

**Báo cáo bài tập lớn**

**Bộ đếm 3-bit nhị phân dti\_bincnt\_ckrp**

***Nhóm 22***

***Revision 1.0***

|  |  |  |
| --- | --- | --- |
| Người hướng dẫn: TS. Nguyễn Vũ Thắng và đội ngũ công ty Dolphin Technology  Nhóm sinh viên thực hiện: Nhóm 22, các thành viên:   |  |  | | --- | --- | | Lê Chí Tuyền | 20193191 | |

**13 January 2023**

**Nhóm 22**

**Báo cáo bài tập lớn**

**Bộ đếm 3-bit nhị phân dti\_bincnt\_ckrp**

**Revision 1.0**

**13 January 2023**

# TABLE OF CONTENTS

[TABLE OF CONTENTS 2](#_Toc156000785)

[LIST OF FIGURES 3](#_Toc156000786)

[LIST OF TABLES 4](#_Toc156000787)

[REVISION HISTORY 5](#_Toc156000788)

[1. QUY TRÌNH THIẾT KẾ VÀ PHÂN CHIA CÔNG VIỆC 6](#_Toc156000789)

[1.1. Danh sách thành viên 6](#_Toc156000790)

[1.2. Quy trình thiết kế 6](#_Toc156000791)

[1.3. Ngôn ngữ lập trình Verilog 7](#_Toc156000792)

[2. DỰ ÁN: Bộ đếm 3-bit nhị phân dti\_bincnt\_ckrp 12](#_Toc156000793)

[2.1. Thông tin dự án 12](#_Toc156000794)

[2.2. Thiết kế module 12](#_Toc156000795)

[2.2.1. Timing diagram 12](#_Toc156000796)

[2.2.2. Mô tả kiến trúc 14](#_Toc156000797)

[2.2.3. Triển khai code RTL bằng phần mềm Vivado 14](#_Toc156000798)

[2.3. Triển khai kiểm thử thiết kế sử dụng System Verilog 19](#_Toc156000799)

[2.3.1. Kiểm tra trạng thái done của bộ đếm 19](#_Toc156000800)

[2.3.2. Kiểm tra quy tắc của bộ đếm 22](#_Toc156000801)

[3. Tổng kết 25](#_Toc156000802)

[4. Tài liệu tham khảo 26](#_Toc156000803)

[5. PHỤ LỤC 27](#_Toc156000804)

[5.1. Code WaveForm 27](#_Toc156000805)

# LIST OF FIGURES

[1‑1 Quy trình thiết kế VLSI - VLSI design flow 6](#_Toc156000468)

[2‑1 Timing diagram 13](#_Toc156000469)

[2‑2 Timing diagram 14](#_Toc156000470)

[2‑3 Mô tả kiến trúc 14](#_Toc156000471)

[2‑4 Tạo project mới 15](#_Toc156000472)

[2‑5 đặt tên project 15](#_Toc156000473)

[2‑6 Các bước tiến hành tạo code 16](#_Toc156000474)

[2‑7 Waveform thực hiện lệnh đếm 19](#_Toc156000475)

[2‑8 Waveform thực hiện lệnh đếm khi thay đổi đầu vào 22](#_Toc156000476)

# LIST OF TABLES

[Bảng 1. Các lần sửa đổi 5](#_Toc156000668)

[Bảng 2. Mô tả tín hiệu Input – Output của bộ đếm dti\_bincnt\_ckprn 12](#_Toc156000669)

[Bảng 3. Quy tắc đếm dti\_bincnt\_ckprn 12](#_Toc156000670)

# REVISION HISTORY

Bảng 1. Các lần sửa đổi

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date** | **Description of Changes** |
| 1.0 | 13 Oct 2023 | Viết Form báo cáo và phân chia công việc trong nhóm |
| 1.1 | 15 November 2023 | Thêm Code và Waveform |
| 1.2 | 13/1/2024 | Viết báo cáo code RTL và phần test case |

# QUY TRÌNH THIẾT KẾ VÀ PHÂN CHIA CÔNG VIỆC

## Danh sách thành viên

* Lê Chí Tuyền

Teams: **Lê Chí Tuyền 20193191 (tuyen.lc193191@sis.hust.edu.vn)**

Zalo: 0359581461

## Quy trình thiết kế

Quy trình thiết kế VLSI có thể được biểu diễn bởi 5 bước cơ bản, bao hàm các thao tác cần thực hiện để có thể đưa ra bản vẽ kĩ thuật cho mạch, gồm:

A diagram of a process

Description automatically generated

1‑1 Quy trình thiết kế VLSI - VLSI design flow

* HDL Coding - Lập trình mô tả phần cứng.
* Mạch điện tử số có thể được mô tả thông qua lập trình, dựa trên chức năng/hành vi của mạch (behavioral), cấu trúc (structural), đặc điểm luồng dữ liệu (data-flow).
* Sử dụng các ngôn ngữ mô tả phần cứng (HDL) như Verilog, VHDL, SystemVerilog, v.v..
* Logic Verification – Kiểm thử logic.
* Được sử dụng để kiểm tra logic của phần cứng được thiết kế trên HDL, đảm bảo tính hoạt động đúng đắn của mạch khi được thiết kế.
* Logic Synthesis & Timing Analysis – Tổng hợp logic & Phân tích hiệu năng thời gian.
* Dựa trên kết quả lập trình HDL đã đúng logic, tiến hành chuyển đổi mô tả của mạch dưới dạng chức năng/cấu trúc sang tập các cổng logic thực tế & liên kết giữa các cổng cần thiết để tạo ra mạch thật.
* Phân tích hoạt động của mạch về mặt thời gian (timing analysis) để xác định hiệu năng hoạt động, từ đó tạo các ràng buộc/giới hạn cho thiết kế.
* Physical Layout – Tạo bản vẽ vật lý.
* Dựa vào thiết kế đã được dịch sang tập các cổng logic & liên kết, cùng với thư viện công nghệ & các ràng buộc thiết kế (hiệu năng thời gian, diện tích, năng lượng, …), tiến hành tạo bản vẽ vật lý của mạch
* Physical Verification & Sign-off – Kiểm thử vật lý & Hoàn thiện.
* Dựa trên kết quả bản vẽ vật lý đã thu được, tiến hành:
* Physical Verification:
  + Kiểm tra bản vẽ của mạch, dựa trên các bộ quy định (design rule) được cung cấp từ phía nhà sản xuất (foundry). Bản vẽ phải đảm bảo không có lỗi so với bộ quy định trên Design Rule Check (DRC)
  + Kiểm tra độ trùng khớp giữa sơ đồ nguyên lý (schematic) so với bản vẽ (layout). Về mặt logic, sơ đồ nguyên lý & bản vẽ phải tương đồng.
  + Layout Vs Schematic (LVS)
* Post-Timing Analysis: Kiểm tra hiệu năng thời gian sau khi đã có bản vẽ. Thiết kế với bản vẽ thu được phải đảm bảo các đường dữ liệu hoạt động theo đúng giới hạn thời gian, không phát sinh trạng thái không xác định trên toàn mạch.

## Ngôn ngữ lập trình Verilog

Verilog là một ngôn ngữ mô tả phần cứng (HDL - Hardware Description Language) được sử dụng để mô phỏng, thiết kế và xác định các hệ thống kỹ thuật số. Được phát triển vào những năm 1980, Verilog đã trở thành một trong những ngôn ngữ phổ biến nhất trong lĩnh vực thiết kế vi mạch.

Verilog được sử dụng để mô tả hành vi và cấu trúc của các mạch kỹ thuật số, từ mạch đơn giản như cổng logic cho đến mạch phức tạp như vi xử lý. Ngôn ngữ này cung cấp cú pháp và cấu trúc cho việc mô phỏng và thiết kế hệ thống số thông qua việc sử dụng các module, tín hiệu, biến, và các câu lệnh điều khiển.

Verilog cung cấp một cách để mô phỏng hoặc tạo ra một mô hình của hệ thống kỹ thuật số và cho phép kiểm tra và xác minh tính đúng đắn của nó trước khi triển khai vào phần cứng thực tế. Nó cũng được sử dụng trong quá trình thiết kế vi mạch, từ việc mô phỏng và kiểm tra đơn giản cho đến tổ chức và xác định các mạch phức tạp hơn.

Trên thực tế, Verilog đã trở thành một ngôn ngữ chuẩn trong lĩnh vực thiết kế mạch tích hợp và thường được sử dụng trong các công cụ thiết kế như Xilinx ISE, Altera Quartus, và Cadence Design Systems.

1. Module

Trong Verilog, một module là một đơn vị cơ bản để mô tả một phần của hệ thống kỹ thuật số. Nó được sử dụng để định nghĩa các thành phần riêng lẻ của mạch và xác định cách chúng tương tác với nhau. Mỗi module trong Verilog có thể được xem như một "khối xây dựng" trong thiết kế phần cứng.

Một module trong Verilog bao gồm các phần chính sau:

* Tên module: Mỗi module được đặt tên duy nhất để xác định nó trong thiết kế.
* Khối chứa: Module có thể chứa các khối khác như input, output, wire, reg, và các module con khác. Các khối này đại diện cho tín hiệu, biến và các thành phần khác trong mạch.
* Cổng vào và cổng ra: Module có thể có các cổng vào (input) và cổng ra (output) để tương tác với các module khác hoặc với môi trường bên ngoài.
* Internal logic: Module có thể chứa các khối logic và câu lệnh điều khiển để thiết lập hành vi bên trong.
* Instantiation: Module có thể được sử dụng như một thành phần trong các module khác bằng cách sử dụng cú pháp instantiation. Điều này cho phép tái sử dụng và phân cấp các module để xây dựng mạch phức tạp hơn.

Module trong Verilog cung cấp một cách để tạo ra các phần tử mạch đơn giản và kết hợp chúng thành mạch phức tạp hơn. Nó giúp tăng tính tái sử dụng, hiệu suất và khả năng mô phỏng của quá trình thiết kế vi mạch.

Dưới đây là một ví dụ đơn giản về mô tả module cho cổng AND 2 bit đầu vào:

module **AND\_gate**(input a, input b, output y);

    assign y = a & b;

endmodule

Trong ví dụ này, chúng ta định nghĩa một module có tên là "AND\_gate". Module này có hai cổng vào (input) là "a" và "b", và một cổng ra (output) là "y". Các cổng vào và cổng ra được khai báo bằng từ khóa "input" và "output".

Dòng "assign y = a & b;" xác định rằng giá trị của cổng ra "y" là kết quả của phép AND logic giữa cổng vào "a" và "b". Dấu "&" đại diện cho phép AND logic trong Verilog. Module "AND\_gate" này có chức năng thực hiện phép AND logic giữa hai tín hiệu đầu vào và đưa ra kết quả qua cổng ra.

1. Kiểu dữ liệu

Trong Verilog, có nhiều kiểu dữ liệu (data types) khác nhau để đại diện cho các loại giá trị và tín hiệu trong mạch kỹ thuật số. Dưới đây là một số kiểu dữ liệu phổ biến trong Verilog:

* Bit và logic: Kiểu dữ liệu 'bit' và 'logic' đại diện cho một bit duy nhất có giá trị logic 0 hoặc 1. Cả hai kiểu dữ liệu này có thể được sử dụng để biểu diễn các tín hiệu logic đơn giản.
* Integer: Kiểu dữ liệu 'integer' đại diện cho một số nguyên có dấu trong khoảng từ -2^31 đến 2^31-1. Nó được sử dụng để biểu diễn các giá trị số nguyên trong mạch.
* Reg: Kiểu dữ liệu 'reg' đại diện cho một thanh ghi (register). Nó được sử dụng để lưu trữ các giá trị tạm thời trong quá trình thực hiện mạch kỹ thuật số.
* Wire: Kiểu dữ liệu 'wire' đại diện cho một tín hiệu không đồng bộ (asynchronous signal) hoặc một liên kết điểm (net). Nó được sử dụng để kết nối các thành phần trong mạch và truyền giá trị giữa chúng.
* Reg và wire vector: Verilog cũng hỗ trợ các kiểu dữ liệu vectơ, bao gồm 'reg [n:0]' và 'wire [n:0]', trong đó n là số bit của vectơ. Các kiểu dữ liệu vectơ này cho phép lưu trữ và xử lý các tín hiệu có độ rộng lớn hơn một bit.
* Struct và Union: Verilog cung cấp cú pháp để định nghĩa các kiểu dữ liệu struct và union tương tự như trong ngôn ngữ lập trình. Các kiểu dữ liệu này cho phép nhóm các thành phần dữ liệu có cấu trúc hoặc không liên quan vào một đối tượng đơn.

Ngoài ra, Verilog còn hỗ trợ các kiểu dữ liệu khác như 'real' (đại diện cho số thực) và các kiểu dữ liệu tùy chỉnh được định nghĩa bởi người dùng. Các kiểu dữ liệu này cung cấp linh hoạt trong việc mô phỏng và biểu diễn các giá trị và tín hiệu trong mạch kỹ thuật số.

Phép gán

Trong Verilog, các phép gán dùng để gán giá trị cho một biến hoặc một tín hiệu. Nó cho phép ta thiết lập giá trị của một biến dựa trên giá trị của các biến khác, hoặc thiết lập giá trị của một tín hiệu dựa trên các điều kiện và logic khác. Có 3 kiểu cơ bản:

|  |  |
| --- | --- |
| always **@** (*condition*) | Luôn thực thi khi thỏa mãn điều kiện (condition) |
| initial | Chỉ thực thi một lần kể từ lúc mô phỏng |
| assign [LHS] = [RHS] | Giá trị LHS luôn được cập nhật khi RHS thay đổi |

1. Một số lưu ý:

* reg chỉ có thể được gán trong các khối initial và always
* wire chỉ có thể được gán thông qua câu lệnh assign
* Nếu như có nhiều dòng lệnh trong khối **initial/always**, cần phải đưa vào trong cặp begin .. end

Dưới đây là một ví dụ về phép gán:

module **Example**;

    reg a, b;

    wire y;

    assign y = a & b;

    initial begin

        a = 1;

        b = 0;

**$display**("y = %b", y);

    end

endmodule

Trong ví dụ này, chúng ta định nghĩa một module có tên là "Example". Module này bao gồm hai biến "a" và "b" kiểu "reg", và một tín hiệu "y" kiểu "wire". Dòng assign y = a & b; thiết lập giá trị của tín hiệu "y" dựa trên phép AND logic giữa "a" và "b".

Trong khối "initial", chúng ta gán giá trị cho biến "a" và "b" là 1 và 0 tương ứng. Sau đó, ta sử dụng $display để hiển thị giá trị của tín hiệu "y" trên màn hình. Kết quả sẽ được hiển thị là "y = 0", vì khi "a = 1" và "b = 0", phép AND logic trả về giá trị 0.

Hierarchy

Hierarchy trong Verilog là khả năng tổ chức mô hình thiết kế thành các cấu trúc phân cấp và nhóm các module lại với nhau. Điều này cho phép xây dựng mạch phức tạp từ các thành phần nhỏ hơn và giúp quản lý và tái sử dụng mã nguồn một cách hiệu quả.

Trong Verilog, hierarchy cho phép định nghĩa các module con bên trong module chính, và các module con có thể chứa các module con khác. Bằng cách sử dụng cú pháp ".tên\_module\_con(tên\_tín\_hiệu)", chúng ta có thể kết nối các module con với nhau và với các tín hiệu ở mức cao hơn. Dưới đây là một ví dụ:

module **TopLevel**;

*// Khai báo module con*

    sub\_module1 U1 (.a(a), .b(b));

    sub\_module2 U2 (.c(c), .d(d));

*// Khai báo tín hiệu*

    wire a, b, c, d, result;

*// Gắn kết tín hiệu*

    assign result = c & d;

*// Các khối logic khác ở mức cao hơn*

*// ...*

endmodule

module **sub\_module1** (input a, input b);

*// Logic của module con 1*

*// ...*

endmodule

module **sub\_module2** (input c, input d);

*// Logic của module con 2*

*// ...*

endmodule

Trong ví dụ này, chúng ta có module TopLevel là module cấp cao nhất trong mô hình thiết kế. Nó chứa hai module con là sub\_module1 và sub\_module2. Các tín hiệu a, b, c, d và result được khai báo ở mức topLevel và được sử dụng trong các module con.

Mỗi module con (sub\_module1 và sub\_module2) có cấu trúc và logic của riêng nó. Chúng được gắn kết với các tín hiệu a, b, c và d tương ứng thông qua cú pháp đã đề cập ở trên.

# DỰ ÁN: Bộ đếm 3-bit nhị phân dti\_bincnt\_ckrp

## Thông tin dự án

Thiết kế bộ đếm nhị phân 3-bit ***dti\_bincnt\_ckprn*** với các đầu vào – ra, và chức năng như sau:

Bảng 2. Mô tả tín hiệu Input – Output của bộ đếm dti\_bincnt\_ckprn

|  |  |  |  |
| --- | --- | --- | --- |
| **Tên Tín Hiệu** | **Độ Rộng** | **Chiều** | **Mô Tả** |
| clk | 1 | Input | Clock, mạch tuần tự hoạt động đồng bộ theo sườn dương của clock này |
| reset\_n | 1 | Input | Reset không đồng bộ, tích cực mức thấp |
| count\_to | 3 | Input | Giá trị cần đếm, là một số nguyên không dấu |
| load | 1 | Input | Nạp giá trị cần đếm |
| count\_en | 1 | Input | Cho phép đếm |
| done | 1 | Output | Cờ báo hiệu hoàn thành việc đếm |

Yêu cầu chức năng:

* Mạch đồng bộ hoạt động theo sườn dương của clock đầu vào.
* Đầu ra ***done*** được chốt bằng Flip Flop (Nói cách khác, ***done*** được nối trực tiếp với đầu ra Q hoặc QN của 1 Flip Flop).
* Giá trị của bộ đếm, tạm gọi là ***count***, được nạp mới hoặc thay đổi giá trị tại sườn dương clock theo quy tắc trong bảng sau:

Bảng 3. Quy tắc đếm dti\_bincnt\_ckprn

|  |  |  |  |
| --- | --- | --- | --- |
| **load** | **count\_en** | **done** | **count** |
| 1’b1 | x | x | Nạp mới, giá trị nạp được xác định bởi **count\_to** |
| 1’b0 | 1’b1 | 1’b0 | Giảm 1 đơn vị |
| 1’b0 | 1’b1 | 1’b1 | Giữ nguyên giá trị |
| 1’b0 | 1’b0 | x | Giữ nguyên giá trị |
| Chú ý: x = don’t care | | | |

* Trước khi có ***load*** = 1, cờ ***done*** giữ giá trị 1’b0. Cờ ***done*** bật lên 1’b1 cùng lúc với ***count*** = 0 và giữ nguyên cho tới lần ***load*** tiếp theo.

## Thiết kế module

### Timing diagram

* Đối với việc đếm

A screenshot of a computer

Description automatically generated

2‑1 Timing diagram

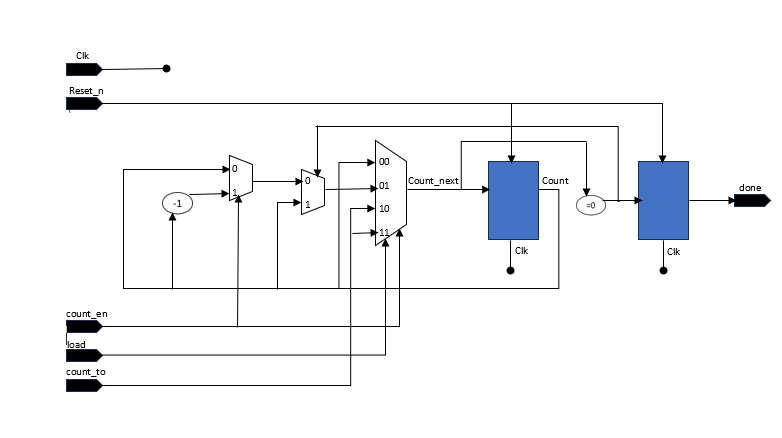
* Sử dụng quy tắc đếm (gán count\_en = 0)

A close-up of a computer screen

Description automatically generated

2‑2 Timing diagram

### Mô tả kiến trúc



2‑3 Mô tả kiến trúc

Vivado Design Suite là tổ hợp các phần mềm của hãng Xilinx. Phần mềm này được tạo ra bằng việc nâng cấp các thế hệ phần mềm thiết kế cũ ISE Design Suite. Vivado được dùng để phát triển các ứng dụng trên thế hệ chip và board Xilinx® 7 series, Zynq®-7000 All Programmable, UltraScale™ devices. Các thế hệ board cũ của Xilinx như Series 6 sẽ vẫn được hỗ trợ bởi ISE, Plan Ahead ...

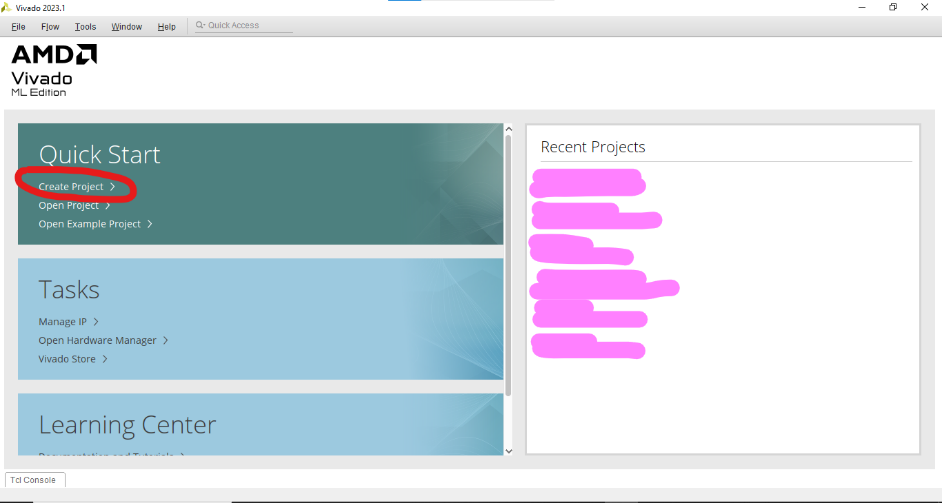
### Triển khai code RTL bằng phần mềm Vivado

Vivado Design Suite là tổ hợp các phần mềm của hãng Xilinx. Phần mềm này được tạo ra bằng việc nâng cấp các thế hệ phần mềm thiết kế cũ ISE Design Suite. Vivado được dùng để phát triển các ứng dụng trên thế hệ chip và board Xilinx® 7 series, Zynq®-7000 All Programmable, UltraScale™ devices. Các thế hệ board cũ của Xilinx như Series 6 sẽ vẫn được hỗ trợ bởi ISE, Plan Ahead ...

Vivado là phần mềm có rất nhiều chức năng. Nói một cách ngắn gọn, nó hỗ trợ tất cả các khâu của quá trình thiết kế Logic sử dụng FPGA. Các bạn sẽ hiểu cụ thể hơn khả năng của Vivado bằng cách tham khảo thêm về quy trình thiết kế FPGA tại đây.

Sau đây em sẽ tiến hành các bước thiết kế module của bài tập lớn. Đầu tiên, cài đặt Vivado theo hướng dẫn sau: <https://youtu.be/DIOll3P65hg>

Sau khi thực hiện các bước trong video, chúng ta tiến hành tạo project trên Vivado. Đầu tiên mở phần mềm và tạo project mới:



2‑4 Tạo project mới

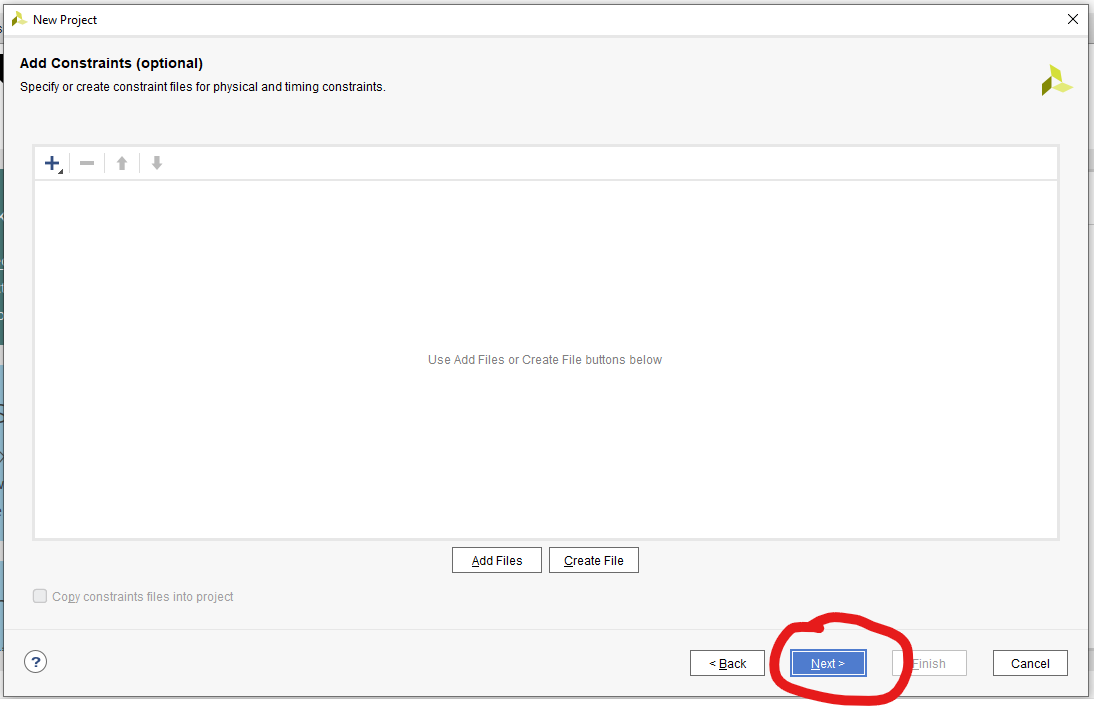
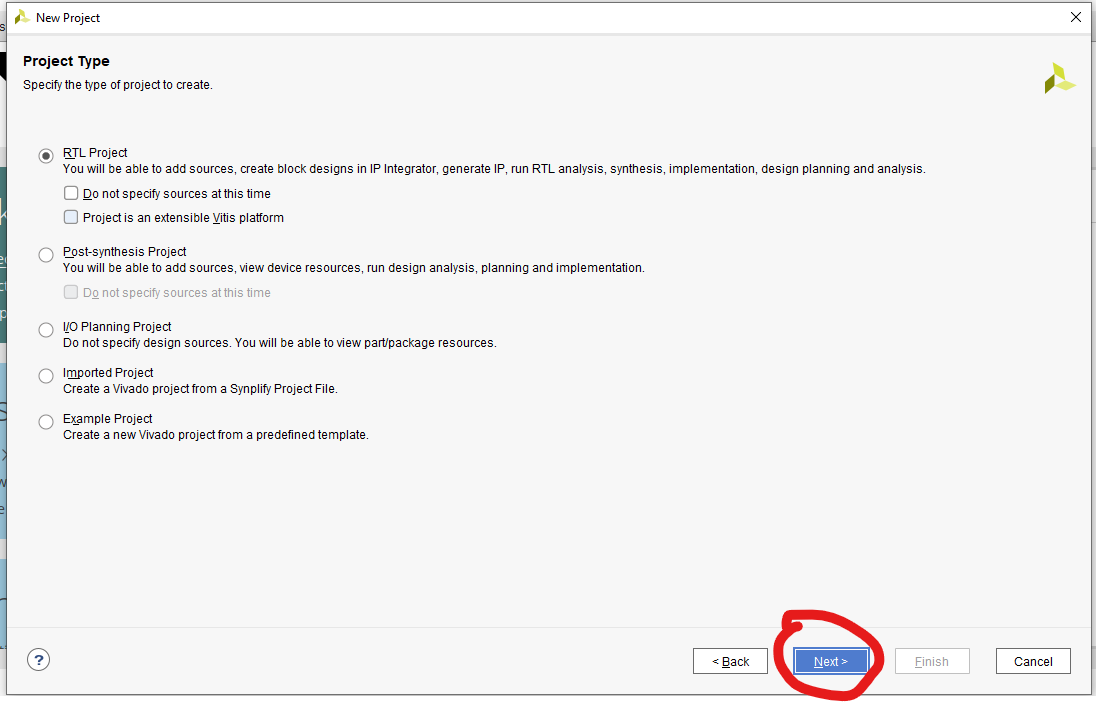
Tạo một project với việc đặt tên và tìm thư mục

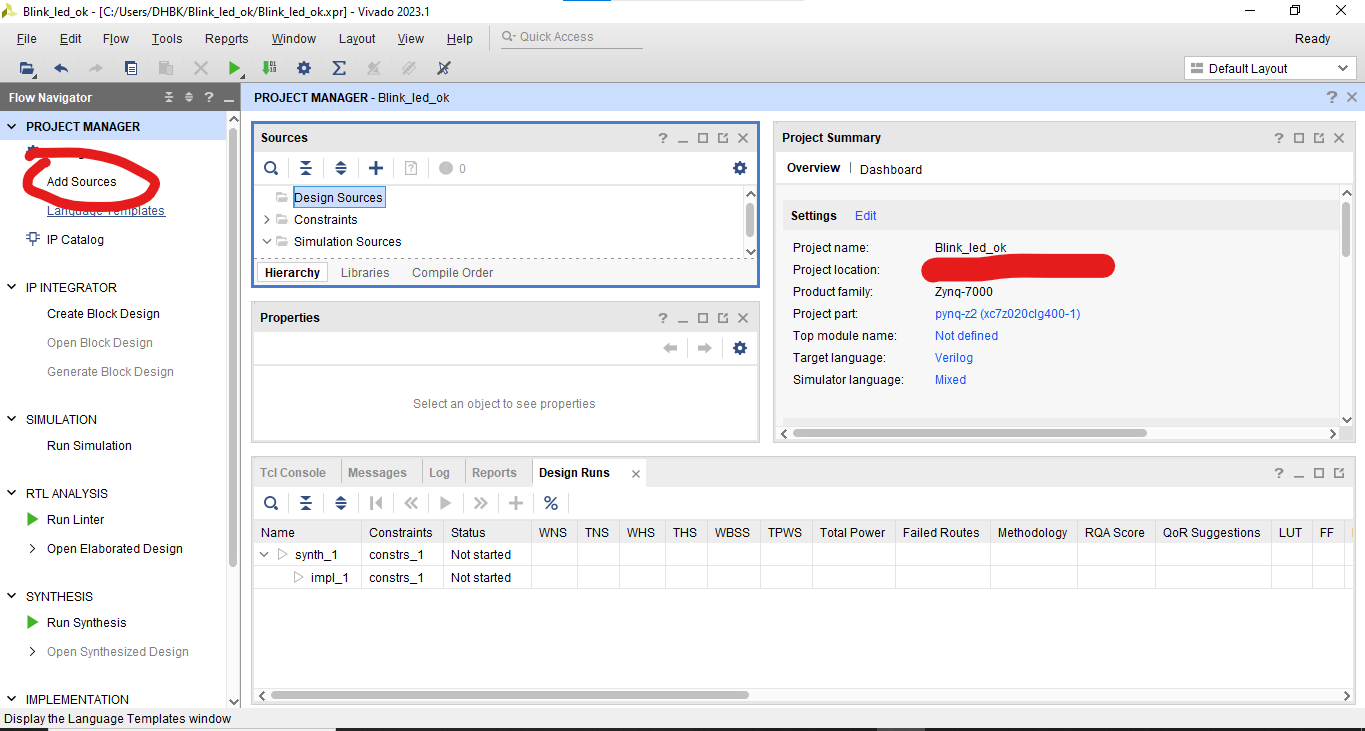
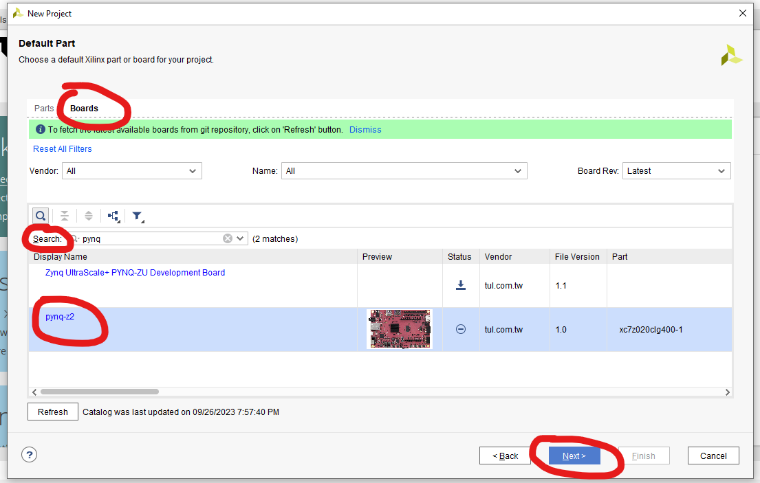
A screenshot of a computer

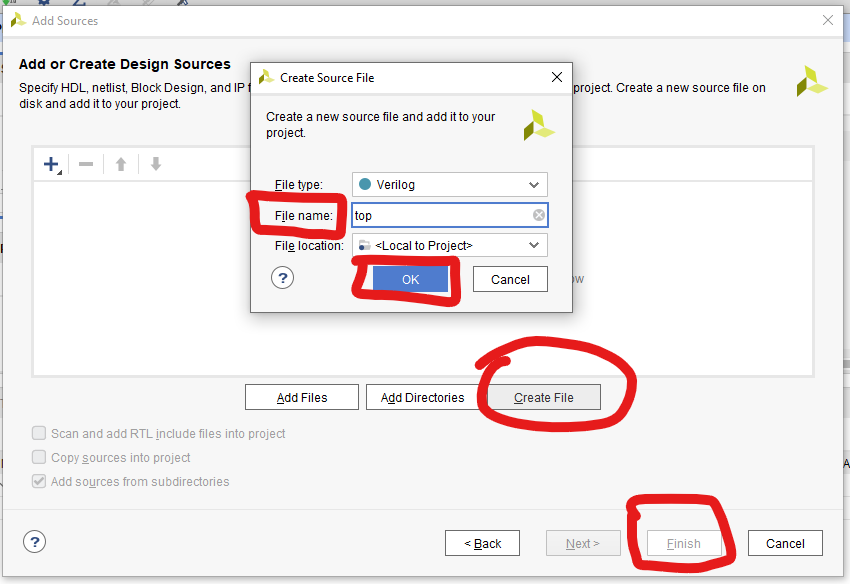
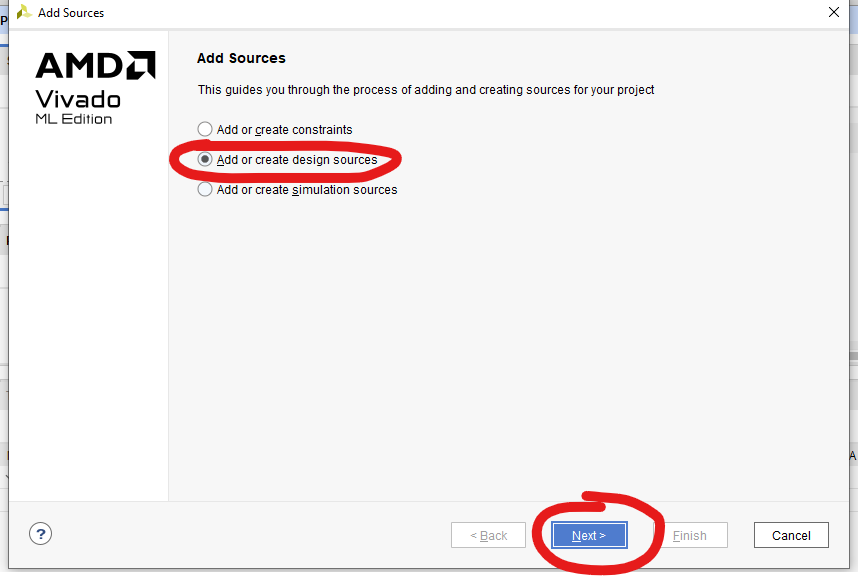
Description automatically generated

2‑5 đặt tên project

Tiếp theo là bỏ qua các bước khởi tạo và đi thẳng đến bước thêm code vào project







2‑6 Các bước tiến hành tạo code

- Đoạn code khai báo cho khối bộ đếm nhị phân 3bit dti\_bincnt\_ckprm, với các tham số đầu vào và đầu ra

module dolphin\_counter (clk, reset\_n, count\_to, count\_en, load, done);

input [2:0] count\_to;

input clk, reset\_n, count\_en, load;

reg [2:0] count, count\_next, count\_mux;

reg pre\_done;

output reg done;

* Count\_to, clk, reset\_n, count\_en, load: là các biến input đầu vào cho module
* Tín hiệu done: đầu ra của bộ đếm
* Biến count để đếm trong hệ thống, count\_next là đầu ra sau mux 4-1 còn count\_mux là đầu ra sau 2 bộ mux 2-1

- Khai báo Flip-Flop

// count Flip-Flop

always @(posedge clk or negedge reset\_n)

begin

if (reset\_n == 0) count <= 0;

else count <= count\_next;

end

* Tín hiệu count được tính bằng cách kiểm tra tín hiệu reset\_n đầu vào. Nếu reset\_n bằng 0 (mức reset), count được gán bằng 0, nếu không thì count gán bằng count\_next

- Mux 4-1

// Mux 4-1

always @ (\*)

begin

case ({count\_en, load})

2'b00 : count\_next = count;

2'b01 : count\_next = count\_to;

2'b10 : count\_next = count\_mux;

2'b11 : count\_next = count\_to;

default: count = 3'b000;

endcase

end

* Đoạn mux để kiểm tra giá trị của count\_en và load dựa trên bẳng Bảng . Quy tắc đếm dti\_bincnt\_ckprn

- 2 Mux 2-1

// 2 Mux 2-1

always @(\*)

begin

if (count\_en == 1'b1)

if (count != 1'b0)

count\_mux = count - 1;

else

count\_mux = count;

else

count\_mux = count;

end

* Dòng đầu tiên của đoạn code kiểm tra xem tín hiệu reset\_n có giá trị logic 1 hay không. Nếu reset\_n là logic 0 thì giá trị của coun\_mux được gán là giá trị count
* Ngược lại, nếu reset\_n là logic 1, tức là tín hiệu reset không được kích hoạt, ta sẽ đi đến kiểm tra cờ done (đếm count về 0)
* Nếu giá trị của count là khác 0, tức là vẫn có thể đếm, ta gán giá trị count\_mux bằng giá trị biến count giảm đi 1 đơn vị. Và ngược lại, nếu count mà bằng 0 là không thể đếm nữa, ta giữ nguyên giá trị count\_mux là count

- Check Done

// Check done

always @(count\_next)

begin

if (count\_next == 3'b000) pre\_done <= 1;

else pre\_done <= 0;

end

* Nếu giá trị count\_nex là 0, ta gán cờ pre\_done là 1 (đã đếm xong) còn không, giá trị là 0 (chưa xong)

- Done Flip-FLop

// Done Flip-Flop

always @(posedge clk or negedge reset\_n)

begin

if (reset\_n == 0) done <= 0;

else done <= pre\_done;

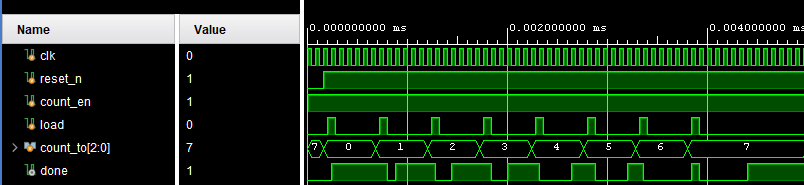
end

* Nếu reset\_n là 0 (mức reset) thì gán tín hiệu done là 0, ngược lại gán done là giá trị của pre\_done.

## Triển khai kiểm thử thiết kế sử dụng System Verilog

* Kiểm tra việc đọc và ghi vào bộ đếm (count\_to)
* Thực hiện ghi dữ liệu từ count\_to
* Kiểm tra giá trị done sau các lần đếm
* Kiểm tra trạng thái done của bộ đếmKiểm tra việc đọc và ghi vào bộ đếm
* Kiểm tra xem đã hoạt động đúng quy tắc đếm dti\_bincnt\_ckpm chưa (reset, count\_en, load)

### Kiểm tra trạng thái done của bộ đếm



2‑7 Waveform thực hiện lệnh đếm

**Các bước thực hiện:**

Bước 1: Đặt giá trị của biến count\_to lần lượt từ 0b000 đến 0b111, đặt giá trị count\_en và reset\_n là 1

Bước 2: Tăng giá trị load lên cao và xuống thấp trong 1 clock

Bước 3: Kiểm tra xem biến count đã đếm đúng chưa

**Cách triển khai code :**

* Tạo vòng lặp giá trị count\_to
* Lặp lại trong xung clk
* Kiểm tra chân done kéo lên có đúng nhịp clock hay không

|  |  |
| --- | --- |
| 5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | **`timescale** **1**ns / **1**ps  **module** tb\_dolphin\_counter();  **reg** clk, reset\_n, count\_en, load;  **reg** [**2**:**0**] count\_to;  **wire** done;  // Biến để ghi lại thời gian  **integer** start\_time, end\_time, done\_active\_time;  **integer** i;  dolphin\_counter **A** (clk, reset\_n, count\_to, count\_en, load, done);  **initial** **begin**  // Khởi tạo các giá trị ban đầu  clk = **1**;  reset\_n = **0**;  load = **0**;  count\_en = **1**;  count\_to = **3'b111**; // Bắt đầu từ 3b111  // Đặt các trạng thái ban đầu  #**160** reset\_n = **1**;  // Vòng lặp giảm dần count\_to từ 3b111 xuống 3b000  **for** (i = **7**; i >= **0**; i = i - **1**) **begin**  count\_to = i[**2**:**0**]; // Cập nhật giá trị count\_to  #**40** load = **1**;  #**80** load = **0**;  start\_time = $time; // Ghi lại thời gian bắt đầu  // Đợi chân done chuyển sang 1  **wait** (done == **1'b1**);  end\_time = $time; // Ghi lại thời gian khi done chuyển sang 1  done\_active\_time = end\_time - start\_time; // Tính thời gian hoạt động của done  $display("Time for done to become %d: %d ns in %d",count\_to, done\_active\_time, (done\_active\_time + **60**)/**80**);  // Dừng mô phỏng  #**100** $stop;  **end**  // Dừng mô phỏng sau khi hoàn thành  **end**  // Tạo xung đồng hồ liên tục  **always** #**40** clk = ~clk;  **endmodule** |

* Kết quả mô phỏng

**INFO:** [**USF**-XSim-**96**] XSim completed. Design snapshot 'tb\_dolphin\_counter\_behav' loaded.

**INFO:** [**USF**-XSim-**97**] XSim simulation ran **for** **1000**ns

**launch\_simulation:** Time (s): cpu = **00**:**00**:**01** ; elapsed = **00**:**00**:**05** . Memory (**MB**): peak = **2851.176** ; gain = **0.000**

restart

**INFO:** [Wavedata **42**-**604**] Simulation restarted

Time **for** done to become **0**: **0** ns in **0** Clock

Time **for** done to become **1**: **60** ns in **1** Clock

Time **for** done to become **2**: **100** ns in **2** Clock

Time **for** done to become **3**: **180** ns in **3** Clock

Time **for** done to become **4**: **260** ns in **4** Clock

Time **for** done to become **5**: **340** ns in **5** Clock

Time **for** done to become **6**: **420** ns in **6** Clock

Time **for** done to become **7**: **500** ns in **7** Clock

**Nhìn vào kết quả mô phỏng ta thấy:**

Bộ đếm hoạt động chính xác, trạng thái load và nhảy cờ done đã hoạt động chính xác.=> PASS

### Kiểm tra quy tắc của bộ đếm

A screenshot of a computer

Description automatically generated

2‑8 Waveform thực hiện lệnh đếm khi thay đổi đầu vào

**Các bước thực hiện:**

Bước 1: Thử nghiệm đang đếm thì thay đổi giá trị count\_en (load = 0)

Bước 2: Kéo tín hiệu reset\_n xuống 0 trong lúc đang đếm (load = 0)

Bước 3: Thay đổi giá trị load trong lúc đếm

**Cách triển khai code :**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59 | **`timescale** **1**ns / **1**ps  **module** tb\_dolphin\_counter();  **reg** clk, reset\_n, count\_en, load;  **reg** [**2**:**0**] count\_to;  **wire** done;  // Biến để ghi lại thời gian  **integer** start\_time, end\_time, done\_active\_time;  **integer** i;  dolphin\_counter **A** (clk, reset\_n, count\_to, count\_en, load, done);  **initial** **begin**  // Khởi tạo các giá trị ban đầu  clk = **1**;  reset\_n = **0**;  load = **0**;  count\_en = **1**;  count\_to = **3'b011**; // Bắt đầu từ 3b011  // Đặt các trạng thái ban đầu  #**160** reset\_n = **1**;  #**200** load = **1**;  #**40** load = **0**;  #**40** count\_en = **0**;  #**80** count\_en = **1**;  $display("Count\_en in 1 Clock");  // Đặt các trạng thái ban đầu  #**160** reset\_n = **1**;  #**40** load = **1**;  #**80** load = **0**;  #**40** reset\_n = **0**;  #**40** reset\_n = **1**;    $display("Reset\_n in 1 Clock");  // Đặt các trạng thái ban đầu  #**300** reset\_n = **1**;  #**40** load = **1**;  #**80** load = **0**;  #**40** load = **1**;  #**40** load = **0**;    $display("Load in 1 Clock");    // Dừng mô phỏng  #**500** $stop;  // Dừng mô phỏng sau khi hoàn thành  **end**  // Tạo xung đồng hồ liên tục  **always** #**40** clk = ~clk;  **endmodule** |

**Nhìn vào kết quả mô phỏng ta thấy:**

Bộ đếm hoạt động chính xác, trạng thái done và các tín hiệu count\_en, reset\_n, load đã hoạt động chính xác.=> PASS

# Tổng kết

Do thời gian còn hạn chế và kiến thức chưa đủ, em đã hoàn thành thiết kế lập trình RTL cho bộ đếm 3 bit. Kết quả kiểm thử mô phỏng đáp ứng được các yêu cầu của Specification đã đề ra. Đồng thời, chúng em lần đầu được tiếp cận tới phần mềm Vivado giúp hoàn thành các bước trong quy trình thiết kế VLSI.

Sau quá trình học tập trên lớp và quá trình thực hiện đề tài cùng công ty Dolphin Technology VN, em đã tiếp thu được các yêu cầu đề ra của đề tài này. Không chỉ trong phạm vi của môn học, nhóm chúng em đã tiếp thu được nhiều kiến thức về ngành vi mạch, về quy trình làm việc chuyên nghiệp của công ty.

Lời đầu tiên nhóm xin cảm ơn thầy, TS. Nguyễn Vũ Thắng, người trực tiếp giảng dạy và kết nối chúng em với công ty. Đây là một cơ hội rất hữu ích để chúng em có thể tiếp cận, được học tập và làm việc với doanh nghiệp, thu hoạch được nhiều kinh nghiệm hữu ích cho quá trình làm việc thực tế sau này. Chúng em đặc biệt xin gửi lời cảm ơn tới đội ngũ công ty Dolphin Technology VN đã tạo điều kiện hỗ trợ, hướng dẫn chúng em trong quá trình thực hiện đề tài này.

# Tài liệu tham khảo

[1] <https://drive.google.com/drive/folders/1MArnlrg1IoIehOtbSweoL2YDK4qLxaK3>, VLSI\_BKHN, Dolphin Technology

[2] <https://vlsiverify.com/verilog/verilog-codes/synchronous-fifo>

# PHỤ LỤC

## Code WaveForm

|  |
| --- |
| {  signal: [  { name: 'clk', wave: 'P.......' },  { name: 'reset\_n', wave: '01......' },  { name: 'count\_to', "wave": "3.4.....", data: ['3\'b111', '3\'b010'] },  { name: 'load', wave: '0.10...1' },  { name: 'count\_en', wave: '0.1.....' },  { name: 'count value', wave: '5.443..4', data: ['3\'b000', '3\'b010', '3\'b001', '3\'b000', '3\'b010'] },  { name: 'done', wave: '0...1..0' }  ],  edge: [  '0.5'  ],  config: { hscale: 3 }  } |
| {  signal: [  { name: 'clk', wave: 'P.......' },  { name: 'reset\_n', wave: '01......' },  { name: 'count\_to', "wave": "3.4.....", data: ['3\'b111', '3\'b010'] },  { name: 'load', wave: '0.10...1' },  { name: 'count\_en', wave: '0..1....' },  { name: 'count value', wave: '5.443..4', data: ['3\'b000', '3\'b010', '3\'b001', '3\'b000', '3\'b010'] },  { name: 'done', wave: '0...1..0' }  ],  edge: [  '0.5'  ],  config: { hscale: 3 }  } |

Link trang web: https://wavedrom.com/editor.html