



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
ВВЕДЕНИЕ.....	4
1 ОСНОВНОЙ РАЗДЕЛ	5
1.1 Основной алгоритм	5
1.2 Алгоритм на основе динамического программирования	6
1.3 Минимальный алгоритм на основе динамического программирования.....	12
1.4 Низкоуровневый алгоритм версии 1	16
1.5 Низкоуровневый алгоритм версии 2	21
1.6 Конечный автомат реализующий заданный алгоритм	26
1.7 Архитектура и микроархитектура последовательного процессорного ядра	34
1.8 Описание программы в машинных кодах	40
1.9 Верификация процессорного ядра.....	54

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

1.1 Основной алгоритм

За основу процессора возьмём одну из самых простых реализаций нахождения 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;

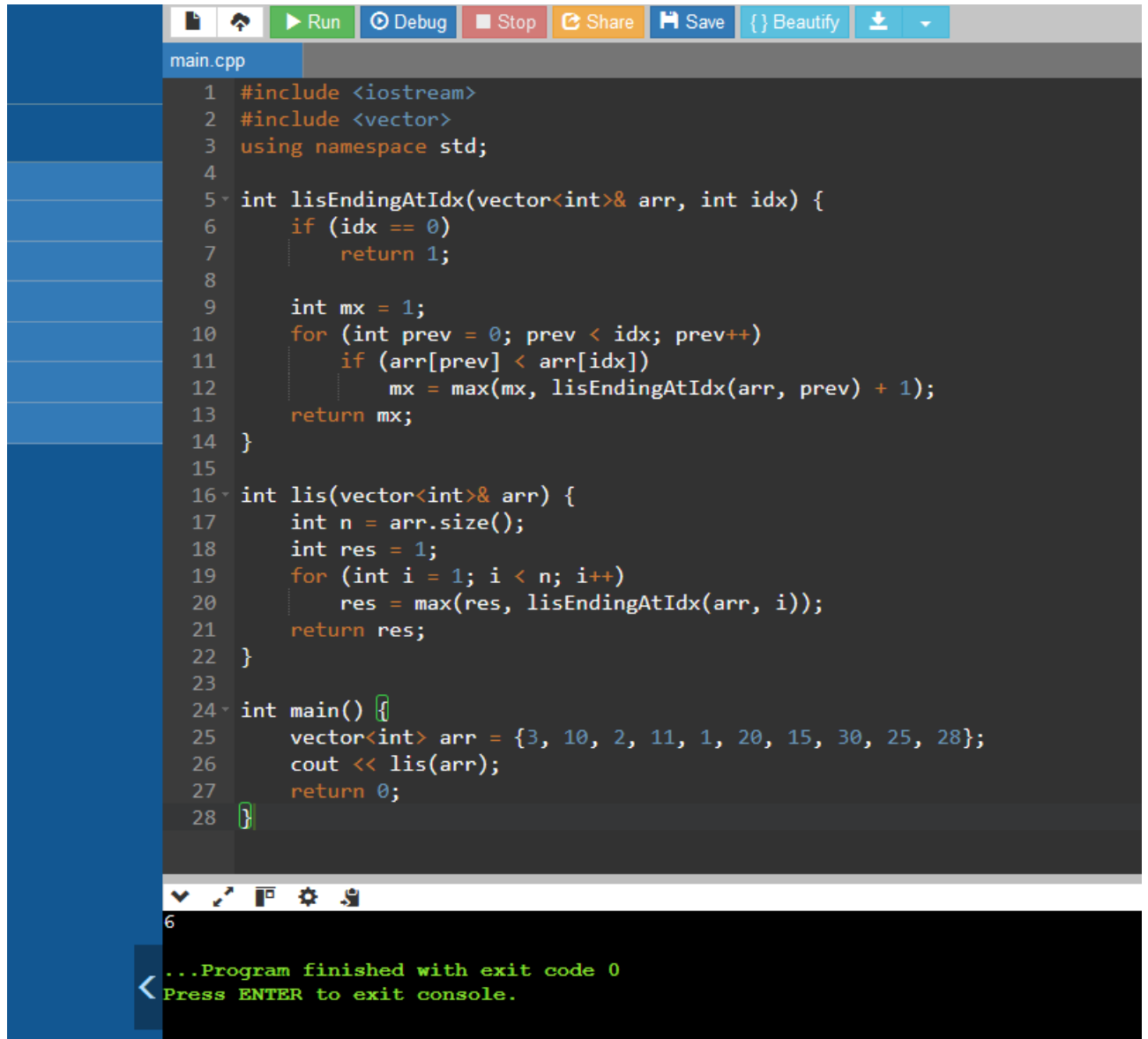
int lisEndingAtIdx(vector<int>& arr, int idx) {
    if (idx == 0)
        return 1;

    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;  
}
```



```
main.cpp  
1 #include <iostream>  
2 #include <vector>  
3 using namespace std;  
4  
5 int lisEndingAtIdx(vector<int>& arr, int idx) {  
6     if (idx == 0)  
7         return 1;  
8  
9     int mx = 1;  
10    for (int prev = 0; prev < idx; prev++)  
11        if (arr[prev] < arr[idx])  
12            mx = max(mx, lisEndingAtIdx(arr, prev) + 1);  
13    return mx;  
14 }  
15  
16 int lis(vector<int>& arr) {  
17     int n = arr.size();  
18     int res = 1;  
19     for (int i = 1; i < n; i++)  
20         res = max(res, lisEndingAtIdx(arr, i));  
21     return res;  
22 }  
23  
24 int main() {  
25     vector<int> arr = {3, 10, 2, 11, 1, 20, 15, 30, 25, 28};  
26     cout << lis(arr);  
27     return 0;  
28 }
```

6
...Program finished with exit code 0
Press ENTER to exit console.

Рисунок 1.1 – результат выполнения основного алгоритма

1.2 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 `src` 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 программирования на 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;
}
```

```
59     R5 = dp[R1]; // 0030 MOV R5, dp[R1]
60     R6 = R1; // 0031 MOV R6, R1
61 }
62 R1 = R1 + 1; // 0032 INCR R1
63 goto find_max_start; // 0033 JMP 0026
64 find_max_end:
65
66 // Restore LIS from dp and prev
67 restore_lis_start:
68 if (R6 == MAX_SIZE) // 0034 CMP R6, MAX_SIZE
69     goto restore_lis_end; // 0035 JEQ 0040
70 lis[lis_length] = input[R6]; // 0036 MOV lis[lis_length], input[R6]
71 lis_length = lis_length + 1; // 0037 INCR lis_length
72 R6 = prev[R6]; // 0038 MOV R6, prev[R6]
73 goto restore_lis_start; // 0039 JMP 0034
74 restore_lis_end:
75
76 // Reverse the lis array manually
77 R1 = 0; // 0040 XOR R1
78 R2 = lis_length - 1; // 0041 SUB R2, lis_length - 1
79 reverse_loop_start:
80 if (R1 >= R2) // 0042 CMP R1, R2
81     goto reverse_loop_end; // 0043 JGE 0050
82 R3 = lis[R1]; // 0044 MOV R3, lis[R1]
83 R4 = lis[R2]; // 0045 MOV R4, lis[R2]
84 lis[R1] = R4; // 0046 MOV lis[R1], R4
85 lis[R2] = R3; // 0047 MOV lis[R2], R3
86 R1 = R1 + 1; // 0048 INCR R1
87 R2 = R2 - 1; // 0049 DECR R2
88 goto reverse_loop_start; // 0050 JMP 0042
89 reverse_loop_end:
90
91 R1 = 0; // 0051 XOR R1
92 output_loop_start:
93 if (R1 >= lis_length) // 0052 CMP R1, lis_length
94     goto output_loop_end; // 0053 JGE 0057
95 result[R1] = lis[R1]; // 0054 MOV result[R1], lis[R1]
96 R1 = R1 + 1; // 0055 INCR R1
97 goto output_loop_start; // 0056 JMP 0052
98 output_loop_end: // 0057 HALT
99
100 for(int i = 0; i < lis_length; i = i + 1)
101     cout << result[i] << endl;
102 return 0;
103 }
```

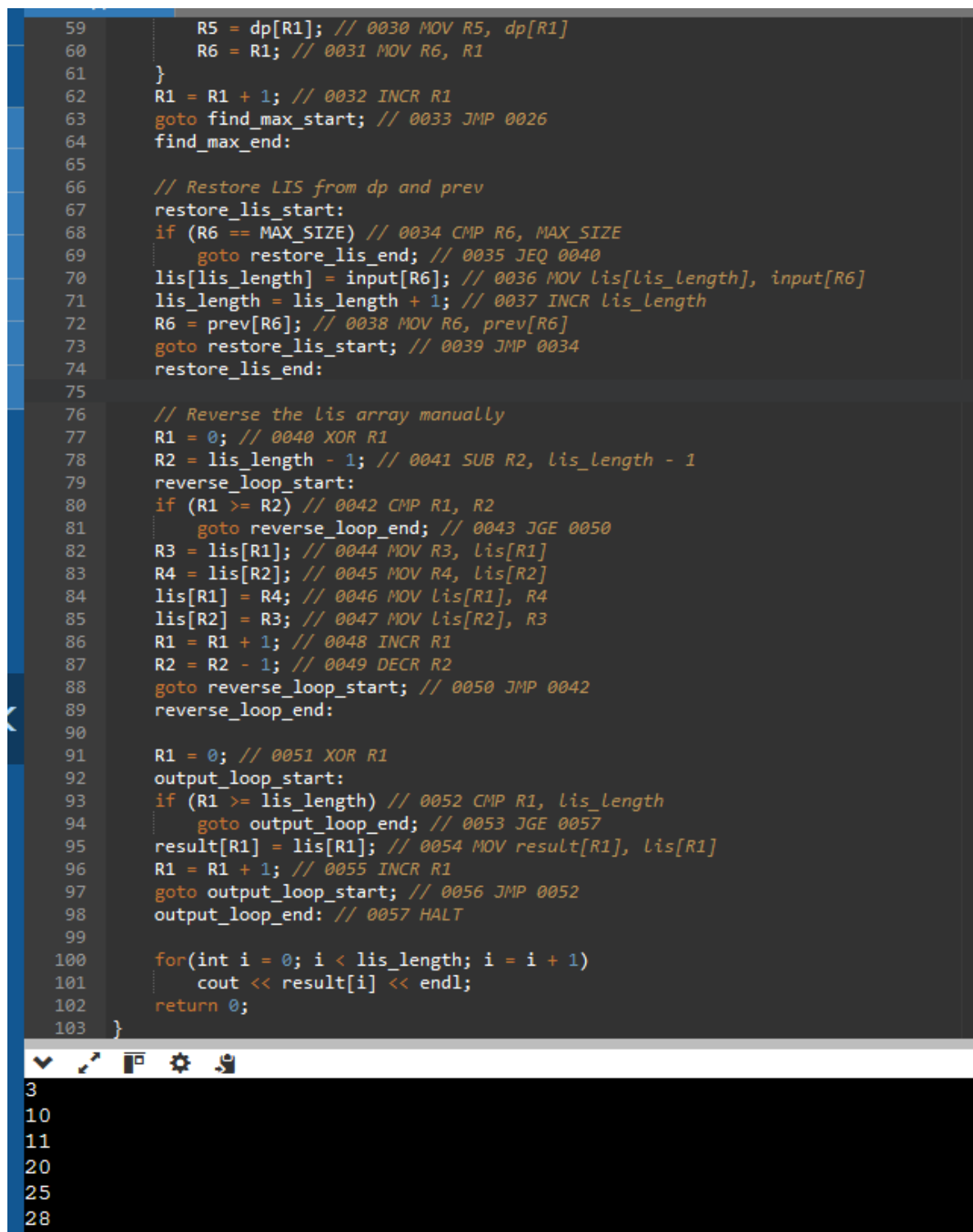


Рисунок 1.2 – результат работы алгоритма на основе динамического программирования

1.3 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 используется 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:

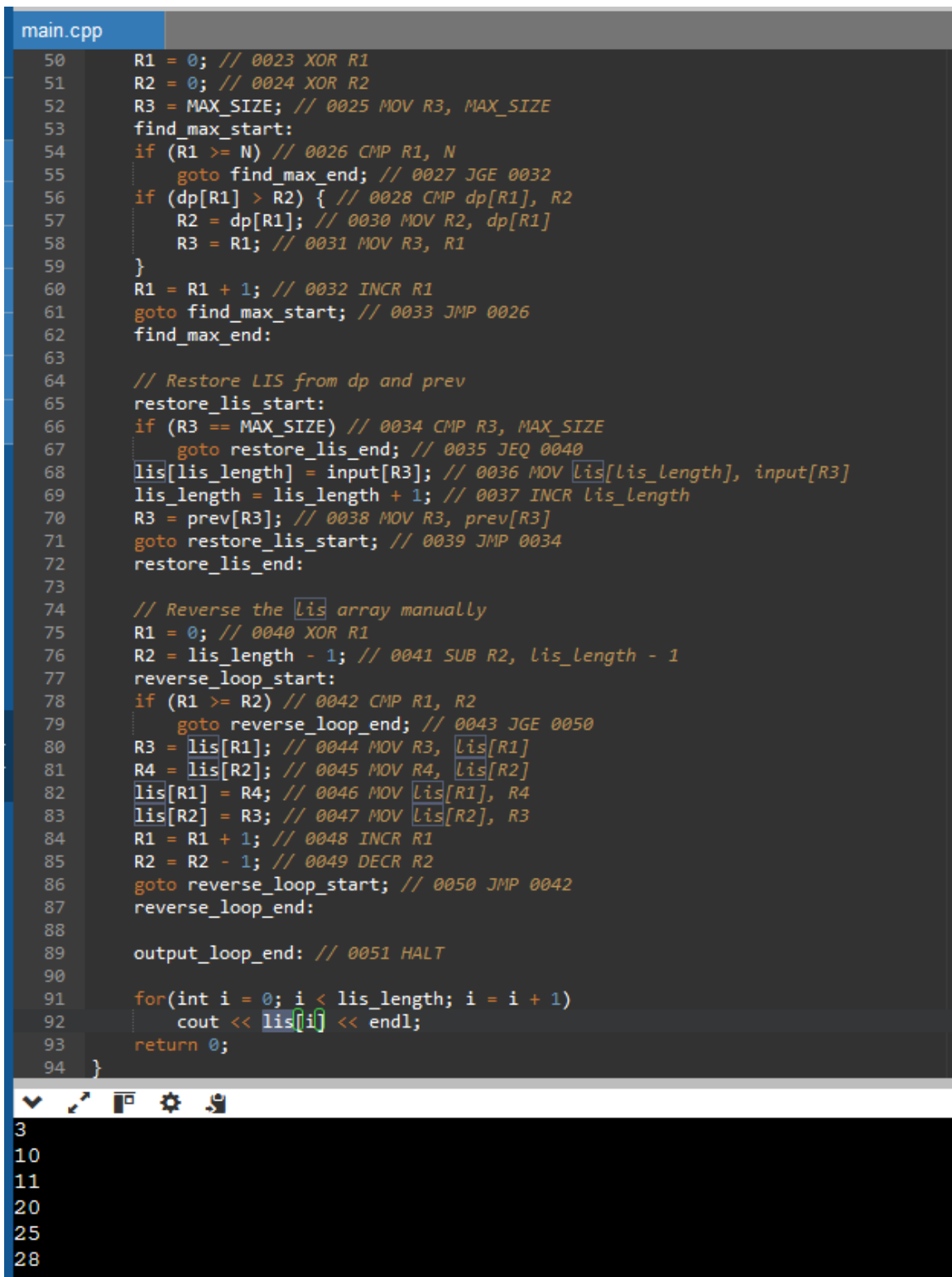
// Find the index of the maximum element in dp
R1 = 0; // 0023 XOR R1
R2 = 0; // 0024 XOR R2
R3 = MAX_SIZE; // 0025 MOV R3, MAX_SIZE
find_max_start:
if (R1 >= N) // 0026 CMP R1, N
    goto find_max_end; // 0027 JGE 0032
if (dp[R1] > R2) { // 0028 CMP dp[R1], R2
    R2 = dp[R1]; // 0030 MOV R2, dp[R1]
    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
```

```
main.cpp
50 R1 = 0; // 0023 XOR R1
51 R2 = 0; // 0024 XOR R2
52 R3 = MAX_SIZE; // 0025 MOV R3, MAX_SIZE
53 find_max_start:
54 if (R1 >= N) // 0026 CMP R1, N
55     goto find_max_end; // 0027 JGE 0032
56 if (dp[R1] > R2) { // 0028 CMP dp[R1], R2
57     R2 = dp[R1]; // 0030 MOV R2, dp[R1]
58     R3 = R1; // 0031 MOV R3, R1
59 }
60 R1 = R1 + 1; // 0032 INCR R1
61 goto find_max_start; // 0033 JMP 0026
62 find_max_end:
63
64 // Restore LIS from dp and prev
65 restore_lis_start:
66 if (R3 == MAX_SIZE) // 0034 CMP R3, MAX_SIZE
67     goto restore_lis_end; // 0035 JEQ 0040
68 lis[lis_length] = input[R3]; // 0036 MOV lis[lis_length], input[R3]
69 lis_length = lis_length + 1; // 0037 INCR lis_length
70 R3 = prev[R3]; // 0038 MOV R3, prev[R3]
71 goto restore_lis_start; // 0039 JMP 0034
72 restore_lis_end:
73
74 // Reverse the lis array manually
75 R1 = 0; // 0040 XOR R1
76 R2 = lis_length - 1; // 0041 SUB R2, lis_length - 1
77 reverse_loop_start:
78 if (R1 >= R2) // 0042 CMP R1, R2
79     goto reverse_loop_end; // 0043 JGE 0050
80 R3 = lis[R1]; // 0044 MOV R3, lis[R1]
81 R4 = lis[R2]; // 0045 MOV R4, lis[R2]
82 lis[R1] = R4; // 0046 MOV lis[R1], R4
83 lis[R2] = R3; // 0047 MOV lis[R2], R3
84 R1 = R1 + 1; // 0048 INCR R1
85 R2 = R2 - 1; // 0049 DECR R2
86 goto reverse_loop_start; // 0050 JMP 0042
87 reverse_loop_end:
88
89 output_loop_end: // 0051 HALT
90
91 for(int i = 0; i < lis_length; i = i + 1)
92     cout << lis[i] << endl;
93     return 0;
94 }
```



```
3
10
11
20
25
28
```

Рисунок 1.3 – результат работы минимального алгоритма на основе динамического программирования

1.4 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:
```

```
// 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;

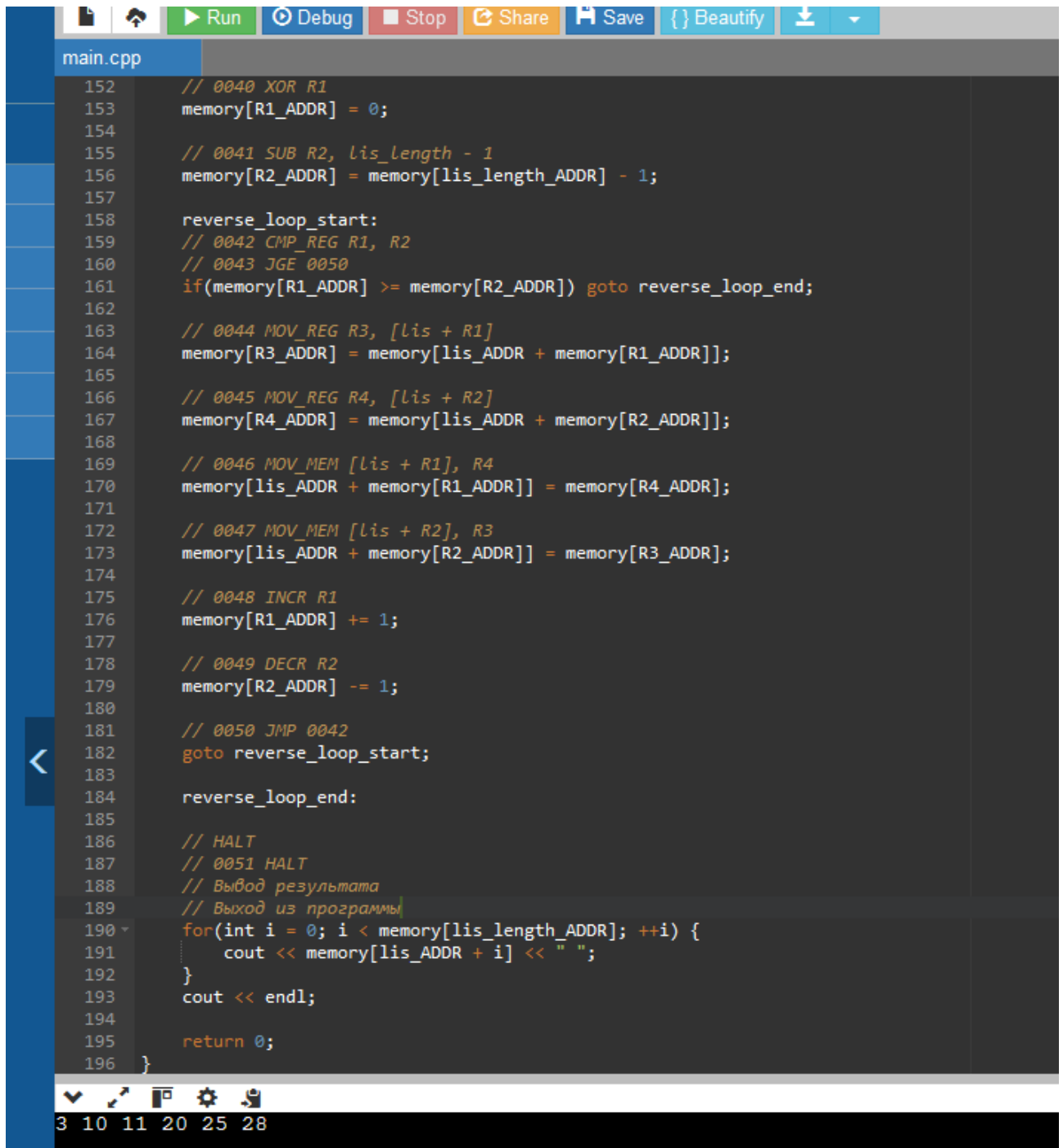
// 0049 DECR R2
memory[R2_ADDR] -= 1;

// 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;
}
```



```
main.cpp
152 // 0040 XOR R1
153 memory[R1_ADDR] = 0;
154
155 // 0041 SUB R2, lis_length - 1
156 memory[R2_ADDR] = memory[lis_length_ADDR] - 1;
157
158 reverse_loop_start:
159 // 0042 CMP_REG R1, R2
160 // 0043 JGE 0050
161 if(memory[R1_ADDR] >= memory[R2_ADDR]) goto reverse_loop_end;
162
163 // 0044 MOV_REG R3, [lis + R1]
164 memory[R3_ADDR] = memory[lis_ADDR + memory[R1_ADDR]];
165
166 // 0045 MOV_REG R4, [lis + R2]
167 memory[R4_ADDR] = memory[lis_ADDR + memory[R2_ADDR]];
168
169 // 0046 MOV_MEM [lis + R1], R4
170 memory[lis_ADDR + memory[R1_ADDR]] = memory[R4_ADDR];
171
172 // 0047 MOV_MEM [lis + R2], R3
173 memory[lis_ADDR + memory[R2_ADDR]] = memory[R3_ADDR];
174
175 // 0048 INCR R1
176 memory[R1_ADDR] += 1;
177
178 // 0049 DECR R2
179 memory[R2_ADDR] -= 1;
180
181 // 0050 JMP 0042
182 goto reverse_loop_start;
183
184 reverse_loop_end:
185
186 // HALT
187 // 0051 HALT
188 // Вывод результата
189 // Выход из программы
190 for(int i = 0; i < memory[lis_length_ADDR]; ++i) {
191     cout << memory[lis_ADDR + i] << " ";
192 }
193 cout << endl;
194
195 return 0;
196 }
```

Рисунок 1.4 – результат работы низкоуровневого алгоритма версии 1

1.5 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];
    }
}
```

```

}

// 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;
}
```

```

152 // 0040 XOR R1
153 memory[R1_ADDR] = 0;
154
155 // 0041 SUB R2, lis_length - 1
156 memory[R2_ADDR] = memory[lis_length_ADDR] - 1;
157
158 reverse_loop_start:
159 // 0042 CMP_REG R1, R2
160 // 0043 JGE 0050
161 if(memory[R1_ADDR] >= memory[R2_ADDR]) goto reverse_loop_end;
162
163 // 0044 MOV_REG R3, [lis + R1]
164 memory[R3_ADDR] = memory[lis_ADDR + memory[R1_ADDR]];
165
166 // 0045 MOV_REG R4, [lis + R2]
167 memory[R4_ADDR] = memory[lis_ADDR + memory[R2_ADDR]];
168
169 // 0046 MOV_MEM [lis + R1], R4
170 memory[lis_ADDR + memory[R1_ADDR]] = memory[R4_ADDR];
171
172 // 0047 MOV_MEM [lis + R2], R3
173 memory[lis_ADDR + memory[R2_ADDR]] = memory[R3_ADDR];
174
175 // 0048 INCR R1
176 memory[R1_ADDR] += 1;
177
178 // 0049 DECR R2
179 memory[R2_ADDR] -= 1;
180
181 // 0050 JMP 0042
182 goto reverse_loop_start;
183
184 reverse_loop_end:
185
186 // HALT
187 // 0051 HALT
188 // Вывод результата
189 // Выход из программы
190 for(int i = 0; i < memory[lis_length_ADDR]; ++i) {
191     cout << memory[lis_ADDR + i] << " ";
192 }
193 cout << endl;
194
195 return 0;
196 }

```

Рисунок 1.5 – результат работы низкоуровневого алгоритма версии 2

1.6 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
```

```

                                $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
                                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
                                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
                                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
endmodule
```

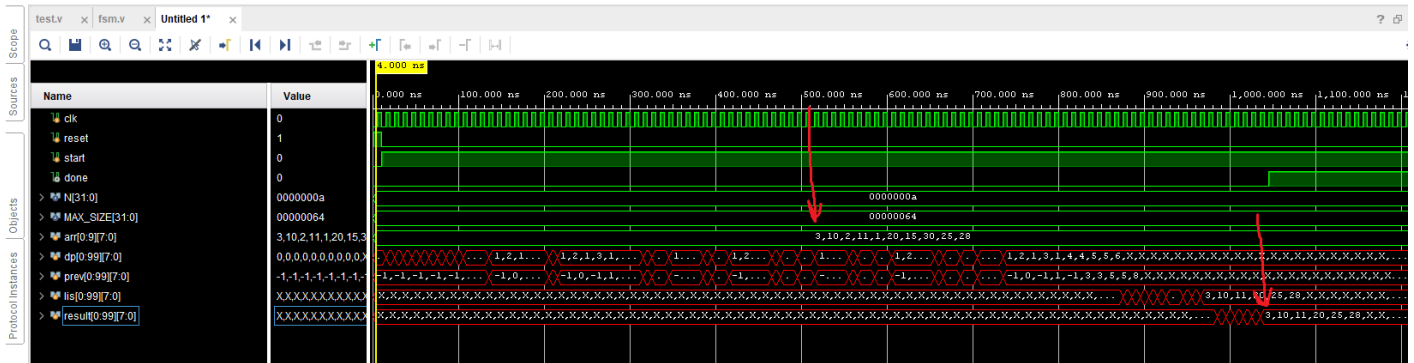



Рисунок 1.6 – результат работы конечного автомата на языке Verilog

```

Tcl Console  x  Messages  Log
[Search] [Zoom In] [Zoom Out] [Full Screen] [Print] [Close]

RESET: All registers initialized.
INIT_LOOP: dp[0] <= 1, prev[0] <= -1
INIT_LOOP: dp[1] <= 1, prev[1] <= -1
INIT_LOOP: dp[2] <= 1, prev[2] <= -1
INIT_LOOP: dp[3] <= 1, prev[3] <= -1
INIT_LOOP: dp[4] <= 1, prev[4] <= -1
INIT_LOOP: dp[5] <= 1, prev[5] <= -1
INIT_LOOP: dp[6] <= 1, prev[6] <= -1
INIT_LOOP: dp[7] <= 1, prev[7] <= -1
INIT_LOOP: dp[8] <= 1, prev[8] <= -1
INIT_LOOP: dp[9] <= 1, prev[9] <= -1
INIT_LOOP: Initialization complete. Moving to OUTER_LOOP.
OUTER_LOOP: Processing index 1.
INNER_LOOP: Comparing arr[0]=3 < arr[1]=10
INNER_LOOP: arr[0] < arr[1]
INNER_LOOP: dp[1]=1 < dp[0]=1 + 1
INNER_LOOP: Updated dp[1] <= 1, prev[1] <= 0
INNER_LOOP: Completed inner loop for index 1. Moving to next OUTER_LOOP.
OUTER_LOOP: Processing index 2.
INNER_LOOP: Comparing arr[0]=3 < arr[2]=2
INNER_LOOP: arr[0] >= arr[2], no action.
INNER_LOOP: Comparing arr[1]=10 < arr[2]=2
INNER_LOOP: arr[1] >= arr[2], no action.
INNER_LOOP: Completed inner loop for index 2. Moving to next OUTER_LOOP.
OUTER_LOOP: Processing index 3.
INNER_LOOP: Comparing arr[0]=3 < arr[3]=11
INNER_LOOP: arr[0] < arr[3]
INNER_LOOP: dp[3]=1 < dp[0]=1 + 1
INNER_LOOP: Updated dp[3] <= 1, prev[3] <= 0
INNER_LOOP: Comparing arr[1]=10 < arr[3]=11
INNER_LOOP: arr[1] < arr[3]
INNER_LOOP: dp[3]=2 < dp[1]=2 + 1
INNER_LOOP: Updated dp[3] <= 2, prev[3] <= 1
INNER_LOOP: Comparing arr[2]=2 < arr[3]=11
INNER_LOOP: arr[2] < arr[3]
INNER_LOOP: No update needed for dp[3].
INNER_LOOP: Completed inner loop for index 3. Moving to next OUTER_LOOP.
OUTER_LOOP: Processing index 4.
INNER_LOOP: Comparing arr[0]=3 < arr[4]=1
INNER_LOOP: arr[0] >= arr[4], no action.
INNER_LOOP: Comparing arr[1]=10 < arr[4]=1
INNER_LOOP: arr[1] >= arr[4], no action.
INNER_LOOP: Comparing arr[2]=2 < arr[4]=1
INNER_LOOP: arr[2] >= arr[4], no action.
INNER_LOOP: Comparing arr[3]=11 < arr[4]=1
INNER_LOOP: arr[3] >= arr[4], no action.

```

Рисунок 1.7 – результат работы конечного автомата на языке Verilog

1.7 Архитектура и микроархитектура последовательного процессорного ядра

Определим архитектуру и микроархитектуру 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 (РС): 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/Регистр 3	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'b0000000;  
    parameter STORE        = 6'b0000001;  
    parameter ADD          = 6'b0000010;  
    parameter SUB          = 6'b0000011;  
    parameter ADD_REG      = 6'b0000100;  
    parameter ADD_ANOTHER_REG = 6'b0000101;  
    parameter INCR_REG     = 6'b0000110;  
    parameter DECR_REG     = 6'b0000111;  
    parameter INCR_MEM     = 6'b0001000;  
    parameter DECR_MEM     = 6'b0001001;  
    parameter MEM_XOR      = 6'b0001010;  
    parameter MEM_ONE      = 6'b0001011;  
    parameter REG_MEM      = 6'b0001100;  
    parameter MEM_REG      = 6'b0001101;  
    parameter MEM_MEM      = 6'b0001110;  
    parameter CMP          = 6'b0001111;  
    parameter JGE          = 6'b0100000;  
    parameter JLE          = 6'b0100001;  
    parameter JEQ          = 6'b0100010;  
    parameter JMP          = 6'b0100011;  
    parameter HALT         = 6'b1111111;  
  
    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};
```

```
// 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
```

```
// 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  
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
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
end
endmodule
```

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

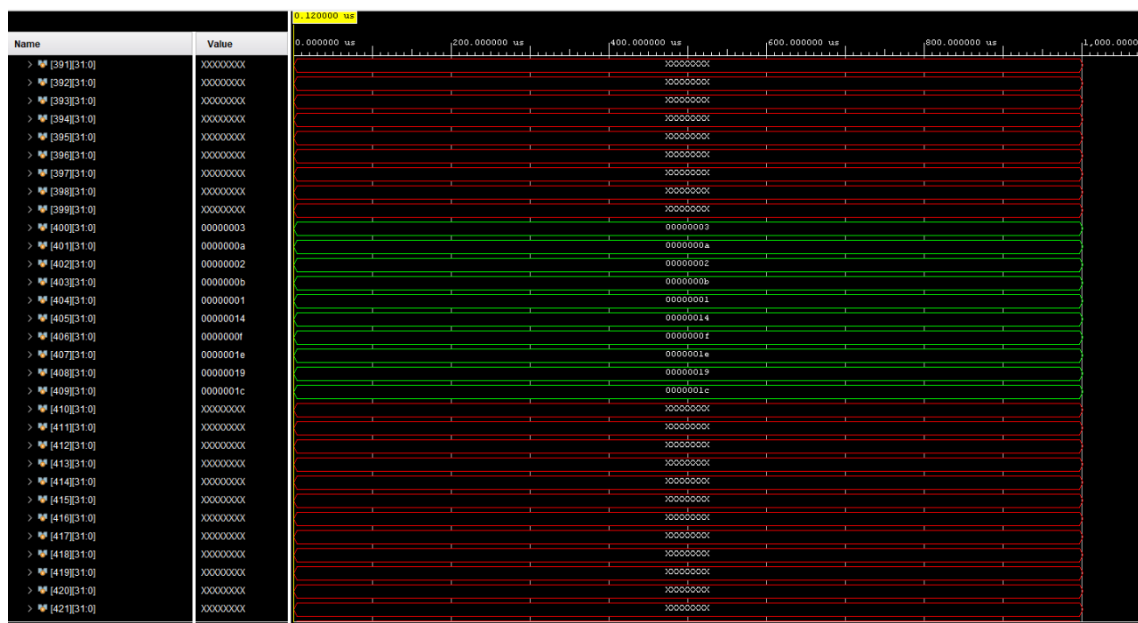


Рисунок 1.7 – Запись последовательности в процессорное ядро

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

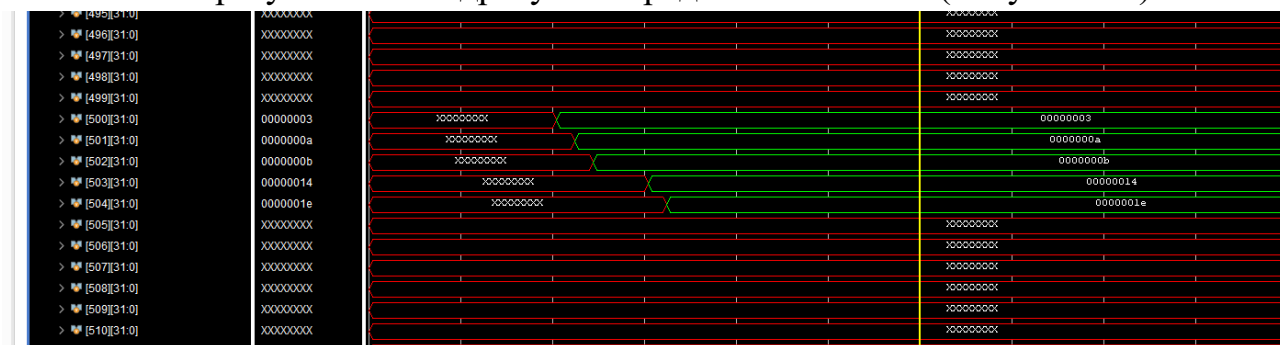


Рисунок 1.8 – Запись результирующей последовательности

Запись перевёрнутой последовательности представлена ниже (Рисунок 1.9).

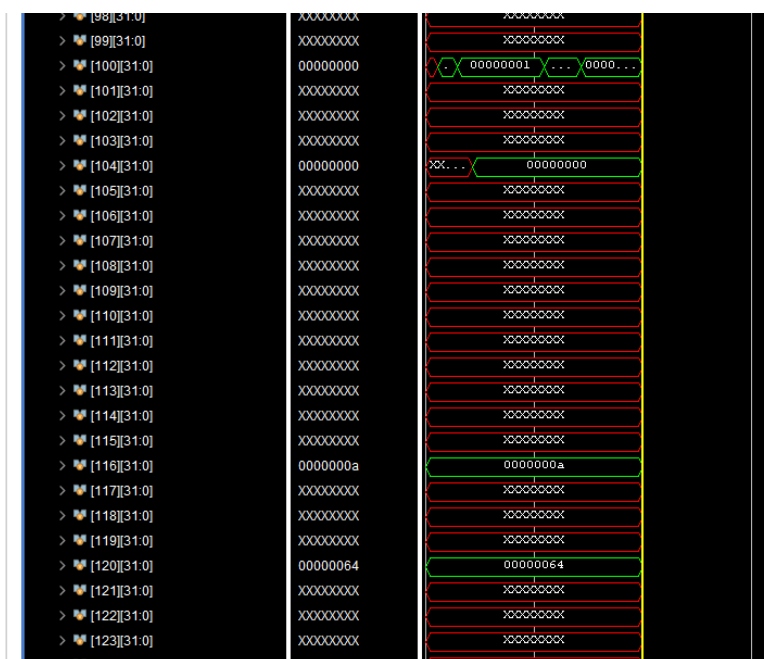


Рисунок 1.9 – Запись перевёрнутой последовательности в процессорное ядро

Общая работа процессора представлена ниже (Рисунок 1.10).

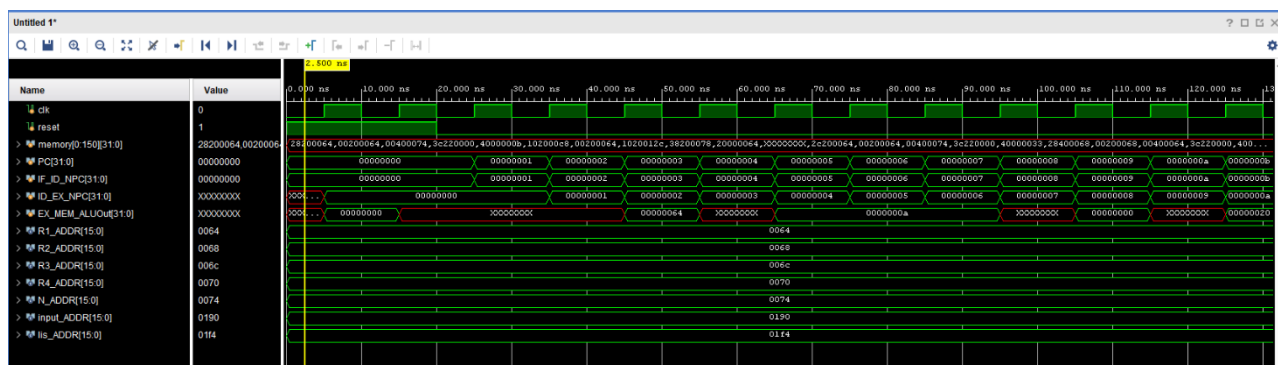


Рисунок 1.10 – Общая работа последовательного процессорного ядра

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?usp=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).