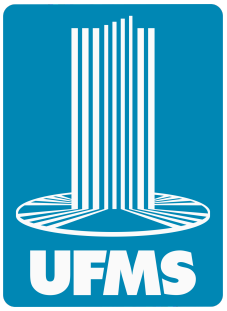


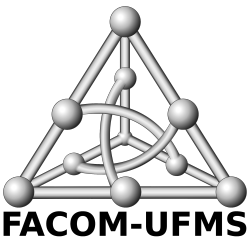
UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL

CIDADE UNIVERSITÁRIA

FACULDADE DE COMPUTAÇÃO

KAUAN LUCAS ALVES DE SOUSA

COMPUTAÇÃO BINÁRIA REVERSÍVEL



Campo Grande - MS
2024

KAUAN LUCAS ALVES DE SOUSA

COMPUTAÇÃO BINÁRIA REVERSÍVEL

Trabalho de Conclusão de Curso submetido à Faculdade de Computação (FACOM) da Universidade Federal de Mato Grosso do Sul (UFMS) como requisito parcial para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Milton Ernesto Romero Romero

Campo Grande - MS
2024

AGRADECIMENTOS

Primeiramente agradeço ao meu professor orientador Milton Ernesto Romero Romero por sua paciência, orientação e dedicação ao longo deste ano de 2024, onde juntos marcamos diversas reuniões, trocamos várias mensagens e me foi fornecido muito conhecimento para que eu me preparasse e obtivesse sucesso na confecção deste trabalho acadêmico.

Ademais, agradeço, também, aos professores Evandro Mazina Martins, Marco Aurélio Stefanes e Renato Porfirio Ishi por aceitarem fazer parte da banca avaliadora deste trabalho e pelo tempo e disponibilidade que será dedicado à avaliação dele.

Agradeço a todos os professores e colegas do curso de Engenharia de Computação e da UFMS que fizeram parte da minha formação acadêmica nesses 5 anos de graduação, onde nesse tempo tive a oportunidade de aprender muito sobre diversos assuntos, tanto disciplinares quanto gerais, indo até mesmo além dos limites acadêmicos.

Agradeço imensamente à minha família, em particular meu pai e minha mãe, por sempre me incentivarem a trilhar o caminho do bem e por nunca terem deixado faltar coisa alguma em casa, permitindo, assim, que eu pudesse me dedicar exclusivamente à minha formação.

RESUMO

Segundo Rolf Landauer o processamento de instruções convencionalmente ocorre de maneira irreversível, onde para cada instrução processada por um chip há dissipação de energia na forma de calor com o meio (vizinhança) que este está embutido. Por sua vez, isso altera a entropia do sistema de forma que ao fim de cada execução a entropia total sofre alteração, podendo, ao fim de todo o processamento, ficar diferente da apresentada inicialmente.

Portanto, surge a motivação da busca por alguma metodologia para diminuir tais perdas, onde para cada instrução processada seja possível alguma redução da energia utilizada para tais operações. Para isso foi feito com que a entropia fosse constante no final de todo o processamento através da utilização de bijeção.

Sobre circuitos construídos com base em portas lógicas, este trabalho propõe reduzir o impacto energético causado pelo processamento irreversível de instruções de forma a propor adaptações nas portas lógicas tradicionais para que estas possam continuar sendo utilizadas para os mesmos propósitos, mas que também suportem tecnologias de redução de gasto energético na forma de calor.

Uma vez implementada a propriedade de reversibilidade nas portas lógicas convencionais, com base em propriedades da termodinâmica, a energia consumida pelo sistema é diminuída.

Alcançado esse ponto é estimado que passos de processamento podem ser realizados de modo a não gerar calor ao usufruir do fato de que nenhuma energia é gasta em um processamento reversível perfeitamente isolado. Logo, o desejo de tornar esse feito real foi uma das motivações por trás deste trabalho.

Palavras-Chave: Processamento, Reversibilidade, Portas Lógicas, Energia, VHDL.

ABSTRACT

According to Rolf Landauer, instruction processing conventionally occurs irreversibly, where for each instruction processed by a chip there is energy dissipation in the form of heat with the environment (neighborhood) in which it is embedded. In turn, this changes the entropy of the system so that at the end of each execution the total entropy changes, and may, at the end of the entire processing, be different from that initially presented.

Therefore, the motivation arises to search for some methodology to reduce such losses, where for each instruction processed it is possible to reduce some of the energy used for such operations. To this end, the entropy was made constant at the end of the entire processing through the use of bijection.

Regarding circuits built based on logic gates, this work proposes to reduce the energy impact caused by the irreversible processing of instructions in order to propose adaptations in traditional logic gates so that they can continue to be used for the same purposes, but that also support technologies to reduce energy expenditure in the form of heat.

Once the reversibility property is implemented in conventional logic gates, based on thermodynamic properties, the energy consumed by the system is reduced.

Once this point is reached, it is estimated that processing steps can be performed in a way that does not generate heat by taking advantage of the fact that no energy is spent in a perfectly isolated reversible process. Therefore, the desire to make this feat real was one of the motivations behind this work.

Keywords: Processing, Reversibility, Logic Gates, Energy, VHDL.

SUMÁRIO

1 INTRODUÇÃO

De acordo com Landauer, considerada a temperatura estando próxima da temperatura ambiente, existe um limite mínimo teórico de aproximadamente $2{,}8 \times 10^{-21}$ Joules para se apagar 1 bit de informação [Landauer, 1961]. Seguida tal linha de raciocínio, há muita energia perdida em cada operação realizada por uma CPU convencional, onde nela há diversas portas lógicas que, atualmente, independentemente do número de entradas, produzem na maioria das vezes apenas uma saída. A busca por um método para reduzir tal perda é justificada e será um dos alvos deste trabalho a ser desenvolvido mais adiante através de propriedades físicas e matemáticas.

Segundo a Física, em um sistema isolado, partindo de um ponto A de estado inicial para um ponto B de estado final através de um processo reversível é possível fazer o caminho inverso, partindo de B e indo até A, através de um processo inverso. Nesses casos a variação de energia no sistema é zero e a entropia é constante, portanto não há consumo algum de energia ao fazer os processos direto e inverso cumulativamente [Halliday et al. 2016].

Exposto tal fato, quando trazida tal teoria para o contexto da computação, tais efeitos podem ser aplicados em processamentos de instruções, uma vez que, nas portas lógicas em uma CPU, se for considerado o estado inicial A (definido por **m** variáveis de estado, onde cada uma é o valor do m-ésimo sinal de entrada) como sendo os valores dos sinais do sistema antes da execução de uma instrução e o estado final B (definido por **n** variáveis de estado, onde cada uma é o valor do n-ésimo sinal de saída) como sendo os valores dos sinais do sistema depois da execução da mesma instrução, quando realizada a instrução de B para A teoricamente duas instruções seriam executadas, ao passo de que a energia consumida pelo CPU fosse zero e a entropia se mantivesse constante. Logo, se fez plausível a busca pela implementação de um conjunto universal de portas lógicas reversíveis na tentativa de se construir circuitos digitais reversíveis.

Os objetivos aqui serão modelar e implementar portas lógicas reversíveis, mostrar que é possível fazer circuitos lógicos reversíveis utilizando tais portas e estimar qual é o consumo de energia total de um sistema que faz uso de portas lógicas reversíveis para execução de instruções, onde ao final será comparado seu consumo com um sistema equivalente que não faz uso de portas lógicas reversíveis.

O mapeamento entre as entradas e saídas das portas foram postulados de forma a manter uma relação necessária e suficiente para que pudesse ser realizado processamento direto e inverso com base nas portas lógicas convencionais. Onde para cada porta estabelecida foram realizados testes com base em análise de consumo de energia e sinais de ondas gerados pelos próprios circuitos constituídos pelas portas lógicas reversíveis.

Durante todo o processo de implementação em código foi utilizada a linguagem de descrição de hardware VHDL [Pedroni, 2010] e mais algumas outras ferramentas para auxílio de visualização e obtenção das ondas.

Os circuitos que serão mostrados mais adiante nas modelagens e implementações irão exemplificar a ideia de processamento reversível e como este pode ser implementado para a obtenção do almejado consumo zero de energia, apesar de que, para efeitos práticos, tal trabalho não é suficiente e é necessário mais tempo e dedicação para se desenvolver métodos para aplicar tais ideias no mundo real. Mas, para uma fundamentação teórica, este documento supre tal necessidade demonstrando quais resultados podem ser obtidos com a computação binária reversível.

2 TRABALHOS RELACIONADOS

2.1 Portas Lógicas Quaternárias Reversíveis

Dentre os trabalhos publicados do curso de mestrado em Computação Aplicada da Universidade Federal de Mato Grosso do Sul (UFMS) está o Universal Set of Reversible Quaternary Logic Gates [Souza et al. 2021]. Nele, portas lógicas reversíveis aliadas à ideia de MVL (Multi-Valued Logic) são utilizadas e as definições seguem ideias similares às apresentadas neste trabalho.

2.2 Computação Reversível

Reversible Computing [Toffoli, 1980], que aborda conceitos de computação reversível e circuitos reversíveis com zero consumo energético foi uma das bases teóricas deste trabalho.

2.3 Computação Reversível Generalizada

A viabilidade de se propor transformações bijectivas para realizar computação reversível, utilizadas mais adiante neste trabalho, está presente no trabalho Generalized Reversible Computing [Frank, 2017].

2.4 Reversibilidade Lógica

A fundamentação de se implementar reversibilidade em uma CPU e sua eficiência energética é exibida no trabalho Logical Reversibility of Computation [Bennett, 1973].

2.5 Síntese Lógica Reversível

O trabalho A Beginning in the Reversible Logic Synthesis of Sequential Circuits [Thapliyal et al. 2005] relaciona circuitos sequenciais reversíveis e computação quântica.

2.6 Computação Quântica

Superconducting Qubits: Current State of Play explora a ideia dos circuitos quânticos supercondutores com a utilização de qubits supercondutores [Kjaergaard et al. 2020].

Circuitos quânticos executados em ambientes controlados possuem propriedades reversíveis com operações unitárias, fazendo com que a energia gasta para os processamentos seja reduzida. O que acabou sendo uma das bases de pesquisa auxiliar neste trabalho.

2.7 Linguagens de Programação Reversíveis

O trabalho Reversible Computing from a Programming Language Perspective aborda computação reversível também, mas possui um estudo direcionado a linguagens de programação reversíveis onde as informações não são perdidas e operações destrutivas não são utilizadas [Glück et al. 2023].

3 CONCEITOS BÁSICOS

3.1 Bijectividade

Bijectividade é uma propriedade de uma operação cuja esta se faz injetora e sobrejetora cumulativamente.

Uma função f: $A \rightarrow B$ é injetora somente se valores diferentes de A são levados a valores diferentes em B. Logo, não há redundância no domínio.

Matematicamente, se

$$f(x1) = f(x2) \tag{1}$$

Se faz verdadeiro, então, que

$$x1 = x2 \tag{2}$$

Uma função f: $A \rightarrow B$ é sobrejetora somente se para cada elemento ‘b’ do contradomínio houver pelo menos um elemento ‘a’ do domínio tal qual a seguinte relação se faz verdadeira:

$$f(a) = b \tag{3}$$

Logo, o contradomínio deve coincidir com o conjunto imagem da função em questão [Guidorizzi, 2001].

3.2 Função Inversa

Dado um domínio, um contradomínio e uma operação qualquer que leva os elementos do domínio até o contradomínio, a operação que leva o contradomínio exatamente no mesmo domínio é chamada de inversa.

Logo, para que uma operação tenha inversa, acaba sendo necessário que a função em questão seja bijetora, havendo um mapeamento 1 para 1 entre as entradas e saídas de um sistema seja ele qual for [Guidorizzi, 2001].

3.3 Portas Lógicas e Álgebra de Boole

Portas lógicas são dispositivos que processam sinais digitais. Com elas é possível fazer a construção de circuitos lógicos, possibilitando fazer modelagem de

funções lógicas. Logo, como em qualquer função, há o processamento de entradas e produção de suas respectivas saídas.

Atrelada às portas lógicas existe a chamada Álgebra de Boole, que descreve como as variáveis lógicas podem ser utilizadas para realizar operações lógicas, que por sua vez são implementadas nas já mencionadas portas lógicas [Floyd, 2012].

Neste trabalho serão apresentadas, utilizadas e adaptadas algumas das portas lógicas definidas a seguir com alterações necessárias para que a reversibilidade de cada porta seja alcançada.

Seguem algumas operações definidas na Álgebra de Boole:

- AND:

Figura 1 - Porta AND.

A	B	OUT
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 1 - Tabela verdade Porta AND.

- OR:

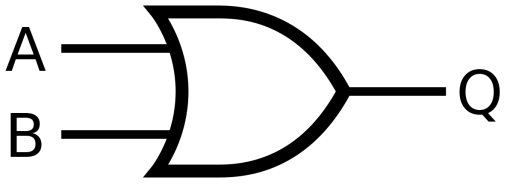


Figura 2 - Porta OR.

A	B	OUT
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 2 - Tabela verdade Porta OR.

- NOT:

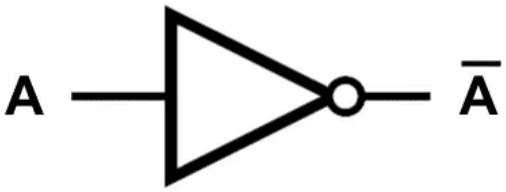


Figura 3 - Porta NOT (Inversora)

A	OUT
0	1
1	0

Tabela 3 - Tabela verdade Porta NOT

- NAND:

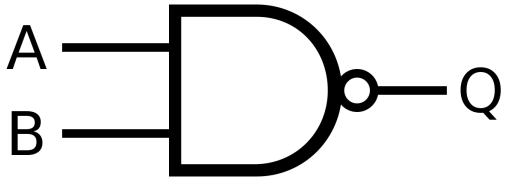


Figura 4 - Porta NAND.

A	B	OUT
0	0	1
0	1	1
1	0	1
1	1	0

Tabela 4 - Tabela verdade Porta NAND.

- NOR:

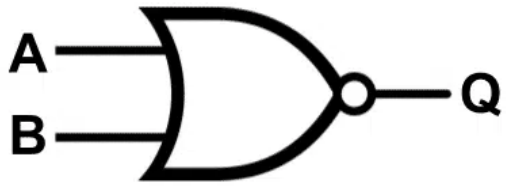


Figura 5 - Porta NOR.

A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

Tabela 5 - Tabela verdade Porta NOR.

3.4 Entropia e Processos Adiabáticos

A entropia é uma grandeza física usada para medir o grau de desordem de um sistema. Nele, quanto maior for a quantidade de calor transmitida ou recebida, maior será a variação de sua entropia.

No que rege tipos de processos possíveis para um sistema existem processos reversíveis e irreversíveis.

Nos processos reversíveis, por conta da volta ao estado inicial, a variação de entropia total (sistema mais meio) é zero e, portanto, tem-se entropia total constante. Dado um processo reversível, uma vizinhança e um sistema com temperatura constante isolados durante todo o processo, a relação de constância de entropia pode ser visualizada de acordo com a seguinte equação [Halliday et al. 2016]:

$$\Delta Ssis + \Delta Smeio = 0 \quad (4)$$

Ademais, a variação de entropia do sistema é dada por [Halliday et al. 2016]:

$$\Delta Ssis = \frac{Q}{T} \quad (5)$$

Onde T é a temperatura absoluta do sistema, Q é o calor trocado com a vizinhança e

$$\Delta Ssis \neq 0 \quad (6)$$

Nos processos irreversíveis a possibilidade de se voltar ao estado inicial é inexistente, o que acarreta em uma variação de entropia total diferente de zero. Logo, a entropia inicial do sistema não é a mesma que a entropia ao final do processo e há perdas ou ganhos de energia, ocasionados, por exemplo, por trocas de calor com o meio em que o sistema está embutido, então [Halliday et al. 2016]:

$$\Delta Ssis + \Delta Smeio > 0 \quad (7)$$

Processos que não permitem troca de calor de um sistema com o seu meio externo são chamados de processos adiabáticos.

3.5 Princípio de Rolf Landauer

Em 1961 o físico alemão Rolf William Landauer descobriu o chamado Princípio de Landauer, que propõe que em toda e qualquer operação de registro ou remoção de informação (bits) é gerado um gasto mínimo de energia dissipada, na forma de calor, do sistema e, consequentemente, a entropia dele é alterada.

O princípio também afirma que tal energia é proporcional à temperatura que o sistema está operando e seu comportamento é dado pela seguinte expressão:

$$E \geq k * T * ln(2) \quad (8)$$

Onde k é a constante de Boltzmann e T é a temperatura absoluta em questão.

3.6 Quartus Prime

Software de prototipação e simulação de sistemas digitais. Serão usadas ferramentas para visualização de sinais em forma de ondas e estimação de consumo energético presentes no software.

3.7 Eda Playground

Plataforma online usada para programação em VHDL de designs e testbenches.

3.8 Linguagens de descrição de hardware e VHDL

Diferente das linguagens de programação, programas feitos em linguagens de descrição de hardware não geram código. Ao invés disso, o produto da compilação (síntese) de tais componentes geram circuitos digitais feitos a base de portas lógicas, sendo essa uma das principais características para sua utilização na confecção deste trabalho no que rege a implementação.

Por conseguinte, dentre as diversas opções de linguagens desse tipo está o VHDL (Very High Speed Integrated Circuit Hardware Description Language), que é uma das mais utilizadas para o propósito.

Em VHDL [Pedroni, 2010] se pode modelar, em termos de código, a estrutura e o comportamento dos circuitos desejados, onde ao fim de todo o processo há a possibilidade de sintetizar e simular os circuitos modelados [Pedroni, 2010].

4 METODOLOGIA

Em algumas das portas lógicas convencionais o número de variáveis de estado inicial é diferente do número de variáveis de estado final assim como exemplificado a seguir.

4.1 Exemplos das portas AND e OR convencionais

4.1.1 Exemplo porta AND convencional

Como bem conhecido, uma porta AND convencional possui uma relação de 2 entradas para 1 saída. Isso implica que os possíveis resultados da porta lógica, na lógica de boole [Floyd, 2012], sejam repetidos, dado que há 4 possibilidades de combinação de entradas (**0, 0**; **0, 1**; **1, 0** e **1, 1**) e apenas duas possibilidades de combinação de saídas (**0** e **1**), ocasionando a repetição do valor lógico ‘0’ na saída. Logo, esta porta viola a propriedade de injectividade necessária, portanto não é bijectiva e consequentemente não é reversível.

4.1.2 Exemplo porta OR convencional

Assim como no caso da porta AND, a porta OR convencional possui uma relação de 2 entradas e 1 saída, ocasionando a repetição do valor lógico ‘1’ na saída, que também implica na violação da propriedade de injectividade. Logo, não possui suporte a reversibilidade.

4.2 Obtenção das portas AND e OR reversíveis

Para que seja possível a aplicação de reversibilidade a uma CPU decomposta em portas lógicas a quantidade **m** de variáveis de estado do estado inicial (sinais antes de passar pela porta lógica) deve ser igual à quantia **n** de variáveis de estado do estado final (sinais depois de passar pela porta lógica) a fim de que nenhum joule seja perdido nos processos.

Ademais, além de uma relação **m** para **n** (com **m = n**) é necessário que a propriedade de bijeção esteja presente no mapeamento das portas lógicas, onde para cada elemento do conjunto de entrada haja apenas uma única correspondência na saída e vice versa. Supridas as condições o resultado esperado é uma porta lógica reversível.

Pensando no funcionamento padrão de uma porta lógica AND segue a tentativa de fazer ela reversível adicionando apenas uma saída auxiliar, gA (garbage A), à porta:

A	B	AandB	gA
0	0	0	0
0	1	0	1
1	0	0	X
1	1	1	0

Tabela 6 - Tentativa de porta rAND com 1 auxiliar na saída.

Observada a Tabela 6, apesar de haver a mesma quantia de entradas e saídas, houve uma violação da propriedade de bijeção, onde no lugar de X não há valor algum válido tal que haja concordância com as propriedades de injeção e sobrejeção.

De maneira análoga segue a tentativa com a porta OR:

A	B	AorB	gA
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	X

Tabela 7 - Tentativa de porta rOR com 1 auxiliar na saída.

Observada a Tabela 7 houve violação, assim como na Tabela 6, de injeção e sobrejeção.

Portanto, uma relação de 2 entradas para 2 saídas é insuficiente para modelagem das portas rAND e rOR desejadas.

Para a realização dos eventuais testes e lógicas requisitadas foram definidas as seguintes portas lógicas reversíveis (as três primeiras colunas são as entradas da porta e as últimas 3 são as saídas):

Ancillary	A	B		gA	AandB	gB
0	0	0	➡	0	0	0

0	0	1	➡	0	0	1
0	1	0	➡	1	0	0
0	1	1	➡	0	1	1
1	0	0	➡	0	1	0
1	0	1	➡	1	0	1
1	1	0	➡	1	1	0
1	1	1	➡	1	1	1

Tabela 8 - Porta rAND.

Ancillary	**A**	**B**		**AorB**	**gA**	**gB**
0	0	0	➡	0	0	0
0	0	1	➡	1	0	0
0	1	0	➡	1	0	1
0	1	1	➡	1	1	0
1	0	0	➡	0	0	1
1	0	1	➡	0	1	0
1	1	0	➡	0	1	1
1	1	1	➡	1	1	1

Tabela 9 - Porta rOR.

Estas duas portas propõem fazer as operações lógicas AND e OR (respectivamente) de maneira direta e inversa, de modo que se for desejado executar a operação inversa basta fazer com que o bit de entrada Ancillary, sinal auxiliar de entrada, seja ‘1’. De maneira análoga, quando desejado executar a operação direta, Ancillary deve ser ‘0’. Como resultado, sinais auxiliares de saída gA e gB (garbage B), que somente serão utilizados ao fazer a operação inversa, foram

acrescentados para que a propriedade de bijectividade fosse mantida com uma relação de 3 entradas e 3 saídas.

Vale ressaltar que a escolha de como o mapeamento é definido não é importante, pois para que haja reversibilidade basta que a escolha respeite a propriedade de bijectividade e tenha uma relação de **m** entradas e **m** saídas.

Para que um conjunto universal de portas lógicas reversíveis pudesse ser obtido, foi, também, necessário a adesão da porta inversora NOT no projeto. Não houve necessidade de fazer alteração alguma nessa porta porque, como citado anteriormente, se houver um mapeamento **m** para **m** onde se faz real a propriedade bijetora, esta já pode ser considerada reversível e como na NOT há apenas uma entrada e uma saída esta já faz uso de reversibilidade. Logo, com o conjunto rAND, rOR e NOT os trabalhos puderam ser elaborados de forma que qualquer operação digital pudesse ser efetuada como uma composição de operações envolvendo esse conjunto de portas lógicas reversíveis.

Foi realizada a síntese dos circuitos em cima das tabelas 8 e 9 para que expressões lógicas fossem obtidas para cada uma das saídas das portas rAND e rOR. E, como exemplos de aplicação, foram obtidas implementações a nível de portas lógicas (gate-level) de um somador completo de 1 bit reversível (rFull_Adder_1) e de um Flip-Flop Set-Reset reversível (rFF_SR) inteiramente compostos por uma combinação do conjunto de portas lógicas reversíveis já citado anteriormente. Tais trabalhos, assim como os resultados das simulações dos códigos (implementação code-level) em VHDL, serão apresentados na seção IMPLEMENTAÇÃO E RESULTADOS deste documento. Todos os códigos se encontram nos apêndices.

Depois de feitas as implementações, foram realizadas duas instruções de soma de 1 bit de maneira direta e reversa. Nelas, foi utilizada uma outra implementação de somador de 1 bit em conjunto da ideia de salvar contextos a fim de voltar para o estado imediatamente anterior ao da respectiva operação. Portanto, para cada instrução executada foi realizada imediatamente sua inversa a fim de diminuir o consumo de energia e manter a entropia o mais constante possível.

Para que pudesse ser consultada e comparada a redução de energia nas execuções foi utilizado o software Intel Quartus Prime [Intel Corporation, 2020]. Testes e simulações foram feitas para analisar o consumo energético das portas construídas.

5 IMPLEMENTAÇÃO E RESULTADOS

5.1 Gate-level

Para a construção dos esquemas dos circuitos em gate-level foram realizados desenhos técnicos bidimensionais assistidos pelo software AutoCAD [Autodesk, 2023]. Seguem os resultados obtidos em cada passo.

5.1.1 Porta rAND

As expressões lógicas obtidas com a síntese da porta, Álgebra de Boole e a Tabela 8 foram:

$$gA = anc * B + A * \neg B \quad (9)$$

$$A * B = anc * \neg B + A * B \quad (10)$$

$$gB = B \quad (11)$$

Com a finalidade de que fossem obtidos os sinais iniciais A e B a partir dos sinais de saída da operação direta bastou serem conectados tais sinais a uma outra porta reversível equivalente à utilizada assim como sucede o esquema exemplo:

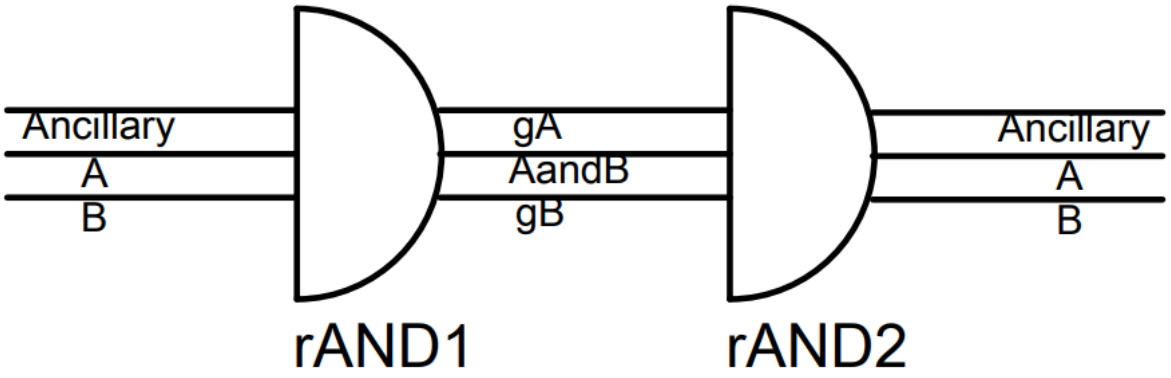


Figura 6 - Porta rAND.

A implementação em um domínio quaternário é possível fazendo a extensão das portas utilizando MVL [Souza et al. 2021].

5.1.2 Porta rOR

O mesmo procedimento efetuado na Porta rAND se mostrou suficiente para a rOR. Seguem, respectivamente, as expressões e o esquema exemplo com base na Tabela 9:

$$A + B = \neg anc * B + A * B + \neg anc * A \quad (12)$$

$$gA = anc * B + A * B + anc * A \quad (13)$$

$$gB = anc * \neg B + anc * A + \neg B \quad (14)$$

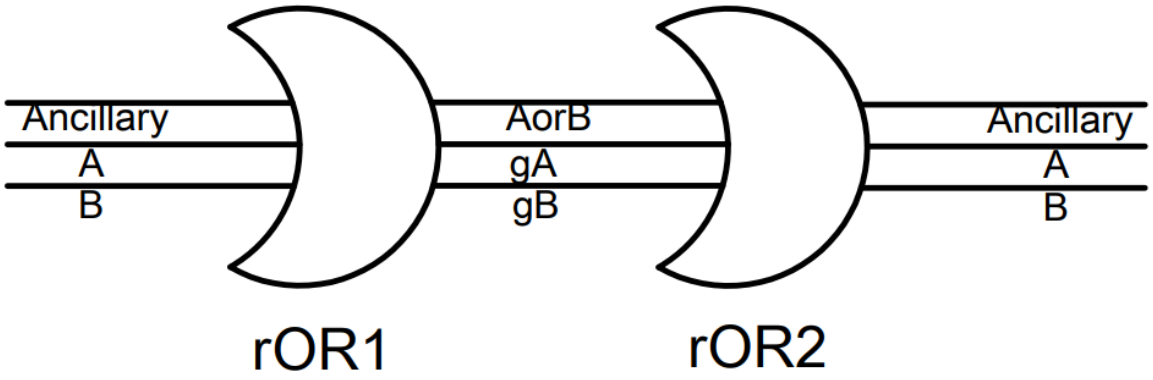


Figura 7 - Porta rOR.

Assim como na porta rAND, a implementação em um domínio quaternário é possível fazendo a extensão das portas utilizando MVL [Souza et al. 2021].

5.1.3 Circuito rFull_Adder_1

Um esquema foi construído inteiramente com base no conjunto de portas lógicas rAND, rOR e NOT. Para auxiliar no entendimento do diagrama, duas linhas foram esboçadas, vermelha e azul. À esquerda da linha vermelha se encontra a parte direta do processamento; à direita dela se encontra a parte inversa. Acima da linha azul se encontra a parte referente ao processamento da soma, onde S é um dos sinais de saída desejados; abaixo dela se encontra a parte referente ao processamento do carry out, onde Cout é um dos sinais de saída desejados:

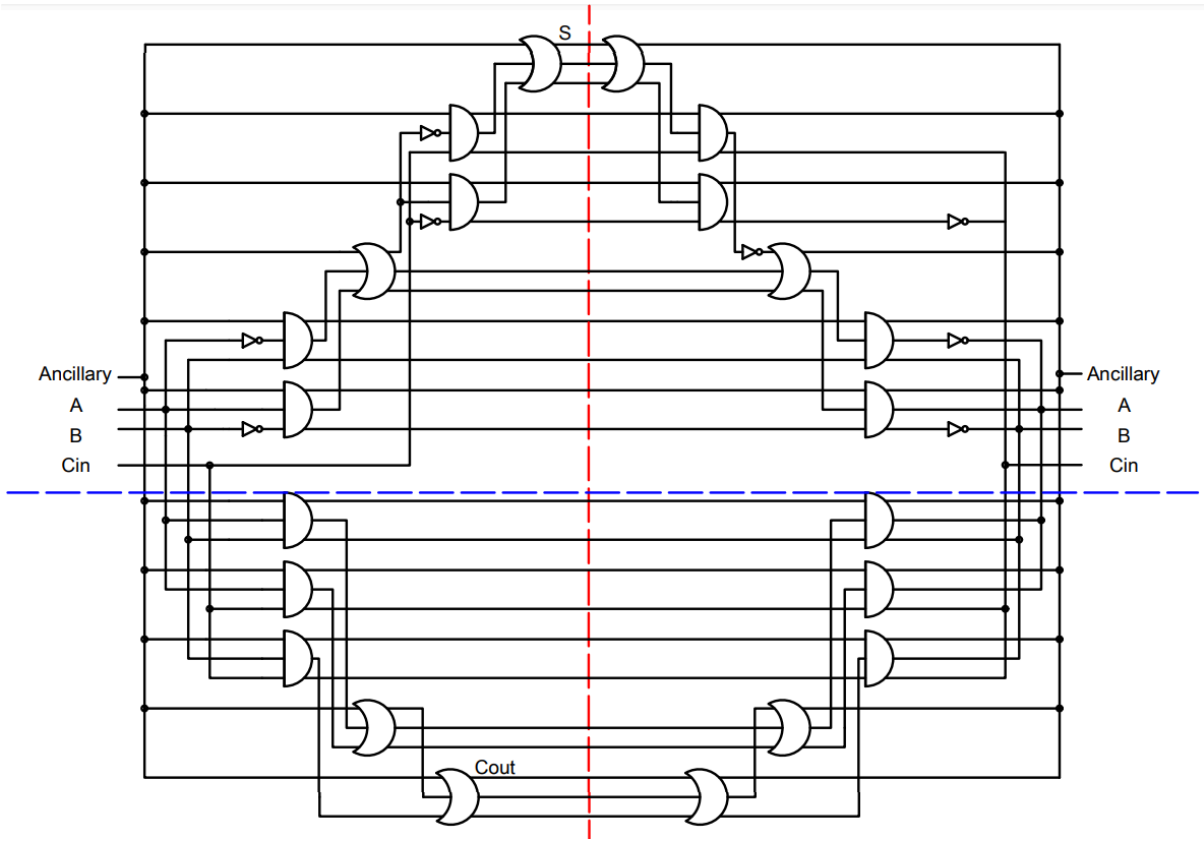


Figura 8 - Somador Completo de 1 bit Reversível (rFull_Adder_1).

5.1.4 Circuito rFF_SR

Um esquema foi construído inteiramente com base no conjunto de portas lógicas rOR e NOT. Nele, Q e ¬Q representam as saídas diretas do Flip-Flop e, se desejado, a continuação do circuito faz jus à obtenção dos sinais Ancillary, S e R iniciais, saídas inversas do Flip-Flop:

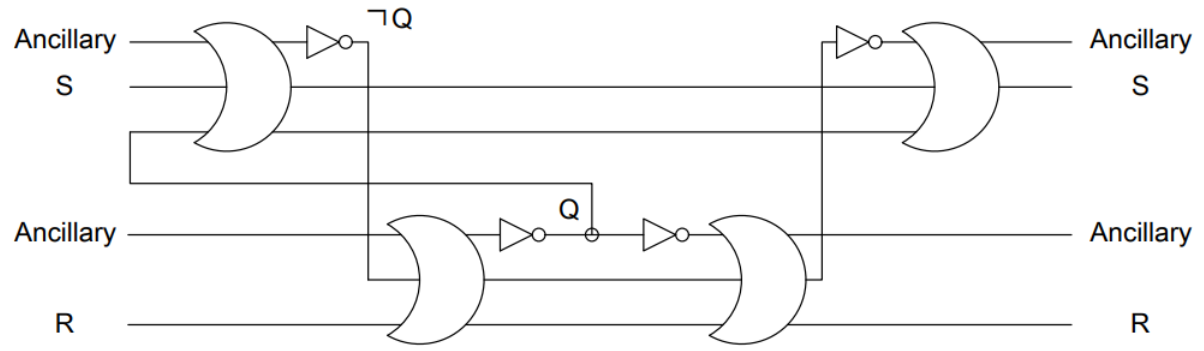


Figura 9 - Flip-Flop Set-Reset Reversível (rFF_SR).

5.2 Code-level

Para a implementação e simulação das portas lógicas reversíveis em código, foram utilizados o software Quartus Prime [Intel Corporation, 2020] e a plataforma EDA Playground [2024] em colaboração com a linguagem de descrição de hardware VHDL [Pedroni, 2010].

Dentro do EDA Playground [2024] foram feitos os códigos em VHDL [Pedroni, 2010] inicialmente e posteriormente simulados no Quartus Prime [Intel Corporation, 2020] para a consulta de consumo de energia no decorrer dos passos de processamento de cada implementação gate-level idealizada.

Os códigos das portas lógicas reversíveis rOR e rAND foram codados utilizando arquiteturas sequenciais, onde neles foram modeladas as possíveis combinações de entradas e saídas de acordo com as tabelas 8 e 9.

Ademais, para a realização de testes para confirmar as funcionalidades das implementações foram escritos, também em VHDL [Pedroni, 2010], testbenches para os propósitos (respectivamente, tb_rOR e tb_rAND) e seguidas as mesmas ideias apresentadas anteriormente a respeito de como fazer processamento direto e inverso de uma porta lógica.

Simulados na plataforma EDA Playground [2024] foi possível a visualização das respectivas ondas, cujas quais foram obtidas assim como seguem.

5.2.1 Porta rAND

Simulado o funcionamento da porta, embutidas as entradas VinA e VinB foram obtidas as seguintes ondas:

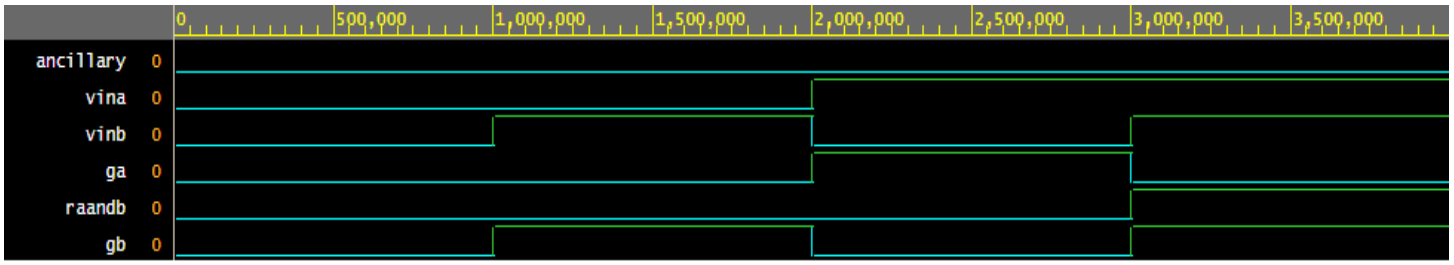


Figura 10 - Simulação no EDA Playground da porta rAND (código rAND).

Simuladas a ida e a volta (round trip) foram obtidas as seguintes ondas:

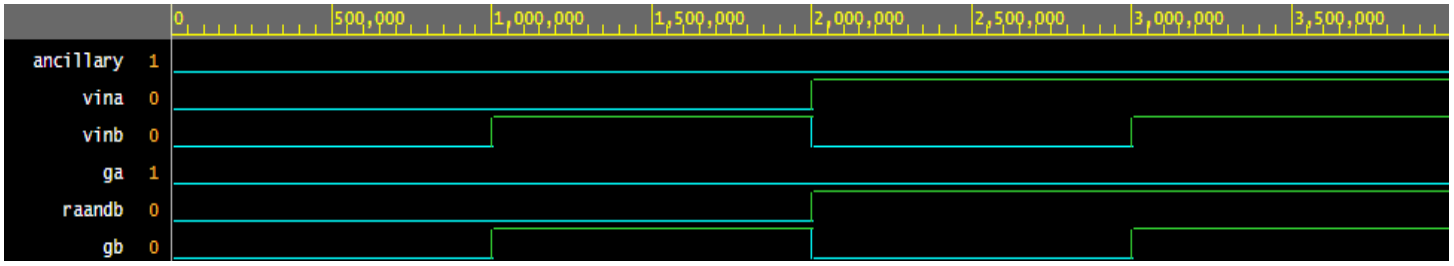


Figura 11 - Simulação no EDA Playground da porta rAND - processamento direto (código rAND_round_trip).

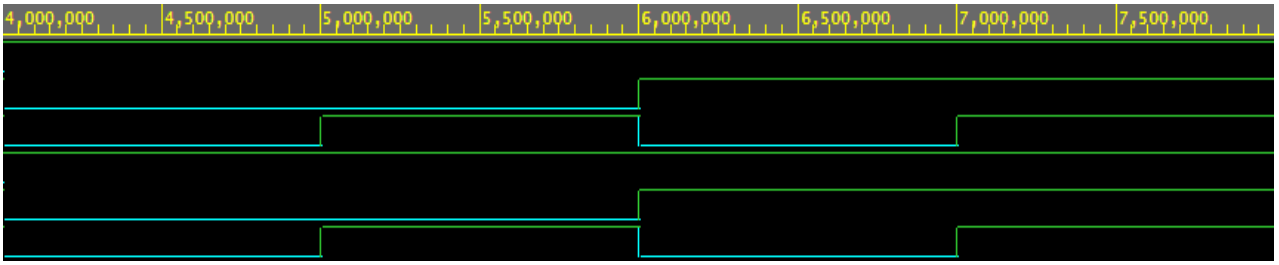


Figura 12 - Simulação no EDA Playground da porta rAND - processamento inverso (código rAND_round_trip).

5.2.2 Porta rOR

Simulado o funcionamento da porta, embutidas as entradas VinA e VinB foram obtidas as seguintes ondas:

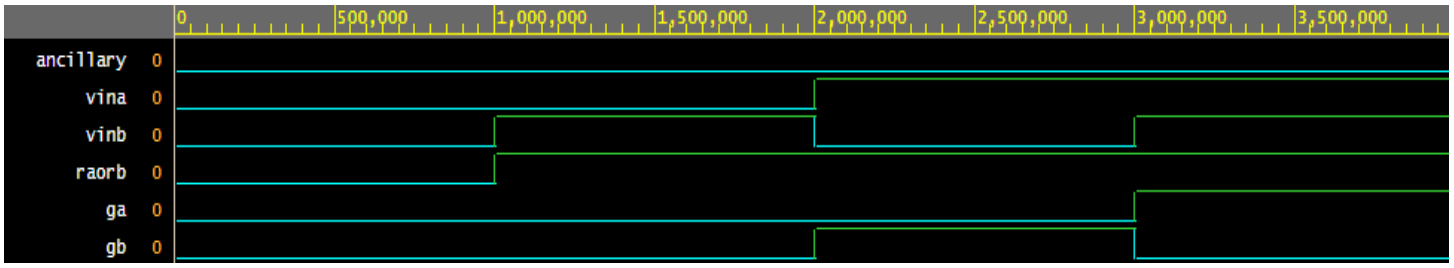


Figura 13 - Simulação no EDA Playground da porta rOR (código rOR).

Simuladas a ida e a volta (round trip) foram obtidas as seguintes ondas:

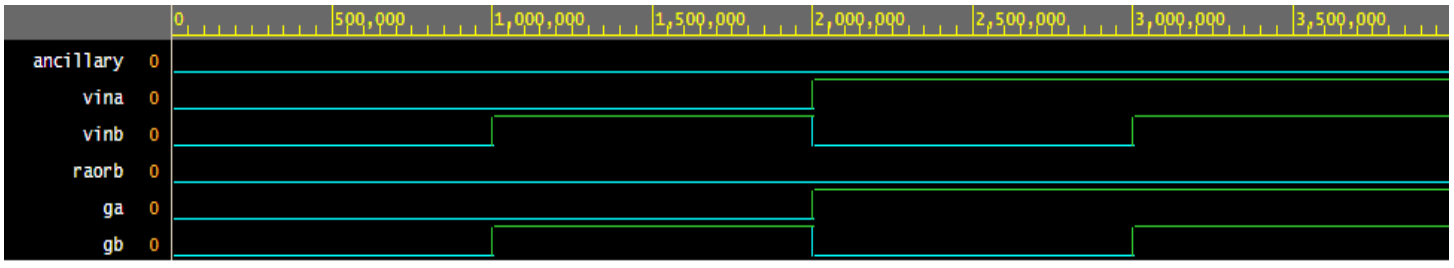


Figura 14 - Simulação no EDA Playground da porta rOR - processamento direto (código rOR_round_trip).

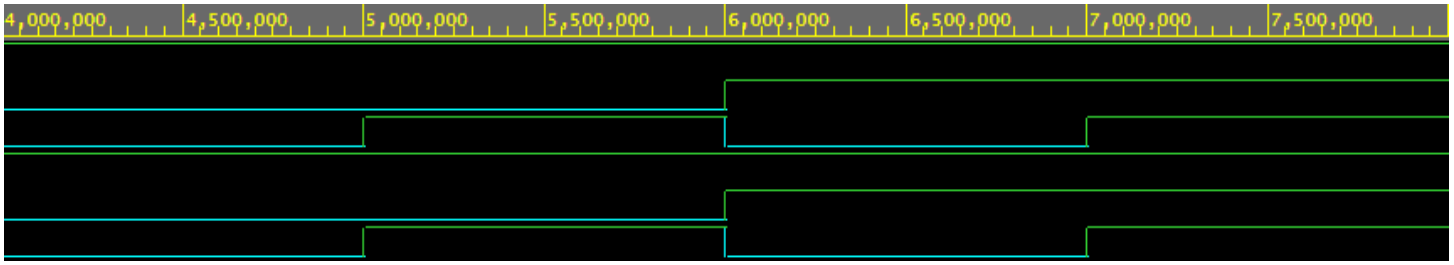


Figura 15 - Simulação no EDA Playground da porta rOR - processamento inverso (código rOR_round_trip).

5.2.3 Circuito rFull_Adder_1

Para o somador completo de 1 bit reversível foram feitas duas diferentes versões plausíveis com o mesmo comportamento reversível, onde a primeira está contida no código rFull_Adder_1_round_trip e a outra está contida no código Direct_Inverse_Sum.

Ambas as implementações fazem uso dos códigos dos rOR e rAND já mencionados, assim como também foram construídas embasadas no funcionamento convencional do somador completo de 1 bit:

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabela 10 - Tabela verdade Full Adder (1 bit) convencional.

A implementação contida no código rFull_Adder_1 foi escolhida para testes.

Assim como feito para as portas rOR e rAND, para a realização de testes para confirmar as funcionalidades das implementações foi escrito, também em VHDL [Pedroni, 2010], um testbench para o propósito (tb_rFull_Adder_1) e seguidas as mesmas ideias apresentadas anteriormente a respeito de como fazer processamento direto e inverso de portas lógicas.

Simulado o funcionamento do somador na plataforma EDA Playground [2024], embutidas as entradas VinA, VinB e Cin foram obtidas as seguintes ondas:

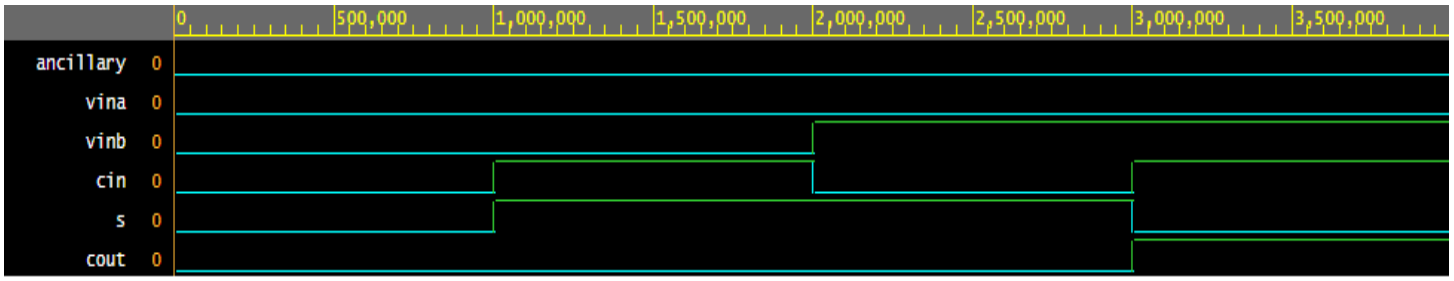


Figura 16 - Simulação no EDA Playground do rFull_Adder_1 - primeira parte (código rFull_Adder_1).

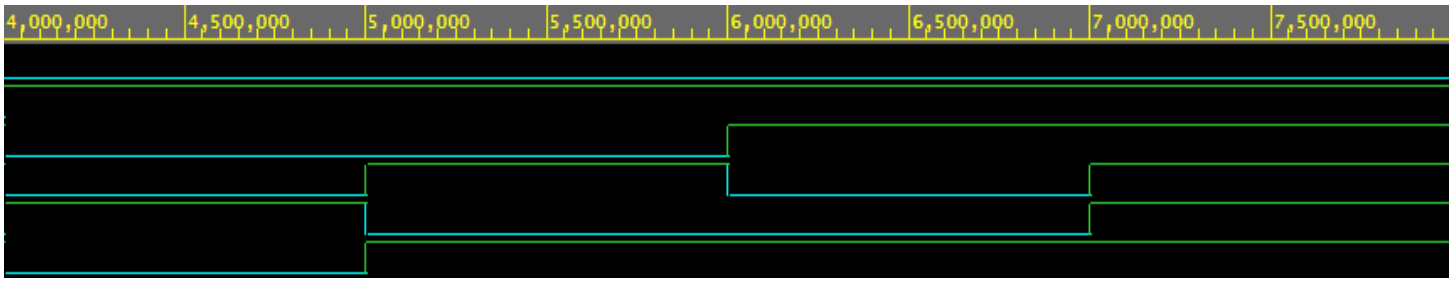


Figura 17 - Simulação no EDA Playground do rFull_Adder_1 - segunda parte (código rFull_Adder_1).

Simuladas a ida e a volta (round trip) foram obtidas as seguintes ondas:

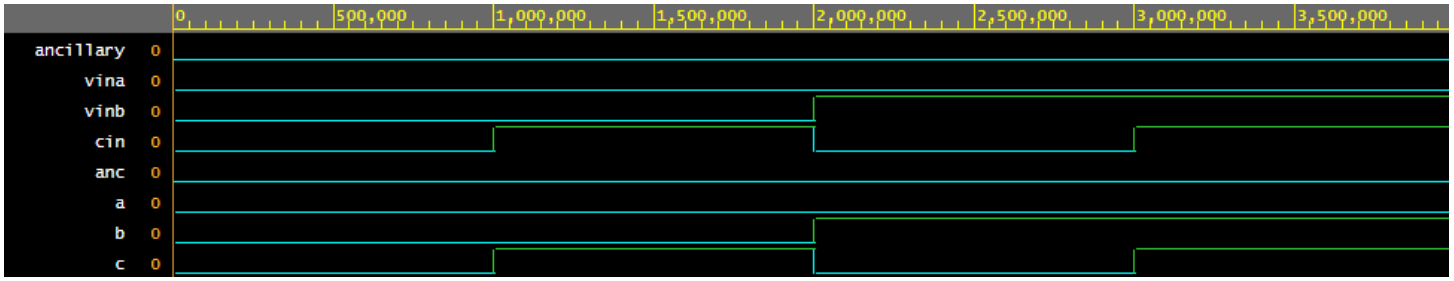


Figura 18 - Simulação no EDA Playground do rFull_Adder_1 (direto e inverso) - primeira parte (código rFull_Adder_1_round_trip).

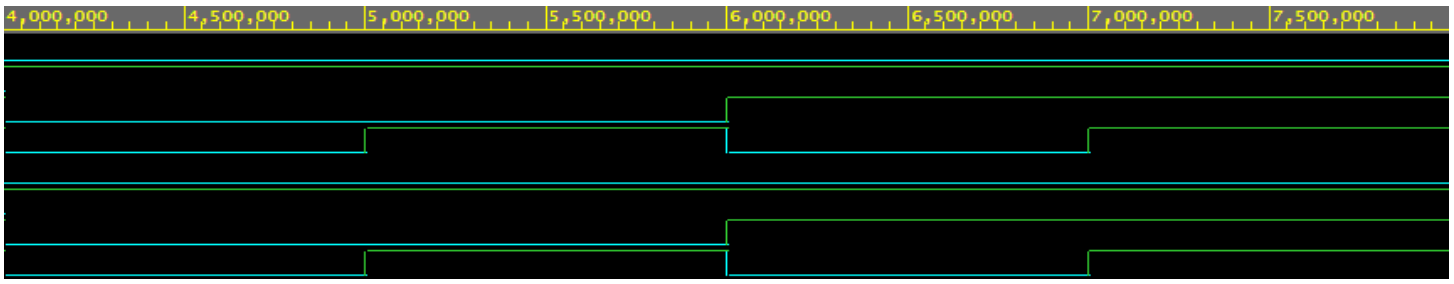


Figura 19 - Simulação no EDA Playground do rFull_Adder_1 (direto e inverso) - segunda parte (código rFull_Adder_1_round_trip).

5.2.4 Circuito rFF_SR

Para o Flip-Flop SR reversível foram utilizadas as portas rOR já codadas em conjunto com portas NOT, onde, de maneira implícita, ilustra a aparência e funcionamento de portas NOR reversíveis.

Fundamentando o funcionamento do circuito, a tabela verdade de um Flip-Flop SR convencional foi considerada (Q* representa o próximo Q no circuito sequencial):

S	R	Q	Q*
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	-
1	1	1	-

Tabela 11 - Tabela verdade Flip-Flop Set-Reset convencional.

Ademais, para a realização de testes para confirmar as funcionalidades das implementações foi escrito, também em VHDL [Pedroni, 2010], um testbench para o propósito (tb_rFF_SR) e seguidas as mesmas ideias apresentadas anteriormente a respeito de como fazer processamento direto e inverso de portas lógicas.

Simulado o funcionamento do Flip-Flop na plataforma EDA Playground [2024], embutidas as entradas S e R foram obtidas as seguintes ondas:

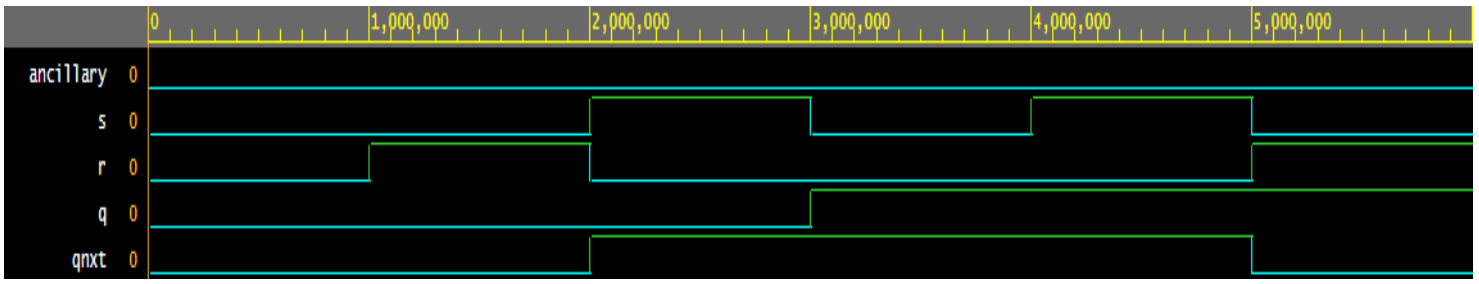


Figura 20 - Simulação no EDA Playground do rFF_SR (código rFF_SR).

Simuladas a ida e a volta (round trip) foram obtidas as seguintes ondas:

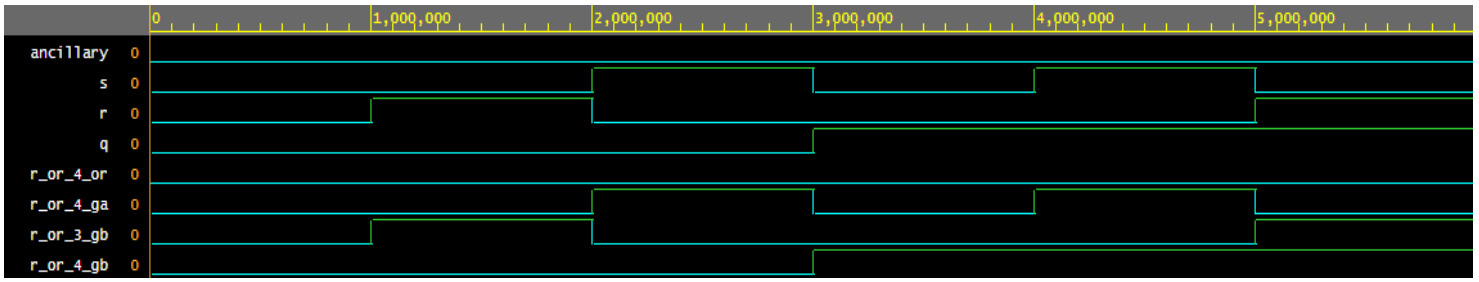


Figura 21 - Simulação no EDA Playground do rFF_SR round trip (código rFF_SR_round_trip).

5.3 Reversibilidade de instruções

Seguindo para a reversibilidade de instruções, foram realizadas duas somas consecutivas, onde estas foram definidas como:

$$x = y + z \tag{15}$$

$$v = x + w \tag{16}$$

Inicialmente foram considerados os valores 0, 0, 1, 0, 1 para as variáveis x, y, z, v, w, respectivamente, com o intuito de se obter os mesmos valores nas mesmas variáveis ao fim do processamento.

Para a aplicação também foram implementados códigos em VHDL [Pedroni, 2010] (Direct_Inverse_Sum e tb_Direct_Inverse_Sum) com o intuito de confirmar a possibilidade de processar de maneira direta e inversa as duas instruções. Onde neles foram utilizados os códigos e conceitos já realizados nas portas rOR, rAND e no somador completo reversível de 1 bit.

Para controle do fluxo de execução foram utilizados dois sinais auxiliares, control_bit e PC, onde controlam, respectivamente, se é executado direto ou inverso e a instrução atual a ser executada.

Ademais, é inviável utilizar a reversibilidade de portas lógicas para fazer o controle dos estados do sistema. Ao executar uma única instrução é possível aplicar os conceitos já postulados anteriormente sobre portas lógicas reversíveis, mas ao executar uma segunda instrução em sequência o estado do sistema no instante de término de execução da primeira instrução é perdido. Logo, para que houvesse reversibilidade das instruções, foi utilizado um mecanismo de armazenamento de estados constituído por um sinal auxiliar de 6 posições (memory), um para cada valor das variáveis mais um para identificação de término da sequência de execução direta.

Por conseguinte, executados os códigos foram obtidas as seguintes ondas:

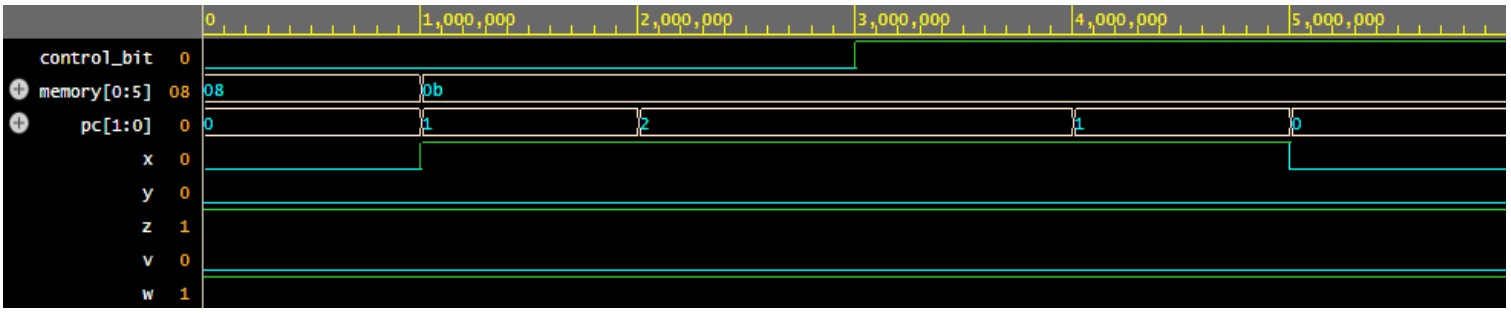


Figura 22 - Simulação direta e inversa no EDA Playground das duas somas (código Direct_Inverse_Sum).

É possível fazer um CPU completo [Oliveira et al. 2020] utilizando conceitos de reversibilidade.

5.4 Consumo Energético

Com relação ao consumo energético, foram simuladas as portas lógicas reversíveis rAND e rOR no Quartus Prime [Intel Corporation, 2020] utilizando a ferramenta de estimação de consumo energético integrada do software, Power Analyser Tool.

5.4.1 Consumo de uma Porta rAND

5.4.1.1 Partes Direta e Inversa da Porta rAND Separadas

Simulada a parte direta da porta em uma simulação 1 e, separadamente, simulada a parte inversa da mesma porta em uma simulação 2 foi obtido que o consumo energético se manteve em 55,55 mW em ambas as simulações.

5.4.1.2 Partes Direta e Inversa Justapostas (round trip)

Simuladas as partes direta e inversa da mesma porta em uma só simulação foi obtido que o consumo energético se manteve em 55,55 mW.

5.4.2 Consumo de uma Porta rOR

5.4.2.1 Partes Direta e Inversa da Porta rOR Separadas

Simulada a parte direta da porta em uma simulação 3 e, separadamente, simulada a parte inversa da mesma porta em uma simulação 4 foi obtido que o consumo energético se manteve em 55,55 mW em ambas as simulações.

5.4.2.2 Partes Direta e Inversa Justapostas (round trip)

Simuladas as partes direta e inversa da mesma porta em uma só simulação foi obtido que o consumo energético se manteve em 55,55 mW.

6 DISCUSSÃO

Analisados os resultados obtidos com a figura 10 foi perceptível que a porta rAND agiu da maneira esperada, pois assim que embutidas as entradas Ancillary, VinA e VinB a respectiva saída rAandB sempre condizia com o sinal esperado de uma porta AND convencional. Ademais, quando analisadas as figuras 11 e 12 foi possível confirmar a reversibilidade esperada, onde para cada entrada embutida na primeira porta rAND foram obtidas saídas, na segunda porta rAND, que coincidiam com as entradas iniciais, evidenciando a recuperação do estado imediatamente anterior ao processamento da primeira porta, quer esta seja executada da esquerda para a direita, quer esta seja executada da direita para a esquerda.

Trazido o assunto para a porta rOR, assim que analisada a figura 13 é possível dizer que a porta se comportou de acordo com o esperado, pois ao ser alimentada com as entradas Ancillary, VinA e VinB uma respectiva saída rAorB sempre se mostrava correta quando comparada com o esperado de uma porta OR convencional. Também, quando analisadas as figuras 14 e 15 foi possível observar que a reversibilidade foi garantida, onde para cada entrada embutida na primeira porta rOR foram obtidas saídas, na segunda porta rOR, que coincidiam com as entradas iniciais, o que prova que o estado imediatamente anterior ao processamento da primeira porta foi restabelecido e se faz verdade tanto da esquerda para a direita quanto da direita para a esquerda.

Com relação ao somador desenvolvido, assim como nas portas rAND e rOR, os resultados obtidos foram congruentes com o esperado, funcionando exatamente como um somador completo convencional ao passo que suporta, também, reversibilidade para que possam ser obtidos os sinais iniciais anteriores à execução do circuito. O que por si só já está dentro do esperado, pois este foi inteiramente construído com o conjunto de portas lógicas reversíveis inicialmente idealizado no trabalho.

Ademais, o funcionamento do Flip-Flop reversível obtido fez juz ao funcionamento de um Flip-Flop convencional, onde para cada entrada de acordo com a tabela 11 houve coerência quando analisada a relação de entradas e saídas na figura 20. E, analisada a figura 21, assim como nos circuitos anteriores este se mostrou reversível, onde para cada combinação de entradas possível foram obtidas respectivas saídas que coincidiam com os valores iniciais de cada sinal de entrada.

Com relação às execuções das instruções, as duas somas, também ocorreram de acordo com o esperado, tendo em vista que o salvamento de contexto por si só já consome energia e memória, mas ainda assim foi útil na tentativa de se recuperar os estados anteriores às operações, onde ao fim da execução foram obtidos valores das variáveis que quando comparados com os valores iniciais houve uma correspondência unânime.

Portanto, os resultados obtidos dos circuitos feitos foram condizentes com os esperados e tudo ocorreu de acordo com o planejado sem que maiores problemas fossem encontrados.

Com relação ao consumo de energia das portas reversíveis os resultados obtidos também foram satisfatórios, pois, apesar de ser esperado que para cada bit apagado fossem dissipados aproximadamente $2{,}8 \times 10^{-21}$ Joules de energia na forma de calor do sistema, até o momento não existem instrumentos e nem softwares de simulação com precisão de 21 casas decimais. O que resultou na mesma estimativa de energia em todos os casos simulados e a medição se tornou inviável. Portanto os resultados foram coerentes com o esperado e tais trabalhos se mostraram insuficientes para validar a diminuição dos gastos de energia que a computação binária reversível pode proporcionar.

7 CONCLUSÃO

Considerado o contexto de reversibilidade e as definições das portas lógicas reversíveis, realizadas as operações foi possível dizer que para que os sinais originais fossem obtidos bastou utilizar as mesmas portas lógicas reversíveis utilizadas para a obtenção das saídas diretas de forma a interligar os sinais garbages assim como as próprias tabelas 8 e 9 sugerem. Logo, foi possível confirmar a possibilidade de se fazer portas lógicas reversíveis, usufruindo do recurso de bijectividade. Assim, foi alcançada uma forma de se realizar operações digitais de maneira direta e inversa, bem como almejado inicialmente neste projeto.

Portanto, caso o objetivo seja executar, respectivamente, de maneira direta e inversa uma operação, por exemplo AND, reversivelmente, basta que as saídas da primeira porta sejam colocadas nas entradas de uma segunda porta lógica do mesmo tipo obedecendo às regras definidas nas tabelas 8 e 9 anteriormente. Logo, ao fim do processo, as saídas da segunda porta serão equivalentes aos valores anteriormente fornecidos na entrada da primeira porta, voltando para o estado inicial, o estado que o sistema estava antes de executar a operação.

De maneira análoga, as operações OR e NOT também funcionaram assim, de forma que uma simples conexão justaposta de duas portas do mesmo tipo é suficiente para se preservar a propriedade de bijeção, respeitar o mapeamento definido e a reversibilidade dos circuitos lógicos implementados a partir do conjunto de portas lógicas reversíveis rAND, rOR e NOT.

Utilizado o conjunto de portas lógicas reversíveis na construção do somador completo de 1 bit reversível, este se demonstrou reversível assim que simulado o circuito, onde foi suficiente substituir as portas convencionais do circuito somador original pelas portas reversíveis construídas neste trabalho, resultando na possibilidade de se realizar somas de maneira direta e inversa. Assim, foram realizadas com sucesso duas instruções de soma justapostas de maneira direta e logo em seguida de maneira inversa, confirmando a possibilidade de se realizar instruções também de maneira direta e inversa com o uso de circuitos reversíveis.

Com relação ao exemplo de circuito sequencial desenvolvido, o rFF-SR, sua implementação evidenciou que é possível fazer circuitos sequenciais de maneira reversível, onde a mesma metodologia utilizada nos circuitos combinacionais reversíveis pode ser aplicada aos circuitos sequenciais para obtenção de circuitos sequenciais reversíveis.

Enfim, utilizado um conjunto de portas lógicas reversíveis é possível fazer qualquer operação digital ser revertida a fim de obter os sinais iniciais. Logo, reversibilidade lógica equivale a reversibilidade operacional quando o assunto é processamento de circuitos digitais.

Sobre o consumo energético, a simulação não foi suficiente para mostrar sua redução, pois não houve manuseio de equipamento com a precisão requerida. Logo, é necessário um ambiente controlado, altamente isolado e um medidor de consumo energético com precisão de 21 casas decimais para que seja perceptível a alteração no consumo de energia dos cenários de teste realizados para que as ideias de constância de entropia e redução no consumo sejam obtidas, o que acabou saindo das possibilidades deste trabalho em simulação e afetou a percepção do consumo de energia dos sistemas testados.

8 TRABALHOS FUTUROS

Com o intuito de continuar esta pesquisa, os próximos interessados que se dispuserem a tal tarefa podem considerar implementar fisicamente os circuitos reversíveis apresentados, podendo ser utilizados transistores para o propósito. E é esperado que, por se tratar de implementação no mundo real e não apenas uma simulação, os resultados a serem obtidos sejam mais fiéis ao idealizado do que foram neste trabalho.

Ademais, a implementação do restante de uma CPU reversível se faz possível bastando incluir, à arquitetura de uma CPU convencional, as portas lógicas reversíveis, os salvamentos de contexto antes de executar cada instrução e o controle dos fluxos de execução direta e inversa das instruções. Uma vez obtido o funcionamento da CPU reversível, pode-se realizar sua síntese em FPGA. Portanto, este também poderia ser um caminho viável para aqueles que assim desejarem continuar este trabalho.

9 BIBLIOGRAFIA E REFERÊNCIAS

- LANDAUER, Rolf. *Irreversibility and heat generation in the computing process*. IBM Journal of Research and Development, v. 5, n. 3, p. 183-191, 1961;
- HALLIDAY, David; RESNICK, Robert; WALKER, Jearl. *Fundamentos de Física*. 10. ed. Rio de Janeiro. LTC, v. 2, 2016;
- PEDRONI, Volnei A. *Eletrônica digital moderna e VHDL*. Rio de Janeiro: Elsevier; Campus, 2010;
- SOUZA, Diogo; ROMERO, Milton; MARTINS, Evandro. *Universal Set of Reversible Quaternary Logic Gates*. International Journal of Computer Applications, v. 174, n. 27, 2021;
- TOFFOLI, Tommaso. *Reversible computing*. International Colloquium on Automata, Languages, and Programming. Springer, p. 632–644, 1980;
- FRANK, Michael. *Generalized Reversible Computing*. Reversible Computation: 9th International Conference, RC, p. 19-34, 2017;
- BENNETT, Charles. *Logical reversibility of computation*. IBM Journal of Research and Development, v. 17, n. 6, p. 525–532, 1973;
- THAPLIYAL, Himanshu; SRINIVAS, M.B; ZWOLINSKI, Mark, *A beginning in the reversible logic synthesis of sequential circuits*. MAPLD, 2005;
- KJAERGAARD, M.; Schwartz, M. D.; Braumüller, J.; Gambetta, J. M. (Eds.). *Superconducting Qubits: Current State of Play*. 1. ed. [S.l.]: [s.n.], 2020;
- GLÜCK, Robert; Yokoyama, Tetsuo. *Reversible Computing from a programming Language Perspective*. Theoretical Computer Science. 953, 2023;
- GUIDORIZZI, Hamilton, *Um curso de Cálculo*. 5. ed. Rio de Janeiro. LTC, v. 1 2001;
- FLOYD, Thomas L. *Sistemas digitais: fundamentos e aplicações*. 10. ed. São Paulo: Pearson Prentice Hall, 2012;
- INTEL CORPORATION**.** *Quartus Prime*. 20.1. Santa Clara, CA: Intel Corporation, 2020;
- AUTODESK, Inc**.** *AutoCAD 2023*. San Rafael, CA: Autodesk, 2023;
- EDA Playground. *EDA Playground*. Disponível em: https://www.edaplayground.com/. Acesso em: 4 abril. 2024;

- OLIVEIRA, Wellington; GOUVEIA, Tiago; ROMERO, Milton. MARTINS, Evandro. *Four Stage Pipeline Quaternary Processor*. Ingeniare. Revista Chilena de Ingeniería, v. 28, n. 3, 2020.

10 APÊNDICES

Apêndice A - Códigos da rAND (código da arquitetura rAND foi reutilizado nos códigos subsequentes)

```
library ieee;
use ieee.std_logic_1164.all;

entity rAND is
   port
        (
      ancillary, VinA, VinB   : in std_logic;
      gA, rAandB, gB                    : out std_logic
   );
end entity;

architecture behavioral of rAND is
begin

        process(ancillary, VinA, VinB) is
   begin
      if ancillary = '0' and VinA = '0' and VinB = '0' then
         gA <= '0';
         rAandB <= '0';
         gB <= '0';

      elsif ancillary = '0' and VinA = '0' and VinB = '1' then
                        gA <= '0';
         rAandB <= '0';
         gB <= '1';

                elsif ancillary = '0' and VinA = '1' and VinB = '0' then
                        gA <= '1';
         rAandB <= '0';
         gB <= '0';
```

```
            elsif ancillary = '0' and VinA = '1' and VinB = '1' then
                    gA <= '0';
        rAandB <= '1';
        gB <= '1';

            elsif ancillary = '1' and VinA = '0' and VinB = '0' then
                    gA <= '0';
        rAandB <= '1';
        gB <= '0';

            elsif ancillary = '1' and VinA = '0' and VinB = '1' then
        gA <= '1';
        rAandB <= '0';
        gB <= '1';

      elsif ancillary = '1' and VinA = '1' and VinB = '0' then
                    gA <= '1';
        rAandB <= '1';
        gB <= '0';

      elsif ancillary = '1' and VinA = '1' and VinB = '1' then
                    gA <= '1';
        rAandB <= '1';
        gB <= '1';
      end if;

   end process;

end behavioral;



library IEEE;
```

```
use IEEE.std_logic_1164.all;

entity teste_rAND is
   port
        (
      ancillary, VinA, VinB   : in std_logic;
      gA, rAandB, gB                    : out std_logic
   );
end entity;

architecture dataflow of teste_rAND is

begin

        rAND1: entity work.rAND(behavioral)
        port map(ancillary, VinA, VinB, gA, rAandB, gB);

end dataflow;



library IEEE;
use ieee.std_logic_1164.all;

entity tb_rAND is
end entity;

architecture mixed of tb_rAND is
   signal ancillary, VinA, VinB       : std_logic;
   signal gA, rAandB, gB                    : std_logic;

begin
   rAND_binaria: entity work.teste_rAND(dataflow)
        port map(ancillary, VinA, VinB, gA, rAandB, gB);
```

```
        Teste: process is
        type linha_tv is record
        ancillary, VinA, VinB : std_logic;
    end record;
            type vet_linha_tv is array (0 to 7) of linha_tv;
    constant tabela_verdade: vet_linha_tv :=

            -- ancillary   A   B
    ( ('0',      '0','0'),
      ('0',      '0','1'),
      ('0',      '1','0'),
      ('0',      '1','1'),
      ('1',      '0','0'),
      ('1',      '0','1'),
      ('1',      '1','0'),
      ('1',      '1','1') );

  begin
      for i in tabela_verdade'range loop
      ancillary <= tabela_verdade(i).ancillary;
       VinA <= tabela_verdade(i).VinA;
       VinB <= tabela_verdade(i).VinB;

       wait for 1 ns;

    end loop;

    report "Fim dos testes";
    wait;
  end process;
end;
```

Apêndice B - Códigos da rAND_round_trip

```
library IEEE;
use IEEE.std_logic_1164.all;

entity teste_rAND is
   port
        (
      ancillary, VinA, VinB   : in std_logic;
      gA, rAandB, gB                    : out std_logic
   );
end entity;

architecture dataflow of teste_rAND is
signal NovoAncillary, NovoA, NovoB: std_logic;

begin

        rAND1: entity work.rAND(behavioral)
        port map(ancillary, VinA, VinB, NovoAncillary, NovoA, NovoB);

        rAND2: entity work.rAND(behavioral)
        port map(NovoAncillary, NovoA, NovoB, gA, rAandB, gB);

end dataflow;



library IEEE;
use ieee.std_logic_1164.all;

entity tb_rAND_round_trip is
end entity;

architecture mixed of tb_rAND_round_trip is
```

```
    signal ancillary, VinA, VinB       : std_logic;
    signal gA, rAandB, gB                      : std_logic;

begin
    rAND_binaria: entity work.teste_rAND(dataflow)
          port map(ancillary, VinA, VinB, gA, rAandB, gB);

          Teste: process is
          type linha_tv is record
          ancillary, VinA, VinB : std_logic;
        end record;
                  type vet_linha_tv is array (0 to 7) of linha_tv;
        constant tabela_verdade: vet_linha_tv :=

                  -- ancillary   A   B
        ( ('0',       '0','0'),
          ('0',       '0','1'),
          ('0',       '1','0'),
          ('0',       '1','1'),
          ('1',       '0','0'),
          ('1',       '0','1'),
          ('1',       '1','0'),
          ('1',       '1','1') );

    begin
          for i in tabela_verdade'range loop
          ancillary <= tabela_verdade(i).ancillary;
           VinA <= tabela_verdade(i).VinA;
           VinB <= tabela_verdade(i).VinB;

           wait for 1 ns;

           assert gA = tabela_verdade(i).ancillary report "Erro";
           assert rAandB = tabela_verdade(i).VinA report "Erro";
```

```
            assert gB = tabela_verdade(i).VinB report "Erro";
        end loop;

        report "Fim dos testes";
        wait;
    end process;
end;
```

Apêndice C - Códigos da rOR(código da arquitetura r_OR foi reutilizado nos códigos subsequentes)

```
library IEEE;
use IEEE.std_logic_1164.all;

entity r_OR is
    port
        (
        ancillary, VinA, VinB   : in std_logic;
        rAorB, gA, gB               : out std_logic
    );
end entity;

architecture behavioral of r_OR is

begin
        process(ancillary, VinA, VinB) is
        begin
        if ancillary = '0' and VinA = '0' and VinB = '0' then
        rAorB <= '0';
         gA <= '0';
         gB <= '0';

      elsif ancillary = '0' and VinA = '0' and VinB = '1' then
        rAorB <= '1';
                        gA <= '0';
```

```
    gB <= '0';

        elsif ancillary = '0' and VinA = '1' and VinB = '0' then
   rAorB <= '1';
                gA <= '0';
    gB <= '1';

        elsif ancillary = '0' and VinA = '1' and VinB = '1' then
   rAorB <= '1';
                gA <= '1';
    gB <= '0';

        elsif ancillary = '1' and VinA = '0' and VinB = '0' then
   rAorB <= '0';
                gA <= '0';
    gB <= '1';

        elsif ancillary = '1' and VinA = '0' and VinB = '1' then
   rAorB <= '0';
    gA <= '1';
    gB <= '0';

elsif ancillary = '1' and VinA = '1' and VinB = '0' then
   rAorB <= '0';
                gA <= '1';
    gB <= '1';

elsif ancillary = '1' and VinA = '1' and VinB = '1' then
   rAorB <= '1';
                gA <= '1';
    gB <= '1';
end if;

   end  process;
```

```
end architecture;



library IEEE;
use IEEE.std_logic_1164.all;

entity teste_r_OR is
   port
        (
      ancillary, VinA, VinB   : in std_logic;
      rAorB, gA, gB               : out std_logic
   );
end entity;

architecture behavioral of teste_r_OR is

begin

        r_OR_1: entity work.r_OR(behavioral)
        port map(ancillary, VinA, VinB, rAorB, gA, gB);

end architecture;



library IEEE;
use ieee.std_logic_1164.all;

entity tb_r_OR is
end entity;

architecture mixed of tb_r_OR is
   signal ancillary, VinA, VinB       : std_logic;
```

```
    signal rAorB, gA, gB                    : std_logic;

begin
    r_OR_binaria: entity work.teste_r_OR(behavioral)
        port map(ancillary, VinA, VinB, rAorB, gA, gB);

        Teste: process is
        type linha_tv is record
        ancillary, VinA, VinB : std_logic;
      end record;
              type vet_linha_tv is array (0 to 7) of linha_tv;
      constant tabela_verdade: vet_linha_tv :=

              -- ancillary   A   B
      ( ('0',      '0','0'),
        ('0',      '0','1'),
        ('0',      '1','0'),
        ('0',      '1','1'),
        ('1',      '0','0'),
        ('1',      '0','1'),
        ('1',      '1','0'),
        ('1',      '1','1') );

    begin
        for i in tabela_verdade'range loop
         VinA <= tabela_verdade(i).VinA;
         VinB <= tabela_verdade(i).VinB;
         ancillary <= tabela_verdade(i).ancillary;

         wait for 1 ns;

      end loop;

      report "Fim dos testes";
```

```
        wait;
    end process;
end;
```

Apêndice D - Códigos da rOR_round_trip

```
library IEEE;
use IEEE.std_logic_1164.all;

entity teste_r_OR is
    port
        (
        ancillary, VinA, VinB   : in std_logic;
        rAorB, gA, gB           : out std_logic
    );
end entity;

architecture dataflow of teste_r_OR is
        signal NovoAncillary, NovoA, NovoB: std_logic;

begin

        r_OR_1: entity work.r_OR(behavioral)
        port map(ancillary, VinA, VinB, NovoAncillary, NovoA, NovoB);

        r_OR_2: entity work.r_OR(behavioral)
        port map(NovoAncillary, NovoA, NovoB, rAorB, gA, gB);

end architecture;



library IEEE;
use ieee.std_logic_1164.all;
```

```
entity tb_r_OR_round_trip is
end entity;

architecture mixed of tb_r_OR_round_trip is
   signal ancillary, VinA, VinB       : std_logic;
   signal rAorB, gA, gB                        : std_logic;

begin
   r_OR_binaria: entity work.teste_r_OR(dataflow)
        port map(ancillary, VinA, VinB, rAorB, gA, gB);

        Teste: process is
        type linha_tv is record
        ancillary, VinA, VinB : std_logic;
      end record;
              type vet_linha_tv is array (0 to 7) of linha_tv;
      constant tabela_verdade: vet_linha_tv :=

              -- ancillary   A   B
      ( ('0',      '0','0'),
        ('0',      '0','1'),
        ('0',      '1','0'),
        ('0',      '1','1'),
        ('1',      '0','0'),
        ('1',      '0','1'),
        ('1',      '1','0'),
        ('1',      '1','1') );

   begin
        for i in tabela_verdade'range loop
         VinA <= tabela_verdade(i).VinA;
         VinB <= tabela_verdade(i).VinB;
         ancillary <= tabela_verdade(i).ancillary;
```

```
            wait for 1 ns;

            assert rAorB = tabela_verdade(i).ancillary report "Erro";
            assert gA = tabela_verdade(i).VinA report "Erro";
            assert gB = tabela_verdade(i).VinB report "Erro";
        end loop;

        report "Fim dos testes";
        wait;
    end process;
end;
```

Apêndice E - Códigos do rFull_Adder_1

```
library ieee;
use ieee.std_logic_1164.all;

entity rFull_Adder_1 is
    port
        (
        ancillary, VinA, VinB, Cin      : in std_logic;
        S, Cout                                     : out std_logic
    );
end entity;

architecture dataflow of rFull_Adder_1 is

    signal gA_aux:              std_logic_vector(0 to 21);
    signal gB_aux:              std_logic_vector(0 to 21);
    signal result_aux:          std_logic_vector(0 to 21);

begin

-- Direct part:
```

```
    -- Sum result, S (A XOR B XOR Cin).
        rAND1: entity work.rAND(behavioral)
    port map(ancillary, "not"(VinA), VinB, gA_aux(0), result_aux(0),
gB_aux(0));

    rAND2: entity work.rAND(behavioral)
        port map(ancillary, VinA, "not"(VinB), gA_aux(1), result_aux(1),
gB_aux(1));

        rOR3: entity work.r_OR(behavioral)
    port map(ancillary, result_aux(0), result_aux(1), result_aux(2),
gA_aux(2), gB_aux(2));

        rAND4: entity work.rAND(behavioral)
        port map(ancillary, "not"(result_aux(2)), Cin, gA_aux(3),
result_aux(3), gB_aux(3));

    rAND5: entity work.rAND(behavioral)
        port map(ancillary, result_aux(2), "not"(Cin), gA_aux(4),
result_aux(4), gB_aux(4));

        rOR6: entity work.r_OR(behavioral)
        port map(ancillary, result_aux(3), result_aux(4), S, gA_aux(5),
gB_aux(5));


    -- Carry out, Cout (A*B+A*Cin+B*Cin).
        rAND7: entity work.rAND(behavioral)
    port map(ancillary, VinA, VinB, gA_aux(6), result_aux(6), gB_aux(6));

    rAND8: entity work.rAND(behavioral)
    port map(ancillary, VinA, Cin, gA_aux(7), result_aux(7), gB_aux(7));

    rAND10: entity work.rAND(behavioral)
```

```
        port map(ancillary, VinB, Cin, gA_aux(8), result_aux(8), gB_aux(8));

        rOR9: entity work.r_OR(behavioral)
        port map(ancillary, result_aux(6), result_aux(7), result_aux(9),
gA_aux(9), gB_aux(9));

        rOR11: entity work.r_OR(behavioral)
                port map(ancillary, result_aux(9), result_aux(8), Cout,
gA_aux(10), gB_aux(10));

    end dataflow;



    library IEEE;
    use IEEE.std_logic_1164.all;

    entity tb_rFull_Adder_1 is
    end entity;

    architecture mixed of tb_rFull_Adder_1 is
      signal ancillary, VinA, VinB, Cin      : std_logic;
      signal S, Cout                                    : std_logic;

    begin
      rFull_Adder_1: entity work.rFull_Adder_1(dataflow)
          port map(ancillary, VinA, VinB, Cin, S, Cout);

          Teste: process is
          type linha_tv is record
          VinA, VinB, Cin : std_logic;
        end record;
                type vet_linha_tv is array (0 to 7) of linha_tv;
        constant tabela_verdade: vet_linha_tv :=
```

```
                    --  A   B   C
            ( ('0','0','0'),
              ('0','0','1'),
              ('0','1','0'),
              ('0','1','1'),
              ('1','0','0'),
              ('1','0','1'),
              ('1','1','0'),
              ('1','1','1') );

        begin
             for i in tabela_verdade'range loop
              VinA <= tabela_verdade(i).VinA;
              VinB <= tabela_verdade(i).VinB;
              Cin <= tabela_verdade(i).Cin;
              ancillary <= '0';

              wait for 1 ns;

            end loop;

            report "Fim dos testes";
            wait;
        end process;
    end;
```

Apêndice F - Códigos do rFull_Adder_1_round_trip

```
    library ieee;
    use ieee.std_logic_1164.all;

    entity rFull_Adder_1_round_trip is
        port
             (
```

```
        ancillary, VinA, VinB, Cin      : in std_logic;
        anc, A, B, C                    : out std_logic
    );
end entity;

architecture dataflow of rFull_Adder_1_round_trip is

    signal gA_aux:              std_logic_vector(0 to 21);
    signal gB_aux:              std_logic_vector(0 to 21);
    signal result_aux:          std_logic_vector(0 to 21);

begin

-- Direct part:

        -- Sum result, S (A XOR B XOR Cin).
            rAND1: entity work.rAND(behavioral)
        port map(ancillary, "not"(VinA), VinB, gA_aux(0), result_aux(0),
gB_aux(0));

        rAND2: entity work.rAND(behavioral)
            port map(ancillary, VinA, "not"(VinB), gA_aux(1), result_aux(1),
gB_aux(1));

            rOR3: entity work.r_OR(behavioral)
        port map(ancillary, result_aux(0), result_aux(1), result_aux(2),
gA_aux(2), gB_aux(2));

            rAND4: entity work.rAND(behavioral)
            port map(ancillary, "not"(result_aux(2)), Cin, gA_aux(3),
result_aux(3), gB_aux(3));

        rAND5: entity work.rAND(behavioral)
```

```
                port map(ancillary, result_aux(2), "not"(Cin), gA_aux(4),
result_aux(4), gB_aux(4));

                rOR6: entity work.r_OR(behavioral)
                port map(ancillary, result_aux(3), result_aux(4), result_aux(5),
gA_aux(5), gB_aux(5));


        -- Carry out, Cout (A*B+A*Cin+B*Cin).
                rAND7: entity work.rAND(behavioral)
        port map(ancillary, VinA, VinB, gA_aux(6), result_aux(6), gB_aux(6));

        rAND8: entity work.rAND(behavioral)
        port map(ancillary, VinA, Cin, gA_aux(7), result_aux(7), gB_aux(7));

        rAND10: entity work.rAND(behavioral)
        port map(ancillary, VinB, Cin, gA_aux(8), result_aux(8), gB_aux(8));

        rOR9: entity work.r_OR(behavioral)
        port map(ancillary, result_aux(6), result_aux(7), result_aux(9),
gA_aux(9), gB_aux(9));

        rOR11: entity work.r_OR(behavioral)
                port map(ancillary, result_aux(9), result_aux(8), result_aux(10),
gA_aux(10), gB_aux(10));


    -- Inverse part:

        -- Inverse Sum:
        rOR12:entity work.r_OR(behavioral)
                port map(result_aux(5), gA_aux(5), gB_aux(5), result_aux(11),
gA_aux(11), gB_aux(11));
```

```
        rAND13:entity work.rAND(behavioral)
                port map(gA_aux(3), gA_aux(11), gB_aux(3), gA_aux(12),
result_aux(12), gB_aux(12));

        rAND16:entity work.rAND(behavioral)
                port map(gA_aux(4), gB_aux(11), gB_aux(4), gA_aux(13),
result_aux(13), gB_aux(13));

        rOR14:entity work.r_OR(behavioral)
                port map("not"(result_aux(12)), gA_aux(2), gB_aux(2),
result_aux(14), gA_aux(14), gB_aux(14));

        rAND15:entity work.rAND(behavioral)
        port map(gA_aux(0), gA_aux(14), gB_aux(0), gA_aux(15),
result_aux(15), gB_aux(15));

        rAND22:entity work.rAND(behavioral)
                port map(gA_aux(1), gB_aux(14), gB_aux(1), gA_aux(16),
result_aux(16), gB_aux(16));


          -- Inverse Carry out:
        rOR17:entity work.r_OR(behavioral)
        port map(result_aux(10), gA_aux(10), gB_aux(10), result_aux(17),
gA_aux(17), gB_aux(17));

        rOR18:entity work.r_OR(behavioral)
        port map(gA_aux(17), gA_aux(9), gB_aux(9), result_aux(18),
gA_aux(18), gB_aux(18));

        rAND19:entity work.rAND(behavioral)
        port map(gA_aux(6), gA_aux(18), gB_aux(6), gA_aux(19),
result_aux(19), gB_aux(19));
```

```
        rAND20:entity work.rAND(behavioral)
        port map(gA_aux(7), gB_aux(18), gB_aux(7), anc, A, gB_aux(20));

        rAND21:entity work.rAND(behavioral)
        port map(gA_aux(8), gB_aux(17), gB_aux(8), gA_aux(21), B, C);

end dataflow;



library IEEE;
use IEEE.std_logic_1164.all;

entity tb_rFull_Adder_1_round_trip is
end entity;

architecture mixed of tb_rFull_Adder_1_round_trip is
    signal ancillary, VinA, VinB, Cin          : std_logic;
    signal anc, A, B, C                                          : std_logic;

begin
    rFull_Adder_1: entity work.rFull_Adder_1_round_trip(dataflow)
          port map(ancillary, VinA, VinB, Cin, anc, A, B, C);

          Teste: process is
          type linha_tv is record
          VinA, VinB, Cin : std_logic;
        end record;
                  type vet_linha_tv is array (0 to 7) of linha_tv;
        constant tabela_verdade: vet_linha_tv :=

                  --  A   B   C
        ( ('0','0','0'),
          ('0','0','1'),
```

```
        ('0','1','0'),
        ('0','1','1'),
        ('1','0','0'),
        ('1','0','1'),
        ('1','1','0'),
        ('1','1','1') );

    begin
        for i in tabela_verdade'range loop
         VinA <= tabela_verdade(i).VinA;
         VinB <= tabela_verdade(i).VinB;
         Cin <= tabela_verdade(i).Cin;
         ancillary <= '0';

         wait for 1 ns;

                        assert anc = '0' report "Erro";
         assert A = tabela_verdade(i).VinA report "Erro";
         assert B = tabela_verdade(i).VinB report "Erro";
         assert C = tabela_verdade(i).Cin report "Erro";
       end loop;

       report "Fim dos testes";
       wait;
    end process;
end;
```

Apêndice G - Códigos do rFF_SR

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rFF_SR is
    port
        (
```

```
        ancillary, S, R                    : in std_logic;
        Q                                  : in std_logic;
        Qnxt                               : out std_logic
    );
end entity;

architecture dataflow of rFF_SR is
        signal r_OR, gA, gB: std_logic_vector(1 to 3);

begin

        r_OR_1: entity work.r_OR(behavioral)
        port map(ancillary, S, Q, r_OR(1), gA(1), gB(1));

        r_OR_2: entity work.r_OR(behavioral)
        port map(ancillary, "not"(r_OR(1)), R, r_OR(2), gA(2), gB(2));

    Qnxt <= not r_OR(2);

end architecture;



library IEEE;
use IEEE.std_logic_1164.all;

entity tb_rFF_SR is
end entity;

architecture mixed of tb_rFF_SR is
    signal ancillary, S, R                 : std_logic;
    signal Q                               :
std_logic;
```

```
        signal Qnxt                                                    :
std_logic;

    begin
      FF_SR_reversivel: entity work.rFF_SR(dataflow)
          port map(ancillary, S, R, Q, Qnxt);

          Teste: process is
          type linha_tv is record
          anc, S, R: std_logic;
        end record;
                type vet_linha_tv is array (0 to 5) of linha_tv;
        constant tabela_verdade: vet_linha_tv :=

                -- anc   S   R
        ( ('0','0','0'),
          ('0','0','1'),
          ('0','1','0'),
          ('0','0','0'),
          ('0','1','0'),
          ('0','0','1'));

      begin
          Q <= '0';
        S <= tabela_verdade(0).S;
        R <= tabela_verdade(0).R;
        ancillary <= '0';

        wait for 1 ns;

          for i in 1 to 5 loop
          Q <= Qnxt;
           S <= tabela_verdade(i).S;
           R <= tabela_verdade(i).R;
```

```
            ancillary <= '0';

            wait for 1 ns;

        end loop;

                wait for 4 ns;

        report "Fim dos testes";
        wait;
    end process;
end;
```

Apêndice H - Códigos do rFF_SR_round_trip

```
library IEEE;
use IEEE.std_logic_1164.all;

entity rFF_SR_round_trip is
    port
        (
        ancillary, S, R                          : in std_logic;
        Q                                                : in std_logic;
        Qnxt                                          : out std_logic;
        r_OR_3_OR, r_OR_3_gA, r_OR_3_gB     : out std_logic;
        r_OR_4_OR, r_OR_4_gA, r_OR_4_gB     : out std_logic
    );
end entity;

architecture dataflow of rFF_SR_round_trip is
        signal r_OR, gA, gB: std_logic_vector(1 to 3);

begin

        r_OR_1: entity work.r_OR(behavioral)
```

```
            port map(ancillary, S, Q, r_OR(1), gA(1), gB(1));

            r_OR_2: entity work.r_OR(behavioral)
            port map(ancillary, "not"(r_OR(1)), R, r_OR(2), gA(2), gB(2));

            r_OR_3: entity work.r_OR(behavioral)
            port map(r_OR(2), gA(2), gB(2), r_OR_3_OR, gA(3), r_OR_3_gB);

        r_OR_4: entity work.r_OR(behavioral)
            port map("not"(gA(3)), gA(1), gB(1), r_OR_4_OR, r_OR_4_gA,
r_OR_4_gB);

            r_OR_3_gA <= gA(3);
        Qnxt <= not r_OR(2);

    end architecture;



    library IEEE;
    use IEEE.std_logic_1164.all;

    entity tb_rFF_SR_round_trip is
    end entity;

    architecture mixed of tb_rFF_SR_round_trip is
        signal ancillary, S, R                                  : std_logic;
        signal Q                                                :
std_logic;
        signal Qnxt                                                     :
std_logic;
        signal r_OR_3_OR, r_OR_3_gA, r_OR_3_gB          : std_logic;
        signal r_OR_4_OR, r_OR_4_gA, r_OR_4_gB          : std_logic;
```

```
begin
    FF_SR_reversivel: entity work.rFF_SR_round_trip(dataflow)
        port map(ancillary, S, R, Q, Qnxt, r_OR_3_OR, r_OR_3_gA,
r_OR_3_gB, r_OR_4_OR, r_OR_4_gA, r_OR_4_gB);

        Teste: process is
        type linha_tv is record
        anc, S, R: std_logic;
    end record;
            type vet_linha_tv is array (0 to 5) of linha_tv;
    constant tabela_verdade: vet_linha_tv :=

            -- anc   S   R
    ( ('0','0','0'),
      ('0','0','1'),
      ('0','1','0'),
      ('0','0','0'),
      ('0','1','0'),
      ('0','0','1'));

  begin
        Q <= '0';
    S <= tabela_verdade(0).S;
    R <= tabela_verdade(0).R;
    ancillary <= '0';

    wait for 1 ns;

            assert r_OR_3_OR = tabela_verdade(0).anc report "Erro";
    assert r_OR_3_gB = tabela_verdade(0).R report "Erro";

    assert r_OR_4_OR = tabela_verdade(0).anc report "Erro";
    assert r_OR_4_gA = tabela_verdade(0).S report "Erro";
```

```
            for i in 1 to 5 loop
            Q <= Qnxt;
             S <= tabela_verdade(i).S;
             R <= tabela_verdade(i).R;
             ancillary <= '0';

             wait for 1 ns;

                              assert r_OR_3_OR = tabela_verdade(i).anc report "Erro";
             assert r_OR_3_gB = tabela_verdade(i).R report "Erro";

             assert r_OR_4_OR = tabela_verdade(i).anc report "Erro";
             assert r_OR_4_gA = tabela_verdade(i).S report "Erro";
          end loop;

                    wait for 4 ns;

          report "Fim dos testes";
          wait;
       end process;
    end;
```

Apêndice I - Códigos da Direct_Inverse_Sum

```
    library ieee;
    use ieee.std_logic_1164.all;

    entity rFull_Adder_1_DIRECT is
       port
            (
       ancillary, Cin          : in std_logic;
       VinA, VinB                    : in std_logic;
       OR_S, OR_Cout                 : out std_logic;
       gA, gB                        : out std_logic_vector(0 to 10)
       );
```

```
end entity;

architecture dataflow of rFull_Adder_1_DIRECT is

   signal gA_aux:              std_logic_vector(0 to 21);
   signal gB_aux:              std_logic_vector(0 to 21);
   signal result_aux:          std_logic_vector(0 to 21);

begin

            rAND1: entity work.rAND(behavioral)
      port map(ancillary, "not"(VinA), VinB, gA_aux(0), result_aux(0),
gB_aux(0));

      rAND2: entity work.rAND(behavioral)
            port map(ancillary, VinA, "not"(VinB), gA_aux(1), result_aux(1),
gB_aux(1));

            rOR3: entity work.r_OR(behavioral)
      port map(ancillary, result_aux(0), result_aux(1), result_aux(2),
gA_aux(2), gB_aux(2));

            rAND4: entity work.rAND(behavioral)
            port map(ancillary, "not"(result_aux(2)), Cin, gA_aux(3),
result_aux(3), gB_aux(3));

      rAND5: entity work.rAND(behavioral)
            port map(ancillary, result_aux(2), "not"(Cin), gA_aux(4),
result_aux(4), gB_aux(4));

            rOR6: entity work.r_OR(behavioral)
            port map(ancillary, result_aux(3), result_aux(4), result_aux(5),
gA_aux(5), gB_aux(5));
```

```
                rAND7: entity work.rAND(behavioral)
        port map(ancillary, VinA, VinB, gA_aux(6), result_aux(6), gB_aux(6));

        rAND8: entity work.rAND(behavioral)
        port map(ancillary, VinA, Cin, gA_aux(7), result_aux(7), gB_aux(7));

        rAND10: entity work.rAND(behavioral)
        port map(ancillary, VinB, Cin, gA_aux(8), result_aux(8), gB_aux(8));

        rOR9: entity work.r_OR(behavioral)
        port map(ancillary, result_aux(6), result_aux(7), result_aux(9),
gA_aux(9), gB_aux(9));

        rOR11: entity work.r_OR(behavioral)
                port map(ancillary, result_aux(9), result_aux(8), result_aux(10),
gA_aux(10), gB_aux(10));



                OR_S <= result_aux(5);
        OR_Cout <= result_aux(10);
        gA <= gA_aux(0 to 10);
        gB <= gB_aux(0 to 10);

    end dataflow;



    library ieee;
    use ieee.std_logic_1164.all;

    entity rFull_Adder_1_INVERSE is
       port
```

```
            (
        OR_S, OR_Cout           : in std_logic;
        gA, gB                  : in std_logic_vector(0 to 10);
        C                       : out std_logic;
        A, B                    : out std_logic;
        anc                             : out std_logic
      );
end entity;

architecture dataflow of rFull_Adder_1_INVERSE is

   signal gA_aux:               std_logic_vector(0 to 21);
   signal gB_aux:               std_logic_vector(0 to 21);
   signal result_aux:           std_logic_vector(0 to 21);

begin

     rOR12:entity work.r_OR(behavioral)
          port map(OR_S, gA(5), gB(5), result_aux(11), gA_aux(11),
gB_aux(11));

     rAND13:entity work.rAND(behavioral)
          port map(gA(3), gA_aux(11), gB(3), gA_aux(12), result_aux(12),
gB_aux(12));

     rAND16:entity work.rAND(behavioral)
          port map(gA(4), gB_aux(11), gB(4), gA_aux(13), result_aux(13),
gB_aux(13));

     rOR14:entity work.r_OR(behavioral)
          port map("not"(result_aux(12)), gA(2), gB(2), result_aux(14),
gA_aux(14), gB_aux(14));

     rAND15:entity work.rAND(behavioral)
```

```
        port map(gA(0), gA_aux(14), gB(0), gA_aux(15), result_aux(15),
gB_aux(15));

        rAND22:entity work.rAND(behavioral)
                port map(gA(1), gB_aux(14), gB(1), gA_aux(16), result_aux(16),
gB_aux(16));



        rOR17:entity work.r_OR(behavioral)
        port map(OR_Cout, gA(10), gB(10), result_aux(17), gA_aux(17),
gB_aux(17));

        rOR18:entity work.r_OR(behavioral)
        port map(gA_aux(17), gA(9), gB(9), result_aux(18), gA_aux(18),
gB_aux(18));

        rAND19:entity work.rAND(behavioral)
        port map(gA(6), gA_aux(18), gB(6), gA_aux(19), result_aux(19),
gB_aux(19));

        rAND20:entity work.rAND(behavioral)
        port map(gA(7), gB_aux(18), gB(7), gA_aux(20), result_aux(20),
gB_aux(20));

        rAND21:entity work.rAND(behavioral)
        port map(gA(8), gB_aux(17), gB(8), gA_aux(21), result_aux(21),
gB_aux(21));

        anc <= gA_aux(15);
        A <= result_aux(16);
        B <= gB_aux(15);
                C <= gB_aux(20);
```

```
end dataflow;



library ieee;
use ieee.std_logic_1164.all;

entity Reversive_Sum is
   port
        (
      ancillary, Cin          : in std_logic;
      VinA, VinB                    : in std_logic;
      anc, A, B                     : out std_logic;
      C                                   : out std_logic;
      S, Cout                       : out std_logic
   );
end entity;

architecture dataflow of Reversive_Sum is

   signal OR_S_aux:           std_logic;
   signal OR_Cout_aux:        std_logic;

        signal gA_aux, gB_aux:    std_logic_vector(1 to 11);

begin

        rFull_Adder_1_DIRECT1: entity
work.rFull_Adder_1_DIRECT(dataflow)
      port map(ancillary, Cin, VinA, VinB, OR_S_aux, OR_Cout_aux, gA_aux,
gB_aux);

      rFull_Adder_1_INVERSE1: entity work.rFull_Adder_1_INVERSE(dataflow)
      port map(OR_S_aux, OR_Cout_aux, gA_aux, gB_aux, C, A, B, anc);
```

```
        S <= OR_S_aux;
        Cout <= OR_Cout_aux;

end dataflow;



library ieee;
use ieee.std_logic_1164.all;

entity Direct_Inverse_Sum is
    port
        (
        control_bit                 : in std_logic;          -- '1' when I want to go
inverse, '0' when I want to go direct.
          PC                                : in std_logic_vector(1 downto 0)
    );
end entity;

architecture mixed of Direct_Inverse_Sum is

        type ram is array(0 to 5) of std_logic;
    signal memory:              ram :=  ('0', '0', '0', '0', '0', '0');

    signal x:                   std_logic := '0';
    signal y:                   std_logic := '0';
    signal z:                   std_logic := '1';
    signal v:                   std_logic := '0';
    signal w:                   std_logic := '1';

        signal ancillary, Cin: std_logic;
    signal VinA, VinB:          std_logic;
    signal anc, A, B:           std_logic;
```

```
    signal C:                       std_logic;
        signal S, Cout:                 std_logic;

begin

        Sums: entity work.Reversive_Sum(dataflow)
    port map(ancillary, Cin, VinA, VinB, anc, A, B, C, S, Cout);

    process(PC)

    begin
        if control_bit = '0' then

                        -- CONTEXT SAVING AND APPLY OF S IN x AND v.
        if PC = "00" then
            memory(0) <= x;
            memory(1) <= y;
            memory(2) <= z;
                        elsif PC = "01" then
            memory(3) <= v;
            memory(4) <= S;
            memory(5) <= w;
            x <= S;
         elsif PC = "10" then
            v <= S;
                        end if;

                        -- INICIATE EXECUTION.
                        if PC = "00" then
                VinA <= y;
            VinB <= z;
            Cin <= '0';
            ancillary <= '0';
```

```
        elsif PC = "01" then
                VinA <= S;
            VinB <= w;
            Cin <= Cout;
            ancillary <= '0';

        end if;

                -- CONTEXT RECOVERING.
                elsif control_bit = '1' then

       if PC = "00" then
                x <= memory(0);
            y <= memory(1);
                z <= memory(2);
        elsif PC = "01" then
                x <= memory(4);
            v <= memory(3);
            w <= memory(5);
                        end if;

                end if;

       end process;

end mixed;



library IEEE;
use IEEE.std_logic_1164.all;

entity tb_Direct_Inverse_Sum is
end entity;
```

```
architecture mixed of tb_Direct_Inverse_Sum is
    signal control_bit          : std_logic;
    signal PC                           : std_logic_vector(1 downto 0);

begin
    Processing: entity work.Direct_Inverse_Sum(mixed)
          port map(control_bit, PC);

          Test: process is
    begin
          control_bit <= '0';
       PC <= "00";

                    wait for 1 ns;

                    control_bit <= '0';
       PC <= "01";

                    wait for 1 ns;

       control_bit <= '0';
       PC <= "10";

                    wait for 1 ns;

                    control_bit <= '1';
       PC <= "10";

                    wait for 1 ns;

       control_bit <= '1';
       PC <= "01";
```

```
                wait for 1 ns;

        control_bit <= '1';
        PC <= "00";

                wait for 1 ns;

                wait for 6 ns;

        report "End of tests";
        wait;
    end process;
end;
```