|  |
| --- |
|  |
| 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 №** **3**

«Процессорное ядро последовательного типа»

по дисциплине

«Схемотехника устройств компьютерных систем»

|  |  |
| --- | --- |
| Выполнил студент группы  ИВБО-01-22 | Воробьев Д.М. |
| Принял ассистент кафедры ВТ | Дуксин Н.А. |
| Практическая работа выполнена | «\_\_»\_\_\_\_\_\_\_2024 г. |
| «Зачтено» | «\_\_»\_\_\_\_\_\_\_2024 г. |

Москва 2024

АННОТАЦИЯ

Данная работа включает в себя 6 рисунок, 4 листинга, 3 таблицы. Количество страниц в работе — 29.

СОДЕРЖАНИЕ

[1 ОСНОВНОЙ РАЗДЕЛ 5](#_Toc180869587)

[1.1 Описание архитектуры процессора 5](#_Toc180869588)

[1.2 Описание формата команд 6](#_Toc180869589)

[1.3 Блок схема алгоритма 9](#_Toc180869590)

[1.4 Описание стадий команд 12](#_Toc180869591)

[1.5 Код модулей на Verilog 15](#_Toc180869592)

[1.6 Тестирование работы процессора 25](#_Toc180869593)

ВВЕДЕНИЕ

Цель работы: целью данной практической работы является создание процессорного ядра и разработка всех необходимых модулей и кодов, обеспечивающих выполнение базовых операций, обработку команд и взаимодействие с памятью.

Постановка задачи: определить архитектуру процессорного ядра, включая структуру регистров, арифметико-логического устройства (АЛУ), системы управления и блоков памяти. Создать и задокументировать набор команд для процессорного ядра, включая их форматы, семантику и функциональность. Разработать программные модули для реализации функционала процессора

Результат выполнения работы: код модулей на Verilog HDL, временные диаграммы, отражающие корректность работы спроектированных модулей.

1 ОСНОВНОЙ РАЗДЕЛ

* 1. Описание архитектуры процессора

Опишем формат команд по заданному варианту: сортировка массива пузырьком.

Сортировка пузырьком — это простой алгоритм сортировки, который многократно проходит по массиву, сравнивая соседние элементы и меняя их местами, если они находятся в неправильном порядке. Процесс повторяется, пока массив не будет отсортирован.

Память данных (MEM) для хранения массива, который будет отсортирован. Размер: N = 10 (10 элементов). Каждый элемент массива будет занимать 32 бита.

Память команд (CM) для хранения команд, которые будут выполняться процессором. Максимальный размер одной команды 39 бит. Потенциально общее количество команд – максимум 64 (6 бит).

Регистры общего назначения (GPR) включают в себя:

* Регистр 0 (константа 0);
* Регистр 1 (константа 1);
* Регистр 2 (количество элементов);
* Регистр 3 (индекс i в цикле сортировки);
* Регистр 4 (хранит результат вычитания в операции SUB, используется для проверки условия продолжения цикла);
* Регистр 5 (временное хранилище для обмена элементов при выполнении SWP);
* Регистр 6 (индекс j в цикле сортировки);
* Регистр 7 (элемент j в цикле сортировки);
* Регистр 8 (индекс j + 1 в цикле сортировки).
  1. Описание формата команд

Каждая команда состоит из нескольких полей:

* Код операции (КОП) – определяет какую операцию выполнить (4 бита, позволяет иметь до 16 команд);
* Адреса регистров или памяти – указывают на места, откуда нужно брать данные или куда записывать результат;
* Литеральные значения – задают значения, которые используются в операциях.

Максимальное количество бит для команды 39 бит.

Адресация регистров осуществляется с помощью 3 бит, что позволяет адресовать до 8 регистров (адрес 1 + адрес 2 + адрес результата).

Для команд, которые используют переходы (например, JLZ и JMP), используется 6 бит, что позволяет указать до 64 различных адресов в памяти команд.

* NOP (No Operation):
  + Код операции (КОП) 4 бита;
  + Общий формат: 000000000000000000000000000000000000000;
* INCR (Увеличение значения регистра на 1):
  + Код операции (КОП) 4 бита;
  + Адрес регистра для инкрементации: 3 бита;
  + Общий формат: 0001 101 00000000000000000000000000000000 (увеличение регистра 3)
* LTR (Загрузка литерала в регистр):
  + Код операции (КОП) 4 бита;
  + Адрес регистра, в который писать: 3 бита;
  + Значение литерала: 32 бита (результат);
  + Общий формат: 0010 011 00000000 00000000 00000000 00000001 (записать в регистр 3 литерал 1).
* NTR (Загрузка 0 в регистр):
  + Код операции (КОП) 4 бита;
  + Адрес регистра, в который писать: 3 бита;
  + Общий формат: 0011 011 00000000000000000000000000000000 (обнуляет регистр 3).
* LTM (Загрузка литерала в память):
  + Код операции (КОП) 4 бита;
  + Номер регистра, в котором хранится адрес ячейки: 3 бита (косвенная адресация);
  + Значение литерала: 32 бита;
  + Общий формат: 0010 011 00000000 00000000 00000000 00001010 (записать значение из регистра 3 в ячейку памяти с адресом 10).
* SUB (Вычитание значений из двух регистров):
  + Код операции (КОП) 4 бита;
  + Адрес 1 регистра: 3 бита (прямая адресация);
  + Адрес 2 регистра: 3 бита (прямая адресация);
  + Адрес регистра, куда записывать результат: 3 бита;
  + Общий формат: 0101 011 010 100 00000000000000000000000000 (вычесть значение регистра 2 из регистра 3 и записать в регистр 4).
* JLZ (Условный переход, если результат меньше нуля):
  + Код операции (КОП) 4 бита;
  + Адрес ячейки памяти команд (куда переходить): 6 бит;
  + Общий формат: 0110 010010 00000000000000000000000000000 (если результат в регистре оказался меньше нуля – перейти по адресу 18).
* PTR (Косвенная передача данных, из памяти в регистры):
  + Код операции (КОП) 4 бита;
  + Адрес в памяти данных (из какого адреса извлекаем, косвенная адресация): 3 бита;
  + Адрес регистра РОН (куда переносить, работаем со значением j и j + 1, помещено будет в регистр k и k + 1): 3 бита;
  + Общий формат: 0111 101 110 0000000000000000000000000000000 (считать значения из памяти по адресам, заданные в регистрах 5 и 6 и поместить их в регистры 7 и 8).

Команда PTR использует два порта для доступа к памяти данных: один порт для чтения значения по адресу «j» и второй порт для чтения значения по адресу «j + 1».

В процессе выполнения операции можно записать одно значение в определенный регистр, а затем в другом такте записать следующее значение в другой регистр или ячейку памяти.

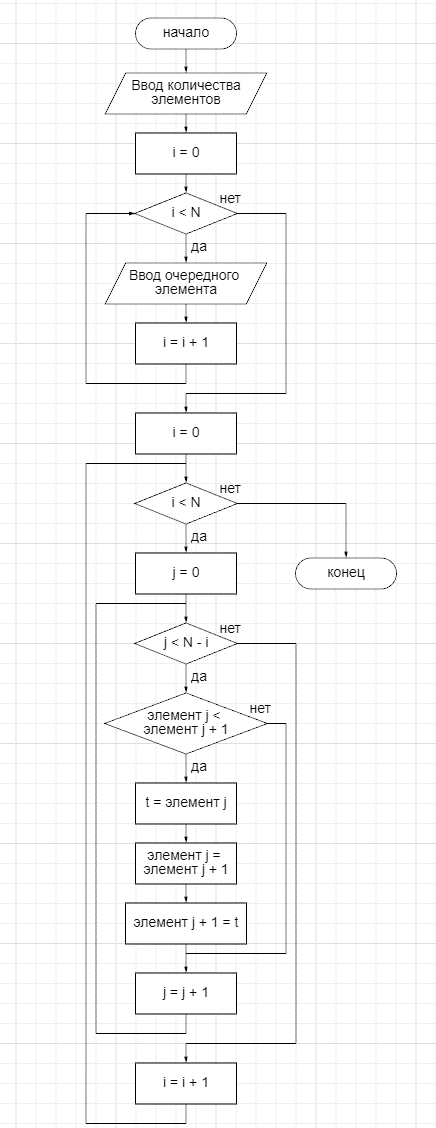
* SWP (Обмен значений между регистрами):
  + Код операции (КОП) 4 бита;
  + Адрес в памяти данных (косвенная адресация, значение t и t + 1): 3 бита;
  + Общий формат: 1000 101 00000000000000000000000000000000 (поменять местами значения, хранящиеся в памяти по адресу, указанному в регистре 5);
* JMP (Безусловный переход):
  + Код операции (КОП) 4 бита;
  + Адрес в памяти команд: 6 бит;
  + Общий формат: 1001 000111 00000000000000000000000000000 (переход по адресу 7).

Все команды отработают за 3 такта, за исключением LTM (4 такта) и PTR, SWP (5 тактов).

* 1. Блок схема алгоритма

Алгоритм начинается с ввода количества элементов массива. Затем происходит инициализация счетчика i = 0 и ввод элементов массива в цикле.

После ввода всех элементов начинается основная часть сортировки: внешний цикл с счетчиком i, внутренний цикл с счетчиком j, сравнение соседних элементов массива (элемент j и элемент j + 1). Если текущий элемент больше следующего, происходит их обмен с помощью временной переменной t (Рисунок 1.1).

Рисунок 1.1 – Блок схема алгоритма сортировки пузырьком

Далее опишем покомандный алгоритм на основе наших команд, описанных выше.

На шаге 0 команда LIR 2, N загружает в регистр 2 длину массива (N). На шаге 1 команда NTR 3 обнуляет регистр 3, который будет использоваться как индекс) для текущей позиции в массиве.

Шаг 2 выполняет безусловный переход к шагу 20 с помощью команды JMP 20, который инициализирует сортировку.

На шаге 3 команда LTR 3 загружает текущий индекс j в регистр. Далее на шаге 4 выполняется операция сравнения: команда SUB 3, 2, 4 вычитает из текущего индекса длину массива. Это нужно для того, чтобы определить, достигнут ли конец массива.

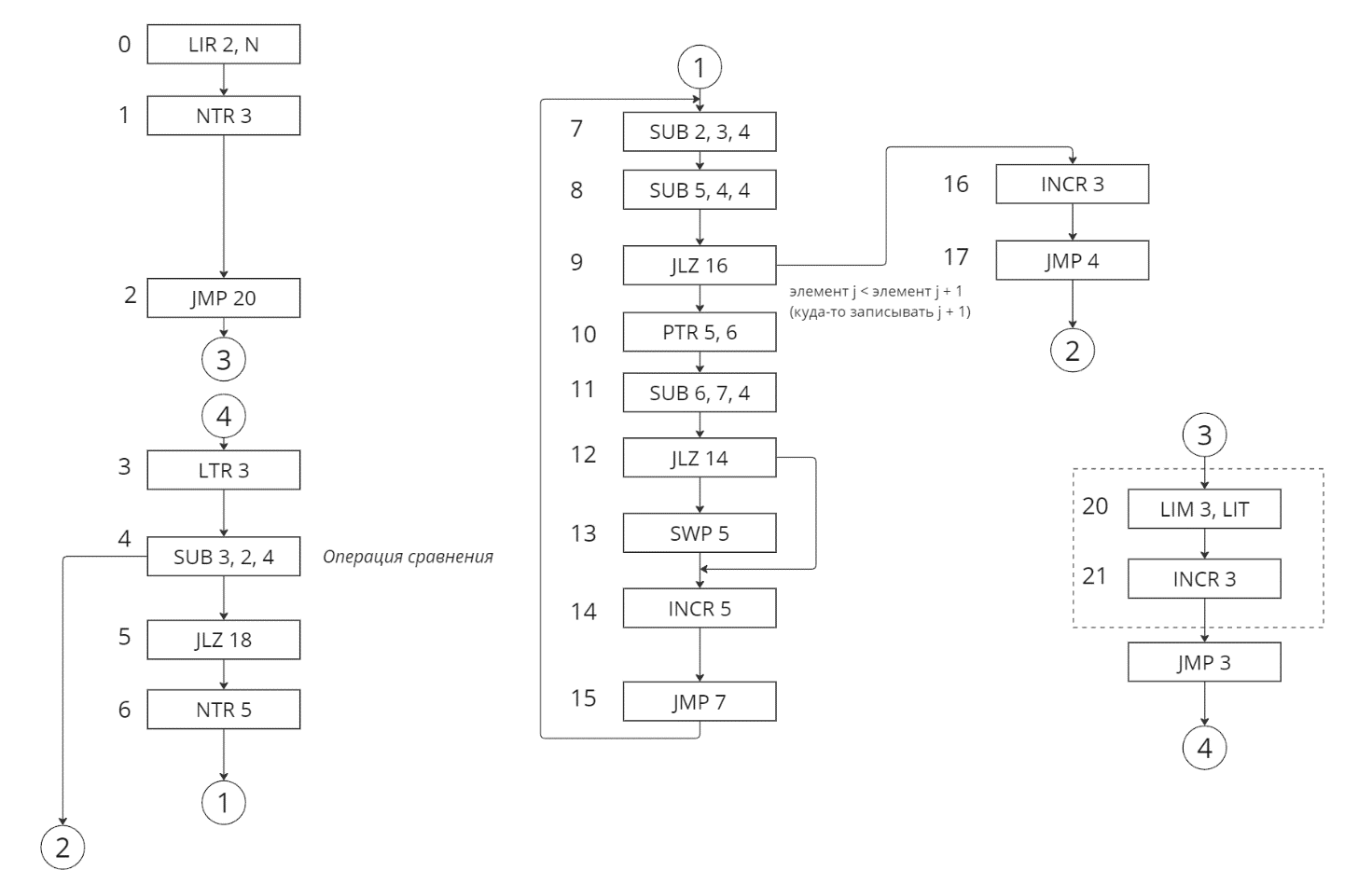
Если результат отрицательный, команда JLZ 18 на шаге 5 переходит к завершению работы сортировки. Если нет, продолжается выполнение.

Внутри цикла выполняется операция сравнения соседних элементов (шаги 7-13). Если элемент в позиции j больше элемента в позиции j+1, они меняются местами с помощью команды SWP 5 на шаге 13.

После каждой итерации индекс (j) увеличивается на 1 с помощью команды INCR 3 на шаге 14.

После инкремента происходит безусловный переход на шаг 4 для повторного выполнения цикла.

Когда индекс (j) достигает значения (N - 1), программа переходит к завершению на шаге 18 и завершает сортировку (Рисунок 1.2).

Рисунок 1.2 – Блок схема алгоритма сортировки пузырьком в процессорных командах

1.4 Описание стадий команд

Опишем выполнение каждой команды на различных стадиях выполнения (fetch, decode, execute, mem access, write back). Каждая команда проходит через 5 стадий — извлечение, декодирование, исполнение, доступ к памяти и запись результата (Таблица 1.1).

Таблица 1.1 – Описание стадий команд в форме текста

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Команда | 0 стадия fetch | 1 стадия decode | 2 стадия исполнение (execute) | 3 стадия доступ к памяти (mem access) | 4 стадия запись результата (write back) |
| NOP | Извлечение команды из памяти. | Команда декодируется как NOP. | - | - | - |
| INCR | Извлечение команды из памяти. | Декодирование команды, получение адреса регистра. | Операция инкремента: значение регистра увеличивается на 1. | - | Результат записывается в регистр. |
| LTR | Извлечение команды из памяти. | Декодирование команды, получение константы и адреса регистра. | Загрузка константы в регистр. | - | Результат записывается в регистр. |
| NTR | Извлечение команды из памяти. | Декодирование команды. | Запись нуля в регистр. | - | Результат записывается в регистр. |
| LTM | Извлечение команды из памяти. | Декодирование команды, получение адреса памяти и регистра. | - | Загрузка значения из памяти по указанному адресу. | Результат записывается в регистр. |
| SUB | Извлечение команды из памяти. | Декодирование команды, получение операндов и адресов регистров. | Вычитание значений двух регистров. | - | Результат записывается в регистр. |
| JLZ | Извлечение команды из памяти. | Декодирование команды, проверка результата предыдущей операции. | Переход, если результат предыдущей операции отрицательный. | - | - |
| PTR | Извлечение команды из памяти. | Декодирование команды, получение адреса памяти и регистра. | Взятие значения по указателю (косвенный доступ к памяти). | Доступ к памяти по косвенному адресу. | Результат записывается в регистр. |
| SWP | Извлечение команды из памяти. | Декодирование команды, получение адресов регистров. | Операция обмена значениями между регистрами. | - | Запись результатов обмена в регистры. |
| JMP | Извлечение команды из памяти. | Декодирование команды, получение адреса перехода. | Переход на указанный адрес. | - | - |

Далее опишем данные таблице в форме команд (Таблица 1.2).

Таблица 1.2 – Описание стадий команд в форме команд

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Команда | 0: Fetch | 1: Decode | 2: Execute | 3: Memory Access | 4: Write Back |
| INCR | PR = CMD[PC] | alu1 = GPR[adr1] alu2 = 1 | res = alu1 + alu2 | - | GPR[adr1] = res pc = pc + 1 |
| LTR | PR = CMD[PC] | - | - | - | GPR[adr1] = lit pc = pc + 1 |
| NTR | PR = CMD[PC] | - | res = 0 | - | GPR[adr1] = res pc = pc + 1 |
| LTM | PR = CMD[PC] | - | - | - | MEM[GPR[adr1]] = lit pc = pc + 1 |
| SUB | PR = CMD[PC] | alu1 = GPR[adr1] alu2 = GPR[adr2] | res = alu1 - alu2 lz = res[31] | - | GPR[adr1] = res pc = pc + 1 |
| JLZ | PR = CMD[PC] | - | if (lz) pc = ja; else pc = pc + 1; | - | - |
| PTR | PR = CMD[PC] | alu1 = GPR[adr1] alu2 = GPR[adr2] | res\_h = alu1 + 1 res\_l = alu2 + 1 | - | GPR[adr1] = MEM[alu1 + 1] pc = pc + 1 |
| SWP | PR = CMD[PC] | alu1 = GPR[adr1] alu2 = GPR[adr2] | - | - | GPR[adr1] = GPR[adr2] GPR[adr2] = alu1 |
| JMP | PR = CMD[PC] | - | - | - | pc = ja |

Далее опишем таблицу на языке Verilog (Таблица 1.3).

Таблица 1.3 – Описание стадий команд в форме команд Verilog

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Команда | 0: Fetch | 1: Decode | 2: Execute | 3: Memory Access | 4: Write Back |
| INCR | pr\_next <= cmd[pc] | AdrA <= addr1\_next,  AdrB <= 1 | res\_next <= alu1 + alu2 | - | DATA <= res[31:0],  AdrWrite <= addr1,  pc <= pc + 1 |
| LTR | pr\_next <= cmd[pc] | - | - | - | DATA <= lit,  AdrWrite <= addr1,  pc <= pc + 1 |
| NTR | pr\_next <= cmd[pc] | - | - | - | DATA <= 0, wen <= 1,  AdrWrite <= addr1,  pc <= pc + 1 |
| LTM | pr\_next <= cmd[pc] | AdrA <= addr1\_next | - | - | mem[GPR[adr1]] <= lit,  pc <= pc + 1 |
| SUB | pr\_next <= cmd[pc] | AdrA <= addr1\_next,  AdrB <= addr2\_next | res\_next <= alu1 - alu2,  lz\_next <= res[31] | - | DATA <= res[31:0],  AdrWrite <= addr1,  pc <= pc + 1 |
| JLZ | pr\_next <= cmd[pc] | - | if (~lz) pc <= ja else pc <= pc + 1 | - | - |
| PTR | pr\_next <= cmd[pc] | AdrA <= addr1\_next,  AdrB <= addr2\_next | res\_next <= {alu1 + 1, alu2 + 1} | DATA <= mem[alu1] | GPR[res[31:0]] <= mem[res[63:32]],  pc <= pc + 1 |
| SWP | pr\_next <= cmd[pc] | - | res\_next <= alu2 + 1 | - | mem[alu2] <= mem[res],  mem[res] <= mem[alu2], |
| JMP | pr\_next <= cmd[pc] | - | - | - | pc <= ja |

1.5 Код модулей на Verilog

Опишем код модуля регистрового файла. Он функционирует как набор регистров, которые могут быть прочитаны или записаны в зависимости от входных сигналов. Используется для хранения временных данных.

DATA\_WIDTH – ширина данных каждого регистра (по умолчанию 32 бита). REG\_FILE\_SIZE – количество регистров в регистровом файле (по умолчанию 8 регистров). ADDR\_WIDTH – ширина адреса, необходимая для адресации всех регистров.

DATA – входные данные для записи в регистр. AdrWrite – адрес регистра для записи данных. AdrA, AdrB – адреса регистров для операций чтения. OperandA, OperandB – выходные данные из регистров, адресуемых AdrA и AdrB.

Инициализирует все регистры (GPR) нулевыми значениями при старте. Первый регистр становится константой единицы (GPR[1]). Если сигнал wen активен и AdrWrite указывает на допустимый регистр (не нулевой и не первый), данные (DATA) записываются в регистр GPR[AdrWrite] (Листинг 1.1).

Листинг 1.1 – Модуль регистрового файла на языке Verilog

`timescale 1ns / 1ps

module reg\_file#(

DATA\_WIDTH = 32,

REG\_FILE\_SIZE = 8,

ADDR\_WIDTH = $clog2(REG\_FILE\_SIZE)

)(

input clk, reset, wen,

input [DATA\_WIDTH - 1 : 0] DATA,

input [ADDR\_WIDTH - 1 : 0] AdrWrite,

input [ADDR\_WIDTH - 1 : 0] AdrA,

input [ADDR\_WIDTH - 1 : 0] AdrB,

output reg [DATA\_WIDTH - 1 : 0] OperandA,

output reg [DATA\_WIDTH - 1 : 0] OperandB

);

reg [DATA\_WIDTH - 1 : 0] GPR [0 : REG\_FILE\_SIZE - 1];

integer i;

initial

begin

for(i = 0; i < REG\_FILE\_SIZE; i = i + 1)

GPR[i] = {DATA\_WIDTH{1'b0}};

OperandA = {DATA\_WIDTH{1'b0}};

OperandB = {DATA\_WIDTH{1'b0}};

GPR[1] = {{(DATA\_WIDTH-1){1'b0}},1'b1};

end

always@(posedge clk)

begin

if (reset)

for(i = 2; i < REG\_FILE\_SIZE; i = i + 1)

GPR[i] <= {DATA\_WIDTH{1'b0}};

else

if(wen && AdrWrite != {ADDR\_WIDTH{1'b0}} && AdrWrite != {{(ADDR\_WIDTH-1){1'b0}},1'b1})

GPR[AdrWrite] <= DATA;

end

always@(posedge clk)

begin

if(reset)

begin

OperandA <= {DATA\_WIDTH{1'b0}};

OperandB <= {DATA\_WIDTH{1'b0}};

end

else

begin

OperandA <= GPR[AdrA];

OperandB <= GPR[AdrB];

end

end

endmodule

Опишем основной модуль процессорного ядра, с заданной системой команд и стадиями.

CMD\_WIDTH и CMD\_ADDR\_WIDTH – параметр, определяющие ширину команды и ширину адреса в памяти команд.

DATA\_WIDTH и MEM\_ADDR\_WIDTH – ширина данных и ширина адреса в памяти данных.

GPR\_FILE\_SIZE и GPR\_ADDR\_WIDTH – размер регистрового файла и ширина адреса для регистров.

Cmd – память команд, загружаемую из внешнего файла.

Mem – память данных.

pr, pr\_next – регистры для хранения текущей и следующей команды.

alu1, alu2, alu1\_next, alu2\_next – регистры для хранения операндов.

res, res\_next – регистры для хранения промежуточных и конечных результатов операций.

stage\_counter – счетчик стадий, управляющий процессом выполнения команд.

AdrA, AdrB, AdrWrite – регистры адресов для операций чтения и записи.

OperandA, OperandB – выводы данных, соответствующие операндам, полученным из регистров.

В стадии 0 загружается команду из памяти команд в pr\_next, используя pc в качестве указателя. Подготавливает адреса регистров для следующей команды.

В стадии 1 определяются операнды, используя адреса AdrA и AdrB. Операнды загружаются из регистрового файла в alu1 и alu2.

В стадии 2 выполняются арифметические и логические операции в зависимости от типа команды. Результат сохраняется в res или устанавливает флаги (например, lz для команды SUB).

В стадии 3 для команд, требующих доступа к памяти (например, PTR), выполняет чтение или запись в память данных.

В стадии 4 записываются результаты в регистровый файл или обновляется память данных. Счётчик команд также обновляется в зависимости от результатов выполнения команд (например, для условных переходов) (Листинг 1.2).

Листинг 1.2 – Модуль вычисления косинуса и синуса по разложению в ряд Тейлора на языке Verilog

`timescale 1ns / 1ps

module cpu(

input clk,

input reset,

output stage\_counter

);

/\*

0000 — NOP пустая команда (ничего не делает).

0001 — INCR увеличение значения регистра на 1.

0010 — LTR загрузка константы в регистр.

0011 — NTR загрузка значения 0 в регистр.

0100 — LTM загрузка значения в память по указанному адресу.

0101 — SUB вычитание значений из двух регистров и сохранение результата.

0110 — JLZ условный переход, если результат предыдущей операции отрицательный.

0111 — PTR косвенное обращение к памяти.

1000 — SWP обмен значений между регистрами.

1001 — JMP безусловный переход на указанный адрес.

\*/

localparam NOP = 0, INCR = 1, LTR = 2, NTR = 3, LTM = 4, SUB = 5, JLZ = 6, PTR = 7, SWP = 8, JMP = 9;

localparam CMD\_WIDTH = 39;

localparam CMD\_ADDR\_WIDTH = 6;

reg [CMD\_WIDTH-1 : 0] cmd [0 : 2\*\*CMD\_ADDR\_WIDTH-1];

initial

begin

$readmemb("mem.mem", cmd);

end

localparam DATA\_WIDTH = 32;

localparam MEM\_ADDR\_WIDTH = 6;

reg [DATA\_WIDTH-1 : 0] mem [0 : 2\*\*MEM\_ADDR\_WIDTH-1];

// Параметры для регистра общего назначения (GPR)

localparam GPR\_FILE\_SIZE = 8;

localparam GPR\_ADDR\_WIDTH = $clog2(GPR\_FILE\_SIZE);

reg [GPR\_ADDR\_WIDTH-1:0] AdrA, AdrB, AdrWrite;

wire [DATA\_WIDTH-1:0] OperandA, OperandB;

reg wen;

reg [DATA\_WIDTH-1:0] DATA;

reg\_file GPR (

.clk(clk),

.reset(reset),

.wen(wen),

.DATA(DATA),

.AdrWrite(AdrWrite),

.AdrA(AdrA),

.AdrB(AdrB),

.OperandA(OperandA),

.OperandB(OperandB)

);

reg [CMD\_ADDR\_WIDTH-1 : 0] pc;

localparam COP\_WIDTH = 4;

reg [CMD\_WIDTH-1 : 0] pr, pr\_next;

reg [2:0] stage\_counter;

always@(posedge clk)

begin

if (reset || stage\_counter == 4)

stage\_counter <= 0;

else

stage\_counter <= stage\_counter + 1;

end

always@(posedge clk)

begin

if (reset)

pc <= 0;

else

if (stage\_counter == 4)

JMP: pc <= ja;

JLZ: if (~lz)

pc <= ja;

else

pc <= pc + 1;

default: pc <= pc + 1;

endcase

end

always@(posedge clk)

begin

if (reset)

pr <= 0;

else

pr <= pr\_next;

end

always@\*

begin

if (stage\_counter == 0)

pr\_next <= cmd[pc];

else

pr\_next <= pr;

end

reg [DATA\_WIDTH-1:0] alu1, alu1\_next, alu2, alu2\_next;

always@(posedge clk)

begin

if (reset)

begin

alu1 <= 0;

alu2 <= 0;

end

else

begin

alu1 <= alu1\_next;

alu2 <= alu2\_next;

end

end

wire [COP\_WIDTH-1 : 0] cop = pr[CMD\_WIDTH-1 -: COP\_WIDTH];

wire [GPR\_ADDR\_WIDTH-1 : 0] addr1 = pr[CMD\_WIDTH-1-COP\_WIDTH -: GPR\_ADDR\_WIDTH];

wire [GPR\_ADDR\_WIDTH-1 : 0] addr2 = pr[CMD\_WIDTH-1-COP\_WIDTH-GPR\_ADDR\_WIDTH -: GPR\_ADDR\_WIDTH];

wire [CMD\_ADDR\_WIDTH-1 : 0] ja = pr[CMD\_WIDTH-1-COP\_WIDTH -: CMD\_ADDR\_WIDTH];

wire [COP\_WIDTH-1 : 0] cop\_next = pr\_next[CMD\_WIDTH-1 -: COP\_WIDTH];

wire [GPR\_ADDR\_WIDTH-1 : 0] addr1\_next = pr\_next[CMD\_WIDTH-1-COP\_WIDTH -: GPR\_ADDR\_WIDTH];

wire [GPR\_ADDR\_WIDTH-1 : 0] addr2\_next = pr\_next[CMD\_WIDTH-1-COP\_WIDTH-GPR\_ADDR\_WIDTH -: GPR\_ADDR\_WIDTH];

always@(\*)

begin

if (stage\_counter == 1)

begin

case(cop)

INCR, LTM, SUB, PTR, NTR:

begin

alu1\_next <= OperandA;

end

default:

alu1\_next <= alu1;

endcase

end

else

alu1\_next <= alu1;

end

always@(\*)

begin

if (stage\_counter == 1)

begin

case(cop)

INCR, SUB, SWP, NTR:

alu2\_next <= OperandB;

PTR:

alu2\_next <= addr2;

default:

alu2\_next <= alu2;

endcase

end

else

alu2\_next <= alu2;

end

always@(\*)

begin

if (stage\_counter == 0)

begin

case(cop\_next)

INCR, LTM, SUB, PTR:

AdrA <= addr1\_next;

NTR:

AdrA <= 0;

default:

AdrA <= 0;

endcase

end

else

AdrA <= 0;

end

always@(\*)

begin

if (stage\_counter == 0)

begin

case(cop\_next)

SUB:

AdrB <= addr2\_next;

NTR:

AdrB <= 0;

INCR:

AdrB <= 1;

SWP:

AdrB <= addr1\_next;

default:

AdrB <= 0;

endcase

end

else

AdrB <= 0;

end

reg [2\*DATA\_WIDTH-1:0] res, res\_next;

always@(posedge clk)

if (reset)

res <= 0;

else

res <= res\_next;

always@(\*)

begin

if (stage\_counter == 2)

begin

case(cop)

INCR:

res\_next <= alu1 + alu2;

NTR, LTM:

res\_next <= alu1;

SUB:

res\_next <= alu1 - alu2;

PTR:

res\_next <= { alu1 + 1, alu2 + 1 };

SWP:

res\_next <= alu2 + 1;

default:

res\_next <= res;

endcase

end

else

res\_next <= res;

end

reg lz, lz\_next;

always@(posedge clk)

if (reset)

lz <= 0;

else

lz <= lz\_next;

always@(\*)

begin

if (stage\_counter == 2)

case(cop)

SUB:

lz\_next <= res\_next[63];

default:

lz\_next <= lz;

endcase

else

lz\_next <= lz;

end

wire [DATA\_WIDTH-1 : 0] lit = pr[DATA\_WIDTH-1 -: DATA\_WIDTH];

always@(\*)

begin

case(stage\_counter)

3: begin

case(cop)

PTR:

begin

wen <= 1;

AdrWrite <= alu2;

DATA <= mem[alu1];

end

default:

begin

wen <= 0;

AdrWrite <= alu2;

DATA <= mem[alu2];

end

endcase

end

4: begin

case(cop)

INCR, NTR:

begin

wen <= 1;

AdrWrite <= addr1;

DATA <= res[31:0];

end

LTR:

begin

wen <= 1;

AdrWrite <= addr1;

DATA <= lit;

end

SUB:

begin

wen <= 1;

AdrWrite <= 4;

DATA <= res[31:0];

end

PTR:

begin

wen <= 1;

AdrWrite <= res[31:0];

DATA <= mem[res[63:32]];

end

default:

begin

wen <= 0;

AdrWrite <= addr1;

DATA <= lit;

end

endcase

end

default:

begin

wen <= 0;

AdrWrite <= addr1;

DATA <= lit;

end

endcase

end

always@(posedge clk)

begin

if (stage\_counter == 4)

case(cop)

SWP:

begin

mem[alu2] <= mem[res];

mem[res] <= mem[alu2];

end

LTM:

begin

mem[res] <= lit;

end

endcase

end

endmodule

Напишем тестовый модуль для проверки работы программы (Листинг 1.3).

Листинг 1.3 – Тестовый модуль на языке Verilog

`timescale 1ns / 1ps

module test;

reg clk = 0;

always #0.5 clk = ~clk;

reg reset = 0;

wire [2:0] stage\_counter;

cpu uut (

.clk(clk),

.reset(reset),

.stage\_counter(stage\_counter)

);

initial

begin

reset = 1;

@(posedge clk);

@(posedge clk);

reset = 0;

end

endmodule

Опишем программу (инструкции в двоичном формате) через mem файл. (Листинг 1.4).

Листинг 1.4 – Программа для процессорного ядра в двоичном формате

0010\_010\_00000000\_00000000\_00000000\_00000011 // LTR 2 (3) (cmd[0] = 1200000003)

0011\_011\_00000000\_00000000\_00000000\_00000000 // NTR 3 (cmd[1] = 1b00000000)

1001\_010100\_00000000\_00000000\_00000000\_00000 // JMP 20 (cmd[2] = 4a80000000)

0010\_011\_00000000\_00000000\_00000000\_00000001 // LTR 3 (1) (cmd[9] = 1300000001)

0101\_011\_010\_100\_00000000\_00000000\_0000000000 // SUB 3, 2, 4 (cmd[10] = 2b50000000)

0110\_010010\_00000000\_00000000\_00000000\_00000 // JLZ 18 (cmd[11] = 3240000000)

0011\_101\_00000000\_00000000\_00000000\_00000000 // NTR 5 (cmd[12] = 1d00000000)

0101\_010\_011\_100\_00000000\_00000000\_0000000000 // SUB 2, 3, 4 (cmd[13, 22, 31..] = 2a70000000)

0101\_101\_100\_100\_00000000\_00000000\_0000000000 // SUB 5, 4, 4 (cmd[14, 23] = 2d90000000)

0110\_010000\_00000000\_00000000\_00000000\_00000 // JLZ 16 (cmd[15, 24] = 3200000000)

0111\_101\_110\_000\_00000000\_00000000\_0000000000 // PTR 5, 6 (cmd[16, 25] = 3dc0000000)

0101\_110\_111\_100\_00000000\_00000000\_0000000000 // SUB 6, 7, 4 (cmd[17, 26] = 2ef0000000)

0110\_001110\_00000000\_00000000\_00000000\_00000 // JLZ 14 (cmd[18, 27] = 31c0000000)

1000\_101\_00000000\_00000000\_00000000\_00000000 // SWP 5 (cmd[19, 28] = 4500000000)

0001\_101\_00000000\_00000000\_00000000\_00000000 // INCR 5 (cmd[20, 29] = 0d00000000)

1001\_000111\_00000000\_00000000\_00000000\_00000 // JMP 7 (cmd[21, 30] = 48e0000000)

0001\_011\_00000000\_00000000\_00000000\_00000000 // INCR 3 (0b00000000)

1001\_000100\_00000000\_00000000\_00000000\_00000 // JMP 4 (4880000000)

0000\_000000\_00000000\_00000000\_00000000\_00000

0000\_000000\_00000000\_00000000\_00000000\_00000

0100\_011\_00000000\_00000000\_00000000\_00000101 // LTM 3 (5) (cmd[3] = 2300000005)

0001\_011\_00000000\_00000000\_00000000\_00000000 // INCR 3 (cmd[4] = 0b00000000)

0100\_011\_00000000\_00000000\_00000000\_00000111 // LTM 3 (7) (cmd[5] = 2300000007)

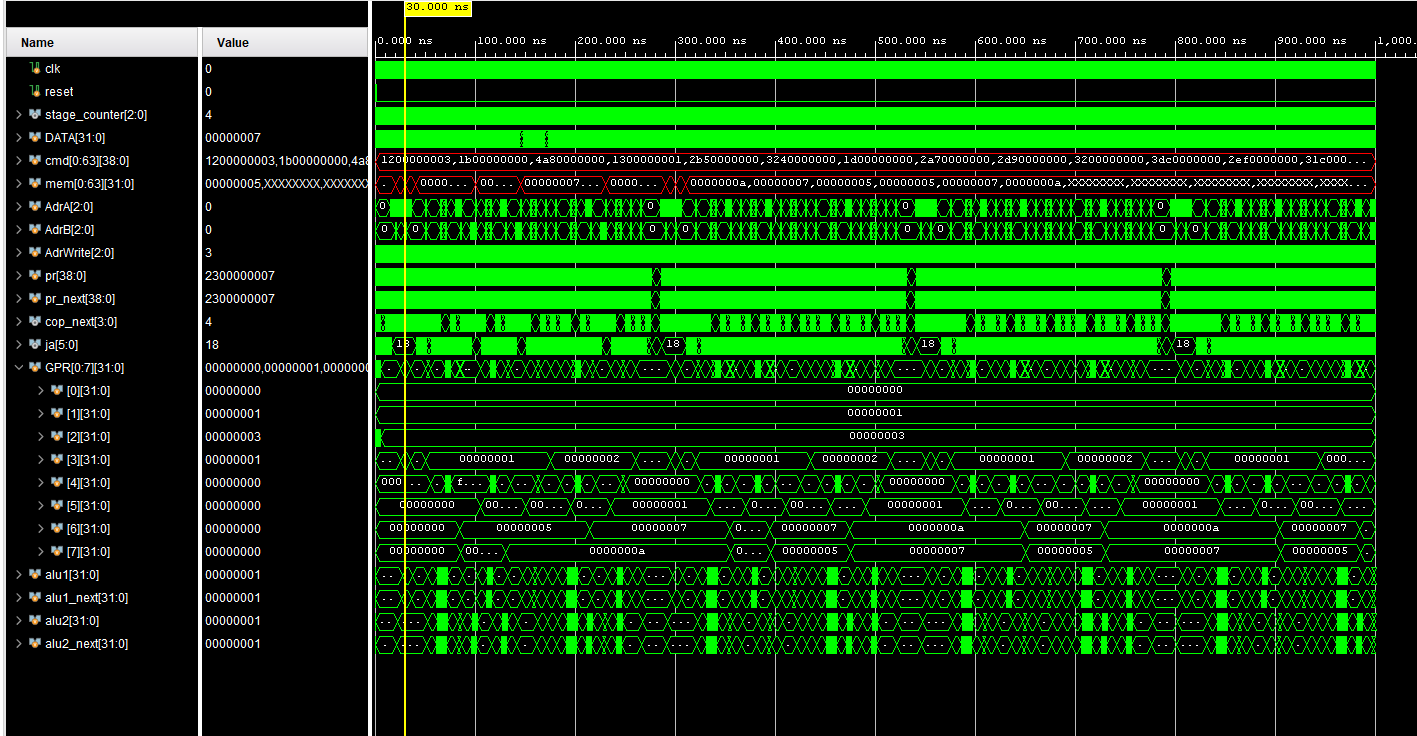
0001\_011\_00000000\_00000000\_00000000\_00000000 // INCR 3 (cmd[6] = 0b00000000)

0100\_011\_00000000\_00000000\_00000000\_00001010 // LTM 3 (10) (cmd[7] = 230000000a)

1001\_000011\_00000000\_00000000\_00000000\_00000 // JMP 3 (cmd[8] = 4860000000)

1.6 Тестирование работы процессора

Протестируем работу процессора. Общая временная диаграмма показана ниже (Рисунок 1.3).

Рисунок 1.3 – Временная диаграмма тестового модуля

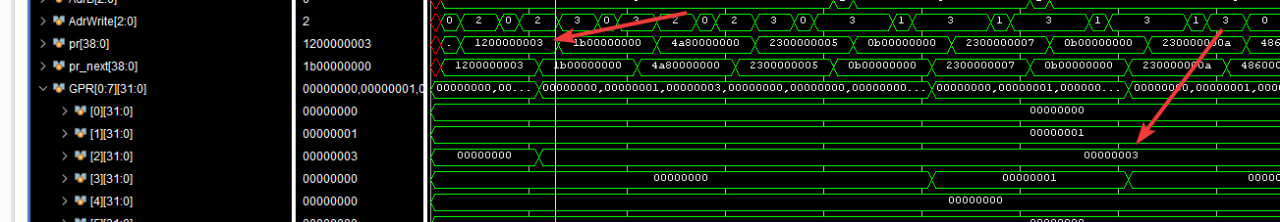
Рассмотрим диаграмму подробнее. Например, команда 1200000003 записывает в регистр 2 литерал 3 (Рисунок 1.4).

Рисунок 1.4 – Результат работы команды LTR 2 3

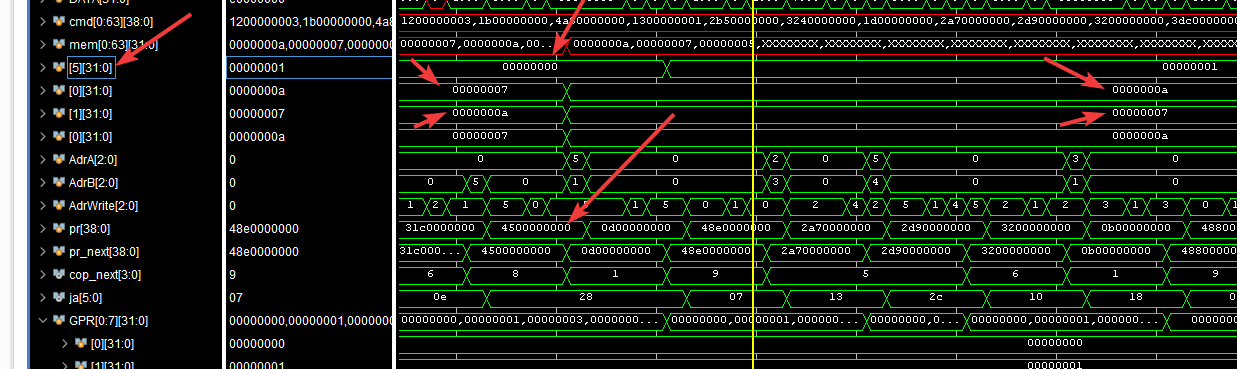
SWP 5 (команда 4500000000 в HEX формате) – поменять местами значения, хранящиеся в памяти по адресу, указанному в регистре 5. В регистре 5 указан адрес нуля на момент исполнения команды, следовательно значения в ячейках 00000000 и 00000001 поменяются местами.

Рисунок 1.5 – Результат работы команды SWP 5

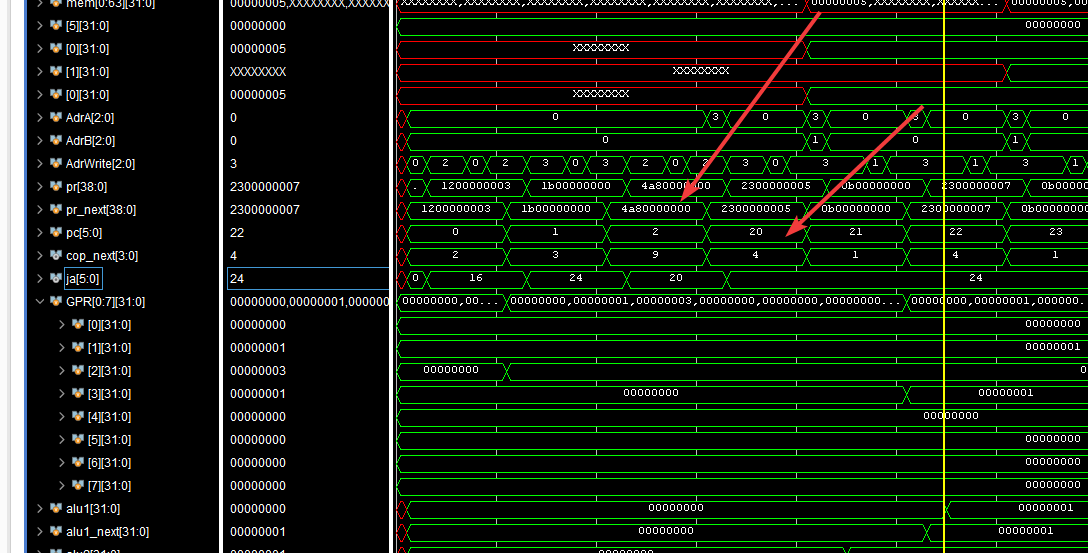
Команда JMP 20 перейдёт по адресу 20, что мы видим в счётчике команд (Рисунок 1.6).

Рисунок 1.6 – Результат работы команды JMP 20

ЗАКЛЮЧЕНИЕ

В рамках данной практической работы спроектировали порстое процессорное ядро и разработали все необходимые модули и коды, обеспечивающих выполнение базовых операций, обработку команд и взаимодействие с памятью.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Методические указания по ПР № 1 — URL: https://online-edu.mirea.ru/mod/resource/view.php?id=405132 (Дата обращения: 23.09.2022).

2. Методические указания по ПР № 2 — URL: https://online-edu.mirea.ru/mod/resource/view.php?id=409130 (Дата обращения: 23.09.2022).

3. Смирнов С.С. Информатика [Электронный ресурс]: Методические указания по выполнению практических и лабораторных работ / С.С. Смирнов — М., МИРЭА — Российский технологический университет, 2018. — 1 электрон. опт. диск (CD-ROM).

4. Тарасов И.Е. ПЛИС Xilinx. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования. — М.: Горячая линия — Телеком, 2021. — 538 с.: ил.

5. Антик М.И. Дискретная математика [Электронный ресурс]: Учебное пособие / Антик М.И., Казанцева Л.В. — М.: МИРЭА — Российский технологический университет, 2018 — 1 электрон. опт. диск (CD-ROM).

6. Антик М.И. Математическая логика и программирование в логике [Электронный ресурс]: Учебное пособие / Антик М.И., Бражникова Е.В.— М.: МИРЭА – Российский технологический университет, 2018. — 1 электрон. опт. диск (CD-ROM).

7. Жемчужникова Т.Н. Конспект лекций по дисциплине «Архитектура вычислительных машин и систем» — URL: https://drive.google.com/file/d/12OAi2\_axJ6mRr4hCbXs-mYs8Kfp4YEfj/view?us  
p=sharing (Дата обращения: 23.09.2022).

8. Антик М.И. Теория автоматов в проектировании цифровых схем [Электронный ресурс]: Учебное пособие / Антик М.И., Казанцева Л.В. — М.: МИРЭА – Российский технологический университет, 2020. — 1 электрон. опт. диск (CD-ROM).