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

Институт Информационных Технологий

Кафедра Вычислительной Техники (ВТ)

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №2**

«Создание последовательного процессорного ядра»

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 10 рисунок, 10 листингов. Количество страниц в работе — 58.

СОДЕРЖАНИЕ

[АННОТАЦИЯ 2](#_Toc184937581)

[ВВЕДЕНИЕ 4](#_Toc184937582)

[1 ОСНОВНОЙ РАЗДЕЛ 5](#_Toc184937583)

[1.1 Основной алгоритм 5](#_Toc184937584)

[1.2 Алгоритм на основе динамического программирования 6](#_Toc184937585)

[1.3 Минимальный алгоритм на основе динамического программирования 12](#_Toc184937586)

[1.4 Низкоуровневый алгоритм версии 1 16](#_Toc184937587)

[1.5 Низкоуровневый алгоритм версии 2 21](#_Toc184937588)

[1.6 Конечный автомат реализующий заданный алгоритм 26](#_Toc184937589)

[1.7 Архитектура и микроархитектура последовательного процессорного ядра 34](#_Toc184937590)

[1.8 Описание программы в машинных кодах 40](#_Toc184937592)

[1.9 Верификация процессорного ядра 54](#_Toc184937593)

ВВЕДЕНИЕ

Целью данной лабораторной работы является проектирование и верификация специализированного последовательного процессорного ядра для решения задачи нахождения длины наибольшей возрастающей подпоследовательности (НВП) в заданном массиве чисел.

В ходе выполнения работы будут рассмотрены следующие этапы:

* Описание алгоритма решения задачи. Будет представлен и проанализирован алгоритм нахождения НВП с использованием динамического программирования;
* Описание архитектуры и микроархитектуры процессорного ядра. Будет разработана архитектура специализированного процессора, оптимизированного для выполнения выбранного алгоритма. Будут определены основные компоненты ядра, такие как регистры, память, арифметико-логическое устройство (АЛУ), и описаны их взаимодействие и функции;
* Описание программы в машинных кодах. Алгоритм решения задачи будет реализован в виде программы на языке ассемблера, разработанном для спроектированного процессорного ядра. Каждая команда программы будет подробно описана.

В качестве индивидуального варианта задания предлагается рассмотреть последовательно несколько различных алгоритмов решения задачи нахождения НВП, реализованных на языке C++ (в количестве 5 штук различного низкоуровневого кода). Каждый алгоритм демонстрирует свой подход к решению задачи, что позволяет проанализировать различные варианты архитектурных решений для процессорного ядра. После чего реализуем самый низкоуровневый алгоритм в форме кода Verilog, далее реализуем процессорную программу и сам процессор.

1 ОСНОВНОЙ РАЗДЕЛ

* 1. Основной алгоритм

За основу процессора возьмём одну из самых простых реализаций нахождение длины наибольшей возрастающей подпоследовательности (НВП) реализован с использованием рекурсивного подхода и динамического программирования.

Алгоритм основан на идее, что длина НВП, заканчивающейся в элементе arr[i], равна максимальному из значений (длина НВП, заканчивающейся в элементе arr[j] + 1), где j находится в диапазоне от 0 до i-1 и arr[j] < arr[i] (Листинг 1.1).

Рассмотрим массив arr = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28}.

В итоге lis(arr) вернет 6 (Рисунок 1.1), что является длиной НВП для данного массива. Однако данный алгоритм нигде не сохраняет заданную последовательность для более удобного отображения.

Листинг 1.1 – Основной алгоритм на языке C++

#include <iostream>

#include <vector>

using namespace std;

int lisEndingAtIdx(vector<int>& arr, int idx) {

if (idx == 0)

return 1;

int mx = 1;

for (int prev = 0; prev < idx; prev++)

if (arr[prev] < arr[idx])

mx = max(mx, lisEndingAtIdx(arr, prev) + 1);

return mx;

}

int lis(vector<int>& arr) {

int n = arr.size();

int res = 1;

for (int i = 1; i < n; i++)

res = max(res, lisEndingAtIdx(arr, i));

return res;

}

Продолжение Листинга 1.1

int main() {

vector<int> arr = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};

cout << lis(arr);

return 0;

}

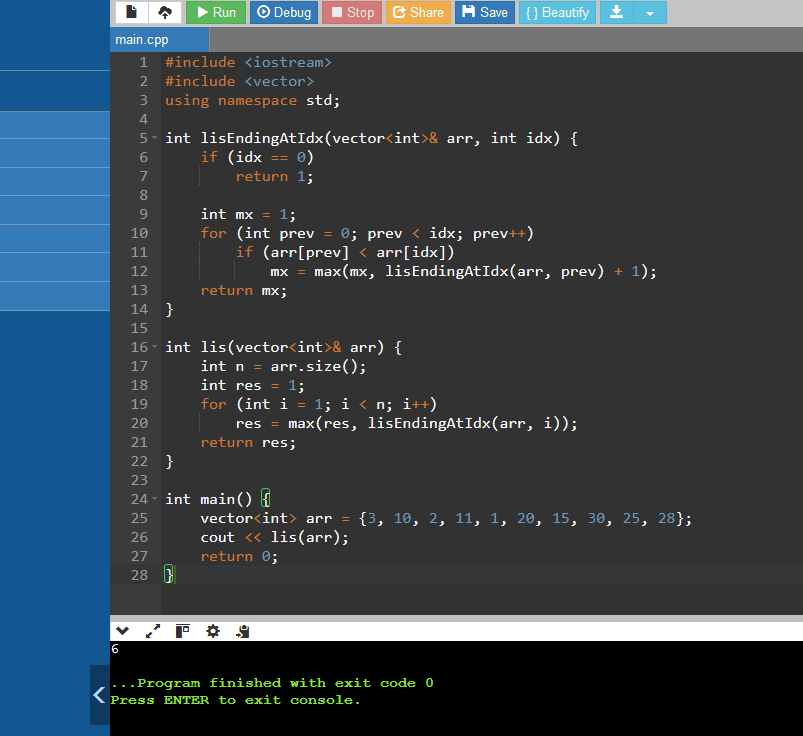


Рисунок 1.1 – результат выполнения основного алгоритма

* 1. Алгоритм на основе динамического программирования

Во второй версии алгоритма откажемся от использования циклов и функций и перейдём на более псевдо ассемблерный код.

Все промежуточные результаты хранятся в массиве dp, а связи между элементами — в массиве prev.

В псевдоассемблерной версии введены массивы:

* dp — для хранения длины LIS, заканчивающихся на каждом элементе;
* prev — для отслеживания предыдущего элемента в LIS;
* lis — для восстановления итоговой последовательности;
* result — для вывода результата.

Алгоритм также использует регистры (например, R1, R2 и т. д.) для хранения временных переменных и указателей.

Применяет понятные инструкции (MOV, CMP, INCR, DECR, JMP), которые имитируют выполнение на процессоре.

* INCR R — Увеличение значения в регистре R на 1;
* DECR R — Уменьшение значения в регистре R на 1;
* XOR R — Установка значения регистра R в 0;
* MOV dest, src — Копирование значения из srrc в dest (например MOV dp[R1], 1 означает dp[R1] = 1, а MOV R5, dp[R1] означает R5 = dp[R1]);
* CMP R1, R2 — Сравнение значений R1 и R2;
* Условные и безусловные переходы.

Все необходимые массивы (dp, prev, lis, result) и регистры (например, R1, R2, R6) инициализируются.

Массив dp заполняется значениями 1, так как каждая отдельная точка (элемент массива input) сама по себе может быть подпоследовательностью длиной 1.

Массив prev заполняется значением MAX\_SIZE, что обозначает отсутствие связи с предыдущими элементами в начале.

Внешний цикл проходит по массиву input с индексом R1 от 1 до N−1.

Внутренний цикл проверяет все предыдущие элементы (R2 от 0 до R1−1) на возможность включения их в возрастающую подпоследовательность (Листинг 1.2).

Условие включения элемента в подпоследовательность: input[R2]<input[R1] — текущий элемент может продолжить возрастающую подпоследовательность, а также условие dp[R1]<dp[R2]+1 — длина подпоследовательности увеличивается.

Если оба условия выполняются: dp[R1] обновляется, увеличиваясь на 1, в массив prev[R1] записывается индекс предыдущего элемента (R2), который стал частью подпоследовательности.

После завершения работы вложенных циклов массив dp содержит длины LIS, заканчивающихся на каждом элементе.

Новый цикл проходит по массиву dp и определяет максимальное значение длины (R5) и индекс элемента, на котором эта длина заканчивается (R6).

С использованием массива prev и индекса R6 начинается обратный проход по цепочке элементов, определенной в prev. Каждый элемент из массива input записывается в массив lis в порядке добавления. Процесс завершается, когда достигается значение MAX\_SIZE в массиве prev (нет предыдущего элемента).

На этапе реверсирования массив lis содержит LIS в обратном порядке. Используя два указателя (R1 и R2), выполняется реверс массива для получения подпоследовательности в правильном порядке.

Итоговая подпоследовательность копируется из массива lis в массив result. Массив result содержит окончательный ответ, который можно вывести.

В результате сохранения мы можем проверить нашу итоговую последовательность и вывести её на экран (Рисунок 1.2)

Листинг 1.2 – Алгоритм на основе динамического программированияна языке C++

#include <iostream>

using namespace std;

const int MAX\_SIZE = 100;

int main() {

int input[] = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};

int N = 10;

int lis\_length = 0;

int dp[MAX\_SIZE];

int prev[MAX\_SIZE];

int lis[MAX\_SIZE];

int result[MAX\_SIZE];

int R1, R2, R3, R4, R5, R6;

// Initialize dp and prev

R1 = 0; // 0000 XOR R1

init\_loop\_start:

if (R1 >= N) // 0001 CMP R1, N

goto init\_loop\_end; // 0002 JGE 0007 >=

dp[R1] = 1; // 0003 MOV dp[R1], 1

prev[R1] = MAX\_SIZE; // 0004 MOV prev[R1], MAX\_SIZE

R1 = R1 + 1; // 0005 INCR R1

goto init\_loop\_start; // 0006 JMP 0001

init\_loop\_end:

// Fill dp and prev

R1 = 1; // 0007 MOV R1, 1

outer\_loop\_start:

if (R1 >= N) // 0008 CMP R1, N

goto outer\_loop\_end; // 0009 JGE 0018

R2 = 0; // 0010 XOR R2

inner\_loop\_start:

if (R2 >= R1) // 0011 CMP R2, R1

goto inner\_loop\_end; // 0012 JGE 0016

if (input[R2] < input[R1]) { // 0013 CMP input[R2], input[R1]

if (dp[R1] < dp[R2] + 1) { // 0015 CMP dp[R1], dp[R2] + 1

dp[R1] = dp[R2] + 1; // 0017 MOV dp[R1], dp[R2] + 1

prev[R1] = R2; // 0018 MOV prev[R1], R2

}

}

R2 = R2 + 1; // 0019 INCR R2

goto inner\_loop\_start; // 0020 JMP 0011

inner\_loop\_end:

R1 = R1 + 1; // 0021 INCR R1

goto outer\_loop\_start; // 0022 JMP 0008

outer\_loop\_end:

// Find the index of the maximum element in dp

R1 = 0; // 0023 XOR R1

R5 = 0; // 0024 XOR R5

R6 = MAX\_SIZE; // 0025 MOV R6, MAX\_SIZE

find\_max\_start:

if (R1 >= N) // 0026 CMP R1, N

Продолжение Листинга 1.2

goto find\_max\_end; // 0027 JGE 0032

if (dp[R1] > R5) { // 0028 CMP dp[R1], R5

R5 = dp[R1]; // 0030 MOV R5, dp[R1]

R6 = R1; // 0031 MOV R6, R1

}

R1 = R1 + 1; // 0032 INCR R1

goto find\_max\_start; // 0033 JMP 0026

find\_max\_end:

// Restore LIS from dp and prev

restore\_lis\_start:

if (R6 == MAX\_SIZE) // 0034 CMP R6, MAX\_SIZE

goto restore\_lis\_end; // 0035 JEQ 0040

lis[lis\_length] = input[R6]; // 0036 MOV lis[lis\_length], input[R6]

lis\_length = lis\_length + 1; // 0037 INCR lis\_length

R6 = prev[R6]; // 0038 MOV R6, prev[R6]

goto restore\_lis\_start; // 0039 JMP 0034

restore\_lis\_end:

// Reverse the lis array manually

R1 = 0; // 0040 XOR R1

R2 = lis\_length - 1; // 0041 SUB R2, lis\_length - 1

reverse\_loop\_start:

if (R1 >= R2) // 0042 CMP R1, R2

goto reverse\_loop\_end; // 0043 JGE 0050

R3 = lis[R1]; // 0044 MOV R3, lis[R1]

R4 = lis[R2]; // 0045 MOV R4, lis[R2]

lis[R1] = R4; // 0046 MOV lis[R1], R4

lis[R2] = R3; // 0047 MOV lis[R2], R3

R1 = R1 + 1; // 0048 INCR R1

R2 = R2 - 1; // 0049 DECR R2

goto reverse\_loop\_start; // 0050 JMP 0042

reverse\_loop\_end:

R1 = 0; // 0051 XOR R1

output\_loop\_start:

if (R1 >= lis\_length) // 0052 CMP R1, lis\_length

goto output\_loop\_end; // 0053 JGE 0057

result[R1] = lis[R1]; // 0054 MOV result[R1], lis[R1]

R1 = R1 + 1; // 0055 INCR R1

goto output\_loop\_start; // 0056 JMP 0052

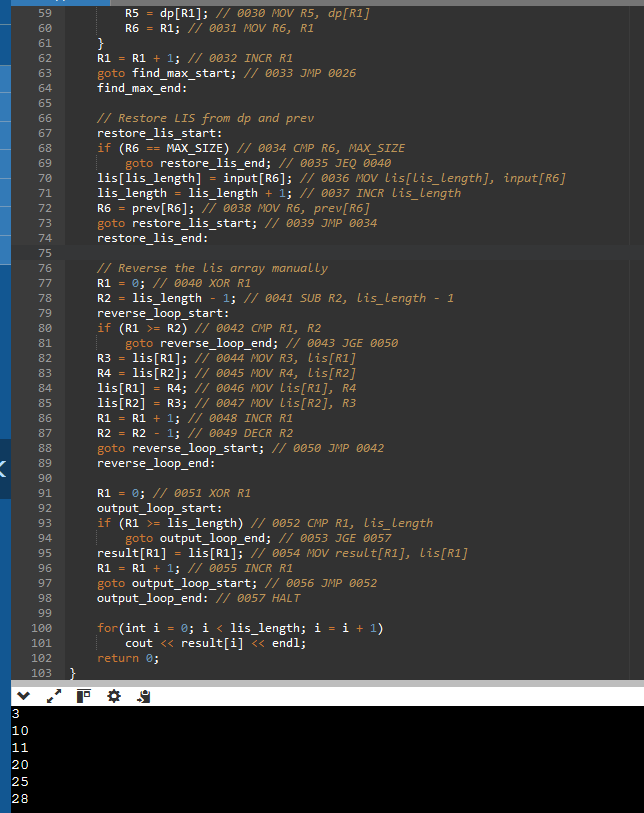
output\_loop\_end: // 0057 HALT

for(int i = 0; i < lis\_length; i = i + 1)

cout << result[i] << endl;

return 0;

}

Рисунок 1.2 – результат работы алгоритма на основе динамического программирования

* 1. Минимальный алгоритм на основе динамического программирования

Оптимизируем и упростим второй алгоритм, данный выше. Ранее мы использовали обычные for-циклы, в новой версии мы будем использовать только вложенные циклы, которые будут реализованы через условные переходы (JMP, CMP).

В первую очередь избавимся от регистра result, будем заносить последовательность и разворачивать её напрямую в одном регистре lis. Это упростит количество использованной памяти.

Во втором алгоритме использовалась индексация через стандартный доступ (arr[i]). Далее используются адресные вычисления и операции над регистрами (MOV, CMP).

Во втором алгоритме дублировались регистры R5 и R6. R5 хранил максимальное значение, найденное в dp. R6 хранил индекс элемента с максимальным значением в dp. Возможно их упрощение до количества регистров равным 4, в следствии чего логика работы алгоритма чуть упроститься.

R1 используется как:

* Счетчик в циклах инициализации dp и prev, внешнем цикле заполнения dp и prev, цикле поиска максимума в dp и цикле разворота lis;
* Индекс текущего элемента в массивах dp, prev, input и lis.

R2 используется как:

* Счетчик во внутреннем цикле заполнения dp и prev;
* Хранит максимальное значение, найденное в dp на данный момент;
* Правая граница (индекс) в цикле разворота lis.

R3 используется как:

* Хранит индекс элемента с максимальным значением в dp;
* Временно хранит элемент lis[R1] при развороте lis.

R4 ипользуется только в цикле разворота lis. Он временно хранит элемент lis[R2] при развороте lis.

Алгоритм и его основные этапы останутся неизменными. Алгоритм также будет работоспособен (Рисунок 1.3).

Листинг 1.3 – Оптимизированный алгоритм на основе динамического программирования на языке C++

#include <iostream>

using namespace std;

int main() {

int MAX\_SIZE = 100;

int dp[MAX\_SIZE];

int prev[MAX\_SIZE];

int lis[MAX\_SIZE];

int result[MAX\_SIZE];

int input[] = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};

int N = 10;

int lis\_length = 0;

int R1, R2, R3, R4;

// Initialize dp and prev

R1 = 0; // 0000 XOR R1

init\_loop\_start:

if (R1 >= N) // 0001 CMP R1, N

goto init\_loop\_end; // 0002 JGE 0007 >=

dp[R1] = 1; // 0003 MOV dp[R1], 1

prev[R1] = MAX\_SIZE; // 0004 MOV prev[R1], MAX\_SIZE

R1 = R1 + 1; // 0005 INCR R1

goto init\_loop\_start; // 0006 JMP 0001

init\_loop\_end:

// Fill dp and prev

R1 = 1; // 0007 MOV R1, 1

outer\_loop\_start:

if (R1 >= N) // 0008 CMP R1, N

goto outer\_loop\_end; // 0009 JGE 0018

R2 = 0; // 0010 XOR R2

inner\_loop\_start:

if (R2 >= R1) // 0011 CMP R2, R1

goto inner\_loop\_end; // 0012 JGE 0016

if (input[R2] < input[R1]) { // 0013 CMP input[R2], input[R1]

if (dp[R1] < dp[R2] + 1) { // 0015 CMP dp[R1], dp[R2] + 1

dp[R1] = dp[R2] + 1; // 0017 MOV dp[R1], dp[R2] + 1

Продолжение Листинга 1.3

prev[R1] = R2; // 0018 MOV prev[R1], R2

}

}

R2 = R2 + 1; // 0019 INCR R2

goto inner\_loop\_start; // 0020 JMP 0011

inner\_loop\_end:

R1 = R1 + 1; // 0021 INCR R1

goto outer\_loop\_start; // 0022 JMP 0008

outer\_loop\_end:

// Find the index of the maximum element in dp

R1 = 0; // 0023 XOR R1

R2 = 0; // 0024 XOR R2

R3 = MAX\_SIZE; // 0025 MOV R3, MAX\_SIZE

find\_max\_start:

if (R1 >= N) // 0026 CMP R1, N

goto find\_max\_end; // 0027 JGE 0032

if (dp[R1] > R2) { // 0028 CMP dp[R1], R2

R2 = dp[R1]; // 0030 MOV R2, dp[R1]

R3 = R1; // 0031 MOV R3, R1

}

R1 = R1 + 1; // 0032 INCR R1

goto find\_max\_start; // 0033 JMP 0026

find\_max\_end:

// Restore LIS from dp and prev

restore\_lis\_start:

if (R3 == MAX\_SIZE) // 0034 CMP R3, MAX\_SIZE

goto restore\_lis\_end; // 0035 JEQ 0040

lis[lis\_length] = input[R3]; // 0036 MOV lis[lis\_length], input[R3]

lis\_length = lis\_length + 1; // 0037 INCR lis\_length

R3 = prev[R3]; // 0038 MOV R3, prev[R3]

goto restore\_lis\_start; // 0039 JMP 0034

restore\_lis\_end:

// Reverse the lis array manually

R1 = 0; // 0040 XOR R1

R2 = lis\_length - 1; // 0041 SUB R2, lis\_length - 1

reverse\_loop\_start:

if (R1 >= R2) // 0042 CMP R1, R2

goto reverse\_loop\_end; // 0043 JGE 0050

R3 = lis[R1]; // 0044 MOV R3, lis[R1]

R4 = lis[R2]; // 0045 MOV R4, lis[R2]

lis[R1] = R4; // 0046 MOV lis[R1], R4

lis[R2] = R3; // 0047 MOV lis[R2], R3

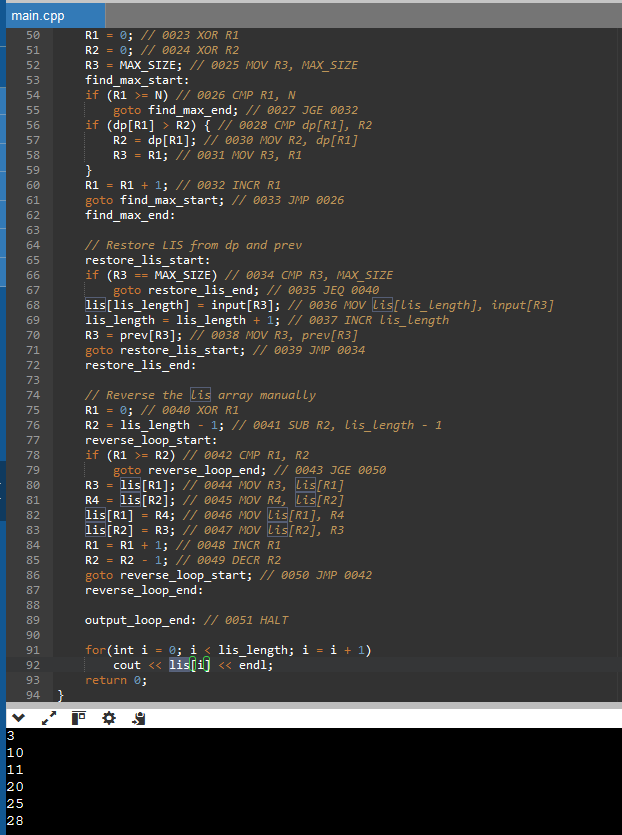
R1 = R1 + 1; // 0048 INCR R1

R2 = R2 - 1; // 0049 DECR R2

goto reverse\_loop\_start; // 0050 JMP 0042

reverse\_loop\_end:

output\_loop\_end: // 0051 HALT

Рисунок 1.3 – результат работы минимального алгоритма на основе динамического программирования

* 1. Низкоуровневый алгоритм версии 1

Продолжим работу над упрощением алгоритма. В четвёртой версии явно эмулируем работу с памятью через общую шину. Вместо непосредственного использования переменных, как в предыдущих версиях (dp, prev, lis, input), здесь все данные хранятся в едином массиве memory, представляющем собой эмуляцию общей памяти. Доступ к данным осуществляется через указатели (адреса).

Ранее алгоритм использовал отдельные массивы dp, prev, lis, input, к которым можно было обращаться напрямую по имени. Теперь же все данные находятся в одномерном массиве memory. Доступ к элементам осуществляется через адреса, заданные константами (dp\_ADDR, prev\_ADDR, lis\_ADDR, input\_ADDR).

Также изменим логику работы указателей. Ранее алгоритм явно использовал указатели при обращении к элементам массивов (например, dp[R1] – это доступ к элементу массива dp по индексу, хранящемуся в R1).

Изменим эту логику. Теперь адреса будут определяться в памяти для каждой переменной и массива. Доступ к данным осуществляется путем вычисления смещения относительно базового адреса. Например, memory[dp\_ADDR + memory[R1\_ADDR]] означает доступ к элементу массива dp, который находится по адресу dp\_ADDR плюс смещение, равное значению, хранящемуся в ячейке памяти с адресом R1\_ADDR.

Также откажемся от абстрактных регистров. Ранее использовались обозначения R1, R2, R3, R4 как абстрактные регистры. Теперь регистры эмулируют адреса, выделенные для них ячейки в памяти (R1\_ADDR, R2\_ADDR, R3\_ADDR, R4\_ADDR). Работа с «регистрами» сводится к чтению и записи значений в соответствующие ячейки памяти.

В новом алгоритме массив memory представляет собой общую память, а все операции с данными (чтение и запись) неявно осуществляются через общую шину. В каждый момент времени через общую шину может передаваться только один запрос (на чтение или запись). Это означает, что, если нескольким частям процессора одновременно потребуются данные из памяти, им придется ждать своей очереди (Листинг 1.4).

Вывод алгоритма остаётся неизменным (Рисунок 1.4).

Листинг 1.4 – Низкоуровневый алгоритм версии 1 на языке C++

#include <iostream>

using namespace std;

const int MEMORY\_SIZE = 1000;

int memory[MEMORY\_SIZE];

// Memory addresses

const int MAX\_SIZE\_ADDR = 0;

const int dp\_ADDR = 1; // dp starts at 1

const int prev\_ADDR = 101; // prev starts at 101

const int lis\_ADDR = 201; // lis starts at 201

const int input\_ADDR = 301; // input starts at 301

const int N\_ADDR = 401;

const int lis\_length\_ADDR = 402;

const int R1\_ADDR = 403;

const int R2\_ADDR = 404;

const int R3\_ADDR = 405;

const int R4\_ADDR = 406;

const int result\_ADDR = 407;

int main() {

// Initialize constants

memory[MAX\_SIZE\_ADDR] = 100;

memory[N\_ADDR] = 10;

// Initialize input

int input\_data[] = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};

for(int i = 0; i < memory[N\_ADDR]; ++i) {

memory[input\_ADDR + i] = input\_data[i];

}

// Initialize dp and prev

// 0000 XOR R1

memory[R1\_ADDR] = 0;

// 0001 CMP\_REG R1, [N]

// 0002 JGE 0007

init\_loop\_start:

if(memory[R1\_ADDR] >= memory[N\_ADDR])

goto init\_loop\_end;

// 0003 MOV\_MEM [dp + R1], 1

memory[dp\_ADDR + memory[R1\_ADDR]] = 1;

// 0004 MOV\_MEM [prev + R1], MAX\_SIZE

Продолжение Листинга 1.4

memory[prev\_ADDR + memory[R1\_ADDR]] = memory[MAX\_SIZE\_ADDR];

// 0005 INCR R1

memory[R1\_ADDR] += 1;

// 0006 JMP 0001

goto init\_loop\_start;

init\_loop\_end:

// Fill dp and prev

// 0007 MOV\_REG R1, 1

memory[R1\_ADDR] = 1;

outer\_loop\_start:

// 0008 CMP\_REG R1, [N]

// 0009 JGE 0018

if(memory[R1\_ADDR] >= memory[N\_ADDR]) goto outer\_loop\_end;

// 0010 XOR R2

memory[R2\_ADDR] = 0;

inner\_loop\_start:

// 0011 CMP\_REG R2, R1

// 0012 JGE 0016

if(memory[R2\_ADDR] >= memory[R1\_ADDR]) goto inner\_loop\_end;

// 0013 CMP\_MEM [input + R2], [input + R1]

if(memory[input\_ADDR + memory[R2\_ADDR]] < memory[input\_ADDR + memory[R1\_ADDR]]) {

// 0015 CMP\_MEM [dp + R1], [dp + R2] + 1

if(memory[dp\_ADDR + memory[R1\_ADDR]] < memory[dp\_ADDR + memory[R2\_ADDR]] + 1) {

// 0017 MOV\_MEM [dp + R1], [dp + R2] + 1

memory[dp\_ADDR + memory[R1\_ADDR]] = memory[dp\_ADDR + memory[R2\_ADDR]] + 1;

// 0018 MOV\_MEM [prev + R1], R2

memory[prev\_ADDR + memory[R1\_ADDR]] = memory[R2\_ADDR];

}

}

// 0019 INCR R2

memory[R2\_ADDR] += 1;

// 0020 JMP 0011

goto inner\_loop\_start;

inner\_loop\_end:

// 0021 INCR R1

memory[R1\_ADDR] += 1;

// 0022 JMP 0008

goto outer\_loop\_start;

outer\_loop\_end:

Продолжение Листинга 1.4

// Find the index of the maximum element in dp

// 0023 XOR R1

memory[R1\_ADDR] = 0;

// 0024 XOR R2

memory[R2\_ADDR] = 0;

// 0025 MOV\_REG R3, MAX\_SIZE

memory[R3\_ADDR] = memory[MAX\_SIZE\_ADDR];

find\_max\_start:

// 0026 CMP\_REG R1, [N]

// 0027 JGE 0032

if(memory[R1\_ADDR] >= memory[N\_ADDR]) goto find\_max\_end;

// 0028 CMP\_MEM [dp + R1], R2

if(memory[dp\_ADDR + memory[R1\_ADDR]] > memory[R2\_ADDR]) {

// 0030 MOV\_MEM [R2], [dp + R1]

memory[R2\_ADDR] = memory[dp\_ADDR + memory[R1\_ADDR]];

// 0031 MOV\_MEM [R3], R1

memory[R3\_ADDR] = memory[R1\_ADDR];

}

// 0032 INCR R1

memory[R1\_ADDR] += 1;

// 0033 JMP 0026

goto find\_max\_start;

find\_max\_end:

// Restore LIS from dp and prev

restore\_lis\_start:

// 0034 CMP\_REG R3, [MAX\_SIZE]

// 0035 JEQ 0040

if(memory[R3\_ADDR] == memory[MAX\_SIZE\_ADDR]) goto restore\_lis\_end;

// 0036 MOV\_MEM [lis + lis\_length], [input + R3]

memory[lis\_ADDR + memory[lis\_length\_ADDR]] = memory[input\_ADDR + memory[R3\_ADDR]];

// 0037 INCR lis\_length

memory[lis\_length\_ADDR] += 1;

// 0038 MOV\_REG R3, [prev + R3]

memory[R3\_ADDR] = memory[prev\_ADDR + memory[R3\_ADDR]];

// 0039 JMP 0034

goto restore\_lis\_start;

restore\_lis\_end:

// Reverse the lis array manually

// 0040 XOR R1

memory[R1\_ADDR] = 0;

Продолжение Листинга 1.4

// 0041 SUB R2, lis\_length - 1

memory[R2\_ADDR] = memory[lis\_length\_ADDR] - 1;

reverse\_loop\_start:

// 0042 CMP\_REG R1, R2

// 0043 JGE 0050

if(memory[R1\_ADDR] >= memory[R2\_ADDR]) goto reverse\_loop\_end;

// 0044 MOV\_REG R3, [lis + R1]

memory[R3\_ADDR] = memory[lis\_ADDR + memory[R1\_ADDR]];

// 0045 MOV\_REG R4, [lis + R2]

memory[R4\_ADDR] = memory[lis\_ADDR + memory[R2\_ADDR]];

// 0046 MOV\_MEM [lis + R1], R4

memory[lis\_ADDR + memory[R1\_ADDR]] = memory[R4\_ADDR];

// 0047 MOV\_MEM [lis + R2], R3

memory[lis\_ADDR + memory[R2\_ADDR]] = memory[R3\_ADDR];

// 0048 INCR R1

memory[R1\_ADDR] += 1;

// 0049 DECR R2

memory[R2\_ADDR] -= 1;

// 0050 JMP 0042

goto reverse\_loop\_start;

reverse\_loop\_end:

// HALT

// 0051 HALT

// Вывод результата

// Выход из программы

cout << "Longest Increasing Subsequence: ";

for(int i = 0; i < memory[lis\_length\_ADDR]; ++i) {

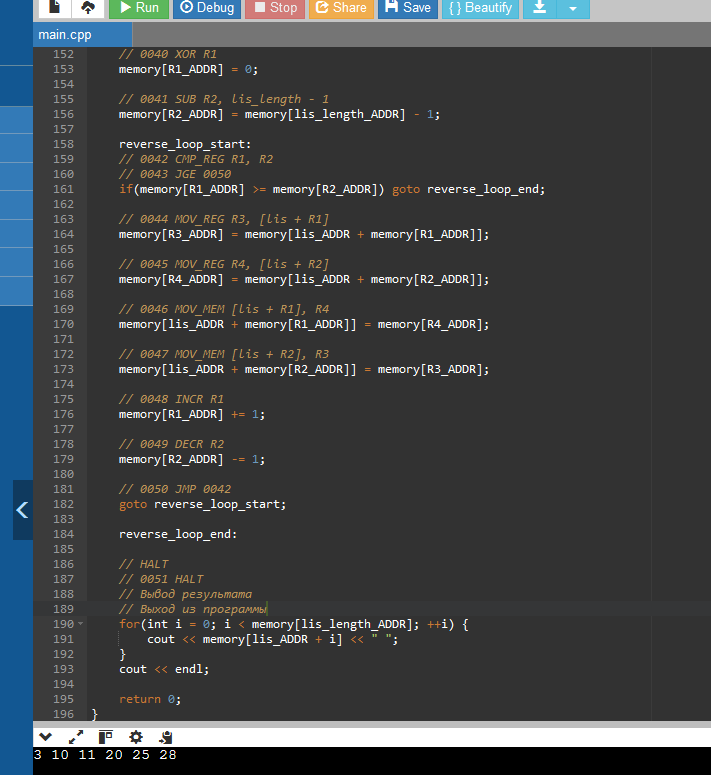
cout << memory[lis\_ADDR + i] << " ";

}

cout << endl;

return 0;

}

Рисунок 1.4 – результат работы низкоуровневого алгоритма версии 1

* 1. Низкоуровневый алгоритм версии 2

Продолжим работу над итоговый алгоритмом. Работа с памятью была слишком сложной для реализации её на языке Verilog. Упростим формат команд и работы с памятью.

Мы будем также эмулировать работу с памятью через общую шину, но введём другой набор инструкций (команд) для взаимодействия с регистрами и памятью.

В предыдущем алгоритме многие операции выполнялись неявно, например, при memory[dp\_ADDR + memory[R1\_ADDR]] = 1; происходило и обращение к памяти, и сложение.

В новом алгоритме операции разделены на более атомарные шаги. Например, сначала значение загружается в регистр (LOAD), затем с ним производятся арифметические операции, и только потом результат записывается обратно в память (MEM\_REG) (Листинг 1.5).

Несмотря на существенные изменения алгоритм продолжает работать аналогично предыдущим (Рисунок 1.5).

Листинг 1.5 – Низкоуровневый алгоритм версии 2 на языке C++

#include <iostream>

using namespace std;

const int MEMORY\_SIZE = 1000;

int memory[MEMORY\_SIZE];

// Memory addresses

const int MAX\_SIZE\_ADDR = 0;

const int dp\_ADDR = 1; // dp starts at 1

const int prev\_ADDR = 101; // prev starts at 101

const int lis\_ADDR = 201; // lis starts at 201

const int input\_ADDR = 301; // input starts at 301

const int N\_ADDR = 401;

const int lis\_length\_ADDR = 402;

const int R1\_ADDR = 403;

const int R2\_ADDR = 404;

const int R3\_ADDR = 405;

const int R4\_ADDR = 406;

const int result\_ADDR = 407;

int main() {

int R1, R2, R3, R4;

// Initialize constants

memory[MAX\_SIZE\_ADDR] = 100;

memory[N\_ADDR] = 10;

// Initialize input

int input\_data[] = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};

for(int i = 0; i < memory[N\_ADDR]; ++i) {

memory[input\_ADDR + i] = input\_data[i];

Продолжение Листинга 1.5

}

// Initialize dp and prev

memory[R1\_ADDR] = 0; // 0000 MEM\_XOR R1\_ADDR

init\_loop\_start:

R1 = memory[R1\_ADDR]; // 0001 LOAD R1, R1\_ADDR

R2 = memory[N\_ADDR]; // 0002 LOAD R2, N\_ADDR

if(R1 >= R2) // 0003 CMP R1, R2

goto init\_loop\_end; // 0004 JGE to 0011

R1 = R1 + dp\_ADDR; // 0005 ADD\_REG R1, dp\_ADDR

R1 = memory[R1\_ADDR]; // 0006 LOAD R1, R1\_ADDR

R1 = R1 + prev\_ADDR; // 0007 ADD\_REG R1, prev\_ADDR

memory[R1] = memory[MAX\_SIZE\_ADDR]; // 0008 MEM\_MEM R1, MAX\_SIZE\_ADDR

memory[R1\_ADDR] += 1; // 0009 INCR\_MEM R1\_ADDR

goto init\_loop\_start; // 0010 JMP 0001

init\_loop\_end:

// Fill dp and prev

memory[R1\_ADDR] = 1; // 0011 MEM\_ONE R1\_ADDR

outer\_loop\_start:

R1 = memory[R1\_ADDR]; // 0012 LOAD R1, R1\_ADDR

R2 = memory[N\_ADDR]; // 0013 LOAD R2, N\_ADDR

if(R1 >= R2) // 0014 CMP R1, R2

goto outer\_loop\_end; // 0015 JGE to 0051

memory[R2\_ADDR] = 0; // 0016 MEM\_XOR R2\_ADDR

inner\_loop\_start:

R1 = memory[R2\_ADDR]; // 0017 LOAD R1, R2\_ADDR

R2 = memory[R1\_ADDR]; // 0018 LOAD R2, R1\_ADDR

if(R1 >= R2) // 0019 CMP R1, R2

goto inner\_loop\_end; // 0020 JGE to 0049

R1 = memory[R2\_ADDR]; // 0021 LOAD R1, R2\_ADDR

R1= R1 + input\_ADDR; // 0022 ADD\_REG R1, input\_ADDR

R3 = memory[R1]; // 0023 REG\_MEM R3, R1

R2 = memory[R1\_ADDR]; // 0024 LOAD R2, R1\_ADDR

R2 = R2 + input\_ADDR; // 0025 ADD\_REG R2, input\_ADDR

R4 = memory[R2]; // 0026 REG\_MEM R4, R2

if(R3 >= R4) // 0027 CMP R3, R4

goto inner2; // 0028 JGE to 0047

R1 = memory[R1\_ADDR]; // 0029 LOAD R1, R1\_ADDR

R3 = R1 + dp\_ADDR; // 0030 ADD\_ANOTHER\_REG R3, R1, dp\_ADDR

R3 = memory[R3]; // 0031 REG\_MEM R3, R3

R2 = memory[R2\_ADDR]; // 0032 LOAD R2, R2\_ADDR

R4 = R2 + dp\_ADDR; // 0033 ADD\_ANOTHER\_REG R4, R2, dp\_ADDR

R4 = memory[R4]; // 0034 REG\_MEM R4, R4

R4 += 1; // 0035 INCR\_REG R4

if(R3 >= R4) // 0036 CMP R3, R4

goto inner2; // 0037 JGE to 0047

R2 = memory[R2\_ADDR]; // 0038 LOAD R2, R2\_ADDR

R3 = R2 + dp\_ADDR; // 0039 ADD\_ANOTHER\_REG R3, R2, dp\_ADDR

R3 = memory[R3]; // 0040 REG\_MEM R3, R3

R1 = memory[R1\_ADDR]; // 0041 LOAD R1, R1\_ADDR

R1 = R1 + dp\_ADDR; // 0042 ADD\_REG R1, dp\_ADDR

R3 += 1; // 0043 INCR\_REG R3

memory[R1] = R3; // 0044 MEM\_REG R1, R3

R1 = memory[R1\_ADDR]; // 0045 LOAD R1, R1\_ADDR

R1 = R1 + prev\_ADDR; // 0044 ADD\_REG R1, prev\_ADDR

Продолжение Листинга 1.5

memory[R1] = memory[R2\_ADDR]; // 0046 MEM\_MEM R1, R2\_ADDR

inner2:

memory[R2\_ADDR] += 1; // 0047 INCR\_MEM R2\_ADDR

goto inner\_loop\_start; // 0048 JMP 0017

inner\_loop\_end:

memory[R1\_ADDR] += 1; // 0049 INCR\_MEM R1\_ADDR

goto outer\_loop\_start; // 0050 JMP 0012

outer\_loop\_end:

// Find the index of the maximum element in dp

memory[R1\_ADDR] = 0; // 0051 MEM\_XOR R1\_ADDR

memory[R2\_ADDR] = 0; // 0052 MEM\_XOR R2\_ADDR

memory[R3\_ADDR] = memory[MAX\_SIZE\_ADDR]; // 0053 MEM\_MEM R3\_ADDR, MAX\_SIZE\_ADDR

find\_max\_start:

R1 = memory[R1\_ADDR]; // 0054 REG\_MEM R1, R1\_ADDR

R2 = memory[N\_ADDR]; // 0055 REG\_MEM R2, N\_ADDR

if(R1 >= R2) // 0056 CMP R1, R2

goto find\_max\_end; // 0057 JGE to 0068

R1 = memory[R1\_ADDR]; // 0058 REG\_MEM R1, R1\_ADDR

R1 = R1 + dp\_ADDR; // 0059 ADD\_REG R1, dp\_ADDR

R3 = memory[R1]; // 0060 REG\_MEM R3, R1

R2 = memory[R2\_ADDR]; // 0061 REG\_MEM R2, R2

if(R3 <= R2) // 0062 CMP R3, R2

goto jmp3; // 0063 JLE to 0066

memory[R2\_ADDR] = R3; // 0064 MEM\_REG R2\_ADDR, R3

memory[R3\_ADDR] = memory[R1\_ADDR]; // 0065 MEM\_MEM R3\_ADDR, R1\_ADDR

jmp3:

memory[R1\_ADDR] += 1; // 0066 INCR\_MEM R1\_ADDR

goto find\_max\_start; // 0067 JMP 0054

find\_max\_end:

restore\_lis\_start:

R1 = memory[R3\_ADDR]; // 0068 REG\_MEM R1, R3\_ADDR

R2 = memory[MAX\_SIZE\_ADDR]; // 0069 REG\_MEM R2, MAX\_SIZE\_ADDR

if(R1 == R2) // 0070 CMP R1, R2

goto restore\_lis\_end; // 0071 JEQ 0083

R1 = memory[lis\_length\_ADDR]; // 0072 REG\_MEM R1, lis\_length\_ADDR

R2 = R1 + lis\_ADDR; // 0073 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

R3 = memory[R3\_ADDR]; // 0074 REG\_MEM R3, R3

R3 = R3 + input\_ADDR; // 0075 ADD\_REG R3, input\_ADDR

R3 = memory[R3]; // 0076 REG\_MEM R3, R3

memory[R2] = R3; // 0077 MEM\_REG R2, R3

memory[lis\_length\_ADDR] += 1; // 0078 INCR\_MEM lis\_length\_ADDR

R1 = memory[R3\_ADDR]; // 0079 REG\_MEM R1, R3\_ADDR

R1 = R1 + prev\_ADDR; // 0080 ADD\_REG R1, prev\_ADDR

memory[R3\_ADDR] = memory[R1]; // 0081 MEM\_MEM R3\_ADDR, R1

goto restore\_lis\_start; // 0082 JMP 0068

restore\_lis\_end:

// Reverse the lis array manually

memory[R1\_ADDR] = 0; // 0083 MEM\_XOR R1\_ADDR

R1 = memory[lis\_length\_ADDR]; // 0084 REG\_MEM R1, lis\_length\_ADDR

R2 = 1; // 0085 REG\_ONE, R2

R1 = R1 - R2; // 0086 SUB R1, R2

memory[R2\_ADDR] = R1; // 0087 MEM\_REG R2\_ADDR, R1

reverse\_loop\_start:

Продолжение Листинга 1.5

R1 = memory[R1\_ADDR]; // 0088 REG\_MEM R1, R1\_ADDR

R2 = memory[R2\_ADDR]; // 0089 REG\_MEM R2, R2\_ADDR

if(R1 >= R2) // 0090 CMP R1, R2

goto reverse\_loop\_end; // 0091 JGE to 0107

R1 = memory[R1\_ADDR]; // 0092 REG\_MEM R1, R1\_ADDR

R1 = R1 + lis\_ADDR; // 0093 ADD\_REG R1, lis\_ADDR

R3 = memory[R1]; // 0094 REG\_MEM R3, R1

R2 = memory[R2\_ADDR]; // 0095 REG\_MEM R2, R2\_ADDR

R2 = R2 + lis\_ADDR; // 0096 ADD\_REG R2, lis\_ADDR

R4 = memory[R2]; // 0097 REG\_MEM R4, R2

R1 = memory[R1\_ADDR]; // 0098 REG\_MEM R1, R1\_ADDR

R2 = R1 + lis\_ADDR; // 0099 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

memory[R2] = R4; // 0100 MEM\_REG R2, R4

R2 = memory[R2\_ADDR]; // 0101 REG\_MEM R2, R2\_ADDR

R2 = R2 + lis\_ADDR; // 0102 ADD\_REG R2, lis\_ADDR

memory[R2] = R3; // 0103 MEM\_REG R2, R3

memory[R1\_ADDR] += 1; // 0104 INCR\_MEM lis\_length\_ADDR

memory[R2\_ADDR] -= 1; // 0105 DECR\_MEM lis\_length\_ADDR

goto reverse\_loop\_start; // 0106 JMP 0088

reverse\_loop\_end:

// 0107 HALT

// Вывод результата

// Выход из программы

cout << "Longest Increasing Subsequence: ";

for(int i = 0; i < memory[lis\_length\_ADDR]; ++i) {

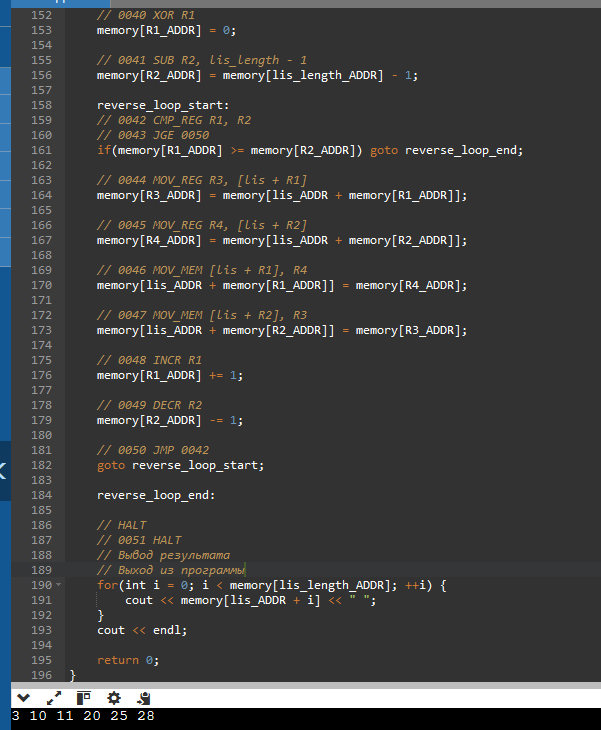
cout << memory[lis\_ADDR + i] << " ";

}

cout << endl;

return 0;

}

Рисунок 1.5 – результат работы низкоуровневого алгоритма версии 2

* 1. Конечный автомат реализующий заданный алгоритм

Реализуем конечный автомат на Verilog для заданного алгоритма, который поможет спроектировать итоговое процессорное ядро (Листинг 1.6).

* N размер входного массива arr;
* MAX\_SIZE – максимальный размер массивов dp, prev, lis, result;
* Arr – массив для хранения входной последовательности чисел. Инициализируется в блоке initial;
* Dp – массив для хранения длин НВП, заканчивающихся в каждом элементе arr;
* Prev – массив для хранения индексов предыдущих элементов в НВП;
* Lis – массив для хранения найденной НВП;
* Result – массив для вывода результата;
* Idx – индексные переменные для различных циклов;
* R5 – хранит максимальное значение, найденное в dp;
* R6 – хранит индекс элемента с максимальным значением в dp;
* lis\_length – хранит длину найденной НВП;
* temp – временная переменная для обмена значений при развороте массива lis.

Далее определим состояния:

* state – переменная, хранящая текущее состояние автомата;
* INIT\_LOOP – начальное состояние, инициализация dp и prev;
* OUTER\_LOOP – внешний цикл алгоритма;
* INNER\_LOOP – внутренний цикл алгоритма;
* FIND\_MAX – поиск максимального элемента в dp;
* RESTORE\_LIS – восстановление НВП по массивам dp и prev;
* REVERSE\_LIS – разворот массива lis;
* OUTPUT\_LIS – вывод результата в массив result;
* DONE\_STATE – конечное состояние.

Автомат проходит через все стадии и сохраняет заданную последовательность (Рисунки 1.6, 1.7).

Листинг 1.6 – Конечный автомат заданного алгоритма на языке Verilog

module LIS\_processor(

input clk,

input reset,

input start,

output reg done

Продолжение Листинга 1.6

);

parameter N = 10;

parameter MAX\_SIZE = 100;

reg [7:0] arr [0:9];

reg [7:0] dp [0:MAX\_SIZE-1];

reg signed [7:0] prev [0:MAX\_SIZE-1];

reg [7:0] lis [0:MAX\_SIZE-1];

reg [7:0] result [0:MAX\_SIZE-1];

reg signed [7:0] idx\_init;

reg signed [7:0] idx\_outer;

reg signed [7:0] idx\_inner;

reg signed [7:0] idx\_find\_max;

reg signed [7:0] R5;

reg signed [7:0] R6;

reg signed [7:0] lis\_length;

reg [7:0] temp;

reg signed [7:0] idx\_reverse\_start;

reg signed [7:0] idx\_reverse\_end;

reg signed [7:0] idx\_output;

reg [3:0] state;

parameter INIT\_LOOP = 0, OUTER\_LOOP = 1, INNER\_LOOP = 2, FIND\_MAX = 3,

RESTORE\_LIS = 4, REVERSE\_LIS = 5, OUTPUT\_LIS = 6, DONE\_STATE = 7;

initial begin

arr[0] = 3; arr[1] = 10; arr[2] = 2; arr[3] = 11;

arr[4] = 1; arr[5] = 20; arr[6] = 15; arr[7] = 30;

arr[8] = 25; arr[9] = 28;

end

always @(posedge clk or posedge reset) begin

if (reset) begin

state <= INIT\_LOOP;

idx\_init <= 0;

done <= 0;

idx\_outer <= 0;

idx\_inner <= 0;

idx\_find\_max <= 0;

R5 <= 0;

R6 <= -1;

lis\_length <= 0;

idx\_reverse\_start <= 0;

idx\_reverse\_end <= 0;

idx\_output <= 0;

for (i = 0; i < N; i = i + 1) begin

Продолжение Листинга 1.6

dp[i] <= 0;

prev[i] <= -1;

end

$display("RESET: All registers initialized.");

end else begin

case (state)

INIT\_LOOP: begin

if (idx\_init < N) begin

dp[idx\_init] <= 1;

prev[idx\_init] <= -1;

$display("INIT\_LOOP: dp[%0d] <= 1, prev[%0d] <= -1", idx\_init, idx\_init);

idx\_init <= idx\_init + 1;

end else begin

$display("INIT\_LOOP: Initialization complete. Moving to OUTER\_LOOP.");

idx\_outer <= 1;

state <= OUTER\_LOOP;

end

end

OUTER\_LOOP: begin

if (idx\_outer < N) begin

$display("OUTER\_LOOP: Processing index %0d.", idx\_outer);

idx\_inner <= 0;

state <= INNER\_LOOP;

end else begin

$display("OUTER\_LOOP: Completed outer loop. Moving to FIND\_MAX.");

idx\_find\_max <= 0;

R5 <= 0;

R6 <= -1;

state <= FIND\_MAX;

end

end

INNER\_LOOP: begin

if (idx\_inner < idx\_outer) begin

$display("INNER\_LOOP: Comparing arr[%0d]=%0d < arr[%0d]=%0d",

idx\_inner, arr[idx\_inner], idx\_outer, arr[idx\_outer]);

if (arr[idx\_inner] < arr[idx\_outer]) begin

$display("INNER\_LOOP: arr[%0d] < arr[%0d]", idx\_inner, idx\_outer);

if (dp[idx\_outer] < dp[idx\_inner] + 1) begin

$display("INNER\_LOOP: dp[%0d]=%0d < dp[%0d]=%0d + 1",

idx\_outer, dp[idx\_outer], idx\_inner, dp[idx\_inner]);

dp[idx\_outer] <= dp[idx\_inner] + 1;

prev[idx\_outer] <= idx\_inner;

$display("INNER\_LOOP: Updated dp[%0d] <= %0d, prev[%0d] <= %0d",

idx\_outer, dp[idx\_outer], idx\_outer, idx\_inner);

end else begin

Продолжение Листинга 1.6

$display("INNER\_LOOP: No update needed for dp[%0d].", idx\_outer);

end

end else begin

$display("INNER\_LOOP: arr[%0d] >= arr[%0d], no action.", idx\_inner, idx\_outer);

end

idx\_inner <= idx\_inner + 1;

end else begin

$display("INNER\_LOOP: Completed inner loop for index %0d. Moving to next OUTER\_LOOP.", idx\_outer);

idx\_outer <= idx\_outer + 1;

state <= OUTER\_LOOP;

end

end

FIND\_MAX: begin

if (idx\_find\_max < N) begin

$display("FIND\_MAX: Comparing dp[%0d]=%0d with current max dp=%0d at index=%0d",

idx\_find\_max, dp[idx\_find\_max], R5, R6);

if (dp[idx\_find\_max] > R5) begin

R5 <= dp[idx\_find\_max];

R6 <= idx\_find\_max;

$display("FIND\_MAX: New max dp found. R5 <= %0d, R6 <= %0d", R5, R6);

end

idx\_find\_max <= idx\_find\_max + 1;

end else begin

$display("FIND\_MAX: Completed finding max. Max dp=%0d at index=%0d", R5, R6);

lis\_length <= 0;

state <= RESTORE\_LIS;

end

end

RESTORE\_LIS: begin

if (R6 !== -8'sd1) begin // Корректное сравнение с -1

$display("RESTORE\_LIS: Adding arr[%0d]=%0d to LIS at lis[%0d]", R6, arr[R6], lis\_length);

lis[lis\_length] <= arr[R6];

lis\_length <= lis\_length + 1;

$display("RESTORE\_LIS: Updated lis\_length <= %0d", lis\_length);

$display("RESTORE\_LIS: Moving to previous index prev[%0d]=%0d", R6, prev[R6]);

R6 <= prev[R6];

end else begin

$display("RESTORE\_LIS: Completed restoring LIS. lis\_length=%0d", lis\_length);

// После восстановления, настройка для реверса

idx\_reverse\_start <= 0;

idx\_reverse\_end <= lis\_length - 1;

state <= REVERSE\_LIS;

end

end

REVERSE\_LIS: begin

Продолжение Листинга 1.6

if (idx\_reverse\_start < idx\_reverse\_end) begin

$display("REVERSE\_LIS: Swapping lis[%0d]=%0d <-> lis[%0d]=%0d",

idx\_reverse\_start, lis[idx\_reverse\_start], idx\_reverse\_end, lis[idx\_reverse\_end]);

// Используем блокирующие присваивания для корректного обмена

temp = lis[idx\_reverse\_start];

lis[idx\_reverse\_start] = lis[idx\_reverse\_end];

lis[idx\_reverse\_end] = temp;

$display("RESTORE\_LIS: Updated lis\_length <= %0d", lis\_length + 1);

$display("REVERSE\_LIS: After swap lis[%0d]=%0d, lis[%0d]=%0d",

idx\_reverse\_start, lis[idx\_reverse\_start], idx\_reverse\_end, lis[idx\_reverse\_end]);

idx\_reverse\_start <= idx\_reverse\_start + 1;

idx\_reverse\_end <= idx\_reverse\_end - 1;

end else begin

$display("REVERSE\_LIS: Completed reversing LIS. Moving to OUTPUT\_LIS.");

// После реверса, настройка для вывода

idx\_output <= 0;

state <= OUTPUT\_LIS;

end

end

OUTPUT\_LIS: begin

if (idx\_output < lis\_length) begin

result[idx\_output] <= lis[idx\_output];

$display("OUTPUT\_LIS: result[%0d] <= lis[%0d]=%0d",

idx\_output, idx\_output, lis[idx\_output]);

idx\_output <= idx\_output + 1;

end else begin

$display("OUTPUT\_LIS: Completed output. Final LIS:");

for (i = 0; i < lis\_length; i = i + 1) begin

$display("LIS[%0d] = %0d", i, lis[i]);

end

state <= DONE\_STATE;

done <= 1;

$display("DONE: LIS processing completed.");

end

end

DONE\_STATE: begin

$display("DONE: Waiting for reset.");

end

default: begin

state <= DONE\_STATE;

$display("DEFAULT: Undefined state. Moving to DONE.");

end

endcase

end end

endmodule

Напишем тестовый модуль (Листинг 1.7).

Листинг 1.7 – Тестовый модуль на языке Verilog

`timescale 1ns / 1ps

module tb\_LIS\_processor();

parameter N = 10;

parameter MAX\_SIZE = 100;

reg clk;

reg reset;

reg start;

wire done;

LIS\_processor #(

.N(N),

.MAX\_SIZE(MAX\_SIZE)

) DUT (

.clk(clk),

.reset(reset),

.start(start),

.done(done)

);

initial begin

clk = 0;

end

always #5 clk = ~clk;

initial begin

reset = 1;

start = 0;

#10;

reset = 0;

start = 1;

wait(done);

$display("LIS computation finished.");

printLIS();

#100;

$finish;

end

task printLIS;

integer i;

begin

$display("Longest Increasing Subsequence (LIS):");

for (i = 0; i < DUT.lis\_length; i = i + 1) begin

$write("%d ", DUT.result[i]);

end

$display("");

end

endtask

Продолжение Листинга 1.7

endmodule

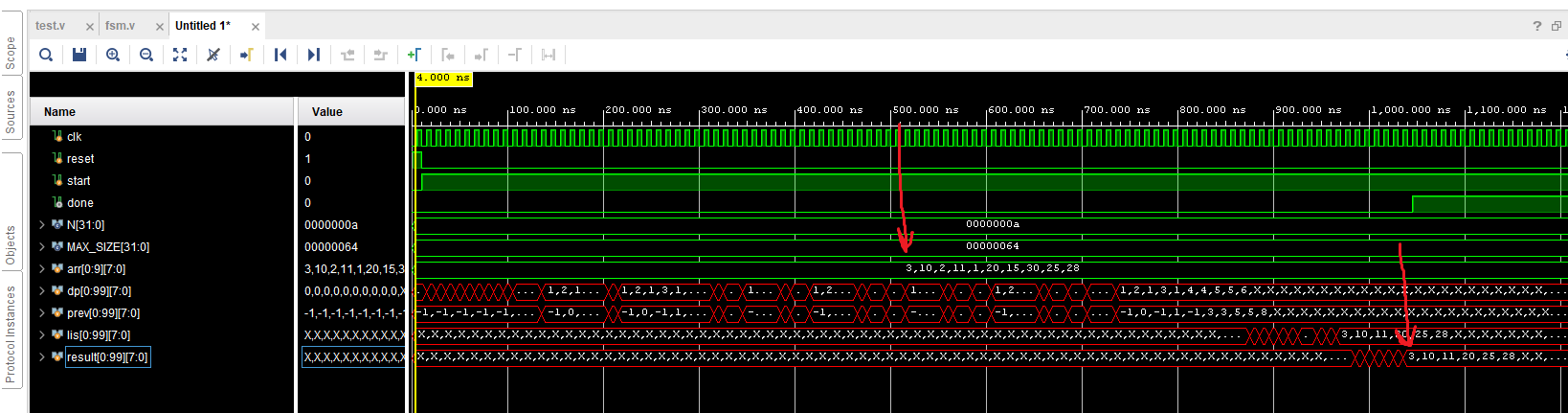
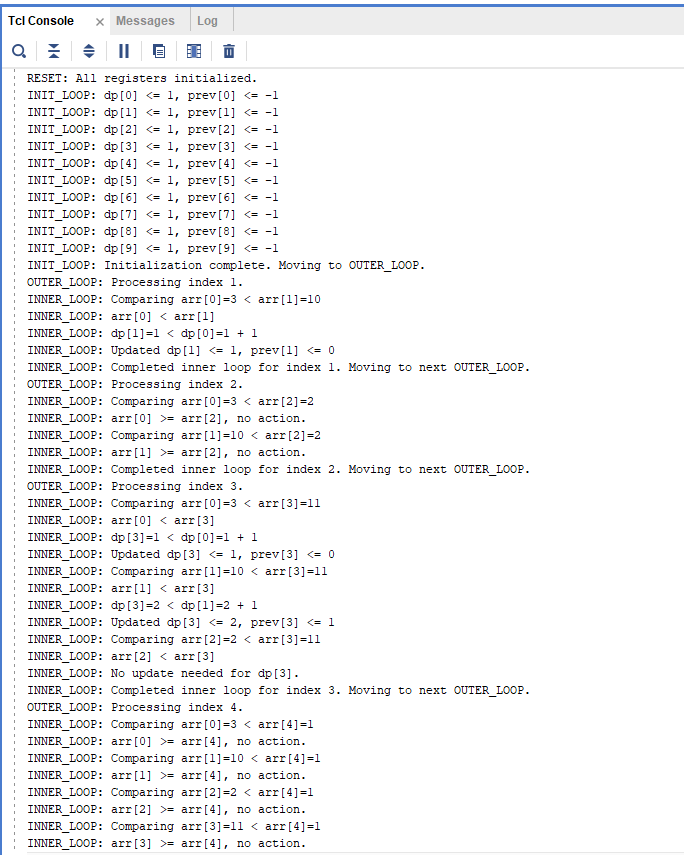
Рисунок 1.6 – результат работы конечного автомата на языке Verilog

Рисунок 1.7 – результат работы конечного автомата на языке Verilog

**1.7 Архитектура и микроархитектура последовательного процессорного ядра**

Определим архитектуру и микроархитектура последовательного процессорного ядра.

* Регистр общего назначения (РОН) – 32 регистра, каждый по 32 бита;
* Память инструкций используется для хранения программного кода;
* Память данных отдельный блок памяти для работы с данными;
* Программный счётчик (PC): используется для хранения адреса текущей команды.

Команды имеют фиксированный формат (32 бита):

* Код команды КОП (6 бит) — определяет тип операции;
* Регистры (5 бит для каждого регистра) — определяют операнды;
* Адреса (16 или 20 бит) — используются для адресации памяти.

Далее определим набор инструкций. Все команды имеют фиксированную длину 32 бита, разбитую на поля (Таблица 1.1).

Таблица 1.1 – Общий формат команд

|  |  |  |
| --- | --- | --- |
| Поле | Длина (в битах) | Описание |
| КОП | 6 |  |
| R1 | 5 | Номер первого регистра источника |
| R2/R3/ЗНАЧЕНИЕ | 5/16/20 | Номер второго регистра источника |
| RD | 5 | Номер регистра результата |
| Дополнительное поле | 11/16 | Поля для адресации |

Далее определим формат команды для каждой команды (Таблица 1.2).

Таблица 1.2 –Формат команд для каждой команды

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| КОП[31:26] | Команда | R1[25:21] | R2[20:16] | Значение[15:0] | Описание |
| 000000 | LOAD | Регистр 1 |  | Адрес ячейки | Загрузить значение из ячейки памяти Значение в регистр R1 |
| 000001 | STORE | Регистр 1 |  | Адрес ячейки | Записать значение из регистра R1 в ячейку памяти Значение |
| 000010 | ADD | Регистр 1 | Регистр 2 |  | Сложить значения регистров R1 и R2, результат записать в R1 |
| 000011 | SUB | Регистр 1 | Регистр 2 |  | Вычесть из значения регистра R1 значение регистра R2, результат записать в R1 |
| 000100 | ADD\_REG | Регистр 1 |  | Адрес ячейки | Сложить значение регистра R1 со значением из ячейки памяти Значение, результат записать в R1 |
| 000101 | ADD\_ANOTHER\_REG | Регистр 1 | Регистр 2 | Адрес/Регистр3 | Сложить значения регистров R1 и R2. Значение либо адрес либо значение регистра |
| 000110 | INCR\_REG | Регистр 1 |  |  | Увеличить значение регистра R1 на 1 |
| 000111 | DECR\_REG | Регистр 1 |  |  | Уменьшить значение регистра R1 на 1 |
| 001000 | INCR\_MEM |  |  | Адрес ячейки | Увеличить значение ячейки памяти Значение на 1 |
| 001001 | DECR\_MEM |  |  | Адрес ячейки | Уменьшить значение ячейки памяти Значение на 1 |
| 001010 | MEM\_XOR |  |  | Адрес ячейки | Обнулить значение в ячейке Значение. |
| 001011 | MEM\_ONE |  |  | Адрес ячейки | Записать 1 в ячейку памяти Значение. |

Продолжение Таблицы 1.2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 001100 | REG\_MEM | Регистр 1 |  | Адрес ячейки | Записать в регистр R1 значение из ячейки Значение. |
| 001101 | MEM\_REG |  | Регистр 1 | Адрес ячейки | Записать в ячейку памяти Значение значение из регистра R1. |
| 001110 | MEM\_MEM |  |  | Адрес1, Адрес2 | Записать в ячейку памяти Адрес1 значение из Адрес2. Адрес1 формируется из Значение[25:16], Адрес2 формируется из Значение[15:0] |
| 001111 | CMP | Регистр 1 | Регистр 2 |  | Сравнить значения регистров R1 и R2. Установить флаг условия, если R1 == R2 |
| 010000 | JGE |  |  | Адрес | Безусловный переход на инструкцию по адресу PC, если предыдущий флаг условия (установленный CMP) указывает, что R1 >= R2 |
| 010001 | JLE |  |  | Адрес | Безусловный переход на инструкцию по адресу PC, если предыдущий флаг условия (установленный CMP) указывает, что R1 <= R2 |
| 010010 | JEQ |  |  | Адрес | Безусловный переход на инструкцию по адресу PC, если предыдущий флаг условия (установленный CMP) указывает, что R1 == R2 |
| 010011 | JMP |  |  | Адрес | Безусловный переход на инструкцию по адресу Адрес |
| 111111 | HALT |  |  |  | Остановка процессора |

Разберем несколько примеров команд из кода программы, чтобы лучше понять, как работает процессор и как интерпретируются поля команд.

Команда memory[0] = {MEM\_XOR, R1, 5'd0, R1\_ADDR}; // 0000 MEM\_XOR R1\_ADDR

* КОП [31:26]: 001010 (MEM\_XOR);
* Команда: MEM\_XOR;
* R1 [25:21]: 00001 (R1);
* R2 [20:16]: 00000 (не используется);
* Значение [15:0]: 0000000001100100 (R1\_ADDR = 100 в десятичной системе);
* Описание: Обнулить значение в ячейке памяти с адресом R1\_ADDR (100).

Команда memory[1] = {LOAD, R1, 5'd0, R1\_ADDR}; // 0001 LOAD R1, R1\_ADDR

* КОП [31:26]: 000000 (LOAD);
* Команда: LOAD;
* R1 [25:21]: 00001 (R1);
* R2 [20:16]: 00000 (не используется);
* Значение [15:0]: 0000000001100100 (R1\_ADDR = 100);
* Описание: Загрузить значение из ячейки памяти с адресом R1\_ADDR (100) в регистр R1.

Команда memory[3] = {CMP, R1, R2, 16'd0}; // 0003 CMP R1, R2

* КОП [31:26]: 001111 (CMP);
* Команда: CMP;
* R1 [25:21]: 00001 (R1);
* R2 [20:16]: 00010 (R2);
* Значение [15:0]: 0000000000000000 (не используется);
* Описание: Сравнить значения регистров R1 и R2.

Команда memory[4] = {JGE, 6'd0, 20'd11}; // 0004 JGE to 0011

* КОП [31:26]: 010000 (JGE);
* Команда: JGE;
* R1 [25:21]: 00000 (не используется);
* R2 [20:16]: 00000 (не используется);
* Значение [15:0]: 0000000000001011 (Смещение = 11);
* Описание: Перейти на инструкцию по адресу PC + Смещение \* 4, если флаг условия, установленный предыдущей командой CMP, указывает, что R1 >= R2.

Команда memory[23] = {REG\_MEM, R3, 5'd0, R1}; // 0023 REG\_MEM R3, R1

* КОП [31:26]: 001100 (REG\_MEM);
* Команда: REG\_MEM;
* R1 [25:21]: 00011 (R3);
* R2 [20:16]: 00000 (не используется);
* Значение [15:0]: 0000000000000001 (Значение из R1);
* Описание: Записать в регистр R3 значение из ячейки, адрес которой находится в регистре R1.

Команда memory[30] = {ADD\_ANOTHER\_REG, R3, R1, dp\_ADDR}; // 0030 ADD\_ANOTHER\_REG R3, R1, dp\_ADDR

* КОП [31:26]: 000101 (ADD\_ANOTHER\_REG);
* Команда: ADD\_ANOTHER\_REG;
* R1 [25:21]: 00011 (R3);
* R2 [20:16]: 00001 (R1);
* Значение [15:0]: 0000000011001000 (dp\_ADDR = 200);
* Описание: Сложить значения регистров R3 и R1, результат записать в R3. Третий операнд dp\_ADDR используется как непосредственное значение или адрес.

Команда memory[99] = {ADD\_ANOTHER\_REG, R2, R1, lis\_ADDR}; // 0099 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

* КОП [31:26]: 000101 (ADD\_ANOTHER\_REG);
* Команда: ADD\_ANOTHER\_REG;
* R1 [25:21]: 00010 (R2);
* R2 [20:16]: 00001 (R1);
* Значение [15:0]: 0000000111110100 (lis\_ADDR = 500);
* Описание: Сложить значения регистров R2 и R1, результат записать в R2.

Команда memory[103] = {MEM\_REG, R2, R3, 16'd0}; // 0103 MEM\_REG R2, R3.

* КОП [31:26]: 001101 (MEM\_REG);
* Команда: MEM\_REG;
* R1 [25:21]: 00010 (R2);
* R2 [20:16]: 00011 (R3);
* Значение [15:0]: 0000000000000000;
* Описание: Записать в ячейку памяти, адрес которой хранится в регистре R2, значение из регистра R3.

**1.8 Описание программы в машинных кодах**

Программа для нахождения наибольшей возрастающей подпоследовательности (НВП) реализована в машинных кодах для специализированного процессорного ядра и состоит из нескольких основных этапов.

В начале происходит инициализация: обнуляются необходимые ячейки памяти, в том числе счетчики циклов, и заполняются начальными значениями массивы dp и prev. Массив dp хранит длины НВП, заканчивающихся в каждом элементе входного массива, а prev - индексы предыдущих элементов в этих НВП.

На данном этапе все элементы dp устанавливаются в 1 (так как каждый элемент сам по себе является возрастающей подпоследовательностью длины 1), а элементы prev заполняются значением MAX\_SIZE (100), указывающим на отсутствие предыдущего элемента в подпоследовательности.

Далее следует основной этап вычислений, где в двух вложенных циклах for происходит заполнение массивов dp и prev. Внешний цикл перебирает элементы входного массива input от 1 до N-1, а внутренний цикл - от 0 до индекса текущего элемента внешнего цикла.

Внутри этих циклов происходит сравнение текущего элемента input[R1] с предыдущими элементами input[R2]. Если input[R2] < input[R1] и при этом длина НВП, заканчивающейся в input[R1] (dp[R1]), меньше, чем длина НВП, заканчивающейся в input[R2] (dp[R2]) плюс 1, то обновляются значения dp[R1] и prev[R1].

Таким образом, для каждого элемента входного массива находится оптимальная длина НВП, заканчивающейся в нем, и запоминается индекс предыдущего элемента в этой подпоследовательности.

После заполнения dp и prev программа переходит к этапу поиска индекса максимального элемента в dp. Для этого инициализируются специальные регистры: R1 обнуляется (используется как счетчик), R2 обнуляется (хранит максимальное значение dp), R3 устанавливается в MAX\_SIZE (хранит индекс элемента с максимальным значением dp). Затем в цикле сравниваются элементы dp с текущим максимумом в R2. Если очередной элемент dp[R1] больше R2, то R2 обновляется этим значением, а R3 - индексом R1.

Следующий этап – восстановление НВП по массиву prev, начиная с найденного индекса максимального элемента в dp (хранится в R3). В цикле происходит движение по массиву prev в обратном направлении: значения из input, соответствующие индексам, хранящимся в prev, добавляются в массив lis. Цикл завершается, когда встречается значение MAX\_SIZE в prev. После этого восстановленная НВП (хранящаяся в lis) разворачивается, так как она была получена в обратном порядке.

Результатом выполнения программы является массив lis, содержащий наибольшую возрастающую подпоследовательность, и его длина, хранящаяся в переменной lis\_length. Данная реализация демонстрирует эффективное использование специализированного процессорного ядра для решения задачи нахождения НВП с применением алгоритма динамического программирования.

Далее рассмотрим код модуля.

В начале задаются константы для кодов операций (опкодов) процессора, таких как LOAD, STORE, ADD, SUB, JMP, HALT и другие. Каждый опкод имеет уникальный 6-битный код.

Определяются адреса регистров общего назначения (R1-R4) и адреса специальных ячеек памяти, используемых в алгоритме (например, R1\_ADDR, N\_ADDR, dp\_ADDR, input\_ADDR и т.д.).

* registers [0:31] массив из 32 32-битных регистров общего назначения;
* memory [0:150] память команд, хранящая 151 32-битную инструкцию;
* data\_memory [0:600] память данных, хранящая данные, используемые программой;
* PC счетчик команд, указывающий на текущую выполняемую инструкцию.

Код программы и процессора представлен ниже (Листинг 1.8).

Листинг 1.8 – Основной модуль последовательного процессорного ядра на языке Verilog

module processor(

input clk,

input reset,

output reg done

);

parameter LOAD = 6'b000000;

parameter STORE = 6'b000001;

parameter ADD = 6'b000010;

parameter SUB = 6'b000011;

parameter ADD\_REG = 6'b000100;

parameter ADD\_ANOTHER\_REG = 6'b000101;

parameter INCR\_REG = 6'b000110;

parameter DECR\_REG = 6'b000111;

parameter INCR\_MEM = 6'b001000;

parameter DECR\_MEM = 6'b001001;

parameter MEM\_XOR = 6'b001010;

parameter MEM\_ONE = 6'b001011;

parameter REG\_MEM = 6'b001100;

parameter MEM\_REG = 6'b001101;

parameter MEM\_MEM = 6'b001110;

parameter CMP = 6'b001111;

parameter JGE = 6'b010000;

parameter JLE = 6'b010001;

parameter JEQ = 6'b010010;

parameter JMP = 6'b010011;

parameter HALT = 6'b111111;

parameter R1 = 5'd1;

parameter R2 = 5'd2;

parameter R3 = 5'd3;

parameter R4 = 5'd4;

parameter R1\_ADDR = 16'd100;

parameter R2\_ADDR = 16'd104;

parameter R3\_ADDR = 16'd108;

parameter R4\_ADDR = 16'd112;

parameter N\_ADDR = 16'd116;

parameter MAX\_SIZE\_ADDR = 16'd120;

parameter dp\_ADDR = 16'd200;

parameter prev\_ADDR = 16'd300;

parameter input\_ADDR = 16'd400;

parameter lis\_ADDR = 16'd500;

parameter lis\_length\_ADDR = 16'd600;

reg [31:0] registers [0:31];

reg [31:0] memory [0:150];

reg [31:0] data\_memory [0:600];

reg [31:0] PC;

reg [31:0] IF\_ID\_IR, IF\_ID\_NPC;

reg [31:0] ID\_EX\_A, ID\_EX\_B, ID\_EX\_NPC, ID\_EX\_IR;

reg [4:0] ID\_EX\_rd;

Продолжение Листинга 1.8

reg ID\_EX\_cond;

reg [31:0] EX\_MEM\_ALUOut, EX\_MEM\_B, EX\_MEM\_IR;

reg [4:0] EX\_MEM\_rd;

reg EX\_MEM\_cond;

reg [31:0] MEM\_WB\_LMD, MEM\_WB\_ALUOut, MEM\_WB\_IR;

reg [4:0] MEM\_WB\_rd;

initial begin

PC = 0;

done = 1'b0;

memory[0] = {MEM\_XOR, R1, 5'd0, R1\_ADDR}; // 0000 MEM\_XOR R1\_ADDR

memory[1] = {LOAD, R1, 5'd0, R1\_ADDR}; // 0001 LOAD R1, R1\_ADDR

memory[2] = {LOAD, R2, 5'd0, N\_ADDR}; // 0002 LOAD R2, N\_ADDR

memory[3] = {CMP, R1, R2, 16'd0}; // 0003 CMP R1, R2

memory[4] = {JGE, 6'd0, 20'd11}; // 0004 JGE to 0011

memory[5] = {ADD\_REG, R1, 5'd0, dp\_ADDR}; // 0005 ADD\_REG R1, dp\_ADDR

memory[6] = {LOAD, R1, 5'd0, R1\_ADDR}; // 0006 LOAD R1, R1\_ADDR

memory[7] = {ADD\_REG, R1, 5'd0, prev\_ADDR}; // 0007 ADD\_REG R1, prev\_ADDR

memory[8] = {MEM\_MEM, R1, 5'd0, MAX\_SIZE\_ADDR}; // 0008 MEM\_MEM R1, MAX\_SIZE\_ADDR

memory[9] = {INCR\_MEM, 5'd0, 5'd0, R1\_ADDR}; // 0009 INCR\_MEM R1\_ADDR

memory[10] = {JMP, 6'd0, 20'd1}; // 0010 JMP to 0001

// 0011 MEM\_ONE R1\_ADDR

memory[11] = {MEM\_ONE, R1, 5'd0, R1\_ADDR};

// 0012 LOAD R1, R1\_ADDR

memory[12] = {LOAD, R1, 5'd0, R1\_ADDR};

// 0013 LOAD R2, N\_ADDR

memory[13] = {LOAD, R2, 5'd0, N\_ADDR};

// 0014 CMP R1, R2

memory[14] = {CMP, R1, R2, 16'd0};

// 0015 JGE to 0051

memory[15] = {JGE, 6'd0, 20'd51};

// 0016 MEM\_XOR R2\_ADDR

memory[16] = {MEM\_XOR, R2, 5'd0, R2\_ADDR};

// 0017 LOAD R1, R2\_ADDR

memory[17] = {LOAD, R1, 5'd0, R2\_ADDR};

// 0018 LOAD R2, R1\_ADDR

Продолжение Листинга 1.8

memory[18] = {LOAD, R2, 5'd0, R1\_ADDR};

// 0019 CMP R1, R2

memory[19] = {CMP, R1, R2, 16'd0};

// 0020 JGE to 0049

memory[20] = {JGE, 6'd0, 20'd49};

// 0021 LOAD R1, R2\_ADDR

memory[21] = {LOAD, R1, 5'd0, R2\_ADDR};

// 0022 ADD\_REG R1, input\_ADDR

memory[22] = {ADD\_REG, R1, 5'd0, input\_ADDR};

// 0023 REG\_MEM R3, R1

memory[23] = {REG\_MEM, R3, 5'd0, R1};

// 0024 LOAD R2, R1\_ADDR

memory[24] = {LOAD, R2, 5'd0, R1\_ADDR};

// 0025 ADD\_REG R2, input\_ADDR

memory[25] = {ADD\_REG, R2, 5'd0, input\_ADDR};

// 0026 REG\_MEM R4, R2

memory[26] = {REG\_MEM, R4, 5'd0, R2};

// 0027 CMP R3, R4

memory[27] = {CMP, R3, R4, 16'd0};

// 0028 JGE to 0047

memory[28] = {JGE, 6'd0, 20'd47};

// 0029 LOAD R1, R1\_ADDR

memory[29] = {LOAD, R1, 5'd0, R1\_ADDR};

// 0030 ADD\_ANOTHER\_REG R3, R1, dp\_ADDR

memory[30] = {ADD\_ANOTHER\_REG, R3, R1, dp\_ADDR};

// 0031 REG\_MEM R3, R3

memory[31] = {REG\_MEM, R3, 5'd0, R3};

// 0032 LOAD R2, R2\_ADDR

memory[32] = {LOAD, R2, 5'd0, R2\_ADDR};

// 0033 ADD\_ANOTHER\_REG R4, R2, dp\_ADDR

memory[33] = {ADD\_ANOTHER\_REG, R4, R2, dp\_ADDR};

// 0034 REG\_MEM R4, R4

memory[34] = {REG\_MEM, R4, 5'd0, R4};

// 0035 INCR\_REG R4

memory[35] = {INCR\_REG, R4, 5'd0, 16'd0};

// 0036 CMP R3, R4

memory[36] = {CMP, R3, R4, 16'd0};

Продолжение Листинга 1.8

// 0037 JGE to 0047

memory[37] = {JGE, 6'd0, 20'd47};

// 0038 LOAD R2, R2\_ADDR

memory[38] = {LOAD, R2, 5'd0, R2\_ADDR};

// 0039 ADD\_ANOTHER\_REG R3, R2, dp\_ADDR

memory[39] = {ADD\_ANOTHER\_REG, R3, R2, dp\_ADDR};

// 0040 REG\_MEM R3, R3

memory[40] = {REG\_MEM, R3, 5'd0, R3};

// 0041 LOAD R1, R1\_ADDR

memory[41] = {LOAD, R1, 5'd0, R1\_ADDR};

// 0042 ADD\_REG R1, dp\_ADDR

memory[42] = {ADD\_REG, R1, 5'd0, dp\_ADDR};

// 0043 INCR\_REG R3

memory[43] = {INCR\_REG, R3, 5'd0, 16'd0};

// 0044 MEM\_REG R1, R3

memory[44] = {MEM\_REG, R1, R3, 16'd0};

// 0045 LOAD R1, R1\_ADDR

memory[45] = {LOAD, R1, 5'd0, R1\_ADDR};

// 0046 ADD\_REG R1, prev\_ADDR

memory[46] = {ADD\_REG, R1, 5'd0, prev\_ADDR};

// 0047 MEM\_MEM R3, R2\_ADDR

memory[47] = {MEM\_MEM, R3, R2\_ADDR, 16'd0};

// 0048 JMP to inner\_loop\_start (0017)

memory[48] = {JMP, 6'd0, 20'd17};

// 0049 INCR\_MEM R1\_ADDR

memory[49] = {INCR\_MEM,5'd0,5'd0, R1\_ADDR};

// 0050 JMP to outer\_loop\_start (0012)

memory[50] = {JMP, 6'd0, 20'd12};

// 0051 MEM\_XOR R1\_ADDR

memory[51] = {MEM\_XOR, R1, 5'd0, R1\_ADDR};

// 0052 MEM\_XOR R2\_ADDR

memory[52] = {MEM\_XOR, R2, 5'd0, R2\_ADDR};

// 0053 MEM\_MEM R3\_ADDR, MAX\_SIZE\_ADDR

memory[53] = {MEM\_MEM, R3, MAX\_SIZE\_ADDR, 16'd0};

// 0054 REG\_MEM R1, R1\_ADDR

memory[54] = {REG\_MEM, R1, 5'd0, R1\_ADDR};

// 0055 REG\_MEM R2, N\_ADDR

memory[55] = {REG\_MEM, R2, 5'd0, N\_ADDR};

Продолжение Листинга 1.8

// 0056 CMP R1, R2

memory[56] = {CMP, R1, R2, 16'd0};

// 0057 JGE to 0068

memory[57] = {JGE, 6'd0, 20'd68};

// 0058 REG\_MEM R1, R1\_ADDR

memory[58] = {REG\_MEM, R1, 5'd0, R1\_ADDR};

// 0059 ADD\_REG R1, dp\_ADDR

memory[59] = {ADD\_REG, R1, 5'd0, dp\_ADDR};

// 0060 REG\_MEM R3, R1

memory[60] = {REG\_MEM, R3, 5'd0, R1};

// 0061 REG\_MEM R2, R2\_ADDR

memory[61] = {REG\_MEM, R2, 5'd0, R2\_ADDR};

// 0062 CMP R3, R2

memory[62] = {CMP, R3, R2, 16'd0};

// 0063 JLE to 0066

memory[63] = {JLE, 6'd0, 20'd66};

// 0064 MEM\_REG R2\_ADDR, R3

memory[64] = {MEM\_REG, R2\_ADDR, R3, 16'd0};

// 0065 MEM\_MEM R3\_ADDR, R1\_ADDR

memory[65] = {MEM\_MEM, R3\_ADDR, R1\_ADDR, 16'd0};

// 0066 INCR\_MEM R1\_ADDR

memory[66] = {INCR\_MEM,5'd0,5'd0, R1\_ADDR};

// 0067 JMP to 0054

memory[67] = {JMP, 6'd0, 20'd54};

// 0068 REG\_MEM R1, R3\_ADDR

memory[68] = {REG\_MEM, R1, 5'd0, R3\_ADDR};

// 0069 REG\_MEM R2, MAX\_SIZE\_ADDR

memory[69] = {REG\_MEM, R2, 5'd0, MAX\_SIZE\_ADDR};

// 0070 CMP R1, R2

memory[70] = {CMP, R1, R2, 16'd0};

// 0071 JEQ to 0083

memory[71] = {JEQ, 6'd0, 20'd83};

// 0072 REG\_MEM R1, lis\_length\_ADDR

memory[72] = {REG\_MEM, R1, 5'd0, lis\_length\_ADDR};

// 0073 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

memory[73] = {ADD\_ANOTHER\_REG, R2, R1, lis\_ADDR};

// 0074 REG\_MEM R3, R3\_ADDR

Продолжение Листинга 1.8

memory[74] = {REG\_MEM, R3, 5'd0, R3\_ADDR};

// 0075 ADD\_REG R3, input\_ADDR

memory[75] = {ADD\_REG, R3, 5'd0, input\_ADDR};

// 0076 REG\_MEM R3, R3

memory[76] = {REG\_MEM, R3, 5'd0, R3};

// 0077 MEM\_REG R2, R3

memory[77] = {MEM\_REG, R2, R3, 16'd0};

// 0078 INCR\_MEM lis\_length\_ADDR

memory[78] = {INCR\_MEM,5'd0,5'd0, lis\_length\_ADDR};

// 0079 REG\_MEM R1, R3\_ADDR

memory[79] = {REG\_MEM, R1, 5'd0, R3\_ADDR};

// 0080 ADD\_REG R1, prev\_ADDR

memory[80] = {ADD\_REG, R1, 5'd0, prev\_ADDR};

// 0081 MEM\_MEM R3\_ADDR, R1

memory[81] = {MEM\_MEM, R3\_ADDR, R1, 16'd0};

// 0082 JMP to 0068

memory[82] = {JMP, 6'd0, 20'd68};

// 0083 MEM\_XOR R1\_ADDR

memory[83] = {MEM\_XOR, R1, 5'd0, R1\_ADDR};

// 0084 REG\_MEM R1, lis\_length\_ADDR

memory[84] = {REG\_MEM, R1, 5'd0, lis\_length\_ADDR};

// 0085 REG\_ONE R2

// Предполагается, что REG\_ONE устанавливает R2 в 1

// Поскольку REG\_ONE не был определен ранее, используем ADD\_REG для этого

memory[85] = {ADD\_REG, R2, 5'd0, 16'd1}; // 0085 REG\_ONE R2

// 0086 SUB R1, R2

memory[86] = {SUB, R1, R2, 16'd0}; // 0086 SUB R1, R2

// 0087 MEM\_REG R2\_ADDR, R1

memory[87] = {MEM\_REG, R2\_ADDR, R1, 16'd0}; // 0087 MEM\_REG R2\_ADDR, R1

// 0088 REG\_MEM R1, lis\_length\_ADDR

memory[88] = {REG\_MEM, R1, 5'd0, lis\_length\_ADDR}; // 0088 REG\_MEM R1, lis\_length\_ADDR

// 0089 REG\_MEM R2, R2\_ADDR

memory[89] = {REG\_MEM, R2, 5'd0, R2\_ADDR}; // 0089 REG\_MEM R2, R2\_ADDR

// 0090 CMP R1, R2

Продолжение Листинга 1.8

memory[90] = {CMP, R1, R2, 16'd0}; // 0090 CMP R1, R2

// 0091 JGE to 0107

memory[91] = {JGE, 6'd0, 20'd107}; // 0091 JGE to 0107

// 0092 REG\_MEM R1, lis\_length\_ADDR

memory[92] = {REG\_MEM, R1, 5'd0, lis\_length\_ADDR}; // 0092 REG\_MEM R1, lis\_length\_ADDR

// 0093 ADD\_REG R1, lis\_ADDR

memory[93] = {ADD\_REG, R1, 5'd0, lis\_ADDR}; // 0093 ADD\_REG R1, lis\_ADDR

// 0094 REG\_MEM R3, R1

memory[94] = {REG\_MEM, R3, 5'd0, R1}; // 0094 REG\_MEM R3, R1

// 0095 REG\_MEM R2, R2\_ADDR

memory[95] = {REG\_MEM, R2, 5'd0, R2\_ADDR}; // 0095 REG\_MEM R2, R2\_ADDR

// 0096 ADD\_REG R2, lis\_ADDR

memory[96] = {ADD\_REG, R2, 5'd0, lis\_ADDR}; // 0096 ADD\_REG R2, lis\_ADDR

// 0097 REG\_MEM R4, R2

memory[97] = {REG\_MEM, R4, 5'd0, R2}; // 0097 REG\_MEM R4, R2

// 0098 REG\_MEM R1, R1\_ADDR

memory[98] = {REG\_MEM, R1, 5'd0, R1\_ADDR}; // 0098 REG\_MEM R1, R1\_ADDR

// 0099 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

memory[99] = {ADD\_ANOTHER\_REG, R2, R1, lis\_ADDR}; // 0099 ADD\_ANOTHER\_REG R2, R1, lis\_ADDR

// 0100 MEM\_REG R2, R4

memory[100] = {MEM\_REG, R2, R4, 16'd0}; // 0100 MEM\_REG R2, R4

// 0101 REG\_MEM R2, R2\_ADDR

memory[101] = {REG\_MEM, R2, 5'd0, R2\_ADDR}; // 0101 REG\_MEM R2, R2\_ADDR

// 0102 ADD\_REG R2, lis\_ADDR

memory[102] = {ADD\_REG, R2, 5'd0, lis\_ADDR}; // 0102 ADD\_REG R2, lis\_ADDR

// 0103 MEM\_REG R2, R3

memory[103] = {MEM\_REG, R2, R3, 16'd0}; // 0103 MEM\_REG R2, R3

// 0104 INCR\_MEM lis\_length\_ADDR

Продолжение Листинга 1.8

memory[104] = {INCR\_MEM,5'd0,5'd0, lis\_length\_ADDR}; // 0104 INCR\_MEM lis\_length\_ADDR

// 0105 DECR\_MEM lis\_length\_ADDR

memory[105] = {DECR\_MEM,5'd0,5'd0, lis\_length\_ADDR}; // 0105 DECR\_MEM lis\_length\_ADDR

// 0106 JMP to 0088

memory[106] = {JMP, 6'd0, 20'd88}; // 0106 JMP to 0088

// 0107 HALT

memory[107] = {HALT, 26'd0}; // 0107 HALT

memory[107] = {HALT, 26'd0}; // 0107 HALT

// Data memory

data\_memory[MAX\_SIZE\_ADDR] = 32'd100;

data\_memory[N\_ADDR] = 32'd10;

// Input data

data\_memory[input\_ADDR + 0] = 32'd3;

data\_memory[input\_ADDR + 1] = 32'd10;

data\_memory[input\_ADDR + 2] = 32'd2;

data\_memory[input\_ADDR + 3] = 32'd11;

data\_memory[input\_ADDR + 4] = 32'd1;

data\_memory[input\_ADDR + 5] = 32'd20;

data\_memory[input\_ADDR + 6] = 32'd15;

data\_memory[input\_ADDR + 7] = 32'd30;

data\_memory[input\_ADDR + 8] = 32'd25;

data\_memory[input\_ADDR + 9] = 32'd28;

end

// Fetch stage

always @(posedge clk or posedge reset) begin

if (reset) begin

PC <= 0;

IF\_ID\_IR <= 0;

IF\_ID\_NPC <= 0;

end else begin

IF\_ID\_IR <= memory[PC];

IF\_ID\_NPC <= PC + 1;

PC <= PC + 1;

end

$display("PC: %d, opcode: %b", PC, IF\_ID\_IR[31:26]);

end

// Decode stage

always @(posedge clk) begin

ID\_EX\_A <= registers[IF\_ID\_IR[25:21]];

ID\_EX\_B <= registers[IF\_ID\_IR[20:16]];

ID\_EX\_IR <= IF\_ID\_IR;

ID\_EX\_NPC <= IF\_ID\_NPC;

ID\_EX\_rd <= IF\_ID\_IR[15:11];

end

Продолжение Листинга 1.8

// Стадия Execute

always @(posedge clk) begin

case (ID\_EX\_IR[31:26]) // opcode

6'b000000: begin // LOAD R1, ADDRESS

EX\_MEM\_ALUOut <= data\_memory[ID\_EX\_IR[15:0]];

end

6'b000001: begin // STORE R1, ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= ID\_EX\_A;

end

6'b000010: begin // ADD R1, R2

EX\_MEM\_ALUOut <= registers[ID\_EX\_IR[25:21]] + registers[ID\_EX\_IR[20:16]];

end

6'b000011: begin // SUB R1, R2

EX\_MEM\_ALUOut <= ID\_EX\_A - ID\_EX\_B;

end

6'b000100: begin // ADD\_REG R1, ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_A + data\_memory[ID\_EX\_IR[15:0]];

end

6'b000101: begin // ADD\_ANOTHER\_REG R1, R2, R3

EX\_MEM\_ALUOut <= registers[ID\_EX\_IR[25:21]] + registers[ID\_EX\_IR[20:16]];

end

6'b000110: begin // INCR\_REG R1

EX\_MEM\_ALUOut <= ID\_EX\_A + 1;

end

6'b000111: begin // DECR\_REG R1

EX\_MEM\_ALUOut <= ID\_EX\_A - 1;

end

6'b001000: begin // INCR\_MEM ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= data\_memory[ID\_EX\_IR[15:0]] + 1;

end

6'b001001: begin // DECR\_MEM ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= data\_memory[ID\_EX\_IR[15:0]] - 1;

end

6'b001010: begin // MEM\_XOR ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= 0;

end

6'b001011: begin // MEM\_ONE ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= 1;

end

6'b001100: begin // REG\_MEM R1, ADDRESS

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

end

6'b001101: begin // MEM\_REG ADDRESS, R1

EX\_MEM\_ALUOut <= ID\_EX\_IR[15:0];

EX\_MEM\_B <= ID\_EX\_A;

end

6'b001110: begin // MEM\_MEM ADDRESS1, ADDRESS2

EX\_MEM\_ALUOut <= ID\_EX\_IR[25:16]; // DEST

Продолжение Листинга 1.8

EX\_MEM\_B <= data\_memory[ID\_EX\_IR[15:0]]; // SRC

end

6'b001111: begin // CMP R1, R2

EX\_MEM\_cond <= (ID\_EX\_A == ID\_EX\_B);

end

// Условия перехода

6'b010000: begin // JGE LABEL

if (ID\_EX\_A >= ID\_EX\_B)

PC <= ID\_EX\_NPC + ({{16{ID\_EX\_IR[15]}}, ID\_EX\_IR[15:0]} << 2);

end

6'b010001: begin // JLE LABEL

if (ID\_EX\_A <= ID\_EX\_B)

PC <= ID\_EX\_NPC + ({{16{ID\_EX\_IR[15]}}, ID\_EX\_IR[15:0]} << 2);

end

6'b010010: begin // JEQ LABEL

if (ID\_EX\_A == ID\_EX\_B)

PC <= ID\_EX\_NPC + ({{16{ID\_EX\_IR[15]}}, ID\_EX\_IR[15:0]} << 2);

end

6'b010011: begin // JMP LABEL

PC <= ID\_EX\_IR[25:0] << 2;

end

6'b111111: begin

done <= 1'b1;

end

default: begin

EX\_MEM\_ALUOut <= 0;

end

endcase

EX\_MEM\_IR <= ID\_EX\_IR;

EX\_MEM\_rd <= ID\_EX\_rd;

end

// Стадия Memory

always @(posedge clk) begin

MEM\_WB\_IR <= EX\_MEM\_IR;

MEM\_WB\_rd <= EX\_MEM\_rd;

case (EX\_MEM\_IR[31:26])

6'b000000: begin // LOAD

MEM\_WB\_LMD <= data\_memory[EX\_MEM\_ALUOut];

end

6'b000001: begin // STORE

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001000: begin // INCR\_MEM

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001001: begin // DECR\_MEM

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001010: begin // MEM\_XOR

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001011: begin // MEM\_ONE

Продолжение Листинга 1.8

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001101: begin // MEM\_REG

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

6'b001110: begin // MEM\_MEM

data\_memory[EX\_MEM\_ALUOut] <= EX\_MEM\_B;

end

default: begin

MEM\_WB\_ALUOut <= EX\_MEM\_ALUOut;

end

endcase

end

// Стадия Write Back

always @(posedge clk) begin

case (MEM\_WB\_IR[31:26])

6'b000000: begin // LOAD

registers[MEM\_WB\_rd] <= MEM\_WB\_LMD;

end

6'b000010: begin // ADD

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b000011: begin // SUB

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b000100: begin // ADD\_REG

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b000101: begin // ADD\_ANOTHER\_REG

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b000110: begin // INCR\_REG

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b000111: begin // DECR\_REG

registers[MEM\_WB\_rd] <= MEM\_WB\_ALUOut;

end

6'b001100: begin // REG\_MEM

registers[MEM\_WB\_rd] <= data\_memory[MEM\_WB\_ALUOut];

end

MEM\_REG: begin // MEM\_REG: записываем данные из памяти в регистр

registers[MEM\_WB\_rd] <= data\_memory[EX\_MEM\_ALUOut];

end

6'b001111: begin // CMP

registers[MEM\_WB\_rd] <= EX\_MEM\_cond;

end

default: begin

// Нет записи в регистр

end

endcase

end

endmodule

Далее опишем регистровый файл (Листинг 1.9).

Во время выполнения программы процессору необходимо где-то хранить промежуточные результаты вычислений. Регистровый файл предоставляет набор ячеек памяти (регистров), доступ к которым осуществляется очень быстро.

Например, при выполнении команды ADD R1, R2 процессор считывает значения из регистров R1 и R2, складывает их в АЛУ, а результат записывает обратно в регистр R1. Все эти операции были бы невозможны без регистрового файла.

В рассматриваемом алгоритме нахождения НВП регистры используются для хранения счетчиков циклов (R1, R2), адресов ячеек памяти (R1\_ADDR, R2\_ADDR), промежуточных результатов (R3, R4) и других важных значений.

Листинг 1.9 – Регистровый файл для процессорного ядра на языке Verilog

module register\_file(

input clk,

input reset,

input [4:0] read\_reg1,

input [4:0] read\_reg2,

input [4:0] write\_reg,

input write\_enable,

input [31:0] write\_data,

output reg [31:0] read\_data1,

output reg [31:0] read\_data2

);

reg [31:0] registers [0:31];

initial begin

for (int i = 0; i < 32; i = i + 1) begin

registers[i] <= 0;

end

end

always @(\*) begin

if (read\_reg1 == 0)

read\_data1 = 0;

else

read\_data1 = registers[read\_reg1];

if (read\_reg2 == 0)

read\_data2 = 0;

else

read\_data2 = registers[read\_reg2];

end

Продолжение Листинга 1.9

// Запись в регистр (синхронная по положительному фронту clk)

always @(posedge clk) begin

if (write\_enable) begin

if (write\_reg == 0) begin

registers[0] <= 0; // Регистр 0 всегда содержит 0

end else begin

registers[write\_reg] <= write\_data;

end

end

end

endmodule

**1.9 Верификация процессорного ядра**

Опишем тестовый модуль, реализующий верификационное окружение (Листинг 1.10).

Листинг 1.10 – Тестовый модуль для процессорного ядра на языке Verilog

`timescale 1ns / 1ps

module test;

reg clk;

reg reset;

processor uut (

.clk(clk),

.reset(reset)

);

initial clk = 0;

always #5 clk = ~clk;

initial begin

reset = 1;

#20;

reset = 0;

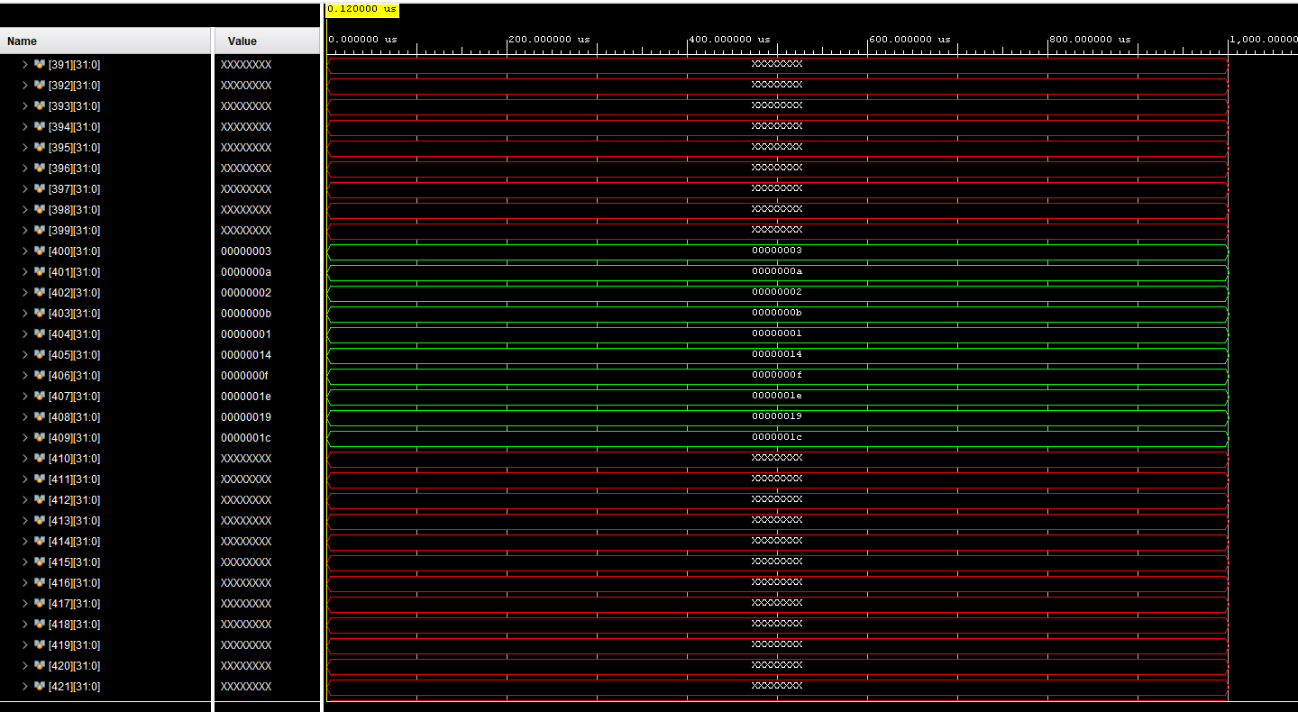
#10000;

$finish;

end

endmodule

Запись последовательности в память данных представлена ниже (Рисунок 1.7).

Рисунок 1.7 – Запись последовательности в процессорное ядро

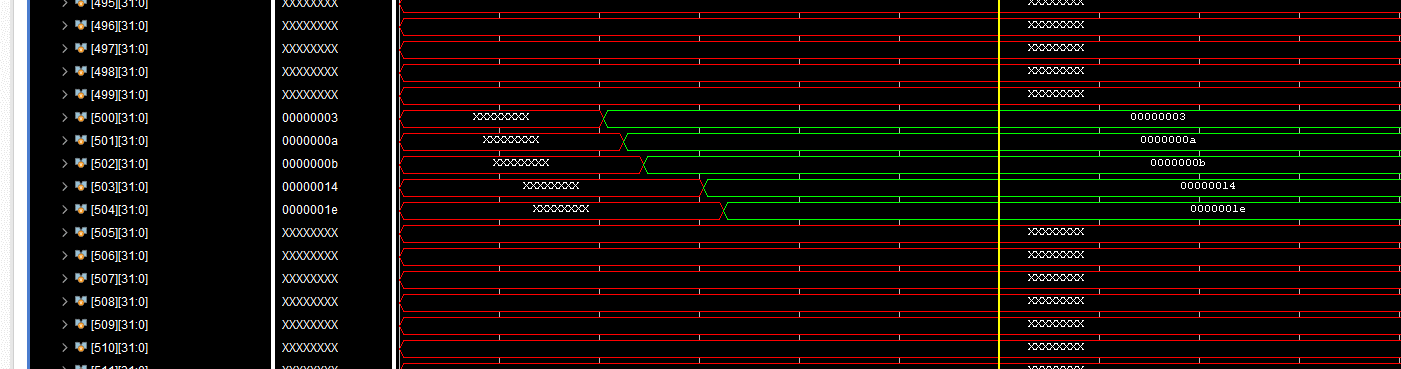
Запись результата по адресу 500 представлена ниже (Рисунок 1.8).

Рисунок 1.8 – Запись результирующей последовательности

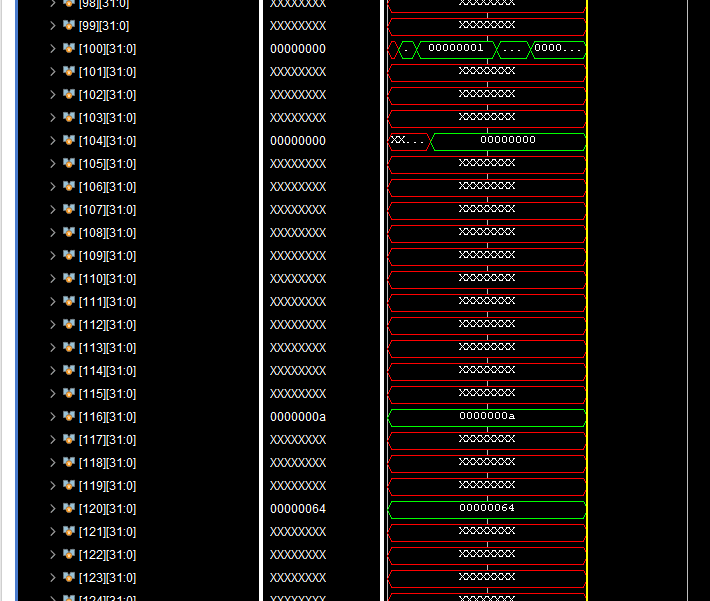
Запись перевёрнутой последовательности представлена ниже (Рисунок 1.9).

Рисунок 1.9 – Запись перевёрнутой последовательности в процессорное ядро

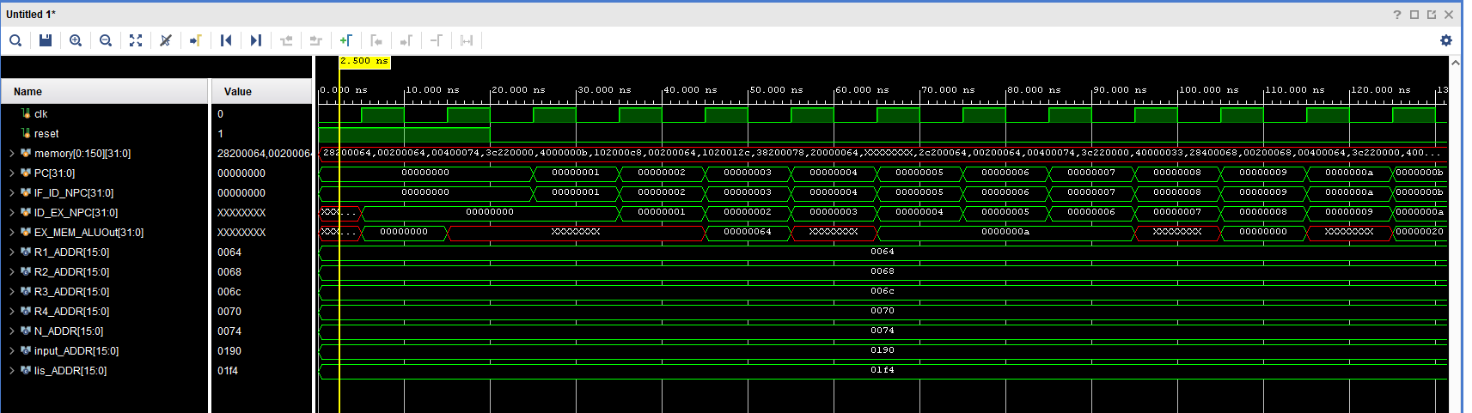
Общая работа процессора представлена ниже (Рисунок 1.10).

Рисунок 1.10 – Общая работа последовательного процессорного ядра

ЗАКЛЮЧЕНИЕ

В рамках данной лабораторной работы был спроектирован и верифицирован специализированный последовательный процессор, предназначенный для решения задачи нахождения наибольшей возрастающей подпоследовательности (НВП).

Был детально рассмотрен и проанализирован алгоритм решения данной задачи, основанный на методе динамического программирования.

На основе этого алгоритма разработана архитектура процессорного ядра, включающая в себя регистровый файл, арифметико-логическое устройство (АЛУ), память команд, память данных и управляющий автомат.

Особенностью данной архитектуры является наличие специализированных команд, оптимизированных для эффективного выполнения операций, часто встречающихся в алгоритме НВП.

Для разработанного процессорного ядра был написан код на языке Verilog, описывающий его структуру и поведение.

Программа, реализующая алгоритм нахождения НВП, была транслирована в машинные коды данного процессора.

Для проверки корректности работы процессора была проведена верификация методом потактовой симуляции с использованием специально разработанных тестовых последовательностей.

В результате выполнения лабораторной работы был получен полностью функционирующий процессор, способный решать поставленную задачу.

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).