

# Linguagem e Descrição de Hardware

## Greatest Common Divisor - (GCD)

Giovanna Fantacini<sup>1</sup>, Higor Grigorio<sup>2</sup>, Leonardo Reneres<sup>3</sup>, Luis<sup>4</sup>, Raul Prado Dantas<sup>5</sup>

### SUMÁRIO

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Fundamentação Teórica</b>	<b>3</b>
2.1	Conceitos Básicos de GCD . . . . .	3
2.2	Algoritmo de Euclides . . . . .	4
2.3	Aplicações do GCD em Computação . . . . .	5
<b>3</b>	<b>Desenho de Arquitetura</b>	<b>5</b>
3.1	Partição em Módulos . . . . .	6
3.1.1	Módulo de Controle . . . . .	6
3.1.2	Módulo Datapath . . . . .	6
3.2	Diagramas de Blocos . . . . .	6
<b>4</b>	<b>Desenvolvimento e Implementação</b>	<b>8</b>
4.1	Descrição do Algoritmo em C . . . . .	8
4.2	Modelagem em Verilog . . . . .	12
4.2.1	Módulo Datapath . . . . .	12
4.2.2	Módulo Controle . . . . .	17
<b>5</b>	<b>Máquina de Estados Finitos (FSM)</b>	<b>19</b>
5.1	Descrição dos Estados . . . . .	19
5.1.1	Estado <i>WAIT</i> (Aguardando novos operandos) . . . . .	20
5.1.2	Estado <i>CALC</i> (Calculando o GCD) . . . . .	20
5.1.3	Estado <i>DONE</i> (Esperando o consumidor receber o resultado) . . . . .	20
5.2	Implementação da FSM . . . . .	20
5.2.1	Definição dos Estados e Sinais de Controle . . . . .	21
5.2.2	Lógica de Controle de Sinais . . . . .	21
5.2.3	Controle de Sinais . . . . .	22

<b>6</b>	<b>Simulações e Verificação</b>	<b>23</b>
6.1	Simulação do módulo GCD . . . . .	23
6.2	Ambiente de Simulação . . . . .	23
6.3	Resultados da Simulação . . . . .	23
<b>7</b>	<b>Otimizações e Melhorias Futuras</b>	<b>23</b>
7.1	Otimização do Datapath . . . . .	23
7.1.1	Implementação do Pipelining . . . . .	23
7.1.2	Benefícios do Pipelining . . . . .	24
7.2	Expansão do Design para Sistemas Maiores . . . . .	25
<b>8</b>	<b>Discussão sobre Integração com Sistemas Maiores e Otimização do Datapath</b>	<b>25</b>
<b>9</b>	<b>Conclusões</b>	<b>25</b>

**RESUMO:** Este artigo apresenta a implementação do algoritmo para cálculo do Greatest Common Divisor (GCD) em hardware, utilizando uma abordagem de Control e Datapath. Serão abordadas as principais etapas de desenvolvimento, incluindo a descrição do algoritmo em linguagem C, a modelagem dos módulos de controle e datapath em Verilog, e a simulação e validação do sistema.

**PALAVRAS-CHAVE:** GCD; Greatest Common Divisor; Hardware Description Language; Control; Datapath.

## Greatest Common Divisor - (GCD)

**ABSTRACT:** This paper presents the implementation of the Greatest Common Divisor (GCD) algorithm in hardware, using a Control and Datapath approach. The main development steps will be addressed, including the description of the algorithm in C language, the modeling of control and datapath modules in Verilog, and the system simulation and validation.

**KEYWORDS:** GCD; Greatest Common Divisor; Hardware Description Language; Control; Datapath;

## 1 INTRODUÇÃO

O *Greatest Common Divisor* (GCD), ou Máximo Divisor Comum (MDC), é um conceito fundamental na teoria dos números. Sua importância se manifesta em diversas áreas da computação, onde desempenha um papel crucial em algoritmos essenciais. Entre essas aplicações, destacam-se os sistemas de criptografia, onde o GCD é utilizado para garantir a segurança das comunicações, e os algoritmos de compressão de dados, onde contribui para a otimização do armazenamento e transmissão de informações.

Além disso, o GCD é amplamente empregado em sistemas de controle em hardware, onde sua implementação é importante para o funcionamento de dispositivos embarcados e sistemas de tempo real. A capacidade do GCD de simplificar operações e reduzir a complexidade dos cálculos faz dele um componente indispensável na construção de circuitos digitais robustos e eficientes. Por isso, o entendimento profundo desse conceito e sua aplicação prática são de extrema relevância para engenheiros e cientistas da computação.

Este estudo busca explorar não apenas a teoria por trás do GCD, mas também sua aplicação prática por meio de implementações computacionais. A pesquisa foca no desenvolvimento de simulações em software, abordando desde a modelagem até a implementação.

## 2 FUNDAMENTAÇÃO TEÓRICA

### 2.1 Conceitos Básicos de GCD

O conceito de Greatest Common Divisor (GCD), ou Máximo Divisor Comum (MDC) em português, é fundamental na teoria dos números e tem ampla aplicação em diversas áreas da matemática e da ciência da computação. O GCD de dois números inteiros é o maior número inteiro que divide ambos sem deixar resto. Esse conceito é importante não apenas na simplificação de frações, onde o GCD é utilizado para reduzir frações ao seu menor termo, mas também em áreas mais avançadas, como criptografia, teoria de algoritmos e circuitos digitais.

Na criptografia, por exemplo, o GCD desempenha um papel crítico em algoritmos de chave pública, como o RSA, onde o conceito de coprimosidade (números cujo GCD é 1) é utilizado para garantir a segurança do sistema. Em algoritmos, o GCD é frequentemente utilizado para otimizar cálculos que envolvem divisões e reduções, sendo a base de algoritmos clássicos como o Algoritmo de Euclides, que é amplamente utilizado por sua eficiência na computação do GCD.

No contexto de circuitos digitais, especificamente em módulos de controle e datapath, como os discutidos nos materiais analisados, o GCD pode ser implementado de forma eficiente através de circuitos que utilizam registradores, multiplexadores e unidades aritméticas. Esses componentes são coordenados por um módulo de controle que decide a sequência de operações, como subtrações e trocas de valores, até que o GCD seja obtido. A implementação eficiente desses circuitos é crucial para o desempenho de sistemas que exigem cálculos frequentes de GCD, especialmente em aplicações embarcadas e de tempo real.

Em resumo, o GCD não é apenas um conceito básico da matemática, mas também um elemento essencial em várias aplicações práticas, desde a simplificação de frações até sistemas complexos de criptografia e arquiteturas de hardware digital.

## 2.2 Algoritmo de Euclides

O Algoritmo de Euclides é um dos métodos mais antigos e eficientes para calcular o Greatest Common Divisor (GCD), ou Máximo Divisor Comum (MDC), de dois números inteiros. Esse algoritmo, atribuído ao matemático grego Euclides, tem uma fundamentação simples e robusta, baseada na observação de que o GCD de dois números não muda se o maior dos dois números é substituído pela diferença entre eles. Essa propriedade permite a construção de um algoritmo iterativo que reduz progressivamente o problema até que o divisor comum seja encontrado.

O Algoritmo de Euclides funciona através de uma série de divisões sucessivas. Dado dois números inteiros  $A$  e  $B$ , onde  $A \geq B$ , o algoritmo segue os seguintes passos:

1. **Divisão:** Divida  $A$  por  $B$  e obtenha o quociente  $q$  e o resto  $r$  tal que  $A = Bq + r$ .
2. **Substituição:** Substitua  $A$  por  $B$  e  $B$  por  $r$ .
3. **Repetição:** Repita os passos acima até que  $r$  seja igual a zero. Quando isso acontecer, o GCD é o valor de  $B$  naquele ponto.

Matematicamente, isso pode ser representado como:

$$\text{GCD}(A, B) = \text{GCD}(B, A \bmod B)$$

Quando  $B$  se torna zero,  $A$  contém o GCD dos dois números iniciais.

Considere a aplicação do algoritmo para encontrar o GCD de 48 e 18:

- $48 \div 18 = 2$  com resto 12 ( $48 = 18 \times 2 + 12$ ).
- Substitua  $A = 18$  e  $B = 12$ .
- $18 \div 12 = 1$  com resto 6 ( $18 = 12 \times 1 + 6$ ).
- Substitua  $A = 12$  e  $B = 6$ .

- $12 \div 6 = 2$  com resto 0 ( $12 = 6 \times 2 + 0$ ).

Como o resto é zero, o GCD é 6.

O Algoritmo de Euclides é conhecido por sua eficiência. A cada iteração, o tamanho dos números envolvidos é reduzido, e em média, o número de iterações necessárias é proporcional ao logaritmo do menor número entre os dois. A complexidade computacional do algoritmo é  $O(\log(\min(A, B)))$ , o que o torna extremamente eficiente mesmo para números muito grandes.

A eficiência do Algoritmo de Euclides o torna preferido em muitos contextos de aplicação, desde cálculos aritméticos básicos até implementações em hardware e criptografia, onde cálculos rápidos e precisos do GCD são frequentemente necessários.

## 2.3 Aplicações do GCD em Computação

O *Greatest Common Divisor* (GCD), ou Máximo Divisor Comum (MDC), é uma operação fundamental com diversas aplicações práticas em computação. Sua relevância se estende desde algoritmos básicos até sistemas complexos de criptografia, compressão de dados e implementação em hardware.

Uma das aplicações mais notáveis do GCD está na criptografia, especialmente em sistemas de criptografia assimétrica, como o algoritmo RSA. O RSA depende da dificuldade de fatorar grandes números primos, e a função GCD é usada para determinar se dois números são *coprimos*, ou seja, se seu GCD é 1. A coprimosidade é um conceito crucial no processo de geração de chaves no RSA, onde o GCD garante que a chave pública e o módulo sejam adequados para a operação de criptografia e descryptografia.

Outra aplicação importante do GCD é na compressão de dados. Técnicas de compressão como o *Run-Length Encoding* (RLE) podem utilizar o GCD para determinar padrões de repetição e reduzir a redundância nos dados. Além disso, algoritmos de compressão baseados em análise de fatores de números, como o *factoring-based compression*, usam o GCD para simplificar expressões numéricas e otimizar o armazenamento.

Em **sistemas de controle**, especialmente em hardware, o GCD é utilizado para simplificar os cálculos que envolvem múltiplos ciclos de operação ou para sincronizar sinais em circuitos digitais. No contexto de *Control and Datapath*, a operação de GCD pode ser implementada em circuitos que exigem operações de redução, como divisões sucessivas, facilitando a minimização de recursos computacionais.

A operação de GCD é relevante em hardware porque ela pode ser implementada de forma eficiente utilizando componentes básicos como registradores, multiplexadores e subtratores, que são comuns em *datapaths* de circuitos digitais. A eficiência do GCD, especialmente quando implementado com o *Algoritmo de Euclides*, permite que ele seja utilizado em sistemas de tempo real e em dispositivos embarcados, onde a velocidade de processamento e o uso mínimo de recursos são críticos. Além disso, o GCD é frequentemente empregado em **algoritmos de síntese lógica** para otimizar o design de circuitos e na verificação formal de *hardware*.

Em resumo, o GCD é uma operação essencial em diversas áreas da computação, proporcionando uma base sólida para algoritmos criptográficos, técnicas de compressão de dados e a implementação eficiente de sistemas de controle em *hardware* digital.

## 3 DESENHO DE ARQUITETURA

Esta seção descreve a arquitetura do sistema, incluindo a partição em módulos de controle e datapath, e os diagramas de blocos que ilustram a interação entre os componentes.

### 3.1 Partição em Módulos

A arquitetura do sistema GCD foi organizada em dois módulos principais: Control e Datapath. Esta divisão permite uma clara separação de responsabilidades, permite uma maior modularidade, facilita a implementação, manutenção e expansão do sistema.

#### 3.1.1 Módulo de Controle

O módulo de Controle é responsável por gerenciar o fluxo de dados entre os diferentes componentes do sistema e controlar as transições de estado. Ele coordena as operações do Datapath com base nas condições de controle, assegurando que as etapas do cálculo do GCD sejam executadas na ordem correta. Este módulo inclui uma Máquina de Estados Finitos (FSM), a qual será discutida posteriormente, que determina a sequência de operações, incluindo a inicialização, execução do cálculo e término.

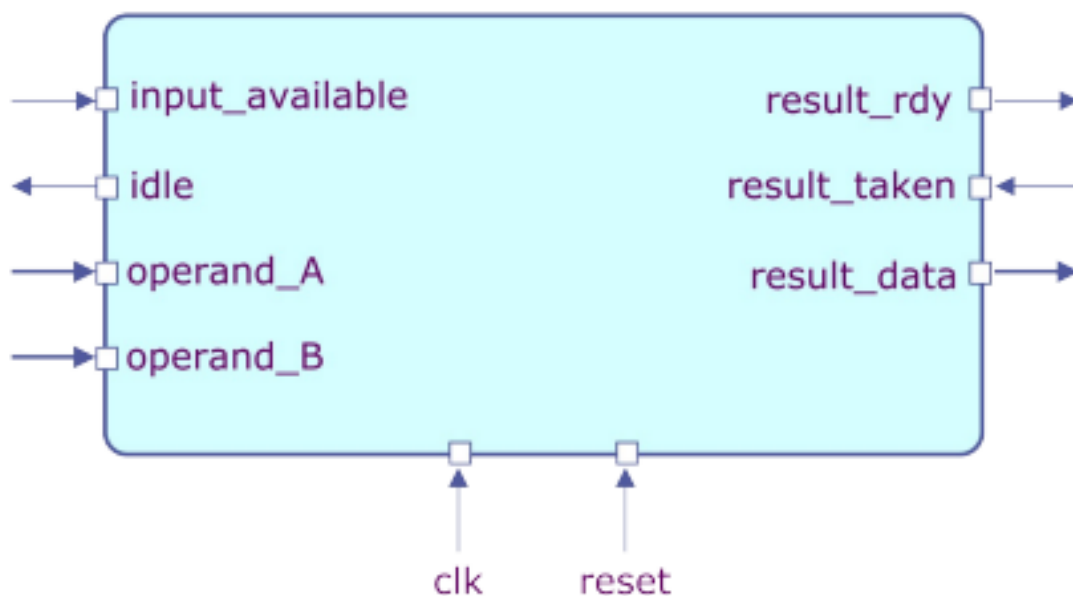
#### 3.1.2 Módulo Datapath

O Datapath realiza as operações aritméticas necessárias para o cálculo do GCD, como subtrações e comparações entre os valores de entrada. Este módulo contém registradores, multiplexadores, e a unidade de subtração, que juntos formam o núcleo da operação matemática. A eficiência do Datapath é crítica para o desempenho global do sistema, uma vez que ele executa as operações fundamentais repetidamente até que o resultado seja obtido.

### 3.2 Diagramas de Blocos

O diagrama de bloco na Figura 1 ilustra a arquitetura de nível superior do sistema GCD, apresentando todas as entradas e saídas em um único bloco. Este diagrama de blocos demonstra como os módulos de Controle e Datapath estão interconectados e como o fluxo de dados é gerenciado entre eles.

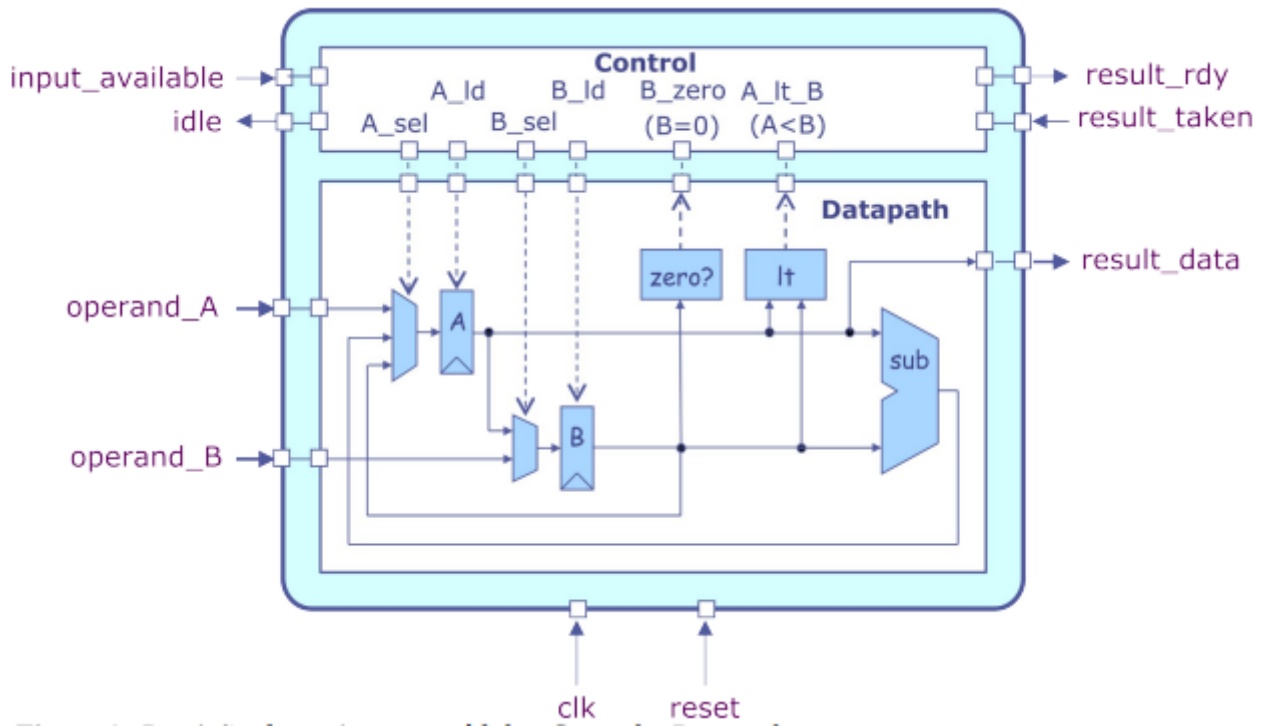
Figura 1: Diagrama de Blocos do Sistema GCD



Fonte: (TAYLOR, 2019)

O diagrama de blocos na Figura 2 ilustra como os componentes principais do sistema GCD se interconectam.

Figura 2: Diagrama de Blocos do Sistema GCD



Fonte: (TAYLOR, 2019)

O Datapath inclui os registradores, multiplexadores e a unidade de subtração, enquanto o Control FSM gerencia as operações com base nos sinais de controle.

## 4 DESENVOLVIMENTO E IMPLEMENTAÇÃO

Os materiais e métodos utilizados no desenvolvimento da pesquisa incluem a descrição do algoritmo em C e a modelagem dos módulos de controle e datapath em Verilog. Além da simulação utilizando o ambiente Intel Quartus.

### 4.1 Descrição do Algoritmo em C

O código a seguir é uma implementação da função GCD (Greatest Common Divisor - Maior Divisor Comum). Essa função calcula o maior divisor comum entre dois números inteiros.

A função GCD recebe dois parâmetros: inA e inB, que representam os dois números inteiros para os quais queremos calcular o maior divisor comum.

O código utiliza um algoritmo chamado "algoritmo de Euclides" para calcular o maior divisor comum. O algoritmo de Euclides é baseado na observação de que o maior divisor comum entre dois números não muda se o menor número for subtraído do maior número repetidamente até que um dos números seja igual a zero.

O algoritmo para cálculo do GCD pode ser implementado de diversas maneiras. Abaixo, apresentamos uma implementação em linguagem C:

```
int GCD(int inA, int inB) {
    int swap;
    int done = 0;
    int A = inA;
    int B = inB;
    while (!done) {
        if (A < B) {
            swap = A;
            A = B;
            B = swap;
        } else if (B != 0) {
            A = A - B;
        } else {
            done = 1;
        }
    }
    return A;
}
```

Aqui está uma explicação passo a passo do funcionamento do código:

1. O código declara algumas variáveis locais:

swap

,

done



,

A

e

B

.

`swap`

é usada para trocar os valores de

A

e

B

quando necessário.

`done`

é uma variável de controle que indica se o cálculo do maior divisor comum está concluído.

A

e

B

são inicializadas com os valores dos parâmetros

`inA`

e

`inB`

, respectivamente.

2. O código entra em um loop

`while`

que continua até que a variável

`done`

seja igual a 1.

3. O loop é usado para realizar as iterações necessárias para calcular o maior divisor comum.

4. Dentro do loop, há uma estrutura

**if-else**

que verifica três condições:

(a) Se

A

for menor que

B

, os valores de

A

e

B

são trocados usando a variável

**swap**

. Isso garante que

A

seja sempre o maior número.

(b) Se

B

for diferente de zero,

A

é atualizado para

A - B

. Isso realiza a subtração repetida até que um dos números seja igual a zero.

(c) Se nenhuma das condições anteriores for verdadeira, significa que o cálculo do maior divisor comum está concluído e a variável

**done**

é definida como 1 para sair do loop.

5. Após o loop, o código retorna o valor de

A

, que representa o maior divisor comum entre os números

`inA`

e

`inB`

.

Por exemplo, se chamarmos a função `GCD(24, 36)`, o algoritmo de Euclides será aplicado da seguinte maneira:

1. Na primeira iteração,

`A`

é atualizado para 36 e

`B`

é atualizado para 24.

2. Na segunda iteração,

`A`

é atualizado para 12 e

`B`

é atualizado para 24.

3. Na terceira iteração,

`A`

é atualizado para 12 e

`B`

é atualizado para 12.

4. Na quarta iteração,

`A`

é atualizado para 0 e

`B`

é atualizado para 12.

5. Como

`B`

é igual a zero, o cálculo do maior divisor comum está concluído e o valor retornado é 12.

## 4.2 Modelagem em Verilog

A implementação do GCD em hardware envolve a criação de módulos para o controle e o datapath. O datapath lida com a movimentação e transformação dos dados, enquanto o módulo de controle gerencia as operações de controle.

### 4.2.1 Módulo Datapath

O código a seguir é um exemplo de um módulo em Verilog que implementa o datapath para um algoritmo de cálculo do Máximo Divisor Comum (GCD - Greatest Common Divisor).

```
module GCDdatapath#( parameter W=16 )
(
    input clk,
    input [W-1:0] operand_A,
    input [W-1:0] operand_B,
    output [W-1:0] result_data,
    input A_ld,
    input B_ld,
    input [1:0] A_sel,
    input B_sel,
    output B_zero,
    output A_lt_B,
    output [W-1:0] A_chk, B_chk, sub_chk, A_mux_chk, B_mux_chk
);

    wire [W-1:0] A;
    wire [W-1:0] B;
    wire [W-1:0] sub_out;
    wire [W-1:0] A_mux_out;
    wire [W-1:0] B_mux_out;

    Mux3#(W) A_mux(
        .in0 (operand_A),
        .in1 (sub_out),
        .in2 (B),
        .sel (A_sel),
        .out (A_mux_out)
    );

    register#(W) A_reg (
        .clk (clk),
        .d (A_mux_out),
        .en (A_ld),
        .q (A)
    );
```

```

Mux2#(W) B_mux (
    .in0 (A),
    .in1 (operand_B),
    .sel (B_sel),
    .out (B_mux_out)
);

register#(W) B_reg (
    .clk (clk),
    .d (B_mux_out),
    .en (B_ld),
    .q (B)
);

assign B_zero = (B==0);
assign A_lt_B = (A < B);
assign sub_out = A - B;
assign result_data = A;
// send checking signals only for debugging purposes
assign A_chk = A;
assign B_chk = B;
assign sub_chk = sub_out;
assign A_mux_chk = A_mux_out;
assign B_mux_chk = B_mux_out;
endmodule

```

A seguir, o funcionamento do código passo a passo:

O módulo GCDdatapath recebe alguns parâmetros, sendo o principal deles W, que define o tamanho dos operandos e do resultado em bits.

O módulo possui várias entradas e saídas, incluindo:

- **clk**  
: um sinal de clock para sincronizar as operações.
- **operand\_A**  
e  
**operand\_B**  
: os operandos de entrada para o cálculo do GCD.
- **result\_data**  
: o resultado do cálculo do GCD.
- **A\_ld**  
e

B\_ld

: sinais de controle para carregar os operandos A e B.

- A\_sel

e

B\_sel

: sinais de seleção para escolher entre os operandos A e B.

- B\_zero

: um sinal indicando se o operando B é igual a zero.

- A\_lt\_B

: um sinal indicando se o operando A é menor que o operando B.

- A\_chk

,

B\_chk

,

sub\_chk

,

A\_mux\_chk

e

B\_mux\_chk

: sinais de depuração para verificar os valores internos do datapath.

O módulo possui várias fiações (wires) para conectar os componentes internos:

A, B, sub\_out, A\_mux\_out

e

B\_mux\_out

: fiações para transportar os valores dos operandos e resultados intermediários.

O módulo utiliza vários componentes internos para realizar as operações do datapath:

- Mux3#(W) A\_mux

: um multiplexador de 3 entradas que seleciona entre o operando A, o resultado da subtração e o operando B, com base no sinal de seleção

A\_sel

. O resultado é armazenado em

A\_mux\_out

.

- register#(W) A\_reg

: um registrador que armazena o valor de

A\_mux\_out

e o atualiza no sinal de clock

clk

quando o sinal de controle

A\_ld

está ativo. O valor atualizado é armazenado em

A

.

- Mux2#(W) B\_mux

: um multiplexador de 2 entradas que seleciona entre o valor atualizado de

A

e o operando

B

, com base no sinal de seleção

B\_sel

. O resultado é armazenado em

B\_mux\_out

.

- register#(W) B\_reg

: um registrador que armazena o valor de

B\_mux\_out

e o atualiza no sinal de clock

`clk`

quando o sinal de controle

`B_ld`

está ativo. O valor atualizado é armazenado em

`B`

.

As atribuições assign são usadas para definir os valores das saídas do módulo:

- `B_zero`

é atribuído como verdadeiro se o valor de

`B`

for igual a zero.

- `A_lt_B`

é atribuído como verdadeiro se o valor de

`A`

for menor que o valor de

`B`

.

- `sub_out`

é atribuído como a diferença entre

`A`

e

`B`

.

- `result_data`

é atribuído como o valor de

`A`



- Os sinais de depuração

A\_chk

,

B\_chk

,

sub\_chk

,

A\_mux\_chk

e

B\_mux\_chk

são atribuídos aos respectivos valores internos do datapath.

O módulo GCDdatapath encapsula todas essas operações e fiações para formar um datapath completo para o cálculo do GCD.

#### 4.2.2 Módulo Controle

```
module GCDcontrol(
    input input_available,
    output reg idle,
    input clk, reset,
    output reg A_ld, B_sel, B_ld,
    output reg [1:0] A_sel,
    input B_zero, A_lt_B,
    output reg result_rdy,
    input result_taken,
    output [1:0] State
);

    // States naming
    localparam WAIT = 2'd0;
    localparam CALC = 2'd1;
    localparam DONE = 2'd2;

    // Constants naming for A_mux selector
    localparam A_SEL_IN = 2'b00;
```

```

localparam A_SEL_SUB = 2'b01;
localparam A_SEL_B = 2'b10;
localparam A_SEL_X = 2'b11;
// Constants naming for B_mux selector
localparam B_SEL_A = 1'b0;
localparam B_SEL_IN = 1'b1;
localparam B_SEL_X = 1'bx;

reg [1:0] CurrentState, NextState;

always @(posedge clk or posedge reset)
begin
    if (reset)
        CurrentState <= WAIT;
    else
        CurrentState <= NextState;
end

always @(CurrentState)
begin
    // default is to stay in the same state
    NextState <= CurrentState;
    case ( CurrentState )
        WAIT :
            if ( input_available )
                NextState <= CALC;
        CALC :
            if ( B_zero )
                NextState <= DONE;
        DONE :
            if ( result_taken )
                NextState <= WAIT;
    endcase
end

always @( * )
begin
    // Default control signals
    A_sel <= A_SEL_X;
    A_ld <= 1'b0;
    B_sel <= B_SEL_X;
    B_ld <= 1'b0;
    idle <= 1'b0;
    result_rdy = 1'b0;
end

```

```

    case ( CurrentState )
        WAIT :
            begin
                idle <= 1'b1;
                if(input_available)begin
                    A_sel <= A_SEL_IN;
                    B_sel <= B_SEL_IN;
                    A_ld <= 1'b1;
                    B_ld <= 1'b1;
                end
            end
        CALC :
            if ( A_lt_B )begin
                A_sel <= A_SEL_B;
                B_sel <= B_SEL_A;
                A_ld <= 1'b1;
                B_ld <= 1'b1;
            end
            else if ( !B_zero )begin
                A_sel <= A_SEL_SUB;
                A_ld <= 1'b1;
            end
        DONE :
            result_rdy <= 1'b1;
    endcase
end

    assign State = CurrentState;
endmodule

```

## 5 MÁQUINA DE ESTADOS FINITOS (FSM)

A máquina de estados finitos (FSM) é uma abordagem essencial para controlar o fluxo de operações em um sistema digital, especialmente na implementação de algoritmos como o cálculo do Greatest Common Divisor (GCD). A FSM divide o processo de cálculo em diferentes estados, cada um representando uma etapa específica na operação, e controla a transição entre esses estados com base nas condições de entrada e saídas esperadas. A figura abaixo (FIGURA 1) mostra a FSM utilizada para controlar o datapath do GCD.

### 5.1 Descrição dos Estados

A FSM apresentada na figura contém três estados principais: *WAIT*, *CALC*, e *DONE*. Cada estado desempenha um papel crucial no ciclo de vida do cálculo do *GCD*:

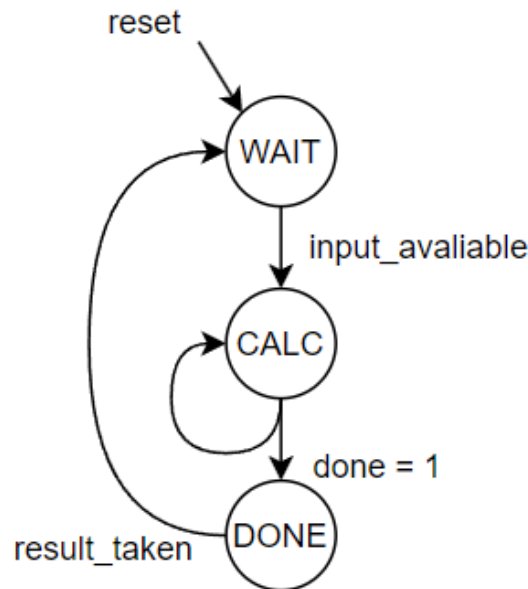


Figura 3: Máquina de Estados Finitos para o GCD

#### 5.1.1 Estado *WAIT* (Aguardando novos operandos)

Este é o estado inicial da FSM, onde o sistema está inativo e aguardando a disponibilidade de novos operandos para iniciar o cálculo. Neste estado, o sistema verifica a sinalização de disponibilidade de entrada (**input\_available**). Quando a sinalização **input\_available** é ativada, indicando que os operandos estão prontos para serem processados, a FSM transita para o estado *CALC*.

#### 5.1.2 Estado *CALC* (Calculando o GCD)

Neste estado, o algoritmo de Euclides é executado, onde os valores dos operandos são trocados e subtraídos repetidamente até que o menor dos dois operandos seja zero. Este processo continua até que a condição de término seja atingida ( $done = 1$ ), o que indica que o GCD foi encontrado. A FSM permanece neste estado enquanto a condição  $done = 1$  não é satisfeita. Quando a condição  $done = 1$  é atingida, o sistema transita para o estado *DONE*.

#### 5.1.3 Estado *DONE* (Esperando o consumidor receber o resultado)

No estado *DONE*, o sistema aguarda que o consumidor ou a unidade de processamento a jusante receba o resultado do cálculo do *GCD*. Durante este estado, o sistema está essencialmente "em espera", até que o sinal **result\_taken** seja ativado. Após o sinal **result\_taken** ser ativado, indicando que o resultado foi processado ou consumido, a *FSM* retorna ao estado *WAIT* para reiniciar o ciclo com novos operandos.

### 5.2 Implementação da FSM

A FSM foi implementada em Verilog, utilizando registradores para armazenar o estado atual e o próximo estado, e lógica combinacional. É possível separar a implementação da FSM em três partes principais: a definição dos estados e sinais de controle, a lógica de transição de estados, e a lógica de controle de sinais.

### 5.2.1 Definição dos Estados e Sinais de Controle

O FSM tem três estados principais definidos como constantes locais (*localparam*):

```
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;
```

Os sinais de controle da FSM são definidos como registradores de saída (*reg*):

```
output reg idle,
output reg A_ld, B_sel, B_ld,
output reg [1:0] A_sel,
output [1:0] State
output reg result_rdy,
```

Também são definidos constantes para os seletores dos multiplexadores A e B:

```
// constants naming for A_mux selector
localparam A_SEL_IN = 2'b00;
localparam A_SEL_SUB = 2'b01;
localparam A_SEL_B = 2'b10;
localparam A_SEL_X = 2'b11;

// constants naming for B_mux selector
localparam B_SEL_A = 1'b0;
localparam B_SEL_IN = 1'b1;
localparam B_SEL_X = 1'bx;
```

e os estados da FSM são definidos como registradores de 2 bits:

```
reg [1:0] CurrentState, NextState;
```

### 5.2.2 Lógica de Controle de Sinais

A FSM é implementada em duas partes principais: a lógica de transição de estados e a lógica de controle de sinais. A lógica de transição de estados é baseada em um bloco *always* que atualiza o estado atual (*CurrentState*) com base no próximo estado (*NextState*):

```
always @(posedge clk or posedge reset)
begin
    // reset the FSM to the initial state when reset signal is active
    if (reset)
        CurrentState <= WAIT;
    else
        CurrentState <= NextState;
end

always @(CurrentState)
```

```

begin
    // default is to stay in the same state
    NextState <= CurrentState;
    case ( CurrentState )
        WAIT :
            if ( input_available )
                NextState <= CALC;
        CALC :
            if ( B_zero )
                NextState <= DONE;
        DONE :
            if ( result_taken )
                NextState <= WAIT;
    endcase
end

```

### 5.2.3 Controle de Sinais

A lógica de controle de sinais é implementada em um bloco *always* que define os sinais de controle com base no estado atual da FSM:

```

always @( * )
begin
    // Default control signals
    A_sel <= A_SEL_X;
    A_ld <= 1'b0;
    B_sel <= B_SEL_X;
    B_ld <= 1'b0;
    idle <= 1'b0;
    result_rdy = 1'b0;

    case ( CurrentState )
        WAIT :
            begin
                idle <= 1'b1;
                if(input_available)begin
                    A_sel <= A_SEL_IN;
                    B_sel <= B_SEL_IN;
                    A_ld <= 1'b1;
                    B_ld <= 1'b1;
                end
            end
        CALC :
            if ( A_lt_B )begin
                A_sel <= A_SEL_B;

```

```

        B_sel <= B_SEL_A;
        A_ld <= 1'b1;
        B_ld <= 1'b1;
    end
    else if ( !B_zero )begin
        A_sel <= A_SEL_SUB;
        A_ld <= 1'b1;
    end
    DONE :
        result_rdy <= 1'b1;
    endcase
end

```

## 6 SIMULAÇÕES E VERIFICAÇÃO

Para validar a implementação, foi realizada a simulação dos módulos utilizando o ambiente Intel QuestaSim. A Figura 4 mostra a simulação do módulo GCD.

### 6.1 Simulação do módulo GCD

Figura 4: Simulação do módulo GCD

Os resultados da simulação confirmam que o design do GCD em hardware funciona conforme esperado. O módulo datapath executa corretamente as operações de subtração e troca, enquanto o módulo de controle gerencia os estados do sistema de maneira eficiente.

### 6.2 Ambiente de Simulação

### 6.3 Resultados da Simulação

## 7 OTIMIZAÇÕES E MELHORIAS FUTURAS

Mesmo se tratando de um algoritmo relativamente simples, o design do GCD em hardware pode ser otimizado e expandido para atender a requisitos mais exigentes. Nesta seção, discutiremos possíveis melhorias no design, como a implementação de pipelining no datapath e a expansão do sistema para suportar operações mais complexas.

### 7.1 Otimização do Datapath

Uma possível otimização é a implementação de pipelining no datapath. Isso permitiria que várias operações de subtração e troca fossem executadas em paralelo, aumentando a eficiência do sistema. Além disso, o uso de registradores adicionais poderia reduzir o número de operações de leitura e escrita na memória, melhorando ainda mais o desempenho.

#### 7.1.1 Implemetação do Pipelining

O pipelining é uma técnica fundamental para aumentar a eficiência e o desempenho de sistemas de processamento. Ao aplicar pipelining no datapath do algoritmo GCD, podemos dividir o processo de

cálculo em estágios distintos que podem ser executados em paralelo, reduzindo o tempo total necessário para a execução e aumentando o throughput (HENNESSY; PATTERSON, 2017).

## 1. Divisão do Algoritmo GCD em Estágios

Para o algoritmo GCD, que geralmente é implementado usando o Algoritmo de Euclides, podemos dividir o processamento em vários estágios:

- **Estágio 1:** Cálculo do Resto - Calcula o resto da divisão entre dois números.
- **Estágio 2:** Atualização dos Valores - Atualiza os valores dos números com base no resto calculado.
- **Estágio 3:** Verificação da Condição de Parada - Verifica se o resto é zero, o que indica que o cálculo está concluído (KNUTH, 1997).

## 2. Implementação do Pipelining

A implementação do pipelining para o GCD pode ser feita da seguinte maneira:

- **Pipeline de Estágios:** Cada estágio do pipeline pode ser otimizado para realizar sua tarefa específica simultaneamente com outros estágios. Por exemplo, enquanto um estágio calcula o resto, outro pode estar atualizando os valores ou verificando a condição de parada.
- **Buffer de Pipeline:** Adicionar buffers entre os estágios para armazenar dados temporários e permitir que o processamento continue sem interrupções.
- **Controle de Fluxo:** Implementar mecanismos de controle para gerenciar a sincronização entre os estágios e garantir que cada estágio receba os dados no momento certo (PATTERSON; HENNESSY, 2013).

### 7.1.2 Benefícios do Pipelining

1. **Redução do Tempo de Execução** Ao dividir o trabalho entre vários estágios e processar diferentes partes do algoritmo simultaneamente, o tempo total para concluir a operação de GCD é reduzido.
2. **Aumento do Throughput:** A capacidade de processar múltiplos cálculos de GCD em paralelo pode aumentar o throughput geral do sistema (HARRIS; HARRIS, 2007).
3. **Paralelismo de dados:**
  - **Execução Paralela:** Implementar execução paralela para processar múltiplos pares de números simultaneamente. Isso pode ser feito usando múltiplos pipelines ou unidades de processamento.
4. **Otimização de Hardware:**
  - **Uso de Recursos Específicos:** Utilizar unidades de hardware específicas para operações matemáticas, como divisores de alto desempenho, para acelerar o cálculo do resto.
  - **Redução de Latência:** Minimizar a latência entre as operações utilizando técnicas de otimização de circuito.



- **Eficiência Energética:** Implementar técnicas para reduzir o consumo de energia, como a otimização do design do circuito e o uso eficiente dos recursos de hardware.

## 5. Algoritmos Alternativos:

- **Exploração de Algoritmos:** Explorar algoritmos alternativos para o cálculo do GCD que possam oferecer melhorias em termos de eficiência e velocidade (SKIENA, 2008).

**Conclusão :** A aplicação de pipelining e outras técnicas de otimização pode significativamente melhorar a eficiência e o desempenho do datapath para o algoritmo GCD. Ao dividir o trabalho em estágios e implementar estratégias para reduzir latência e aumentar o throughput, podemos alcançar um sistema mais rápido e eficiente.

## 7.2 Expansão do Design para Sistemas Maiores

Outra melhoria futura seria a expansão do design para suportar sistemas maiores ou operações mais complexas. Por exemplo, o módulo GCD poderia ser integrado a um processador ou coprocessador específico para criptografia. Isso exigiria a adaptação do módulo para se comunicar com outros componentes do sistema e lidar com algoritmos mais avançados.

## 8 DISCUSSÃO SOBRE INTEGRAÇÃO COM SISTEMAS MAIORES E OTIMIZAÇÃO DO DATAPATH

É importante ressaltar que essas otimizações e melhorias futuras podem trazer desafios adicionais, como a complexidade do design e a necessidade de recursos adicionais. Portanto, é necessário realizar uma análise cuidadosa dos requisitos e restrições do sistema antes de implementar essas melhorias.

Em resumo, as otimizações e melhorias futuras no design do algoritmo GCD em hardware são uma área de interesse contínua para estudantes de Engenharia de Computação. Com a combinação certa de conhecimento teórico e prático, é possível criar soluções eficientes e inovadoras que atendam às demandas cada vez maiores da computação moderna.

## 9 CONCLUSÕES

Neste artigo, apresentamos a implementação do algoritmo GCD em hardware, detalhando a modelagem dos módulos de controle e datapath em Verilog. A simulação mostrou que a abordagem utilizada é eficiente e atende aos requisitos do projeto. Futuras melhorias podem incluir a otimização do datapath e a integração com outros módulos de um sistema maior.

## REFERÊNCIAS

- HARRIS, D. M.; HARRIS, S. L. *Digital Design and Computer Architecture*. [S.l.]: Morgan Kaufmann, 2007.
- HENNESSY, J. L.; PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*. 6th. ed. [S.l.]: Morgan Kaufmann, 2017.
- KNUTH, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. 3rd. ed. [S.l.]: Addison-Wesley, 1997.

PATTERSON, D. A.; HENNESSY, J. L. *Computer Organization and Design: The Hardware/Software Interface*. 5th. ed. [S.l.]: Morgan Kaufmann, 2013.

SKIENA, S. S. *The Algorithm Design Manual*. 2nd. ed. [S.l.]: Springer, 2008.

TAYLOR, M. B. *CSE 141L: Introduction to Computer Architecture Laboratory*. [S.l.], 2019. Disponível em: <<https://cseweb.ucsd.edu/classes/sp10/cse141L/pdf/02/02-Verilog2.pdf>>. Acesso em: 13 Ago. 2024.