|  |
| --- |
|  |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение  высшего образования  **«МИРЭА - Российский технологический университет»**  **РТУ МИРЭА** |

Институт Информационных Технологий

Кафедра Вычислительной Техники (ВТ)

**ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ № 2**

«Проектирование вычислительных устройств для синтеза сигналов трансцендентных функций»

по дисциплине

«Схемотехника устройств компьютерных систем»

|  |  |
| --- | --- |
| Выполнил студент группы  ИВБО-20-23 | Деревянных В.С. |
| Принял старший преподаватель кафедры ВТ | Дуксин Н.А. |
| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2025 г. |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2025 г. |

Москва 2025

Аннотация

Работа включает в себя 2 рисунка, 10 листингов. Количество страниц в работе — 18.

Содержание

[Введение 4](#_Toc211030610)

[1 Постановка задачи 5](#_Toc211030611)

[2 Проектирование и реализация 6](#_Toc211030612)

[2.1 Методы решения 6](#_Toc211030613)

[2.2 Реализация конвейера без независимого продвижения 6](#_Toc211030614)

[2.3 Реализация конвейера с последовательным независимым продвижением 9](#_Toc211030615)

[2.4 Реализация конвейера с параллельным независимым продвижением 11](#_Toc211030616)

[3 ТЕСТИРОВАНИЕ 14](#_Toc211030617)

[3.1 Тестовый модуль 14](#_Toc211030618)

[Заключение 17](#_Toc211030619)

[Список использованных источников 18](#_Toc211030620)

Введение

Одной из фундаментальных задач схемотехники устройств компьютерных систем является повышение производительности вычислительных блоков. Классический подход к построению комбинационных схем, при котором результат вычисления появляется на выходе после установления сигналов на всех элементах цепи (после прохождения критического пути), обладает принципиальным ограничением. Тактовая частота синхронного устройства в таком случае должна быть обратно пропорциональна времени задержки самой длинной последовательности операций. Это ограничивает максимальное быстродействие всего вычислительного тракта.

Для преодоления указанного ограничения широко применяется метод конвейерной обработки (pipelining), заимствованный из технологии промышленного производства. Основная идея конвейеризации применительно к схемотехнике заключается в разбиении сложного многостадийного вычислительного процесса на последовательность более простых, независимых этапов (ступеней, стадий). Каждая стадия выполняет строго определенную часть общей задачи и имеет собственную триггерную регистровую память на выходе.

Процесс вычисления организуется таким образом, что как только первая стадия завершает обработку текущего набора данных (операндов), она передает промежуточный результат на свою выходную регистровую защелку (так называемый конвейерный регистр или регистр-защелка), после чего немедленно приступает к обработке следующего набора данных. В то же время вторая стадия начинает обработку результата, полученного от первой стадии на предыдущем такте. Таким образом, в течение одного тактового импульса одновременно работают все стадии конвейера, каждая над своим набором данных.

1. Постановка задачи

В ходе выполнения практической работы необходимо выполнить следующие задачи:

* Изучить различные методы конвейеризации;
* Реализовать параметризированный конвейер с valid-chain;
* Реализовать конвейер с последовательной логикой сдвига конвейера;
* Реализовать конвейер с параллельной логикой сдвига конвейера;
* Провести верификацию правильности работы всех методов;
* Составить отчёт

1. Проектирование и реализация

## Методы решения

Конвейеризация процессов достигается путём разделения общей логики на несколько участков с одинаковой задержкой и установки между ними синхронных регистров, которые сохраняют промежуточные результаты вычислений.

Поскольку конвейер работает непрерывно, параллельно с основным потоком данных формируется дополнительный конвейер, передающий сигналы, подтверждающие корректность (валидность) данных на каждом этапе.

На практике данные поступают в конвейер не постоянно, а с паузами. Чтобы повысить эффективность работы, при остановке выходных стадий целесообразно позволять продолжать движение тем этапам, которые не заблокированы следующими стадиями и перед которыми присутствует так называемый «пузырь» — пустой слот без данных.

Выделяют два основных подхода к формированию разрешающего сигнала, управляющего продвижением конвейера:

1. Последовательный способ. В этом случае логика формирования разрешающего сигнала shift\_rdy для каждой стадии строится последовательно и зависит от сигналов shift\_rdy следующих стадий.
2. Параллельный способ. В данном варианте управляющая логика для всех стадий формируется одновременно — каждый сигнал shift\_rdy рассчитывается с учётом состояния всех стадий конвейера.

## Реализация конвейера без независимого продвижения

Реализация конвейера с valid-chain.

Листинг 2.2.1 — pipeline.v

|  |
| --- |
| `timescale 1ns / 1ps  module pipeline#(  parameter DATA\_SIZE = 8,  parameter STAGE\_COUNT = 16  )(  input clk, reset, fifo\_out\_ready, valid\_in,  input [STAGE\_COUNT - 1: 0] valid\_stages\_in,  input [DATA\_SIZE - 1: 0] data\_in,  output [DATA\_SIZE - 1: 0] data\_out  );  reg [DATA\_SIZE - 1: 0] pipe\_reg [0: STAGE\_COUNT - 1];  always@(posedge clk)  begin  if (reset)  begin  pipe\_reg[0] <= 0;  end  else if (fifo\_out\_ready && valid\_in)  begin  pipe\_reg[0] <= data\_in;  end  end  assign data\_out = pipe\_reg[STAGE\_COUNT - 1];  genvar i;  generate  for (i = 1; i < STAGE\_COUNT; i = i + 1)  begin : pipeline\_increment  always@(posedge clk)  begin  if (reset)  begin  pipe\_reg[i] <= 0;  end  else if (fifo\_out\_ready && valid\_stages\_in[i - 1])  begin  pipe\_reg[i] <= pipe\_reg[i - 1] + 1;  end  end  end  endgenerate  endmodule |

Листинг 2.2.2 — valid\_chain.v

|  |
| --- |
| `timescale 1ns / 1ps  module valid\_chain#(  parameter STAGE\_COUNT = 16  )(  input clk, reset, valid\_in, fifo\_out\_ready,  output reg [STAGE\_COUNT - 1: 0] valid\_out  );  always@(posedge clk)  begin  if (reset)  begin  valid\_out[0] <= 0;  end  else if (fifo\_out\_ready)  begin  valid\_out[0] <= valid\_in;  end  end  genvar i;  generate  for (i = 1; i < STAGE\_COUNT; i = i + 1)  begin : valid\_chain  always@(posedge clk)  begin  if (reset)  begin  valid\_out[i] <= 0;  end  else if (fifo\_out\_ready)  begin  valid\_out[i] <= valid\_out[i - 1];  end  end  end  endgenerate    endmodule |

Листинг 2.2.3 — top.v

|  |
| --- |
| `timescale 1ns / 1ps  module top#(  parameter DATA\_SIZE = 8,  parameter STAGE\_COUNT = 16  )(  input clk, reset, fifo\_out\_ready, valid\_in,  input [DATA\_SIZE - 1: 0] data\_in,  output [DATA\_SIZE - 1: 0] data\_out,  output valid\_out  );  wire [STAGE\_COUNT - 1: 0] valid\_stages;  pipeline #(  .DATA\_SIZE(DATA\_SIZE),  .STAGE\_COUNT(STAGE\_COUNT)  ) pipeline\_unit (  .clk(clk),  .reset(reset),  .fifo\_out\_ready(fifo\_out\_ready),  .data\_in(data\_in),  .data\_out(data\_out),  .valid\_in(valid\_in),  .valid\_stages\_in(valid\_stages)  );  valid\_chain #(  .STAGE\_COUNT(STAGE\_COUNT)  ) valid\_chain\_unit (  .clk(clk),  .reset(reset),  .valid\_in(valid\_in),  .fifo\_out\_ready(fifo\_out\_ready),  .valid\_out(valid\_stages)  );  assign valid\_out = valid\_stages[STAGE\_COUNT - 1];  endmodule |

## Реализация конвейера с последовательным независимым продвижением

В данном варианте реализации каждая стадия конвейера получает разрешение на продвижение данных последовательно — от выходной к входной. Продвижение происходит только при условии готовности всех последующих стадий.

Листинг 2.3.1 — pipeline.v

|  |
| --- |
| `timescale 1ns / 1ps  module pipeline#(  parameter DATA\_SIZE = 8,  parameter STAGE\_COUNT = 16  )(  input clk, reset, fifo\_out\_ready, valid\_in,  input [STAGE\_COUNT - 1: 0] valid\_stages\_in,  input [DATA\_SIZE - 1: 0] data\_in,  output [DATA\_SIZE - 1: 0] data\_out  );  reg [DATA\_SIZE - 1: 0] pipe\_reg [0: STAGE\_COUNT - 1];  always@(posedge clk)  begin  if (reset)  begin  pipe\_reg[0] <= 0;  end  else if (valid\_in && valid\_stages\_in[0])  begin  pipe\_reg[0] <= data\_in;  end  end  assign data\_out = pipe\_reg[STAGE\_COUNT - 1];  genvar i;  generate  for (i = 1; i < STAGE\_COUNT; i = i + 1)  begin : pipeline\_increment  always@(posedge clk)  begin  if (reset)  begin  pipe\_reg[i] <= 0;  end  else if (valid\_stages\_in[i - 1])  begin  pipe\_reg[i] <= pipe\_reg[i - 1] + 1;  end  end  end  endgenerate  endmodule |

Листинг 2.3.2 — valid\_chain.v

|  |
| --- |
| `timescale 1ns / 1ps  module valid\_chain#(  parameter STAGE\_COUNT = 16  )(  input clk, reset, valid\_in, fifo\_out\_ready,  output [STAGE\_COUNT - 1: 0] back\_tracking  );  reg [STAGE\_COUNT - 1: 0] valid\_out;  always@(posedge clk)  begin  if (reset)  begin  valid\_out[0] <= 0;  end  else if (back\_tracking[0])  begin  valid\_out[0] <= valid\_in;  end  end  assign back\_tracking[STAGE\_COUNT - 1] = fifo\_out\_ready || ~valid\_out[STAGE\_COUNT - 1];  genvar i;  generate  for (i = 1; i < STAGE\_COUNT; i = i + 1)  begin : valid\_chain  always@(posedge clk)  begin  if (reset)  begin  valid\_out[i] <= 0;  end  else if (back\_tracking[i])  begin  valid\_out[i] <= valid\_out[i - 1];  end  end  end    for (i = 0; i < STAGE\_COUNT - 1; i = i + 1)  begin  assign back\_tracking[i] = (back\_tracking[i + 1] && valid\_out[i]) || ~valid\_out[i];  end  endgenerate    endmodule |

Листинг 2.3.3 — top.v

|  |
| --- |
| `timescale 1ns / 1ps  module top#(  parameter DATA\_SIZE = 8,  parameter STAGE\_COUNT = 16  )(  input clk, reset, fifo\_out\_ready, valid\_in,  input [DATA\_SIZE - 1: 0] data\_in,  output [DATA\_SIZE - 1: 0] data\_out,  output valid\_out  );  wire [STAGE\_COUNT - 1: 0] valid\_stages;  pipeline #(  .DATA\_SIZE(DATA\_SIZE),  .STAGE\_COUNT(STAGE\_COUNT)  ) pipeline\_unit (  .clk(clk),  .reset(reset),  .fifo\_out\_ready(fifo\_out\_ready),  .data\_in(data\_in),  .data\_out(data\_out),  .valid\_in(valid\_in),  .valid\_stages\_in(valid\_stages)  );  valid\_chain #(  .STAGE\_COUNT(STAGE\_COUNT)  ) valid\_chain\_unit (  .clk(clk),  .reset(reset),  .valid\_in(valid\_in),  .fifo\_out\_ready(fifo\_out\_ready),  .back\_tracking(valid\_stages)  );  assign valid\_out = valid\_stages[STAGE\_COUNT - 1];  endmodule |

## Реализация конвейера с параллельным независимым продвижением

Параллельная логика продвижения данных устраняет зависимость между стадиями: разрешающие сигналы формируются одновременно для всех ступеней конвейера. При этом возрастает сложность управляющей логики.

Листинг 2.4.1 — pipeline.v

|  |
| --- |
| module pipeline #(  DATA\_SIZE = 8,  STAGE\_COUNT = 16  )(  input clk,  input reset,  input fifo\_out\_rdy,  input valid\_in,  input [STAGE\_COUNT-1:0] shift\_rdy,  input [DATA\_SIZE-1:0] data\_in,  output [DATA\_SIZE-1:0] data\_out  );  reg [DATA\_SIZE-1:0] pipe\_reg [0:STAGE\_COUNT-1];  always @(posedge clk)  if (reset)  pipe\_reg[0] <= 0;  else if (valid\_in)  pipe\_reg[0] <= data\_in;  assign data\_out = pipe\_reg[STAGE\_COUNT-1];  genvar i;  generate  for (i = 1; i < STAGE\_COUNT; i = i + 1)  begin : pipeline\_increment  always @(posedge clk)  if (reset)  pipe\_reg[i] <= 0;  else if (shift\_rdy[i])  pipe\_reg[i] <= pipe\_reg[i-1] + 1;  else  pipe\_reg[i] <= pipe\_reg[i];  end  endgenerate  endmodule |

Листинг 2.4.2 — top.v

|  |
| --- |
| module Top #(  parameter DATA\_SIZE = 8,  parameter STAGE\_COUNT = 64  )(  input clk,  input reset,  input fifo\_out\_rdy,  input valid\_in,  input [DATA\_SIZE-1:0] data\_in,  output [DATA\_SIZE-1:0] data\_out,  output valid\_out  );  wire [STAGE\_COUNT-1:0] valid\_stages;  wire [STAGE\_COUNT-1:1] shift\_rdy;  pipeline #(  .DATA\_SIZE(DATA\_SIZE),  .STAGE\_COUNT(STAGE\_COUNT)  ) pipeline\_unit (  .clk(clk),  .reset(reset),  .fifo\_out\_rdy(fifo\_out\_rdy),  .valid\_in(valid\_in),  .shift\_rdy(shift\_rdy),  .data\_in(data\_in),  .data\_out(data\_out)  );  valid\_chain\_parallel #(  .STAGE\_COUNT(STAGE\_COUNT)  ) valid\_chain\_unit (  .clk(clk),  .reset(reset),  .valid\_in(valid\_in),  .fifo\_out\_rdy(fifo\_out\_rdy),  .valid\_out(valid\_stages),  .shift\_rdy(shift\_rdy)  );  assign valid\_out = valid\_stages[STAGE\_COUNT-1];  endmodule |

Листинг 2.4.3 — valid\_chain\_ parallel.sv

|  |
| --- |
| module valid\_chain\_parallel #(  STAGE\_COUNT = 16  )(  input logic clk,  input logic reset,  input logic valid\_in,  input logic fifo\_out\_rdy,  output logic [STAGE\_COUNT-1:0] valid\_out,  output wire [STAGE\_COUNT-1:0] shift\_rdy  );  assign shift\_rdy[0] = ~(&valid\_out[STAGE\_COUNT-1:0]);  always\_ff @(posedge clk) begin : valid\_chain\_first  if (reset)  valid\_out[0] <= 0;  else if (fifo\_out\_rdy | shift\_rdy[1])  valid\_out[0] <= valid\_in;  else  valid\_out[0] <= valid\_out[0];  end : valid\_chain\_first  genvar i;  for (i = 1; i < STAGE\_COUNT-1; i++) begin : valid\_chain\_middle  assign shift\_rdy[i] = ~(&valid\_out[STAGE\_COUNT-1:i]);  always\_ff @(posedge clk)  if (reset)  valid\_out[i] <= 0;  else if (fifo\_out\_rdy | shift\_rdy[i])  valid\_out[i] <= valid\_out[i-1];  else  valid\_out[i] <= valid\_out[i];  end : valid\_chain\_middle  assign shift\_rdy[STAGE\_COUNT-1] = ~valid\_out[STAGE\_COUNT-1];  always\_ff @(posedge clk)  if (reset)  valid\_out[STAGE\_COUNT-1] <= 0;  else if (fifo\_out\_rdy | ~valid\_out[STAGE\_COUNT-1])  valid\_out[STAGE\_COUNT-1] <= valid\_out[STAGE\_COUNT-2];  else  valid\_out[STAGE\_COUNT-1] <= valid\_out[STAGE\_COUNT-1];  endmodule |

1. ТЕСТИРОВАНИЕ

## Тестовый модуль

Для верификации методов создадим тестовый модуль. Он представлен в листинге 3.1.

Листинг 3.1 — test.v

|  |
| --- |
| `timescale 1ns / 1ps  module test;  localparam DATA\_SIZE = 8;  localparam STAGE\_COUNT = 16;  reg clk;  always #10 clk <= ~clk;  reg reset, fifo\_out\_ready, valid\_in;  reg [DATA\_SIZE-1:0] data\_in;  initial  begin  clk = 0;  data\_in = 5;  reset = 1;  fifo\_out\_ready = 0;  valid\_in = 0;  @(posedge clk);  @(posedge clk);  reset = 0;  fifo\_out\_ready = 1;  valid\_in = 1;  repeat (6)  @(posedge clk);  fifo\_out\_ready = 0;  @(posedge clk);  @(posedge clk);  valid\_in = 0;  data\_in = 0;  repeat (3)  @(posedge clk);  valid\_in = 1;  data\_in = 10;  repeat (3)  @(posedge clk);  valid\_in = 0;  data\_in = 0;  repeat (12)  @(posedge clk);  fifo\_out\_ready = 1;  repeat (12)  @(posedge clk);  $finish;  end  top #(  .DATA\_SIZE(DATA\_SIZE),  .STAGE\_COUNT(STAGE\_COUNT)  ) top\_unit (  .clk(clk),  .reset(reset),  .fifo\_out\_ready(fifo\_out\_ready),  .valid\_in(valid\_in),  .data\_in(data\_in),  .data\_out(data\_out),  .valid\_out(valid\_out)  );  endmodule |

Результаты тестирования представлены на рисунке 3.1 - 3.2.

Изображение выглядит как снимок экрана, Красочность, пространство, линия

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 3.1 — Регистры данных конвейера

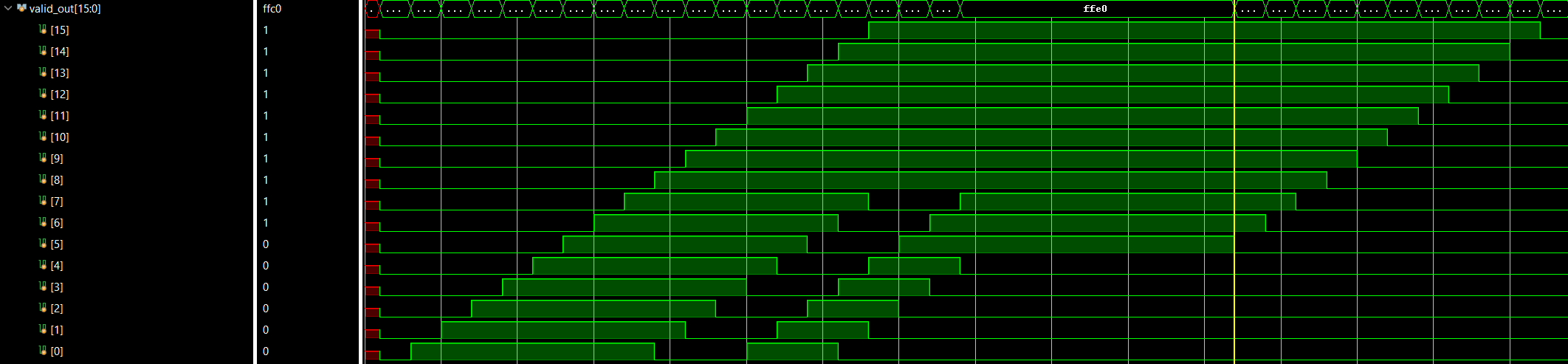


Рисунок 3.2 — Регистры валидности конвейера (с последовательным независимым продвижением)

Сравним показатели WNS (нс) у трёх реализаций при разных длинах конвейера, результаты приведены в таблице 3.1.

Таблица 3.1 – Сравнение показателей WNS, нс

|  |  |  |  |
| --- | --- | --- | --- |
| Вид реализации\Число стадий конвейера | 16 | 32 | 64 |
| Без независимого продвижения | 7.252 | 7.284 | 6.95 |
| Последовательное независимое продвижение | 5.672 | 4.345 | 0.863 |
| Параллельное независимое продвижение | 6.379 | 5.835 | 5.696 |

На основании полученных результатов можно сделать следующие выводы. При реализации без механизма независимого продвижения значение показателя WNS остаётся неизменным, так как добавляемые стадии не содержат сквозной логики, и этот показатель определяется только внутренней логикой каждой отдельной стадии. Параллельная реализация показывает преимущество по сравнению с последовательной благодаря независимости логики — отсутствие строгих взаимосвязей между стадиями позволяет эффективнее применять стратегии размещения и, как следствие, повышает общую производительность системы.

Заключение

В рамках практической работы были разработаны и проверены на практике различные способы реализации конвейера на языке Verilog. Проведён сравнительный анализ трёх методов — без независимого продвижения, с последовательным и параллельным независимым продвижением. По результатам экспериментов установлено, что наилучшие показатели производительности демонстрирует параллельная реализация.

Список использованных источников

1. Xilinx (AMD Official Website) [Электронный ресурс]. - URL: <https://www.xilinx.com/>.

2. Угрюмов Е. П. Цифровая схемотехника: учеб. пособие для вузов. —3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2010. — 816 с.: ил.

3. Тарасов И. Е. ПЛИС Xilinx/AMD. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования. - М.: Горячая линия - Телеком, 2022. - 538 с.: ил.

4. Схемотехника устройств компьютерных систем. Ч. 1 [Электронный ресурс]: лабораторный практикум // Н. А. Дуксин, Д. В. Люлява, И. Е. Тарасов. — М.: РТУ МИРЭА, 2023. (дата обращения: 23.02.2025).

5. IEEE Standard Verilog® Hardware Description Language (IEEE Std 1364-2001).

6. Соловьев В. В. Основы языка проектирования цифровой аппаратуры Verilog. — М.: Горячая линия — Телеком, 2014. — 208 с.

7. Дуксин Н.А., Люлява Д.В., Долидзе И.И. Вопросы проектирования специализированных вычислителей конвейерного типа. Сборник трудов XII Международной научной конференции «ИТ – СТАНДАРТ 2023», с. 181-193.