A SOFTWARE APPROACH FOR HAZARD DETECTION AND COLLISION PREVENTION IN PIPELINED SISD MACHINES

A Thesis Presented to
The Faculty of the College of Engineering and Technology
Ohio University

in Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
ROGER G. BITAR
November 1987
# TABLE OF CONTENTS

LIST OF FIGURES

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>PRINCIPLES OF PIPELINING AND SISD COMPUTER ORGANIZATION</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>Pipelined Computers</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Instruction Execution in Non-pipelined SISD Machines</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Instruction Pipelining in SISD Machines</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Instruction Dependencies and Hazards</td>
<td>28</td>
</tr>
<tr>
<td>2.5</td>
<td>Job Sequencing and Collision Prevention</td>
<td>31</td>
</tr>
<tr>
<td>3.</td>
<td>DESCRIPTION OF THE METHOD</td>
<td>36</td>
</tr>
<tr>
<td>3.1</td>
<td>A Software Approach for Hazard Detection</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>The Computer Implementation</td>
<td>43</td>
</tr>
<tr>
<td>3.3</td>
<td>Simulation Results</td>
<td>45</td>
</tr>
<tr>
<td>3.4</td>
<td>Justification of the Results</td>
<td>55</td>
</tr>
<tr>
<td>3.5</td>
<td>Remarks</td>
<td>57</td>
</tr>
<tr>
<td>4.</td>
<td>A GREEDY CYCLE DETECTION TECHNIQUE</td>
<td>60</td>
</tr>
<tr>
<td>4.1</td>
<td>The Computer Implementation</td>
<td>60</td>
</tr>
<tr>
<td>5.</td>
<td>CONCLUSION</td>
<td>75</td>
</tr>
<tr>
<td></td>
<td>REFERENCES</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td>APPENDIX</td>
<td>79</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Instruction execution process</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>Organization of a pipelined processor</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Arithmetic pipelining</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>A partitioning model for executing the instruction classes</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>Reservation tables for a non-pipelined processor</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Reservation tables for a pipelined processor (simple prefetch)</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>Reservation tables for a pipelined processor (with increase in hardware)</td>
<td>26</td>
</tr>
<tr>
<td>8</td>
<td>Abstract tree representation of the program</td>
<td>47</td>
</tr>
<tr>
<td>9</td>
<td>A 2-D array representation of the state diagram</td>
<td>62</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Parallel processing is an efficient form of information processing that emphasizes the exploitation of concurrent events in the computing process. Spatial and temporal parallelisms are two fundamental mechanisms which may be used to achieve concurrencies in advanced computer architecture. In spatial parallelism, simultaneous events occur in multiple resources during the same time interval. In temporal parallelism, pipelined events are overlapped in time.

State-of-the-art parallel computer systems which achieve any of these parallelisms can be characterized into three structural classes; array processors, multiprocessor systems, and pipelined computers. Array processors use multiple synchronized processing elements to achieve spatial parallelism. These are mostly custom designed systems for specific applications. They usually require relatively difficult programming strategy due to their rigid architectures. Multiprocessor systems achieve asynchronous parallelism through a set of interactive processors with shared resources (memories, data base, etc.). They are very flexible in general purpose applications, but require expensive hardware in the implementation. Pipelined computers perform overlapped
computations to exploit temporal parallelism. Because they cost less and their operating systems are well developed to achieve better resource utilization and higher performance, pipelined computers dominate the commercial market in both business and scientific applications.

Designing these machines requires solving many difficult and complex problems. Interrupt and branch handling, hazard detection and resolution, collision prevention and job scheduling are some of the major difficulties which have to be resolved. Hazard detection and resolution is especially important in pipelined computers because their operations depend heavily on the overlapped execution of instructions. Overlapping several instructions in a pipelined computer often places sequencing constraints on these instructions. However, there exists dependencies related to the execution of instructions that cause hazards. They include memory accessing, register utilization, operand availability, and branch possibilities. These dependencies have to be detected and ordered to obtain a correct result following the execution of a program. Several techniques have been developed and used to this end, mainly at the hardware level. For example, Shar and Davidson [1] proposed a method based on sharing the pipeline by several instruction streams. This method is similar to that used in multiple instruction and multiple data (MIMD) machines. In this approach, the pipeline must be quite rigid with well defined reservation tables that
determine the path of each instruction type. For each instruction stream, there is only one instruction active at any time in the pipeline. This eliminates any potential hazard. Also, a control mechanism is needed to choose an instruction from an instruction stream that does not have a currently executing instruction in the pipeline. This optimizes stage usage while avoiding collisions.

The concept of data-flow architecture has been suggested by Miranker [2] and Syre et al. [3]. In this scheme, sequentiality is replaced by a directed graph where processing elements are represented by nodes, and their output values by tokens. These values or tokens reach other nodes through the links of the directed graph. In this architecture, performing an operation involves absorbing all input tokens and placing an output token on a link. A token on a link may be replicated to all units that need it. This approach does not use any program counter thus eliminating potential hazards found in sequential programs. The data-flow program is placed in a set of instruction cells containing an opcode, input operands, and a list of other cells that should receive the result. The opcode of each cell specifies which operation unit is to receive the inputs and to generate the results. These operation units are controlled by an arbitration network. A distribution network routes copies of the result into the operand spaces of the instruction cells. The main problem with this architecture is the mapping requirement from high level languages to low-level data-flow programs. This architecture is
still under study and recent development can be found in [4] and [5].

Other approaches to hazard resolution are based on detecting dependencies. For example, the IBM system 360 model 91 uses a common data bus (CDB) and a register tagging scheme (Tomasulo [6]) to allow simultaneous execution of independent instructions while preserving the essential precendences inherent in instruction stream. The floating point execution unit of the model 91 consists of one add and one multiply/divide unit. Each unit is connected to a set of registers (control, sink, source) called a reservation station. There are three add and two multiply/divide reservation stations. The source and sink registers in a reservation station are identified by unique tags. There are four scratch pad registers, each associated with a busy bit. When the busy bit is set, it identifies the corresponding register as sink waiting for a result. Once the register receives the result, its busy bit is reset. The CDB collects the outputs of the add and multiply/divide unit reservation stations, and the output of the buffer that holds the block of data fetched from memory. The CDB in turn feeds the floating point registers (FLR), the reservation stations, and the storage buffer. After an instruction is decoded, the busy bit of the specified sink register is set, and its tag is updated by the tag of the reservation station of the execution unit performing the operation. Upon decoding of the following instruction, if the busy bit of the specified FLR is on,
the reservation station tag in the execution unit is changed to that of the register. This way the result of the previous instruction is forwarded immediately to the reservation station of the execution unit. The FLR tag is also updated to point to the new reservation station. After the specified operation is performed, the result of the execution unit and the tag of the corresponding reservation station are transferred to the CDB. Each active reservation station compares the tags of its sink and source registers with that of the CDB. If there is a match, the reservation station receives the data from the CDB. Also, all busy registers compare their tags with the tag on the CDB. In case of a match, the registers will be loaded from the CDB and their busy bits are reset. By passing tags around, all operations having the same sink are correctly sequenced, thus preserving dependencies and eliminating hazards. All other operations, which do not use the same sink, are allowed to proceed independently.

Another hardware-oriented method for detecting dependencies has been suggested by Tjaden and Flynn [7]. This method involves simultaneous decoding of n instructions. The m independent instructions which are taken from a block of n instructions are immediately executed. Two instructions \(I_i\) and \(I_j\) are said to be weakly independent if the source registers of instruction \(i\) and the sink registers of instruction \(j\), and vice versa, were determined to be different after comparing their effects and
dependencies. On the other hand, two instructions are said to be strongly independent if the source and sink memory locations are not the same. The execution of weakly and strongly independent instructions is handled differently in the processor unit. This unit consists of a set of scratch pad registers, several arithmetic units, a control unit, a predecode stack (PDS), several data buffers, and an effective address buffer (EAB). The PDS detects strongly and weakly independent instructions and sends the addresses of the weakly ones to the EAB. In the EAB each effective address is compared bit by bit with all the addresses in the buffer to determine whether or not the instruction is strongly independent. The instruction format used in this architecture must represent explicitly the dependencies and effects of each instruction. This is achieved by including one dependency field and one effect field in the opcode part of an instruction. In this approach, the amount of hardware needed to implement the described matching process would increase exponentially with the size of PDS. Therefore the cost to simultaneously decode \( n \) instructions will rise exponentially with \( n \). In addition, the amount of time required for the PDS to check the independent instructions would increase with the size of PDS. Also, since the decoding time is directly related to \( n \), this approach is more suitable for small number of simultaneous decoding.

In this thesis we describe a new method which regards dependency and hazard problems at the software level. The
software tool developed can detect all the dependencies in a given program and produces all the potential hazards as well as the locations of their occurrences. The input to this program should be provided by a parser in a two-dimensional array form as explained later. This would lead to a hazard free execution by introducing additional non-computing delays in the implementation. This is left as a further research topic. A possible use of this technique would be in three different applications. It may be utilized as an interpreter replacing hazard detection hardware. Another use would be to create a simulation environment for the operation of a pipelined computer processor. Finally, it could be used to create a microprogram code for the operation of a control unit. This would be an alternative to the use of explicit information in the instruction format as was suggested by Tjaden and Flynn [7].

In the following chapters, we first describe the principles of pipelining and a single instruction-single data (SISD) computer organization. Then in chapter three, we present the software approach for hazard detection. A greedy cycle detection technique is discussed in chapter four. Chapter five is devoted to the conclusion. The simulation program is presented in the appendix.
CHAPTER 2

PRINCIPLES OF PIPELINING AND SISD COMPUTER ORGANIZATION

2.1. Pipelined Computers

The execution of an instruction in a computer involves four major steps; instruction fetch (IF), instruction decode (ID), operand fetch (OP), and execute (EX) (Fig. 1.a). In a non-pipelined machine, these four steps must be completed before the next instruction can be issued (Fig. 1.b). However in a pipelined computer, these four steps can be overlapped in time to reduce the instruction cycle significantly (Fig. 1.c). In this architecture, an instruction cycle consists of multiple pipeline cycles, which depend on the organization and operation of the pipeline stages. The flow of information from stage to stage is triggered by a common pipeline clock which is usually set to the delay of the slowest pipeline stage. High speed interface latches are used between adjacent segments to hold the intermediate results. Once the pipeline is filled up, an output is produced on each cycle thereby reducing the instruction cycle by one fourth over the non-pipelined approach (Fig. 1.c).

Pipelines can be made as unifunctional systems which perform only fixed and dedicated functions, or organized as
Figure 1. Instruction execution process. (a) Steps involved. (b) Space-time diagram for a non-pipelined process. (c) Space-time diagram for a pipelined process (Hwang et al. [8]).
multifunctional units which execute different functions simultaneously (dynamic) or at different times (static). Pipelines can also be organized for processing a sequence of scalar operands or handling vector instructions over vector operands. According to the level of processing, pipelining can be classified as either instruction or arithmetic. In instruction pipelining, the execution of instructions can be overlapped by the execution of the current instruction with the fetch, decode, and operand fetch of subsequent instructions (Fig. 2). In arithmetic pipelining, the input task (process) is subdivided into a sequence of subtasks, each of which can be executed by a specialized hardware stage. The stages, which operate concurrently, are pure combinational circuits performing arithmetic or logic operations. They are separated by latches that hold intermediate results (Fig. 3.a). A space time diagram shown in Fig. 3.b describes the overlapped execution of several subtasks. This space time diagram is more generally known as a reservation table (Fig. 3.c). Multiple entries in a column of the reservation table correspond to simultaneous usage of several pipeline stages. Multiple entries in a row of the reservation table represent a repeated usage of a given stage.
Figure 2. Organization of a pipelined processor (Hwang et al. [8]).
Figure 3. Arithmetic pipelining. (a) Pipelined execution unit. (b) Space-time diagram depicting the overlapping of several subtasks. (c) The corresponding reservation table (Hwang et al. [8]).
2.2. Instruction Execution in Non-pipelined SISD Machines

Four classes of instructions in a typical non-pipelined SISD machine may be identified as add, store, unconditional jump, and conditional branch as suggested by Kogge [9]. A possible partitioning model for executing an instruction has also been proposed in [9]. The execution of add class instructions has been divided into eight stages (Fig. 4.a). The instruction fetch stage (IFETCH) fetches the instructions from memory. The decode stage (DECODE) decodes the instructions to be executed. The effective address generation stage (EAGEN) computes the effective address of the operand. The first operand fetch stage (OPFETCH 1) fetches the operand using the generated effective address. The second operand fetch stage (OPFETCH 2) transfers the other operand from a processor register to the execution unit. The execute stage (EXEC) performs the specified operation. The save stage (SAVE) moves the result to a destination register. The last stage (ENDOP) sets error flags and condition codes, increments the program counter (PC), and checks possible interrupts.

The execution of store class instructions has been partitioned into six stages; IFETCH, DECODE, EAGEN, operand fetch from registers (OPFETCH), store data in memory (STORE), and ENDOP (Fig. 4.b). For unconditional jump instructions, the execution has been divided into five stages: IFETCH, DECODE, EAGEN, change PC to effective address (PC=EA), and ENDOP (Fig. 4.c). For the
Figure 4. A partitioning model for executing the instruction classes. (a) Add class (Kogge [9]).
Figure 4 (continued). A partitioning model for executing the instruction classes. (b) Store class. (c) Jump class (Kogge [9]).
Figure 4 (continued). A partitioning model for executing the instruction classes. (d) Conditional branch class (Kogge [9]).
conditional branch class, execution partitioning consists of either six or four stages. If a branch is taken, these stages are IFETCH, DECODE, TEST, EAGEN, PC=EA, and ENDOP (Fig. 4.d). Otherwise, they are IFETCH, DECODE, TEST, and ENDOP (Fig. 4.d). Table 1 represents a time allocation to the above partitioning model with the following assumptions: Since memory is the slowest unit it is assumed that a memory access requires five time units, while a cpu operation may be completed in one, two or three time unit(s) depending on the operation being performed. For example, simple cpu operations like loading the PC, performing ENDOP, and testing are done in one time unit. Other operations like DECODE and EAGEN are performed in two time units. Arithmetic and logic operations take three time units, which are the average time between relatively short (add, load or logic operations) and long (multiply, divide) operations. Figure 5 represents the reservation tables according to this model.
### 2.3. Instruction Pipelining in SISD Machines

Instruction pipelining involves prefetching the next instruction while the current one is being executed in the processing unit. This provides overlapped execution of several
Figure 5. Reservation tables for a non-pipelined processor. (a) Add class. (b) Store class (Kogge [9]).
Figure 5 (continued). Reservation tables for a non-pipelined processor. (c) Unconditional jump class. (d) Conditional branch (taken) class (Kogge [9]).
Table 5 (continued). Reservation table for a non pipelined processor. (e) Conditional branch (not taken) class (Kogge [9]).

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
</tr>
</thead>
<tbody>
<tr>
<td>M.</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

X denotes instruction Ii
+ denotes instruction Ij
instructions during the same period of time. Indeed it is the basic technique for achieving temporal parallelism in the design of pipelined SISD computer systems. The overlapped execution of instructions serves also as the main time-saving mechanism. In order to demonstrate this concept, the execution of pipelined instructions is illustrated in the reservation tables of Fig. 6. By comparing the reservation tables of Figs. 5 and 6, it can be concluded that six time units are saved for the add class instructions in the simple prefetch pipelined operation. Also, one time unit is saved for the store class instructions with simple prefetch. This results from fetching instruction \( I_j \) at \( t_{14} \) and \( t_{15} \) for the add and store classes in simple prefetch pipelined operation while fetching the same instruction at \( t_{20} \) and \( t_{16} \) in non-pipelined operation.

The performance of this simple prefetch operation can further be improved by about 4.5 \% by using more hardware to perform the IFETCH at time \( t_7 \) for the store and branch classes (Fig. 7). For the store class instructions, prefetching starts just after the current instruction is decoded; i.e., at \( t_7 \). If the store and the ifetch operations correspond to the same memory location, this may cause a memory conflict or hazard. Therefore a test must be made to compare the effective addresses of the store and prefetch operations (Fig. 7). In case of a match, the second IFETCH is discarded and reinitiated after the store operation is completed. Similarly, prefetching can start at time \( t_7 \) for the branch class
Figure 6. Reservation tables for a pipelined processor (simple prefetch). (a) Add class. (b) Store class (Kogge [9]).
Figure 6 (continued). Reservation tables for a pipelined processor (simple prefetch). (c) Unconditional jump class. (d) Conditional branch (taken) class (Kogge [9]).
Table 2. Reservation table for a pipelined processor (simple prefetch).

| M | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| CPU | | | | | | | | | | | | | | | | | | | |

X denotes instruction $i_i$
+ denotes instruction $i_j$

Figure 6 (continued). Reservation table for a pipelined processor (simple prefetch). (e) Conditional branch (not taken) class (Kogge [9]).
Figure 7. Reservation table for a pipelined processor (with increase in hardware). (a) Store class with no hazard. (b) Store class with hazard (Kogge [9]).
<table>
<thead>
<tr>
<th>IFETCH</th>
<th>IFETCH</th>
<th>IFETCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>!X</td>
<td>+</td>
<td>!X</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>!X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IFETCH</th>
<th>IFETCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
</tr>
<tr>
<td>!X</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 7 (continued). Reservation tables for a pipelined processor (with increase in hardware). (c) Conditional branch (taken) class. (d) Conditional branch (not taken) class (Kogge [9]).
instructions immediately after decoding the current instruction. If the branch is successful, the prefetch is discarded and the proper instruction is fetched again. In this case, the reservation table shows the same timing as in simple prefetch. This design improves the performance by about 25% over non-pipelined architecture.

2.4. Instruction Dependencies and Hazards

The overlapped execution of instructions can cause potential hazards if operations initiated by instructions \(i+1\), \(i+2\), ... depend on the results from instruction \(i\) whose execution has not yet been completed. There are four classes of hazards; read after write (RAW), write after read (WAR), write after write (WAW), and read after read (RAR) which may be caused if domains and ranges of any two instructions are not independent. The domain \(D\) is defined to be the storage media (memory locations, registers, flags) that affect the execution of an instruction. It is usually represented by the variables on the right hand side of an arithmetic or logic operator. The range \(R\) characterizes the storage media which are affected by the execution of an instruction. It is represented by the variable on the left hand side of an arithmetic or logic operator.

RAW hazard occurs when instruction \(j\) (\(I_j\)), that logically
follows instruction i (Ii), attempts to read some data that is modified by Ii. If Ij reads the data before Ii is completed, Ij reads the wrong value. To demonstrate this hazard we consider, without loss of generality, the following sequence of instructions:

\[(Ih): X2 = 1 \quad \text{(write into } X2, X2=R)\]
\[(Ii): X2 = 0 \quad \text{(write into } X2, X2=R)\]
\[(Ij): X3 = X2+X3 \quad \text{(read } X2, X2=D)\]

The first instruction (Ih) is a write operation in which X2 is a range (X2=R). The second one (Ii) also performs a write operation regarding X2 as a range. The last one (Ij) executes a read operation on X2 and uses it as a domain (X2=D). The domain of instruction j (Dj) and the range of instruction i (Ri) have X2 in common. This can be expressed mathematically as \(R(i) \cap D(j) \neq 0\) for X2. If Ij starts before Ii is completed, X3 has the wrong value (X3 = 1+X3).

WAR hazard occurs when Ij modifies some data to be read by Ii. If Ij accesses the data before Ii, Ii reads the wrong value. The following example describes the WAR hazard.

\[(Ih) : X2 = 1 \quad \text{(write into } X2, X2=R)\]
\[(Ii): X3 = X2+X3 \quad \text{(read } X2, X2=D)\]
\[(Ij): X2 = 0 \quad \text{(write into } X2, X2=R)\]

Since \(D(i) \cap R(j) \neq 0\) for X2, then if Ij starts before Ii is
completed, \( X_3 \) has the wrong value.

WAW hazard occurs when both \( I_i \) and \( I_j \) attempt to update the same data. If \( I_i \) changes the data after \( I_j \), then the final value of the data is not what is expected. This is represented by the following example.

\[
(I_i): X_2 = 0 \quad \text{(write into} \ X_2, \ X_2=R) \\
(I_j): X_2 = 1 \quad \text{(write into} \ X_2, \ X_2=R) \\
(I_k): X_3 = X_2+X_3 \quad \text{(read} \ X_2, \ X_2=D)
\]

Since \( R(i) \cap R(j) \neq 0 \) for \( X_2 \), then if \( I_j \) is performed before \( I_i \), \( X_3 \) has the wrong value.

RAR hazard occurs when both \( I_i \) and \( I_j \) try to read the same data. Therefore, the order of execution of \( I_i \) and \( I_j \) is immaterial. This hazard is explained below.

\[
(I_h): X_2 = 10 \quad \text{(write into} \ X_2, \ X_2=R) \\
(I_i): X_3 = X_2+X_3 \quad \text{(read} \ X_2, \ X_2=D) \\
(I_j): X_4 = X_2+x_4 \quad \text{(read} \ X_2, \ X_2=D)
\]

since \( D(i) \cap D(j) \neq 0 \) for \( X_2 \), overlapping \( I_i \) and \( I_j \) does not affect the rest of the program. Hence RAR is not considered a case causing hazard.
2.5. Job Sequencing and Collision Prevention

To achieve optimal performance in an arithmetic pipeline, several tasks must be initiated so that they are executed concurrently in the pipeline. A collision occurs when two or more initiations attempt to use the same stage at the same time. Job sequencing involves proper task initiations in order to avoid collisions and to achieve maximum throughput. The number of time units between two job initiations is known as "latency". A latency sequence refers to the sequence of latencies between successive initiations. A latency sequence that repeats itself indefinitely forms a latency cycle. Latency cycles in which each state is used only once are called simple cycles. The average latency of a cycle is the sum of its latencies divided by the number of states used. Latencies that cause collisions between two initiations constitute a forbidden set of latencies which can be determined from the reservation table. The number of time units between any two entries on the same row of the reservation table is a member of the forbidden set of latencies. This set is used to create the collision vector which in turn can be used to determine a collision free latency sequence. It is a binary vector whose size is equal to the largest forbidden latency that may be obtained from the reservation table. The bits in the collision vector are labelled from right to left in the following way: A one (1) in position i indicates that a new initiation which
starts i-time units after the last initiation is a member of the forbidden set of latencies and therefore can cause a collision. A zero entry in the collision vector indicates that there will not be any conflict between the new initiation and the previous one that is in the pipeline.

To achieve collision free initiations, the collision vector is loaded into a shift register which is shifted right one bit at a time, entering zero's from the left end. A new initiation is allowed at time instant t+k if a zero is shifted out of the register after k shifts from time t. This is implemented in a state diagram where at each time unit the current pipeline configuration corresponds to one of the states. A new state on the diagram represents the content of the shift register after a certain number of shifts (equal to the latency) and a logical OR with the initial collision vector. The ORing is necessary to avoid collisions with previously initiated tasks. Successive states on the diagram are linked by arcs labelled according to the latency. The shifting and ORing continue until no more new states can be generated. This process is demonstrated on the reservation table given below.
The forbidden set of latencies for this reservation table is

\{ 1, 3, 6, 7 \}

The collision vector may be obtained from this set as

01100101

The corresponding state diagram developed from the collision vector is presented below.
From the state diagram, we can detect the following simple cycles:

(8) (4) (5) (4,8) (5,8) (2,8) (2,2,8)

The average latency is obtained for each of the simple cycles as

8 4 5 6 6.5 5 4
Since the maximum performance of a pipeline is proportional to the reciprocal of the average latency, a job sequencing algorithm is used to detect the minimal average latency (MAL). Finding MAL can be done by detecting greedy cycles from each state in the diagram. A greedy cycle is defined as a simple cycle where the states in this cycle are determined by following the arcs with the minimal latencies. For the above example, cycle (2,2,8) is greedy. Therefore to achieve a collision free maximum performance, task scheduling should be done according to the greedy cycle, in this case (2,2,8), which specifies the number of time units between successive task initiations.
CHAPTER 3

DESCRIPTION OF THE METHOD

The work that have been done can be divided into two parts: The first part is dedicated to hazard detection, and the second part is devoted to the computation of the greedy cycle.

3.1. A Software Approach for Hazard Detection

The software approach developed for hazard detection is based on the Backus-Naur Form (BNF). BNF was first introduced by John Backus (Backus [10]) as a method for defining the syntax of algol 60. The objective was to create a universal programming language [11]. The syntax was later improved by Peter Naur (Naur [12]), who gave a complete description of the international algorithmic language (algol 60).

In order to utilize this algorithm, a program must first be written in mini-language core (Marcotty et al. [13]). This is a special purpose language which displays many of the features of high level languages. The program written in this way can be represented by an abstract tree using a context free syntax in BNF (Marcotty et al. [13]). The nodes at the bottom of the tree
are called terminals, and the parent nodes producing the terminal nodes are called non-terminals. The rule that governs the generation of a terminal from a non-terminal is called a production. Table 2 represents seventeen different productions (labelled from 1 through 17) of the context free syntax of mini-language core in BNF. These productions are used for the computer simulation of the algorithm proposed in this research. In each of the productions, a symbol "::=" seperates non-terminals (on the left), from terminals (on the right). The terminal and non-terminal nodes are enclosed in corne-brackets "<.>". In a production there may be several recursive or non recursive alternative terminals, which can be generated from a non-terminal. These alternatives, separated by vertical bars "|", are sequentially numbered from left to right. Some productions contain symbols typed in capital letters to separate the terminals and to show the identifiers. Other terminals are also separated by arithmetic operators (+, -, *, /), or comparison operators (=, ≠, <, >).
Table 2. Context free syntax of mini-language core in BNF
(Marcotty et al. [13]).
11. <input-statement> ::= INPUT <identifier>;
12. <output-identifier> ::= OUTPUT <identifier>;
13. <expression> ::= <factor> | 
    <factor> + <expression> | 
    <factor> - <expression>
14. <factor> ::= <operand> | 
    <factor> * <operand> | 
    <factor> / <operand>
15. <operand> ::= <identifier> | 
    <integer>
16. <integer> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
17. <comparison> ::= <operand> = <operand> | 
    <operand> ≠ <operand> | 
    <operand> < <operand> | 
    <operand> > <operand>

Table 2 (continued). Context free syntax of mini-language core in BNF (Marcotty et al. [13]).

After an abstract tree representation of the program is formed, the next task is to find a two-dimensional (2-D) array representation of the tree in order to store it in memory. For this, the nodes of the tree are numbered sequentially from top to bottom in a recursive operation. This may be achieved by traversing the tree from top to bottom branch by branch or level
by level. The 2-D array is then created in the following way. Its rows correspond to the node numbers. The first column (column 0) represents the non-terminal numbers for each node as specified in table 2. Column one represents the alternative numbers for the particular production. Columns 2, 3, 4 are pointers that point to the children (terminals) of a parent node. The following example demonstrates how such an array can be formed from the constructed tree representation of a program.

```
<program> 0

<table>
<thead>
<tr>
<th></th>
<th>PROG &lt;decl-seq.&gt; 1</th>
<th>BEGIN &lt;stmt-seq.&gt; END 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;decl.&gt; 2</td>
<td>&lt;stmt.&gt; 4</td>
<td>&lt;stmt-seq&gt; 5</td>
</tr>
</tbody>
</table>
```

Node 0 corresponds to the first production which generates two terminals <decl-seq.> and <stmt-seq>. These two terminals are represented by the nodes 1 and 3. Therefore, row 0 of the
array will have 1 (non-terminal number of <program>) in column 0, 
1 (one possible alternative for <program>) in column 1, 1 (pointer 
to node 1) in column 2, and 3 (pointer to node 3) in column 3.

Node 1 corresponds to the second production which generates 
one terminal if the first alternative is chosen. Row 1 of the 
array, representing node 1 of the tree, will have 2 (non-terminal 
number) in column 0, 1 (alternative number) in column 1, and 2 
(pointer to node 2) in column 2.

Row 2 of the array will have 3 in column 0 representing the 
third production.

Node 3 corresponds to the sixth production which generates 
two terminals (numbered 4 and 5) if the second alternative is 
chosen. Row 3 of the array will have 6 (non-terminal number) in 
column 0, 2 (alternative number) in column 1, 4 (pointer to node 
4) in column 2, and 5 (pointer to node 5) in column 3.

Rows 4 and 5 will have 7 and 6, respectively, in column 0 
corresponding to the non-terminal numbers. This array 
representation is shown below.
<table>
<thead>
<tr>
<th>Row number</th>
<th>Column number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4</td>
</tr>
<tr>
<td>0</td>
<td>1 1 1 3 -</td>
</tr>
<tr>
<td>1</td>
<td>2 1 2 - -</td>
</tr>
<tr>
<td>2</td>
<td>3 - - - -</td>
</tr>
<tr>
<td>3</td>
<td>6 2 4 5 -</td>
</tr>
<tr>
<td>4</td>
<td>7 - - - -</td>
</tr>
<tr>
<td>5</td>
<td>6 - - - -</td>
</tr>
</tbody>
</table>

Row number = node number.
Column 0 = non-terminal number.
Column 1 = alternative number.
Column 2 = pointer.
Column 3 = pointer.
Column 4 = pointer.
- = not used.

After the array has been created, it is used as input to the simulation software which interprets the array and outputs
warning statements specifying the possible hazards and identifying their types. The simulation program also performs the computations corresponding to the input program and obtains the correct results as if the instructions were independent or executed sequentially. This is made possible since the context free syntax used allows arithmetic operations.

3.2. The Computer Implementation

The simulation software tool developed in this research is called "The.c" (see appendix for the program). It consists of a main program and sixteen subroutines. These subroutines, some of which are recursive, correspond to the sixteen productions of BNF grammar. The following flowchart represents the interaction between the subroutines.
Each of the subroutines performs all the functions of the alternative of a production as specified in table 2. These functions include calling other subroutines and performing arithmetic or logic operations. The abstract tree representation of a given program is presented to the main routine in the form of a 2-D data form called "program". The variables in the program are stored in a 2-D table called "symtbl" (for symbol table). They are assumed to be represented as X1, X2, etc. The rows in symtbl represent the subscripts of the variables (e.g., row 1 represents variable X1, row 5 represents variable X5, etc.). The first column (column 0) of symtbl holds the value of a variable.
The second column is used to check for variable declaration (i.e., a 1 in column 1 indicates that the variable is declared). The third column (column 2) is used to identify the variable as range represented by 1, or domain indicated by 2. Then each time the variable is referenced, column 2 is updated to represent the variable as domain or range. The same column is also used to specify the type of potential hazard, according to the rules of section 2.4. In accordance with this organization, symtbl for the statement \( X_{2}=5 \), for example, has the following form:

<table>
<thead>
<tr>
<th>Row number</th>
<th>column number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 1 2</td>
</tr>
</tbody>
</table>

----------------------

0
1
2 | 5 1 1

Row number = subscript of \( X \).
Column 0 = value of the variable.
Column 1 = declaration flag.
Column 2 = domain or range flag.

3.3. Simulation Results

The above simulation technique is tested on the following
program written in mini-language core. This program represents the four classes of instructions (add, store, unconditional jump, and conditional branch).

```
PROG DECLARE X1, X2, X3;
BEGIN
    input X1;
    input X2;
    input X3;
    WHILE X2 ≠ X3 LOOP
        IF X2 > X3 THEN
            X2 = X2 - X3;
            X1 = X1 - 1;
        ELSE
            X3 = X3 - X2;
            X1 = X1 + 1;
        ENDIF
    END LOOP
    output X1;
    output X2;
    output X3;
END;
```

X1 = 50, X2 = 9, X3 = 36.

The abstract tree representation of the program is obtained as shown in Fig. 8.
Figure 8. Abstract tree representation of the program.
Figure 8 (continued). Abstract tree representation of the program.
The above tree is converted to the following 2-D array representation:

<table>
<thead>
<tr>
<th>Row number</th>
<th>Column number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 1 1 9 -</td>
</tr>
<tr>
<td>1</td>
<td>2 1 2 - -</td>
</tr>
<tr>
<td>2</td>
<td>3 1 3 - -</td>
</tr>
<tr>
<td>3</td>
<td>4 2 4 5 -</td>
</tr>
<tr>
<td>4</td>
<td>5 0 1 - -</td>
</tr>
<tr>
<td>5</td>
<td>4 2 6 7 -</td>
</tr>
<tr>
<td>6</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>7</td>
<td>4 1 8 - -</td>
</tr>
<tr>
<td>8</td>
<td>5 0 3 - -</td>
</tr>
<tr>
<td>9</td>
<td>6 2 10 13 -</td>
</tr>
<tr>
<td>10</td>
<td>7 4 11 - -</td>
</tr>
<tr>
<td>11</td>
<td>11 1 12 - -</td>
</tr>
<tr>
<td>12</td>
<td>5 0 1 - -</td>
</tr>
<tr>
<td>13</td>
<td>6 2 14 17 -</td>
</tr>
<tr>
<td>14</td>
<td>7 4 15 - -</td>
</tr>
<tr>
<td>15</td>
<td>11 1 16 - -</td>
</tr>
<tr>
<td>16</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>17</td>
<td>6 2 18 21 -</td>
</tr>
<tr>
<td>18</td>
<td>7 4 19 - -</td>
</tr>
<tr>
<td>19</td>
<td>11 1 20 - -</td>
</tr>
<tr>
<td>Row number</td>
<td>Column number</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>20</td>
<td>5 0 3 - -</td>
</tr>
<tr>
<td>21</td>
<td>6 2 22 85 -</td>
</tr>
<tr>
<td>22</td>
<td>7 3 23 - -</td>
</tr>
<tr>
<td>23</td>
<td>10 1 24 29 -</td>
</tr>
<tr>
<td>24</td>
<td>17 2 25 27 -</td>
</tr>
<tr>
<td>25</td>
<td>15 1 26 - -</td>
</tr>
<tr>
<td>26</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>27</td>
<td>15 1 28 - -</td>
</tr>
<tr>
<td>28</td>
<td>5 0 3 - -</td>
</tr>
<tr>
<td>29</td>
<td>6 1 30 - -</td>
</tr>
<tr>
<td>30</td>
<td>7 2 31 - -</td>
</tr>
<tr>
<td>31</td>
<td>9 2 32 37 61</td>
</tr>
<tr>
<td>32</td>
<td>17 4 33 35 -</td>
</tr>
<tr>
<td>33</td>
<td>15 1 34 - -</td>
</tr>
<tr>
<td>34</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>35</td>
<td>15 1 36 - -</td>
</tr>
<tr>
<td>36</td>
<td>5 0 3 - -</td>
</tr>
<tr>
<td>37</td>
<td>6 2 38 49 -</td>
</tr>
<tr>
<td>38</td>
<td>7 1 39 - -</td>
</tr>
<tr>
<td>39</td>
<td>8 1 40 41 -</td>
</tr>
<tr>
<td>40</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>41</td>
<td>13 3 42 45 -</td>
</tr>
<tr>
<td>Row number</td>
<td>Column number</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>42</td>
<td>14 1 43 - -</td>
</tr>
<tr>
<td>43</td>
<td>15 1 44 - -</td>
</tr>
<tr>
<td>44</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>45</td>
<td>13 1 46 - -</td>
</tr>
<tr>
<td>46</td>
<td>14 1 47 - -</td>
</tr>
<tr>
<td>47</td>
<td>15 1 48 - -</td>
</tr>
<tr>
<td>48</td>
<td>5 0 3 - -</td>
</tr>
<tr>
<td>49</td>
<td>6 1 50 - -</td>
</tr>
<tr>
<td>50</td>
<td>7 1 51 - -</td>
</tr>
<tr>
<td>51</td>
<td>8 1 52 53 -</td>
</tr>
<tr>
<td>52</td>
<td>5 0 1 - -</td>
</tr>
<tr>
<td>53</td>
<td>13 3 54 57 -</td>
</tr>
<tr>
<td>54</td>
<td>14 1 55 - -</td>
</tr>
<tr>
<td>55</td>
<td>15 1 56 - -</td>
</tr>
<tr>
<td>56</td>
<td>5 0 1 - -</td>
</tr>
<tr>
<td>57</td>
<td>13 1 58 - -</td>
</tr>
<tr>
<td>58</td>
<td>14 1 59 - -</td>
</tr>
<tr>
<td>59</td>
<td>15 2 60 - -</td>
</tr>
<tr>
<td>60</td>
<td>16 0 1 - -</td>
</tr>
<tr>
<td>61</td>
<td>6 2 62 73 -</td>
</tr>
<tr>
<td>62</td>
<td>7 1 63 - -</td>
</tr>
<tr>
<td>63</td>
<td>8 1 64 65 -</td>
</tr>
<tr>
<td>Row number</td>
<td>Column number</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td>0  1  2  3  4</td>
</tr>
<tr>
<td>64</td>
<td>5  0  3    -  -</td>
</tr>
<tr>
<td>65</td>
<td>13 3 66 69 -</td>
</tr>
<tr>
<td>66</td>
<td>14 1 67    -  -</td>
</tr>
<tr>
<td>67</td>
<td>15 1 68    -  -</td>
</tr>
<tr>
<td>68</td>
<td>5 0 3     -  -</td>
</tr>
<tr>
<td>69</td>
<td>13 1 70    -  -</td>
</tr>
<tr>
<td>70</td>
<td>14 1 71    -  -</td>
</tr>
<tr>
<td>71</td>
<td>15 1 72    -  -</td>
</tr>
<tr>
<td>72</td>
<td>5 0 2     -  -</td>
</tr>
<tr>
<td>73</td>
<td>6 1 74    -  -</td>
</tr>
<tr>
<td>74</td>
<td>7 1 75    -  -</td>
</tr>
<tr>
<td>75</td>
<td>8 1 76 77 -</td>
</tr>
<tr>
<td>76</td>
<td>5 0 1     -  -</td>
</tr>
<tr>
<td>77</td>
<td>13 2 78 81 -</td>
</tr>
<tr>
<td>78</td>
<td>14 1 79    -  -</td>
</tr>
<tr>
<td>79</td>
<td>15 1 80    -  -</td>
</tr>
<tr>
<td>80</td>
<td>5 0 1     -  -</td>
</tr>
<tr>
<td>81</td>
<td>13 1 82    -  -</td>
</tr>
<tr>
<td>82</td>
<td>14 1 83    -  -</td>
</tr>
<tr>
<td>83</td>
<td>15 2 84    -  -</td>
</tr>
<tr>
<td>84</td>
<td>16 0 1     -  -</td>
</tr>
<tr>
<td>85</td>
<td>6 2 86 89 -</td>
</tr>
<tr>
<td>Row number</td>
<td>Column number</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>86</td>
<td>7 5 87 - -</td>
</tr>
<tr>
<td>87</td>
<td>12 1 88 - -</td>
</tr>
<tr>
<td>88</td>
<td>5 0 1 - -</td>
</tr>
<tr>
<td>89</td>
<td>6 2 90 93 -</td>
</tr>
<tr>
<td>90</td>
<td>7 5 91 - -</td>
</tr>
<tr>
<td>91</td>
<td>12 1 92 - -</td>
</tr>
<tr>
<td>92</td>
<td>5 0 2 - -</td>
</tr>
<tr>
<td>93</td>
<td>6 1 94 - -</td>
</tr>
<tr>
<td>94</td>
<td>7 5 95 - -</td>
</tr>
<tr>
<td>95</td>
<td>12 1 96 - -</td>
</tr>
<tr>
<td>96</td>
<td>5 0 3 - -</td>
</tr>
</tbody>
</table>

This array representation is then presented to the simulation program as the input data. The program internally creates a symbol table and updates its content for every iteration. The simulation routine also generates the types of hazards for each variable used during the execution of two or more overlapped macro instructions, and gives their resulting values. This is shown in the following computer printout:

WAW for X2
RAR for X3
RAR for X3
RAW for X2
WAR for X3
WAR for X1
WAR for X2
RAW for X3
RAW for X2
WAR for X3
RAW for X1
WAR for X1
WAR for X2
RAW for X3
RAW for X2
WAR for X3
RAW for X1
WAR for X1
WAR for X2
RAW for X3
RAW for X2
RAW for X3
X1=53
X2=9
3.4. Justification of the Results

By following the execution of the given program instruction by instruction, we have also demonstrated that the hand and machine computations are equivalent. This is demonstrated below.

Results obtained by hand and by machine

Going first time through the loop:

<table>
<thead>
<tr>
<th>By hand</th>
<th>By machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_2 \neq X_3$</td>
<td></td>
</tr>
<tr>
<td>$X_2 = R$ , $X_3 = D$</td>
<td></td>
</tr>
<tr>
<td>$X_2 &lt; X_3$</td>
<td></td>
</tr>
<tr>
<td>$X_2 = R$ (WAW) , $X_3 = D$ (RAR)</td>
<td>WAW for $X_2$</td>
</tr>
<tr>
<td></td>
<td>RAR for $X_3$</td>
</tr>
<tr>
<td>$X_3 - X_2$</td>
<td></td>
</tr>
<tr>
<td>$X_3 = D$ (RAR) , $X_2 = D$ (RAW)</td>
<td>RAR for $X_3$</td>
</tr>
<tr>
<td></td>
<td>RAW for $X_2$</td>
</tr>
<tr>
<td>$X_3 = X_3 - X_2 = 27$</td>
<td></td>
</tr>
<tr>
<td>$X_3 = R$ (WAR)</td>
<td>WAR for $X_3$</td>
</tr>
<tr>
<td>$X_1 + 1$</td>
<td></td>
</tr>
<tr>
<td>$X_1 = D$</td>
<td></td>
</tr>
<tr>
<td>$X_1 = X_1 + 1 = 51$</td>
<td></td>
</tr>
<tr>
<td>$X_1 = R$ (WAR)</td>
<td>WAR for $X_1$</td>
</tr>
</tbody>
</table>

$X_3 = 9$
Going second time through the loop:

---

By hand

\[
\begin{align*}
X_2 &\neq X_3 & X_2 &= R \text{ (WAR)} , & X_3 &= D \text{ (RAW)} \\
X_2 &< X_3 & X_2 &= R \text{ (WAW)} , & X_3 &= D \text{ (RAR)} \\
X_3 - X_2 & & X_3 &= D \text{ (RAR)} , & X_2 &= D \text{ (RAW)} \\
X_3 &= X_3 - X_2 = 18 & X_3 &= R \text{ (WAR)} \\
X_1 + 1 & & X_1 &= D \text{ (RAW)} \\
X_1 &= X_1 + 1 = 52 & X_1 &= R \text{ (WAR)}
\end{align*}
\]

By machine

\[
\begin{align*}
&\text{WAR for } X_2 \\
&\text{RAW for } X_3 \\
&\text{WAR for } X_2 \\
&\text{RAW for } X_3 \\
&\text{RAR for } X_3 \\
&\text{WAR for } X_1 \\
&\text{RAW for } X_1 \\
\end{align*}
\]

Going third time through the loop:

---

By hand

\[
\begin{align*}
X_2 &\neq X_3 & X_2 &= R \text{ (WAR)} , & X_3 &= D \text{ (RAW)} \\
X_2 &< X_3 & X_2 &= R \text{ (WAW)} , & X_3 &= D \text{ (RAR)} \\
X_3 - X_2 & & X_3 &= D \text{ (RAR)} , & X_2 &= D \text{ (RAW)} \\
X_3 &= X_3 - X_2 = 18 & X_3 &= R \text{ (WAR)} \\
X_1 + 1 & & X_1 &= D \text{ (RAW)} \\
X_1 &= X_1 + 1 = 52 & X_1 &= R \text{ (WAR)}
\end{align*}
\]

By machine

\[
\begin{align*}
&\text{WAR for } X_2 \\
&\text{RAW for } X_3 \\
&\text{WAW for } X_2 \\
&\text{RAR for } X_3 \\
&\text{WAR for } X_3 \\
&\text{RAW for } X_1 \\
&\text{RAW for } X_1 \\
\end{align*}
\]
X3 = X3 - X2 = 9  
X1 + 1  
X1 = X1 + 1 = 53

Fourth time:

------------

By hand

X2 = X3 = 9  
X2 = R (WAR), X3 = D (RAW)

By machine

X3 = R (WAR)  |  WAR for X3
X1 = D (RAW)  |  RAW for X1
X1 = R (WAR)  |  WAR for X1

At the output we have:

X1 = 53
X2 = 9
X3 = 9

3.5. Remarks

The algorithm developed here acts as an interpreter for any program written in mini-language core. An interpreter for a source language accepts a source program written in that language and executes it. Unlike a compiler, it does not generate a machine language version of the program for an assembler. An interpreter, like a compiler, consists of two phases. First,
it analyzes the complete source code and translates it into an internal form. This is identical to the first phase of a compiler. The second phase executes this internal form of a source program, while in a compiler this phase is dedicated to the generation of a machine code.

The first phase consists of lexical and syntactic analyzers. The lexical analyzer subdivides the input program into its elementary constituents, thereby identifying delimiters, operators, blanks, and comments. The syntactic analyzer (parser) identifies the larger program structure; i.e., statements, declarations, expressions, subprogram calls, etc., using lexical items. The second phase consists of semantic analyzer which processes the syntactic structures and checks for errors.

In the interpreter developed here, the parser is based on BNF grammar which determines the organization of syntactic and semantic analyses. It parses the source program into a tree corresponding to the syntactic categories assigned by BNF. After the various components of an expression are recognized, an appropriate routine performs the appropriate semantic analysis. The algorithm presented in this thesis takes as input the syntactic analyzer implemented as a 2-D array, and performs the semantic analysis and the execution. Indeed it implements these two functions correctly as can be deduced after comparing the computer results with the computations obtained by hand.
Therefore, if the conversion of the source program to a tree representation and the array implementation are done correctly, then the performance of the interpreter reflects all the rules given in BNF.

In real time application, the lexical analyzer produces an input to the syntactic one which in turn feeds the semantic analyzer. In this hierarchy, the proposed software can replace the last analyzer. This way it determines all the possible hazards.
CHAPTER 4

A GREEDY CYCLE DETECTION TECHNIQUE

Starting from the given reservation table, the forbidden set of latencies and the corresponding collision vector are first determined. This collision vector represents the initial state in the state diagram and is loaded into a shift register. The positions of the zero bits in the collision vector are then determined and labeled as the allowed latencies. For each of these latencies, the contents of the shift register are then shifted right, and ORed with the original collision vector. This gives another collision vector as the new state of the machine. This process is repeated at every state until no more new states could be generated. The greedy cycle is then obtained by selecting a single cycle from each state, and by following the minimum latencies.

4.1. The Computer Implementation

The second part of the simulation program "The.c" is dedicated to generating the greedy cycles and is presented in three steps.
Step 1:

The collision vector is stored in the first five columns (columns 0 through 4) of a two dimensional array called sdiag (for state diagram). Starting from column 4, the position of the first zero bit is then detected and stored in column five as the allowed latency (Fig. 9). The original collision vector is then shifted to the right with zeros entering from the left. The number of shifts is determined by the latency value. A logical OR operation is performed next using the shifted values and the initial collision vector. This is done in the "logor" routine in the program. This yields a new collision vector that is stored in columns 6 through 10 of the first row of sdiag. This new collision vector represents the new state in the state diagram that is connected to the original state by a link labelled according to the latency value in column five. This process is then repeated for the next zero that is detected in the original collision vector. The newly generated state will be placed in the next row of sdiag, from column 6 through column 10. In the same row, columns 0 through 4 are loaded with the original collision vector and column 5 is loaded with the latency. This process is repeated until no more zeros can be detected in the original collision vector. The control is then transferred to step two which is implemented as subroutine "check".
Figure 9. A 2-D array representation of a state diagram.

Step 2:

In this step, we compare the new state in row 0 (columns 6 through 10) to the states in columns 0 through 4 of each row in the table. If no match is found, the new state is then loaded in the next available row on the left handside of sdiag; i.e., in columns 0 through 4. The control is then transferred back to step 1 where the above procedure (detecting zeros, and loading new states in the right hand side of sdiag) is then repeated.

Step 3:

At this point all the states have been generated. This step is represented by subroutine "greedy" in the implementation (see
This routine detects the greedy cycle for each state in columns 0 through 4 of sdiag. Starting from row zero, we compare the state on the right hand side of sdiag (columns 6 through 10) with each state on the left hand side (columns 0 through 4) until a match is found. This match indicates the next state in the greedy cycle since it is chosen with the minimum latency. (Latencies are stored in increasing order, from top to bottom, in sdiag). After a match is found, subroutine "greedy" is called again to select the next state to complete the greedy cycle. This process continues until the original state is reached. Step 3 is repeated to select the greedy cycles for the other states in sdiag.

The above implementation is tested on the reservation table presented below:
The forbidden set of latencies determined from this table is \( \{2, 5\} \). The collision vector obtained from the above set is \( 1\ 0\ 0\ 1\ 0 \). We determine the permissible latencies from this vector as \( 1, 3, 4 \). The collision vector is presented as the input data to the simulation program, which generates internally the states from the reservation table and the greedy cycles. For this example, the computer printout is shown below.

<table>
<thead>
<tr>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Row number</td>
<td>Column number</td>
</tr>
<tr>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>0</td>
<td>1 0 0 1 0 1 1 1 0 1 1</td>
</tr>
<tr>
<td>1</td>
<td>1 0 0 1 0 3 1 0 0 1 0</td>
</tr>
<tr>
<td>2</td>
<td>1 0 0 1 0 4 1 0 0 1 1</td>
</tr>
<tr>
<td>3</td>
<td>1 0 0 1 0 6 1 0 0 1 0</td>
</tr>
<tr>
<td>4</td>
<td>1 1 0 1 1 3 1 0 0 1 1</td>
</tr>
<tr>
<td>5</td>
<td>1 1 0 1 1 6 1 0 0 1 0</td>
</tr>
<tr>
<td>6</td>
<td>1 0 0 1 3 1 0 0 1 0</td>
</tr>
<tr>
<td>7</td>
<td>1 0 0 1 4 1 0 0 1 1</td>
</tr>
<tr>
<td>8</td>
<td>1 0 0 1 6 1 0 0 1 0</td>
</tr>
</tbody>
</table>

The greedy cycles are 1,3,3 for state 1 0 0 1 0,
3,3,1 for state 1 1 0 1 1, and
3,1,3 for state 1 0 0 1 1.

Hand computation provides the following state diagram and the greedy cycles.
For state 1 0 0 1 0, the greedy cycle is 1,3,3.
For state 1 1 0 1 1, the greedy cycle is 3,3,1.
For state 1 0 0 1 1, the greedy cycle is 3,1,3.
It is noted that from each state a link to the original state is added with a latency of 6.

To check the computer results, the following computations are performed.

For latency = 1 in the original collision vector.

Shift right one position: 0 1 0 0 1
OR with original collision vector: 0 1 0 0 1  
          + 1 0 0 1 0  
        ------
New state with latency = 1 is 1 1 0 1 1  which is stored in row 0 of sdiag, columns 6 through 10.

For latency = 3 in the original collision vector:  
----------------------------------------------
Shift right three positions: 0 0 0 1 0  
OR with original collision vector: 0 0 0 1 0  
          + 1 0 0 1 0  
        ------
New state with latency = 3 is 1 0 0 1 0  which is stored in row 1 of sdiag, columns 6 through 10.

For latency = 4 in the original collision vector:  
----------------------------------------------
Shift right four positions: 0 0 0 0 1  
OR with original collision vector: 0 0 0 0 1  
          + 1 0 0 1 0  
        ------
New state with latency = 4 is 1 0 0 1 1  which is stored in row 2 of sdiag, columns 6 through 10.
For latency = 6 in the original collision vector:
---------------------------------------------------------------------
Shift right six positions : 0 0 0 0 0 0
OR with original collision vector : 0 0 0 0 0 0
    + 1 0 0 1 0
        --------
New state with latency = 6 is 1 0 0 1 0 which is stored in row 3 of sdiag, columns 6 through 10.

For latency = 3 in the collision vector 1 1 0 1 1:
---------------------------------------------------------------------
Shift right three positions : 0 0 0 1 1
OR with original collision vector : 0 0 0 1 1
    + 1 0 0 1 0
        --------
New state with latency = 3 is 1 0 0 1 1 which is stored in row 4 of sdiag, columns 6 through 10.

For latency = 6 in the collision vector 1 1 0 1 1:
---------------------------------------------------------------------
Shift right six positions : 0 0 0 0 0 0
OR with original collision vector : 0 0 0 0 0 0
    + 1 0 0 1 0
        --------
New state with latency = 6 is 1 0 0 1 0 which is stored in
For latency = 3 in the collision vector 1 0 0 1 1:

Shift right three positions: 0 0 0 1 0
OR with original collision vector: 0 0 0 1 0

\[ \begin{array}{c}
+ 1 0 0 1 0 \\
\hline
\end{array} \]

New state with latency = 3 is 1 0 0 1 0 which is stored in row 6 of sdiag, columns 6 through 10.

For latency = 4 in the collision vector 1 0 0 1 1:

Shift right four positions: 0 0 0 0 1
OR with original collision vector: 0 0 0 0 1

\[ \begin{array}{c}
+ 1 0 0 1 0 \\
\hline
\end{array} \]

New state with latency = 4 is 1 0 0 1 1 which is stored in row 7 of sdiag, columns 6 through 10.

For latency = 6 in the collision vector 1 0 0 1 1:

Shift right six positions: 0 0 0 0 0
OR with original collision vector: 0 0 0 0 0
+ 1 0 0 1 0
--------
New state with latency = 6 is 1 0 0 1 0 which is stored in row 8 of sdiag, columns 6 through 10.

When all the above generated states are entered into a 2-D array, we obtain the following array representation:

<table>
<thead>
<tr>
<th>Row number</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column number</td>
<td>0 0 0 1 0 1 1 1 0 1 1</td>
<td>1 0 0 1 0 3 1 0 0 1 0</td>
<td>1 0 0 1 0 4 1 0 0 1 1</td>
<td>1 0 0 1 0 6 1 0 0 1 0</td>
<td>1 1 0 1 1 3 1 0 0 1 1</td>
<td>1 1 0 1 6 1 0 0 1 0</td>
<td>1 0 0 1 1 3 1 0 0 1 0</td>
<td>1 0 0 1 1 4 1 0 0 1 1</td>
<td>1 0 0 1 1 6 1 0 0 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above array is the same as the one obtained from the machine.
In order to determine the performance of the pipelined system from the results obtained, we present the pipeline with four initiations for each greedy cycle in the following 3 cases:

Case 1: For the greedy cycle (1,3,3) the following reservation table results with four initiations:

<table>
<thead>
<tr>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>1</td>
<td>X Y Z X Y U W Z U W</td>
</tr>
<tr>
<td>2</td>
<td>X Y X Y Z U W U W</td>
</tr>
<tr>
<td>3</td>
<td>X Y Z U W</td>
</tr>
<tr>
<td>4</td>
<td>X Y Z U W</td>
</tr>
<tr>
<td>5</td>
<td>X Y Z U W</td>
</tr>
</tbody>
</table>

Y is initiated 1 time unit after X
Z is initiated 3 time units after Y
U is initiated 3 time units after Z
W is initiated 1 time unit after U
By computing the number of marks in each row, we obtain the following table:

<table>
<thead>
<tr>
<th>Row</th>
<th>Number of number</th>
<th>marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Case 2: For the greedy cycle (3,3,1) we obtain the following reservation table with four initiations:

<table>
<thead>
<tr>
<th>State</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12 13</td>
</tr>
<tr>
<td>-------</td>
<td>------------------</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>------</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>------</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>------</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>------</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
</tr>
</tbody>
</table>
Y is initiated 3 time units after X
Z is initiated 3 time units after Y
U is initiated 1 time unit after Z
W is initiated 3 time units after U

The number of marks per row are represented in the following table:

<table>
<thead>
<tr>
<th>Row</th>
<th>Number of marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Case 3: For the greedy cycle (3,1,3), four initiations are implemented in the reservation table given below, where

Y is initiated 3 time units after X
Z is initiated 1 time unit after Y
U is initiated 3 time units after Z
W is initiated 3 time units after U
The following table gives the number of marks per row:

<table>
<thead>
<tr>
<th>Row</th>
<th>Number of marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>
The software tool developed here performs the semantic analysis and the execution for an interpreter of a SISD machine. It also detects the dependencies and produces all the possible hazards (RAW, WAR, WAW, RAR) and the locations of their occurrences in a given program for a pipelined computer system. Additional non-computing delays may be introduced between the instructions at hazard locations in order to obtain a hazard free execution in pipelined computer systems. This is left as a further research topic. However, care must be taken in introducing these non-computing delays in the real time implementation to achieve a proper overlapping of instructions and a fast execution.

This approach can be substituted for hardware type hazard detection and thereby avoids the extra costs introduced by hardware in the implementation. In fact it can replace the last function (semantic analysis) of an existing interpreter that uses BNF grammar. The input data should be provided to this software by a parser in a two-dimensional array form as explained earlier. This is also left for further research. In its present form, the only limitation the method appears to have is the restrictions which may be imposed by the BNF grammar and mini-language core.
In the second part, the results obtained from detecting the greedy cycle show 3 possible implementations for the given reservation table. The tables obtained in each case are identical. They show similar utilization rate of the hardware for all three cycles in a collision free execution. This result agrees with the following lemma given by Kogge [9]: If the maximum number of marks in any row of the reservation table equals the number of 1's in the initial collision vector, then all greedy cycles are optimal and any of the ones detected may be chosen. However, a greedy cycle may not always indicate the optimal cycle, but it always insure a collision free execution.
REFERENCES


APPENDIX

The method developed in this research was implemented in VAX 11/750 under the UNIX 4.2 BSD operating system. In the following, the simulation program " the.c " written in C language is given. The results obtained from simulating a program are then presented at the end.
include <stdio.h>
#include <stdlib.h>
int prog[100][4]
int eds[200][11]
int swtbl[10][3]
FILE *infile, *outfile

/**
** This program can be divided into two main parts: the first part is
** dedicated to detecting the hazard, while the second part is devoted to
** finding the speed cycle. The program uses three main arrays
** program and swtbl for the first part, and eds for the second part.
** While array program holds the abstract tree representation of the input
** program, swtbl represents all the variables that are used. EDS holds
** the original collision vector and all the subsequently generated states
** along with their allowed latencies.
** /
main()
{
    int prog[];
    int logor();
    int check();
    int speed();
    int i=11; j=12; m=13; n=14; l=15; i=16; j=17; k=18; n=19; l=11; k=12; r=13; s=14; t=15;
    int li=4;
    int u=6;
    int J=6;
    int j=0;
    int n=0;
    int e=0;
    int iter=0;
    while (iter<9)
    {
        /* Part One: A Software Approach for Hazard Detection.
** The main routine reads the array program and calls subroutine program.
** /
        (swtbl[iter][2]=0)
        iter+=iter+1
    }
infile=fopen("table", "r");
outfile=fopen("answer", "w");
do
{
    fscanf(infile,"Xd Xd Xd Xd", &1, &2, &3, &4,
        &5);
    program[0][0]=1;
    program[1][2]=1;
    program[2][3]=1;
    program[3][4]=1;
    program[4][5]=1;
}
while (program[iter][0]==0)
    printf(outfile,"array representation of the ptn\n")
    printf(outfile,"non-term num s-num =-\n")
    printf(outfile,"= %d\n", n)
    n = n-2;
    for (i=0; i<n/2+1)
    {
        if (program[1][0])
            p2=program[1][1]
        if (program[1][0])
            p2=program[1][1]
    }
Part Two: Greedy Cycle Detection Technique

The main routine determines the allowed latencies. It calls subroutine `proc` to perform a logical OR operation. Then it calls subroutine `check` to determine whether or not the generated state is a new state in order to compute its latencies. Finally, it calls subroutine `greedy` which determines the greed cycle.

```c
proc(a)
```

```c
while (sdias[0][0][0] == 0)
    a = a - 1;
    while (sdias[0][0][0] == 0)
        a = a - 1;
        if (sdias[0][0][0] == 0)
            j = j - 1;
            while (j > 0)
                k = 0;
                j = j - 1;
```

```c
while (jj < jj + 1)
    if (jj == jj + 1)
        sdias[0][0][0] = sdias[0][0][0] * (jj - jj + 1);`
After generating the new states (columns 6 through 10), this loop copies the
**collision vector in columns 0 through 4 for each row.**

```c
for(o=11; o<=18; o++)
```

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```c

```c
while(sdias[am][0]!=2)
```

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```c

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```c

```c
while(sdias[am][0]!=2)
```

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```c

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```c

```c
/*
L = collision vector in column 0 through 4 for each row.
*/

```
while(sdiast(aa)(5)==
  (  
    aa=aa+1
  )
  )
  
  if(sdiast(aa)(0)==2)
  { 
    break
  }
  
  fclose(infile);
  fclose(outfile);

/#
/# This subroutine represents the first production:
/# (program::(=KVARA <declaration_sequence> BEGIN <statement_sequence> END)
/# if first checks for the appropriate non-terminal number, then calls
/# dec_seo and sta_seo.
/#
int a(
{ 
  int il;  // i is a row numbers.       
  int dec_seo();
  int sta_seo();
  il=program[a][0];
  if (il=program[a][0]==1)
    { 
      printf(outfile,"rs.a incorrect Id but call node 1\n", il);
    }
  dec_seo(program[a][2]);
  sta_seo(program[a][3]);
}

/#
/# This recursive subroutine represents the second production:
/# (declaration_sequence) if<declaration> <declaration_sequence>
/# if checks for the appropriate non-terminal numbers, then calls decl. If
/# the second alternative is chosen, then it calls dec_seo again.
/#
int a(
{ 
  int il;  // i is a row numbers.       
  int dec1();
  il=program[a][0];
  if (il=program[a][0]==2)
    { 
      printf(outfile,"rs.b incorrect Id but call node 2\n", il);
    }
  dec1(program[a][2]);
  if (il=program[a][3]==2)
    { 
      }
```c

dec_seq(proram[a][3]);
}

/*
** This subroutine represents the third production:
** <declaration>::=<identifier>;<identifier>
** It first checks for the appropriate non-terminal number, then calls id_list.
*/
int a;

t if 1
  id_list();
  id_list(a);
  if (program[a][0]=3)
     printf(outfile,"ms.c incorrect Xd but call node 3\n", 1);
  id_list(program[a][2]);
}

/*
** This recursive subroutine represents production (4):
** <identifier_list>::=,<identifier>;<identifier>
** It first checks for the appropriate non-terminal number, then calls id_list.
** If the second alternative is chosen, then it calls id_list again.
*/
int a;

t 1
  id();
  id(program[a][0]);
  if (1=4)
     printf(outfile,"ms.c incorrect Xd but call node 4\n", 1);
  id(program[a][2]);
  if (program[a][1]=2)
     id_list(program[a][3]);
}

/*
** This subroutine represents production (5):
** <identifier>::=0111213141516171819.
** It first checks for the appropriate non-terminal number, then stores a (1) in the second column of symtbl to indicate that the variable is declared.
*/
int a;

t 1
  id(a);
  id(a);
  if (1)
     id_list(proram[a][0]);
```
if (11==5)
{
  printf(outfile,"p.e incorrect Xd but call node 3\n", i);
}
setbl[program[a][2]][1]=1;

/#
## This recursive subroutine represents production (6)
## <statement_sequence>!1=<statement><statement_sequence>
## It first checks for the appropriate non-terminal number, then calls stat.
## if the second alternative is chosen, then it calls stmt for assign.
/#
stmt(a)
int i)
{
  int j; /* i are row numbers, */
  int stat();
  j=program[a][0];
  if (j==6)
  { /*print(outfile,"p.e incorrect Xd but call node 6\n", i);*/
    stmt(a)
    if (program[a][1]==2)
    {
      stmt(a)
    }
  }

/#
## This subroutine represents production (7)
## <statement>!4=<statement><if_statement><loop_statement>
## <input_statement><output_statement>
## It first checks for the appropriate non-terminal number. Then depending
## on the chosen alternative it calls assign, iff, loop, in, or out.
/#
stmt(a)
int i)
{
  int j; /* i are row numbers, */
  int assign(); iff(); loop(); in(); out();
  j=program[a][0];
  if (j==7)
  {
    printf(outfile,"p.e incorrect Xd but call node 7\n", i);
    for (i=program[a][1])
      switch(a)
      {
        case 1:
          assign(program[a][2]);
          break;
        case 2:
          iff(program[a][2]);
          break;
        case 3:
          loop(program[a][2]);
          break;
        case 4:
          in(program[a][2]);
          break;
        case 5:
          out(program[a][2]);
          break;
        case 6:
          printf(outfile,"p.e incorrect Xd but call node 6\n", i);
      }
```c
break;

case 5:
    out(Prog[a][2]);
    break;

default:
    fprintf(outfile,"invalid ps. executing in line Xd\n", a);
```

```
//
// This subroutine corresponds to production (8):
// assign_stement::{identifier}={<expression>}
// It first checks for the appropriate non-terminal number; the cells express.
// It prints two hazard statements: WAW or WAR.
// assign(a)
int a;
{
    int temp[2][1]; 
    /* temp; l, r, 1 are row numbers. */
    int express(); 
    /* temp; is the result returned from express. */
    int Prog[a][0];
    if (a=8)
        fprintf(outfile,"error in incorrect Xd but cell node Xd\n", i);
    else  
        Prog[a][2];
    temp[2][express(Prog[a][3]);
    swtbl[Prog[tem[2][2]][0] = temp[2];
    if (swtbl[Prog[tem[2][2]][2] == 1)
        fprintf(outfile,"waw for Xd at node Xd in the tree representation\n", Prog[tem][2][2];)
    else
        fprintf(outfile,"war for Xd at node Xd in the tree representation\n", Prog[tem][2][2];)
    swtbl[Prog[tem][2][2] = 1;]
```

```
//
// This subroutine correspond to production (9):
// if_statement::=IF<comparison>THEN<statement_sequence>ENDIF IIF<comparison>
// THEN<statement_sequence>ELSE<statement_sequence>ENDIF
// It first checks for the appropriate non-terminal number; then calls
// statement. Then depending on the chosen alternative, it calls
// stmt once or twice.
// iff(a)
int a;
{
    int l,r; /* l, r are row numbers. */
    int stmt(); /* k is the result returned from comparison. */
```
int compare();
int program[10];
if (i==0)
{
    printf(outfile, "ps.1 incorrect Xd but call node 9\n", i);
}
k=compare(program[2]);
if (k==1)
{
    sta_see(program[3]);
}
else if (program[11]==2)
{
    sta_see(program[4]);
}

/* This subroutine corresponds to production (10)
** <loop_statement1>WHILE<comparison>LOOP<statement_sequence>ENDLOOP
** It first checks for the non_terminal number, then calls compare and sta_see.
**/ loop(a)
int a;
{
int i,k; /* i, k are row numbers. */
int sta_see(); /* k is the result returned from compare. */
int compare();
int program[10];
if (i==10)
{
    printf(outfile, "ps.1 incorrect Xd but call node 10\n", i);
}
k=compare(program[2]);
while (k==1)
{
    sta_see(program[3]);
k=compare(program[2]);
}

/*
** This subroutine corresponds to production (11)
** <input_statement1>=[INPUT<identifier>]<input_statement2>
** It first checks for the appropriate non_terminal number, then it loads the
** value of the variable into column (0) of swtbl.
**/ in(a)
int a;
{
int i,j;data_term; /* i, j are row numbers. */
=program[3][0];
if (i==11)
{
    printf(outfile, "ps.1 incorrect Xd but call node 11\n", i);
}
term=program[2];
if (swtbl[term][2][1]==1)
```c
int main()
{
    float n, i,
    if (scanf(infile, "x: &data)"
        swatbli[program[tem][2]][0]=data
    )
    else
        printf(outfile, "var not declared\n")
);

/*
  # This subroutine corresponds to production (12)
  # <output statement>!1=OUTPUT<identifier>!
  # It first checks for the appropriate non-terminal number, then outputs the
  # value of the variable stored in swatbli.
  */
  out(u)
    int n
    {
      int temp1; /* n is a roundnumber */
      temp1=program[m][0];
      if (j=12)
          printf(outfile, "m: incorrect x but call node 12\n")
        temp=program[m][2];
        printf(outfile, "x = x\n", program[tem][2], swatbl=program[tem][2][0])
    }

/*
  # This recursive subroutine corresponds to production (13)
  # <expression>:1=<factor>1<factor>1<factor>1<factor>1<expression>
  # It first checks for the appropriate non-terminal number, then calls fact.
  # If any of the alternatives 2 or 3 were chosen, then it calls expression
  # again, compute the arithmetic value, and return the result.
  */
  int express(u)
    int n
    {
      int temp1, temp2; /* temp1, temp2 are algebraic values */
      int fact(t)
        temp1=program[m][0];
        if (j=13)
            printf(outfile, "m: incorrect x but call node 13\n")
        if (program[m][1]==1)
            temp=fact+program[m][2]
        if (program[m][1]==2)
            temp=fact+program[m][2]
        temp1=express(program[m][3])
        temp=temp2*temp1
    }
    if (program[m][1]==3)
```
```c
{ temp2=fact(program[2]);
 temp1=expr(program[3]);
 temp=temp2*temp1;
 return(temp);
}

/\ This recursive subroutine corresponds to production (14):
/\ (factor):(factor)<operand>(factor)\(factor)/<operand>
/\ It first checks for the appropriate non-terminal number, then calls or.
/\ If any of the alternative 2 or 3 are chosen, then it calls fact again,
/\ computes a factors, and return the result.
/\
int fact(a)
int a;
{ int i,temp1,temp2; /* temp1, temp2 are algebraic values*/
 int or();
 i=program[0];
 if (i=14) {
 printf(outfile,"\%s.n incorrect Id but call node 14\n", i);
 i=1; }
 if (program[1]==1) {
 temp=or(program[2],1); }
 else if (program[1]==2) 
 { temp=or(program[2],1); temp1=fact(program[3]); temp= temp2*temp1; }
 else if (program[1]==3)
 { temp1=or(program[2],1); temp2=fact(program[3]); temp1=temp2/temp1; }
 return(temp);
}

/\ This subroutine corresponds to production (15):
/\ It first check for the appropriate non-terminal number, then outputs four.
/\ types of hazards: WAW or WAW if it is called from comparison (i=0),
/\ KAW or KAH if it is called from fact (i=1).
/\
int or(a,1)
int a=1;
{ int temp1,temp2; /* temp is the value of the variables*/
 i=program[0]; /* temp1 is an input integer*/
 if (i=15)
}
I
4-
u
O
rn
o
N
v
I-.
u
u
Yt+
n
n
n
n
h-a
L
O.
ne(3
nnun
*a
-4
LIL
.L
.I.
4.3
ubu
L
u
1
LCL
rr
nam
a:
020
40
&
4
I
Mu
L
LL*
WL4
on-
L.L
.I
b3
h
Y
(.
-A-Y
LILA,
/# This subroutine corresponds to production (17)!
"<comparison:1=operand>"<operand>"<operand>"<operand>"<operand>"<operand>"<operand>"<operand>"<operand> boilerplate
# It first checks for the appropriate non-terminal number, then performs the
# logic operations and returns the results.
int compare(int a)
{
    int l, r, t1, t2, t3, t4;    // t1, t2, t3, t4 are values of operands
    int or();    // l is a flag
    if (Program[0] == 17)
    {
        printf(outfile, "Error: incorrect 3d but call node 17\n", i);
    }
    if (Program[1] == 1)
    {
        if (t1 == t2)
            k = 1;
        else
            k = 0;
    }
    if (Program[1] == 2)
    {
        if (t1 == t3)
            k = 1;
        else
            k = 0;
    }
    if (Program[1] == 3)
    {
        if (t1 == t4)
            k = 1;
        else
            k = 0;
    }
    if (Program[1] == 4)
    {
        if (t1 > t2)
            k = 1;
        else
            k = 0;
    }
    return(k);
```c
/*
 ** This subroutine performs a logical AND operation.
 */
void and(int i)
{
    int j; /* j is a pointer to the rightmost bit in the collision vectors */
    int k; /* k is a pointer to the leftmost bit in the collision vectors */
    while (j < 10)
    {
        if (sdis[i][j] && sdis[i][k])
            break;
        j++; k++;
    }
}

/*
 ** This subroutine compares each generated state in columns 6 through 10
 ** with each state in columns 0 through 4. If a mismatch occurs, the new
 ** state is entered in column 0 through 4 in order for its intensities to
 ** be determined in main.
 */
void check(int i)
{
    int j;
    int a=0;
    int i=0;
    int j=0;
    int k=0;
    int l=0;
    while (a<1)
    {
        while (i<4 && j<10)
            /* this loop compares a particular state in columns 6 through 10 with each
```
c
I.

m
L
r.
J
0
Y
a
I.
L
0
Y
0
c,
>
-3
L
c
I.
C
Y
::
L
C.
4

while(sdiss[1][]==sdiss[1][][J])
{
if(i<=4 && j<=10)
{
    a=1;
    i=i+1;
    j=j+1;
    m=1;
    break;
}
else if(i<=4 && j=10)
{
    a=0;
    i=i+1;
    j=j+1;
}

if(m==0)
{
    a=a+1;
    if(a=1)
    {
        i=5;
        j=11;
    }
    else if(a=1)
    {
        i=0;
        j=6;
    }
}

if(a=0) // enter the new state in columns 0 through 4 of sdiss #/
{
    i=0;
    j=61;
    while(i<=4 && j<=10)
    {
        sdiss[1][][i]=sdiss[1][][j];
        i=i+1;
        j=j+1;
    }
    r=1;
    p=p+1;
    m=0;
    i=0;
    j=61;
}

// this is a recursive subroutine that determines the greedy cycle. it shows
// it the output how to go from state to state by following the smallest
/*
 * latencies.
 */

greedy(p, k)
int r=0;
int a=0;
int j=0;
int k=1;
int r=0;
int j=6;
int a=0;

/* Starting from row 0, compare the states in columns 4 through 10 bit by bit
with those in columns 0 through 4 to determine the next state in the cycle*/

while(i==4 && j==10)
{
    while(sdisp[i][j]==sdisp[i][j])
    {
        if(i==4 && j==10)
        {
            a=1;
            i=11;
            j=11;
            k=1;
            break;
        }
        else if(i==4 && j!=10)
        {
            a=0;
            i=11;
            j=11;
        }
    }
    if(a==0)
    {
        a=1;
        i=0;
        j=6;
    }
    if(a==1)
    {
        unsdisp[i][j]=
        fprintf(stdout,"from state row %d so to state row %d with latency %d
", i,k,r);
    }
    if(r==k)
    {
        r=1;
    }
    else if(r!=k)
    {
        r=0;
    }
    if(r==1)
    {
        fprintf(stdout,"cycle is closed
");
    }
    else if(r==0) // call greedy again to determine the next state in the cycle*/
    {
        greedy(p,k);
    }
}
raw for \( x = 2 \) at node 72 in the tree representation
raw for \( x = 3 \) at node 64 in the tree representation
raw for \( x = 1 \) at node 80 in the tree representation
raw for \( x = 1 \) at node 76 in the tree representation
raw for \( x = 2 \) at node 26 in the tree representation
raw for \( x = 3 \) at node 28 in the tree representation

\( x_1 = 33 \)
\( x_2 = 9 \)
\( x_3 = 9 \)

State diagram:

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 0 & 3 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 4 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 4 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 3 & 1 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 6 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 4 & 1 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 4 & 1 & 0 & 0 \\
\end{array}
\]

The greedy cycles are:

- From state row 0 so to state row 4 with latency 1
- From state row 4 so to state row 6 with latency 3
- From state row 6 so to state row 0 with latency 3

Cycle is closed.

- From state row 4 so to state row 6 with latency 3
- From state row 6 so to state row 0 with latency 3
- From state row 0 so to state row 4 with latency 1

Cycle is closed.

- From state row 6 so to state row 0 with latency 3
- From state row 0 so to state row 4 with latency 3
- From state row 4 so to state row 0 with latency 1

Cycle is closed.