DNA Computing using Self-Assembly : Implementing Finite Field Arithmetic

A dissertation submitted in partial fulfillment
of the requirements of M.Tech.(Computer Science)
degree of Indian Statistical Institute, Kolkata

by

Shantanu Das

under the supervision of

Prof. Rana Barua

Indian Statistical Institute
Kolkata-700 108.

14th July 2003

Indian Statistical Institute

203, Barrackpore Trunk Road,

Kolkata-700 108.

Certificate of Approval

This is to certify that this thesis titled **"DNA Computing using Self-Assembly : Implementing Finite Fièld Arithmetic "** submitted by **Shantanu Das** towards partial fulfillment of requirements for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata embodies the work done under my supervision.

Rana Barua

Professor Rana Barua,
Division of Theoretical Statistics & Mathematics,
Indian Statistical Institute,
Kolkata-700 108.

Acknowledgements

I take this opportunity in thanking Professor Rana Barua for his guidance throughout the dissertation period. Without his constant encouragement and support this work would not have been possible.

I express my deep gratitude to all my teachers in the course. I am grateful to Prof. Tom Head of Binghamton University for taking the trouble of sending me the papers that I requested. These were of infinite help during my work.

Finally, my classmates and friends have been very helpful and supportive during the course of this work. I would specially like to thank Avishek Adhikari, Bireswar Das and M. Prem Laxman Das for their help and support.

Shantanu Das

Abstract

DNA computing is the new and emerging interdisciplinary field of study, where the tools of bio-technology are applied to problems in computing. In recent years, there has been a lot of research in this area and several methods have been proposed for solving various computational problems (specially the NP complete problems), using DNA. There has also been a few attempts to implement the basic operations in computing viz - the arithmetic and logical operations, using DNA computing. We look at the problem of implementing the finite field arithmetic operations using DNA computing. Finite field arithmetic has applications in the area of cryptology and coding theory and these operations are the basic operations in many coding/encryption algorithms. We propose two different DNA methods for implementing finite field arithmetic and discuss the merits and demerits of each. The first method is based on the traditionally used bio-chemical operations while the second method employs the process of self-assembly of DNA tiles. By using DNA computing for finite field arithmetic, we hope to parallely execute multiple such operations at a very low cost.

Contents

1 **Introduction** 1

2 **Operations on DNA** 3
- 2.1 Structure of DNA . . . 3
- 2.2 The operations . . . 5
  - 2.2.1 Synthesizing a DNA strand . . . 5
  - 2.2.2 Annealing of DNA strands . . . 5
  - 2.2.3 Melting . . . 5
  - 2.2.4 Ligating . . . 5
  - 2.2.5 Extraction . . . 5
  - 2.2.6 Amplifying with PCR . . . 6
  - 2.2.7 Gel Electrophoresis . . . 6
  - 2.2.8 Cutting using restriction enzymes . . . 6
  - 2.2.9 Sequencing of DNA strands . . . 7

3 **Methods and Models of DNA Computation** 8
- 3.1 Adleman's Experiment . . . 8
- 3.2 Splicing Systems . . . 9
- 3.3 Sticker Model . . . 9
- 3.4 Boolean Circuit Model . . . 10
- 3.5 Whiplash PCR . . . 10

4 **Algorithmic Self Assembly** 12
- 4.1 Wang Tiles . . . 12
- 4.2 DNA Tiles . . . 12
- 4.3 Blocked Cellular Automata(BCA) . . . 14
- 4.4 Implementing BCA using DNA Self Assembly . . . 14

5 **Finite Field Arithmetic** 16
- 5.1 Finite fields . . . 16
- 5.2 Binary Arithmetic using DNA . . . 17
- 5.3 Finite Field Addition . . . 18
- 5.4 Finite Field Multiplication . . . 18
- 5.5 Analysis of the method . . . 19

6 **Finite Field Arithmetic using Self-Assembly** 20
- 6.1 DNA TX Tiles . . . 20
- 6.2 Finite Field Arithmetic . . . 21
  - 6.2.1 Finite Field Multiplication . . . 22

6.2.2 An Example 26
6.2.3 Finite Field Addition 27
6.3 Implementation Issues 27
6.4 Analysis of the method 30

7 Conclusion **31**

Chapter 1

Introduction

In the quest for smaller and faster computers, people have started exploring alternative means of computing other than the standard electronic computers. Some of the unconventional methods of computing that have received much attention in recent years are quantum computing and biomolecular computing or DNA computing. DNA computing deals with the theoretical study as well as the practical implementation of methods of computing by manipulation of natural or artificial DNA molecules.

DNA (DeoxyriboNucleic Acid) is the carrier of genetic information in living organisms. DNA molecules consist of sequences of units called nucleotides. There are four nucleotide bases that are present in DNA named Adenine, Guanine, Cytosine and Thymine abbreviated as A,G,C, and T respectively. The particular sequence of A,T,G and C, that is present in a DNA molecule determines its information content. Thus DNA can be used to store information, by encoding the information using the four letter alphabet $\Sigma = \{A, G, C, T\}$ (As we all know, just two letters say, 0 and 1, are enough to encode any information).

In a broad sense, computing can be thought of as the conversion of a input string to an output string over some alphabet Σ. In that sense, DNA computation would involve converting the DNA encoding of a given input string into the DNA sequence encoding the corresponding output string. This conversion can be done using the bio-chemical reactions that can manipulate DNA molecules. Indeed, with the advancement of bio-molecular technology in last few decades, we have in our hand, a multitude of tools and techniques that can be used to operate on DNA molecules and manipulate them according to our requirements. It is even possible to create a DNA molecule having any particular nucleotide sequence. These bio-chemical tools (which are explained in Chapter 2), make DNA computing a reality.

The possibility of computing using DNA was first realized by L. Adleman, who in 1994, demonstrated the feasibility of DNA computing by a small experiment that he conducted in the laboratory (see [Adl94]). He was actually able to solve an instance of the *Hamiltonian Path Problem*(HPP), which is known to be an NP-complete problem, by performing some bio-chemical reactions on a set of test-tubes containing DNA molecules. The Hamiltonian Path Problem is - given a directed graph, one has to find a path(i.e. a sequence of adjacent edges) that starts at a vertex and ends at another vertex, passing through every other vertex exactly once. Adleman choose a small graph of seven vertices and encoded it using some DNA molecules. He then, conducted a series of biochemical reactions on these DNA molecules, at the end of which he was able to obtain a DNA strand

encoding a Hamiltonian path through the graph. The success obtained by Adleman in his experiment prompted many others to study the computational capabilities of DNA. Soon after Adleman published his result, R.Lipton came up with a generalized model for solving any problem using DNA and he gave a solution to the well-known NP complete problem called SAT(Satisfiability problem) using his method [Lip95]. With these two results, it was realized that not only is computing using DNA possible, but it has the potential of solving many problems which cannot be solved by the standard electronic computers. This gave birth to the field of DNA computing.

There are many advantages of DNA computing. The principal ones are its capability of providing high information density and allowing parallelism in computing at a low cost.

In the next chapter, we explain the basic bio-chemical operations that are used to manipulate DNA molecules. In chapter three, we review some of proposed methods of DNA computation and study the various theoretical models for DNA computing. The Algorithmic Self-Assembly method that was proposed by Winfree is studied in more detail, in chapter four. We then look at the problem of computing finite field arithmetic and discuss two DNA methods for implementing these, in chapters five and six. Finally we conclude by looking at the advantages and limitations of the proposed methods and discussing the future prospects of DNA computing.

Chapter 2

Operations on DNA

In order to understand the DNA computing methods presented in this thesis, the reader needs to know about the structure of DNA and the various bio-chemical operations that can be used to manipulate the DNA structures. We briefly describe below the primary operations that can be performed on DNA structures, in the laboratory. But first let us have a look at the structure of DNA molecules.

2.1 Structure of DNA

Naturally occurring DNA (that is found in almost all living beings), has a double helical shape. A double helix of DNA is made up of two single strands of DNA, each of which is a chain of nucleotides. The nucleotides are the basic building blocks of DNA. A nucleotide is an organic structure with three basic parts: a phosphate group, a 5-carbon sugar group and a nitrogenous side group. The five carbon atoms are labeled $1'$ through $5'$. The nitrogenous group called the base comes in four different varieties that are named *Adenine, Guanine, Thymine* and *Cytosine* abbreviated as A, G, T, and C respectively. The nucleotide itself is identified by the base it contains and is labeled by the same letter as its base.

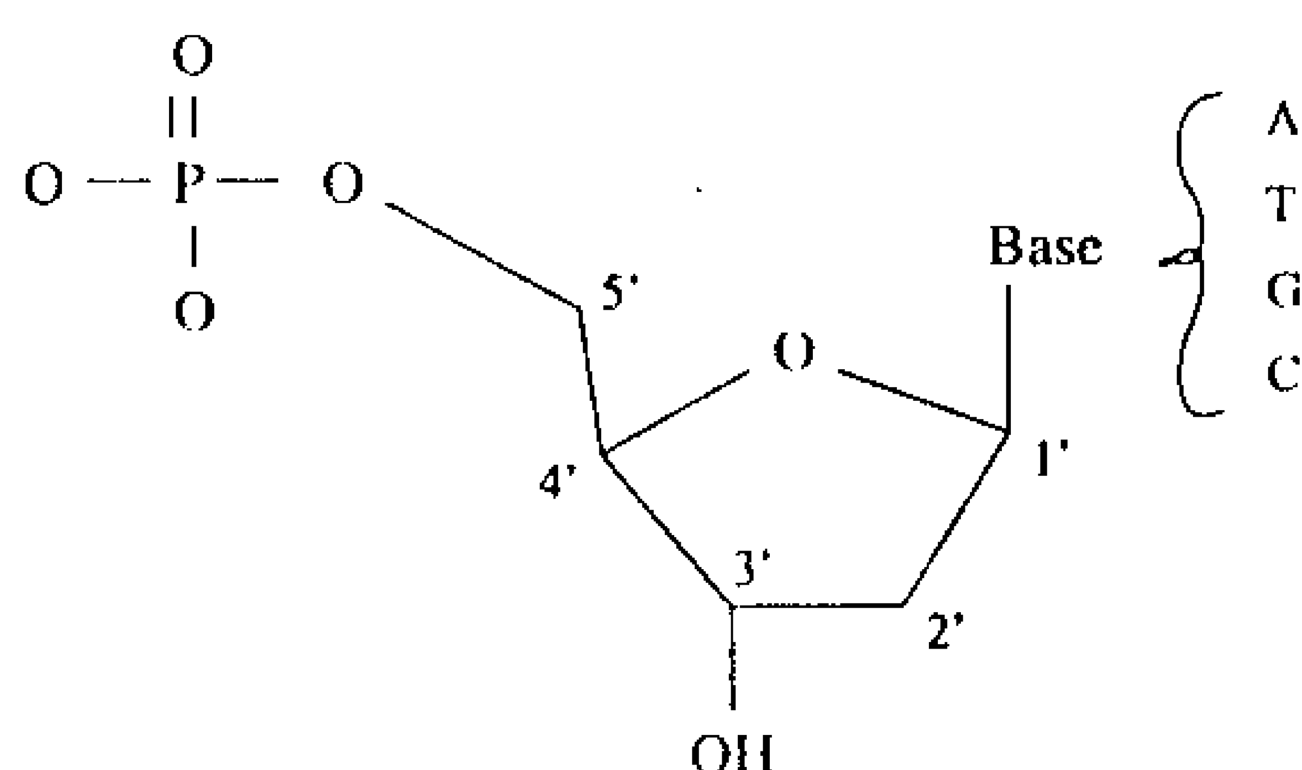


Figure 2.1: Structure of a nucleotide

A single strand of DNA is a chain of nucleotides, with the sugar group of one nucleotide bound to the phosphate group of the next nucleotide and so on. Thus, there is a chain of alternating sugar and phosphate groups, which forms the backbone of a DNA strand, with a free phosphate group on one end and a free sugar group on the other end. The end with the free phosphate group is called the $5'$ end (because the phosphate group is connected

to the carbon atom labeled 5′) and the other end is called the 3′ end (because it contains a free hydroxyl(OH) group connected to the 3′ carbon atom). So, the two ends of a DNA single strand are distinct - this gives a polarity to the DNA strand. We can represent a DNA single strand in the form $5'GAATCCGT3'$, meaning that starting from the 5′ end the nucleotides contained in the strand are G, A, A, T, C, C, G, and T in that order. (A short single strand of DNA with upto 20 nucleotides is often called an *oligonucleotide*).

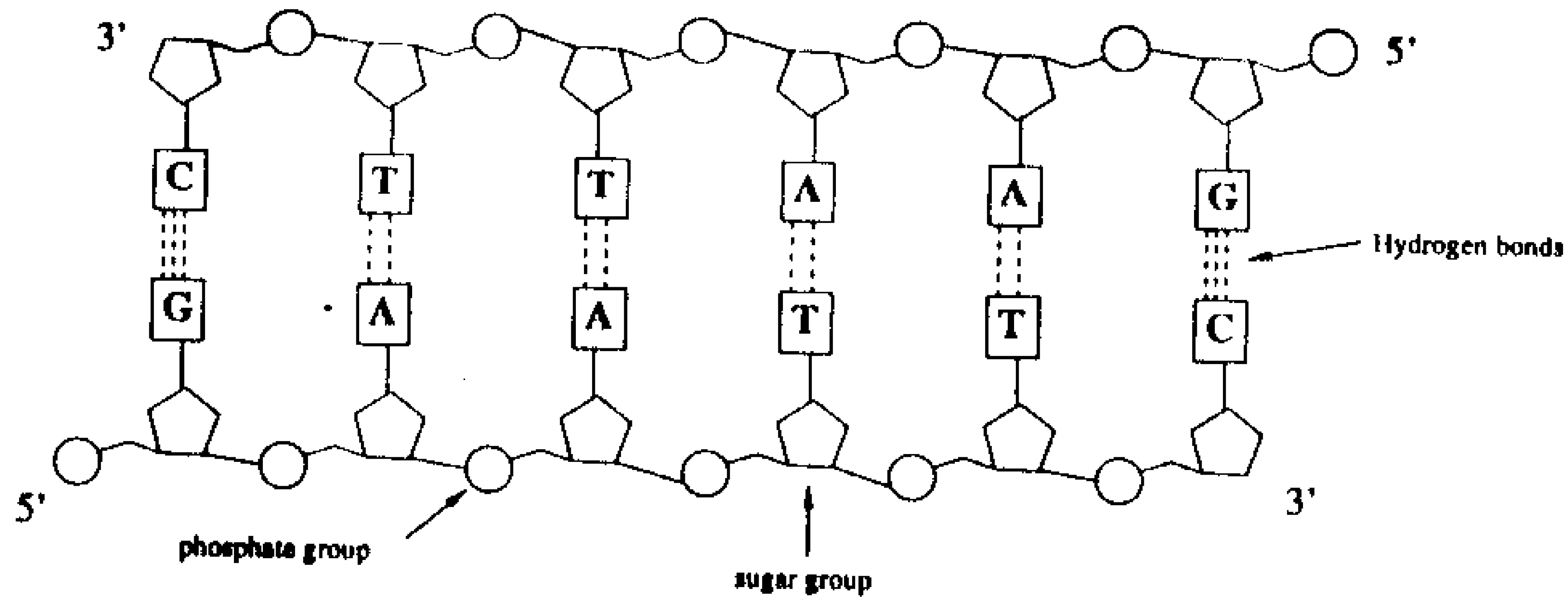


Figure 2.2: Structure of DNA

As stated earlier, the naturally occurring DNA is in the form of a double helix with two strands twisted around each-other. If we could un-twist the double helix DNA (hypothetically speaking), we would get the structure shown in Figure 2.2, with two single strands running antiparallely and the corresponding nucleotides joined with each-other through hydrogen bonds. Each nucleotide in a DNA has a complement with which it can bind (through hydrogen bonds). A and T are complementary, and G and C are complementary (i.e. A can only bind with T and G can only bind with C). Thus corresponding to a single strand $5'GAATCCGT3'$, we have a complementary strand $3'CTTAGGCA5'$ and the two can join to form a double helix DNA. In each double helix DNA molecule, the two constituent strands have to be complementary to each other. This is called Watson-Crick complementarity.

We use the notation

$$\updownarrow GAATCCGT$$

to represent the double stranded DNA molecule whose upper strand is $5'GAATCCGT3'$ and the lower strand is the complementary strand $3'CTTAGGCA5'$.

$$\uparrow GAATCCGT$$

would represent the upper strand of the complex and

$$\downarrow GAATCCGT$$

would represent the lower strand. (Note that $\downarrow GAATCCGT$ represents $3'CTTAGGCA5'$ and not $3'GAATCCGT5'$.) These notations have been borrowed from [BDS96].

2.2 The operations

2.2.1 Synthesizing a DNA strand

A DNA strand having the desired nucleotide sequence can be synthesized by a laboratory procedure. In standard solid-phase DNA synthesis, a desired DNA molecule is built up nucleotide by nucleotide on a support particle in sequential coupling steps. For example, the first nucleotide (monomer), say A, is bound to a glass support. A solution containing C is poured in, and the A reacts with the C to form a two-nucleotide (2-mer) chain AC. After washing the excess C solution away, one could have the C from the chain AC coupled with T to form a 3-mer chain (still attached to the surface) and so on. Note that, nowadays DNA strands having specific nucleotide sequences can be bought straight from the market. So, it is not necessary to execute this step in the laboratory.

2.2.2 Annealing of DNA strands

Two complementary DNA single strands can be made to anneal or bind together to form a double strand, by just mixing together the two strands and cooling the solution. At a low temp, the hydrogen bonds between the nucleotides form, and we get the double helix DNA structure from its constituent strands.

Sometimes two strands which are almost complementary but not perfectly complementary (i.e. some of the nucleotides are not complementary) can also anneal together to form an irregular double helix with the unmatched portions looping out. This is a source of possible error during the annealing operation.

2.2.3 Melting

This is the opposite of the annealing operation where a solution containing a DNA double strand is heated to break it apart into the constituent single strands. Heating breaks up the weak hydrogen bonds that bind together the two single strands, thus, allowing them to separate. This operation is always error-free.

2.2.4 Ligating

The enzyme *Ligase* joins together two DNA strands end to end or ligates them. When a partially double stranded molecule having a single-stranded overhanging end (called sticky end), attaches with another such molecule having the complementary sticky end, then the ends of the strands adjacent to each other can be joined using *Ligase*. This is the ligating operation.

2.2.5 Extraction

DNA single strands containing a particular pattern (nucleotide sequence) as a substring can be extracted out of a heterogeneous pool of DNA strands using the affinity purification operation. To do this, the nucleotide sequence complementary to the particular pattern sequence is created (synthesized) and these are attached to magnetic beads. Then the solution is passed over the beads. Those strands which contain the particular pattern sequence get annealed to the complementary strands that are attached to beads and thus

these can be separated out using a magnetic field. This operation does not always succeed in extracting all the strands containing the particular pattern and also sometimes a imperfectly matching sequence may get extracted. Thus, this operation is not error-free.

2.2.6 Amplifying with PCR

The Polymerase Chain Reaction(PCR), uses the *DNA polymerase* enzyme to replicate DNA. The replication reaction requires a guiding DNA single strand called the template and a shorter strand called the primer, that is annealed to it. Under these conditions, the DNA polymerase catalyses DNA synthesis by successively adding nucleotides to one end of the primer. The primer is thus extended in one direction until the desired strand that starts with the primer and is complementary to the template is obtained. For amplifying (i.e. making several copies of) a DNA molecule, it is first melted into the constituent single strands which act as the templates. These are annealed to the primers and then the primer is extended by DNA polymerase. This cycle is repeated many times, each time doubling the number of target DNA molecules, to get an exponential growth in the number of such molecules.

2.2.7 Gel Electrophoresis

The gel electrophoresis method is used to separate DNA strands based on their length. The size-wise separation of DNA strands is done by placing them at the top of a wet gel and to which an electric field is applied, drawing them to the bottom. Larger molecules travel more slowly through the gel. After a period, the molecules spread out into distinct bands according to size. This operation is very sensitive and is mostly error-free.

2.2.8 Cutting using restriction enzymes

DNA double strands can be cut at specific sites using restriction enzymes called *restriction endonucleases*. There are a lot of such enzymes, each of which recognizes a particular short DNA sequence called restriction site. On applying such a restriction enzyme, any double-stranded DNA that contains the specific restriction site within its sequence is cut by the enzyme at that site. For example, the enzyme *EcoRI* recognizes the restriction site $\updownarrow$ *GAATTC* and the result of cutting by *EcoRI* is shown in the Figure below.

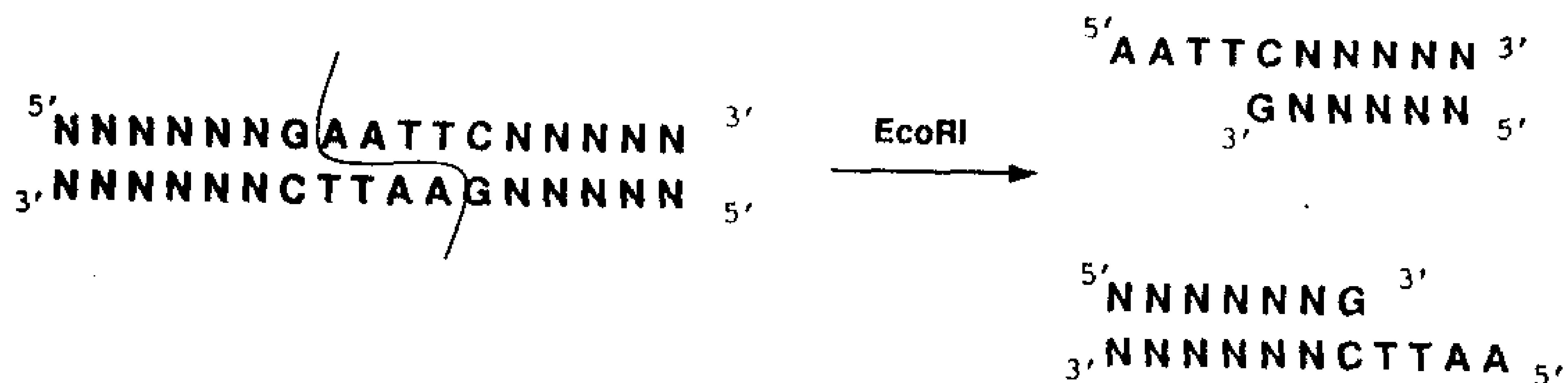


Figure 2.3: Cutting of a DNA strand with the EcoRI restriction enzyme

Another restriction enzyme that we shall be using is *MunI* and it recognizes the site $\updownarrow$ *CAATTG*.

2.2.9 Sequencing of DNA strands

Sequencing is the process of reading out the DNA sequence, nucleotide by nucleotide. This is generally done with the help of PCR and gel electrophoresis, in the following way. For detection of the positions of A's in the target strand, a blocking agent is used that prevents the templates from being extended beyond A's during PCR. As a result of this modified PCR, a population of subsequences is obtained, each corresponding to a different occurrence of A in the original strand. Separating them by length using gel electrophoresis will give away the positions where A occurs in the strand. The process can then be repeated for each of C , G and T, to yield the sequence of the strand. Recent methods use four different fluorescent dyes, one for each base, which allows all four bases to be processed simultaneously.

Chapter 3

Methods and Models of DNA Computation

In this chapter, we briefly look at some of the DNA computation methods and models proposed by various authors till now. In the next chapter, we shall explore the algorithmic self-assembly method as proposed by Winfree [Win98], in detail.

3.1 Adleman's Experiment

Adleman in 1994, tried to solve an instance of the Hamiltonian path problem(HPP) using DNA. A directed graph G with two designated vertices V_{in} and V_{out} is said to have a Hamiltonian path if and only if there exists a sequence of edges $e_1, e_2, ...e_m$ (each adjacent to its previous one), that begins at V_{in} and ends at V_{out} and enters every other vertex exactly once. Adleman choose a small graph of seven nodes with vertex-1(v_1) and vertex-7(v_7) as the start and the end node respectively, and he obtained a Hamiltonian path from v_1 to v_7 by executing the following algorithmic steps :

1. Generate random paths through the graph.
2. Keep only those paths that begin with v_1 and end with v_7.
3. From these selected paths, keep those that contain n vertices (here $n = 7$).
4. From the paths selected above, keep those that enter each of the other vertices (i.e. v_2, v_3, v_4, v_5, and v_6) at least once.
5. If any path remains, answer 'YES', otherwise answer 'NO'.

To implement step-1, every vertex of the graph was encoded by a unique 20-nucleotide sequence and every edge was encoded by a sequence consisting of the complements of - the second-half of the sequence encoding the source vertex and first half of the sequence encoding the target vertex. The sequences representing the various vertices and edges were synthesized and mixed together, so that they annealed to form the various paths through the graph. To implement step-2, the result of step-1 was amplified by PCR, using the sequences for v_1 and v_7 as primers, so that only molecules that encoded paths starting at v_1 and ending at v_7 were amplified. To implement step-3, the gel electrophoresis method was used to separate the molecules by length and keep only those which have the right length

for representing paths of 7 vertices. To implement step-4, a series of affinity purification operations were executed, one for each of the vertices v_2 through v_6. The result of each stage was used as input to the next stage, so that the final result contained only those strands encoding paths passing through each of those vertices. Finally, in step 5, the result of step 4 was amplified and sequenced to detect the presence of a Hamiltonian path. As a result of the sequencing, Adleman was able to verify that the remaining DNA molecule was indeed an encoding of a Hamiltonian path starting at v_1 and ending at v_7.

3.2 Splicing Systems

Splicing systems were first introduced by Tom Head in his seminal paper [Hea87] in 1987, much before the arrival of DNA computing, as we know it. Head proposed a theoretical model based on DNA recombination, which consists of the two operations of cutting DNA strands with restriction enzymes and joining them back with *Ligase*. The mathematical abstraction corressponding to DNA recombination is called the splicing operation and is defined as follows. If R and S are two strings over some alphabet Σ, then the splicing operation on them consists of cutting the strings at specific locations(restriction sites) and concatenating the resulting prefix of R with the suffix of S and similarly, concatenating the prefix of S with the suffix of R. Thus we get two new strings from the two older ones. Head studied the language generation capabilities of the splicing operation. There has been a lot of research on splicing systems in recent years and it has been shown that it can emulate the working of a Turing Machine. This can be intuitively understood by viewing the splicing operation as conversion of an input string (representing a particular configuration of the Turing Machine) to an output string (representing the next configuration) based on some fixed rules. Thus, being Turing equivalent, splicing systems are computationally universal. Though there has been a lot of research on splicing systems, there hasn't been any practical implementation of the system for solving a problem.

3.3 Sticker Model

This model of DNA computation was introduced by Roweis et al.[RWB+96]. In the sticker model, a binary string is represented by a DNA strand called the memory strand, containing specific nucleotide sequences for each of the bit positions and a set of short strands called stickers which have the sequences complementary to the above sequences. A bit is set to 1, if the corresponding bit position in the memory strand has the matching sticker sticking to it. The bit contains 0 otherwise. The computation is done by using the following operations on sets of such binary strands :

1. **Combine:** Two sets of binary strands can be combined by mixing together the contents of the two test-tubes containing them

2. **Separation based on a bit value:** A given set of binary strands can be separated into two new sets - one containing all those original strands where a particular bit is on and the other containing those strands where that bit is off.

3. **Setting a bit:** Setting the value of a particular bit to 1, in all the binary strands of a set. This can be done by adding copies of the sticker sequence for that particular bit.

4. **Resetting a bit:** Reset the value of a particular bit to 0, in all the binary strands of a set. This can be done by extracting out the stickers corresponding to that particular bit.

Adleman et al.[ARR96] have shown how the sticker model can be used to break the DES(Data Encryption Standard) encryption algorithm

3.4 Boolean Circuit Model

Boolean circuits are an important Turing equivalent model of parallel computation. An n-input bounded fan-in Boolean circuit may be viewed as directed acyclic graph, with n input nodes having in-degree zero and the other nodes (called gate nodes) having in-degree at most two. Some of the gate nodes have out-degree zero and are designated as output nodes. Each gate is associated with a boolean function from the set of functions Ω, called the circuit basis. The Boolean circuit has two measures - its size denotes the number of gates present, and its depth denotes the length of the longest path from an input node to an output node. Ogihara and Ray [OR96] proposed a model of DNA computation for simulation of a Boolean circuit. In their model, each gate i is represented by a unique string or DNA sequence, σ_i. The basic structure operated upon is a *tube*, U, which contains strings representing the output of each gate at a particular depth. The presence of the string σ_i in U denotes that the ith gate evaluates to 1, otherwise the output of the ith gate is 0. The initial tube contains strings representing those input nodes having value 1. For each level starting from 0 to k (where k is the depth of the circuit), the gates present at that level are evaluated parallely, as follows. If the gate is an OR gate and any of the strings representing the inputs to the gate are present in U, then the string representing that gate node is added to U. If the gate is an AND gate and both the strings representing the two inputs to the gate are present in U, then the string representing the gate node is added to U. After all the levels have been processed, the strings present in U would represent those output nodes having value 1. Thus, this method simulates a Boolean circuit in $O(k)$ bio-steps, where k is the depth of the circuit.

3.5 Whiplash PCR

Hagiya et al.[HAK+97] came up with a novel method of computing where each DNA molecule acts as a computing machine. The problem is entirely encoded in a DNA sequence and using a series of intra-molecular reactions, the solution is obtained within the same molecule. The process employs a modified form of PCR, where the primer and the template are part of the same strand. The primer forms one end of the strand (the head) and it anneals with its complementary sequence present somewhere in the middle of the strand, to a form a hairpin kind of structure. The primer is then extended by *polymerase* enzyme upto a certain point where the reaction is terminated by a stopper sequence in the template. On heating, the hairpin stretches out into a single strand again, but now the DNA strand contains a new head, which acts as the primer during the next cycle. Thus the method proceeds in a series of thermal cycles (the solution is periodically cooled and heated), during which the strand keeps growing from one end. The process begins with a strand of the following form

$$5' - stopper - new_1 - old_1 - \ldots - stopper - new_n - old_n - head - 3'$$

During each cycle the *head* anneals to the sequence old_i (for some i) and the complement of new_i is synthesized at the end of the strand. The authors have shown that using these operations, it is possible to evaluate a μ-formula (i.e. a boolean formula where each variable is used once only). The authors have also tried to implement the method in the laboratory and they have successfully executed upto two successive steps of the algorithm.

This method of computing has been termed *Whiplash PCR* because the movements of the DNA strand during the procedure resemble the lash of a whip. The Whiplash PCR is a more efficient method of computation than earlier methods and it opens up the possibility of doing "one-pot" computation. But it is not known how the method scales up to large problems. With the increase in problem-size, the length of the DNA strand would have to be larger and this may lead to major complications.

Chapter 4

Algorithmic Self Assembly

The self assembly process utilizes the property that DNA strands having Watson-Crick complementary nucleotide sequences, tend to join or anneal with one-another when they come in contact. When Adleman [Adl94] solved the Hamiltonian Path problem using DNA, he used this annealing property of DNA to create strands of DNA representing various paths through a graph. He created some specially chosen DNA oligonucleotides representing the vertices and edges of the graph, and mixed them together such that they joined with one-another to form all possible paths in the graph. Thus, the oligonucleotides self-assembled to form the paths through the graph. Winfree realized that this process of self-assembly has the power to compute. His idea was to create two-dimensional(2-D) DNA structures which could attach with one-another through complementary DNA strands sticking out of them, to assemble into complex super-structures, and in the process do the computation. This idea was based on the mathematical concept of Wang tiles.

4.1 Wang Tiles

Wang tiles are equal-sized square tiles whose edges are colored by different colors. These were introduced by Hao Wang [Wan61] in 1961, in connection with the tiling problem, which can be stated as follows. Given an infinite supply of a finite number of tile types, is it possible to tile up an infinite plane, subject to the constraint that any two adjacent tiles should have same colored edges facing each-other. It was later shown that the tiling problem was undecidable [Ber66], contrary to what Wang had conjectured. It was also shown that it is possible to construct a set of Wang tiles for emulating the working of any Turing Machine. Thus, Wang tiles have the power to compute. This is illustrated by the example in Figure 4.1 (taken from [Win00]), where a set of seven tiles are used to implement a binary counter.

4.2 DNA Tiles

Winfree's idea was to construct biological tiles made of DNA with unpaired strands sticking out such that only tiles with matching sticky ends (i.e the unpaired single strands sticking out) can attach with one-another. This would emulate the condition on Wang tiles that only tiles with matching edges can be placed adjacent to each-other. Thus, once the appropriate DNA tiles are constructed, under proper conditions, the tiles would self-assemble and do

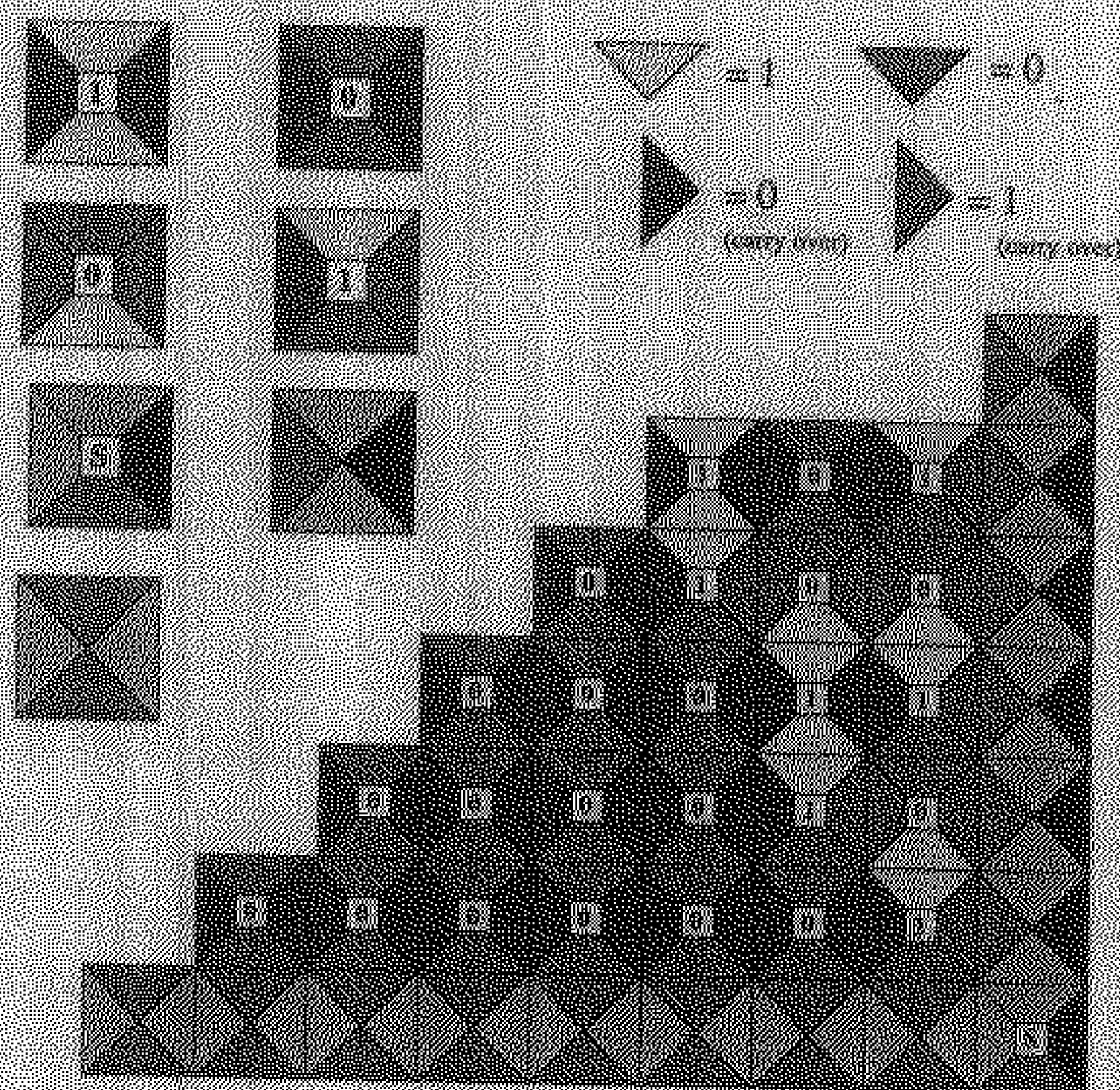


Figure 4.1: Wang Tiles: Implementing a binary counter

the computation automatically. This process has been termed as *Algorithmic Self-Assembly* by Winfree.

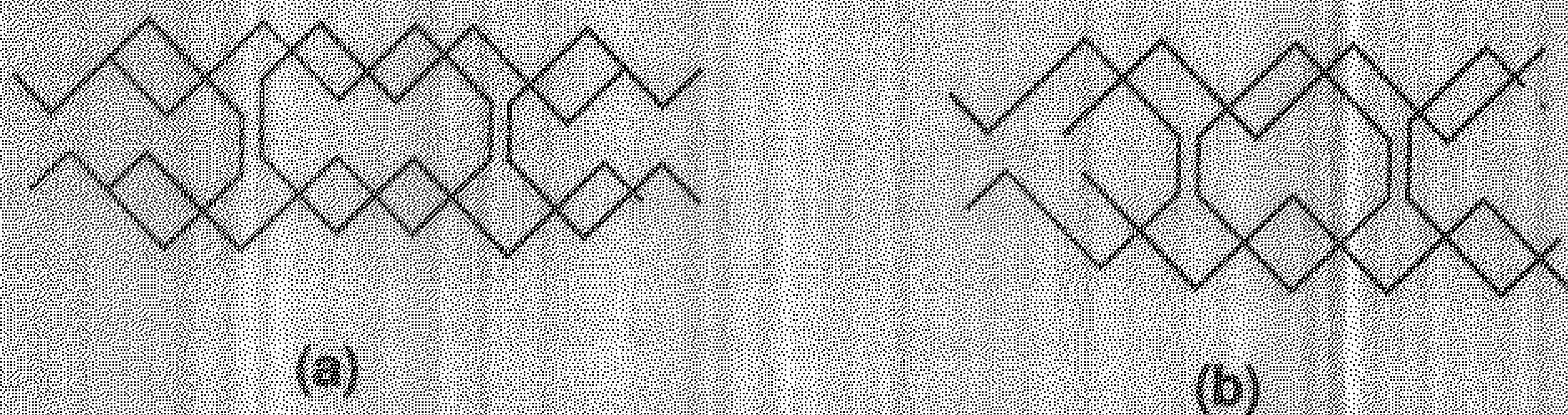


Figure 4.2: 2-D DNA Tiles: (a)DAO tile (b)DAE tile

In order to assemble properly over a plane, the DNA tiles have to be rigid and planar . Winfree and N. Seeman constructed some two dimensional DNA tiles, as shown in Figure 4.2. These are called antiparallel double crossover(DX) tiles because they consist of two DNA double strands placed side by side (in opposite directions) that are linked by two crossover junctions. This gives them a rigid planar structure which makes them ideal for using as biological Wang tiles. Two types of antiparallel DX tiles are shown in the figure, called DAO(double crossover antiparallel with odd spacing) and DAE (double crossover antiparallel with even spacing). These two tiles were found to be the most stable of the DX tiles. Notice that each of these tiles have four unpaired strands (called sticky ends) sticking out of its four corners, using which other tiles can attach to it. This kind of assembly of tiles can tile-up a plane, in the same way as Wang tiles. Winfree used these DAE tiles to implement a formal model of computation called the blocked cellular automata (BCA) and hence showed that computation by self-assembly is universal.

4.3 Blocked Cellular Automata(BCA)

The Blocked Cellular Automata is a formal model of computation, where the information is stored in an infinite one-dimensional tape(as in a Turing machine) with each cell containing any one of the symbols from a finite set of symbols(the alphabet). The computation occurs in steps and in each step, the whole tape is translated into a new tape, according to a finite set of rules called the rule table. The translation occurs locally and in parallel, such that each pair of cells translates into a new pair of cells (simultaneously), according to one of the rules from the rule table. The pairing of cells alternates from step to step. The rules are of the form $\{(x, y) \rightarrow (u, v)\}$ where $x, y, u, v \in \Sigma$, the alphabet. The rule table corresponds to a program for the BCA. The output or answer provided by the BCA is the contents of the final tape, at the time the computation ends. There are two conventions for determining the end of the computation - one is when the tape stops changing, the computation is assumed to be complete; the other approach takes the computation to be complete only when a special symbol called the *Halting Symbol* is written for the first time anywhere on the tape.

A BCA can simulate the operations of a Turing Machine, thus BCA are computationally universal. But, in a practical implementation of the BCA, the tapes cannot be infinite. So we have to use a finite tape. The finite tape BCA can also be computationally universal, depending on the boundary conditions, i.e. how the unpaired cells at the boundary are handled. One possible solution is to directly copy the unpaired cell into the new tape (this model is not computationally universal). Another approach is to ignore the unpaired cell at the boundary - this will lead to the tape getting smaller and smaller with each step and the computation would not be able to proceed after a few steps. The solution to this problem is to use a fixed pattern of symbols to pad the input from each side, whenever the tape gets too short. This model is computationally universal. There is another possible approach which is also universal, that is to let the rule set decide the boundary condition; thus depending on what symbol is in the boundary cell, it may either be ignored (making the tape shrink) or a new cell may be added at the end (making the tape expand).

4.4 Implementing BCA using DNA Self Assembly

Winfree implemented the BCA by constructing special DNA structures for representing the tape and the rule set of the BCA. The initial tape was represented by the DNA structure shown in Figure 4.3(a) Here, the sticky ends at the top represent the symbols written on the tape. Each symbol x in the alphabet is represented by a unique nucleotide sequence $D(x)$ and the sticky end representing the symbol contains the same nucleotide sequence. Each rule $\{(x, y) \rightarrow (u, v)\}$ is represented by a DAE tile whose sticky ends on the lower helix are $\overline{D}(x)$ and $\overline{D}(y)$ (the complementary sequences to $D(x)$ and $D(y)$), while the sticky ends on the upper helix are $D(u)$ and $D(v)$. Such a rule tile would attach to the slot in the tape structure where the sticky ends are $D(x)$ and $D(y)$. When all the rule tile attach to the specific locations, one step of the BCA would be completed and at that stage the top layer consisting of the top parts of these tiles would represent the new tape (Figure 4.3(b)). In this way each step of the BCA would be computed.

The completion of the computation would be signaled by the special halting symbol being written to the tape, which in this case would correspond to a special sticky end motif being incorporated into the lattice. Winfree suggests that this special motif be chosen as the recognition site for a binding protein, which could subsequently catalyse a phosphorescent

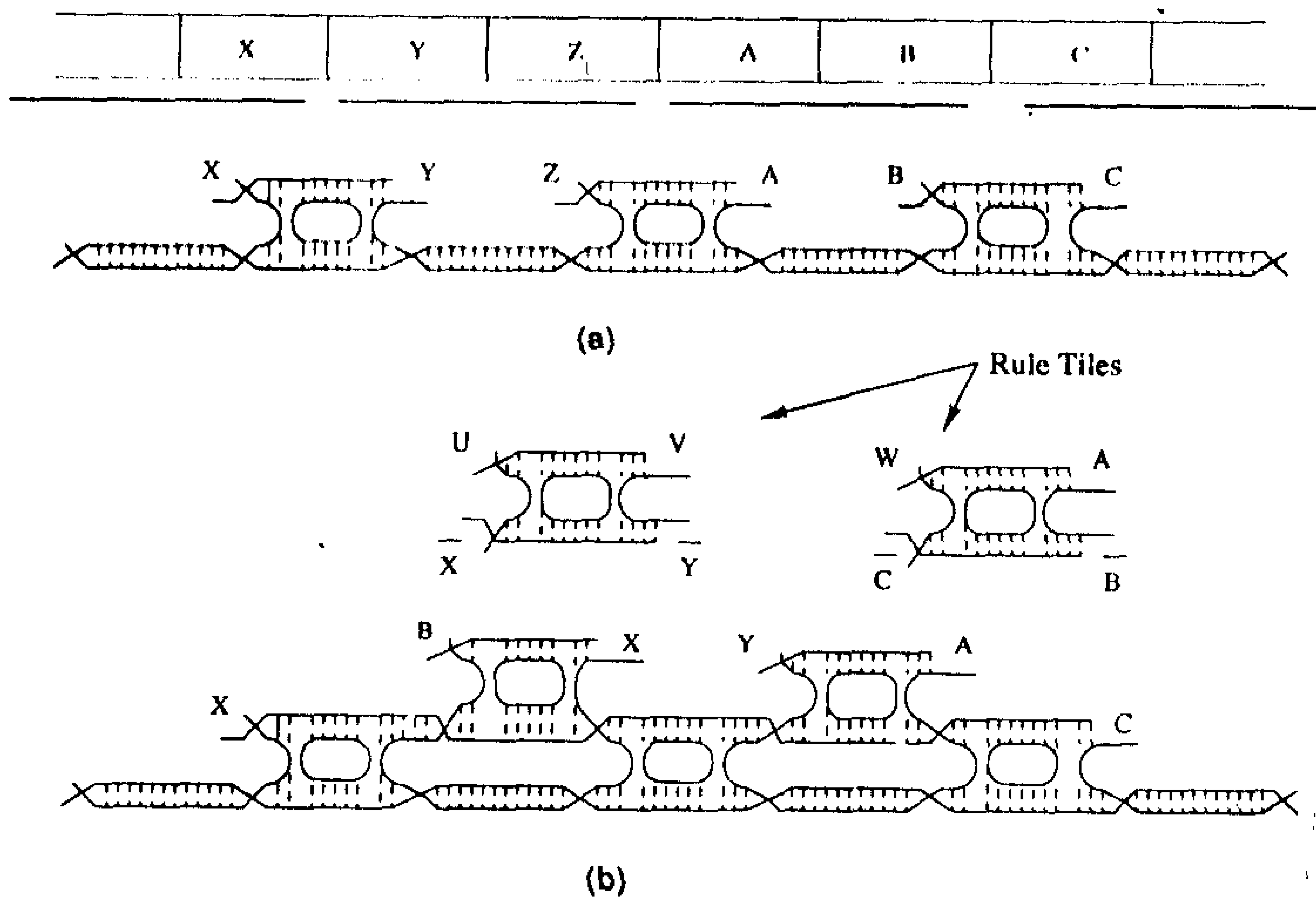


Figure 4.3: DNA Implementation of BCA : (a) The initial tape (b) The rule tiles join on the initial tape structure to form the new tape for the next step

reaction, turning the solution blue. This would signal the end of the computation. At this stage the top layer would contain the symbols representing the result of the computation. To obtain this result, first *Ligase* is added to the solution to bind all the annealed segments together. Thus there would now be a single strand passing through the top layer of the assembly that contains the result. Once the solution is heated to break the hydrogen bonds and separate the strands, the result strand can be extracted out using the affinity purification method (see chapter 2). The result is obtained by sequencing this strand.

The above process shows how the DNA self-assembly process can be used to implement the blocked cellular automata. As stated earlier, the BCA can emulate a Turing Machine and thus is computationally universal. This means that the algorithmic self-assembly of DNA is also computationally universal. Thus, theoretically at least, any program that can be run on an electronic computer or any other computing model, can also be implemented using the algorithmic self-assembly of DNA.

Chapter 5

Finite Field Arithmetic

In the previous few chapters, we have looked at different methods of implementing computations using DNA. Most of the proposed methods try to implement the solutions to some of the well-known problems in computing. But, in order to do general purpose computing using DNA, we should have a system, where any algorithm or program can be executed. For that purpose we have to implement the basic operations that any program or algorithm is made up of. These are the arithmetic and logical operations.

There have many attempts at implementing the basic binary arithmetic and logical operations, using DNA computing. See, for example [GFB96], [GPZ97], [MLRS00] and [BM02]. But till date no attempts have been made to use DNA computing for finite field arithmetic. Finite field arithmetic has applications in cryptography, coding theory and other fields and it is of considerable interest to obtain a method for fast computation of finite field multiplication, in particular. By using DNA computing for finite field computation, we should be able to parallely execute multiple computations at a very low cost. In this chapter we describe a method for implementing finite field arithmetic operations using DNA.

5.1 Finite fields

A finite field is a field with a finite number of elements. For each prime p and each natural number n, there exists a unique finite field of p^n elements denoted by $GF(p^n)$. The elements of this field can be represented as polynomials in x, of degree less than n and having coefficients from $Z_p = \{1, 2, ...p-1\}$. We shall be concerned only with finite fields of the form $GF(2^n)$, whose elements can be represented as binary numbers by taking the coefficients of x^{n-1} through x^0 in their polynomial representation. The addition of two such elements of $GF(2^n)$ is equivalent to the bit-wise XORing of the corresponding binary numbers. The multiplication operation in $GF(2^n)$ is similar to polynomial multiplication except that if the result is of degree n or more, then we cut it down to less than n, by going modulo a fixed irreducible polynomial R, of degree n, in $GF_2[x]$ (i.e. having coefficients from Z_2).

We shall try to give a DNA algorithm for computing finite field arithmetic, by extending the method for computing the usual binary arithmetic that was proposed by Barua and Misra [BM02].

5.2 Binary Arithmetic using DNA

This was the method proposed by [BM02] for binary arithmetic. Here, each binary number is represented by the set of integers denoting the positions where the bit value is 1. The DNA representation of the number consists of a test-tube containing DNA double strands of the form

$$ds_i = \updownarrow S_0(GAATTGC^5)^i GAATTC$$

for each integer i (i.e each position i having bit value 1)
For the binary number $a = a_n...a_2a_1$, the abstract representation is

$$X[a] = \{i : a_i = 1\}$$

The addition of two numbers a and b is computed recursively as

$$Add(X[a], X[b]) = \begin{cases} X[a] & \text{if } X[b] = \phi \\ X[b] & \text{if } X[a] = \phi \\ Add((X[a] \oplus X[b]), (X[a] \cap X[b])+) & \text{otherwise} \end{cases}$$

where $X \oplus Y = \{x : x \in X \cup Y and x \notin X \cap Y\}$ and $Z+ = \{x + 1 : x \in Z\}$ (and $Z + i = \{x + i : x \in Z\}$)

The authors have shown that this process always terminates and computes the result of the addition of the two numbers. To implement the method, we have to construct test-tubes representing $(X[a] \oplus X[b])$ and $(X[a] \cap X[b])+$, given two test-tubes representing X[a] and X[b], denoted by T[a] and T[b]. This is done by first melting double strands in T[a] and T[b] and then, extracting the up strands from T[a] (using $\downarrow S_0$)and down strands from T[b] (using $\uparrow S_0$). These extracted strands are mixed together and allowed to anneal. As a result we get single strands from $(X[a] \oplus X[b])$ and double strands from $(X[a] \cap X[b])$. The single strands and double strands are separated into two test-tubes labeled T[a] and T[b]. The single strands in T[a] are converted to double strands using the PCR reaction with S_0 as the primer. Thus, T[a] now represents $(X[a] \oplus X[b])$. The double strands in T[b] are cut by the restriction enzyme *EcoRI* to obtain double strands with a hanging 5'end of the form

$$\updownarrow S_0(GAATTGC^5)^i G \downarrow AATT$$

(The restriction enzyme *EcoRI* cuts at the site GAATTC) Now the up strands

$$\uparrow AATTGC^5GAATTC$$

and the *Ligase* enzyme are added to obtain double strands of the form

$$\updownarrow S_0(GAATTGC^5)^i GAATT \uparrow GC^5GAATTC$$

These strands are then polymerized to form strands of the form

$$\updownarrow S_0(GAATTGC^5)^i GAATTGC^5GAATTC$$

Thus the strands in the test-tube T[b] now represent $(X[a] \cap X[b])+$. This process can be repeated untill one of T[a] or T[b] becomes empty. At that time the process terminates and we get the result of the addition.

Let us see how we can extend this method for computing finite field arithmetic.

5.3 Finite Field Addition

As stated earlier, we would be working on finite fields of the form $GF(2^n)$ only. Let $a = a_{n-1}...a_1a_0$ and $b = b_{n-1}...b_1b_0$ be two elements of $GF(2^n)$, then their addition can be calculated by performing a bit-wise XOR. Thus if X[a] and X[b] be the representation of the two numbers, as above, then the result of the addition is given by $(X[a] \oplus X[b])$, which can be calculated by the above procedure, in just two biochemical steps.

5.4 Finite Field Multiplication

The procedure for finite field multiplication would be a little more complicated. As before, let $a = a_{n-1}...a_1a_0$ and $b = b_{n-1}...b_1b_0$ be the two n bit numbers to be multiplied, represented by X[a] and X[b] respectively.(where X[w] is defined as above) In polynomial notation we can write the numbers as
$a = a_{n-1} * x^{n-1} + ... + a_1 * x + a_0$ and
$b = b_{n-1} * x^{n-1} + ... + b_1 * x + b_0$.
Let $r = x^n + r_{n-1} * x^{n-1} + ... + r_1 * x + r_0$ be the fixed irreducible polynomial used for the field operations.
The result of multiplication of a and b, say c, is given by

$$c = (a_{n-1} * x^{n-1} + ... + a_1 * x + a_0)(b_{n-1} * x^{n-1} + ... + b_1 * x + b_0)(modulo\ r)$$

Here, $(a_{n-1} * x^{n-1} + ... + a_1 * x + a_0)(b_{n-1} * x^{n-1} + ... + b_1 * x + b_0) = \sum_{b_i=1}(a_{n-1} * x^{n-1} + ... + a_1 * x + a_0) * x^i$ which can be represented as

$$\sum_{b_i=1}(X[a] + i)$$

To compute the result c, we create test-tube T[c] representing it, from the test-tubes T[a] and T[b] representing a and b respectively, by the following procedure. Here, the DNA representation of the numbers would be slightly different. We would use the double strand

$$ds_i = \updownarrow S_0CAATTGC^5(GAATTGC^5)^{i-1}GAATTC$$

for each integer i, denoting the bit positions set to 1.(For position 0, we use $\updownarrow S_0GAATTC$) We execute the following steps to compute the multiplication.

1. The DNA strands in T[a] are duplicated to make n copies of T[a], labeled as T_1 to T_n. To each of the test-tubes, the restriction enzyme *EcoRI* is added to form DNA double strands of the form

   $$\updownarrow S_0CAATTGC^5(GAATTGC^5)^{i-1}G \downarrow AATT$$

   with a hanging 5' end.

2. For each double strand ds_i in T[b] do the following :
   Make n copies of the strand and add the restriction enzyme *MunI* to the strands to form

   $$\uparrow AATT \updownarrow GC^5(GAATTGC^5)^{i-1}GAATTC$$

(Note : The restriction enzyme *MunI* cuts at the site CAATTG) These double strands with hanging 5'end are added to one of the test-tubes from the above step. On allowing the strands to anneal and then adding *Ligase*, we get strands representing X[a]+i in the test-tube.

3. After executing Step2 for each double strand in T[b], we would get test-tubes representing (X[a]+i), for each position i in b where b_i is set to 1. We divide the test-tubes in pairs and execute the $\oplus$ operation on them. The test-tubes obtained as result are again divided into pairs and the same operation is repeated. This process is continued till we obtain a single test-tube T_0. This would be the DNA representation of

$$\sum_{b_i=1}(X[a]+i)$$

4. We construct the test-tubes $Tr_j = \{ds_i + j : r_i = 1\}$ for $j = 0$ to $j = n - 2$. Now for each i from 2n-2 to n, we try to extract the DNA strand ds_i from the test-tube T_0. If the strand ds_i is found in T_0, we compute $T_0 \oplus Tr_{i-n}$ and label the new test-tube as T_0. At the end of this step, the strands present in T_0 would represent the result of the multiplication operation. Thus this would be the result T[c].

5.5 Analysis of the method

Let us see how many bio-steps are required to implement the above multiplication process. If we assume that making n copies of a double strand takes $log(n)$ bio-operations (by duplicating each time) then, Step1 takes log(n)+1 bio-steps and Step2 takes log(n)+2 bio-steps. Step3 can take a maximum of 2*log(n) bio-steps, assuming each $\oplus$ operation to be of two bio-steps. Step4 can take a maximum of 3*(n-1) bio-steps by the same logic. So, the multiplication of n-bit numbers by this method takes 0(n) bio-steps. Thus, the time complexity of the method is not good, considering the fact that each bio-step takes a large amount of time to implement.

The advantage of the method is that the result test-tube, in case of both addition and multiplication can be used in further computations, because it has the same representation as the numbers that were used as input. If the result is to be reported, that can also be done very easily by length-based separation of the result strands using the gel electrophoresis method.

Despite these advantages, this method of computing finite field multiplication is quite time consuming and rather clumsy. Further, there are many chances of error, as the process relies heavily on the extraction operation which is error-prone. Thus, we are forced to look for a more elegant and efficient method of computing finite field arithmetic using DNA. We will describe one such method based on self-assembly, in the next chapter.

Chapter 6

Finite Field Arithmetic using Self-Assembly

One of the limitations of DNA computing is that each of the bio-step in a DNA algorithm is manually executed in the laboratory, taking a lot of time. In DNA self-assembly, most of the computation is done by the automatic assembly of the DNA tiles and the only time consuming operations are the initial step of creating the tiles and the final step of extracting and reading the result. Thus the self-assembly method is more efficient than the traditional DNA computing techniques, that were used in the previous chapter to compute finite field arithmetic operations. In this chapter, we show how the process of Algorithmic self-assembly of DNA tiles can be used for computing the finite field arithmetic operations.

6.1 DNA TX Tiles

In chapter four, we had explained the algorithmic self-assembly process and introduced the DNA double crossover(DX) tiles. Here we look at some more complicated DNA tiles called Triple Crossover or TX tiles that contain three double helixes of DNA intertwined with one-another. These tiles are more stable and rigid, and can have upto six sticky ends through which they can attach with other tiles. We will look at three different types of TX tiles, which we will be using :

- TAO tiles - These tiles have the structure shown in Figure 6.1(a). These tiles are formed by the annealing of four DNA single strands(shown in different colors) and they have four sticky ends at the four corners. Notice the strand (colored Green) passing from the bottom left to the top right of the tile. When TAO tiles join together diagonally (see Figure 6.1(b)), and we seal the joins by applying *Ligase*, we have a single DNA strand passing through the assembly from bottom left to top right (shown in bold). This property of the TAO tiles is important and will be made use of in our procedure.
- TAE tiles - The structure of these tiles is as shown in Figure 6.2(a). Here we have six DNA strands forming the tile and there are six sticky ends, three on the left and three on the right. However if all the six sticky ends are not needed, then we can bend some of these into loops so that they are not used as sticky ends.

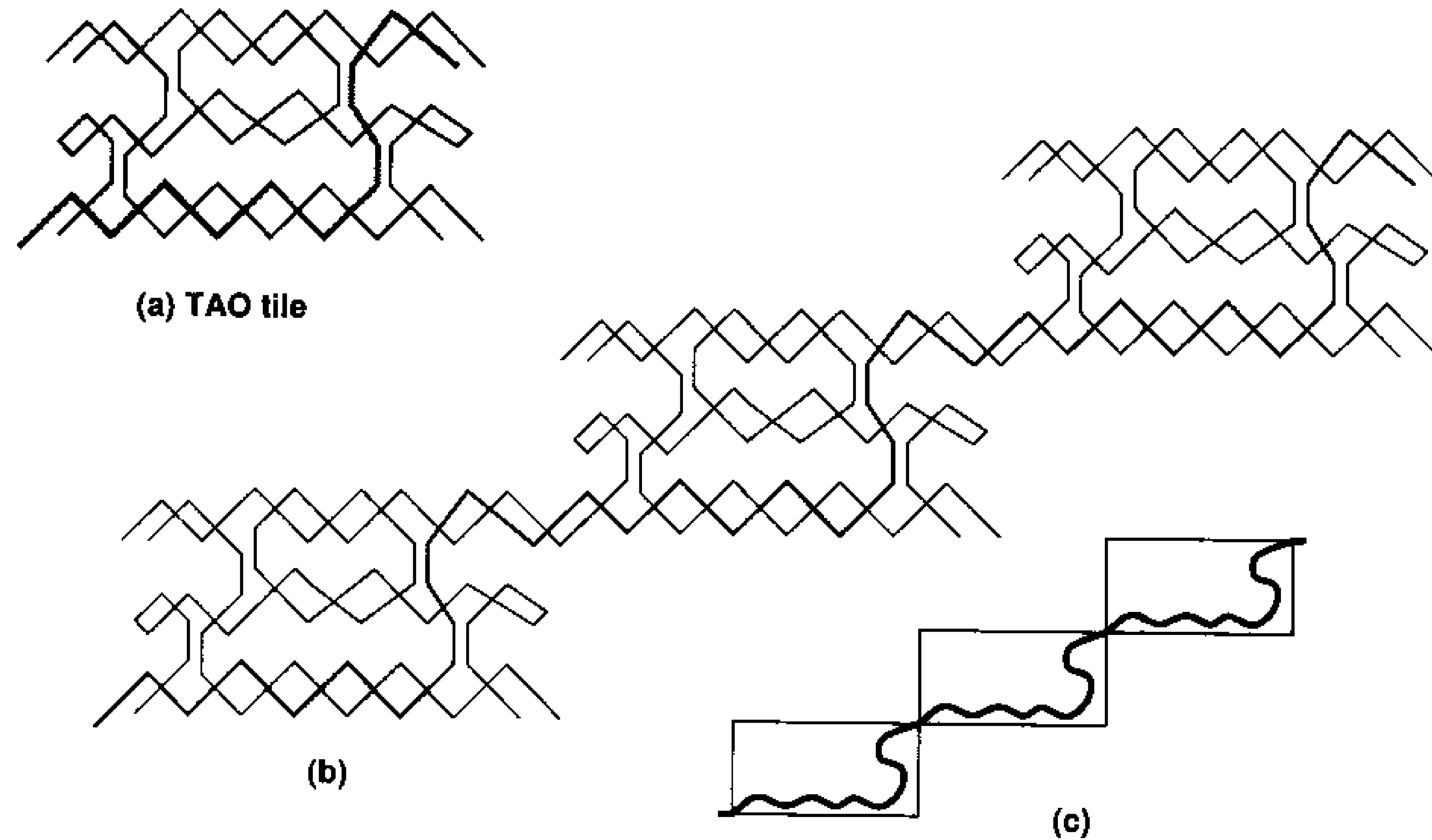


Figure 6.1: TAO tiles : (a) Structure of a single TAO tile (b) Three TAO tiles join diagonally (c) A single DNA strand runs through the three tiles.

- Rotated TX tiles - The tile in Figure 6.2(b) has only two sticky ends. These two sticky ends are so designed that when these attach with the sticky ends of two neighboring TAE tiles, the tile gets rotated (by an angle close to 90°) relative to the plane of the TAE tiles. This rotated tile can fit in the small gap left in the center when four TAE tiles attach together. So, the TAE tiles and these rotated tiles can assemble to form a closely packed two-dimensional lattice structure as shown in Figure 6.2(d). Notice that when the rotated tile joins with two TAE tiles (Figure 6.2(c)), a single DNA strand (shown in bold) passes through the middle of the three tiles.

These tiles can be constructed using a variety of possible nucleotide sequences. We can use different sequences to denote different symbols or values. For example, we can have one sequence denoting the value 0 and another sequence denoting the value 1. So tiles constructed using different sequences can store different values or symbols. We can also use the sticky ends of a tile to encode certain values or symbols. The tile that attaches to this tile would have the complementary sticky end encoding the same value or symbol. In this way, we can pass information from one tile to its adjoining tile.

6.2 Finite Field Arithmetic

Recall that the elements of the finite field $GF(2^n)$ are represented as polynomials in x, of degree less than n and having coefficients from $Z_2 = \{1, 2\}$. An alternate representation is in the form of a binary numbers obtained by taking the coefficients of x^{n-1} through x^0 in the polynomial representation. The addition of two such elements of $GF(2^n)$ is equivalent to the bit-wise XORing of the corresponding binary numbers. The multiplication operation in $GF(2^n)$ is similar to polynomial multiplication except that if the result is of degree n or more, then we cut it down to less than n, by going modulo a fixed irreducible polynomial R, of degree n, in $GF_2[x]$ (i.e. having coefficients from Z_2). As the addition operation is quite simple, we will first concentrate on the multiplication operation, which is the difficult

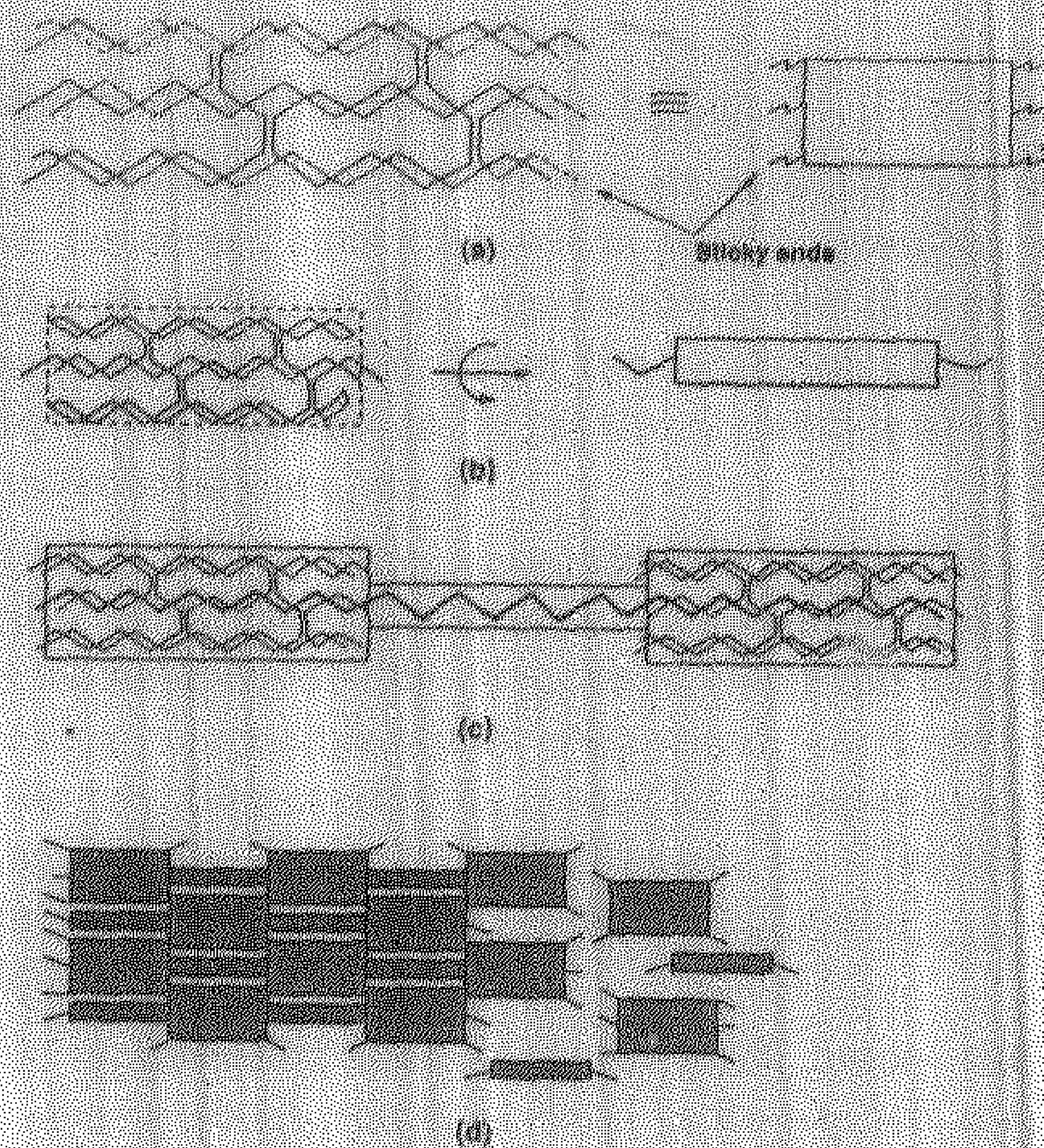


Figure 6.2: TAE Tiles : (a) Structure of the TAE tile (b) Rotated TX tile (c) The rotated tile joins with two TAE tiles on its either side (d) The two types of tiles can assemble to form a compact lattice structure.

part. Let us first see how the finite field multiplication operation can be implemented using self-assembly of DNA tiles.

6.2.1 Finite Field Multiplication

Suppose we want to multiply two numbers A and B in $GF(2^n)$ going modulo R, where
$A = (a_{n-1} * x^{n-1} + ... + a_1 * x + a_0)$,
$B = (b_{n-1} * x^{n-1} + ... + b_1 * x + b_0)$, and
$R = (x^n + r_{n-1} * x^{n-1} + ... + r_1 * x + r_0)$.
Then the result of the multiplication, C is given by:

$$C = \sum a_i * B * x^i (moduloR)$$

or,

$$C = \sum_{i \in \{0..n-1\}, a_i = 1} [B * x^i (moduloR)]$$

Note that, here the addition of the coefficients is done modulo 2 . We will denote $(B * x^i (moduloR))$ by B_i.
The basic operation here, is computing $(B * x(moduloR))$ from a given number B. We have

$$B * x(moduloR) = (b_{n-2} * x^{n-1} + ... + b_1 * x^2 + b_0 * x)$$

$$+ y * (r_{n-1} * x^{n-1} + ... + r_1 * x + r_0)$$

where $y = b_{n-1}$.
So, the i-th bit, i.e. the coefficient of x^i is given by $(b_{i-1} + y * r_i)$.

Suppose we represent the number B as a string of DNA tiles, with each tile storing one bit of B. We can construct such tiles which would assemble on top of this sequence of tiles to form another sequence of tiles representing the bits of $B_1 = B * x(moduloR)$. This operation is shown in Figure 6.3. Here the labels on the sticky ends represent the information carried by them. We denote by $\lfloor x \rfloor$ the nucleotide sequence encoding the value x and by $\lceil x \rceil$ the complementary sequence.

Notice the tiles (with six sticky ends) floating above the tile assembly for B. The sticky ends of these tiles are labeled by variables x,y,z and r of which x,y and r can take any value from $\{0, 1\}$. So we have eight different tiles of this type for various values of x,y and r. Only those tiles having compatible sticky ends could join together. Thus the tile which assembles on top of the two tiles storing b_{n-1} and b_{n-2} respectively, would have $x = b_{n-2}$, $r = r_{n-1}$ and the same y value as the 'M' tile adjacent to it (which is equal to b_{n-1}). This tile then computes the value $x + y * r = b_{n-2} + b_{n-1} * r_{n-1} = b^1_{n-1}$, which is the (n-1)th bit of B_1 (see Figure 6.3(b)).

Thus, the tile representing the (i-1)th bit of B, passes the value of the bit to the i-th tile in the upper layer through the sticky end in its top left corner. The i-th bit tile in the upper layer gets the value of the (i-1)th bit, b_{i-1} and computes the value $(b_{i-1} + y * r_i)$ of the i-th bit of B_1. The value of y is passed to each tile from the tile on its left, through the sticky end in the middle. The value of y is initially obtained from the leftmost tile of the B sequence as b_{n-1}, and is passed to the upper layer through the tile labeled 'M'.

On repeating this simple operation, we can get $B_2 = B * x^2(moduloR)$, on top of B_1 and $B_3 = B * x^3(moduloR)$ on top of B_2 and so on. Thus, we would have an assembly of multiple layers of DNA tiles, where the i-th row would represent B_i. Now we only have to perform an addition of all such B_i's for which $a_i = 1$. For this we maintain a partial sum, S_i at each layer,i such that

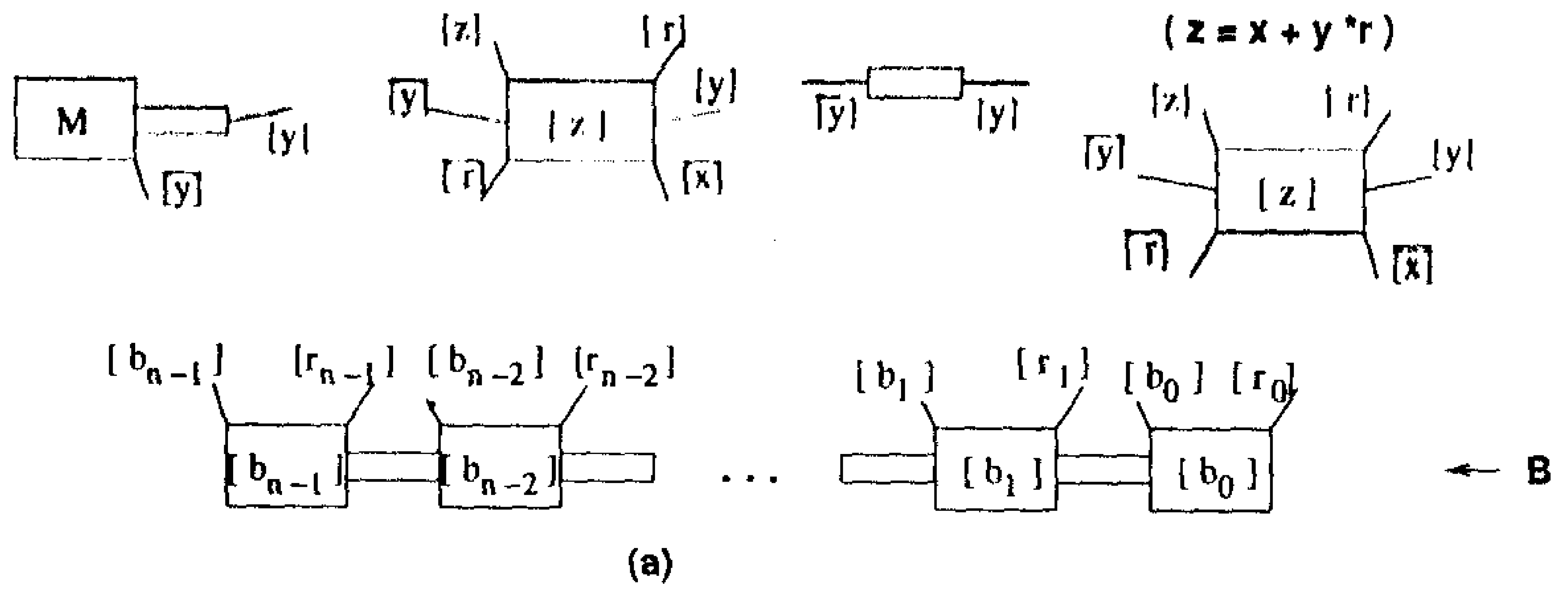
$$S_{-1} = 0$$

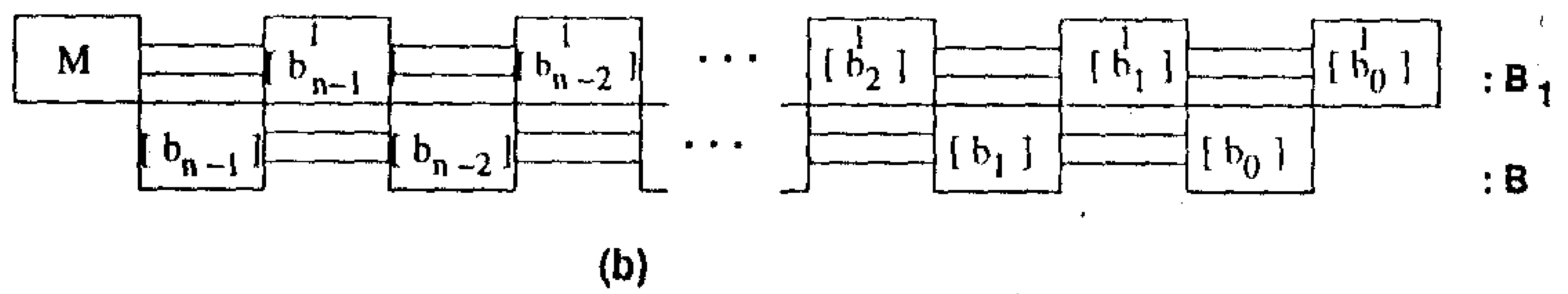
and

$$S_i = S_{i-1} + a_i * B_i$$

The bits of S_{i-1} are passed to the i-th layer through the bottom left sticky end of the tiles, along with the r value. But, in order to calculate the corresponding bits of S_i, the value of a_i should be available. The solution is to construct a diagonal tile assembly representing the bits of A (as shown in Figure 6.3(c)), and allow this to join to the left side of our earlier assembly such that the tile storing the i-th bit of A is attached to the i-th row. The value of a_i would be passed along the row, through the sticky ends in the middle, along with the y value.

This is done using the tiles shown in Figure 6.4. Here, $H(x)$,$G(w,r)$ and $Q(y,m)$ denote nucleotide sequences encoding the value x, w and r, and y and m respectively. The tile shown in Figure 6.4(c) is the computation tile which computes the bits of B_i as explained before. But, it also computes the partial sum bits as w', using the value of partial sum bit w that it receives from the lower layer through the sticky end at its bottom left. The value of y and $m = a_i$ are passed from one tile to another in the same row, using the bridge tiles shown in Figure 6.4(d). The value of a_i is initially passed to the layer representing B_i, through the tile labeled 'M1' shown in Figure 6.4(b), to which the ith bit tile of A would attach. Finally the tiles shown in Figure 6.4(e) are used for adding a zero at the end, while computing B_i from B_{i-1}.





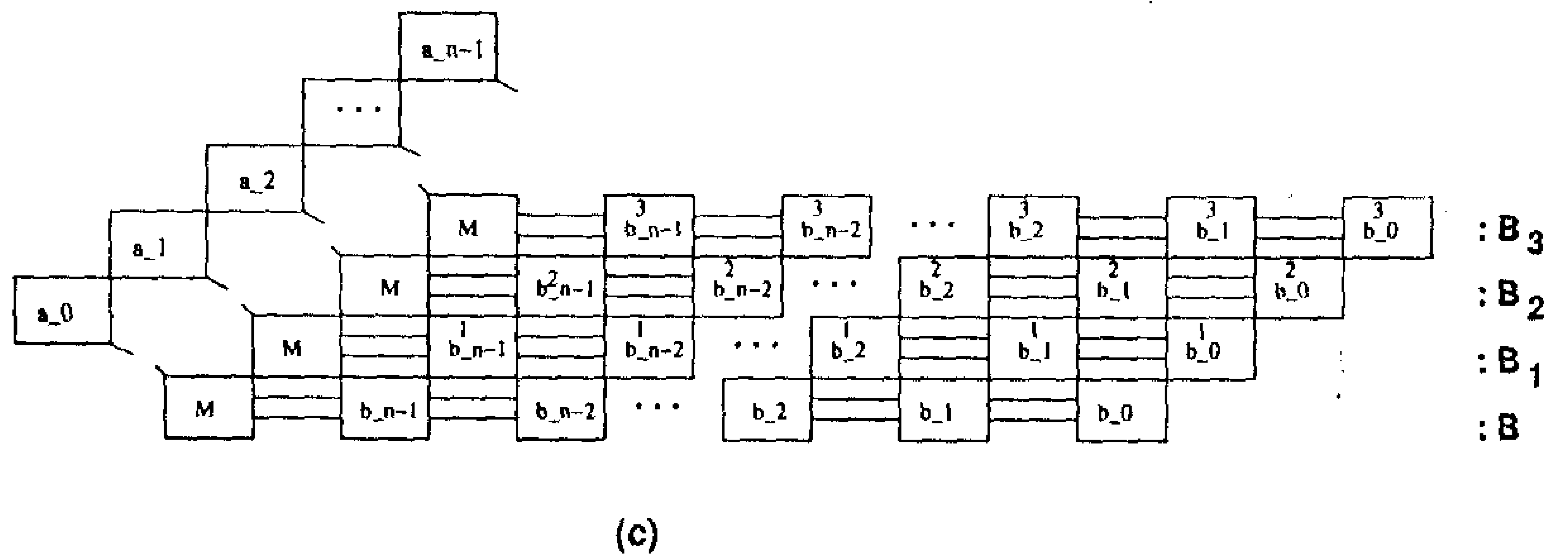


Figure 6.3: Implementing the basic step of computing $B * x(moduloR)$ from B: (a) Tiles representing the bits of B with other computation tiles hanging above (b) The tiles assemble to form $B * x(moduloR)$ over B (c) After multiple iterations of the basic step, we get each B_i. The tile structure for A is shown alongside.

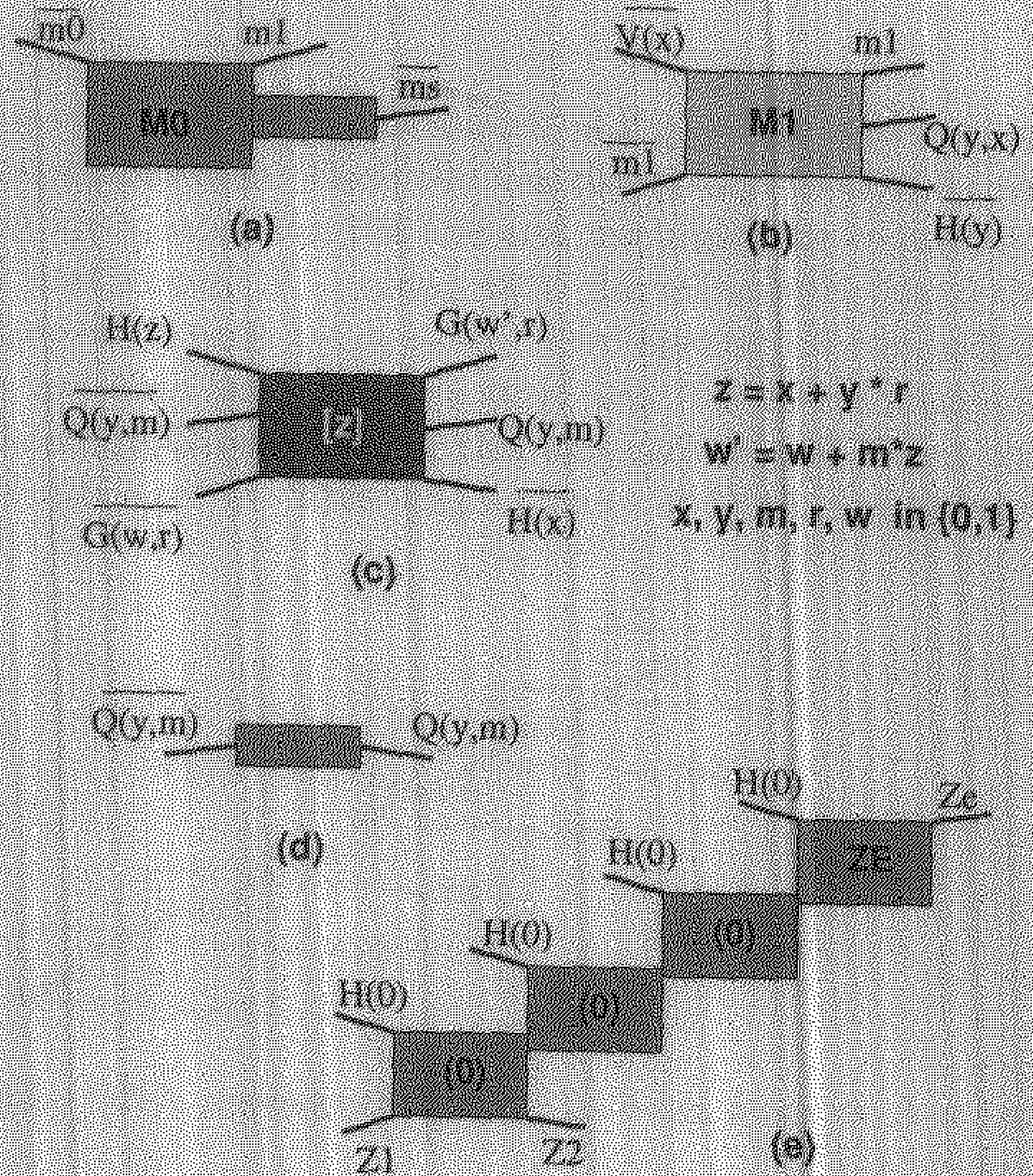


Figure 6.4: DNA Tiles required for computing finite field multiplication. The tile shown in (d) is a rotated TX tile and all the other tiles are TAE tiles.

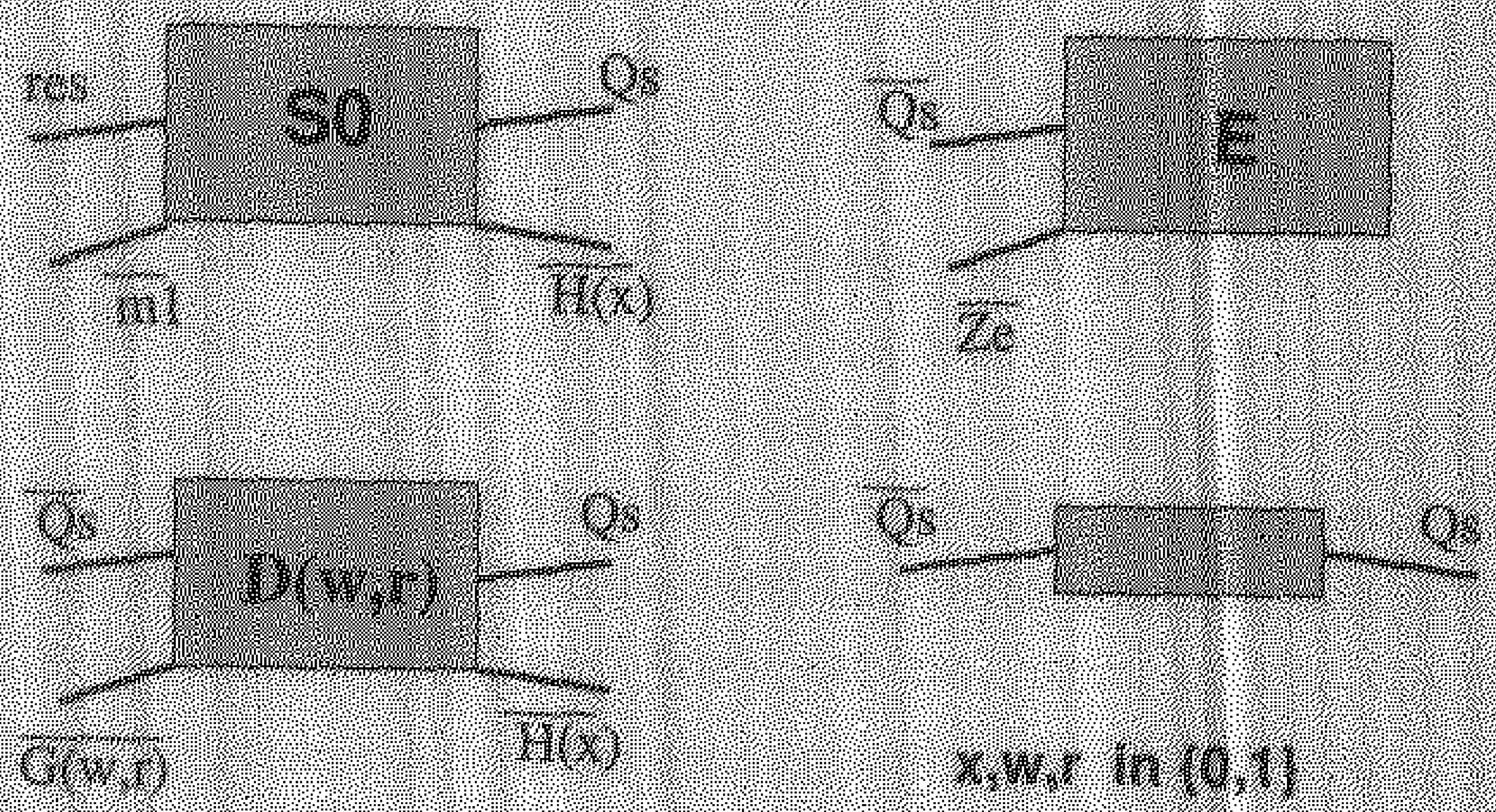


Figure 6.5: Output Tiles: These tiles are used to store the result of the computation

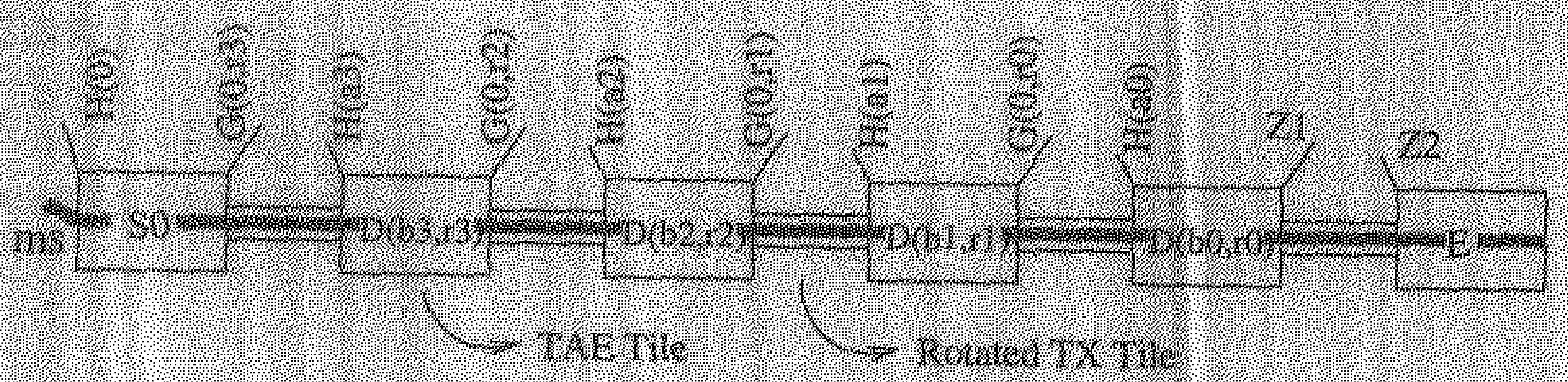


Figure 6.6: Input Tiles-II: DNA tile structure representing the 4-bit number $B = (b3, b2, b1, b0)$

The tile structure for B would be as shown in Figure 6.6 and this consists of TAE tiles with rotated TX tiles in between. Each tile has two sticky ends at its top, the left one encoding the value of the bit b_i stored in the tile and the right one encoding the value 0 as the i-th bit of the initial partial sum S_{-1}, along with the value of r_i (the i-th bit of R). So, the r_i value is stored along with each bit b_i of B. Two special tiles encoding the symbols 'S0' and 'E' respectively, are used to denote the start and the end of number. Note that, a single strand of DNA (shown in Green), passes through each tile of B and thus contains an encoding $D(b_i, r_i)$ of each bit b_i of B. This single strand of DNA can be used to represent the number B.

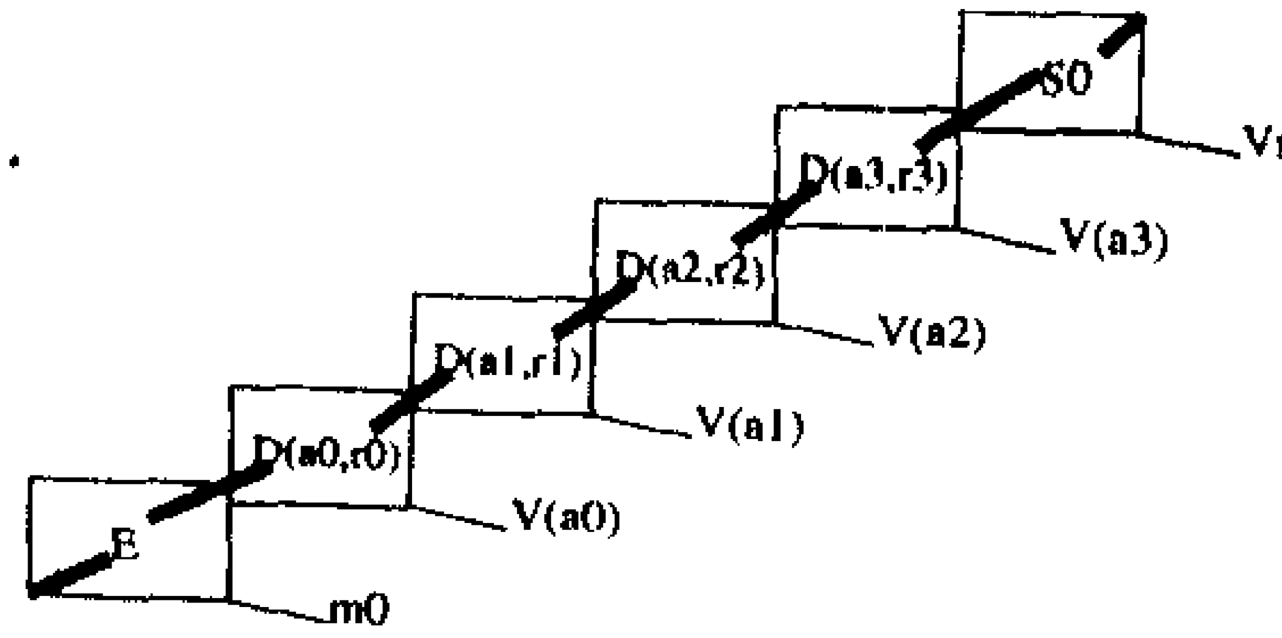


Figure 6.7: Input Tiles-I : DNA representation of the 4-bit number $A = (a3, a2, a1, a0)$

Figure 6.8: ssDNA representation of the n-bit number $A = (a_{n-1}...a_1a_0)$

Figure 6.7 shows the tile structure for A. Here each tile has only one sticky end and it encodes the value of the bit stored in the tile. This tile structure consists of TAO tiles and as explained earlier, a single strand of DNA passes diagonally through the structure. This single strand passes through each of the tiles and encodes the bit value stored in each tile. Note that this encoding is the same as that used for representing the number B.(Here, the r-bits are retained only for uniformity, and they are not used in the computation).

So, we have a unique representation for each number in $GF(2^n)$ as a single stranded DNA molecule as shown in Figure 6.8. This is the form in which we would be storing all numbers, in our system.

6.2.2 An Example

Let us now look at an example computation of finite field multiplication using our method. For simplicity we will consider four bit numbers, or elements of $GF(2^4)$. We take,

$$A = x^3 + x$$

or, $A = (1010)$

$$B = x^3 + x^2 + 1$$

or, $B = (1101)$ and,

$$R = x^4 + x + 1$$

The numbers A and B, which are in the form of ssDNA (as in Figure 6.8) are converted to appropriate tile structures as in Figure 6.7 and Figure 6.6 respectively. Now, we add the other tiles (Figure 6.4) required for computation and allow them to anneal together. Figure 6.9(a) shows the initial stage of the computation while Figure 6.9(b) shows the pre-final stage of the computation when the last layer representing B_3 has been computed. At this stage the sticky ends of the final layer tiles contain the result of the multiplication. In order to output the result in the form of a DNA strand, we require some output tiles of the form shown in Figure 6.5. On adding these tiles, and allowing them to anneal, we get the the final tile assembly as shown in Figure 6.9(c). On adding *Ligase* to seal the bonds, we will have a single strand of DNA passing through the tiles in the final output layer, that encodes the result of the computation. This single strand begins with the unique nucleotide sequence labeled "res".

To extract the ssDNA representing the result, we first break up the hydrogen bonds to decompose the tile assembly into DNA single strands and then do an extraction operation using the nucleotide sequence complementary to the "res" sequence. The ssDNA obtained as the result of the computation would be in the same format (Figure 6.8) as the original inputs, and thus can be used as the input in further computations.

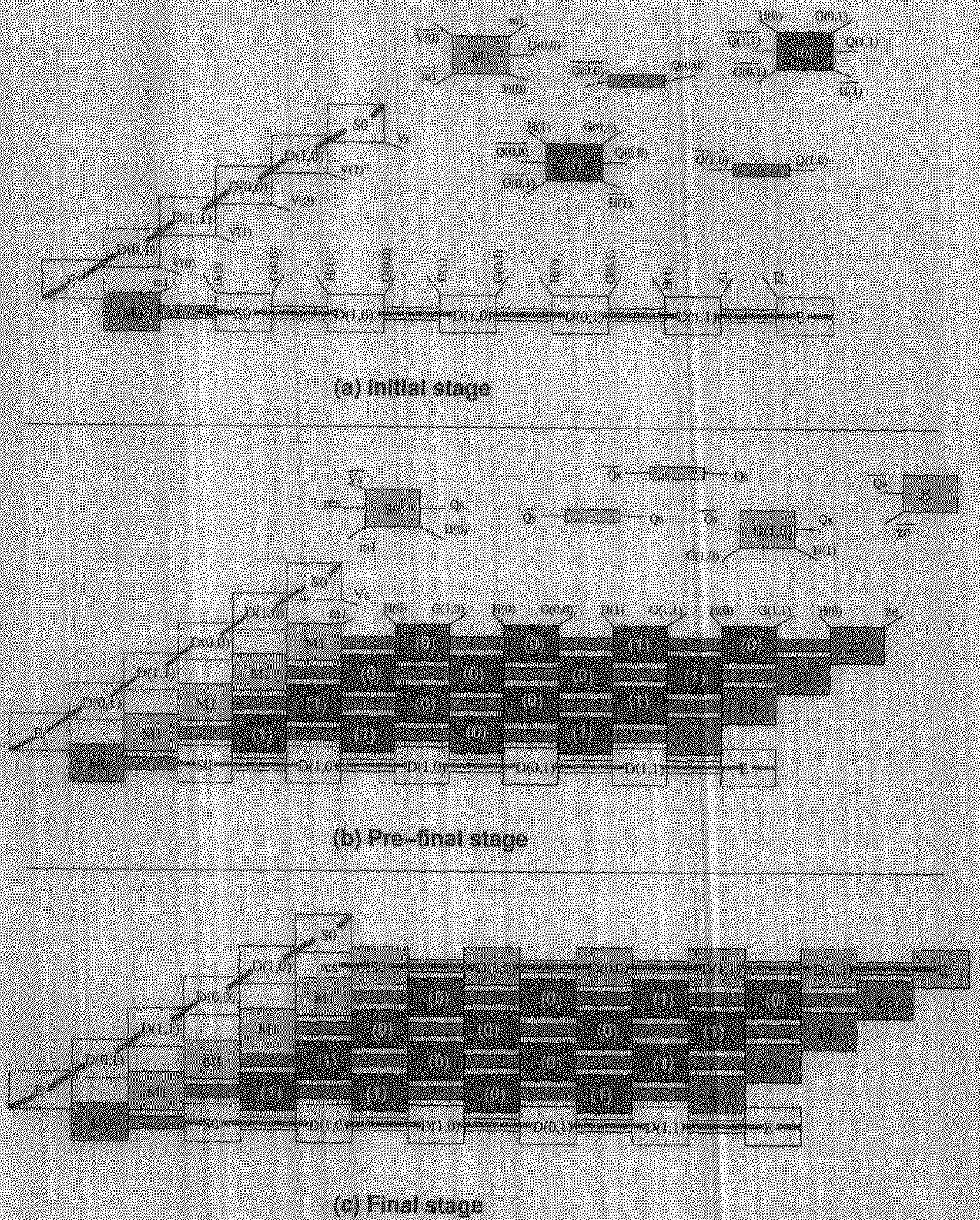
6.2.3 Finite Field Addition

The finite field addition operation can be implemented in a similar way. Figure 6.10 shows the addition of two four-bit numbers $A = (a3, a2, a1, a0)$ and $B = (b3, b2, b1, b0)$. The number A would be represented by the tile structure shown in the top while the number B would be represented by a similar tile structure with sticky ends at the top of the tiles instead of the bottom. The blue-colored tiles shown in the figure form the result, $C = (c3, c2, c1, c0)$ of the computation. Note that a single strand encoding the result passes through these tiles, and contains the 'res' sequence in the front. This strand can be extracted as before and used in further computations.

6.3 Implementation Issues

To implement our method of finite field arithmetic, we have to find suitable nucleotide sequences for encoding all the symbols used in our computation, such as S0, E, V(0), V(1), D(0,0), D(0,1) etc. and we have to ensure that these sequences (and their complementary sequences) are sufficiently different from each-other. The other issues to be considered while implementing our method are :

- Constructing the input DNA strands - The input DNA strands of the form shown in Figure 6.8 consists of the nucleotide sequence $S0$ followed by $D(0, r_i)$ (if i-th bit is zero) or, $D(1, r_i)$ (if i-th bit is 1) for each $i \in \{n-1, ..0\}$, finally followed by the sequence E (here r_i's are the bits of the irreducible polynomial R). Between the sequences encoding two consecutive bits we have another nucleotide sequence, say SJ which will be required for forming the tile structure. Now, we can initially construct all possible numbers by joining together the different nucleotide sequences forming the numbers. We will store multiple copies of all the numbers as a data pool and at the time of executing a computation we will extract the DNA strand encoding the particular number to be used as input.



(a) Initial stage

(b) Pre–final stage

(c) Final stage

Figure 6.9: Example Computation of 4-bit finite field multiplication

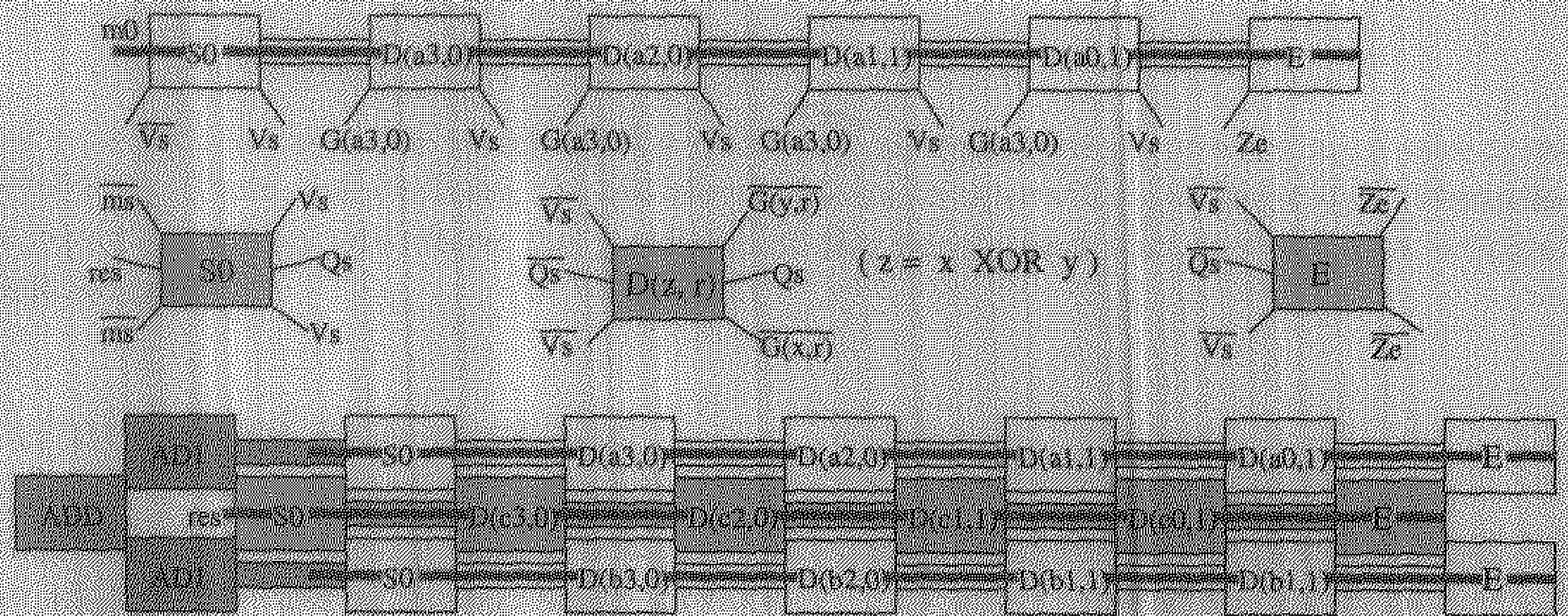


Figure 6.10: Finite field addition of four bit numbers

- Converting the input ssDNA to tiles - The DNA tiles are formed by annealing together of the constituent single strands making up the tile. For example, a TAO tile consists of four nucleotide strands as explained before (Figure 6.1). For converting the input ssDNA into the TAO tile structure (Figure 6.7), we just have to mix the other three constituent strands for each of the tiles in the structure and allow them to anneal to the long input strand. The input strand will act as a scaffold to which the other strands will join forming the required tile structure. Similarly the other tile structure (Figure 6.6) can be constructed.

- Constructing the computation tiles & output tiles - Figure 6.4 and Figure 6.5 show the various computation and output tiles that have to be constructed. These tiles are constructed by first constructing the constituent strands (by the standard method) and then allowing them to anneal together to form the tile. We will need multiple copies of these tiles during our computation, so we have to duplicate them. For duplicating any tile, it is first broken up into its constituent tiles; these are duplicated by the usual PCR reaction and then allowed to anneal together to form the tiles.

- Executing the Computation - The computation can be executed in a small chamber or cell containing the computation tiles, into which the input tile assemblies are added. The temperature and other conditions of the chamber would be adjusted by external controls, to allow for the annealing of complementary sticky ends. We can have multiple such chambers for parallely executing multiple such computations. At the end of each computation, *Ligase* is added to seal the joints and then, the temperature of the chamber is increased to break up the tile structure into single strands. The result strand is then extracted by affinity purification using the complement of the 'res' sequence. The 'res' nucleotide sequence would contain the site of a restriction enzyme at the end, so that the 'res' portion can be cut-off from the result strand by applying the appropriate restriction enzyme. As a result we will get an ssDNA of the same format as the input strands. This strand can then, be used in further computations.

- Reading the result - To output the result of a computation, we do the PCR reaction on the result strand using (as primer) the complement of $S0$ and each of $D(0,0), D(0,1), D(1,0)$, and $D(1,1)$ separately. We color the results of the first two

reactions and the last two reactions by different colored dyes (representing 0 and 1 respectively). Then, we perform the gel electrophoresis operation on these dyed strands, the result of which will show the position of the zero's and the one's in the original result strand.

6.4 Analysis of the method

Our method for implementing finite field computation, extends the technique used by LaBean et al.[LWR99] for binary addition and XOR. The advantage of our method, is that once the initial strands are constructed, each multiplication operation is computed very fast through the self-assembly process and the output of one computation can be directly passed as input to another computation. The only time consuming operation is the reading of the output. As this would be done only once at the end of a series of computations (or, a program), this would not much affect the total computation time.

To conclude, we look at the possible errors that can occur during our process of computation. The possible sources of errors are, either an error in constructing the tiles, or an erroneous binding of tiles. The former error can be minimized by appropriately choosing the nucleotide sequences used in the tile. LaBean et al.[LWR99] have shown that the nucleotide sequences of the constituent strands forming a tile, can be chosen in such a way that, whenever these strands are allowed to anneal, they almost always form the desired tile structure, without forming other unwanted structures. The other possible error is when a tile attaches to a site meant for some other tile. Now, if we choose the sticky end sequences encoding different symbols to be sufficiently different from each other and their complement, then only matching sticky ends will anneal together. In that case, a wrong tile attaching to a site is still possible if a tile with three sticky ends uses only two of them to attach to the site, with the third sticky ends remaining unmatched i.e. hanging. If that happens then the resulting structure will be highly unstable relative to the stable and compact 2D structure that we get for all correct bindings. Thus, the probability of such a mismatch occurring is very low.

Chapter 7

Conclusion

In this thesis, firstly we have introduced the subject of DNA computing and looked at the various operations that can be performed on DNA molecules. These operations have been utilized in various ways by scientists working on DNA computing, to come up with different methods of computing using DNA. We looked at some of these methods in chapter three. Most of these methods were proposed as theoretical models of computing, which have not been supported by practical experiments. Indeed, there has been very few practical implementation of DNA computing methods since the time when Adleman performed his experiment. One of the reasons which makes it difficult to practically implement DNA computing methods, is that the major bio-chemical operations used in DNA computing are very tedious and time-consuming. To make DNA computing feasible, we have to do away with such lengthy and tedious operations so that, DNA algorithms can be run faster. The 2-D self-assembly method proposed by Winfree is a step in this direction. As we saw in chapter four, the self-assembly process can work very fast, once the tiles have been designed and constructed. But, there are still some problems with the process; the initial step of constructing the tiles and the final step of extracting the result can be a bottleneck. Moreover, the success of the process depends on each tile attaching correctly in its position. The occurrence of a wrong attachment due to partial matching can lead to errors. Thus, we have to devise methods for minimizing the chances of error during self-assembly.

In chapter five and six, we looked at two methods for implementing the finite field arithmetic operations. The first method uses the cutting of DNA strands by restriction enzymes in a clever way, to obtain a seemingly feasible method of computing finite field arithmetic. While the finite field addition can be very efficiently executed by this method, the multiplication method takes $O(n)$ bio-steps, which is not very efficient. The second method uses the self-assembly of specially constructed DNA tiles, to compute the finite field arithmetic operations. Here, the major time-consuming step is the construction of the tiles. If the tiles can be constructed before-hand, then the process of computation would be very fast. Also it is possible to execute a series of finite field computations at a go, because the output of one step can be easily passed as the input of the next step. The possible pitfall here is the error-prone extraction operation that is used to obtain the result of the computation. If the error in the extraction operation can be minimized, then this would indeed be a most efficient and effective method for finite field computations.

Bibliography

[Adl94] L. Adleman, "Molecular computation of solutions to combinatorial problems", Science, 266:1021-1024, Nov.11, 1994.

[ARR96] L.M. Adleman, P.W.K. Rothemund, S. Rowesis, and E. Winfree, "On Applying molecular computation to the Data Encryption Standard", In Proc. of the Second Annual Meeting on DNA Based Computers, Princeton Univ., June 1996.

[Amo97] M. Amos, "DNA Computation", PhD thesis, Department of Computer Science, University of Warwick, UK, September 1997.

[BM02] R. Barua and J. Misra, "Binary arithmetic for DNA computers", 8th International Workshop on DNA-Based Computers(DNA8), Sapporo, Japan, 2002.

[BD03] R. Barua and S. Das, "Finite Field Arithmetic using Self-Assembly of DNA Tilings", Submitted to the CEC2003 Special Session on Bio-molecular Computing, University of New South Wales, Canberra, Australia, 2003.

[Ber66] R. Berger, "The Undecidability of the Domino Problem", Memoirs of the American Mathematical Society, 66:1-72, 1966.

[BDL95] D. Boneh, C. Dunworth, and R. J. Lipton, "Breaking DES using a molecular computer", Technical Report CS-TR-489-95, Princeton University, May 1995.

[BDS96] D. Boneh, C. Dunworth, and J. Sgall, "On the computational power of DNA", Discrete Applied Mathematics, 71(1-3):79–94, 1996.

[GFB96] F. Guarneiri, M. Fliss and C. Bancroft, "Making DNA Add", Science 273:220-223, 1996.

[GPZ97] V. Gupta, S. Parthasarathy and M.J. Zaki, "Arithmetic and Logic Operations with DNA", In Proc. of 3rd DIMACS Workshop on DNA Based Computers, U. Penn, 212-220, 1997.

[HAK+97] M. Hagiya, M. Arita, D. Kiga, K. Sakamoto, and S. Yokoyama, "Towards parallel evaluation and learning of boolean μ-formulas with molecules", In Proceedings of the 3rd DIMACS Workshop on DNA Based Computers, pp 105-114, University of Pennsylvania, June 23-25, 1997.

[Hea87] T. Head, "Formal Language Theory and DNA : an analysis of the generative capacity of specific recombinant behaviors", Bulletin of Mathematical Biology, 49(6):737-759, 1987.

[LWR99] T.H. LaBean, E. Winfree, J.H. Reif, "Experimental Progress in Computation by Self-Assembly of DNA Tilings", 5th International Meeting on DNA Based Computers(DNA5), MIT, Cambridge, M.A., June 1999.

[LYK+00] T.H. LaBean, H. Yan, J. Kopatsch, F. Liu, E. Winfree, J.H. Reif and N.C. Seeman, "Construction, Analysis, Ligation, and Self-Assembly of DNA Triple Crossover Complexes", J.Am.Chem.Soc. 122:1848-1860, 2000.

[Lip95] R. J. Lipton, "DNA solution of hard computational problems", Science 268:542–545, April 28, 1995.

[MLRS00] C. Mao, T.H. LaBean, J.H. Reif and N.C. Seeman, "Logical Computation using Algorithmic Self-Assembly of DNA Triple-Crossover Molecules", Nature 407: 493-496, 2000.

[OR96] M. Ogihara and A. Ray, "Simulating boolean circuits on a DNA computer", Technical Report TR 631, University of Rochester, Computer Science Department, August 1996.

[RWB+96] S. Rowesis, E. Winfree, R. Burgoyne, N. Chelyapov, M. Goodman, P. Rothemund, and L. Adleman, "A Sticker based Architecture for DNA Computation", In Proc. of the Second Annual Meeting on DNA Based Computers, Princeton Univ., June 1996.

[Wan61] H. Wang, "Proving theorems by pattern recognition ", BellSystems Technical Journal, 40:1-42, 1961

[Wan63] H. Wang, "Dominoes and the AEA case of the Decision Problem", In Proc. Symposion on Mathematical Theory of Automata, pp. 23-55, Polytechnic Press, New York, 1963.

[Win98] E. Winfree, "Algorithmic self-assembly of DNA", Ph.D.thesis at California Institute of Technology, August 1998.

[Win00] E. Winfree, "Algorithmic Self-Assembly of DNA: Theoretical Motivations and 2D Assembly Experiments", Journal of Biomolecular Structure and Dynamics, 11(2):263-270, 2000.

[WLWS98] E. Winfree, F. Liu, L. Wenzler and N.C. Seeman, "Design and self-assembly of two-dimensional DNA crystals", Nature 394:539-544, Aug.6, 1998.