vez, possui os seguintes sinais ligados ao programa: *Ready*, indicando se o circuito não está processando nenhum indivíduo, *DoneProcessing*, sinali-

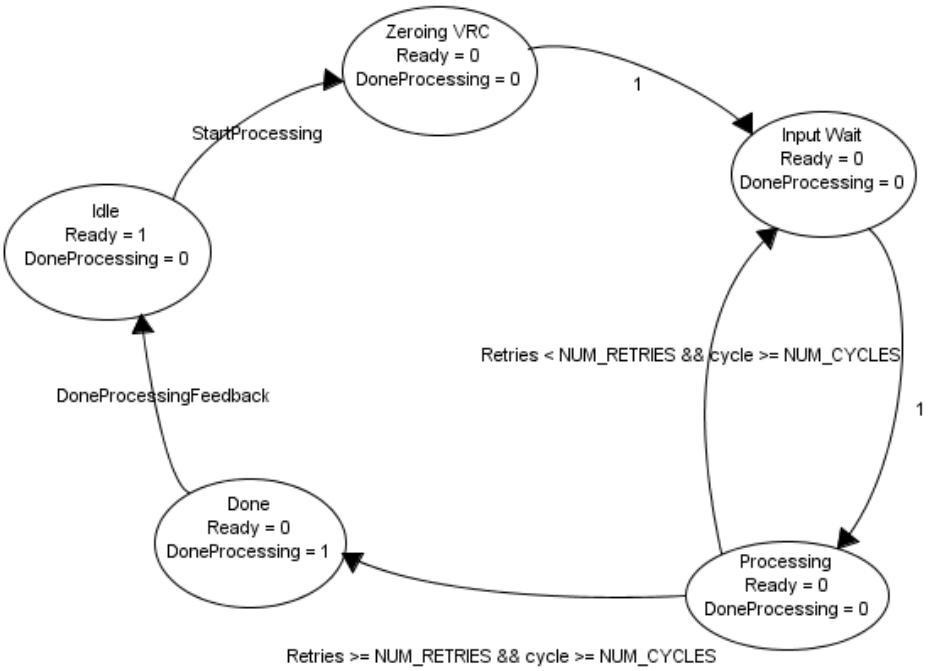


Figura 3.7: Máquina de estados que gerencia a comunicação entre programa e circuito.

zando o fim do processamento de um indivíduo e disponibilizando o resultado final para o programa. Seus estados possuem as seguintes funcionalidades:

- *Idle*: sinaliza que está pronto para processar para o programa.
- *Zeroing VRC*: estado em que se passa apenas um ciclo zerando a descrição inserida no circuito de avaliação.
- *Input Wait*: para garantir que não haverá oscilações transitórias, que possam afetar o comportamento do indivíduo sendo avaliado, a cada mudança de elemento da sequência de entradas, um ciclo é reservado para guardar o próximo valor em um registrador. Isso permite uma mudança de valores sem esse problema.
- *Processing*: estado em que se está processando um indivíduo. O estado termina sua função quando passar por toda a sequência de entradas e saídas. Se o número de *retries* ainda não tiver alcançado o limite, volta para o estado de *Input Wait*. Senão, vai para *Done*.
- *Done*: estado que sinaliza o término de um processamento. Aguarda o sinal de *feedback* antes de voltar para o estado de *Idle*.

Neste capítulo, a metodologia principal é introduzida e detalhada. O problema a ser resolvido é explicado e as abordagens propostas apresentadas.

Utilizando-se de uma evolução sem restrições intrínseca, em um ambiente FPGA, buscamos evoluir circuitos sequenciais. Para isso, o próximo capítulo detalhará os experimentos realizados e características interessantes encontradas.

Capítulo 4

Resultados Obtidos

Este capítulo apresentará os resultados obtidos da aplicação da metodologia proposta em diversos circuitos sequenciais, assíncronos e síncronos. Nos resultados está incluso também análises em diferentes ambientes de avaliação, incluindo a simulação temporal. Além disso, há uma discussão sobre o desempenho obtido para as evoluções e se é possível estimar tempos necessários de execução.

4.1 Experimentos

Os experimentos foram realizados na placa de desenvolvimento *DE1-SoC*, contendo o FPGA *Cyclone V*.

Para todos os experimentos, os parâmetros descritos na metodologia proposta possuem os seguintes valores:

- $\lambda = 4$
- $M = 1000$
- $D = 5$
- $N_R = 30000$
- $p_m = 0{,}15$

Para o algoritmo genético $(1 + \lambda)$, o número de filhos gerados é $\lambda = 4$, com uma taxa de mutação p_m de 15%. Na avaliação do indivíduo, a amostragem utiliza $M = 1000$ amostras para a análise de cada elemento da sequência, descartando as $D = 5$ primeiras. O *clock* disponível no FPGA utilizada permite que isso seja feito a uma frequência de $f_a =$ 50MHz. Com isso, o tempo de estabilização dado para os circuitos evoluídos é 100ns. A

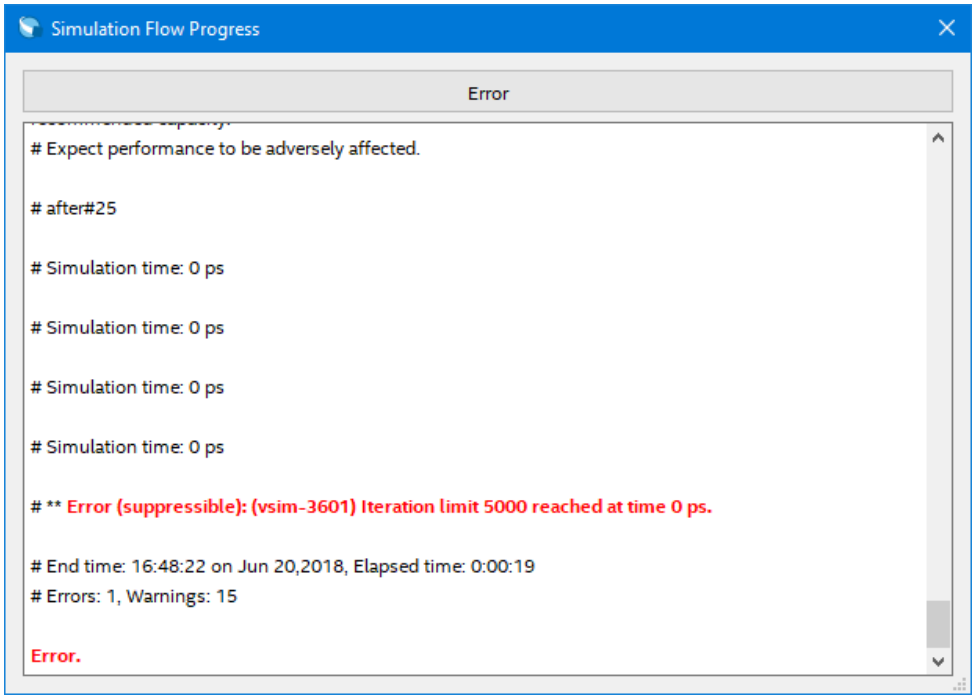


Figura 4.1: Erro encontrado durante simulação funcional pelo simulador *Modelsim*.

avaliação da sequência é repetida $N_R = 30000$ vezes antes de determinar que o indivíduo avaliado está correto.

Como um algoritmo genético intrínseco encontra soluções específicas para o ambiente em que ele foi evoluído [16], para avaliar a generalização dos circuitos encontrados quando implementados em outros ambientes, estes circuitos são testados também nas placas de desenvolvimento *Altera DE2-70* e *Altera DE2-115*, contendo os FPGAs das famílias *Cyclone II* e *Cyclone IV* respectivamente. Além disso, os circuitos encontrados também são simulados temporalmente utilizando o simulador *Modelsim*, tendo como base os dois FPGAs previamente citados. Simulações temporais não são suportadas pelo *software Quartus Prime v.18* para o FPGA *Cyclone V*, e as simulações funcionais são incapazes de simular os circuitos evoluídos, como mostra a Figura 4.1, que mostra um erro de simulação em que foi atingido o número de iterações do simulador *Modelsim*.

Um dos problemas de se descrever o circuito de avaliação na linguagem *SystemVerilog*, uma linguagem de alto nível, é a compilação para o *bitstream* utilizado no FPGA. Como ele apresenta naturalmente *loops combinacionais* necessários para a descrição do circuito VRC, e pelo compilador presente no *software Quartus Prime* ser altamente otimizador, foi observado que a semântica desses *loops* não se preserva nessa compilação, supostamente ao tentar adaptar essa lógica aos seus elementos lógicos internos. O *Quartus* provê, porém, uma ferramenta chamada *LCELL*, que é usada para forçar a alocação de uma célula lógica, permitindo que esses *loops* de sinais sejam preservados, possibilitando, então, a execução

dos experimentos.

Os circuitos utilizados para os testes descritos neste trabalho foram escolhidos por ordem de sua complexidade, sendo os mais simples os diversos tipos de *latches* (assíncronos), tais como, *latch* SR, *latch* D, *latch XOR*, *latch* JK, *latch* D multiplexada, *latch* de duas portas e *latch* BILBO. Os circuitos mais complexos testados foram os circuitos síncronos *flip-flop* tipo D e o paridade-2.

4.1.1 *Latch* SR

A *latch* SR é o circuito mais simples que possui a característica única de circuitos sequenciais em relação aos combinacionais: persistência de sinais, ou também conhecido como memória. Sua função de excitação $Q = S + \overline{R}q$, com a restrição de que $SR = 0$, descreve um circuito que tem sua saída em 1 pela ativação de S, ou 0 pela ativação de R ou mantém o último *bit* quando ambas as entradas estão em 0. Esse comportamento está descrito na Tabela 4.1. Para referência, sua implementação usual está apresentada na Figura 4.2.

Tabela 4.1: Tabela verdade para *latch* SR.

SR	Q
00	Q
01	0
10	1

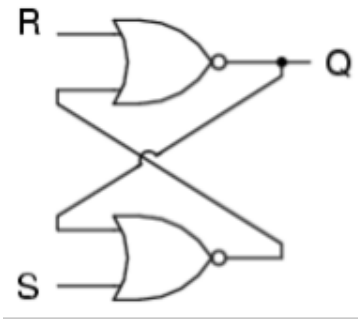


Figura 4.2: Circuito de referência para implementação da *latch* SR.

Para a evolução desse circuito específico, foi utilizada a sequência presenta na Tabela I.1. Por ser uma sequência mínima que descreve o comportamento da *latch* SR [48], ela é suficientemente adequada para a execução do experimento.

Os parâmetros específicos dos circuitos são $(N, T, U) = (6, 2, 1)$. Assim, este experimento é executado com uma sequência de tamanho $N = 6$, onde o circuito a ser evoluído possui $T = 2$ *bits* de entrada e $U = 1$ *bit* de saída.

Como esse experimento produz um circuito muito simples, todas as soluções encontradas foram idênticas umas às outras, sendo também iguais ao circuito de referência, visto na Figura 4.2. Um exemplo de *bitstream* de saída, gerado como resultado da execução do algoritmo genético, que descreve um desses circuitos, é 00000000000000000000000000000000000000000000100100110001000000001110001001000000.

Como o circuito VRC utilizado possui uma configuração máxima de 25 células lógicas disponíveis e como o cromossomo deve especificar a configuração de todas essas células lógicas, em conjunto com suas saídas, todos tem o mesmo tamanho fixo de 423 *bits*. Assim, mesmo um circuito como esse, que se utiliza de apenas 2 células lógicas, tem um cromossomo que descreve todos os 25.

A simulação temporal do circuito encontrado neste experimento, baseado no FPGA da família *Cyclone IV*, pode ser vista na Figura 4.3. As linhas indicam os seguintes comportamentos no tempo:

- *KEY[3]*: Sinal de carga do cromossomo na VRC
- *KEY[0]*: Sinal de reset da sequência
- *KEY[1]*: Sinal que indica o *clock* utilizado para a troca de elementos das sequências.
- *entrada*: Entrada inserida ao indivíduo avaliado.
- *Cromossomo*: Identificação do Cromossomo utilizado
- *Sequencia*: Identificação da sequência utilizada
- *endereco*: Entrada que indica o índice do elemento das sequências sendo avaliado.
- *validacao[0]*: *Bit* de índice 0 que indica quando a avaliação do indivíduo deve levar em conta sua saída atual no cálculo de seu *fitness*. Nas simulações, quando esse sinal volta ao nível 0, isso sinaliza que não há mais elementos na sequência.
- *saida[0]*: *Bit* 0 da saída esperada do indivíduo.
- *fenotipo[0]*: *Bit* 0 da saída do cromossomo em sua representação por VRC.
- *circuito[0]*: *Bit* 0 da saída do circuito equivalente ao cromossomo sem a presença do mecanismo VRC.

- *referencia[0]*: *Bit* 0 da saída do circuito de referência.

No gráfico, a linha azul vertical no tempo $2.0\mu s$ indica a partir de quando a saída do cromossomo é válida.

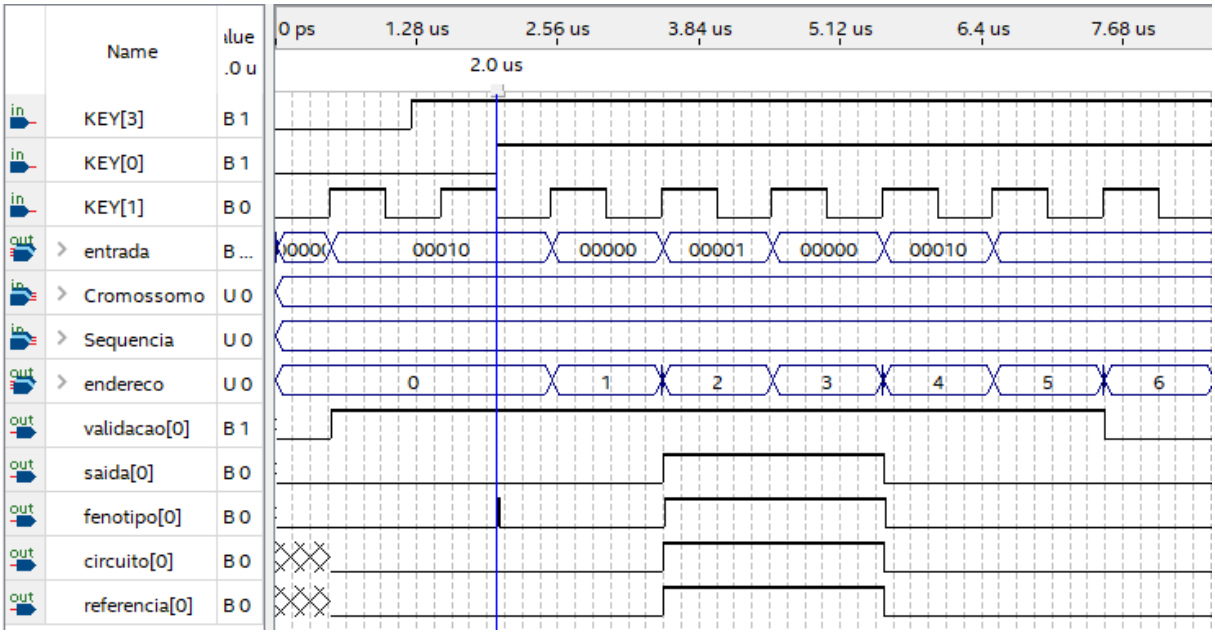


Figura 4.3: Simulação temporal da *latch* SR por *Modelsim*.

Nessa simulação é possível notar que o indivíduo obtido se comporta como esperado. Há uma pequena oscilação presente na linha *fenotipo[0]*, logo no início da avaliação da sequência, indicado pelo nível da linha *KEY[0]*, mas dentro da janela de 100ns dada aos indivíduos para estabilização. A simulação deste circuito para o FPGA da família *Cyclone II* pode ser vista na Figura III.1. Note que essa simulação apresentou o mesmo comportamento oscilatório inicial obtido pela *Cyclone IV* para o indivíduo em conjunto com seu circuito *vrc*.

4.1.2 *Latch* D

Como a *latch* SR, a *latch* D também é um dos circuitos mais simples capaz de persistir sinais, sua função de excitação pode ser descrita como $Q = CD + \overline{C}q$. Diferentemente da *latch* SR, porém, a *latch* D utiliza sinais de entrada C, que habilita ou não a mudança do estado da *latch*, e D, que é o dado a ser armazenado. Uma consequência deste comportamento é que o circuito se torna total, característica não presente na *latch* SR por proibir a combinação $SR = 1$. Seu comportamento completo, incluindo o circuito de referência, pode ser revisado na seção 2.2.1.

Pelo circuito ser total, a sequência de evolução também precisa levar em conta mais casos do que aquela utilizada para a evolução da *latch* SR. Ela pode ser vista na Ta-

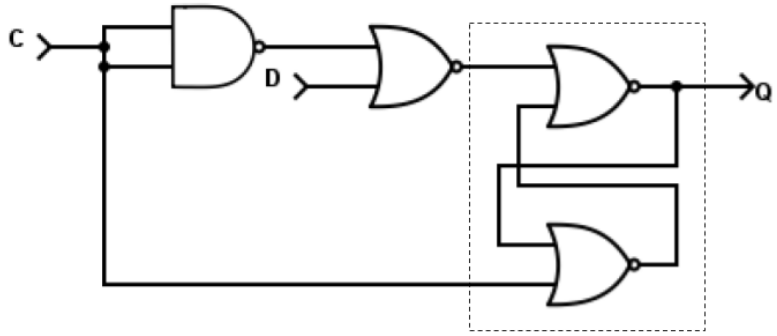


Figura 4.4: Um dos circuitos, implementando o comportamento da *latch* D, encontrados pela evolução, com *latch* SR interna destacada. Simulação pela *Cyclone IV* na Figura II.2 e pela *Cyclone II* na Figura III.2.

bela I.2. Essa sequência também é garantida ser a miníma necessária para descrever o comportamento de uma *latch* D [48].

Os parâmetros de evolução específicos para este experimento são $(N, T, U) = (7, 2, 1)$.

Como a implementação de referência já é bem eficiente, em termos de número de transistores necessários, não foram encontradas muitas soluções melhores. Ainda assim, um circuito, se utilizando da combinação de entradas proibida da *latch* SR, foi encontrado, possibilitando a economia de 2 transistores em relação à referência. Ele pode ser visto na Figura 4.4. Nesse circuito também não há a possibilidade dessa *latch* SR interna ter uma transição de 11 para 00, que pode causar uma oscilação, devido ao atraso de propagação inerente à entrada C e às portas *NAND* e *NOR* ligadas a ela. Sua simulação temporal no FPGA da família *Cyclone IV* pode ser vista na Figura 4.5. Sua simulação no FPGA *Cyclone II* pode ser vista na Figura III.2.

Outros circuitos encontrados para a *latch* D podem ser vistos na Figura 4.6. Suas respectivas simulações podem ser encontradas nas Figura II.3, Figura II.4 e Figura II.5 para a *Cyclone IV* e nas Figura III.3, Figura III.4 e Figura III.5 para a *Cyclone II*. Observe que, comparando as simulações do circuito da Figura 4.6b, nos gráficos vistos na Figura II.4 e na Figura III.4, é possível notar que ambos o cromossomo e seu circuito real possuem comportamento correto para a família *Cyclone IV*, mas incorreto para a família *Cyclone II*.

4.1.3 *Latch XOR*

A *latch XOR* possui a função de excitação $Q = C(D \oplus S) + \overline{C}q$. Ela pode ser informalmente descrita como uma *latch* D em que sua entrada D está ligada a uma porta *XOR* ligada

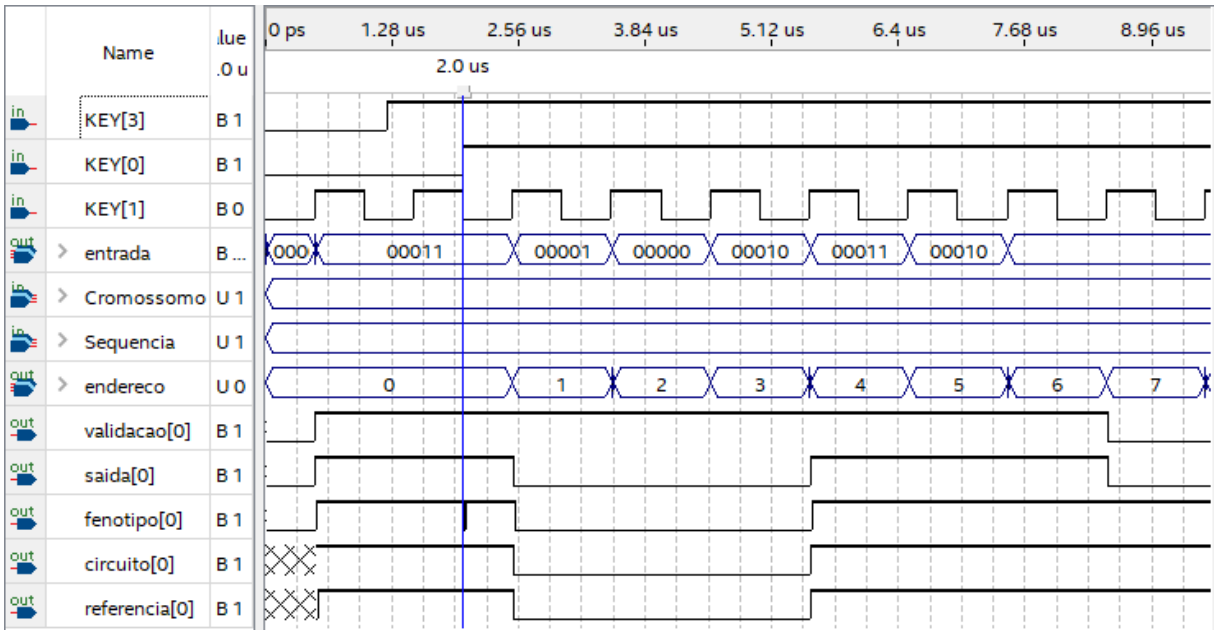
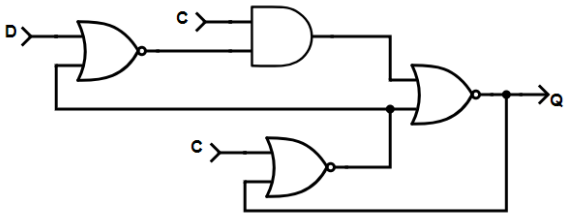
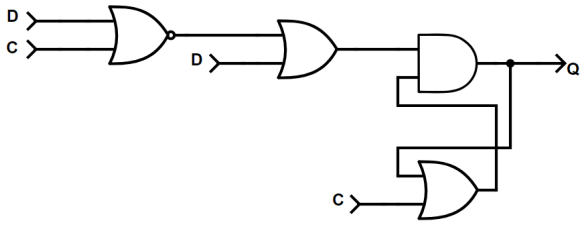


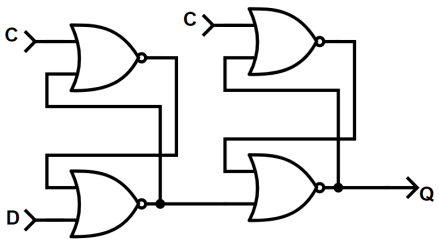
Figura 4.5: Simulação temporal do circuito apresentado na Figura 4.4.



(a) Simulação pela *Cyclone IV* na Figura II.3 e pela *Cyclone II* na Figura III.3.



(b) Simulação pela *Cyclone IV* na Figura II.4 e pela *Cyclone II* na Figura III.4.



(c) Simulação pela *Cyclone IV* na Figura II.5 e pela *Cyclone II* na Figura III.5.

Figura 4.6: Outros circuitos encontrados pela evolução da *latch* D.

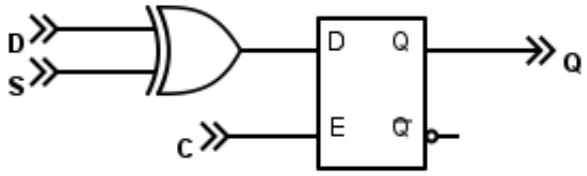


Figura 4.7: Circuito de referência para a *latch XOR*.

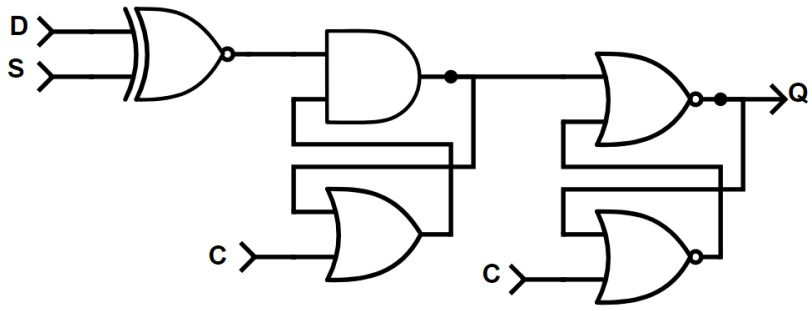


Figura 4.8: Circuito evoluído com o comportamento de uma *latch XOR*. Simulação pela *Cyclone IV* na Figura II.8 e pela *Cyclone II* na Figura III.8.

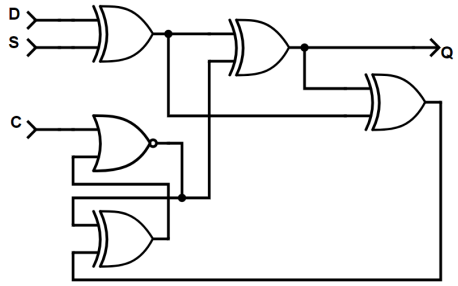
às entradas D e S, sendo, portanto uma *latch* de três entradas. Seu circuito de referência pode ser visto na Figura 4.7 e sua tabela verdade na Tabela 4.2.

CSD	Q
0XX	Q
100	0
101	1
110	1
111	0

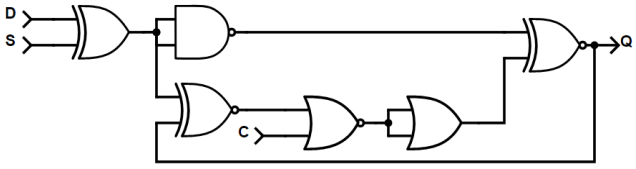
Tabela 4.2: Tabela verdade para *latch XOR*.

A sequência utilizada para evolução, também garantida ser mínima [48], pode ser vista na Tabela I.3. Assim, seus parâmetros de evolução são $(N, T, U) = (13, 3, 1)$.

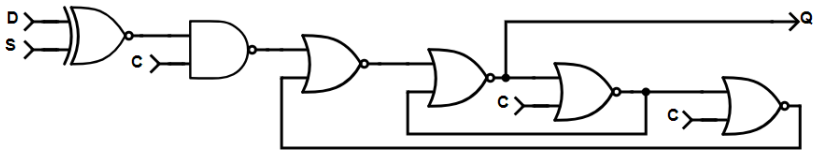
A Figura 4.8 mostra um dos circuitos encontrados pela evolução cujas simulações podem ser vistas na Figura II.8 e na Figura III.8. Seu funcionamento envolve uma *latch* SR cuja entrada *set* está ligada a C e a entrada *reset* ao restante do circuito. Para a saída se tornar 1, esse restante tem como saída 0 e para se tornar 1, o circuito utiliza o estado proibido em que ambas as entradas da *latch* SR são 1 para forçar 0 em sua saída. O *feedback* entre as portas *AND* e *OR* mantém a saída da porta *AND* em 1 quando as entradas transicionam de entradas como $CSD = 001$ para $CSD = 000$.



(a) Simulação pela *Cyclone IV* na Figura II.6 e pela *Cyclone II* na Figura III.6.



(b) Simulação pela *Cyclone IV* na Figura II.7 e pela *Cyclone II* na Figura III.7.



(c) Simulação pela *Cyclone IV* na Figura II.9 e pela *Cyclone II* na Figura III.9.

Figura 4.9: Outros circuitos encontrados pela evolução da *latch XOR*.

Outros circuitos evoluídos podem ser vistos na figura Figura 4.9 e suas simulações na Figura II.6, Figura II.7, Figura II.9 (*Cyclone IV*) e Figura III.6, Figura III.7, Figura III.9 (*Cyclone II*). Em especial, os circuitos da Figura 4.9a e da Figura 4.9b são circuitos que foram evoluídos, atendem a sequência utilizada para evolução, mas não tem o comportamento de uma *latch XOR*. Seus comportamentos, com as entradas $CSD = 000$, é ter 0 na saída, como esperado, porém quando as entradas, em seguida, se tornam $CSD = 010$, a saída vista é 1, diferente do esperado, com a saída mantendo-se em 0. Isso pode ser considerado um defeito da sequência utilizada por não ter esse caso explícito nela.

As simulações feitas para o circuito da Figura 4.9b, vistas na Figura II.7 e na Figura III.7 mostram comportamentos consideravelmente distintos, alterando-se apenas o FPGA simulado. Isso mostra como o funcionamento correto desse circuito depende fortemente do ambiente em que é sintetizado.

Nenhum dos circuitos apresentou características melhores que o circuito de referência neste experimento.

4.1.4 *Latch* JK

Na literatura tradicional, alguns autores [5] introduzem o *flip-flop* JK como um circuito que possui uma funcionalidade similar a um *latch* SR, invertendo seu estado interno quando $J = K = 1$. Para permitir que isso aconteça corretamente, é introduzida uma terceira entrada C que atua como um pulso que tenha uma característica de tempo suficiente para que a inversão seja feita apenas uma vez, sem que hajam oscilações. Esse

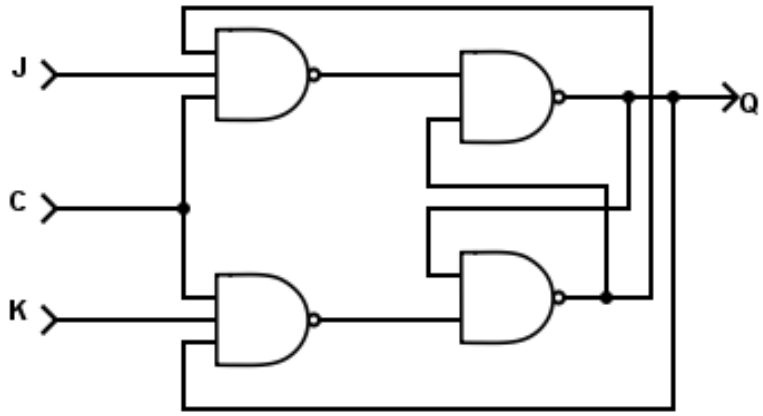


Figura 4.10: *Flip-flop* JK ativado por pulso comumente encontrado na literatura.

circuito pode ser visto na figura Figura 4.10. A parte mais delicada dessa implementação vem da necessidade do conhecimento dos atrasos internos para determinar qual é o comprimento máximo que este pulso pode tomar.

Para este experimento, evoluímos um circuito que possui apenas as entradas J e K, sem precisar de um mecanismo de pulso para controlar a inversão, invertendo seu estado interno quando J ou K transicionarem para 1 quando a outra entrada também for 1. Essencialmente, isso o torna um circuito com um comportamento síncrono, por não ser mais sensível ao nível das entradas em todas as situações. Portanto, não há um circuito de referência, mas sua tabela verdade pode ser vista na Tabela 4.3.

JK	Q
00	Q
01	0
10	1
11	$\overline{Q}$

Tabela 4.3: Tabela verdade para *latch* JK.

A sequência utilizada para evolução deste circuito pode ser vista na Tabela I.4, com os parâmetros de evolução sendo $(N, T, U) = (12, 3, 1)$.

O circuito visto na Figura 4.11 implementa o comportamento desejado de maneira compacta, necessitando de apenas 24 transistores, e utilizando-se de uma característica interessante. As entradas J e K estão ligadas a duas portas *NOR* que podem ser tomadas como uma *latch* SR. Acima delas, à esquerda, há outras duas portas *NOR* formando uma outra *latch* SR, que tem como suas entradas *set* e *reset*, as saídas da primeira *latch*. Para as entradas 00, 10, 01, essa *latch* superior não atua na saída do circuito, mas quando a entrada se torna 11, o valor que estava armazenado nela é usado para decidir a inversão

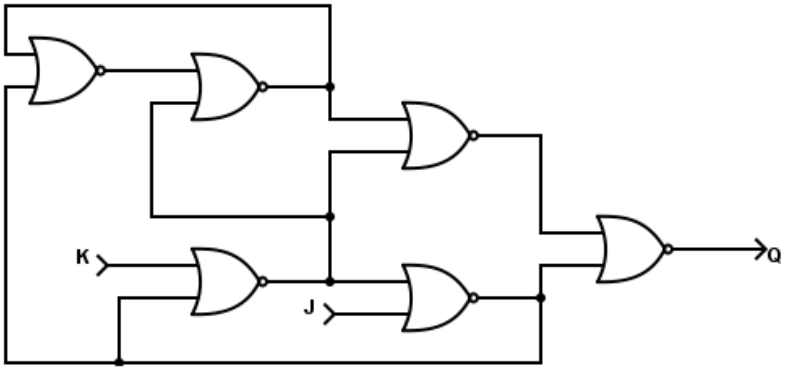


Figura 4.11: Circuito evoluído com o comportamento de uma *latch* JK, como definido pela sequência na Tabela I.4. Simulação pela *Cyclone IV* na Figura II.11 e pela *Cyclone II* na Figura III.11.

da saída anterior do circuito. Note que suas simulações, vistas na Figura II.11 e na Figura III.11 para os FPGAs *Cyclone IV* e *Cyclone II* respectivamente, mostram um comportamento que não se altera, sugerindo que este circuito possui uma estrutura que *não* depende de seu ambiente para o funcionamento correto. A oscilação vista ao final ocorre após o término da avaliação de sua sequência de entradas, portanto o circuito ainda pode ser considerado correto.

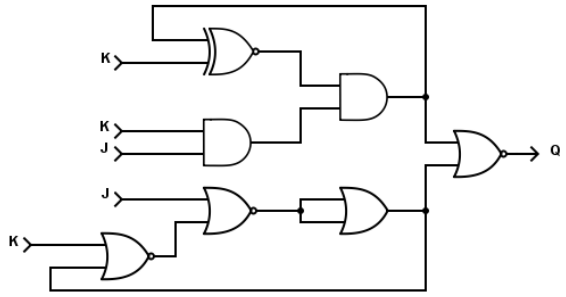
Outros circuitos encontrados para a *latch* JK podem ser vistos na Figura 4.12 e suas simulações na Figura II.10, Figura II.12 e Figura II.13 para a *Cyclone IV* e na Figura III.10, Figura III.12 e Figura III.13 para a *Cyclone II*.

4.1.5 *Latch* D multiplexada

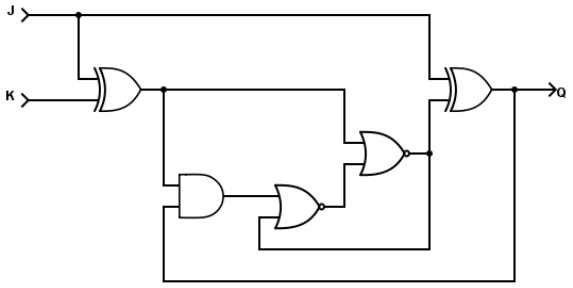
Uma *latch* D multiplexada possui um multiplexador de 2 *bits* em sua entrada, com sua função de excitação sendo $Q = C(TS + \overline{T}D) + \overline{C}q$. O circuito de referência para esse experimento pode ser visto na Figura 4.13 e sua tabela verdade na Tabela 4.4.

A sequência utilizada para evolução, também garantida ser mínima [48], está apresentada na Tabela I.5. Parâmetros de evolução $(N, T, U) = (26, 4, 1)$.

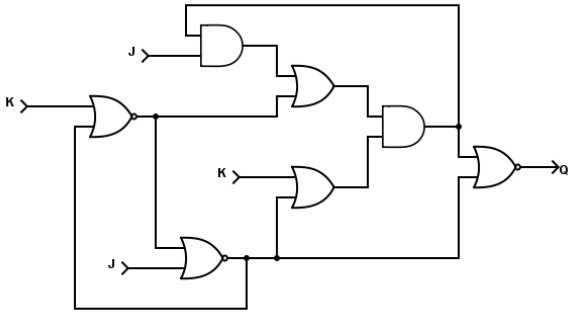
Uma das características mais interessante de um dos circuitos encontrados pelo algoritmo genético, visto na Figura 4.14, se utiliza dos atrasos das portas lógicas ligadas à entrada C para conseguir armazenar o nível 0. Quando $C = 0$, a porta *NAND* na qual está diretamente ligada se torna 1 e, se o sinal de saída estiver em 0, a porta *OR* na qual a entrada está ligada se torna 0. O sinal dessa porta *OR* se propaga mais rapidamente para a porta *AND* do que o sinal da porta *NAND*, que precisa passar por uma porta *OR* antes de chegar na *AND*. O resultado é que o sinal da *AND* nunca será 1, portanto a *OR* ligada à entrada também se mantém em 0. Suas simulações, vistas na Figura II.16



(a) Simulação pela *Cyclone IV* na Figura II.10 e pela *Cyclone II* na Figura III.10.



(b) Simulação pela *Cyclone IV* na Figura II.12 e pela *Cyclone II* na Figura III.12.



(c) Simulação pela *Cyclone IV* na Figura II.13 e pela *Cyclone II* na Figura III.13.

Figura 4.12: Outros circuitos encontrados pela evolução da *latch* JK.

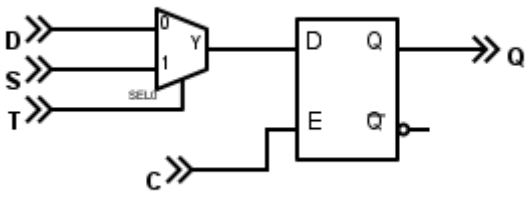


Figura 4.13: Circuito de referência para implementação da *latch* D multiplexada.

Tabela 4.4: Tabela verdade para a *latch* D multiplexada.

CTSD	Q
0XXX	Q
1000	0
1001	1
1010	0
1011	1
1100	0
1101	0
1110	1
1111	1

e na Figura III.16, mostram comportamentos idênticos, indicando que, apesar desse fator de atrasos inerente ao circuito, ele se comporta corretamente independente de onde é avaliado.

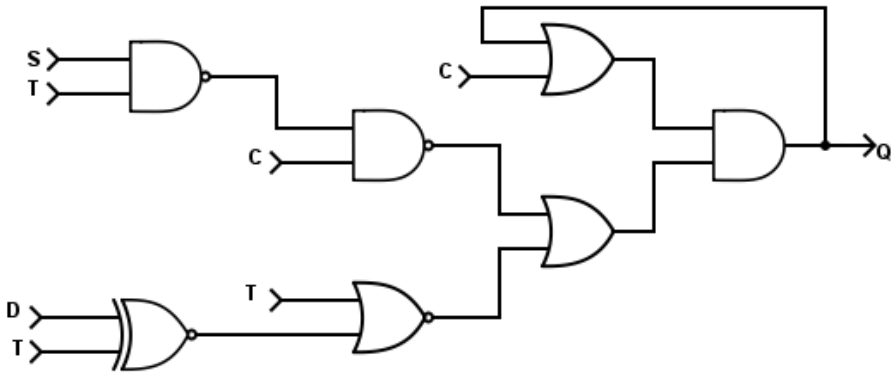
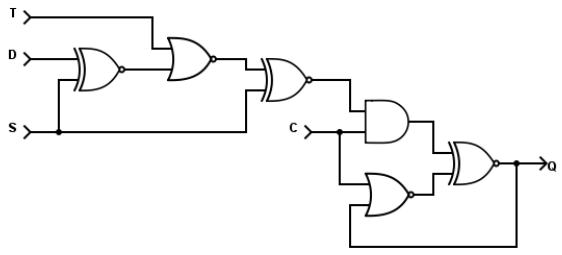


Figura 4.14: Um dos circuitos encontrados pelo algoritmo genético para o experimento da *latch* D multiplexada. Simulação pela *Cyclone IV* na Figura II.16 e pela *Cyclone II* na Figura III.16.

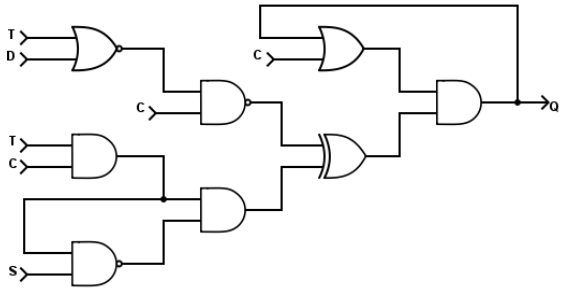
Mais circuitos encontrados pela evolução podem ser vistos na Figura 4.15. As simulações para esses circuitos podem ser vistas na Figura II.14, na Figura II.15 e na Figura II.17 (*Cyclone IV*) e na Figura III.14, na Figura III.15 e na Figura III.17 (*Cyclone II*).

4.1.6 *Latch* de duas portas

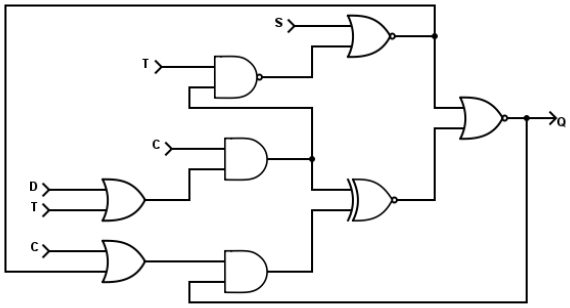
Uma *latch* de duas portas possui duas entradas de controle, com a entrada determinada pelo controle ativo, com uma função de excitação $Q = C_1D_1 + C_2D_2 + \overline{C_1} \cdot \overline{C_2}q$. Por conta dessa característica, a restrição $C_1C_2 = 0$ deve ser feita. Sua tabela verdade, portanto, pode ser vista na Tabela 4.5 e seu circuito de referência na Figura 4.16.



(a) Simulação pela *Cyclone IV* na Figura II.15 e pela *Cyclone II* na Figura III.15.



(b) Simulação pela *Cyclone IV* na Figura II.15 e pela *Cyclone II* na Figura III.15.



(c) Simulação pela *Cyclone IV* na Figura II.17 e pela *Cyclone II* na Figura III.17.

Figura 4.15: Outros circuitos encontrados pela evolução da *latch* D multiplexada.

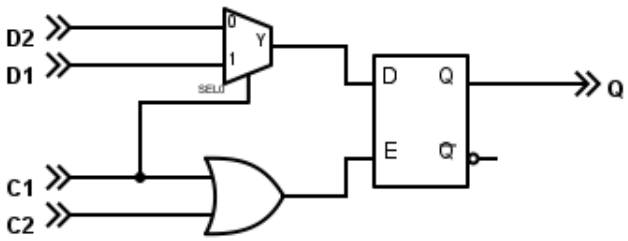


Figura 4.16: Circuito de referência de implementação para a *latch* de duas portas.

Tabela 4.5: Tabela verdade para a *latch* de duas portas.

$C_1D_1C_2D_2$	Q
0X0X	Q
100X	0
110X	1
0X10	0
0X11	1

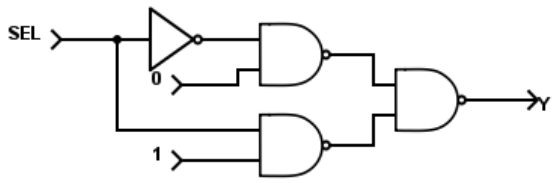


Figura 4.17: Multiplexador 2x1.

A sequência utilizada para evolução, também mínima [48], está apresentada na Tabela I.6. Os parâmetros para evolução são $(N, T, U) = (23, 4, 1)$.

Se considerarmos a implementação de um multiplexador 2x1 como aquele visto na Figura 4.17, contendo um total de 14 transistores, podemos comparar o aproveitamento de espaço da implementação de referência, totalizando, então, 36 transistores, com os circuitos evoluídos. Em especial, apesar do circuito presente na Figura 4.18 apresentar o mesmo número de transistores, 36, e o mesmo comportamento, sua organização se mostra ser consideravelmente distinta da referência. Suas simulações, vistas na Figura II.18 e Figura III.18, mostram que o circuito se comporta corretamente em todas as circunstâncias.

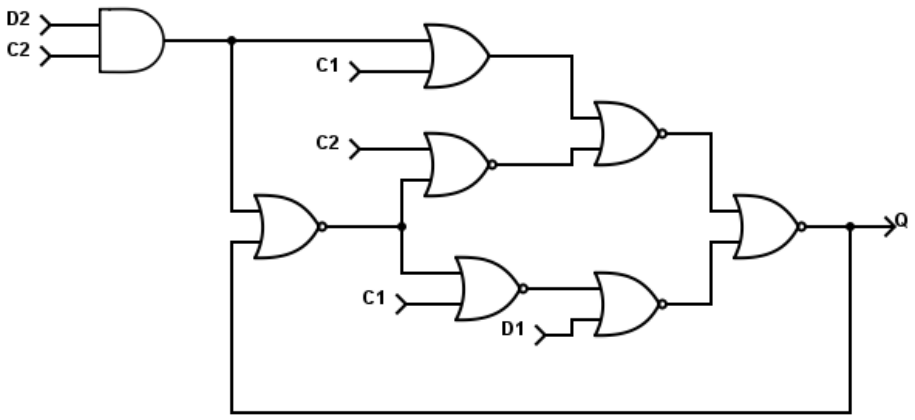
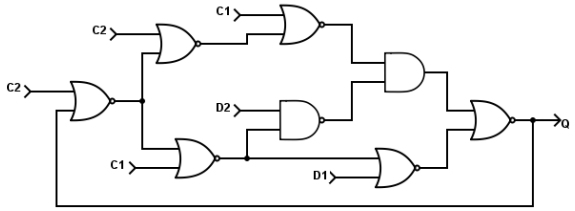
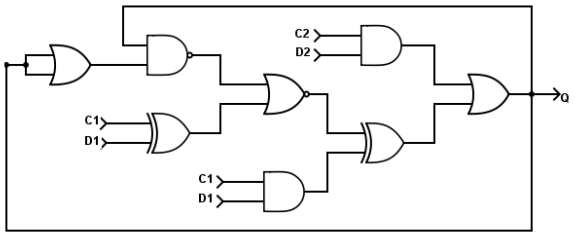


Figura 4.18: Circuito evoluído para *latch* de duas portas. Simulação pela *Cyclone IV* na Figura II.18 e pela *Cyclone II* na Figura III.18.

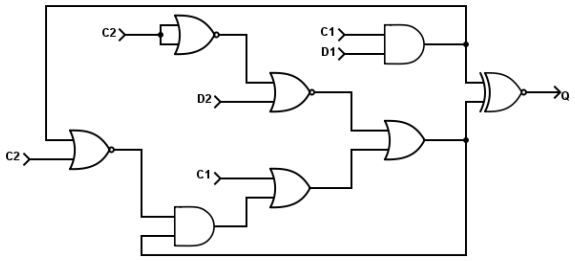
Mais circuitos encontrados pela evolução podem ser vistos na Figura 4.19 e suas simulações na Figura II.19, Figura II.20, Figura II.21 para a *Cyclone IV*. As simulações desses circuitos para a *Cyclone II* podem ser vistas na Figura III.19, na Figura III.20 e na Figura III.21. Um detalhe interessante é que as simulações do circuito da Figura 4.19b, vistas na Figura II.20 e na Figura III.20, mostram que o circuito se mostrou funcionar corretamente somente quando encontrado originalmente pela evolução, no FPGA *Cyclone V*. Em todos os outros testes, mostrou um comportamento incorreto, indicando que sua funcionalidade depende completamente do ambiente em que foi evoluído.



(a) Simulação pela *Cyclone IV* na Figura II.19 e pela *Cyclone II* na Figura III.19.



(b) Simulação pela *Cyclone IV* na Figura II.20 e pela *Cyclone II* na Figura III.20.



(c) Simulação pela *Cyclone IV* na Figura II.21 e pela *Cyclone II* na Figura III.21.

Figura 4.19: Outros circuitos encontrados pela evolução da *latch* de duas portas.

4.1.7 *Latch* BILBO

A *latch* BILBO, ou *latch Built-In Logic Block Observer*, é usada para aplicações *Built-in Self-test* (BIST) [48]. Sua função de excitação $Q = C(B_1D \oplus \overline{B_2}S) + \overline{C}q$ descreve uma *latch* capaz de ser configurada para carregar D (quando $B_1B_2 = 11$), resetá-la (quando $B_1B_2 = 01$), carregar S (quando $B_1B_2 = 00$) ou carregar $S \oplus D$ (quando $B_1B_2 = 10$). Sua tabela verdade pode ser vista na Tabela 4.6 e seu circuito de referência na Figura 4.20.

A sequência de evolução, também mínima [48], está apresentada na Tabela I.7. Uma análise da sequência mostra, porém, que ela não representa a função de excitação descrita. No primeiro e no quinto elemento da sequência, por exemplo, há uma mudança de saída

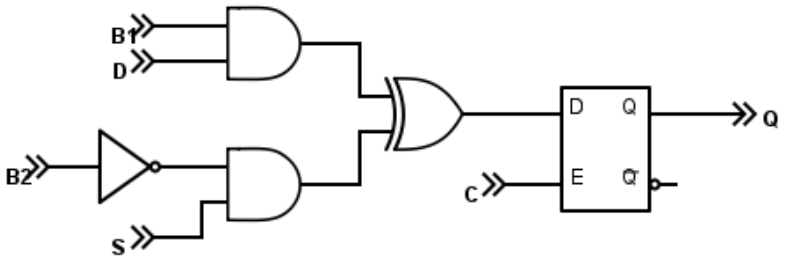


Figura 4.20: Circuito de referência utilizado para a *latch* BILBO.

Tabela 4.6: Tabela verdade para a *latch* BILBO.

DSB_2B_1C	Q
XXXX0	Q
X0001	0
X1001	1
XX101	0
00011	0
10011	1
01011	1
11011	0
0X111	0
1X111	1

para 1 quando a entrada C é habilitada, mesmo todas as outras estando em 0. Pela função de excitação, a saída deveria ser 0.

Ainda assim, foram encontrados circuitos para essa sequência, mesmo que não representem a funcionalidade esperada para uma *latch* BILBO. Eles podem ser vistos na figura Figura 4.23 e suas simulações na Figura II.22, na Figura II.23, na Figura II.24 e na Figura II.25 para a *Cyclone IV* e na Figura III.22, na Figura III.23, na Figura III.24 e na Figura III.25 para a *Cyclone II*. Observe que o sinal do circuito de referência não coincide com aquele visto no sinal *saida[0]*, ou seja, difere da sequência da evolução realizada por este experimento.

Outro detalhe interessante está visível nas diferentes simulações realizadas na Figura II.23 e na Figura III.23, onde ambas apresentam circuitos com grandes quantidades de sinais transientes em trocas de entradas. Em especial, é possível ver pela Figura 4.21 que o pior transiente presente na simulação para a *Cyclone IV* ainda se encontra dentro do limite de 100ns para estabilização. Por outro lado, o mesmo transiente presente na simulação para a *Cyclone II*, apresentado na Figura 4.22, mostra um intervalo de tempo entre troca da entrada e estabilização do sinal maior que 100ns, dando ao circuito um

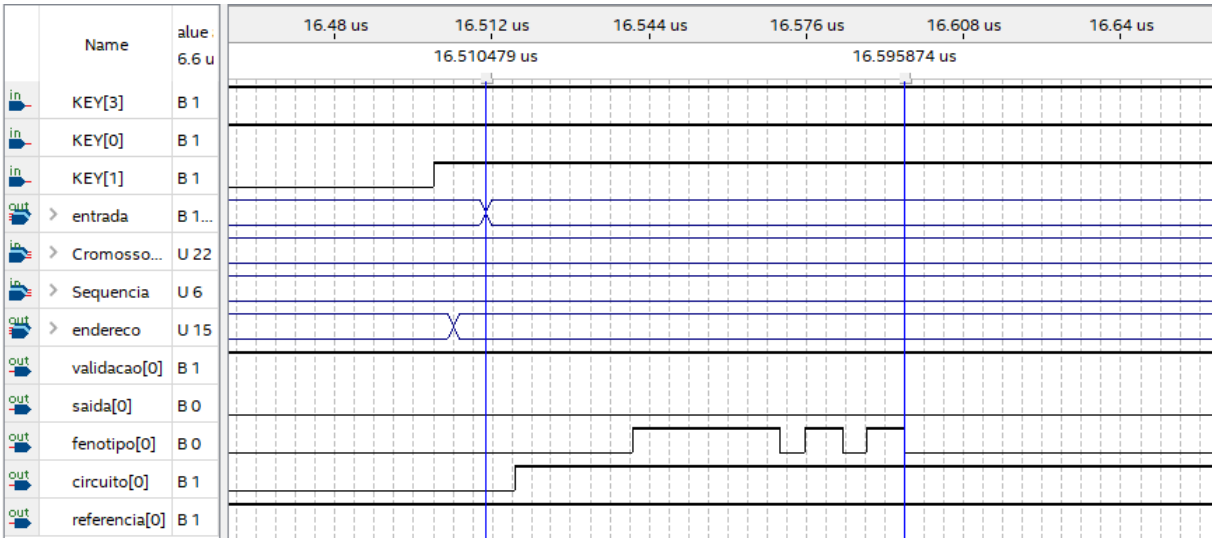


Figura 4.21: Sinal transiente de maior duração encontrado para a simulação presente na Figura II.23, para a *Cyclone IV*.

comportamento determinado como incorreto.

Como o circuito de referência e os circuitos evoluídos diferem em funcionalidade, não é possível fazer comparações entre eles. Dentre os evoluídos, a implementação presente na Figura 4.23d apresenta um total de 54 transistores, a mais eficiente em espaço.

Os parâmetros para evolução deste experimento são $(N, T, U) = (58, 5, 1)$.

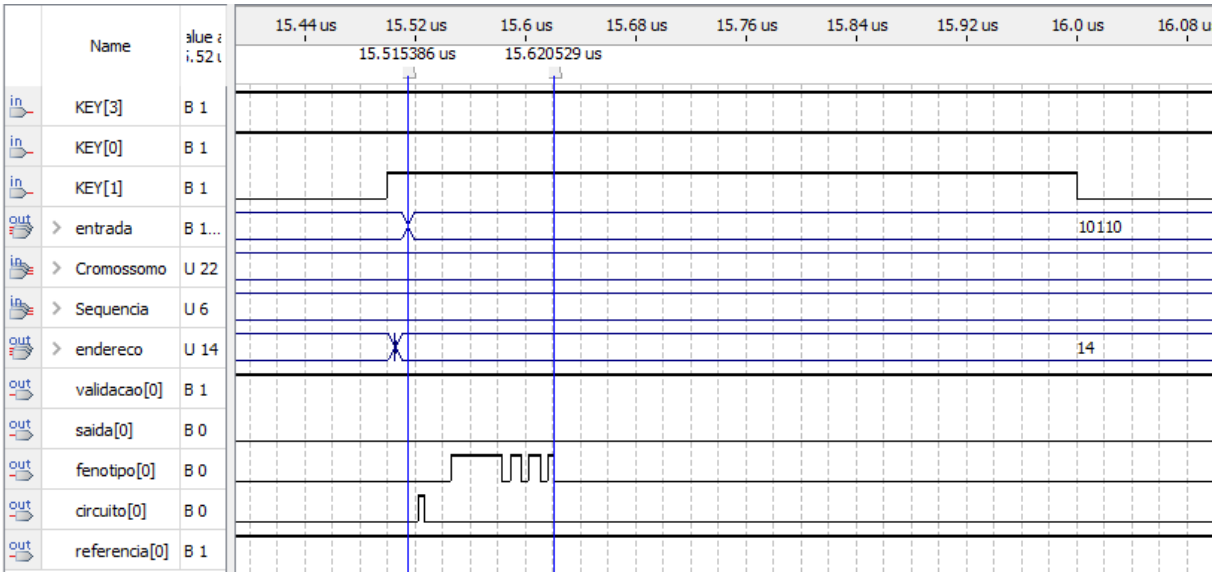
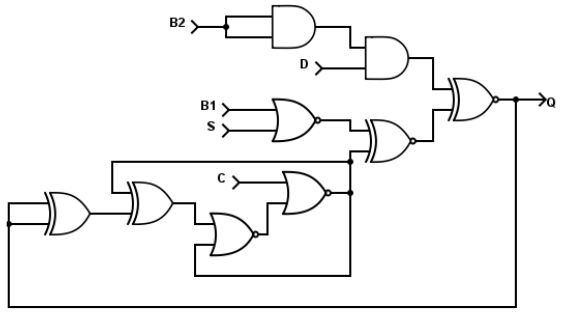
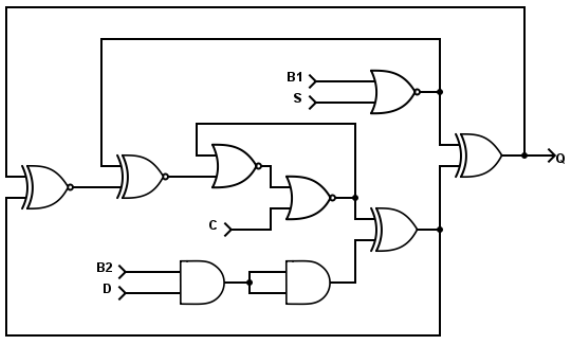


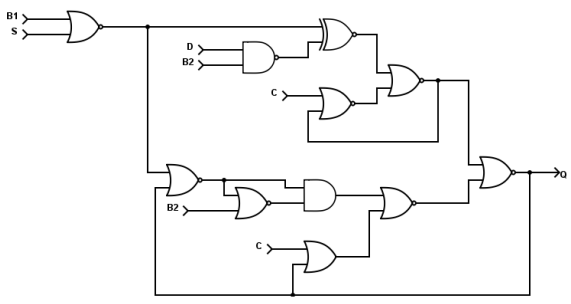
Figura 4.22: Sinal transiente de maior duração encontrado para a simulação presente na Figura III.23, para a *Cyclone II*.



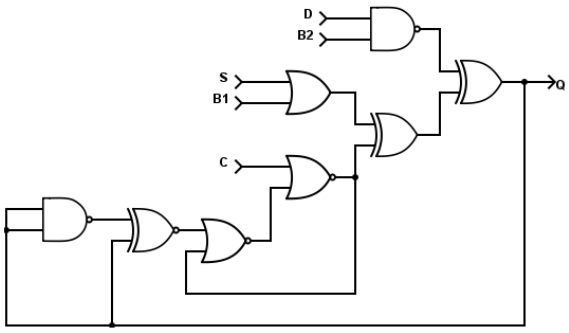
(a) Simulação pela *Cyclone IV* na Figura II.22 e pela *Cyclone II* na Figura III.22.



(b) Simulação pela *Cyclone IV* na Figura II.23 e pela *Cyclone II* na Figura III.23.



(c) Simulação pela *Cyclone IV* na Figura II.24 e pela *Cyclone II* na Figura III.24.



(d) Simulação pela *Cyclone IV* na Figura II.25 e pela *Cyclone II* na Figura III.25.

Figura 4.23: Circuitos para a *latch* BILBO encontrados por evolução a partir da sequência da Tabela I.7.

4.1.8 *Flip-flop* D

A estrutura e funcionamento de um *flip-flop* D estão descritos em detalhes na seção 2.2.1, assim como o circuito de referência didático. Para fins de comparação, a referência vista no texto, na figura Figura 2.15, está na Figura 4.24, onde se mostra sua configuração interna, possuindo um total de 32 transistores. Uma referência de implementação comercial possível pode ser vista na Figura 4.25, que se trata da implementação encontrada no componente eletrônico *SN7474*, contendo 26 transistores. Diferente das *latches*, o *flip-flop* D tem um comportamento síncrono, então ele armazena o dado D apenas quando a entrada CLK transiciona de 0 para 1.

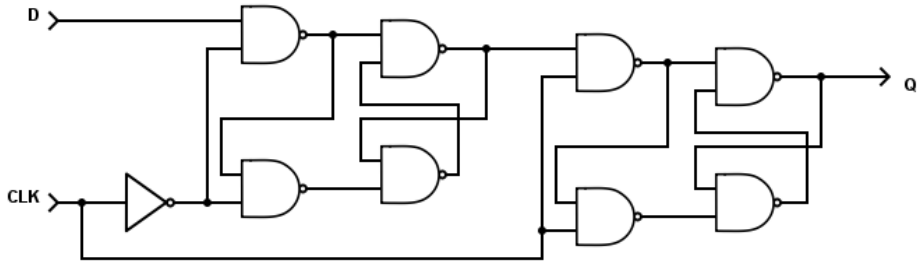


Figura 4.24: Implementação de referência para o *flip-flop* D.

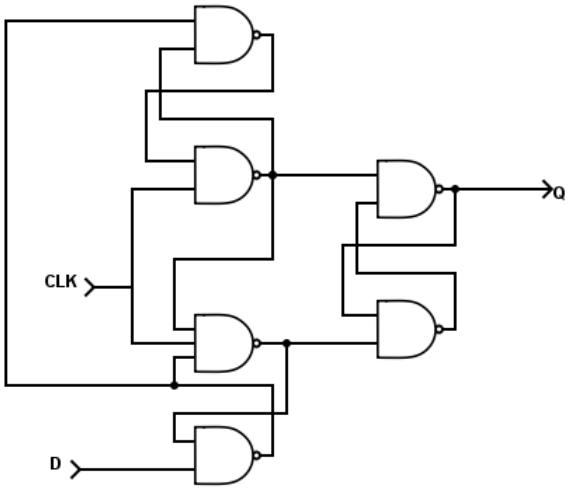


Figura 4.25: Implementação comercial para o *flip-flop* D.

Esse circuito apresenta mais uma característica interessante a ser evoluída, além do comportamento temporal, que é a lógica executada em transições de entradas. Uma sequência, projetada manualmente, que busca evoluir o *flip-flop* D levando em conta esse tipo de funcionalidade está apresentada na Tabela I.8. A sequência busca cobrir todas as

combinações possíveis de entradas, estado e transições tais como: guardar o mesmo valor dentro do circuito, tentar alterar o valor de entrada com o sinal de *clock* parado (em 0 e 1), e outras. Note que o primeiro elemento da sequência possui a entrada 00 e a saída como X, indicando que a saída não é computada no *fitness* total do indivíduo, pois não é possível determinar em qual estado inicial o *flip-flop* está nesse instante.

Para este experimento, seus parâmetros de evolução são $(N, T, U) = (22, 2, 1)$.

Como este circuito apresenta uma restrição diferente dos circuitos assíncronos, foram encontradas uma grande diversidade de soluções para este experimento. Dentre elas está o circuito que pode ser visto na Figura 4.26, que apresenta uma organização diferente do que normalmente é visto em circuitos que implementam o comportamento de *flip-flops*. Para uma implementação *CMOS* o número de transistores necessários para sua implementação é 24, ou seja, 8 transistores a menos do que a implementação de referência e 2 transistores a menos do que a comercial, vista na Figura 4.25.

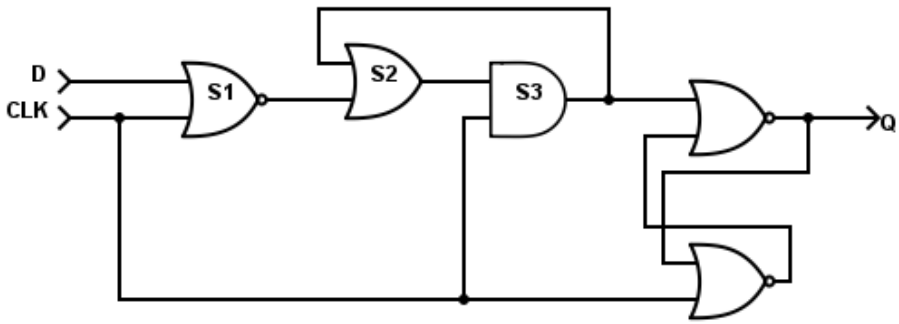


Figura 4.26: Um dos circuitos *flip-flop* D encontrados pela evolução.

O circuito é composto de uma organização de uma *latch* SR interna com a entrada de *set* ligada ao *clock* e a entrada *reset* às outras portas, rotuladas como *S1*, *S2* e *S3*. Quando sua entrada *clock* for 0, portanto, não há mudanças nessa *latch* interna, pois *S3* estará com sinal 0 necessariamente. Para analisar o armazenamento de um *bit* 1, consideremos o cenário em que $D = 1$ e $CLK = 0$, tendo $S1 = S2 = S3 = 0$ portanto. Quando $CLK = 1$, a *latch* terá sua entrada *set* alterada para 1, forçando, então, a saída 1 do circuito.

O mecanismo interessante desse circuito vem da maneira como valores $D = 0$ são armazenados. Consideremos o cenário em que $D = 0$ e $CLK = 0$, e, consequentemente, $S1 = S2 = 1$ e $S3 = 0$. Quando $CLK = 1$, temos $S1 = 0$ e, analisando o circuito temporalmente, diferentes valores possíveis para $S2$ e $S3$! Se o sinal $S1$ se propagar mais rapidamente para a porta $S2$ antes da mudança do sinal $S3$ chegar a ela, é possível que ela se torne 0, forçando então a porta $S3$ a também se tornar 0. Assim, a *latch* estaria dando

como resultado um sinal de saída 1, ou seja, um resultado incorreto. Caso a mudança do sinal $S3$ se propague mais rapidamente para a porta $S2$, então ela ficará presa ao sinal 1, sem levar em consideração a mudança de sinal da porta $S1$. Dessa maneira, com a porta $S3$ em 1, temos ambas as entradas da *latch* em 1, resultando em uma saída 0. Essa "corrida" de sinais pode ser vista na Figura 4.27, onde há a simulação *SPICE* desse circuito. Este gráfico é dividido em duas partes: a superior, mostrando as entradas do circuito e sua saída, digitalmente, e a inferior, mostrando o comportamento analógico das portas $S1$, em azul, $S2$, em vermelho, e $S3$, em verde. Como a troca de sinais na entrada da porta lógica $S2$ ocorre simultaneamente, não há tempo de seu sinal de saída se alterar e, portanto, se mantém estável em 1.

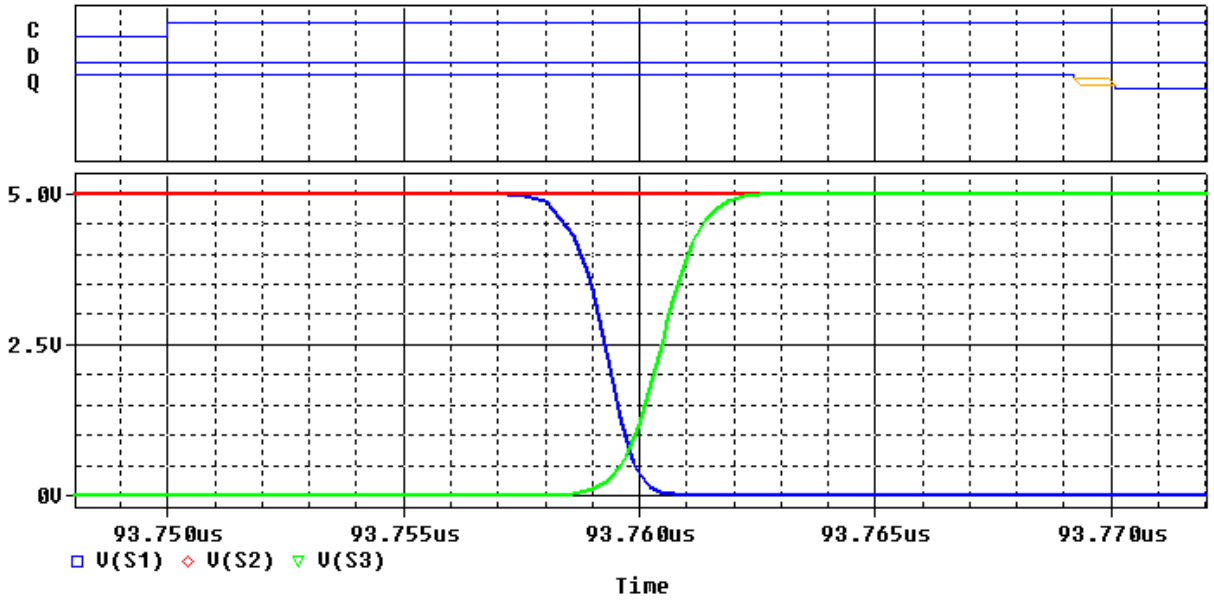


Figura 4.27: Simulação *PSpice* destacando "corrida" de sinais.

Uma comparação de tempos de propagação de sinal pode ser feita entre as diferentes implementações, simulado por *Modelsim* com base nos FPGAs *Cyclone IV*, vista na Figura 4.28 e na Figura 4.29, e *Cyclone II*, na Figura 4.30 e na Figura 4.31. Elas mostram os diferentes tempos de subida e descida do sinal de saída dos diferentes *flip-flops*. Os sinais apresentados na simulação possuem os seguintes significados:

- *in[1]*: Corresponde à entrada D.
- *in[0]*: Corresponde ao *clock*.
- *Elemento_Logico*: *Flip-flop* presente no elemento lógico do FPGA.
- *FFD_7474*: Implementação do *flip-flop* presente no componente eletrônico 7474, fornecido pelo *software Quartus Prime*.

- *Evoluído*: Indivíduo evoluído visto na Figura 4.26.
- *Referencia*: Referência didática, vista na Figura 2.15.
- *Comercial*: Implementação do *flip-flop* visto na Figura 4.25.

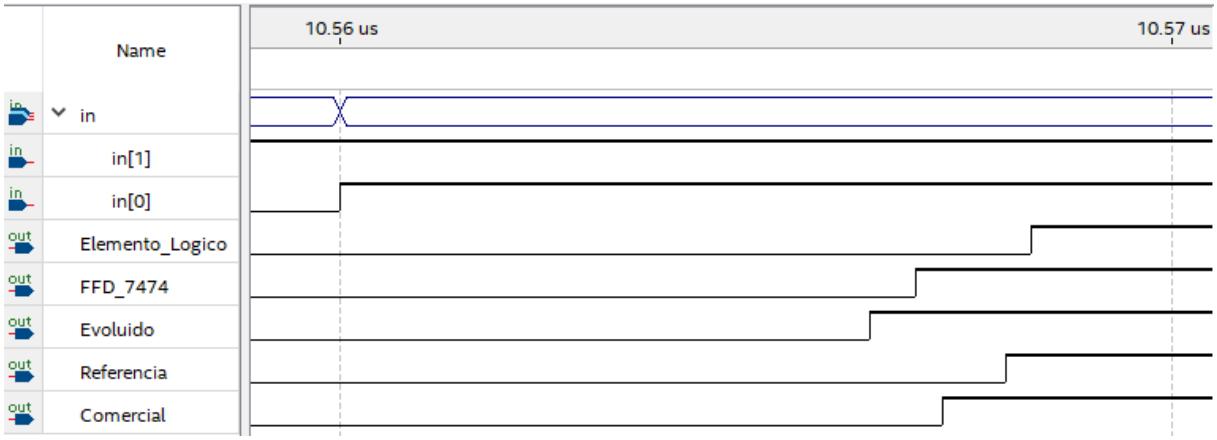


Figura 4.28: Tempos de subida para as diferentes implementações do *flip-flop* D (*Cyclone IV*).

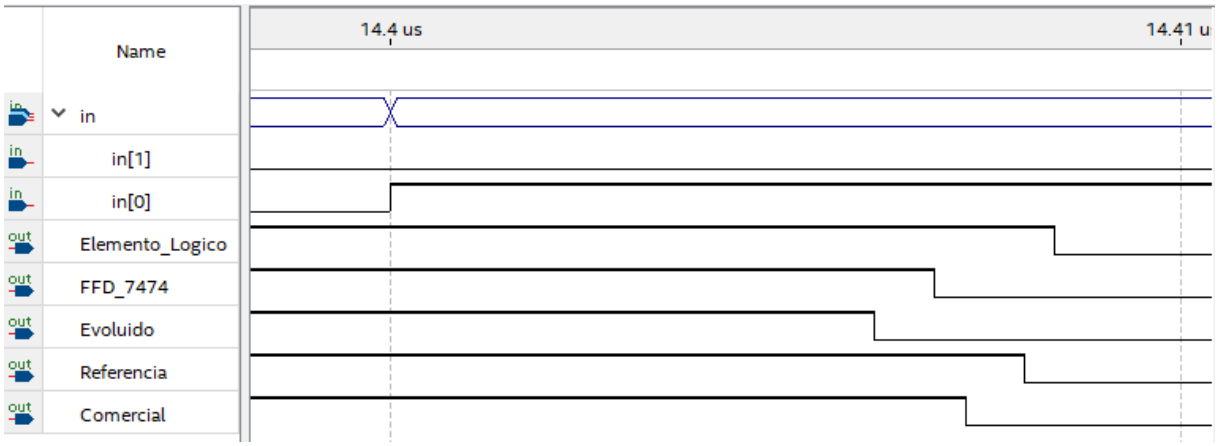


Figura 4.29: Tempos de descida para as diferentes implementações do *flip-flop* D (*Cyclone IV*).

Dos gráficos das Figura 4.28, Figura 4.29, Figura 4.30 e Figura 4.31, obtém-se os tempos de clock para saída (t_{co}) para as transições *low to high* (t_{lh}) e *high to low* (t_{hl}) mostrados na Tabela 4.7, para a *Cyclone IV*, e na Tabela 4.8, para a *Cyclone II*.

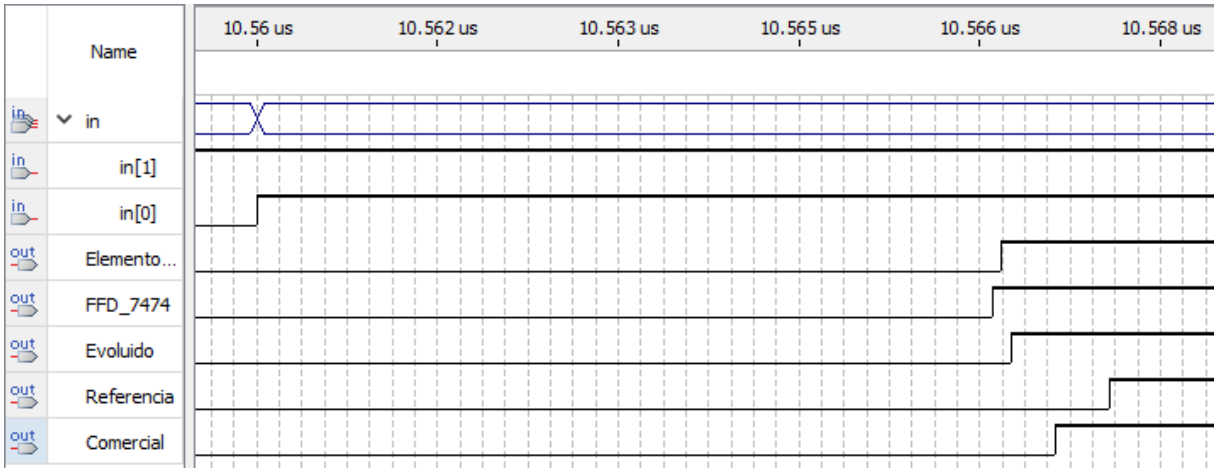


Figura 4.30: Tempos de subida para as diferentes implementações do *flip-flop* D (*Cyclone II*).

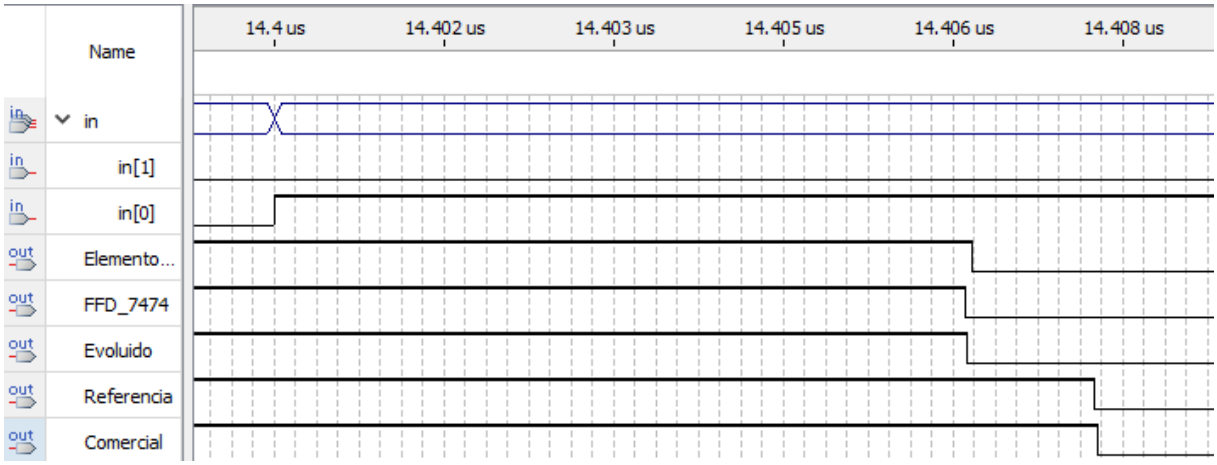


Figura 4.31: Tempos de descida para as diferentes implementações do *flip-flop* D (*Cyclone II*).

Tabela 4.7: Tempos (em *ns*) das transições *clock-output* pela simulação para a *Cyclone IV*.

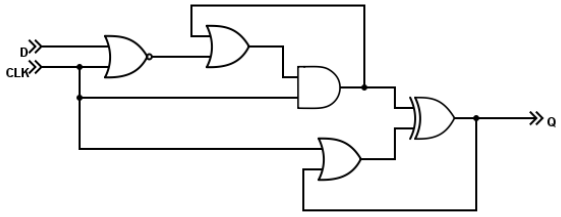
Tipo	t_{lh}	t_{hl}
LE	8,295	8,402
7474	6,908	6,888
Evoluído	6,368	6,135
Referência	7,995	8,027
Comercial	7,291	7,291

Tabela 4.8: Tempos (em ns) das transições *clock-output* pela simulação para a *Cyclone II*.

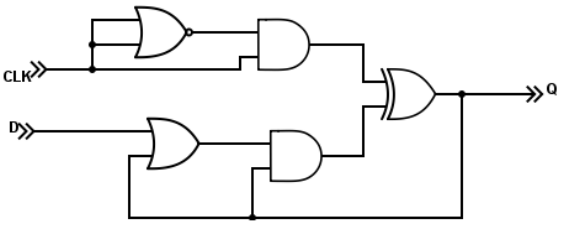
Tipo	t_{lh}	t_{hl}
LE	6,587	6,587
7474	6,651	6,512
Evoluído	6,678	6,543
Referência	7,550	7,741
Comercial	7,074	7,771

É possível notar que, dentre as 5 implementações testadas no *chip* FPGA *Cyclone IV*, a implementação evoluída, da Figura 4.26, apresenta o menor tempo para mudanças serem propagadas para sua saída, sendo cerca de 25% ($2ns$) mais rápido que o *flip-flop* original existente no Elemento Lógico da *Cyclone IV*. Em contrapartida, para o FPGA *Cyclone II*, o circuito evoluído possui um tempo 0,4% maior para a subida e descida quando comparado à implementação *7474*, a mais eficiente para essa simulação.

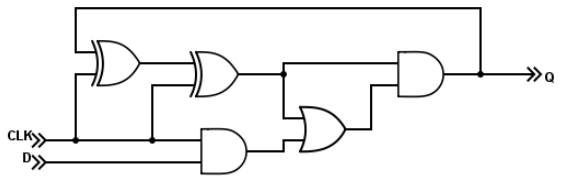
Outros circuitos encontrados neste experimento podem ser vistos na Figura 4.32. Suas simulações para a *Cyclone IV* podem ser vistas na Figura II.26, na Figura II.27, na Figura II.28 e na Figura II.29. Para a *Cyclone II*, na Figura III.26, na Figura III.27, na Figura III.28 e na Figura III.29.



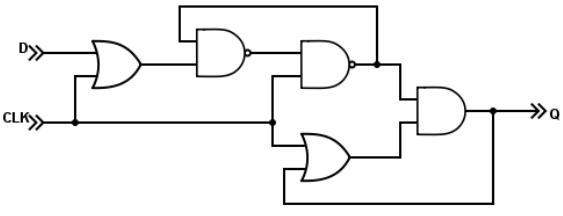
(a) Simulação pela *Cyclone IV* na Figura II.26 e pela *Cyclone II* na Figura III.26.



(b) Simulação pela *Cyclone IV* na Figura II.27 e pela *Cyclone II* na Figura III.27.



(c) Simulação pela *Cyclone IV* na Figura II.28 e pela *Cyclone II* na Figura III.28.



(d) Simulação pela *Cyclone IV* na Figura II.29 e pela *Cyclone II* na Figura III.29.

Figura 4.32: Outros circuitos encontrados pela evolução do *flip-flop* D.

Destes circuitos, no entanto, nenhum obteve desempenho melhor que o apresentado na Figura 4.26.

4.1.9 Paridade-2

Um circuito paridade-2 síncrono é aquele que, dada uma sequência de entradas de 1 *bit*, produz uma sequência de saídas de 1 *bit* realizando a operação *XOR* entre a entrada atual e a anterior, sincronizados por um *clock*. Em essência, esse circuito necessita de algum mecanismo para armazenar o *bit* e um processamento envolvendo amostras temporais. Seu circuito de referência pode ser visto na figura 3.2.

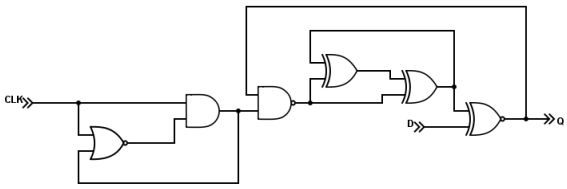
A sequência utilizada para a evolução é igual àquela usada para a evolução do *flip-flop* D, vista na Tabela I.8, com a diferença de que as saídas computam a operação *XOR* envolvendo a entrada A atualmente armazenada e a presente na entrada do circuito. Esta sequência pode ser vista na Tabela I.9. Esse experimento possui os parâmetros de evolução $(N, T, U) = (22, 2, 1)$.

Alguns dos circuitos encontrados pela evolução estão apresentados na Figura 4.33. Suas simulações para a *Cyclone IV* podem ser vistas na Figura II.30, na Figura II.31, na Figura II.32 e na Figura II.33. Para a *Cyclone II*, na Figura III.30, na Figura III.31, na Figura III.32 e na Figura III.33.

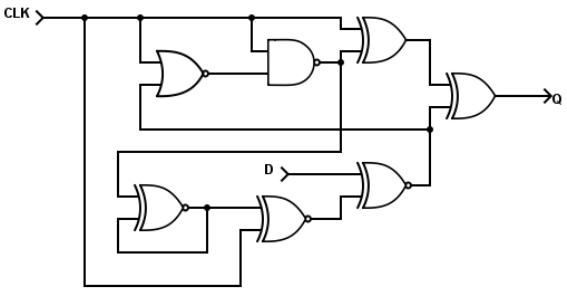
Pelas simulações é possível notar que essa evolução foi uma das que se mostraram de mais difícil generalização da solução, tendo um comportamento errado em quase todos os casos, incluindo o fato de que, para nenhum circuito real, a simulação foi capaz de gerar saídas determinadas para sequer parte do processo de avaliação da sequência, como pode ser visto na Figura II.30 e na Figura III.30 por exemplo. Ainda assim, a presença do circuito VRC faz com que algumas das simulações tenham a funcionalidade esperada, como pode ser visto, por exemplo, na Figura II.33.

4.2 Sumário

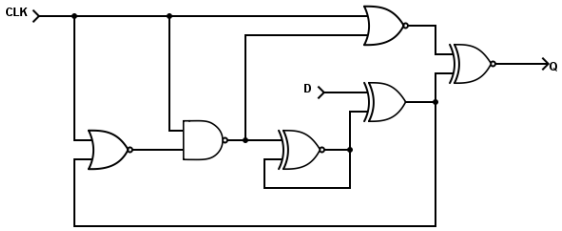
A Tabela 4.9 possui os resultados de simulações e verificações físicas dos circuitos, referenciadas por suas figuras, onde *70*, *115* e *1* indicam os FPGAs das placas *DE2-70*, *DE2-115* e *DE1-SoC* respectivamente e "M" e "F" indicam simulações por *Modelsim* e verificações físicas, estando vazias se forem consideradas corretas ou marcadas com o símbolo "✗" senão. Além disso, a tabela apresenta uma parte indicada como "Puro", que denota a descrição dos circuitos evoluídos por portas lógicas em *SystemVerilog*, sem os multiplexadores e células lógicas que compõem o circuito *VRC* estarem envolvidos. Note também que, na coluna "VRC", coluna "1 F", todos os circuitos estão marcados como corretos



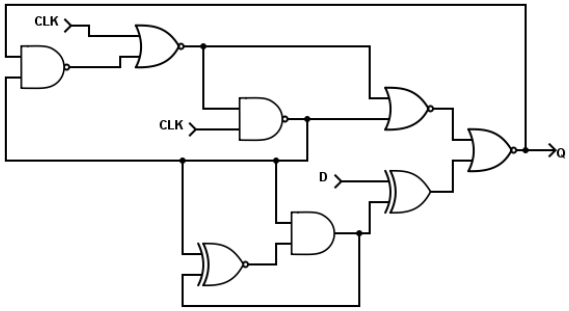
(a) Simulação pela *Cyclone IV* na Figura II.30 e pela *Cyclone II* na Figura III.30.



(b) Simulação pela *Cyclone IV* na Figura II.31 e pela *Cyclone II* na Figura III.31.



(c) Simulação pela *Cyclone IV* na Figura II.32 e pela *Cyclone II* na Figura III.32.



(d) Simulação pela *Cyclone IV* na Figura II.33 e pela *Cyclone II* na Figura III.33.

Figura 4.33: Circuitos para a função paridade-2 encontrados pela evolução.

pois representam o ambiente de avaliação original, envolvendo a placa de desenvolvimento *DE1-SoC*.

As simulações, como aquela vista na Figura 4.3, são consideradas corretas se os sinais de saída dos circuitos corresponderem ao sinal *saida[0]*. A verificação física consiste em utilizar os dispositivos de entrada e saída físicos, como as chaves e os *LED*s, das placas de desenvolvimento para verificar que os comportamentos dos circuitos correspondem às sequências usadas em suas evoluções.

A Tabela 4.9 mostra ainda algumas características interessantes. Circuitos como os da *latch* D mostrado na Figura 4.4 e da *latch* de duas portas mostrado na Figura 4.18 se mostraram corretos em todos os ambientes testados e avaliados, sugerindo que seu projeto não depende de nenhum mecanismo específico da síntese do circuito, especificamente para o FPGA *Cyclone V*, utilizado para evolução. Em contrapartida, há circuitos como os da *latch* de duas portas mostrado na Figura 4.19b e do *flip-flop* D mostrado na Figura 4.32c em que o único ambiente em que o circuito se comporta corretamente é naquele da evolução original, indicando uma possível dependência com o circuito VRC e do dispositivo FPGA *Cyclone V* para seu funcionamento.

É importante notar que há diferentes motivos pelo qual um circuito pode estar marcado

Tabela 4.9: Sumário da corretude dos circuitos evoluídos nos diferentes FPGAs e configurações.

		VRC					Puro				
	Figura	70 M	70 F	115 M	115 F	1 F	70 M	70 F	115 M	115 F	1 F
SR	4.2										
D	4.4										
	4.6a										
	4.6b	✗			✗		✗	✗			
	4.6c										
XOR	4.9a	✗	✗		✗			✗	✗	✗	✗
	4.9b	✗	✗	✗	✗		✗	✗		✗	✗
	4.8										
	4.9c										
JK	4.12a	✗	✗	✗			✗	✗	✗	✗	✗
	4.11										
	4.12b		✗		✗		✗				✗
	4.12c	✗		✗				✗			✗
D mult.	4.15a	✗									✗
	4.15b										
	4.14										
	4.15c	✗	✗	✗				✗			
2 portas	4.18										
	4.19a	✗	✗	✗							
	4.19b	✗	✗	✗	✗		✗	✗	✗	✗	✗
	4.19c	✗		✗							
BILBO	4.23a	✗	✗	✗			✗	✗	✗	✗	✗
	4.23b							✗	✗		
	4.23c	✗		✗	✗				✗	✗	✗
	4.23d									✗	✗
Flip-flop D	4.32a	✗			✗		✗				✗
	4.32b	✗	✗	✗			✗	✗	✗		✗
	4.32c	✗	✗	✗	✗		✗	✗	✗	✗	✗
	4.32d	✗	✗	✗	✗					✗	
Paridade-2	4.33a	✗	✗	✗	✗		✗		✗		✗
	4.33b				✗		✗		✗		✗
	4.33c			✗	✗		✗	✗	✗	✗	
	4.33d	✗	✗		✗		✗	✗	✗	✗	✗

como incorreto. Um deles é a não correspondência entre a saída obtida e saída esperada, com um exemplo podendo ser visto na Figura II.20, onde ambos os sinais *fenotipo[0]* e *circuito[0]* diferem do sinal esperado *saida[0]*. Outro motivo é por sinais transientes que passam do limite de estabilização do circuito após a alteração da entrada, calculado como 100ns para os experimentos realizados, com um exemplo na Figura III.6 para o sinal *fenotipo[0]*. Circuitos que oscilam também são considerados incorretos, como pode ser visto na simulação apresentada na Figura III.14 no sinal *fenotipo[0]*.

4.3 Desempenho

4.3.1 Avaliação dos Indivíduos

Para a análise do desempenho da avaliação dos indivíduos, há dois casos a serem considerados: caso o indivíduo avaliado esteja errado ou caso ele esteja certo. Se o indivíduo estiver errado, a sequência será executada apenas uma vez. Se estiver certo, ela será executada N_R vezes.

Assim, o número de ciclos necessário para avaliar a sequência apenas uma vez pode ser calculado como

$$C_S = M \times N \tag{4.1}$$

onde M é o número de amostras por elemento da sequência e N o número de elementos na sequência. Para $M = 1000$, como realizado nos experimentos, a avaliação de um indivíduo cuja sequência possua tamanho $N = 22$, como no experimento do *flip-flop* D, leva $C_S = 1000 \times 22 = 22000$ ciclos. A uma frequência de 50MHz, isso corresponde a um tempo total de aproximadamente 440μs.

A possibilidade de executar o programa no mesmo ambiente em que os indivíduos são avaliados permite minimizar o tempo da comunicação entre o processador e o dispositivo. Uma estimativa de comunicação serial por *RS-232* com uma taxa de transferência de 115000 *bits* por segundo adiciona um tempo de cerca de 4ms apenas para a transmissão de um indivíduo, que possui um tamanho de 423 *bits* para os experimentos realizados. O resultado é um tempo de cerca de 4,4ms para a avaliação, ou seja, aproximadamente 10 vezes maior.

Outra alternativa que evita a necessidade de comunicação externa é a adoção de evolução extrínseca, isto é, utilização de simuladores de circuitos como *PSpice*. Um dos problemas dessa abordagem é o grande tempo necessário para a realização da simulação da avaliação de um indivíduo, necessitando cerca de 410ms a 670ms para uma simulação de 100μs utilizando o simulador *PSpice A/D* v.17.2, ou seja, um tempo cerca de 932 vezes maior em comparação ao tempo da avaliação realizada diretamente no FPGA.

4.3.2 Algoritmo Genético

O algoritmo genético, sendo executado no processador *ARM* embarcado no FPGA utilizado, possui um custo de tempo associado ao cálculo de suas operações que também deve ser computado para ter uma noção completa da eficiência da técnica desenvolvida. O programa é responsável pela inicialização, cálculo da adaptação do cromossomo para ser

enviado ao FPGA, operação de mutação, cálculo final do *fitness* e seleção de cromossomo para a próxima geração.

Para estimar o tempo que o algoritmo leva para executar, um temporizador foi utilizado para medir o tempo de execução de 1000 gerações, com parâmetros $\lambda = 4$, $N = 22$, taxa de mutação de 15% e com o número de células lógicas disponíveis para evolução variando entre os valores de 5 a 25. O programa, escrito em C++, é compilado com a diretiva de otimização *-O3* para o cálculo do tempo necessário para sua execução. Esses dados estão disponíveis na Tabela 4.10. Dos tempos obtidos é possível retirar o tempo utilizado para a avaliação dos indivíduos, como calculado anteriormente. Como cada geração necessita a avaliação de 5 indivíduos, para 1000 gerações temos um tempo total de avaliação $t_a = 5 \times 1000 \times (4,4 \times 10^{-4}) = 2{,}2$s.

Tabela 4.10: Tempos de execução para a evolução de 1000 gerações.

Células lógicas	Tempo (s)
5	7,129
6	7,411
7	8,431
8	8,842
9	10,078
10	10,631
11	12,096
12	12,698
13	13,302
14	15,197
15	15,882
16	17,872
17	18,801
18	20,950
19	21,897
20	24,245
21	25,661
22	27,910
23	28,949
24	30,173
25	33,059

Da tabela, é possível concluir que o desempenho do programa depende do número de células lógicas disponíveis para a evolução no VRC. Retirando-se o tempo de avaliação total, $t_a = 2,2s$, para cada conjunto de execuções, observa-se que o tempo total é tomado pela execução do programa à medida que o número de células lógicas disponíveis aumenta. Como o programa foi desenvolvido tendo corretude como prioridade, sem con-

siderar eficiência, há ainda espaço considerável para ganhos de desempenho em tempo de execução.

O gráfico visto na Figura 4.34 mostra um ajuste de curva dos dados da Tabela 4.10. Observa-se que esses dados correspondem a uma curva quadrática, com sua equação presente na parte de cima da figura. Isso permite que possamos estimar o tempo tomado por uma evolução com um número arbitrário n de células lógicas.

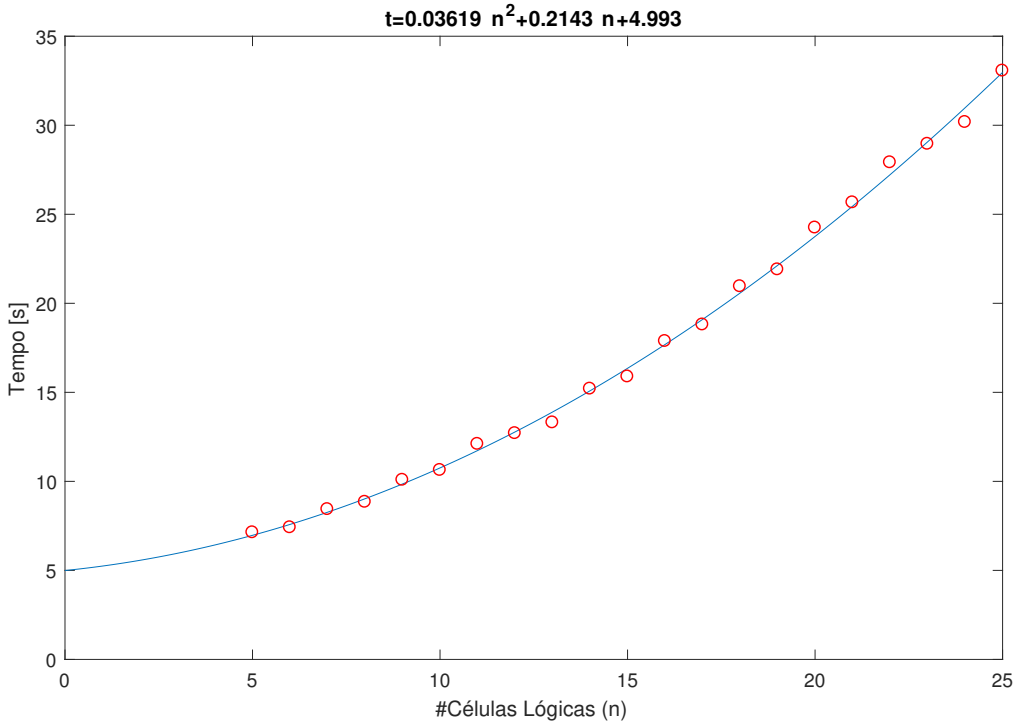


Figura 4.34: Tempo de evolução de 1000 gerações × número de células lógicas.

Para fins de comparação entre um tempo real e o cálculo de seu tempo baseado no número de gerações necessário para encontrar uma solução, usaremos o exemplo de uma evolução do circuito paridade-2. Seu gráfico de melhor *fitness* ao longo das gerações pode ser visto na Figura 4.35, mostrando a evolução de 200000 gerações para 6 células lógicas, em que uma solução não foi encontrada, 200000 gerações para 7, em que uma solução também não foi encontrada, e para 8 células lógicas, cuja solução foi encontrada na geração 75309.

Usando a Tabela 4.10 como referência, temos os tempos médios de execução $t_{le6} =$ 7,411s para 6 células lógicas, $t_{le7} =$ 8,431s para 7 e $t_{le8} =$ 8,842 para 8 em 1000 gerações. Assim, o cálculo do tempo para 200000 gerações de 6 células lógicas pode ser calculado como $t_6 = t_{le6} \times \frac{200000}{1000} =$ 1482,2s, para 7 elementos, $t_7 = t_{le7} \times \frac{200000}{1000} =$ 1686,2s e, para 8 elementos, $t_8 =$ 665,9. Somando-se os três, temos um tempo total de $t_{al} = t_6 + t_7 + t_8 = 1482,2 + 1686,2 + 665,9 =$ 3834,3s, ou seja, cerca de 63 minutos e 54 segundos.

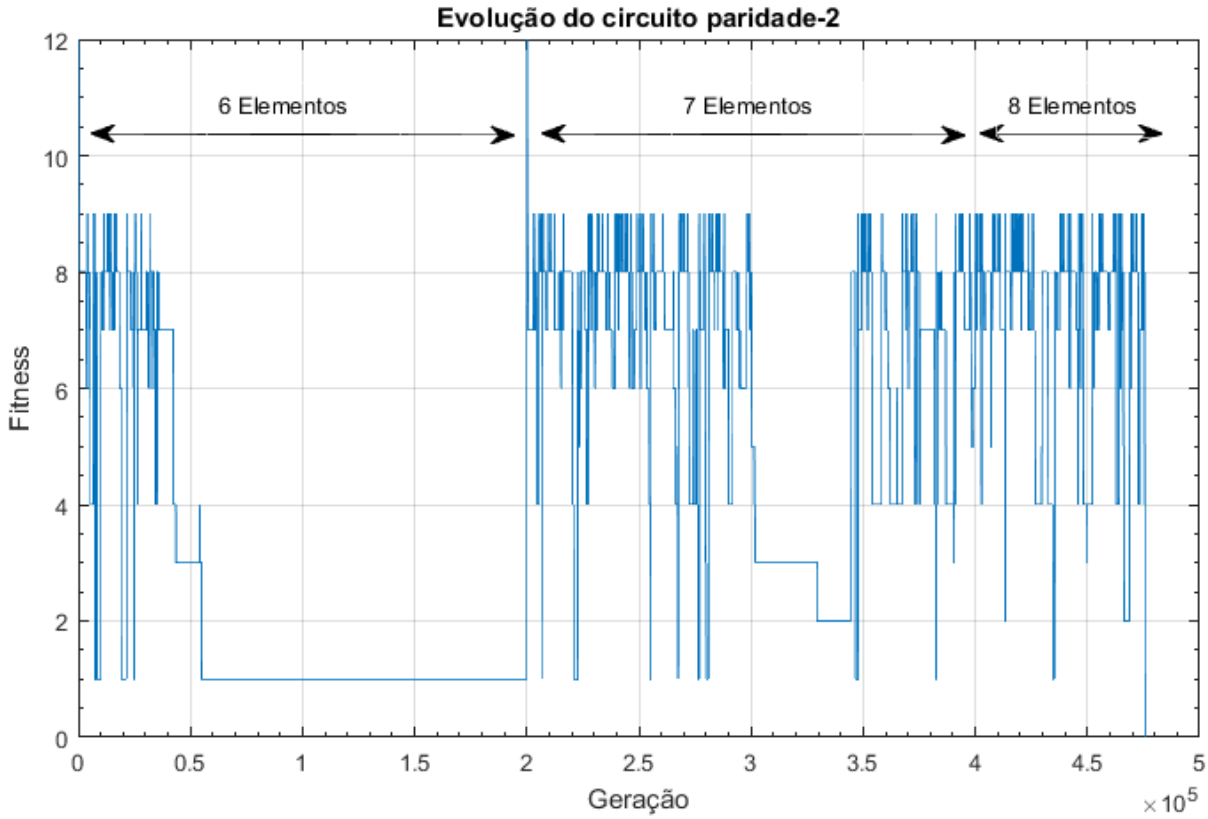


Figura 4.35: Gráfico de *fitness* para uma evolução do circuito paridade-2, com 6, 7 e 8 células lógicas.

Um fator constante de tempo é adicionado ao final do algoritmo, ao encontrar um indivíduo correto, em que o número de ciclos utilizado para sua avaliação pode ser calculado como $N_R \times C_S$. Para $N_R = 30000$, o resultado é $30000 \times 22000 = 660 \times 10^6$ ciclos, ou seja, cerca de 13,2 segundos para uma frequência de 50MHz. Adicionando esse tempo a t_{al}, temos o tempo final $t_f = t_{al} + \frac{N_R \times C_S}{f_a} = 3847,5$, ou cerca de 64 minutos e 7 segundos.

O tempo real medido para esta evolução foi de 64 minutos e 7 segundos, mostrando então que o cálculo realizado corresponde aos tempos observados.

O tempo de avaliação total utilizado pelo algoritmo pode ser calculado utilizando como base o tempo de 440μs de avaliação para um indivíduo calculado anteriormente. Com isso, tendo $t_{seq} = 4,4 \times 10^{-4}$, $\lambda = 4$ e $n_{gen} = 475309$ sendo o número total de gerações, o tempo de avaliação total pode ser calculado como $t_{av} = (1+\lambda) \times t_{seq} \times n_{gen} = 5 \times (4,4 \times 10^{-4}) \times 475309 = 960$,1s, ou cerca de 16 minutos. Como ilustração, para $t_{seq} = 4,4 \times 10^{-3}$, como no caso da avaliação serial por *RS-232*, o tempo de avaliação total seria $t_{av} = 10456$,8s, ou cerca de 174 minutos e 16 segundos. Caso fosse feita por evolução extrínseca, $t_{seq} = 4,1 \times 10^{-1}$, teríamos um tempo de avaliação total de $t_{av} = 974383$,5s, ou cerca de 11 dias! Isso mostra o quão importante é ter uma avaliação eficiente para algoritmos genéticos que sejam executados por um número extenso de gerações. Como

a avaliação intrínseca, porém, é realizada no ambiente limitado de um FPGA, é possível que em algum momento, devido a recursos computacionais mais avançados e um possível uso de paralelismo, a evolução extrínseca por simulação se torne mais rápida que aquela realizada neste trabalho.

4.4 Trabalhos relacionados

Como mencionado na revisão dos trabalhos relacionados a evolução de circuitos sequenciais, a quase totalidade das pesquisas buscam abstrair o funcionamento de uma máquina de estados, retirando o armazenamento da evolução e trabalhando apenas com as funções de transição e saída. Por um lado, esse tipo de abordagem permite que circuitos mais complexos consigam ser evoluídos, mas, por outro lado isso representa uma restrição do espaço de soluções, possivelmente excluindo soluções que possam ser interessantes. Por causa disto, não há muita sobreposição de resultados entre o que é tratado neste trabalho e no que é visto em trabalhos relacionados.

Um dos estudos [46], utilizando a abordagem descrita, realizou a evolução de um circuito contador de 2 *bits*, um detector de sequências e um detector 1010. Outro estudo [45] buscou tentar a evolução de diferentes tipos de detectores de sequência, mais especificamente um detector 1010 e um detector de sequências com 6 estados. Um outro estudo [49] buscou aprimorar a técnica utilizada em estudos anteriores, resultando em uma evolução de um detector de sequências. Devido à abordagem de complexidade incremental adotada por este trabalho e à limitação de tempo para concluí-lo, a evolução de circuitos similares aos reportados não puderam ser testados.

4.5 Considerações finais

O mecanismo básico explorado por este trabalho para a evolução de circuitos sequenciais, a presença de *loop combinacionais*, em geral é algo indesejável em circuitos digitais. Assim, diversos problemas foram encontrados no desenvolvimento realizado.

A maior parte dos problemas encontrados é relacionado à descrição do circuito VRC e sua síntese para ser utilizado nos FPGAs utilizando a ferramenta de síntese *Quartus Prime*.

Um dos primeiros problemas que tivemos foi o tratamento que o compilador que realiza a síntese do circuito dá aos *loops combinacionais* explícitos. Há a possibilidade de que os circuitos, nessa tradução para os elementos lógicos do FPGA, percam a semântica esperada. Assim, circuitos como a *latch* SR não funcionam corretamente. Para amenizar este problemas, o *software* disponibiliza *LCELL*, que força a alocação de uma célula lógica

e permite que a semântica esperada seja preservada. Um fator que contribuiu para que esse problema persistisse por um tempo considerável foi o tempo de compilação do circuito VRC no *Quartus Prime* versão 16.1 demandar cerca de 20 minutos a 1 hora em uma máquina com processador *Core i5* modelo *4210M* de 2.6GHz e 8GB de memória *RAM*.

A falta de documentação relacionada ao processador embarcado e como usá-lo para criar uma comunicação com um circuito sintetizado também nos custou bastante tempo.

Ao tentarmos obter resultados de diferentes fontes possíveis, como simulações, outras barreiras foram encontradas:

1. Diferentes versões do *Quartus Prime* dão suporte a diferentes subconjuntos de FPGAs, forçando-nos a usar diferentes versões. A versão 13.1 do *Quartus-II* é a última a dar suporte ao dispositivo FPGA *Cyclone II*, no entanto esta versão não fornece suporte à *Cyclone V*.

2. A falta de suporte de todas as versões (atualmente está na versão 18.0) às simulações temporais, capazes de simular comportamentos cíclicos, para o FPGA *Cyclone V*, usado nos experimentos;

3. Mudanças não relacionadas no circuito sintetizado podem afetar o resultado das simulações.

SystemVerilog é uma linguagem de descrição de circuitos um tanto limitada que não possui uma análise estática forte quando comparada a linguagens tradicionais de programação. Em conjunto com o sintetizador utilizado pelo *Quartus Prime*, projetos produzidos com base nela são, portanto, propícios a erros triviais que afetam o tempo de desenvolvimento. Por exemplo, erros relacionados a diferença de tamanhos entre parâmetros de módulos e seus locais de uso são comuns, fazendo o circuito ter um comportamento incorreto.

Em relação ao programa que executa o algoritmo genético, uma das características mais limitantes é o considerável atraso da versão do compilador *g++*, 4.x, recomendado pelo *Quartus Prime*, não permitindo que usássemos construções modernas disponíveis em versões mais recentes, tais como aquelas presentes no padrão *C++14*. Outro problema relacionado a isso é a geração de códigos de comportamentos diferentes em níveis diferentes de otimização. O compilador parece incapaz de analisar que parte do código pode ser alterada fora dele e aplica otimizações que não fazem sentido, concluindo que, por exemplo, um laço será sempre infinito erroneamente.

Esses fatores contribuíram para, infelizmente, retardar nosso progresso. Apesar disso, acreditamos ter conseguido resultados promissores e que merecem mais atenção futuramente.

Capítulo 5

Conclusão

Este trabalho procurou responder questões sobre a possibilidade de utilização de algoritmos genéticos para realizar a tarefa de projetar circuitos digitais sequenciais. Questões como "é possível utilizar algoritmos genéticos para projetar circuitos digitais sequenciais intrinsecamente?" e "esses projetos possuem alguma característica destoante de projetos humanos, sendo, possivelmente mais eficientes?" foram os principais motivadores para a realização deste trabalho.

Primeiramente, foi conduzido um estudo detalhado sobre como álgebra booleana e seus métodos clássicos de otimização são formulados. Em seguida, estudou-se uma plataforma reconfigurável comum para prototipação e verificação de circuitos digitais, o FPGA. Conceitos sobre algoritmos genéticos foram apresentados, mostrando classificações, algoritmos comumente utilizados e os métodos evolutivos principais vistos neste trabalho. Em seguida, todas essas ideias foram reunidas para definir conceitos relacionados à área de *Hardware Evolutivo*. Ao final, uma breve discussão sobre o estado da arte presente nesta área, apontando também a pouca atenção dada a circuitos sequenciais em comparação a combinacionais.

Foi proposta uma abordagem nova para a evolução intrínseca de circuitos sequenciais. Ela envolve a utilização de um FPGA (*Cyclone V*) com um processador ARM A9 *dual core* embarcado como ambiente de avaliação, permitindo que o algoritmo como um todo seja executado em apenas um dispositivo, minimizando custos de comunicação. Dessa maneira, utilizando-se dessa característica eficiente, esse tipo de evolução se torna viável. Uma estratégia baseada na amostragem do sinal de saída é utilizada para identificar problemas de oscilações, possíveis em circuitos que permitem *feedbacks* explícitos. Deste modo torna-se possível obter soluções que estão fora do alcance de métodos convencionais.

Os problemas escolhidos para os experimentos foram, em ordem de complexidade, a síntese dos circuitos assíncronos básicos *latches* SR, D, *XOR*, JK, D multiplexada, de duas portas e BILBO, e a síntese dos circuitos síncronos *flip-flop* D e paridade-2. Para se ter

maior clareza quanto às soluções obtidas, elas também foram testadas e simuladas nas famílias de FPGAs *Cyclone II* e *Cyclone IV*.

A aplicação do método para esses circuitos mostra que não só é possível encontrar circuitos com soluções altamente complexas, que se aproveitam de diversas características do circuito, como tempos de propagação, mas também circuitos mais eficientes do que atualmente conhecidos. Como circuitos evoluídos não utilizam abstrações como os humanos em seus projetos, eles estão livres para explorar fatores como a combinação de entradas proibidas da *latch* SR em suas construções internas para encontrar soluções fora do escopo dessas abstrações. Isso permitiu que encontrássemos circuitos como uma *latch* D que precisa de 2 transistores a menos do que a referência utilizada ou um *flip-flop* D que também economiza 2 transistores em relação à referência mais eficiente.

A possibilidade de algoritmos genéticos poderem ser implementados utilizando o ambiente intrínseco de um FPGA abre várias portas para que outros trabalhos consigam usufruir disto. Para este trabalho, um dos grandes problemas que não possibilitou o teste da técnica para circuitos mais complexos é o projeto das sequências de avaliação, sendo feitas manualmente quando não disponíveis de outros trabalhos. Uma limitação deste trabalho é também termos realizados experimentos sem variar o número de entradas das células lógicas, mantendo-nos em apenas 2. Um próximo passo poderia envolver mais experimentos variando esse parâmetro, possivelmente resultando em circuitos mais compactos do que os que foram aqui encontrados. Um outro possível futuro estudo pode envolver o desenvolvimento de sequências automaticamente a partir da definição de uma máquina de estados, não só permitindo seu uso para circuitos maiores, mas também garantindo sequências sem erros. Com isso, será possível a análise da viabilidade de evolução de circuitos mais complexos e uma possível revisão de outras técnicas evolutivas como alternativas à estratégia $(1 + \lambda)$.

O estudo realizado neste trabalho resultou em uma publicação no *12th International Symposium on Applied Reconfigurable Computing* (ARC 2016) [44] (Qualis B1) com os resultados parciais dessa proposta. Foram realizadas também submissões para a conferência *IEEE Congress on Evolutionary Computation* (CEC 2018) e para *31st Symposium on Integrated Circuits and Systems Design* (SBCCI 2018).

Referências

[1] Altera. De1 development and education board user manual. `https://www.altera.com/content/dam/altera-www/global/en_US/portal/dsn/42/doc-us-dsnbk-42-4904342209-de1-usermanual.pdf`, 2013. Acessado: 09-06-2018. xi, 48

[2] Charles Darwin. On the origin of species. *Murray, London*, page 360, 1859. 1, 26

[3] Philip Garcia, Katherine Compton, Michael Schulte, Emily Blem, e Wenyin Fu. An overview of reconfigurable hardware in embedded systems. *EURASIP Journal on Embedded Systems*, 2006. 1

[4] Mitchell Melanie. *An Introduction to Genetic Algorithms*. Bradford, 5th edition, 1999. 2, 26

[5] Victor P. Nelson, H. Troy Nagle, J. David Irwin, e Bill D. Carroll. *Digital Logic Circuit Analysis & Design*. Prentice-Hall, Inc., 1st edition, 1995. 4, 10, 11, 62

[6] Randall Hyde. *The Art of Assembly Language*. no starch press, 2001. 4

[7] Anant Agarwal e Jeffrey H. Lang. *Foundations of Analog and Digital Electronic Circuits*. Denise E. M. Penrose, 1st edition, 2005. 5, 9, 11

[8] Mark Balch. *Complete digital design: a comprehensive guide to digital electronics and computer system architecture*. McGraw-Hill, Inc., 2003. 10, 20

[9] M. Morris Mano. *Digital Logic and Computer Design*. Prentice-Hall, 1st edition, 2006. 10, 12, 13

[10] Venu G. Gudise e Ganesh K. Venayagamoorthy. Evolving digital circuits using particle swarm. *Neural Networks, 2003. Proceedings of the International Joint Conference on*, 2003. 11

[11] David Tarnoff. *Computer Organization and Design Fundamentals*. Lulu, 2007. 15, 16, 17

[12] Ali Belgasem. *Evolutionary Algorithms for Synthesis and Optimization of Sequential Logic Circuits*. PhD thesis, Edinburgh Napier University, 2003. 18, 19, 21, 22, 34, 36

[13] MJ Avedillo, JM Quintana, e JL Huertas. Smas: A program for the concurrent state reduction and state assignment of finite state machines. In *Circuits and Systems, 1991., IEEE International Symposium on*, pages 1781–1784. IEEE, 1991. 22

[14] José Nelson Amaral, Kagan Tumer, e Joydeep Ghosh. Designing genetic algorithms for the state assignment problem. *Systems, Man and Cybernetics, IEEE Transactions on*, 25(4):687–694, 1995. 22

[15] Pong P. Chu. *FPGA Prototyping by Verilog Examples*. Wiley, 3rd edition, 2008. 22, 25

[16] Adrian Thompson. *Hardware Evolution*. Springer, 1998. 23, 35, 36, 39, 55

[17] Jim Torresen. An evolvable hardware tutorial. In *Field Programmable Logic and Application*, pages 821–830. Springer, 2004. 23

[18] Altera. Cyclone ii device handbook, volume 1. `https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc2/cyc2_cii51002.pdf`, 2007. Acessado: 13-04-2018. 23

[19] Altera. Cyclone iv device handbook, volume 1. `https://www.altera.com/en_US/pdfs/literature/hb/cyclone-iv/cyclone4-handbook.pdf`, 2013. Acessado: 18-04-2018. 24

[20] Altera. Cyclone v device handbook. `https://www.altera.com/en_US/pdfs/literature/hb/cyclone-v/cv_5v2.pdf`, 2018. Acessado: 11-07-2018. 24

[21] Uwe Meyer-Baese. *Digital Signal Processing with Field Programmable Gate Arrays*. Springer, 3rd edition, 2007. 24

[22] Brad L. Miller, Brad L. Miller, David E. Goldberg, e David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995. 29

[23] Huayang Xie e Mengjie Zhang. *Tuning Selection Pressure in Tournament Selection*. School of Engineering and Computer Science, Victoria University of Wellington, 2009. 30

[24] James Hilder, James Alfred Walker, e Andy Tyrrell. Use of a multi-objective fitness function to improve cartesian genetic programming circuits. In *Adaptive Hardware and Systems (AHS), 2010 NASA/ESA Conference on*, pages 179–185. IEEE, 2010. 30, 38

[25] James Alfred Walker, Katharina Völk, Stephen L Smith, e Julian Francis Miller. Parallel evolution using multi-chromosome cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 10(4):417–445, 2009. 30, 34, 38

[26] Julian F Miller, Dominic Job, e Vesselin K Vassilev. Principles in the evolutionary design of digital circuits—part i. *Genetic programming and evolvable machines*, 1(1-2):7–35, 2000. 30, 31

[27] H. Beyer. Towards a theory of 'evolution strategies'. some asymptotical results from the (1,+lambda)-theory. *Evolutionary Computation*, 1:165–188, 1993. 31

[28] Julian F Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, 1999. 31, 33, 38, 42

[29] Julian F Miller. *Cartesian Genetic Programming*. Springer, 1st edition, 2011. 31, 32

[30] Julian F Miller e Stephen L Smith. Redundancy and computational efficiency in cartesian genetic programming. *Evolutionary Computation, IEEE Transactions on*, 10(2):167–174, 2006. 34

[31] Vesselin K Vassilev e Julian F Miller. The advantages of landscape neutrality in digital circuit evolution. In *Evolvable systems: from biology to hardware*, pages 252–263. Springer, 2000. 34

[32] John R Koza. *Genetic programming II, automatic discovery of reusable subprograms*. MIT Press, Cambridge, MA, 1992. 34

[33] Tetsuya Higuchi, Masaya Iwata, Isamu Kajitani, Hitoshi Iba, Yuji Hirao, Tatsumi Furuya, e Bernard Manderick. Evolvable hardware and its application to pattern recognition and fault-tolerant systems. In *Towards evolvable hardware*, pages 118–135. Springer, 1996. 34

[34] Tetsuya Higuchi, Masaya Iwata, Didier Keymeulen, Hidenori Sakanashi, Masahiro Murakawa, Isamu Kajitani, Eiichi Takahashi, Kenji Toda, N Salami, Nobuki Kajihara, et al. Real-world applications of analog and digital evolvable hardware. *IEEE transactions on evolutionary computation*, 3(3):220–235, 1999. 34

[35] GW Timothy Gordon e J Peter Bentley. On evolvable hardware. In *Soft Computing in Industrial Electronics*, pages 279–323. Springer, 2002. 34

[36] RO Canham e AM Tyrrell. Evolved fault tolerance in evolvable hardware. In *Proceedings of IEEE Congress on Evolutionary Computation*. Citeseer, 2002. 35

[37] Jim Torresen. Evolvable hardware-a short introduction. In *Proceedings of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems*, volume 1, pages 674–677, 1997. 37

[38] Xin Yao e Tetsuya Higuchi. Promises and challenges of evolvable hardware. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 29(1):87–97, 1999. 37

[39] P C Haddow e Andy Tyrrell. Challenges of evolvable hardware: Past, present and the path to a promising future. *Journal of Genetic Programming and Evolvable Machine*, 12(3):183–215, 9 2011. 38

[40] P Rustem. Toward ehw 2.0–some ideas in hdl-based evolution of digital circuits. *Glob J Tech Opt*, 6(174):2, 2015. 38

[41] Morten Hartmann, Per Kristian Lehre, e Pauline C Haddow. Evolved digital circuits and genome complexity. In *Evolvable Hardware, 2005. Proceedings. 2005 NASA/DoD Conference on*, pages 79–86. IEEE, 2005. 38

[42] Vitor Coimbra de Oliveira. Projeto e otimização de circuitos digitais por técnicas de evolução artificial. `http://bdm.unb.br/handle/10483/11045`, 2015. 38

[43] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, e TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002. 38

[44] Vitor Coimbra e Marcus Vinicius Lamar. Design and optimization of digital circuits by artificial evolution using hybrid multi chromosome cartesian genetic programming. In *International Symposium on Applied Reconfigurable Computing*, pages 195–206. Springer, 2016. 39, 49, 89

[45] Parisa Soleimani, Reza Sabbaghi-Nadooshan, Sattar Mirzakuchaki, e Mahdi Bagheri. Using genetic algorithm in the evolutionary design of sequential logic circuits. *arXiv preprint arXiv:1110.1038*, 2011. 39, 42, 86

[46] Belgasem Ali, AEA Almaini, e Tatiana Kalganova. Evolutionary algorithms and theirs use in the design of sequential logic circuits. *Genetic Programming and Evolvable Machines*, 5(1):11–29, 2004. 39, 40, 42, 86

[47] Lukáš Sekanina. Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *International Conference on Evolvable Systems*, pages 186–197. Springer, 2003. 51

[48] Samy Makar e Edward McCluskey. Using checking experiments to test two-state latches. `http://www-crc.stanford.edu/crc_papers/CRC-TR-94-11.pdf`, 1994. 56, 59, 61, 64, 68, 69

[49] P Soleimani, S Mirzakuchaki, R Sabbaghi-Nadooshan, e M Bagheri. Improvement of optimization in design of synchronous sequential circuits by using evolvable hardware. *International Journal of Computer Theory and Engineering*, 4(5):812, 2012. 86

Anexo I

Sequências de Evolução

Tabela I.1: Sequência utilizada para a evolução da *latch* SR.

RS	Q
10	0
00	0
01	1
00	1
10	0
00	0

Tabela I.2: Sequência utilizada para a evolução da *latch* D.

DC	Q
11	1
01	0
00	0
10	0
11	1
10	1
00	1

Tabela I.3: Sequência utilizada para a evolução da *latch XOR*.

CSD	Q
110	1
100	0
000	0
010	0
110	1
010	1
011	1
111	0
011	0
001	0
101	1
001	1
000	1

Tabela I.4: Sequência utilizada para a evolução da *latch* JK.

JK	Q
01	0
00	0
10	1
00	1
01	0
00	0
01	0
11	1
10	1
11	0
01	0
11	1

Tabela I.5: Sequência utilizada para a evolução da *latch* D multiplexada.

CTSD	Q
1101	0
1001	1
0001	1
0000	1
1000	0
0000	0
0001	0
0011	0
1011	1
0011	1
0010	1
1010	0
0010	0
0110	0
1110	1
0110	1
0100	1
1100	0
0100	0
0110	0
0111	0
1111	1
0111	1
0101	1
1101	0
0101	0

Tabela I.6: Sequência utilizada para a evolução da *latch* de duas portas.

$C_2D_2C_1D_1$	Q
0011	1
0010	0
0000	0
0100	0
1100	1
0100	1
0000	1
1000	0
0000	0
0001	0
0011	1
0001	1
1001	0
0001	0
0101	0
1101	1
0101	1
0100	1
0110	0
0100	0
0101	0
0111	1
0101	1

Tabela I.7: Sequência utilizada para a evolução da *latch* BILBO.

DSB_2B_1C	Q
00001	1
00011	0
00010	0
00000	0
00001	1
00000	1
01000	1
01001	0
01000	0
00000	0
10000	0
10010	1
10011	0
10010	0
10110	0
10111	1
10110	1
00110	1
00111	0
00110	0
00100	0
00101	1
00100	1
01100	1
01101	0
01100	0
11100	0
11101	1
11100	1
11000	1
11001	0
11000	0
10000	0
10001	1
10000	1
10100	1
10101	0
10100	0
10110	0
10111	1
11111	1
11011	0
11010	0
11110	0
11111	1
11110	1
01110	1
01111	0
01110	0
11110	0
11111	1
11110	1
11010	1
10010	1
00010	1
01010	1
01011	0
01010	0

Tabela I.8: Sequência utilizada para a evolução do *flip-flop* D.

D CLK	Q
00	X
01	0
11	0
01	0
11	0
10	0
00	0
10	0
00	0
01	0
00	0
10	0
11	1
01	1
11	1
10	1
00	1
10	1
11	1
01	1
00	1
01	0

Tabela I.9: Sequência utilizada para a evolução do circuito de paridade-2.

A CLK	Q
00	X
01	0
11	1
01	0
11	1
10	1
00	0
10	1
00	0
01	0
00	0
10	1
11	0
01	1
11	0
10	0
00	1
10	0
11	0
01	1
00	1
01	0

Anexo II

Simulações dos Resultados - Cyclone IV

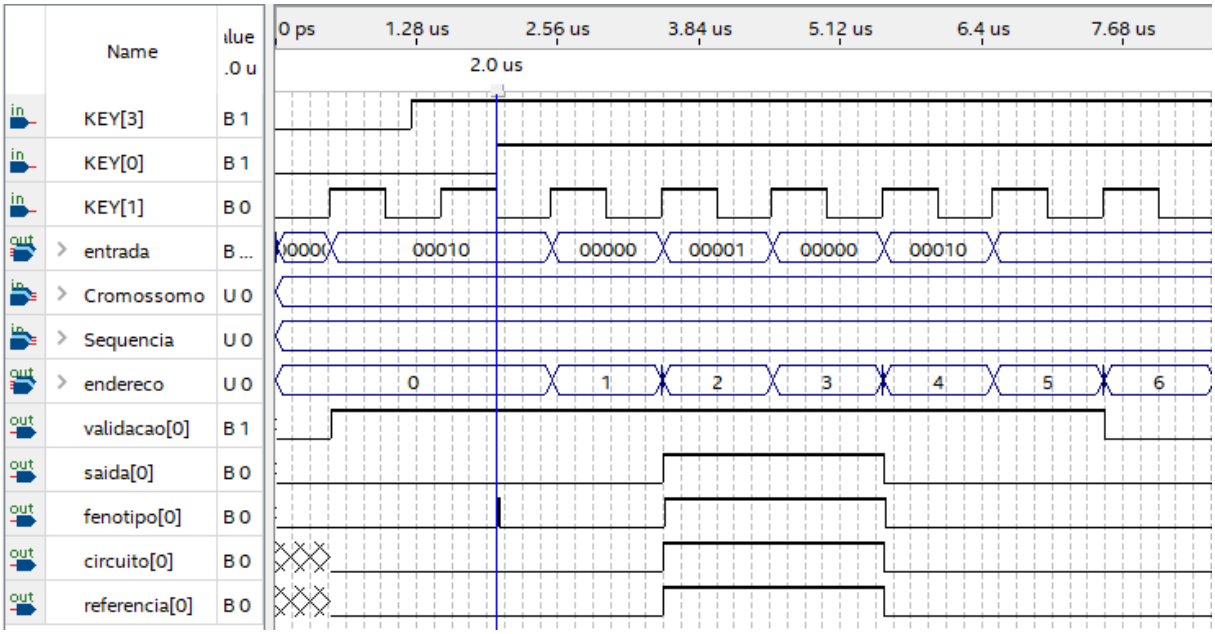


Figura II.1: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* SR da Figura 4.2.

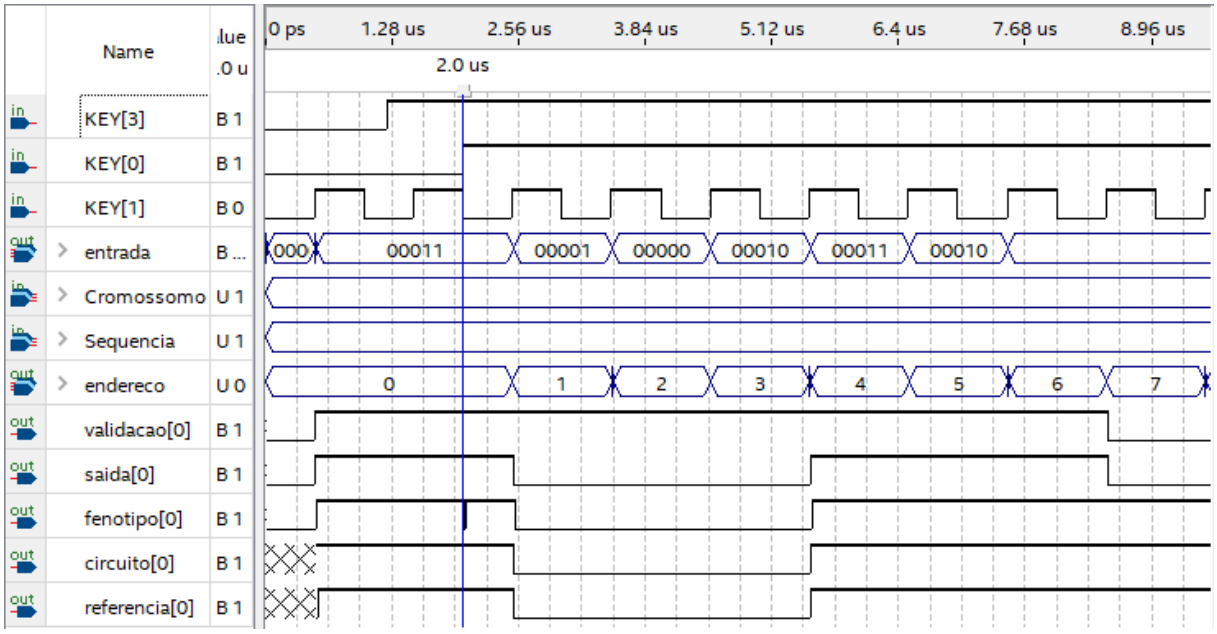


Figura II.2: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D da Figura 4.4.

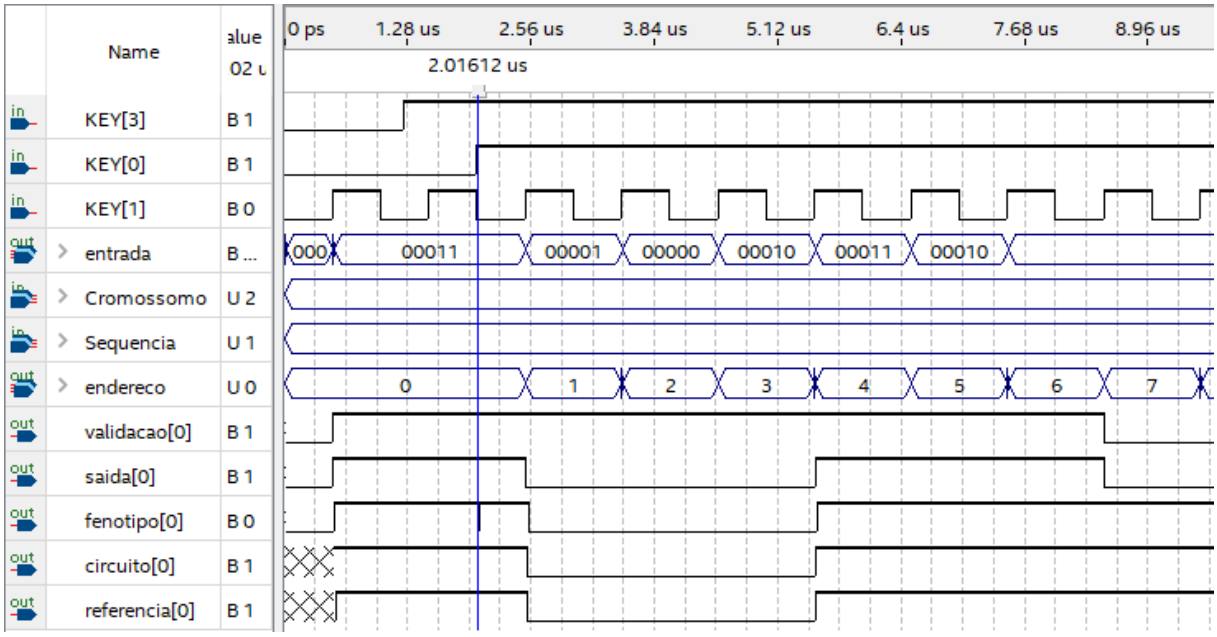


Figura II.3: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D da Figura 4.6a.

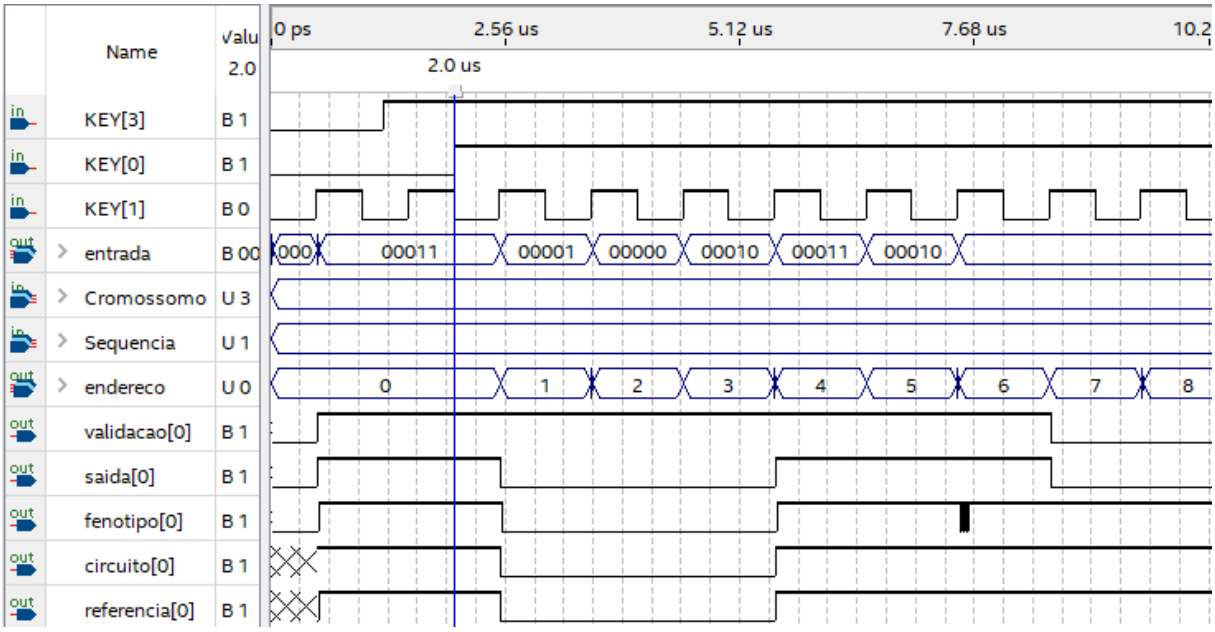


Figura II.4: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D da Figura 4.6b.

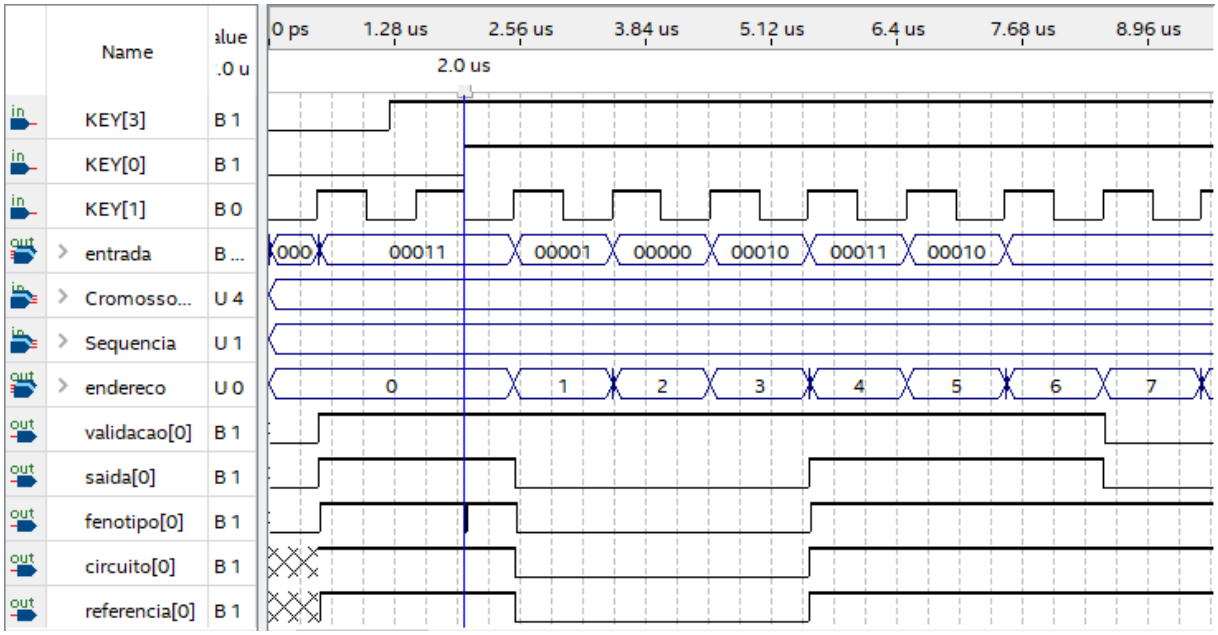


Figura II.5: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D da Figura 4.6c.

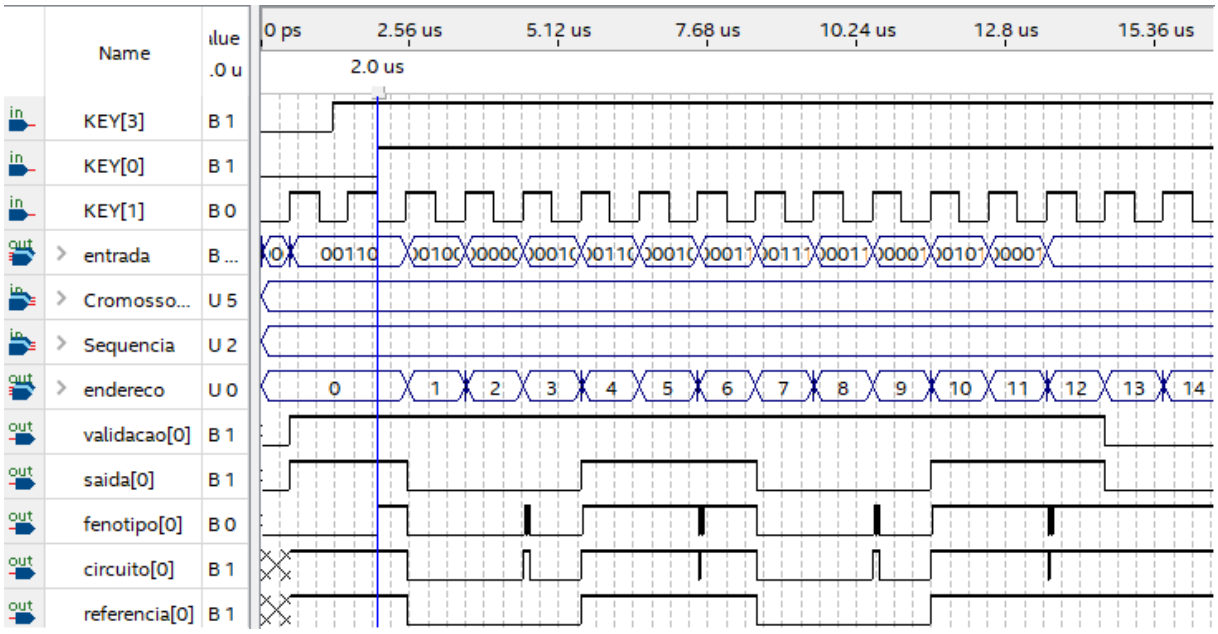


Figura II.6: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch XOR* da Figura 4.9a.

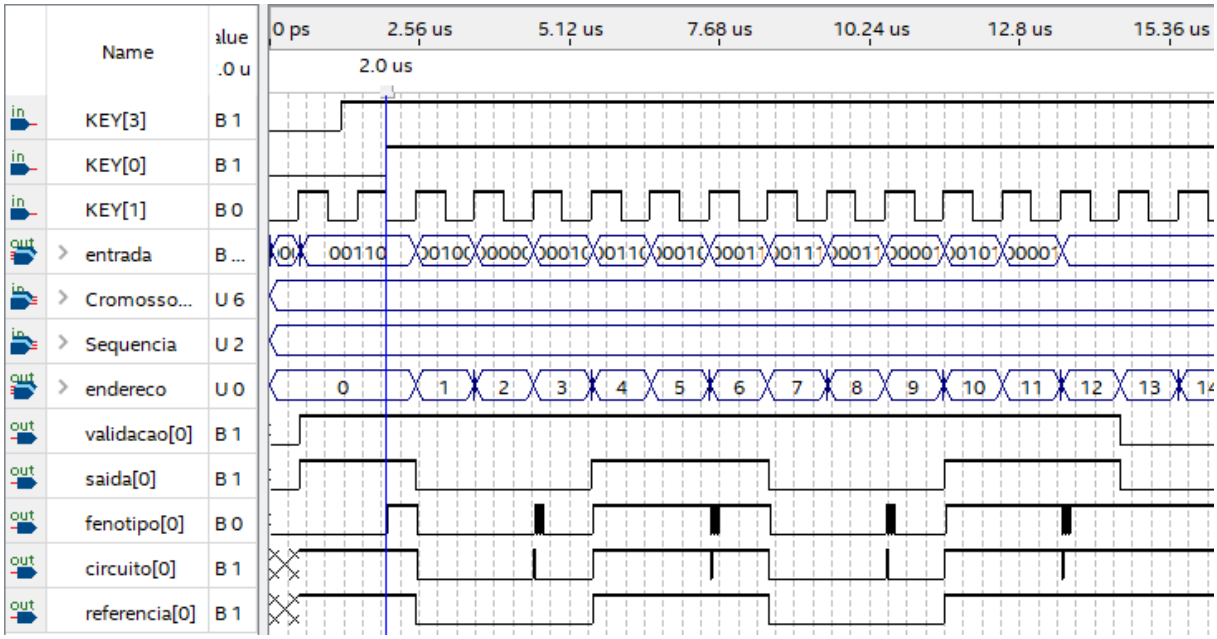


Figura II.7: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch XOR* da Figura 4.9b.

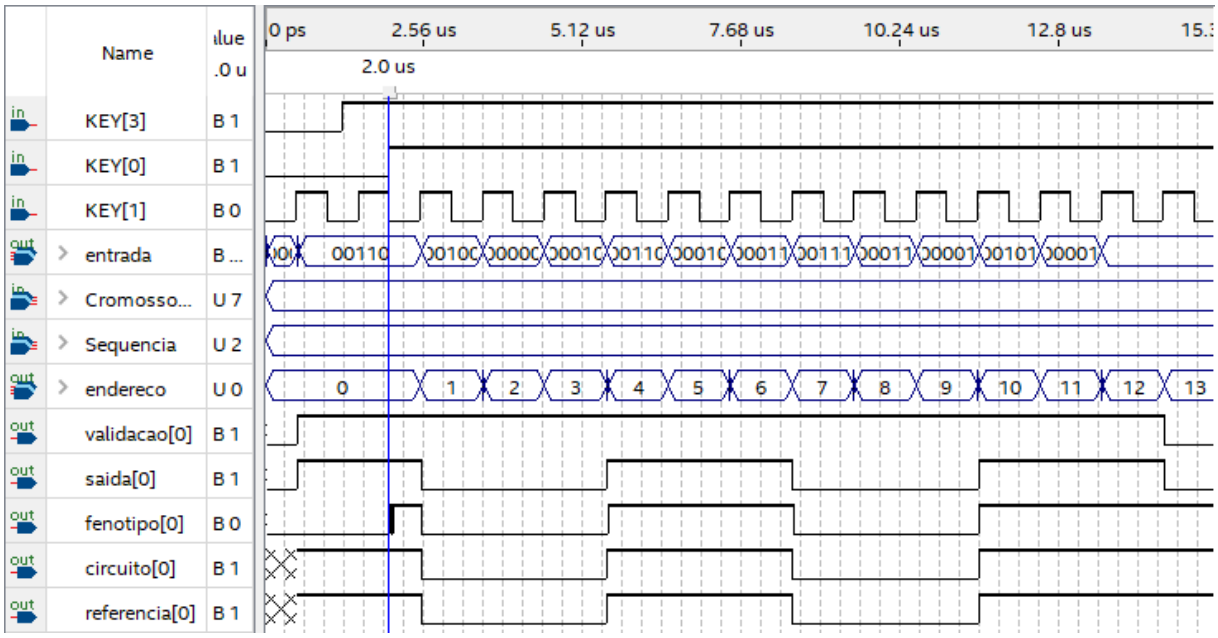


Figura II.8: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch XOR* da Figura 4.8.

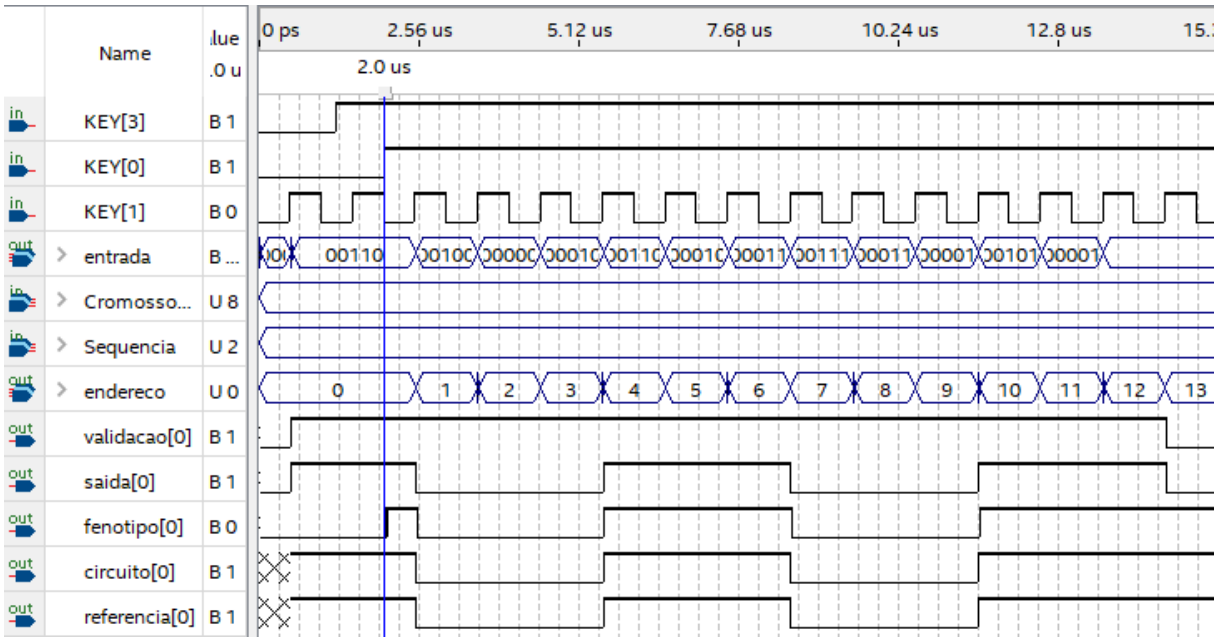


Figura II.9: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch XOR* da Figura 4.9c.

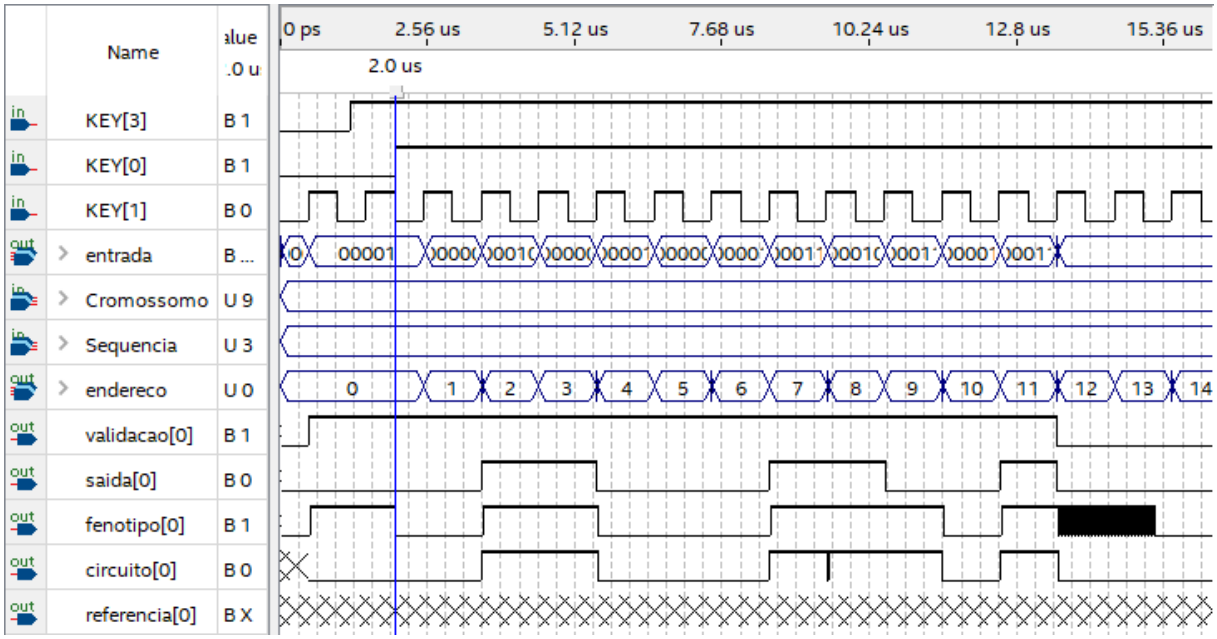


Figura II.10: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* JK da Figura 4.12a.

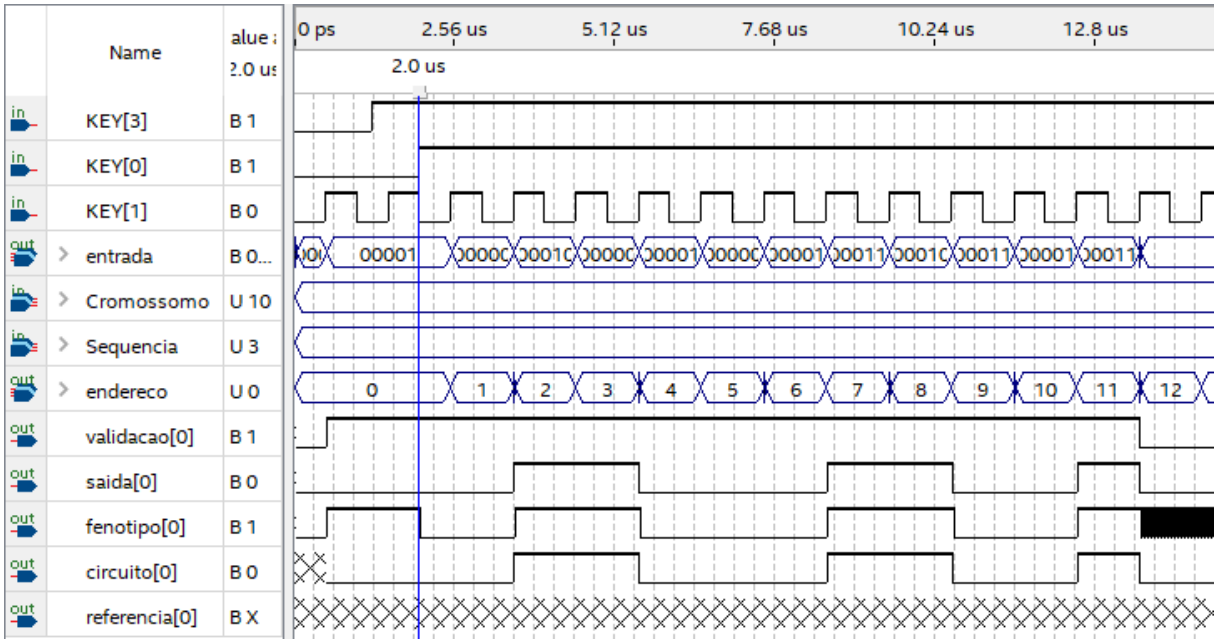


Figura II.11: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* JK da Figura 4.11.

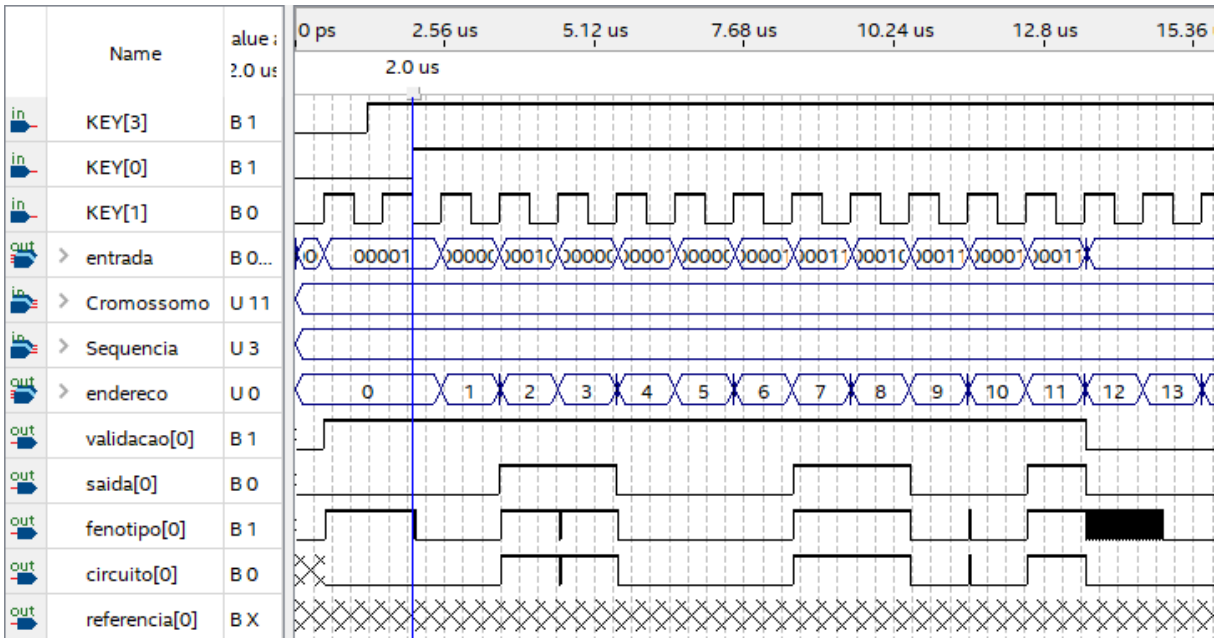


Figura II.12: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* JK da Figura 4.12b.

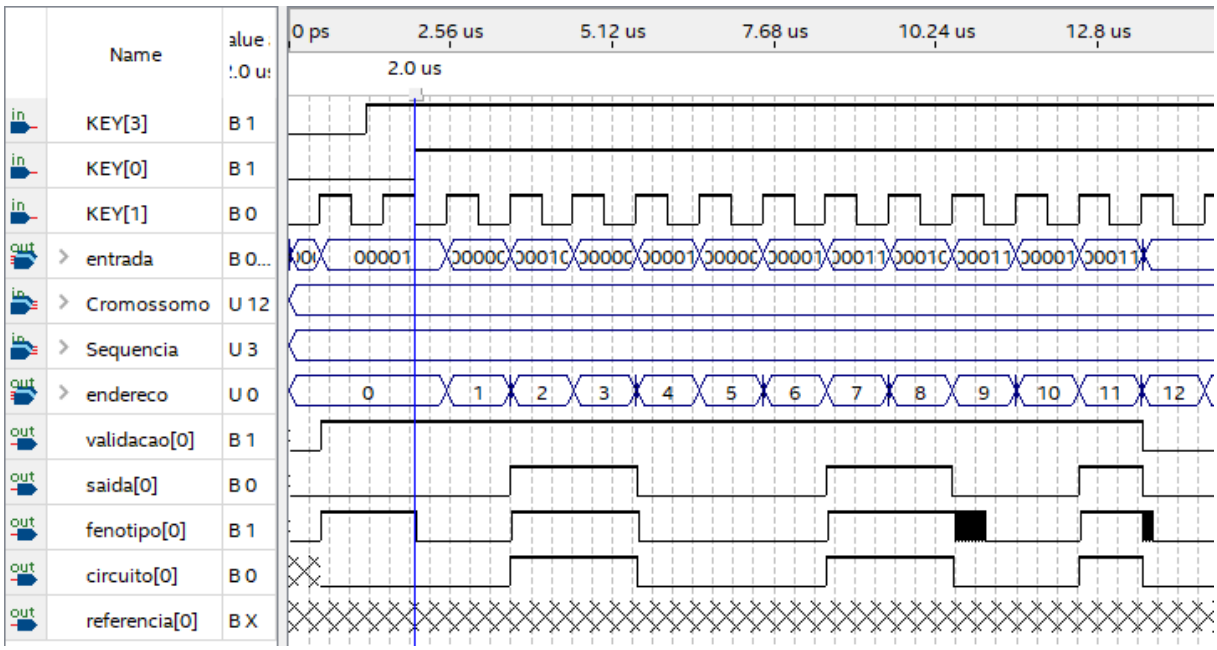


Figura II.13: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* JK da Figura 4.12c.

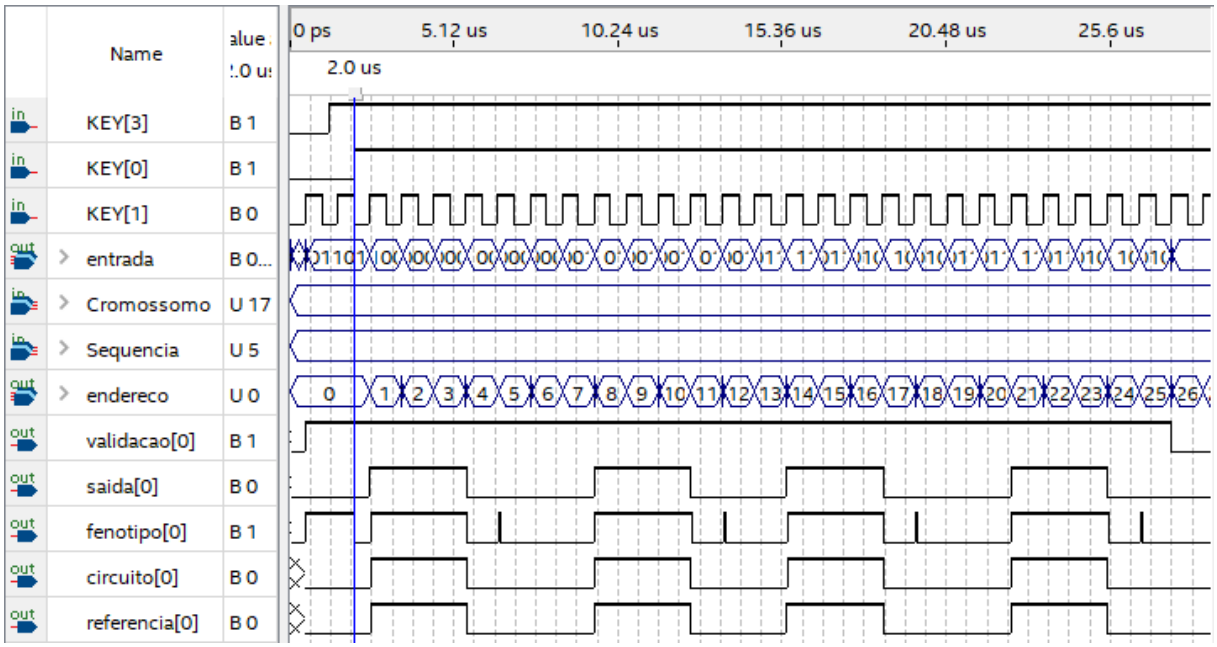


Figura II.14: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D multiplexada da Figura 4.15a.

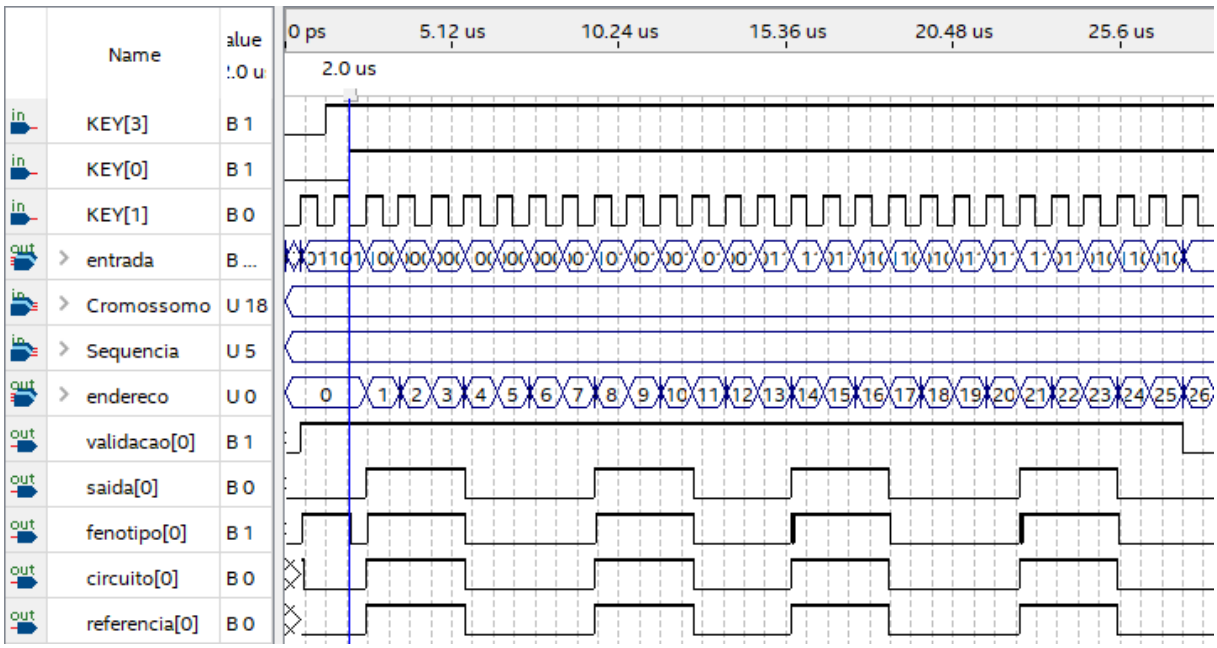


Figura II.15: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D multiplexada da Figura 4.15b.

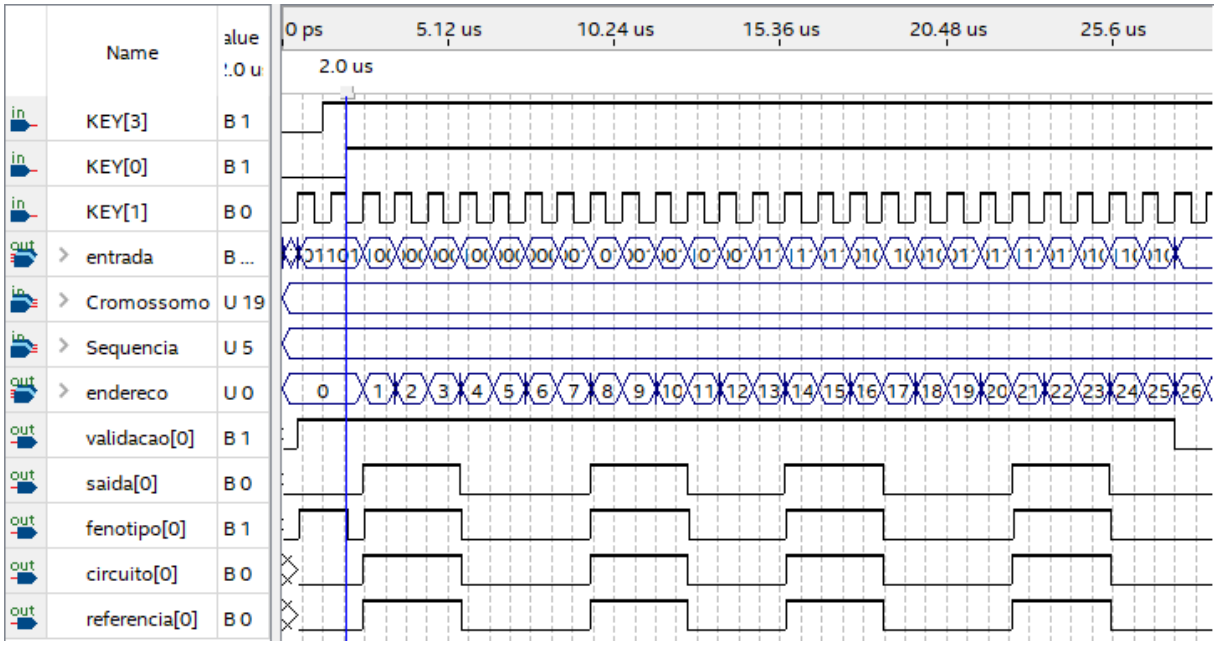


Figura II.16: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D multiplexada da Figura 4.14.

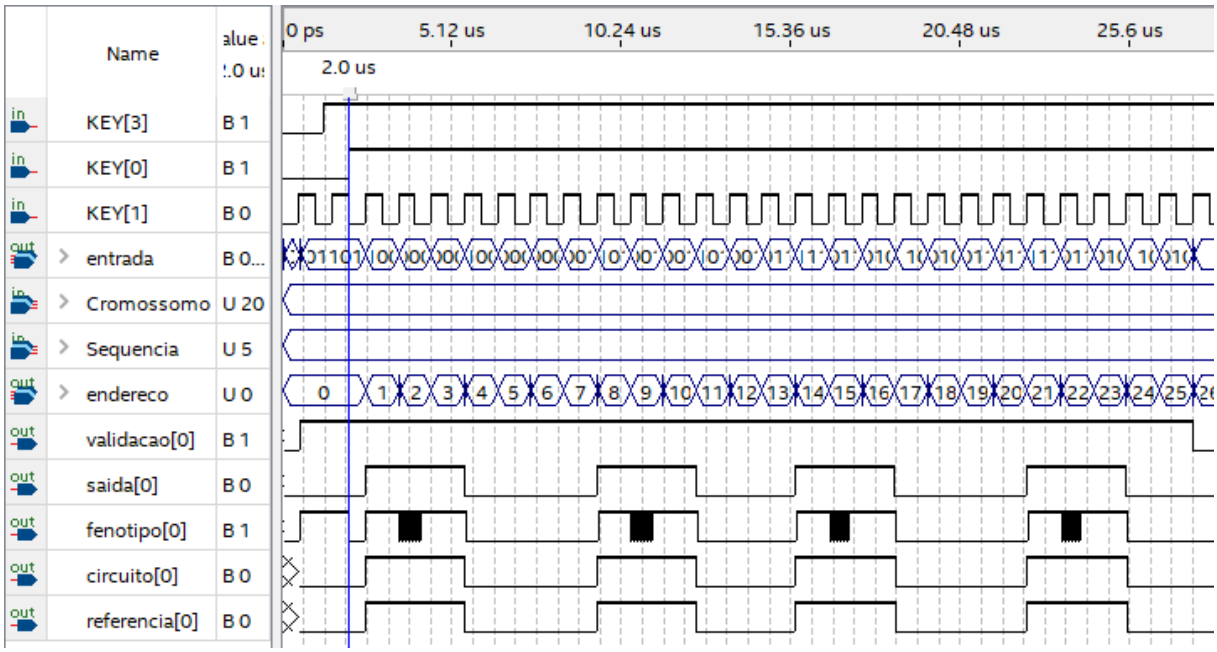


Figura II.17: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* D multiplexada da Figura 4.15c.

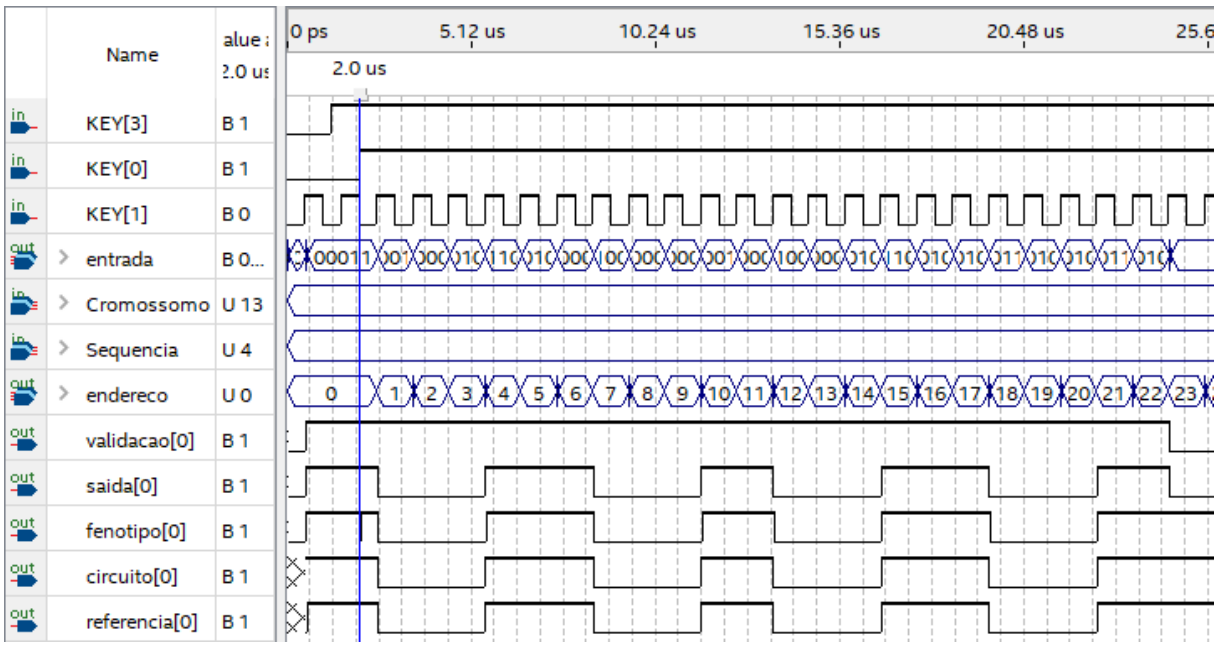


Figura II.18: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* de duas portas da Figura 4.18.

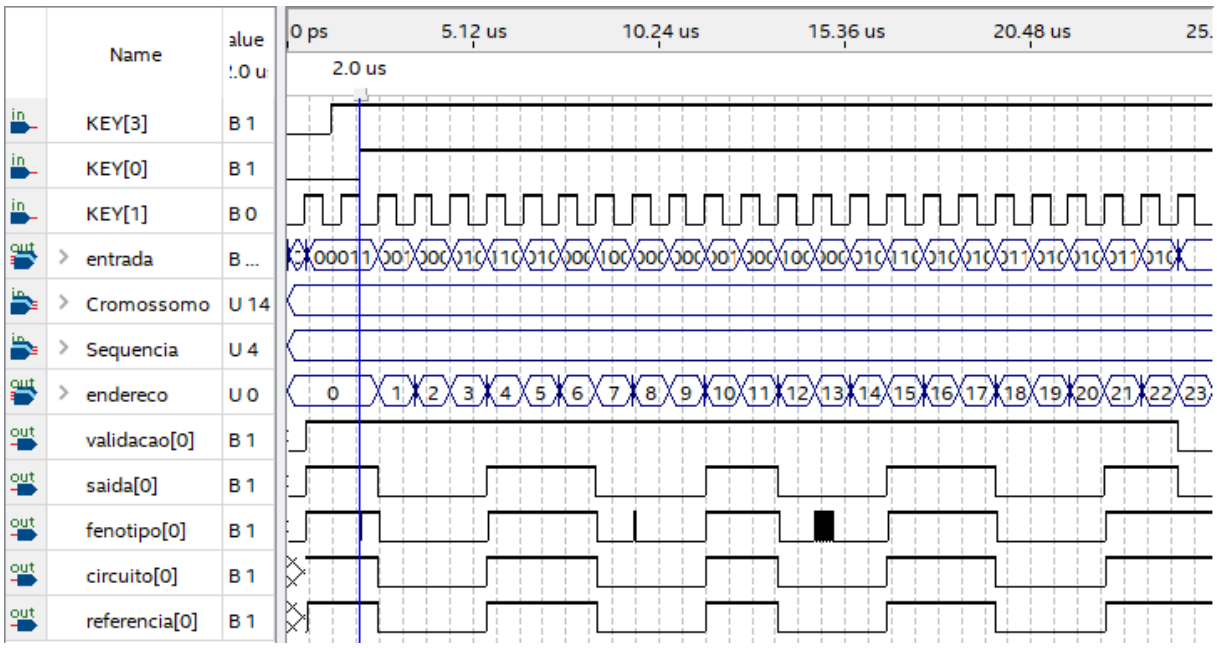


Figura II.19: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* de duas portas da Figura 4.19a.

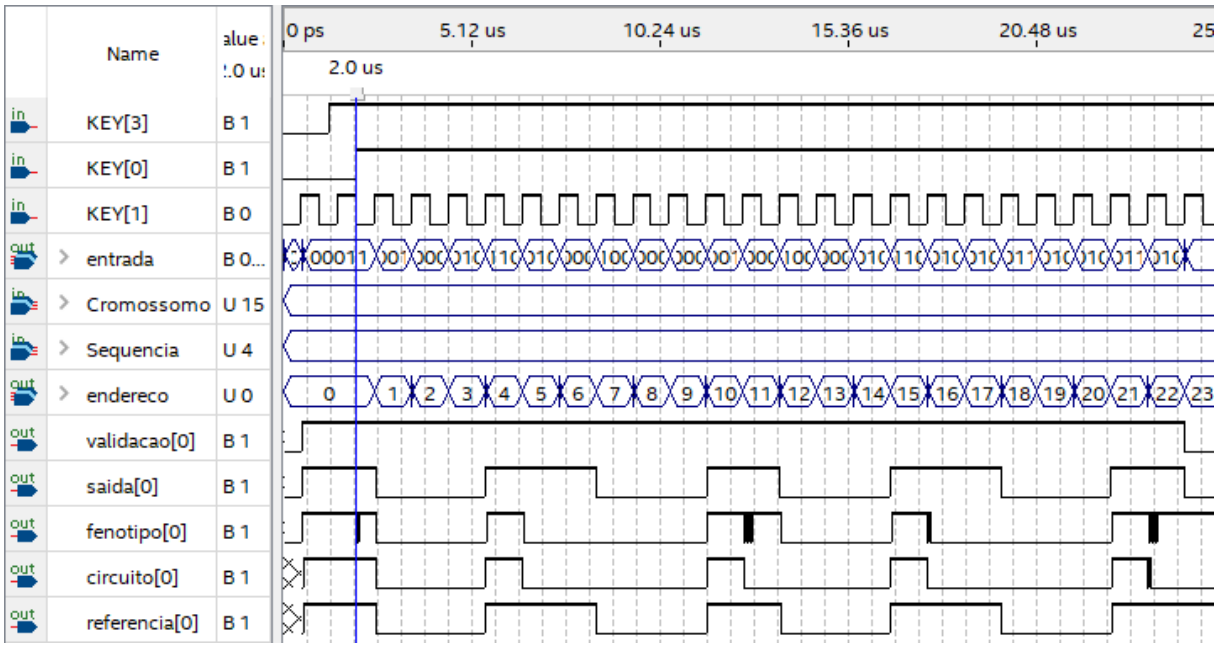


Figura II.20: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* de duas portas da Figura 4.19b.

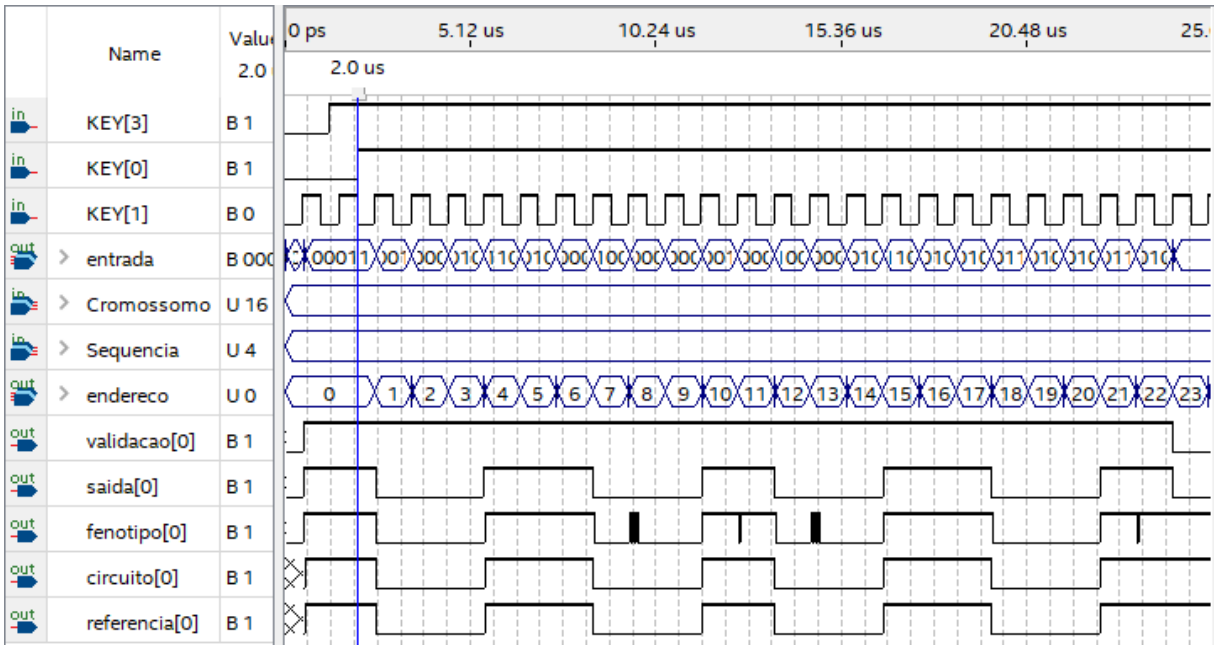


Figura II.21: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* de duas portas da Figura 4.19c.

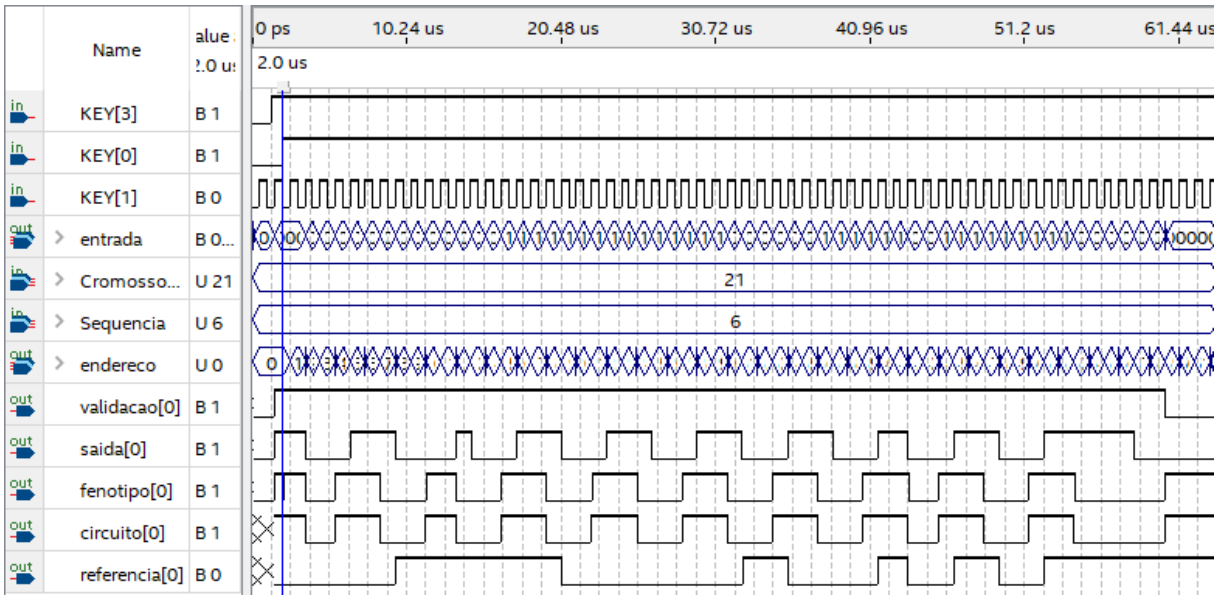


Figura II.22: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* BILBO da Figura 4.23a.

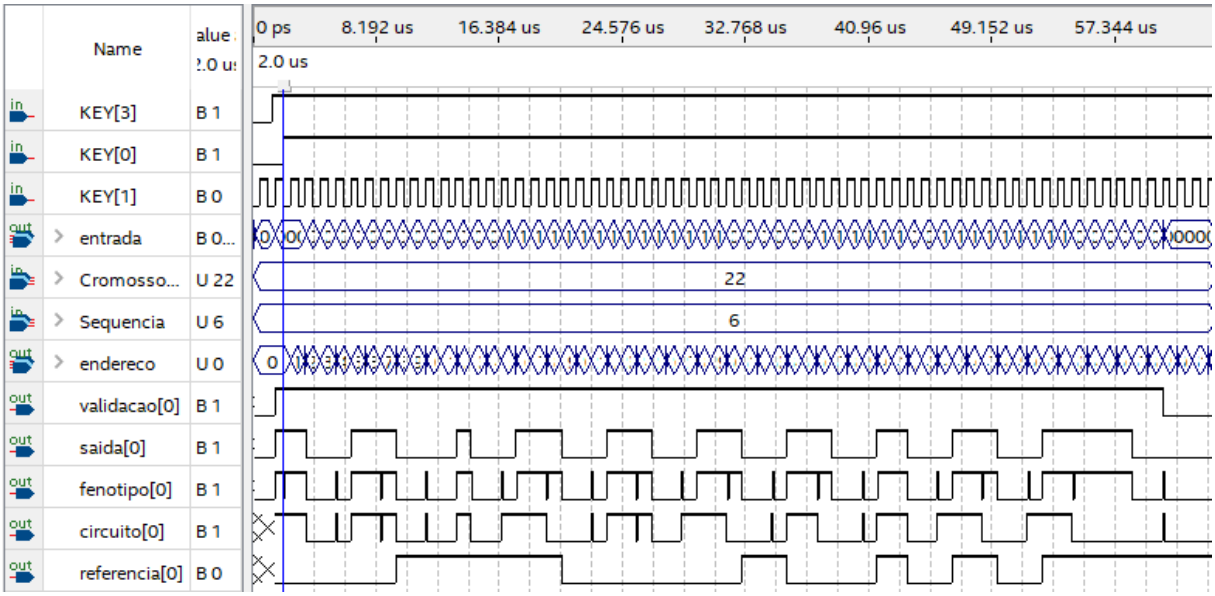


Figura II.23: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* BILBO da Figura 4.23b.

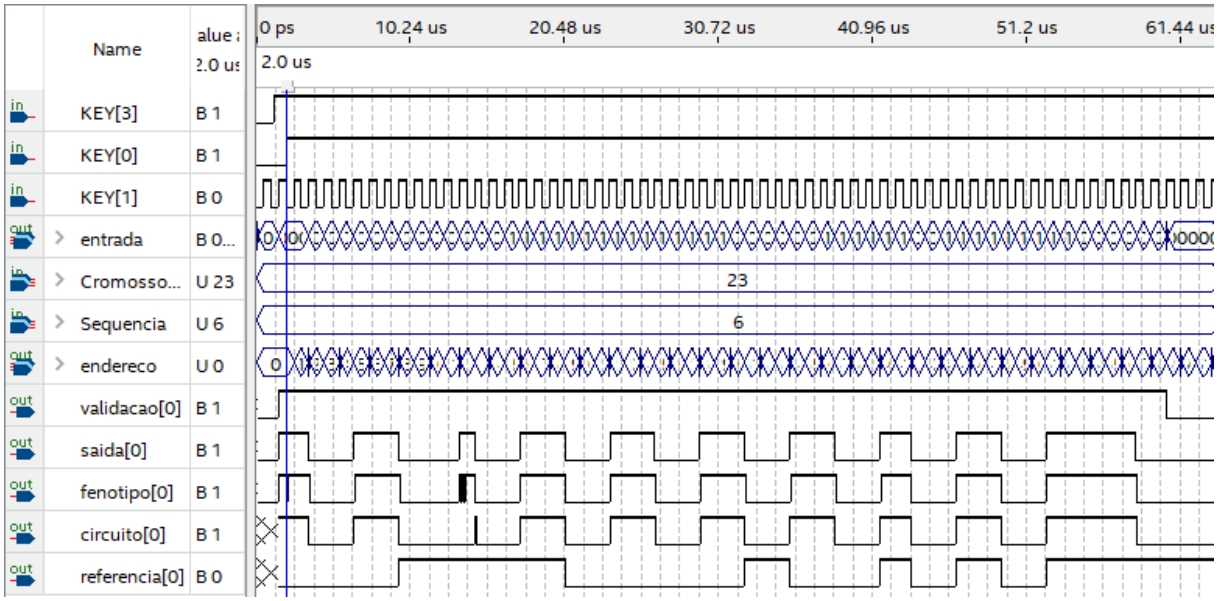


Figura II.24: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* BILBO da Figura 4.23c.

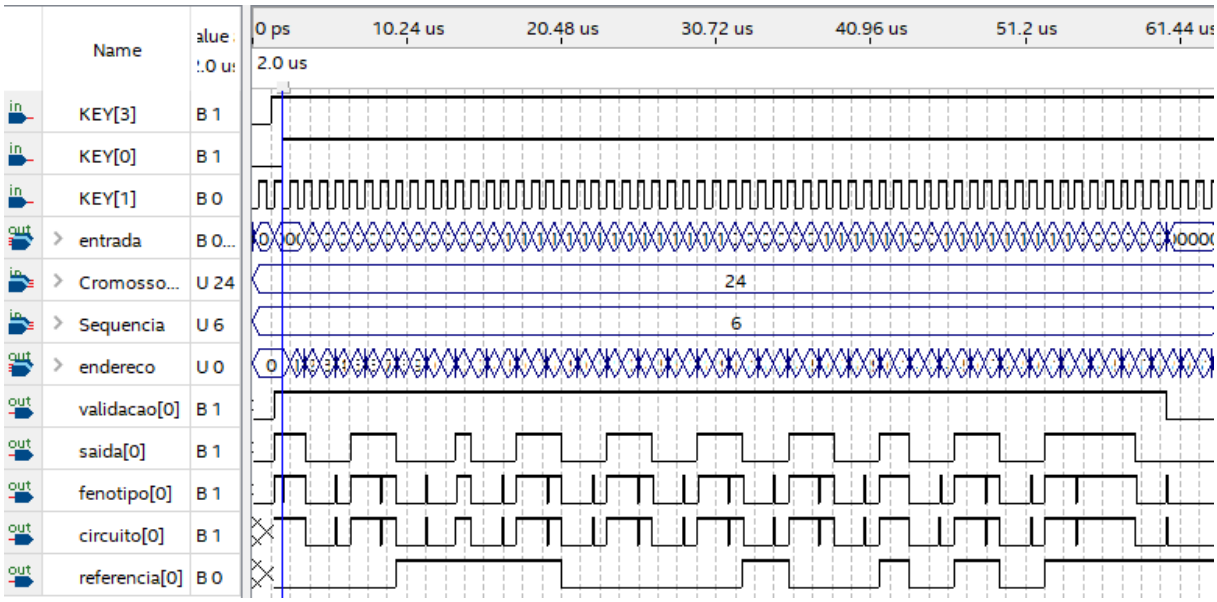


Figura II.25: Simulação *Modelsim* pela *Cyclone IV* do circuito *latch* BILBO da Figura 4.23d.

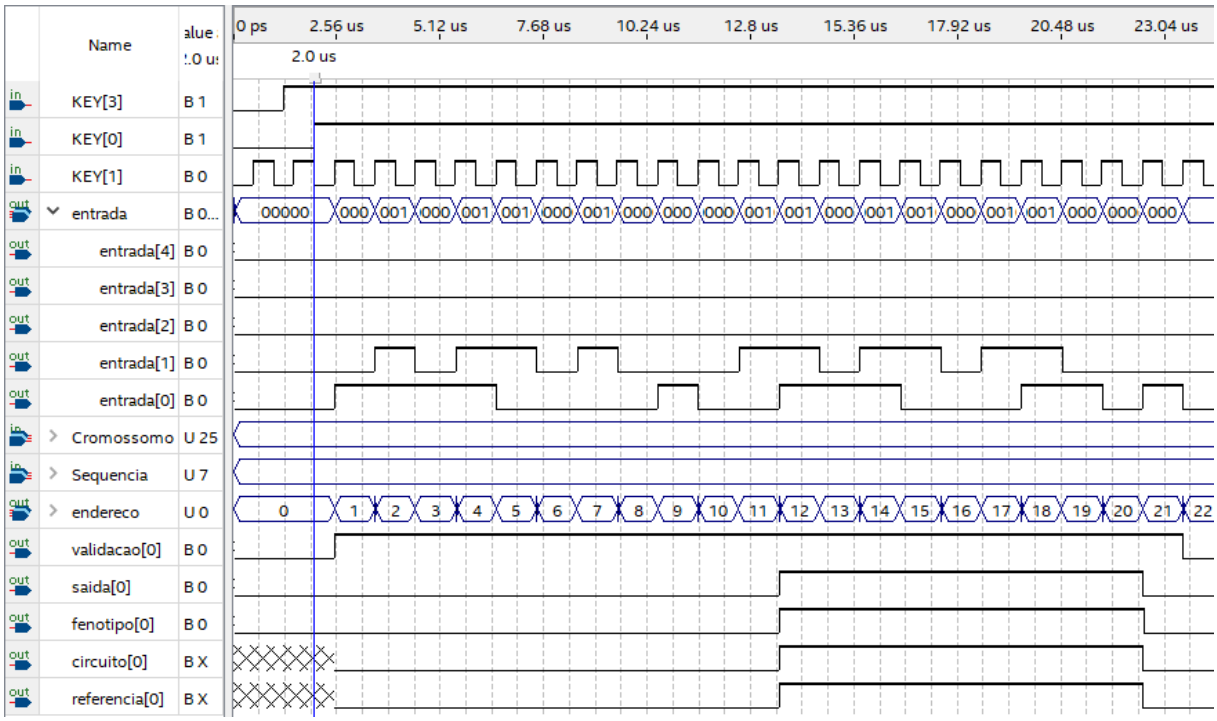


Figura II.26: Simulação *Modelsim* pela *Cyclone IV* do circuito *flip-flop* D da Figura 4.32a.

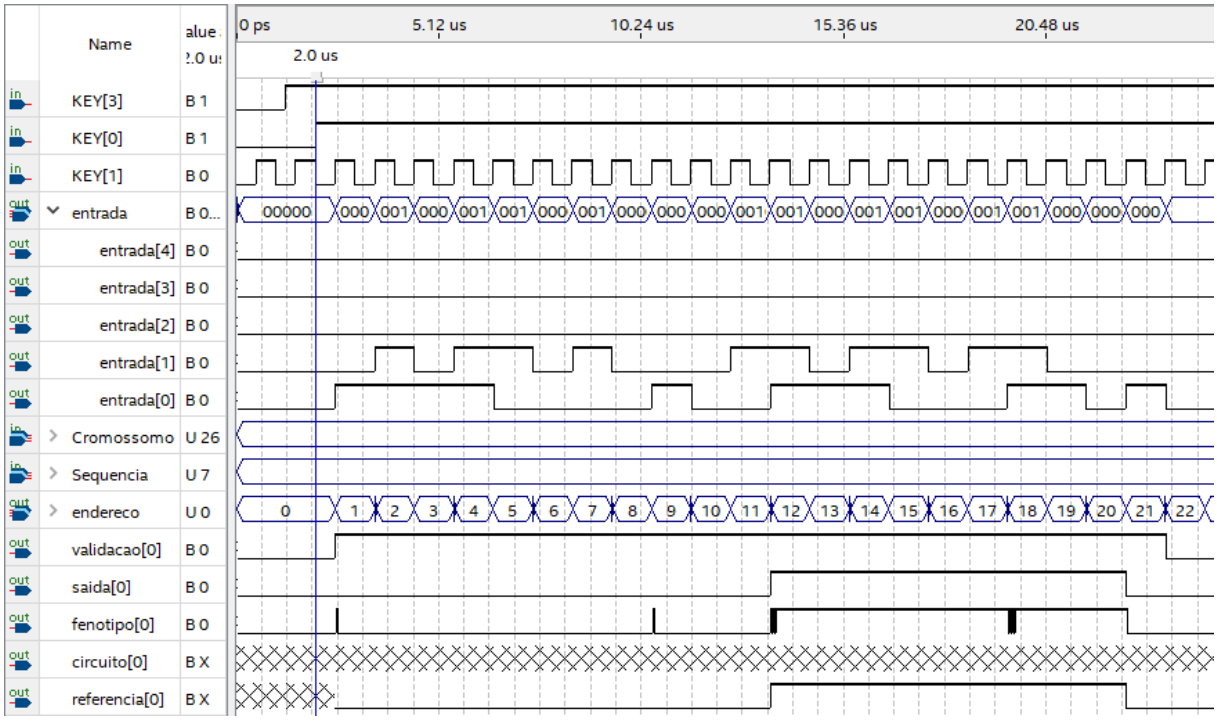


Figura II.27: Simulação *Modelsim* pela *Cyclone IV* do circuito *flip-flop* D da Figura 4.32b.

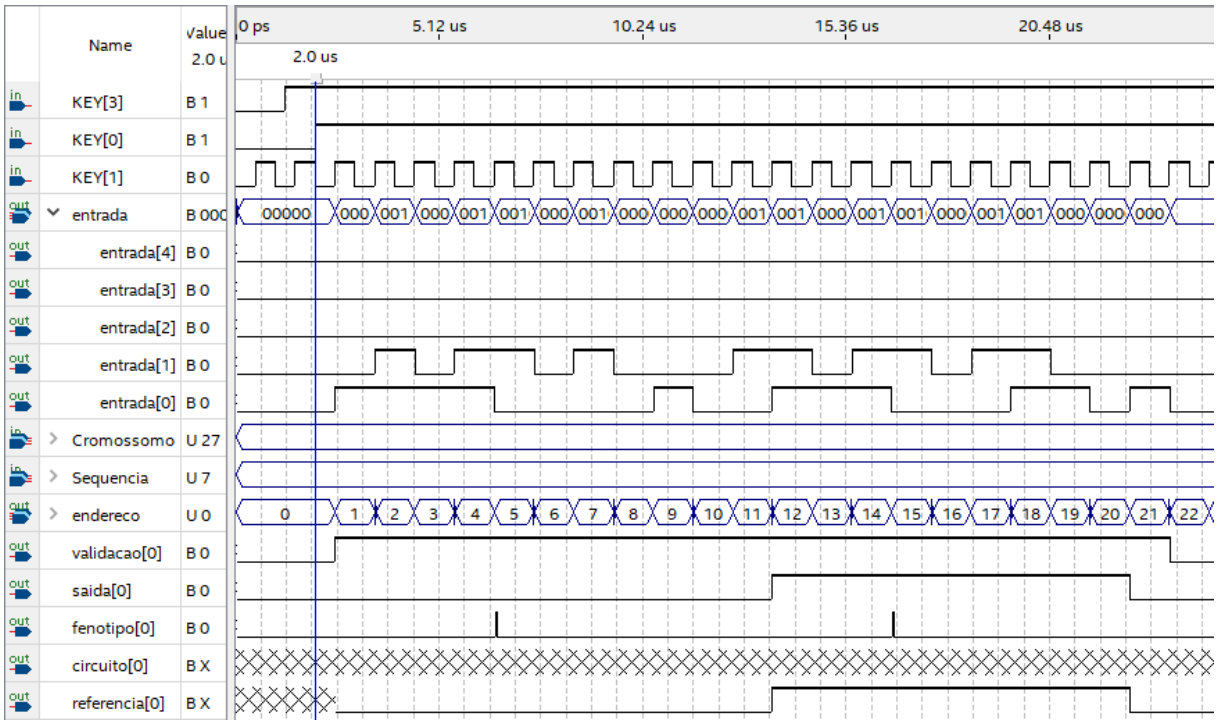


Figura II.28: Simulação *Modelsim* pela *Cyclone IV* do circuito *flip-flop* D da Figura 4.32c.

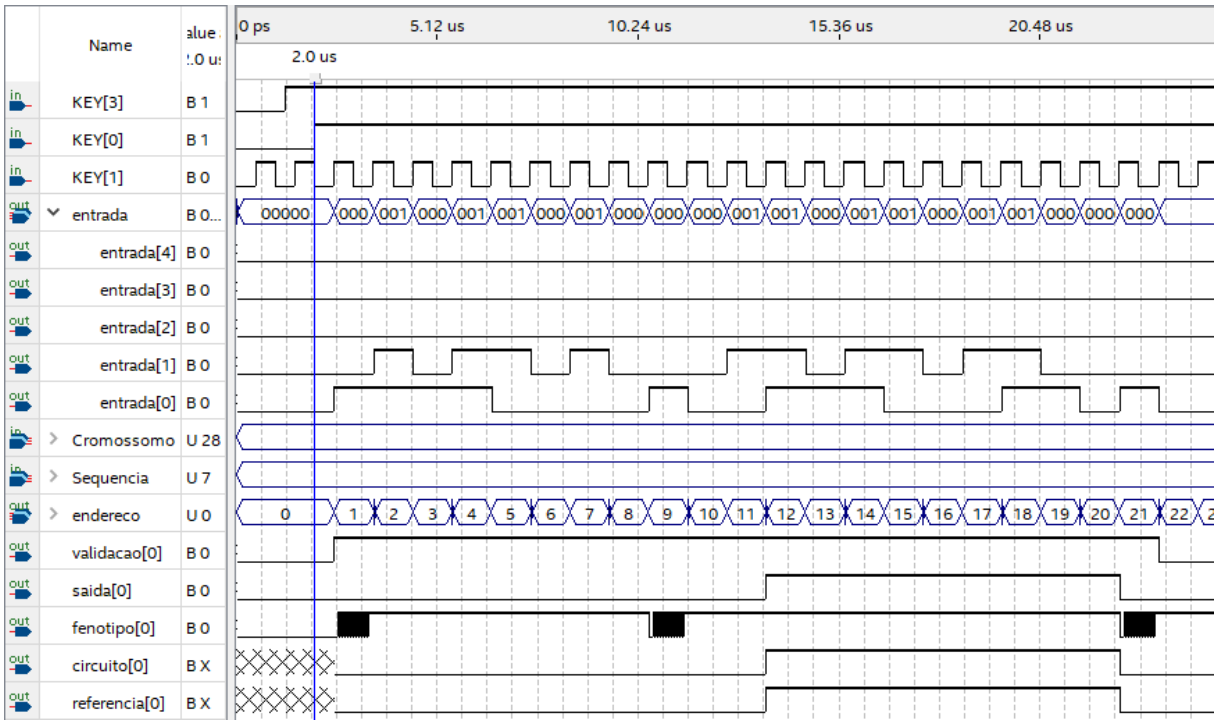


Figura II.29: Simulação *Modelsim* pela *Cyclone IV* do circuito *flip-flop* D da Figura 4.32d.

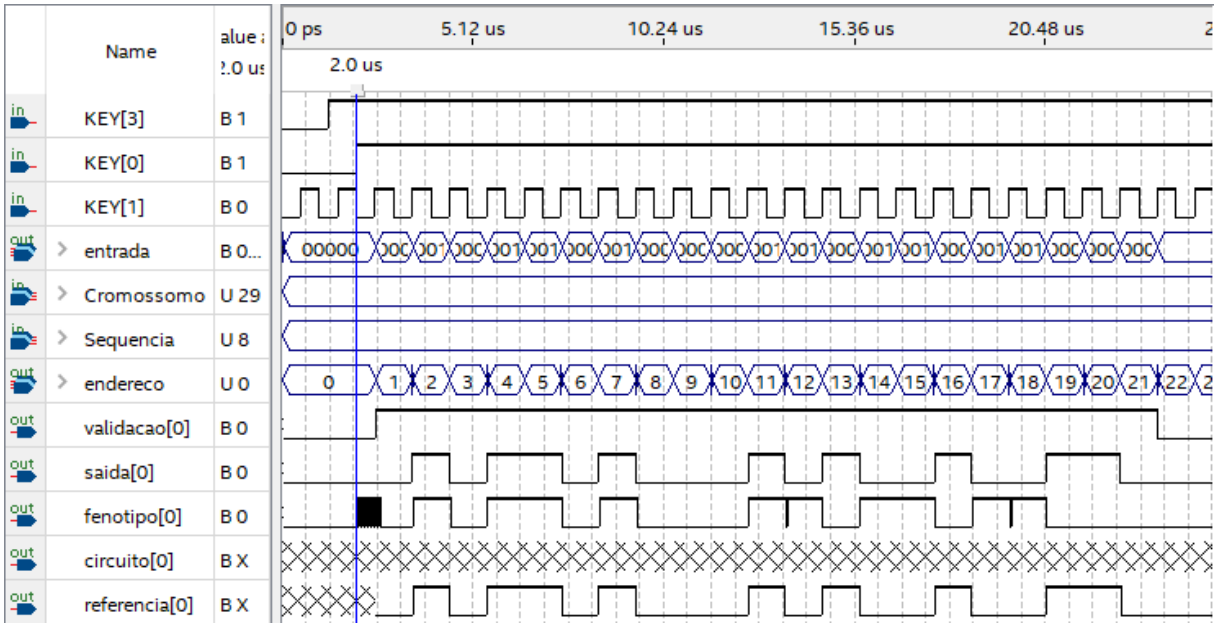


Figura II.30: Simulação *Modelsim* pela *Cyclone IV* do circuito paridade-2 da Figura 4.33a.

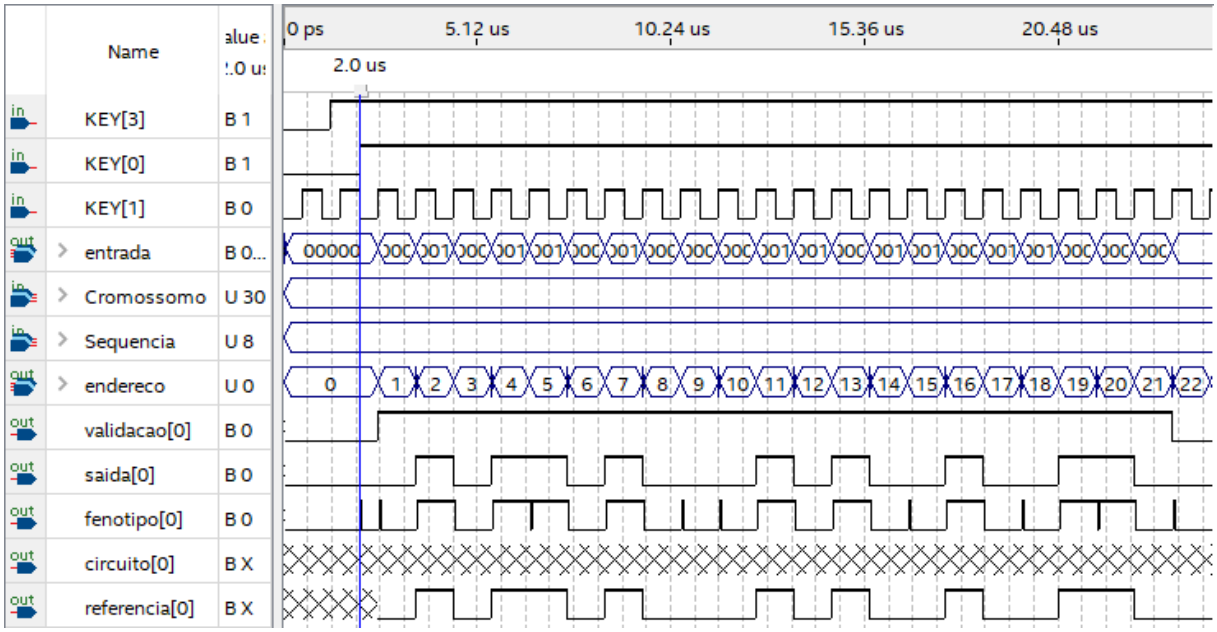


Figura II.31: Simulação *Modelsim* pela *Cyclone IV* do circuito paridade-2 da Figura 4.33b.

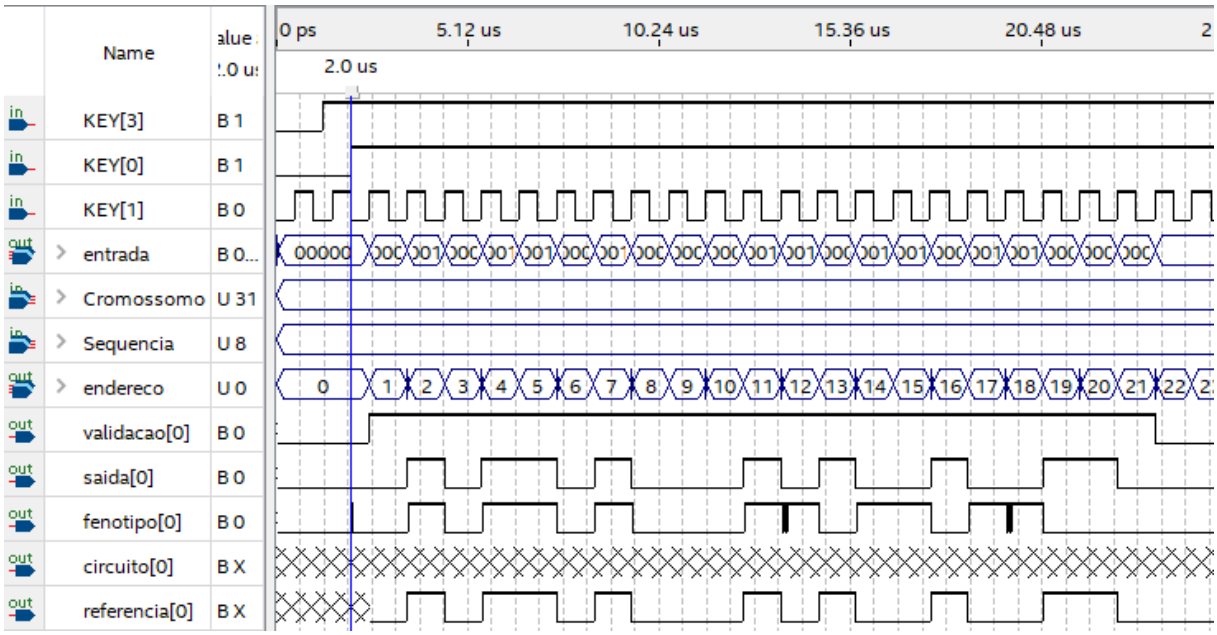


Figura II.32: Simulação *Modelsim* pela *Cyclone IV* do circuito paridade-2 da Figura 4.33c.

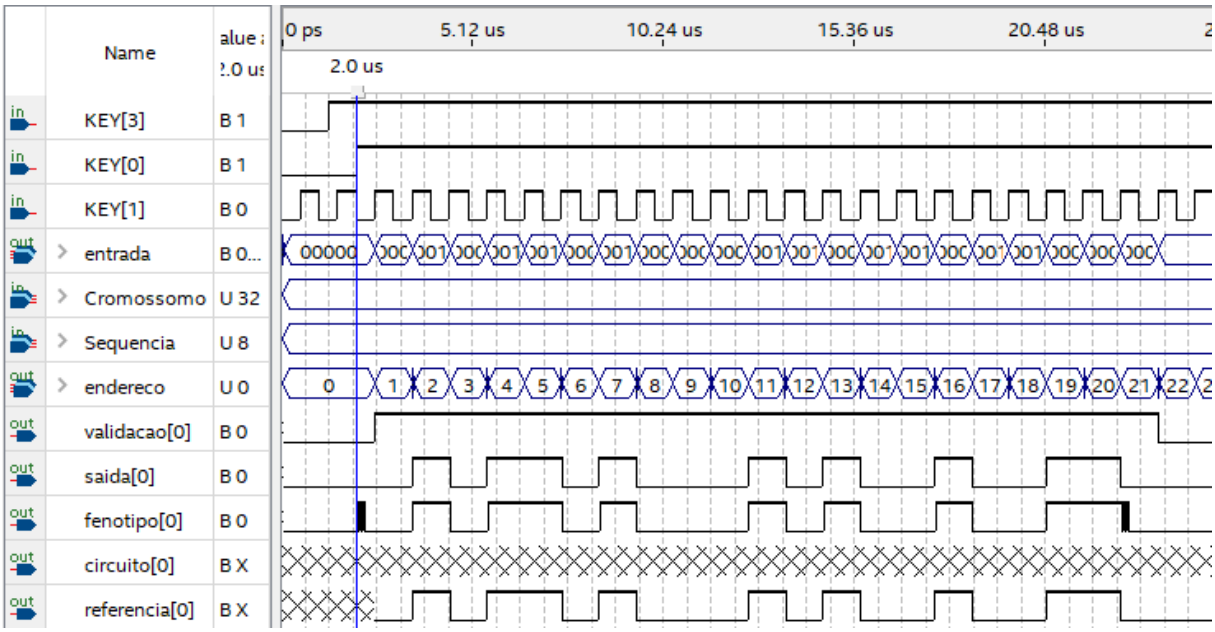


Figura II.33: Simulação *Modelsim* pela *Cyclone IV* do circuito paridade-2 da Figura 4.33d.

Anexo III

Simulações dos Resultados - Cyclone II

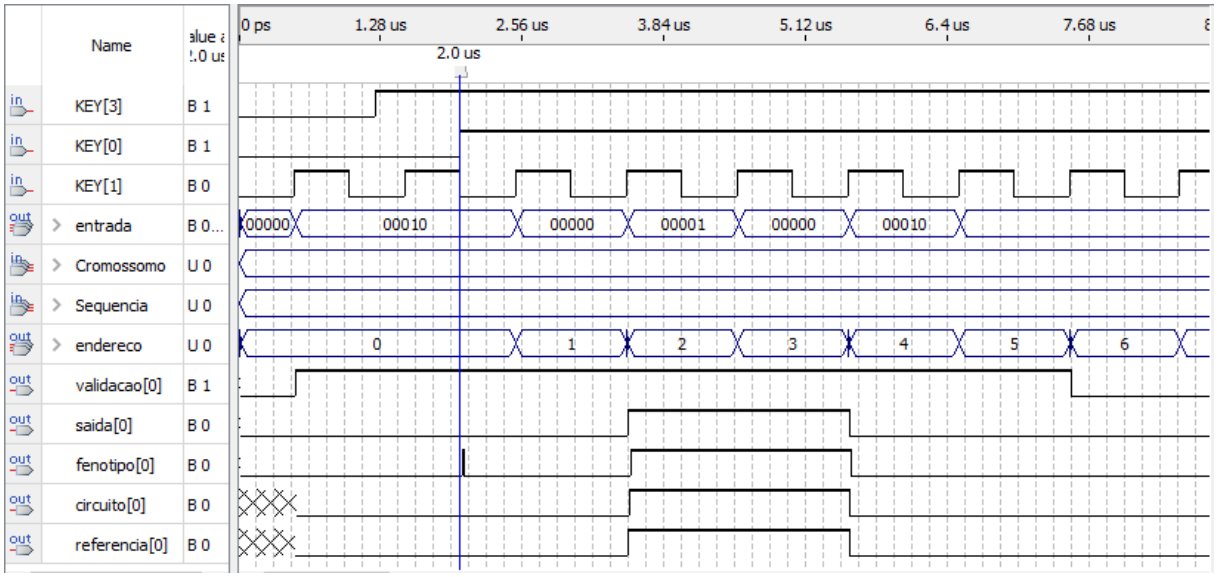


Figura III.1: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* SR da Figura 4.2.

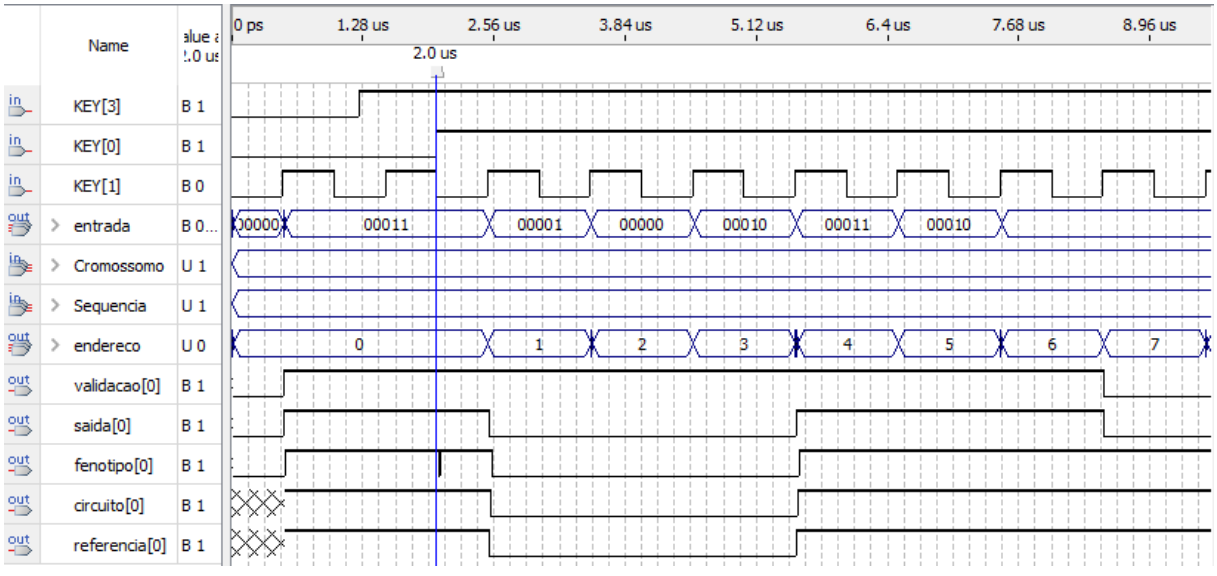


Figura III.2: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D da Figura 4.4.

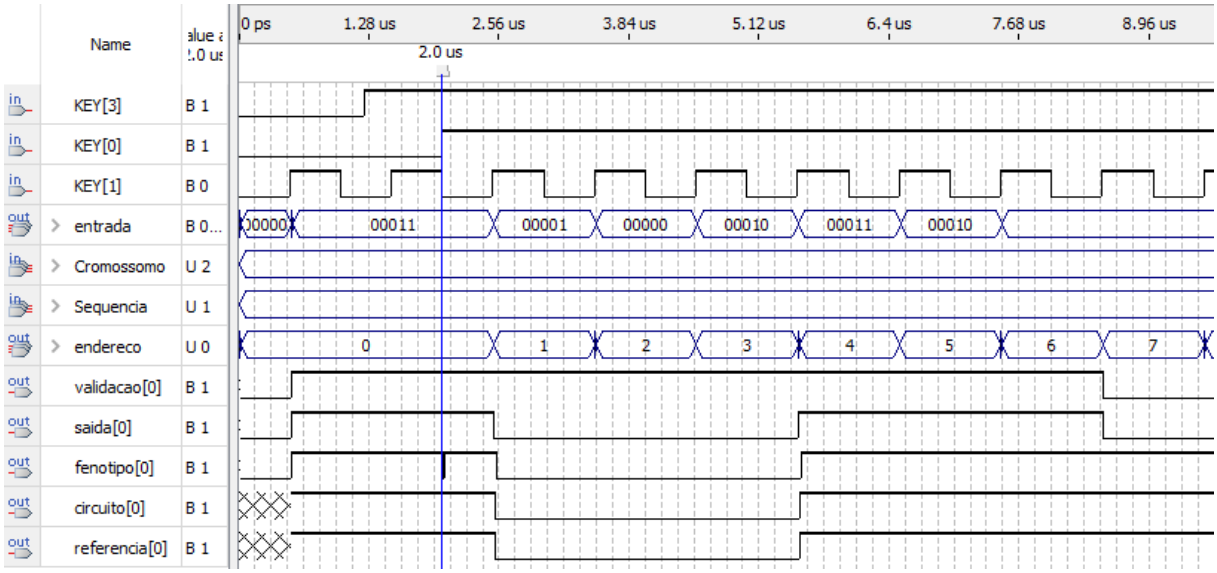


Figura III.3: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D da Figura 4.6a.

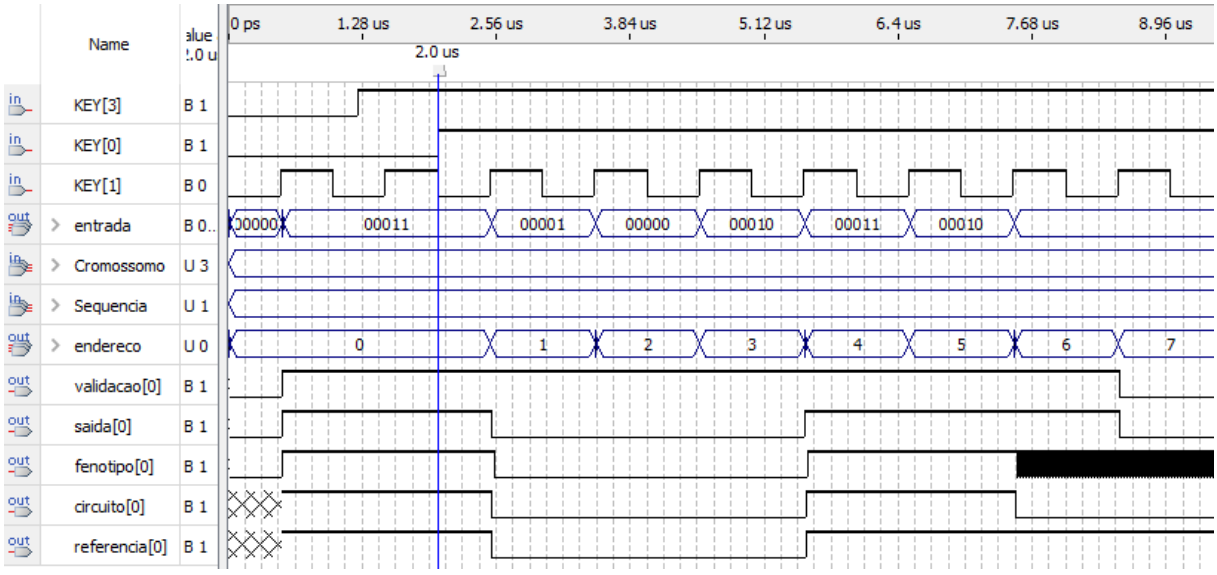


Figura III.4: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D da Figura 4.6b.

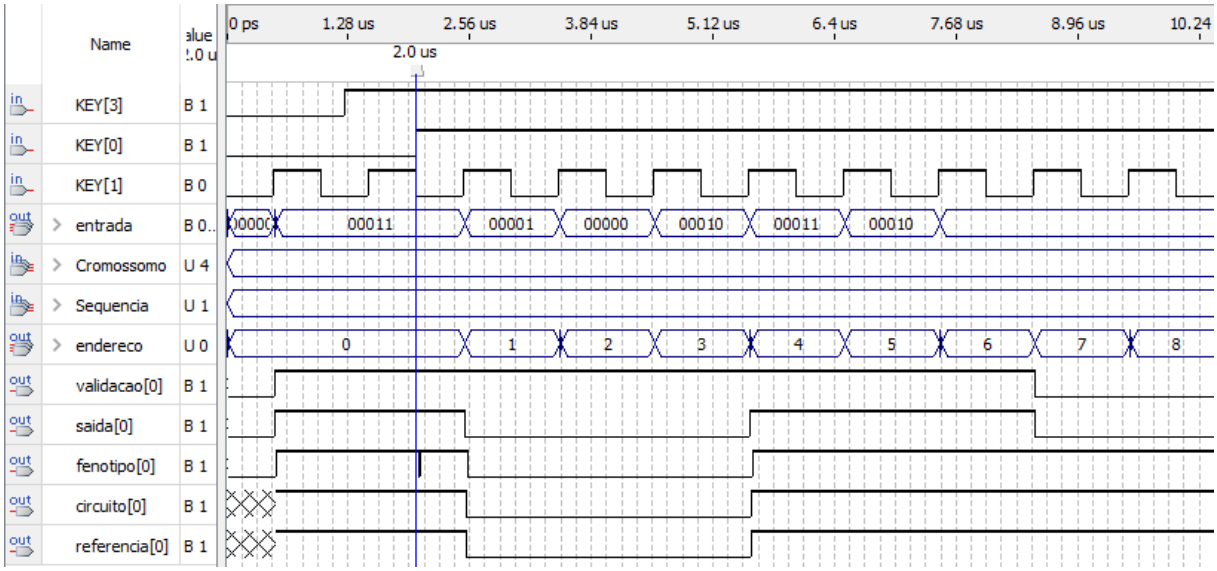


Figura III.5: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D da Figura 4.6c.

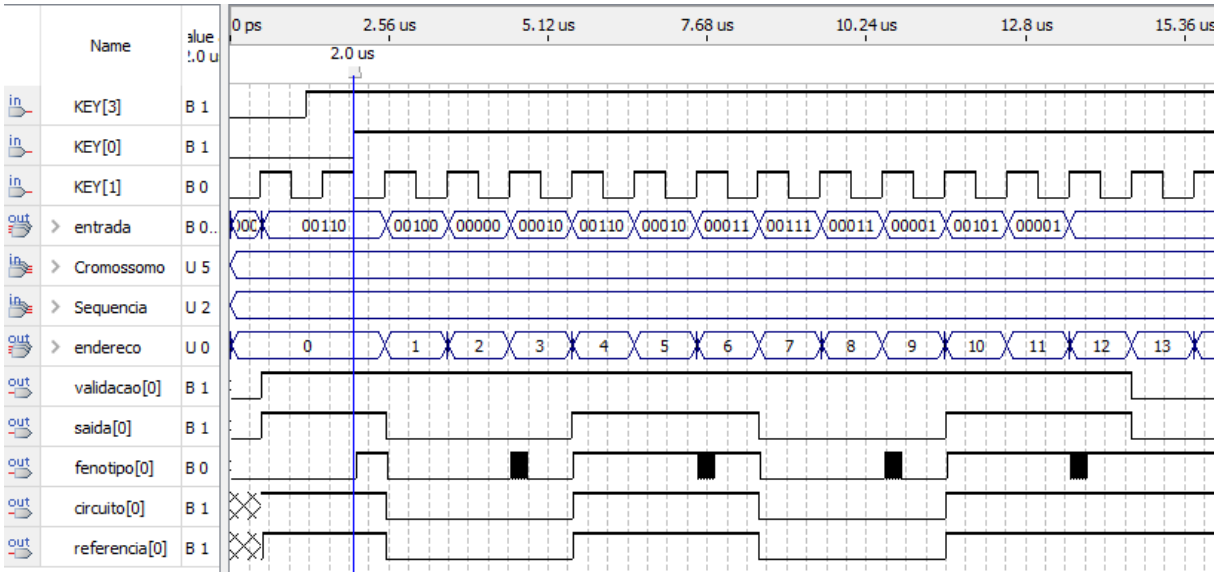


Figura III.6: Simulação *Modelsim* pela *Cyclone II* do circuito *latch XOR* da Figura 4.9a.

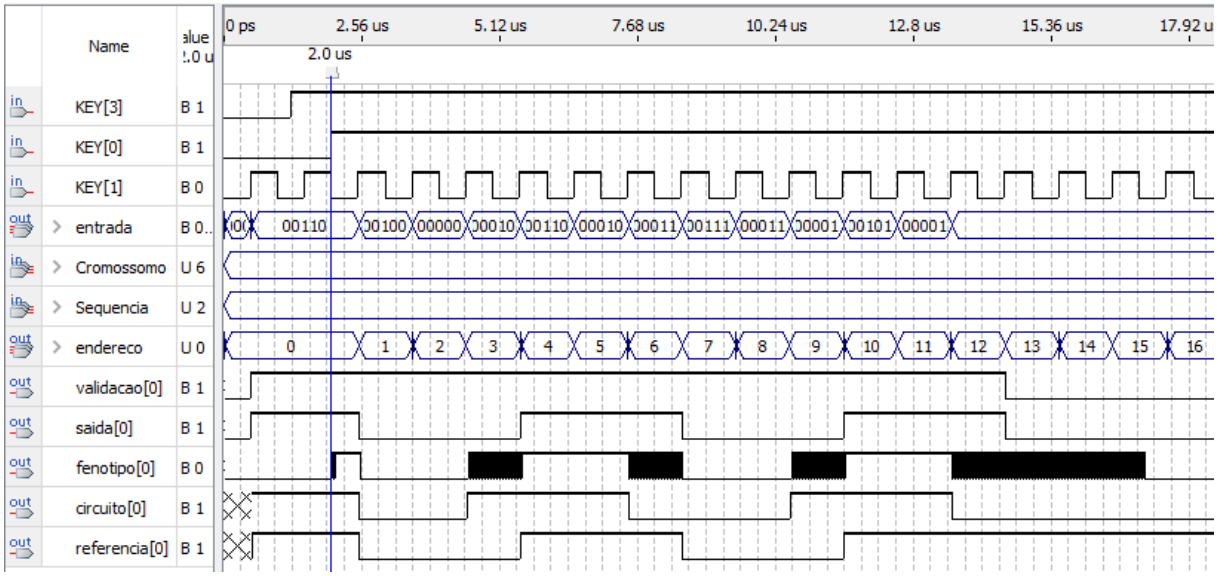


Figura III.7: Simulação *Modelsim* pela *Cyclone II* do circuito *latch XOR* da Figura 4.9b.

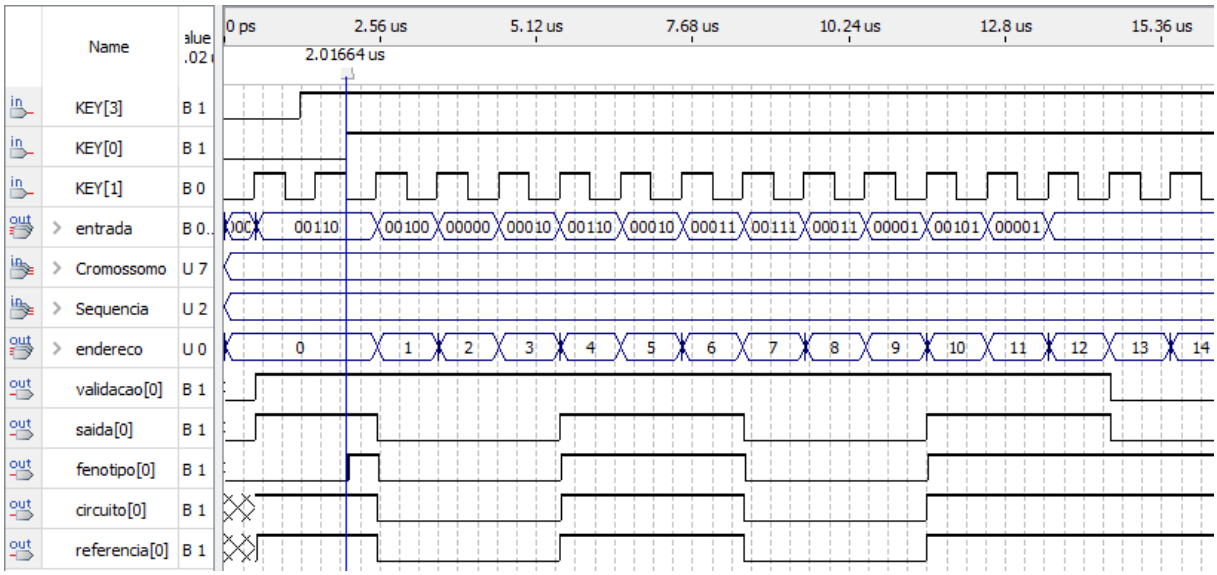


Figura III.8: Simulação *Modelsim* pela *Cyclone II* do circuito *latch XOR* da Figura 4.8.

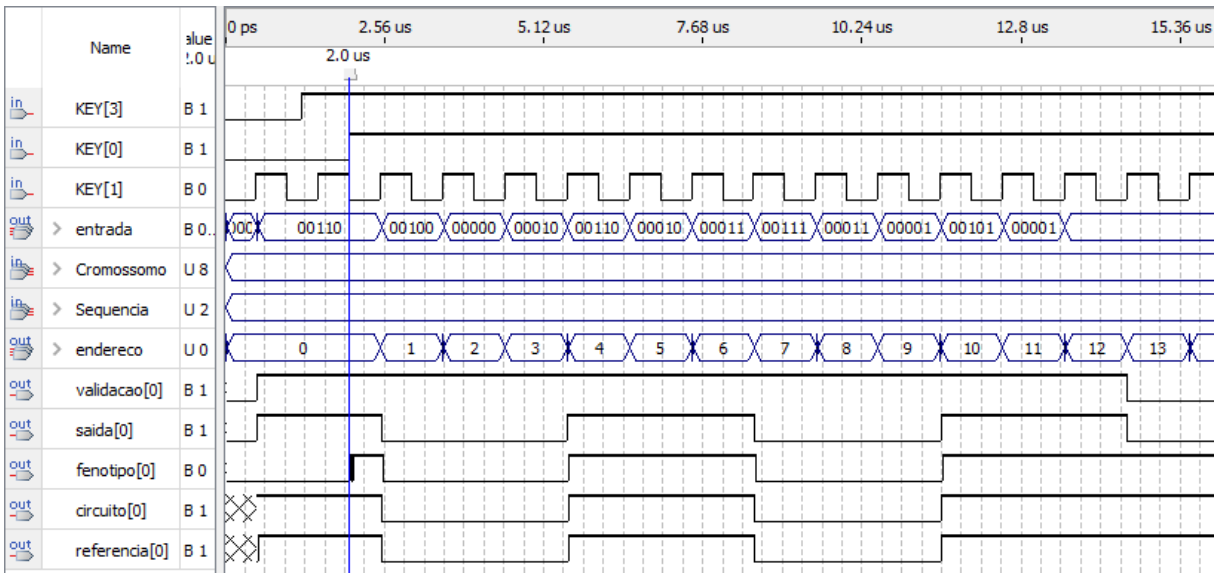


Figura III.9: Simulação *Modelsim* pela *Cyclone II* do circuito *latch XOR* da Figura 4.9c.

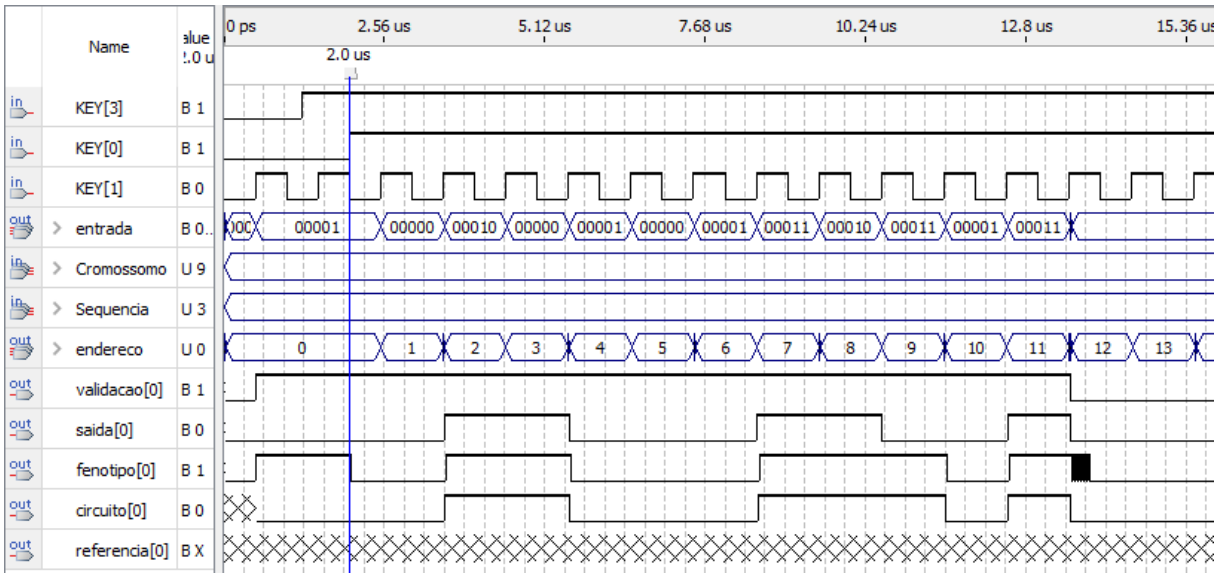


Figura III.10: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* JK da Figura 4.12a.

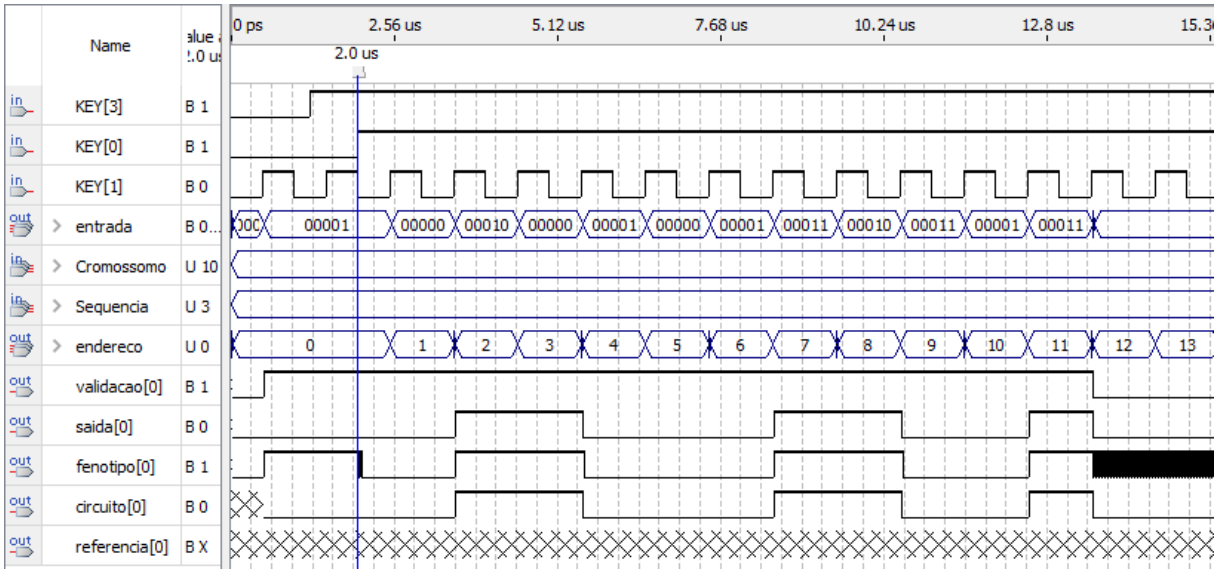


Figura III.11: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* JK da Figura 4.11.

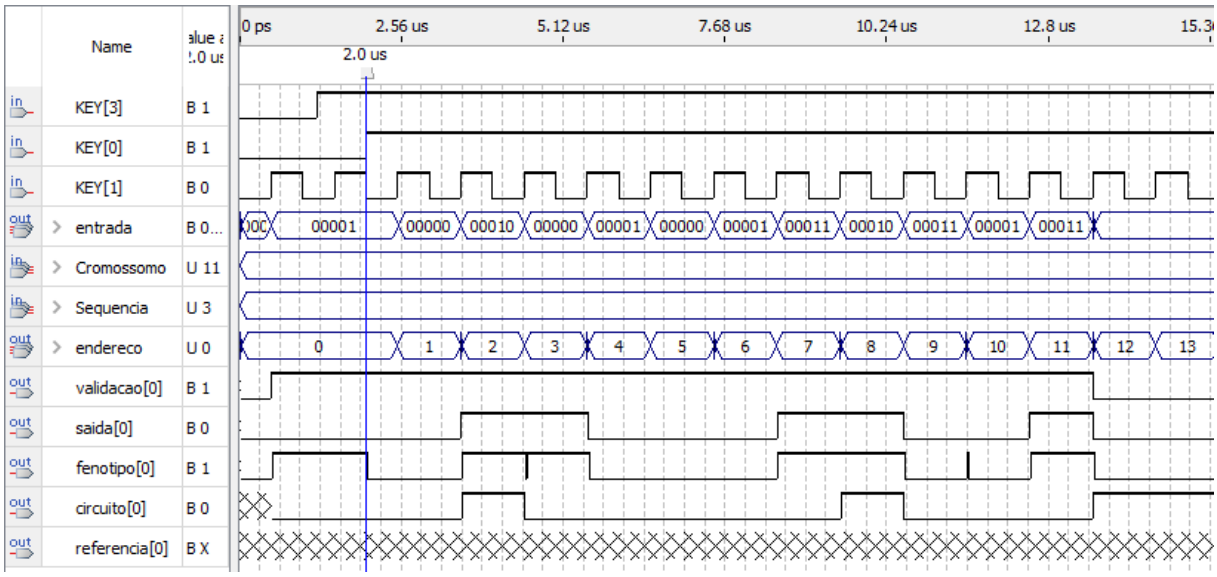


Figura III.12: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* JK da Figura 4.12b.

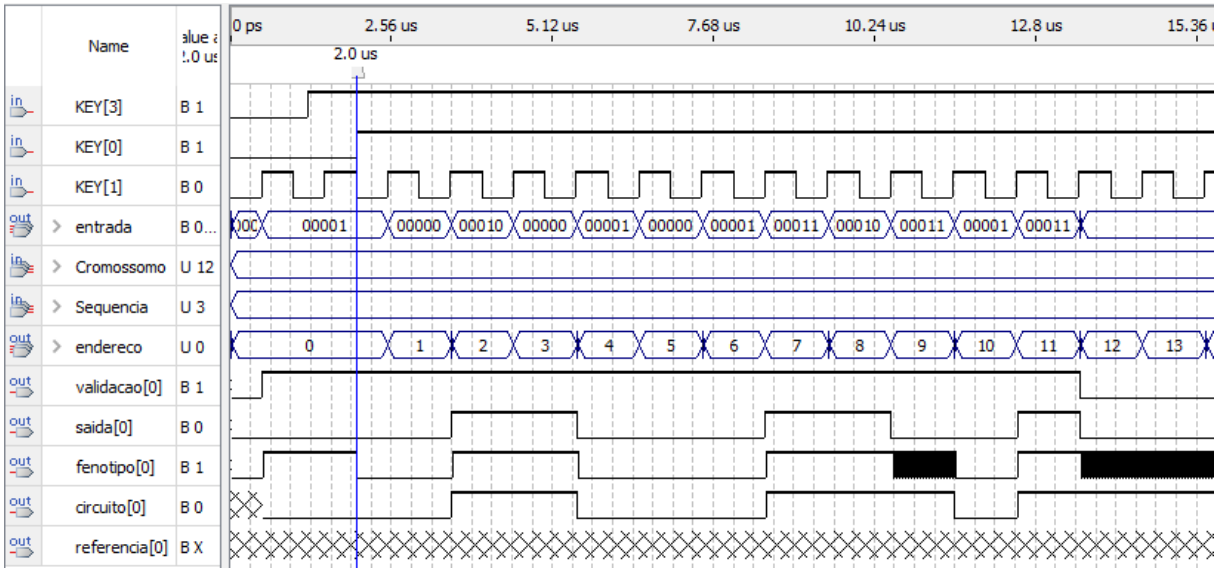


Figura III.13: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* JK da Figura 4.12c.

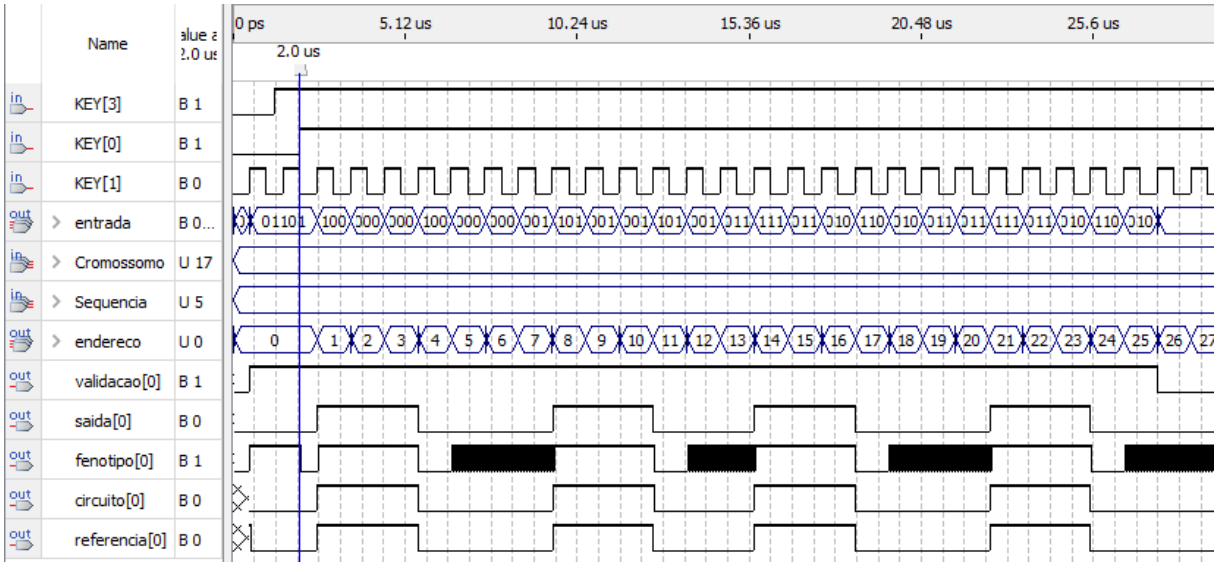


Figura III.14: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D multiplexada da Figura 4.15a.

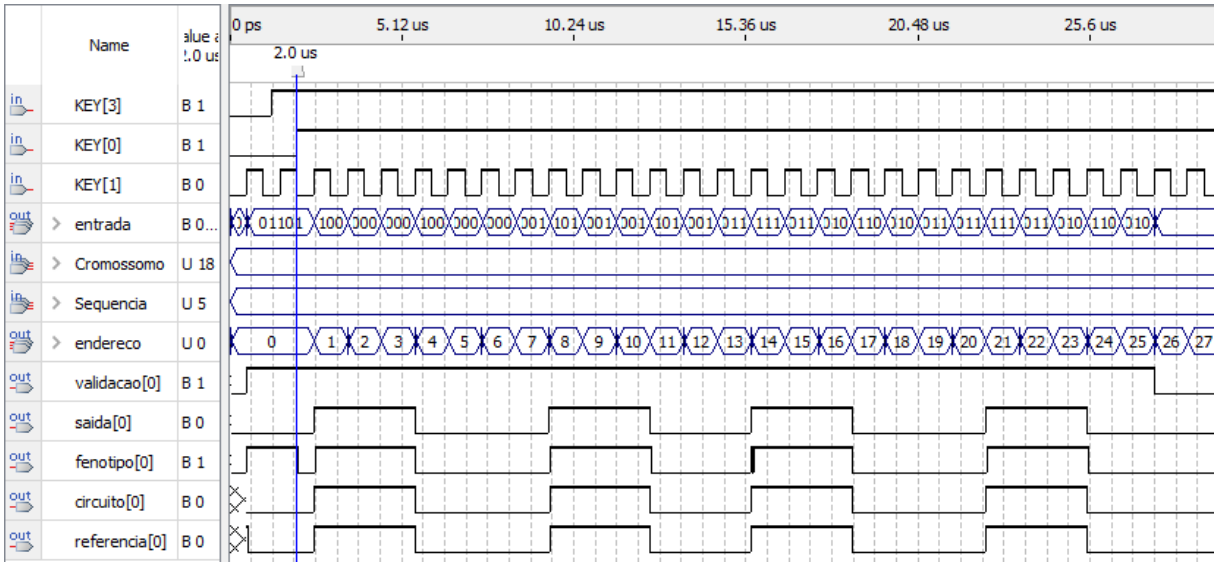


Figura III.15: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D multiplexada da Figura 4.15b.

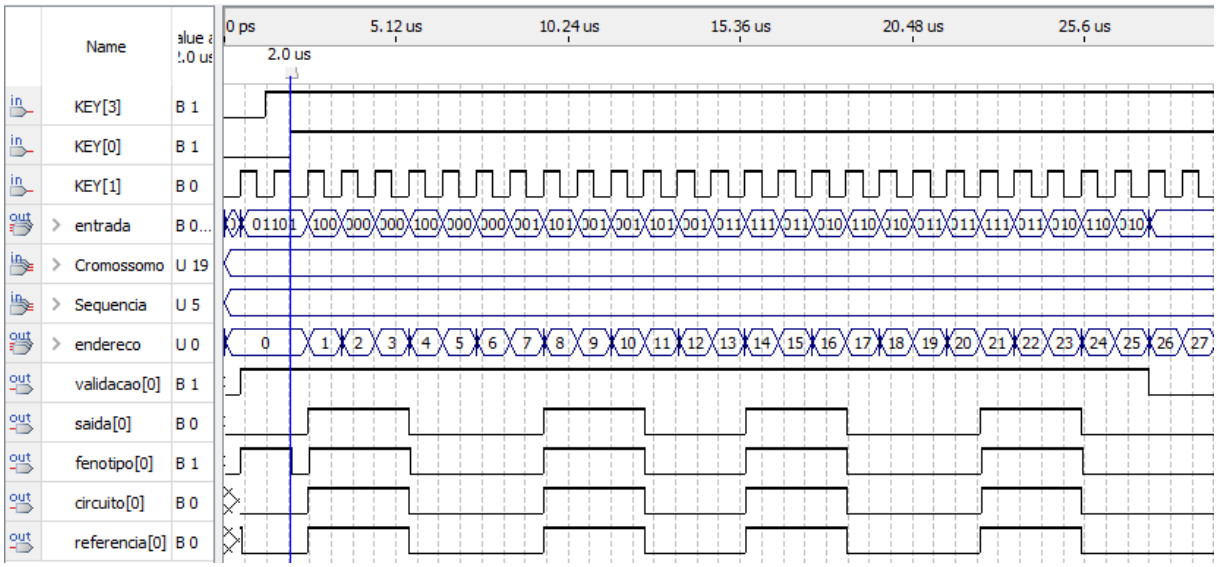


Figura III.16: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D multiplexada da Figura 4.14.

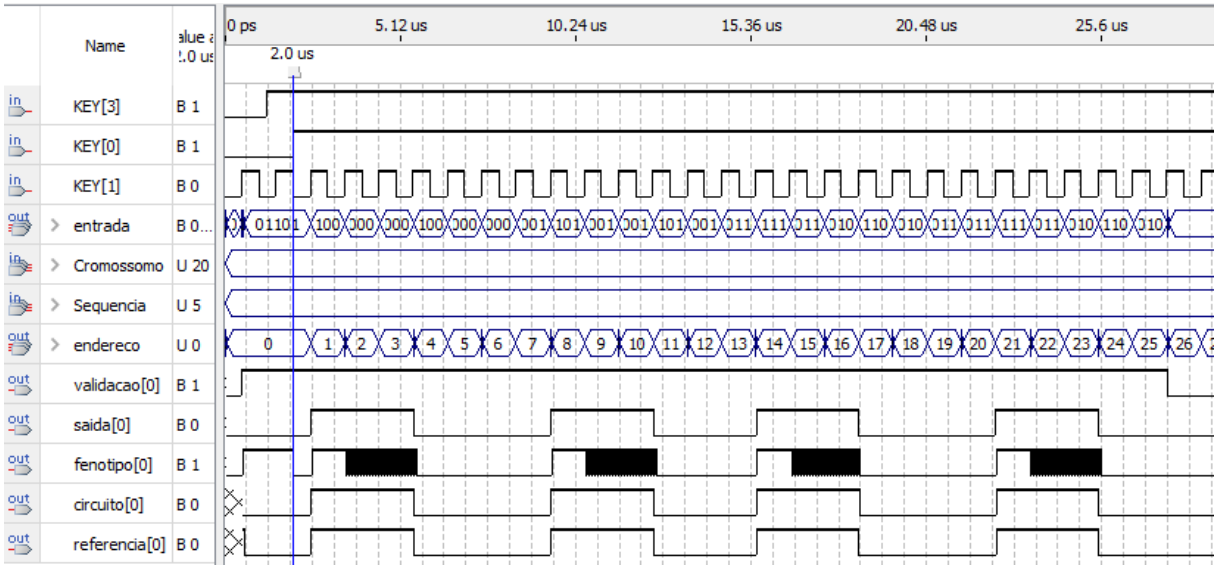


Figura III.17: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* D multiplexada da Figura 4.15c.

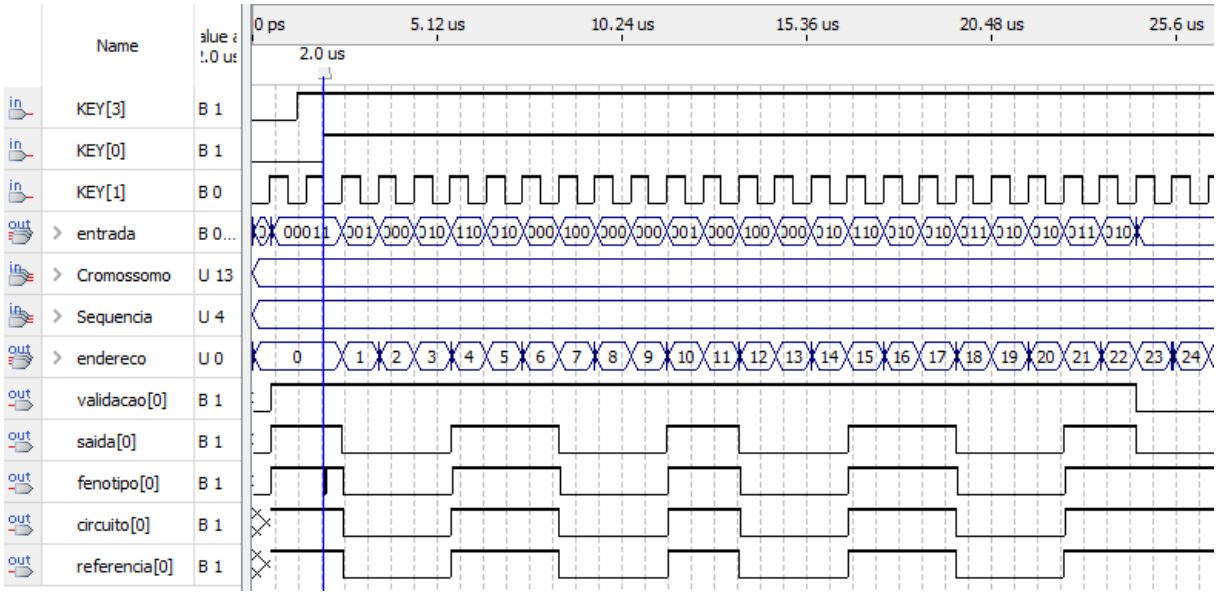


Figura III.18: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* de duas portas da Figura 4.18.

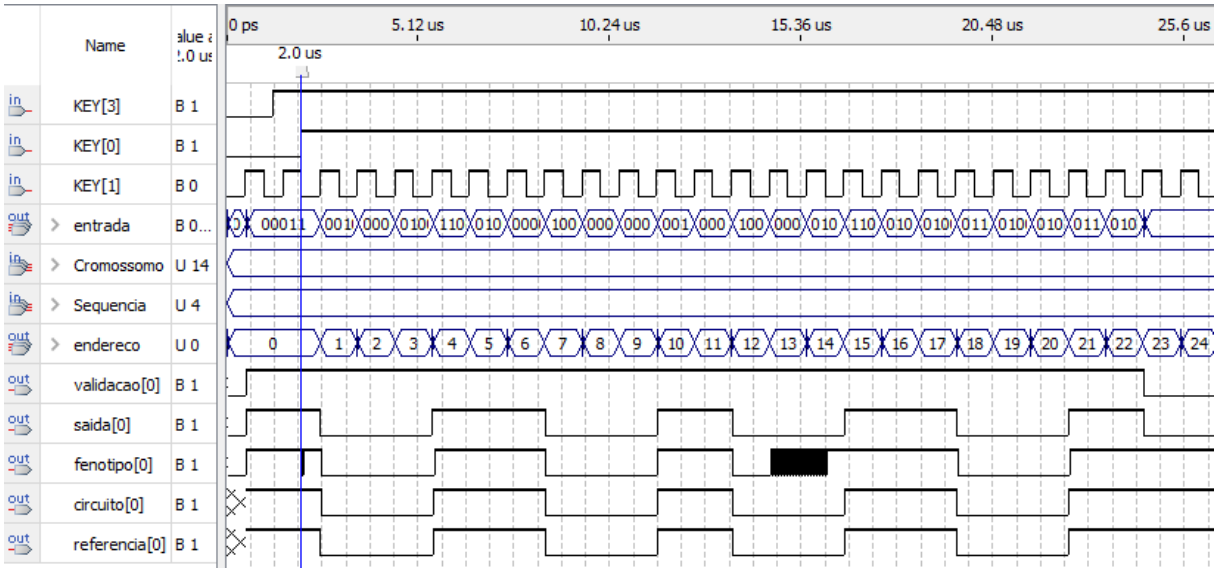


Figura III.19: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* de duas portas da Figura 4.19a.

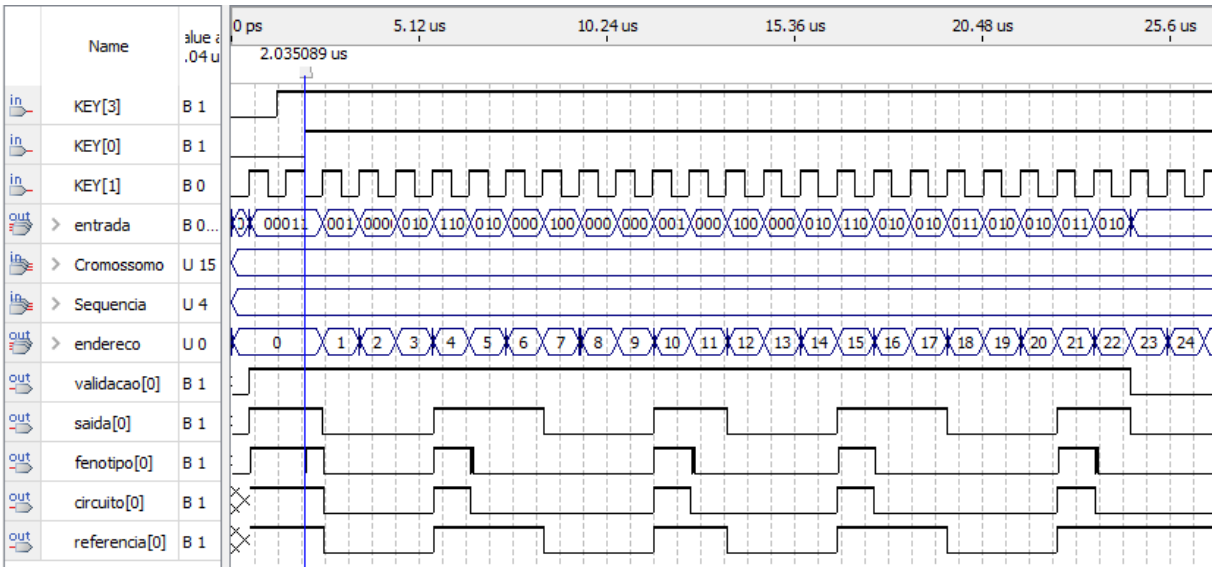


Figura III.20: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* de duas portas da Figura 4.19b.

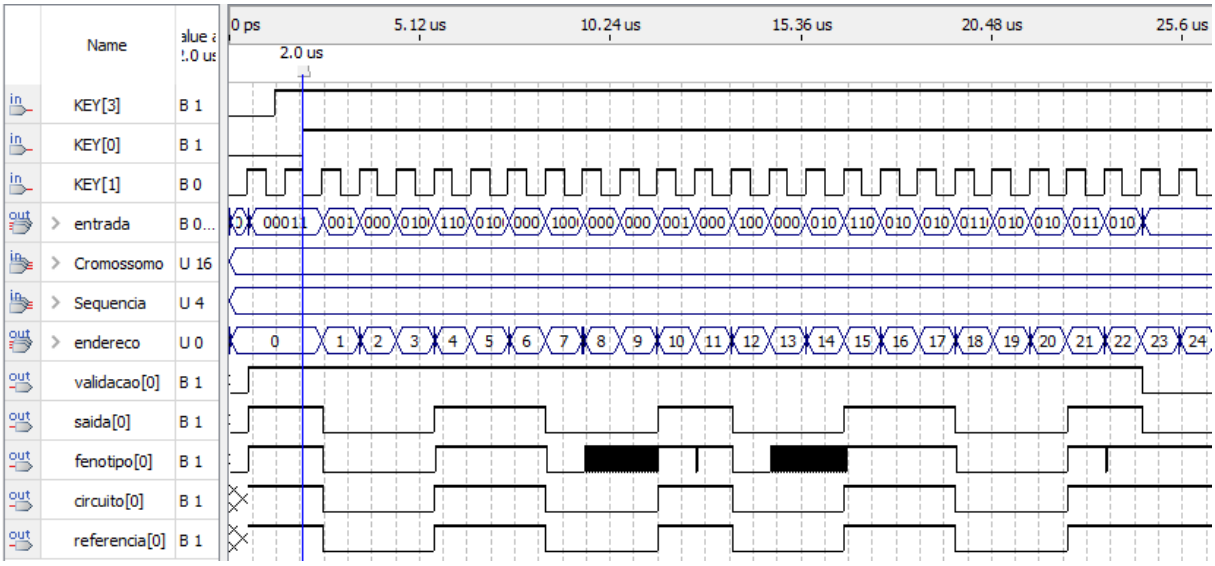


Figura III.21: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* de duas portas da Figura 4.19c.

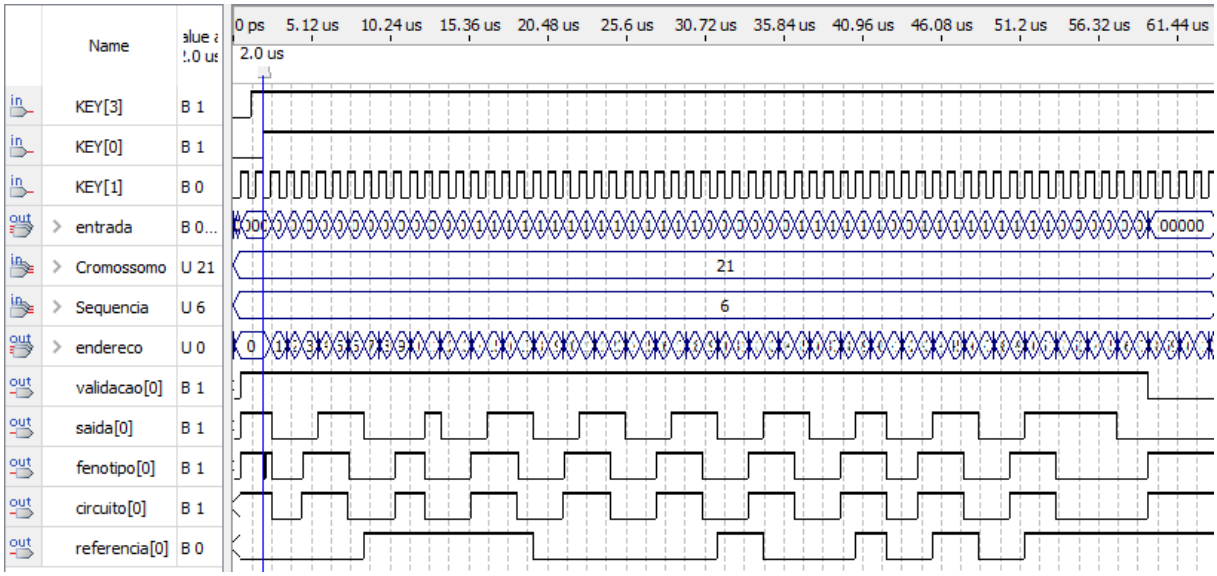


Figura III.22: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* BILBO da Figura 4.23a.

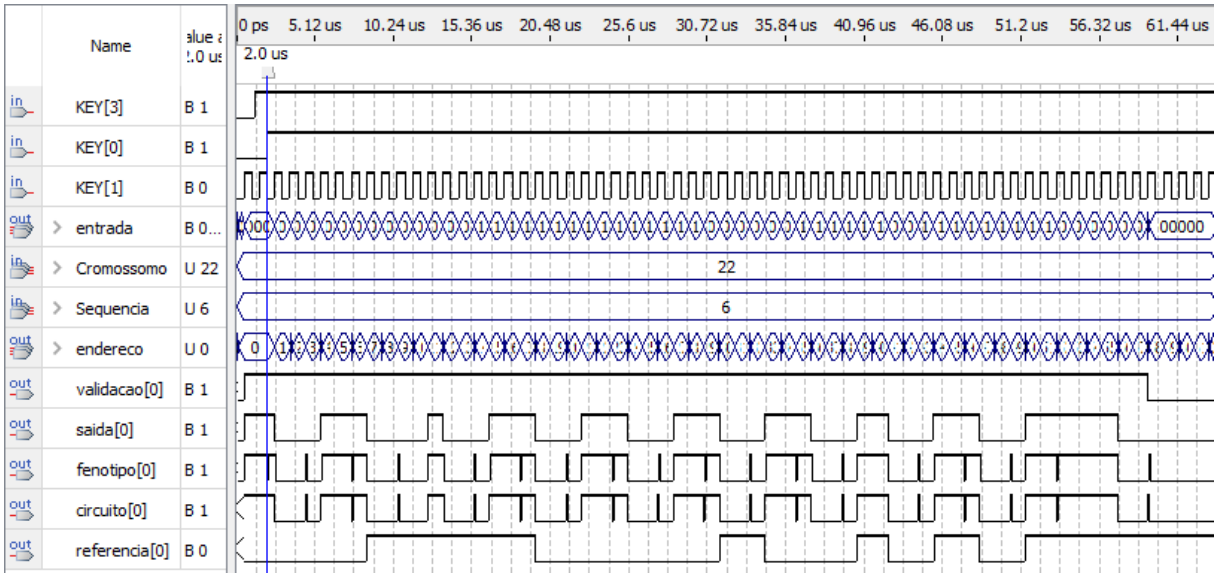


Figura III.23: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* BILBO da Figura 4.23b.

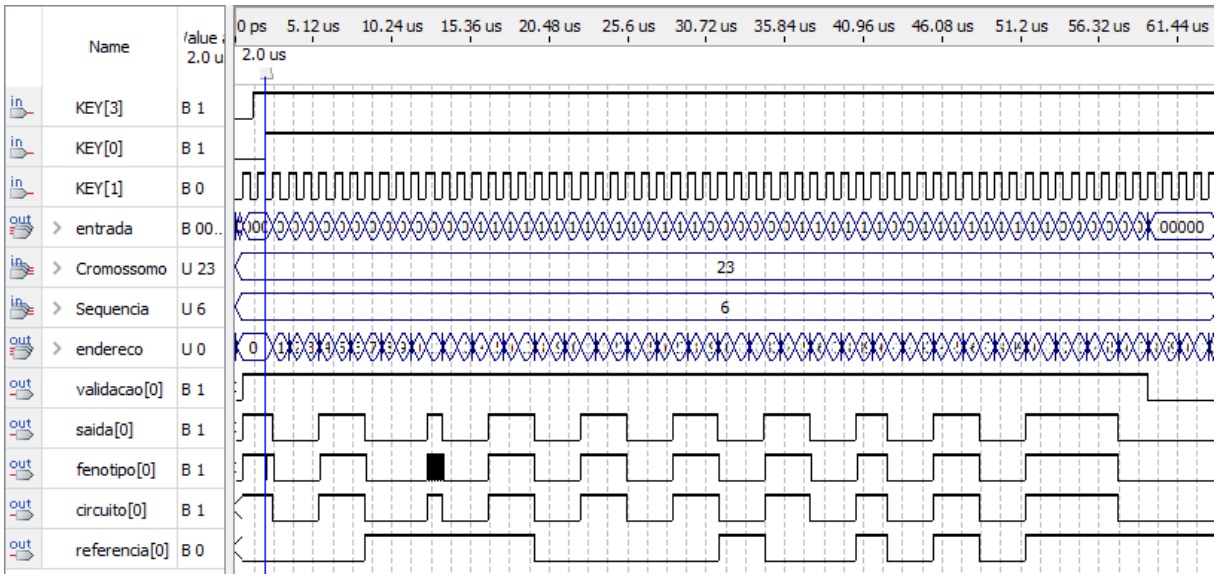


Figura III.24: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* BILBO da Figura 4.23c.

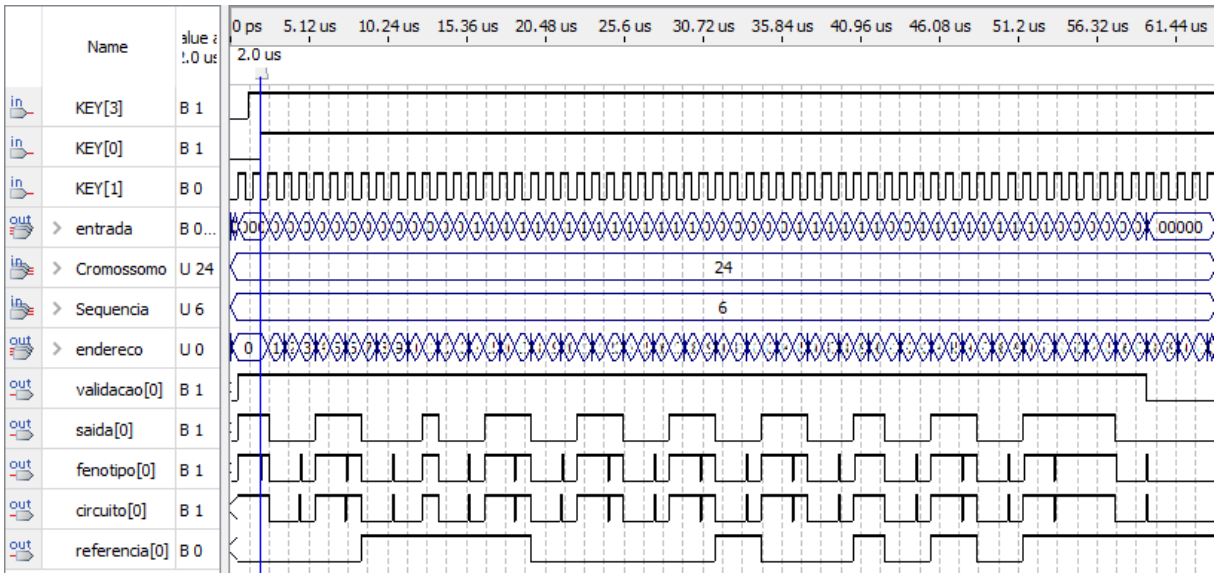


Figura III.25: Simulação *Modelsim* pela *Cyclone II* do circuito *latch* BILBO da Figura 4.23d.

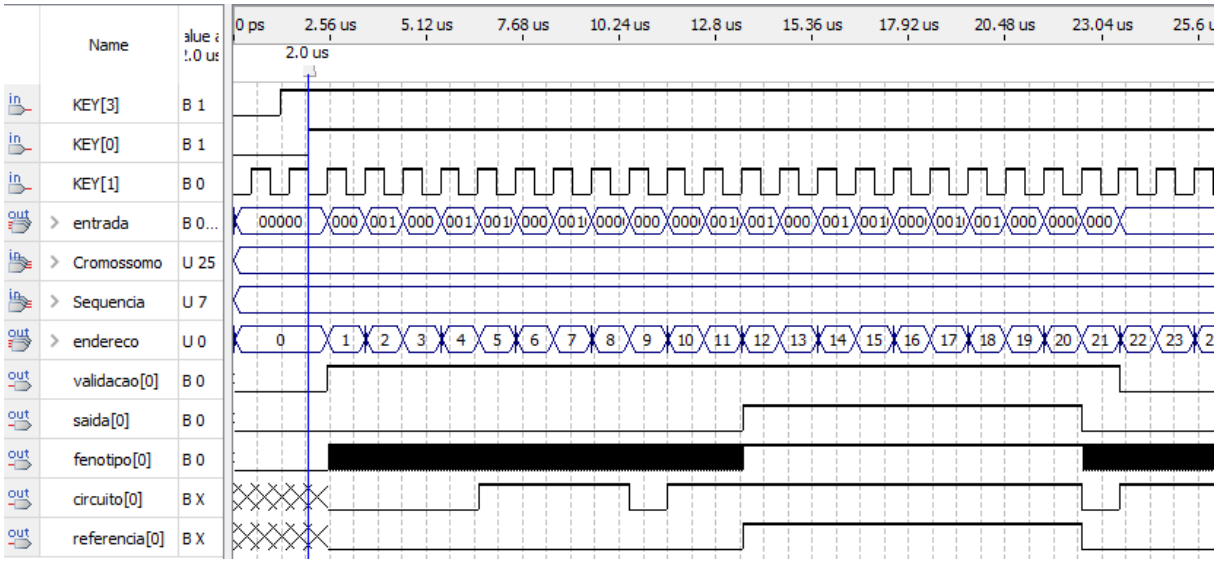


Figura III.26: Simulação *Modelsim* pela *Cyclone II* do circuito *flip-flop* D da Figura 4.32a.

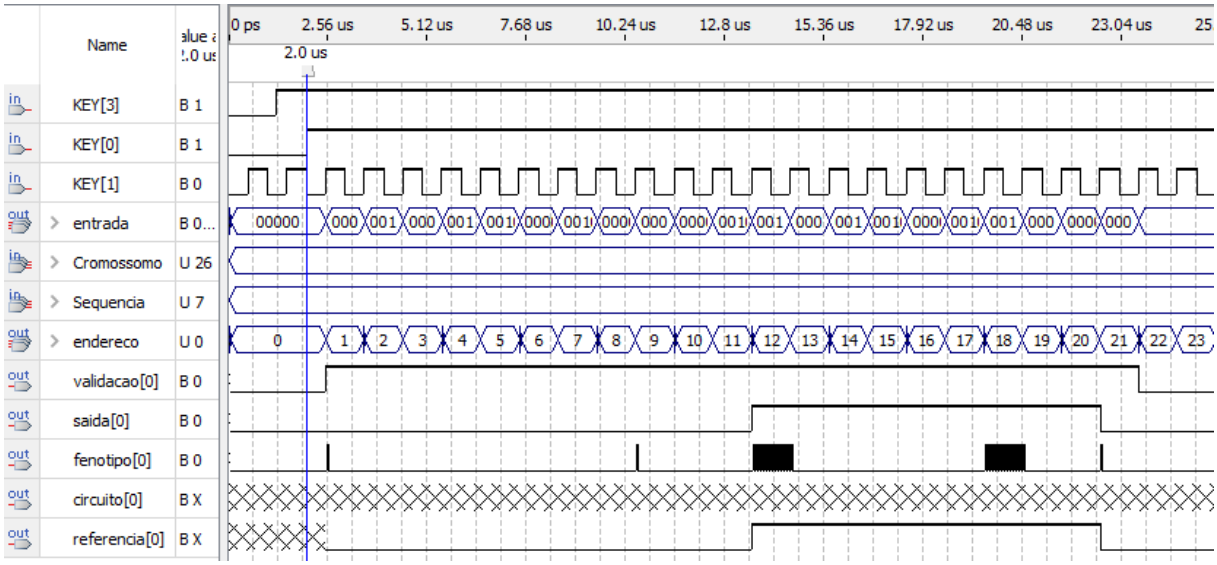


Figura III.27: Simulação *Modelsim* pela *Cyclone II* do circuito *flip-flop* D da Figura 4.32b.

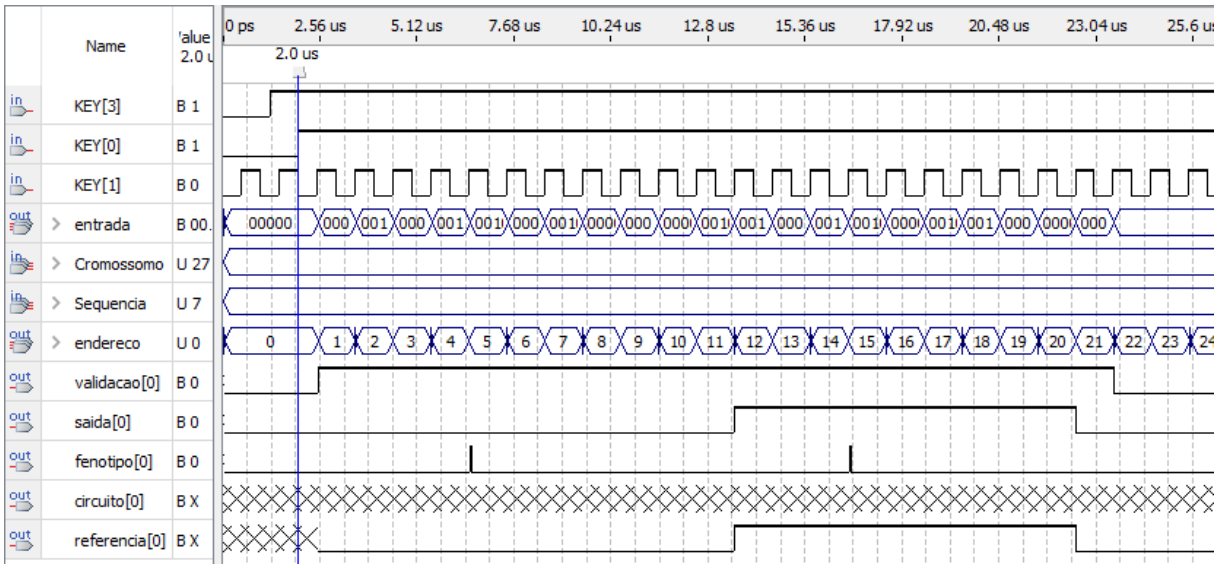


Figura III.28: Simulação *Modelsim* pela *Cyclone II* do circuito *flip-flop* D da Figura 4.32c.

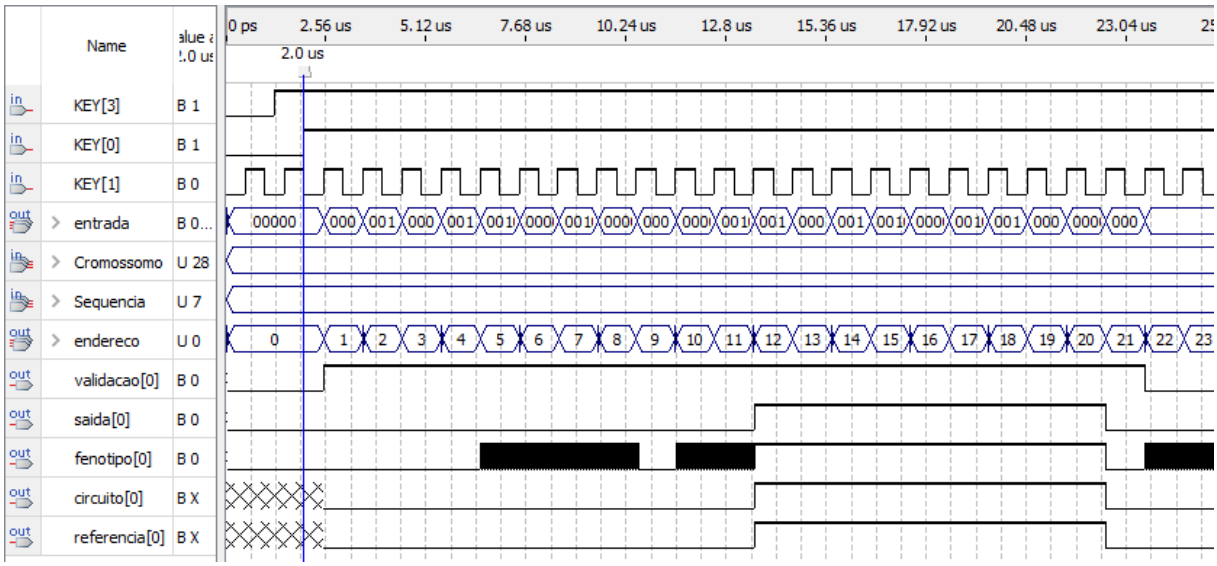


Figura III.29: Simulação *Modelsim* pela *Cyclone II* do circuito *flip-flop* D da Figura 4.32d.

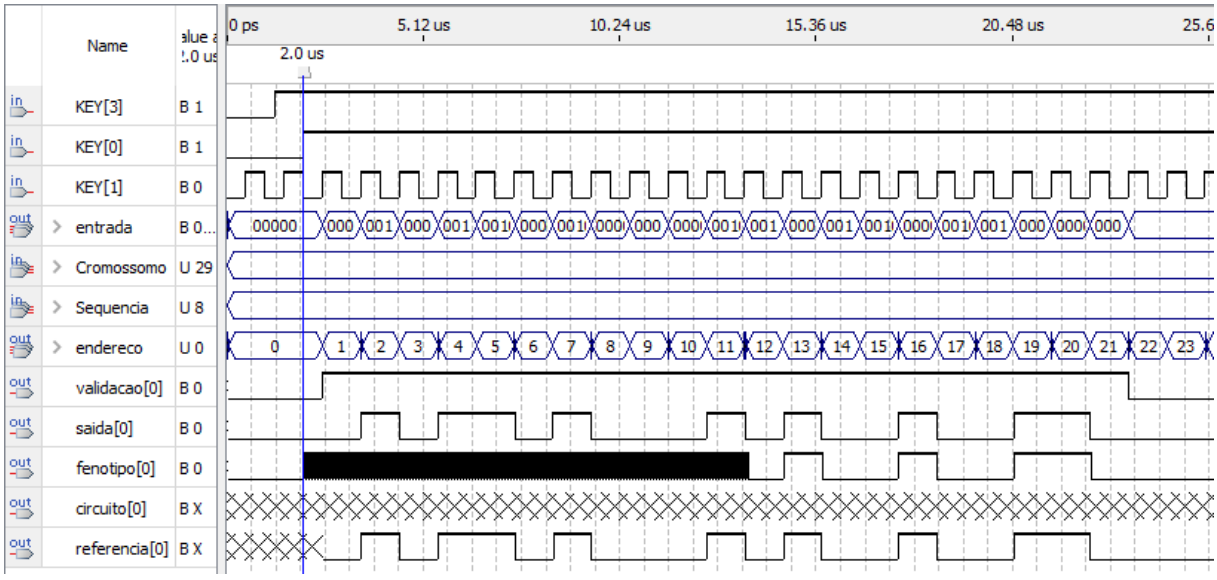


Figura III.30: Simulação *Modelsim* pela *Cyclone II* do circuito paridade-2 da Figura 4.33a.

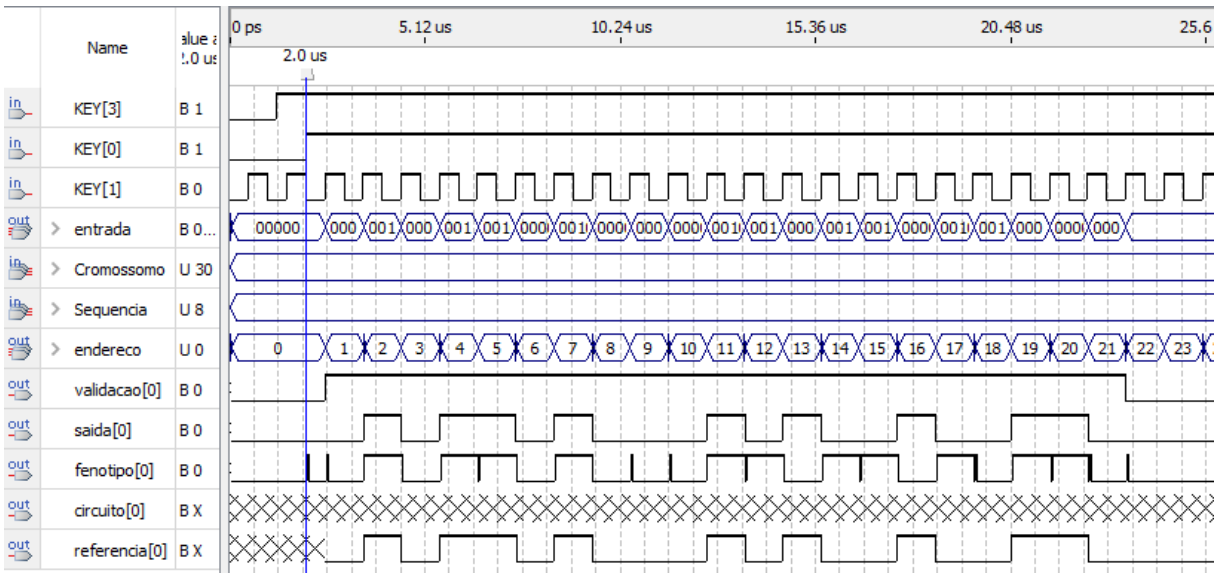


Figura III.31: Simulação *Modelsim* pela *Cyclone II* do circuito paridade-2 da Figura 4.33b.

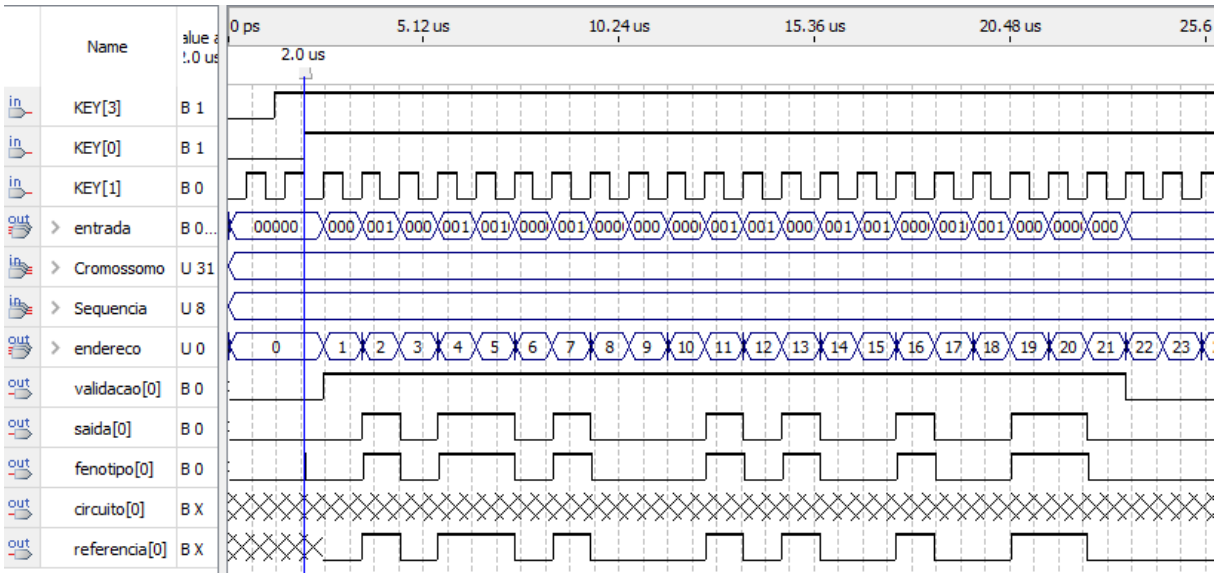


Figura III.32: Simulação *Modelsim* pela *Cyclone II* do circuito paridade-2 da Figura 4.33c.

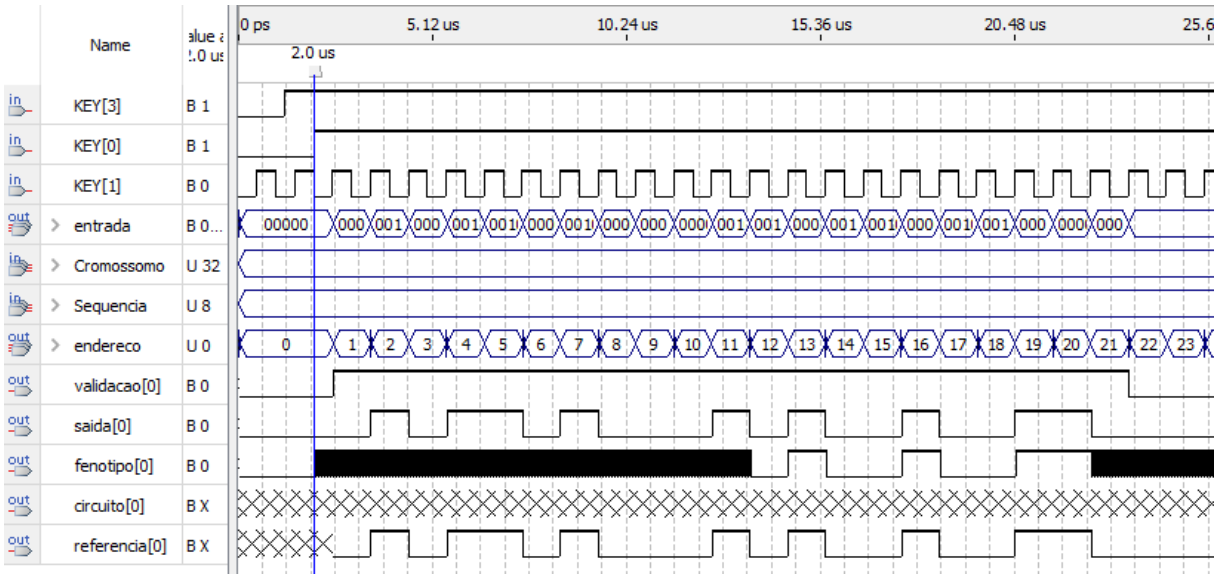


Figura III.33: Simulação *Modelsim* pela *Cyclone II* do circuito paridade-2 da Figura 4.33d.