Theory of Automata

Lecturer: Zain Noreen
Section: CS-4K

GROUP MEMBERS:

LAIBA MOHSIN (22K-4246)
ANIQA AZHAR (22K-4228)
AISHA JALIL (22K-4649)

THEORY OF AUTOMATA - PROJECT REPORT

Turing machine for arithmetic operations for at least 5 user-provided arithmetic operators.

PROJECT DESCRIPTION:

The Turing Machine for Arithmetic Operations, developed using C++, performs arithmetic operations including addition, subtraction, multiplication, division, and the function 4x + 5. The Turing machine, a theoretical computational model, consists of a tape divided into cells, a read/write head that moves left or right along the tape, and a set of states that determine the machine's behavior.

Users can input arithmetic expressions containing these operators along with numerical operands and observe the step-by-step execution of the machine as it processes the input string symbol by symbol.

In this implementation, each state represents a step in the computation, and each transition rule specifies how the machine should update its state, tape contents, and head position based on the current state and the symbol read from the tape. The provided codes define transition rules for various arithmetic operations, encoded in unary and binary representations, and utilize vectors to store states, transition rules, and tape contents.
The codes iterate through transition rules for the current state, updating the tape and head position accordingly. If no valid transition is found, it throws a runtime error. Finally, they output the tape contents, head position, and final state.

LANGUAGE USED: C++

CODE:

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include <string>
#include <unordered_map>
#include <algorithm>

using namespace std;

// Define a structure to represent the transition rule
struct Transition {
    char read_condition;
    char write_value;
    char move_direction; // Change type to char
    int new_state;
};

struct DivTransition {
    char write_val;
    char move_dir;
    int new_state;
};


int main() {

    int choice;
    int position = 0;
    int state = 0, num1, num2, count;
    char num;

    cout << ".......Turing Machine.......\n";
    cout << "Select an option from below: \n";
    cout << "1. Addition.\n";
    cout << "2. Subtraction.\n";
    cout << "3. Multiplication.\n";
    cout << "4. Division.\n";
    cout << "5. Addition Function.\n";

    cout << "Enter choice: ";
    cin >> choice;

    while (choice > 5 || choice < 1) {
        cout << "Enter valid choice" <<endl;
        cin >> choice;
    }
    vector<int> accept_states;

    vector<int> func_states = { 0, 1, 2, 3, 4, 5, 6, 7 };
    vector<int> add_states = { 1, 2, 3, 4, 5 };
    vector<int> sub_states = { 0, 1, 2, 3, 4, 5, 6, 7 };
    vector<int> mult_states = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    vector<int> divstates = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
    vector<char> tape;
    unordered_map<int, unordered_map<char, DivTransition>> rules;
    // Define transition func_rules as a map from (state, read_condition) to Transition
    vector<vector<Transition>> func_rules = {
        {{'1', '1', 'r', 0}, {'0', '0', 'r', 0}, {'_', '0', 'r', 1}},
        {{'_', '0', 'r', 2}},
        {{'_', '_', 'l', 3}},
        {{'0', '1', 'l', 4}},
        {{'0', '0', 'l', 5}},
        {{'1', '0', 'l', 5},{'0', '1', 'l', 6}, {'_', '1', 'l', 6}},
        {{'1', '1', 'l', 6}, {'0', '0', 'l', 6}, {'_', '_', 'r', 7}}
    };

    // Define transition add_rules as a map from (state, read_condition) to Transition
    vector<vector<Transition>> add_rules = {
    {{'1', '_', 'r', 2}, {'+', '_', 'r', 3}},
    {{'1', '1', 'r', 2}, {'+', '1', 'r', 3}},
    {{'1', '1', 'r', 3}, {'_', '_', 'l', 4}},
    {{'1', '1', 'l', 4}, {'_', '_', 'r', 5}}
    };
```

```
// Define transition sub_rules as a map from (state, read_condition) to Transition
vector<vector<Transition>> sub_rules = {
    {{'1', '_', 'r', 1}, {'-', '-', 'r', 5}},
    {{'1', '1', 'r', 1}, {'-', '-', 'r', 1}, {'_', '_', 'l', 2}},
    {{'1', '_', 'l', 3}, {'-', '1', 'l', 4}},
    {{'1', '1', 'l', 3}, {'-', '-', 'l', 3}, {'_', '_', 'r', 0}},
    {},
    {{'_', '_', 'l', 6}, {'1', '1', 'l', 7}},
    {{'-', '_', 'l', 7} }
};

// Define transition mult_rules as a map from (state, read_condition) to Transition
vector<vector<Transition>> mult_rules = {
    {{'1', '_', 'r', 1}, {'*', '_', 'r', 9}},
    {{'1', '1', 'r', 1}, {'*', '*', 'r', 2}},
    {{'1', 'x', 'r', 3}, {'_', '_', 'l', 7}},
    {{'1', '1', 'r', 3}, {'_', '_', 'r', 4}},
    {{'1', '1', 'r', 4}, {'_', '1', 'l', 5}},
    {{'1', '1', 'l', 5}, {'_', '_', 'l', 6}},
    {{'1', '1', 'l', 6}, {'x', 'x', 'r', 2}},
    {{'x', '1', 'l', 7}, {'*', '*', 'l', 8}},
    {{'1', '1', 'l', 8}, {'_', '_', 'r', 0}},
    {{'1', '_', 'r', 9}, {'_', '_', 'r', 10}}
};

rules = {
{1, {{'_', {'_', 'r', 2}}}},
{2, {{'1', {'_', 'r', 3}}, {'/', {'_', 'r', 10}}}},
{3, {{'1', {'1', 'r', 3}}, {'/', {'/', 'r', 4}}}},
{4, {{'a', {'a', 'r', 4}}, {'1', {'1', 'r', 4}}, {'_', {'_', 'l', 5}}, {'b', {'b', 'l', 5}}}},
{5, {{'a', {'a', 'l', 5}}, {'1', {'a', 'l', 6}}}},
{6, {{'/', {'/', 'r', 7}}, {'1', {'1', 'l', 8}}}},
{7, {{'b', {'b', 'r', 7}}, {'a', {'1', 'r', 7}}, {'_', {'b', 'l', 8}}}},
{8, {{'b', {'b', 'l', 8}}, {'1', {'1', 'l', 8}}, {'/', {'/', 'l', 9}}}},
{9, {{'1', {'1', 'l', 9}}, {'_', {'_', 'r', 2}}}},
{10, {{'1', {'_', 'r', 10}}, {'b', {'1', 'r', 10}}, {'_', {'_', 'r', 11}}}}
};
```

```
switch (choice) {
case 1:

    cout << "Addition:\n";

    accept_states = { 5 };




    state = 1;

    cout << "Enter number 1: ";
    cin >> num1;
    cout << "Enter number 2: ";
    cin >> num2;

    for (int i = 0; i < num1; i++) {
        tape.push_back('1');
    }

    tape.push_back('+');

    for (int i = 0; i < num2; i++) {
        tape.push_back('1');
    }

    while (find(accept_states.begin(), accept_states.end(), state) == accept_states.end()) {
        char read_val = (position < tape.size()) ? tape[position] : '_';
        bool transition_found = false;

        // Iterate through transition add_rules for the current state
        for (const auto& transition : add_rules[state - 1]) {
            if (transition.read_condition == read_val) {
                tape[position] = transition.write_value;
                if (transition.move_direction == 'l') {
                    position = max(position - 1, 0);
                }
                else if (transition.move_direction == 'r') {
                    position++;
```

```
                        if (position >= tape.size()) {
                            tape.push_back('_');
                        }
                    }
                    state = transition.new_state;
                    transition_found = true;
                    break;
                }
            }

            if (!transition_found) {
                throw runtime_error("No transition found for state " + to_string(state) + " and
read condition " + read_val);
            }
        }

        // Output the tape, position, and state
        cout << "Tape: ";
        for (int i = 0; i < tape.size(); ++i) {
            if (i == position) {
                cout << "(" << tape[i] << ")";
            }
            else {
                cout << tape[i];
            }
        }
        cout << endl;
        cout << "Position: " << position << endl;
        cout << "State: " << state << endl;
        break;

    case 2:

        cout << "Subtraction:\n";

        accept_states = { 4, 7 };



        state = 0;
```

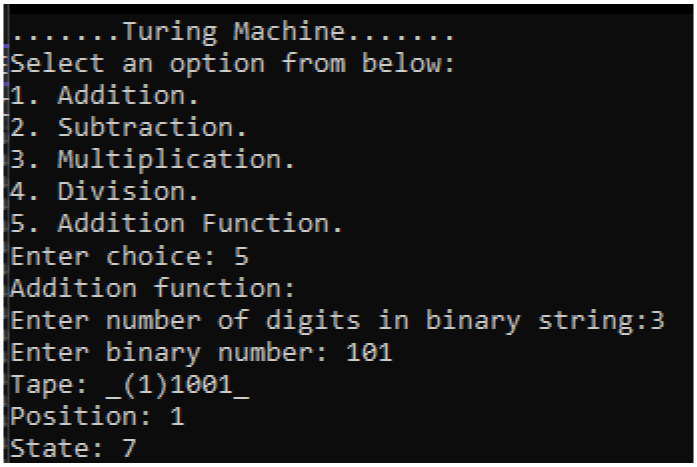
```
cout << "Enter number 1: ";
cin >> num1;
cout << "Enter number 2: ";
cin >> num2;

for (int i = 0; i < num1; i++) {
    tape.push_back('1');
}

tape.push_back('-');

for (int i = 0; i < num2; i++) {
    tape.push_back('1');
}


while (find(accept_states.begin(), accept_states.end(), state) == accept_states.end()) {
    char read_val = (position < tape.size()) ? tape[position] : '_';
    bool transition_found = false;

    // Iterate through transition sub_rules for the current state
    for (const auto& transition : sub_rules[state]) {
        if (transition.read_condition == read_val) {
            tape[position] = transition.write_value;
            if (transition.move_direction == 'l') {
                position = max(position - 1, 0);
            }
            else if (transition.move_direction == 'r') {
                position++;
                if (position >= tape.size()) {
                    tape.push_back('_');
                }
            }
            state = transition.new_state;
            transition_found = true;
            break;
        }
    }
```

```
            if (!transition_found) {
                throw runtime_error("No transition found for state " + to_string(state) + " and
read condition " + read_val);
            }
        }

        // Output the tape, position, and state
        cout << "Tape: ";
        for (int i = 0; i < tape.size(); ++i) {
            if (i == position) {
                cout << "(" << tape[i] << ")";
            }
            else {
                cout << tape[i];
            }
        }
        cout << endl;
        cout << "Position: " << position << endl;
        cout << "State: " << state << endl;
        break;

    case 3:
        cout << "Multiplication:\n";

        accept_states = { 10 };


        // Input number in unary (1-3)


        state = 0;
        cout << "Enter number 1: ";
        cin >> num1;
        cout << "Enter number 2: ";
        cin >> num2;

        for (int i = 0; i < num1; i++) {
            tape.push_back('1');
        }
    tape.push_back('*');

    for (int i = 0; i < num2; i++) {
      tape.push_back('1');
    }


    while (find(accept_states.begin(), accept_states.end(), state) == accept_states.end()) {
      char read_val = (position < tape.size()) ? tape[position] : '_';
      bool transition_found = false;

      // Iterate through transition mult_rules for the current state
      for (const auto& transition : mult_rules[state]) {
        if (transition.read_condition == read_val) {
          tape[position] = transition.write_value;
          if (transition.move_direction == 'l') {
            position = max(position - 1, 0);
          }
          else if (transition.move_direction == 'r') {
            position++;
            if (position >= tape.size()) {
              tape.push_back('_');
            }
          }
          state = transition.new_state;
          transition_found = true;
          break;
        }
      }

      if (!transition_found) {
        throw runtime_error("No transition found for state " + to_string(state) + " and
read condition " + read_val);
      }
    }

    // Output the tape, position, and state
    cout << "Tape: ";
    for (int i = 0; i < tape.size(); ++i) {
      if (i == position) {
            cout << "(" << tape[i] << ")";
        }
        else {
            cout << tape[i];
        }
    }
    cout << endl;
    cout << "Position: " << position << endl;
    cout << "State: " << state << endl;
    break;

case 4:

    accept_states = { 11 };

    cout << "Division: \n";

    cout << "Enter number 1: ";
    cin >> num1;
    cout << "Enter number 2: ";
    cin >> num2;


    for (int i = 0; i < num1; i++) {
        tape.push_back('1');
    }
    tape.push_back('/');
    for (int i = 0; i < num2; i++) {
        tape.push_back('1');
    }
    // Initialize position and state
    position = 0;
    state = 2;

    while (find(accept_states.begin(), accept_states.end(), state) == accept_states.end()) {
        char read_val = (position < tape.size()) ? tape[position] : '_';
        if (rules[state].find(read_val) == rules[state].end()) {
            throw runtime_error("No transition found.");
        }
        DivTransition transition = rules[state][read_val];
        tape[position] = transition.write_val;
        if (transition.move_dir == 'l') {
            position--;
            if (position < 0) {
                position++;
                tape.insert(tape.begin(), '_');
            }
        }
        else if (transition.move_dir == 'r') {
            position++;
            if (position >= tape.size()) {
                tape.push_back('_');
            }
        }
        state = transition.new_state;
    }
    // Print final tape, position, and state
    cout << "Tape: ";
    for (int i = 0; i < tape.size(); ++i) {
        cout << ((i == position) ? "(" + string(1, tape[i]) + ")" : string(1, tape[i]));
    }
    cout << endl << "Position: " << position << endl;
    cout << "State: " << state << endl;

    break;

case 5:

    cout << "Addition function: \n";

    accept_states = { 7 };


    state = 0;


    cout << "Enter number of digits in binary string:";
    cin >> count;

    cout << "Enter binary number: ";
```

```
        for (int i = 0; i < count; i++) {
            cin >> num;
            tape.push_back(num);
        }



        while (find(accept_states.begin(), accept_states.end(), state) == accept_states.end()) {
            char read_val = (position < tape.size()) ? tape[position] : '_';
            bool transition_found = false;

            // Iterate through transition func_rules for the current state
            for (const auto& transition : func_rules[state]) {
                if (transition.read_condition == read_val) {
                    tape[position] = transition.write_value;
                    if (transition.move_direction == 'l') {
                        position--;
                        if (position < 0) {
                            tape.insert(tape.begin(), '_');
                            position = 0;
                        }
                    }
                    else if (transition.move_direction == 'r') {
                        position++;
                        if (position >= tape.size()) {
                            tape.push_back('_');
                        }
                    }
                    state = transition.new_state;
                    transition_found = true;
                    break;
                }
            }

            if (!transition_found) {
                throw runtime_error("No transition found for state " + to_string(state) + " and
read condition " + read_val);
            }
        }
        // Output the tape, position, and state
        cout << "Tape: ";
        for (int i = 0; i < tape.size(); ++i) {
            if (i == position) {
                cout << "(" << tape[i] << ")";
            }
            else {
                cout << tape[i];
            }
        }
        cout << endl;
        cout << "Position: " << position << endl;
        cout << "State: " << state << endl;
        break;
    }

    return 0;
}
```

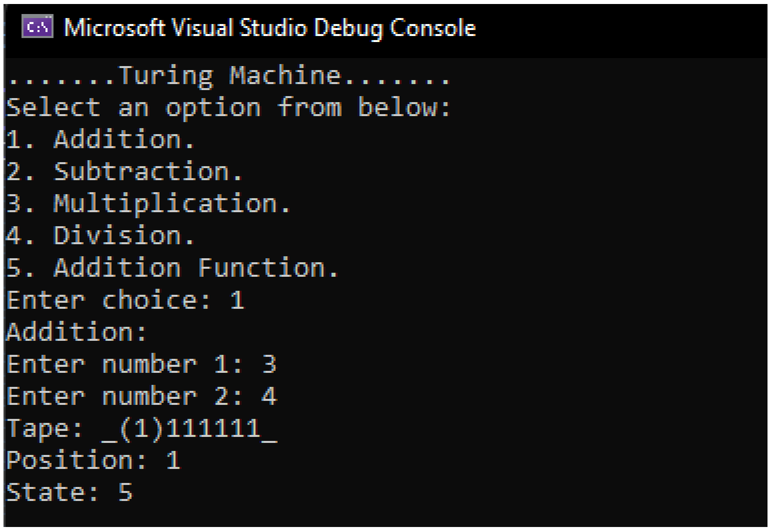
OUTPUTS:

Output for 4x+5 in binary:



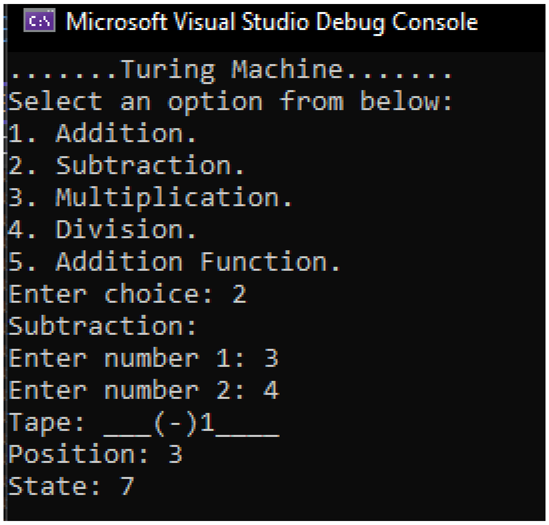
```
.......Turing Machine.......
Select an option from below:
1. Addition.
2. Subtraction.
3. Multiplication.
4. Division.
5. Addition Function.
Enter choice: 5
Addition function:
Enter number of digits in binary string:3
Enter binary number: 101
Tape: _(1)1001_
Position: 1
State: 7
```

Output of unary addition machine:



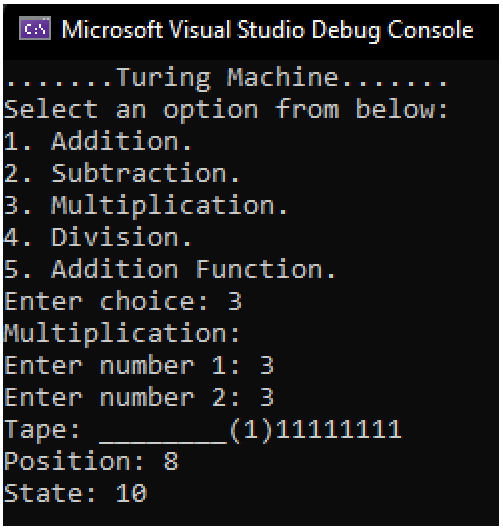
```
.......Turing Machine.......
Select an option from below:
1. Addition.
2. Subtraction.
3. Multiplication.
4. Division.
5. Addition Function.
Enter choice: 1
Addition:
Enter number 1: 3
Enter number 2: 4
Tape: _(1)111111_
Position: 1
State: 5
```

Output of unary Subtraction machine:



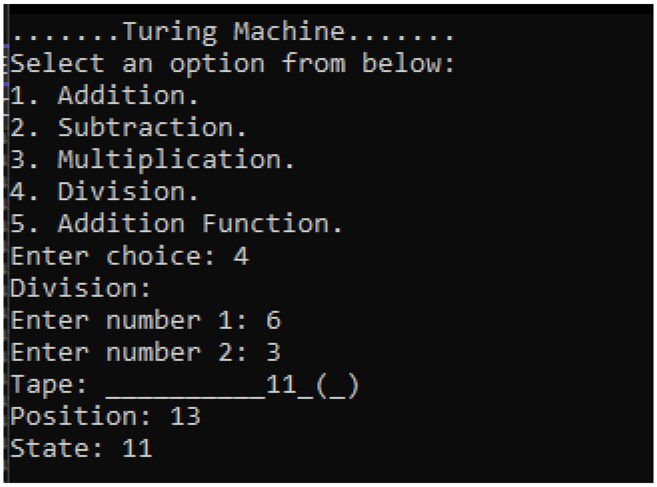
```
.......Turing Machine.......
Select an option from below:
1. Addition.
2. Subtraction.
3. Multiplication.
4. Division.
5. Addition Function.
Enter choice: 2
Subtraction:
Enter number 1: 3
Enter number 2: 4
Tape: ___(-)1____
Position: 3
State: 7
```

Output of unary multiplication machine:



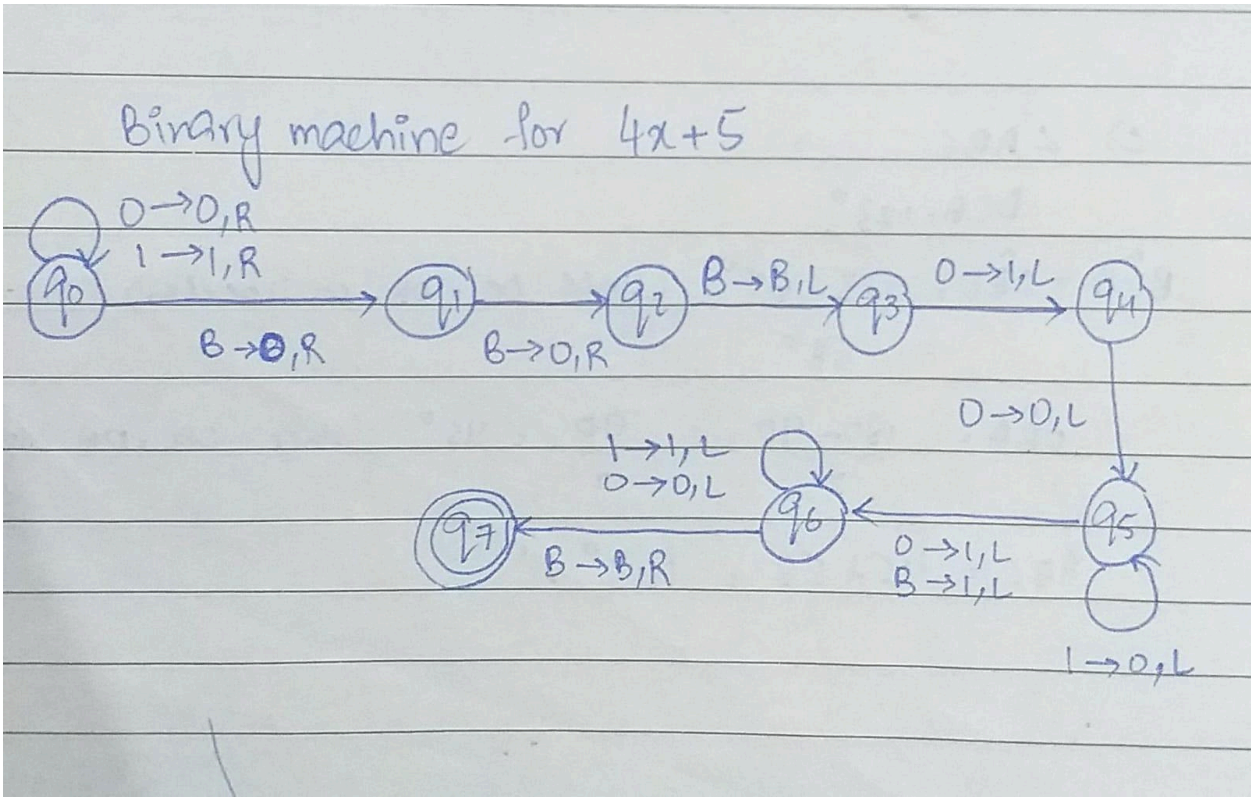
```
.......Turing Machine.......
Select an option from below:
1. Addition.
2. Subtraction.
3. Multiplication.
4. Division.
5. Addition Function.
Enter choice: 3
Multiplication:
Enter number 1: 3
Enter number 2: 3
Tape: ________(1)11111111
Position: 8
State: 10
```

Output of unary Division:



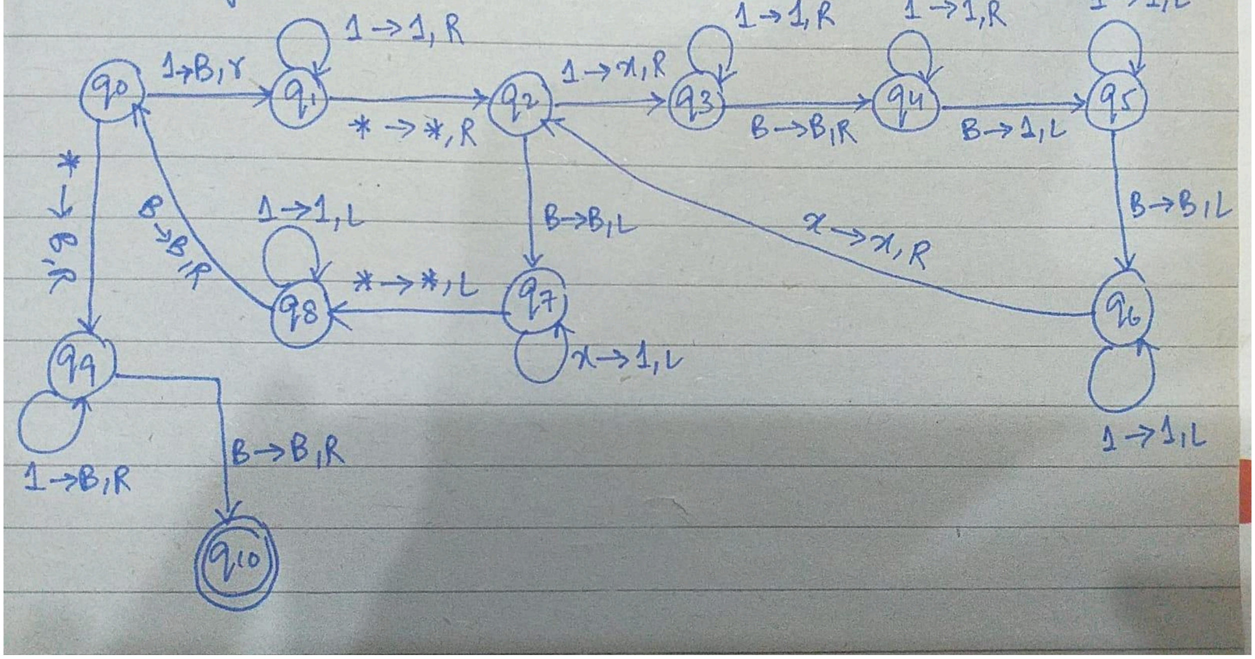
```
.......Turing Machine.......
Select an option from below:
1. Addition.
2. Subtraction.
3. Multiplication.
4. Division.
5. Addition Function.
Enter choice: 4
Division:
Enter number 1: 6
Enter number 2: 3
Tape: __________11_(_)
Position: 13
State: 11
```

MACHINES:

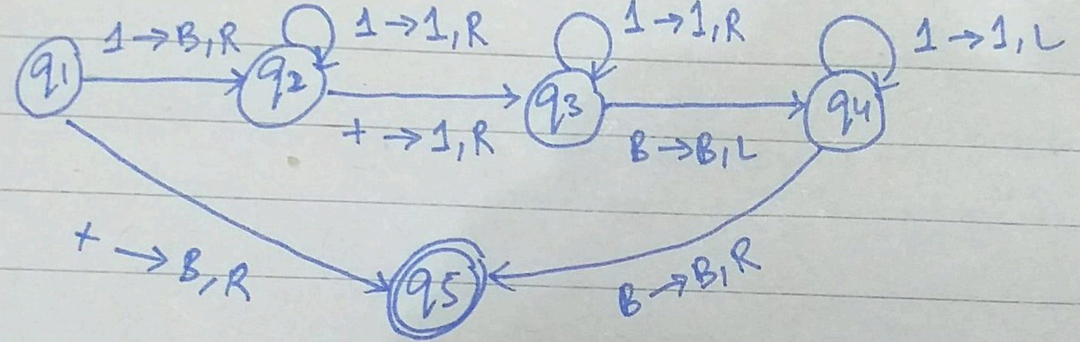


Binary machine for 4x+5

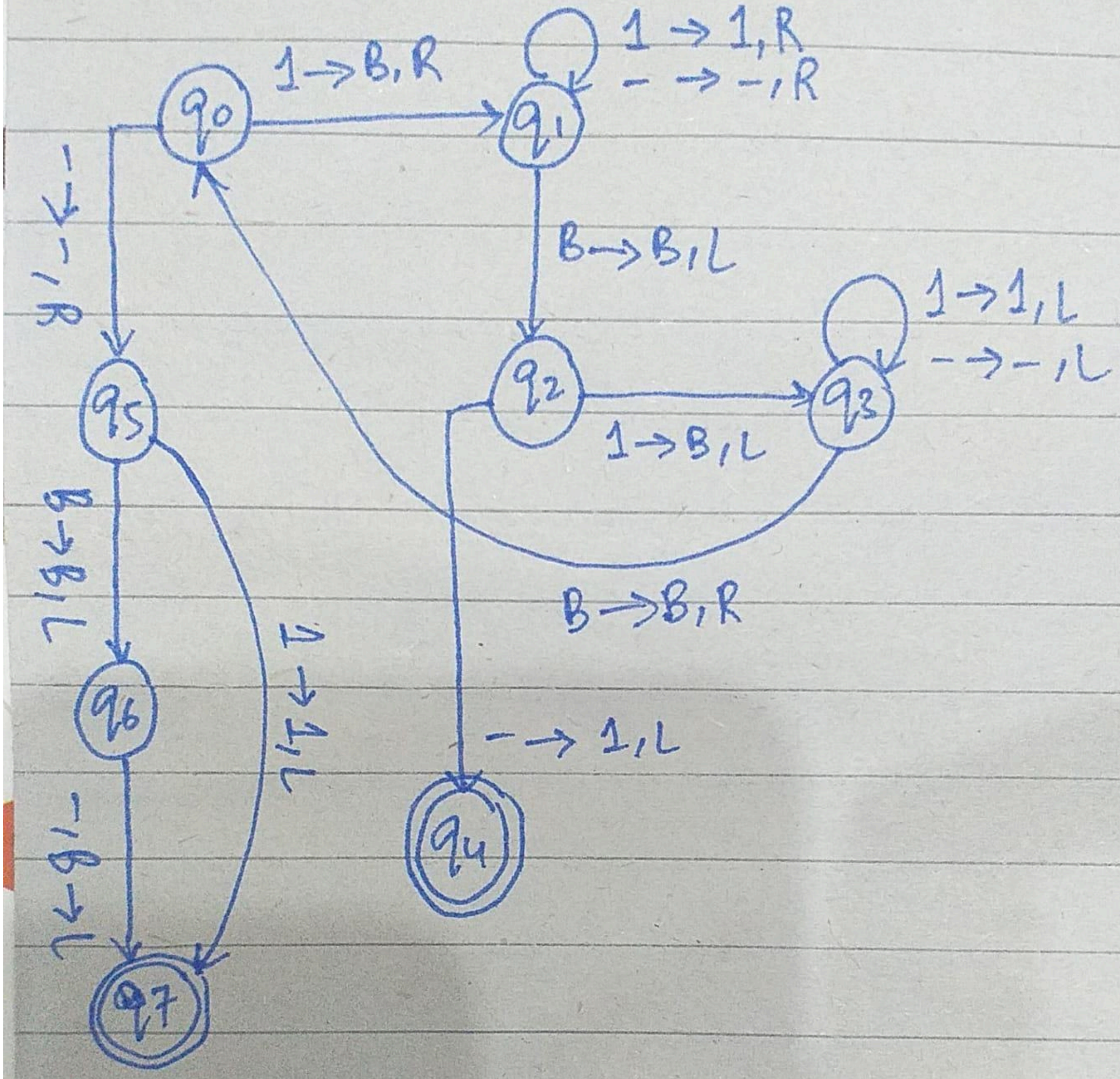
Unary multiplication machine:



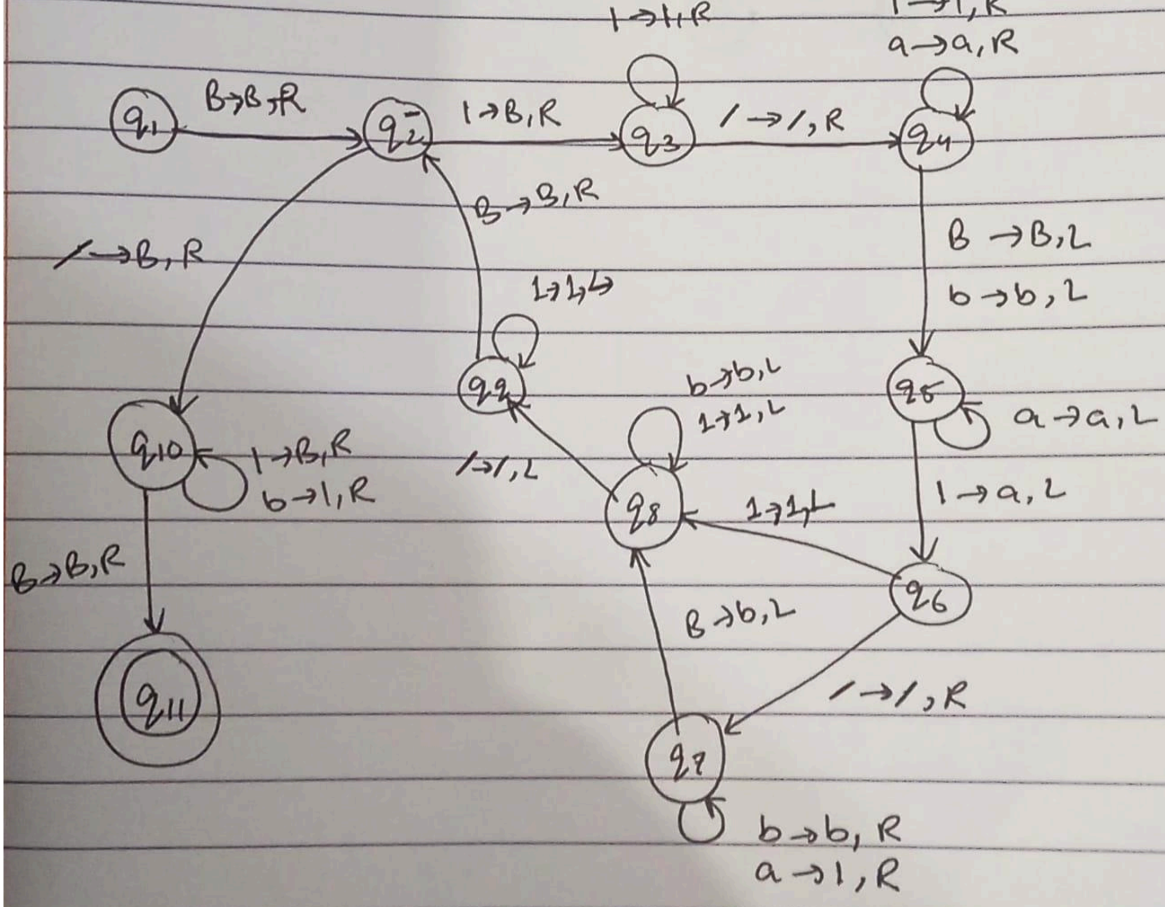
Unary addition machine:



Unary Subtraction machine :



Unary Machine for Division



1→1,R
1→1,R
a→a,R
q1
B→B,R
q2
1→B,R
q3
1→1,R
q4
B→B,R
1→B,R
B→B,L
b→b,L
1→1,L
q9
b→b,L
1→1,L
q5
a→a,L
q10
1→B,R
b→1,R
1→1,L
1→a,L
q8
1→1,L
B→B,R
q6
B→b,L
q11
1→1,R
q7
b→b,R
a→1,R