Implementation of a Transparent Peripheral Component Interconnect (PCI) Bridge using FPGA

P.B. Mane* and S.M. Jagdale**

Abstract

PCI (Peripheral Component Interconnect) has become one of the most popular bus standards, not only for personal computers, but also for industrial computers, communication switches, routers, and instrumentation. The function of the PCI bridge is to map various control signals and address space from one bus to another. The PCI bridge can be programmed to connect directly to the Non-Multiplexed mode or the Multiplexed mode 8-bits, 16-bits, or 32-bit Local Bus. The 32-bit 33-MHz PCI interface provides a user-friendly interface between a Local Bus and a PCI bus. It is fully compliant with PC local bus specification. The PCI interface has both the master and target capabilities. This PCI interface can be designed on ASIC (Application Specific Integrated Circuit) and PLD (Programmable Logic Devices) implementation. The ASIC designed for a PCI bridge to be sent for fabrication is a time consuming and costly process whereas the PLD are programmable logic chips that can be configured and made similar to the PCI interface.

This paper presents the implementation of the functionality of a PCI bridge on FPGA SPARTAN-III Target device using the HDL Verilog language. Timing Simulation is done with Modelsim Simulation software. The implementation of PCI Bridge on FPGA has led to easy reconfigurability, improved performance, less time to market and cost effective designs. The synthesis report shows that the area of utilization on FPGA is about 45 % which would save power by about 30%.

Key Words: HDL, Modelsim, PCI

* E&TC dept., Bharati Vidyapeeth's COEW, Pune, India, (e-mail: pbmane6829@rediff.com)

** E&TC dept. Bharati Vidyapeeth's COEW, Pune, India (e-mail :sumati_jagdale@yahoo.com)

1. Introduction

The notion of bridging plays a significant role in PCI architecture due to electrical limitations on the number of devices residing on a single PCI bus segment. There are three basic types of PCI bridges: PCI host controllers, transparent PCI bus bridges, and nontransparent PCI bus bridges. The PCI-to-PCI bridge provides a connection path between two independent PCI buses. A PCI bridge (Fig. 1) has two PCI interfaces. The bridge functions as a target and also as a master interface during the transaction between the buses [1]. PCI Bridge allows relatively slow local bus designs to achieve 132 MB/s burst transfers on a PCI bus. The Bridge can be programmed to connect directly to multiplexed or non-multiplexed 8, 16 or 32 bit local bus. The 8- and 16- bit modes enable easy conversion of ISA (Industry Standard Architecture) design to PCI.

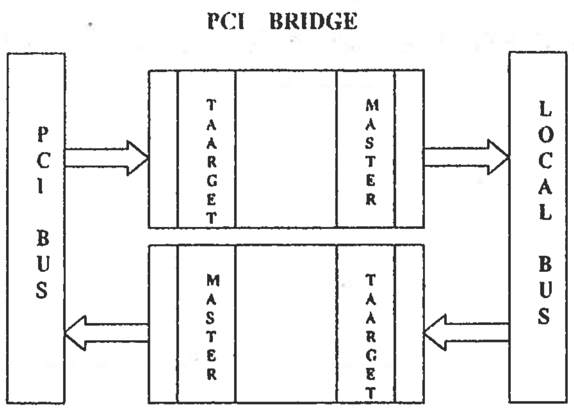


Fig. 1 Block diagram of PCI Bridge

1.1 Supported PCI Features

List of the main features supported by the PCI Bridge

i) 32-bit/33Mhz data bus

ii) Parity is generated and possible errors are checked and reported to back-end application.

iii) Type zero Configuration Space.

iv) Burst zero-wait configuration read and write cycles

v) Dual-access support: the Configuration Space is accessible from the PCI and the back-end side[2].

2. The PCI bridge architecture

The PCI Bridge architecture (Fig.2) consists of main modules such as Master, Target, Configuration space and Parity Generator/Checker, which consists of a generic PCI bridge core and user dependent parts such as Target Data Interface, Dual-port memory and Host bus interface.

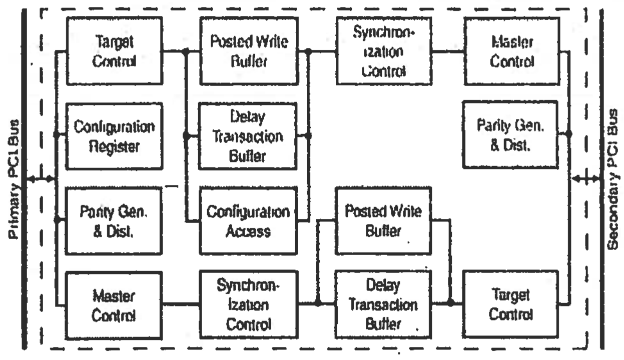


Fig. 2 PCI Architecture

2.1 Functional Description

- *Master*: an agent that initiates transactions on the PCI Bridge; it drives commands on the address phase requesting write or read accesses to one of the three address spaces of the PCI Bridge i.e. Configuration, I/O, Memory [3],[2].

- *Target*: the slave, which claims and responds to the transaction initiated on the PCI Bridge by a master agent.

- *Configuration Space Registers*: When a machine is first powered on, the configuration software must scan the various buses in the system to determine which devices exist and what configuration requirement they have. This process is called scanning of the bus.

- *Parity:* is generated and possible errors are checked and reported to back-end application.

3. Implementation of PCI bridge

All modules from PCI Bridge Architecture are implemented on FPGA SPARTAN III device for the verification of standard PCI specifications. All modules are written in behavioral style architecture and combined with the structural style architecture. Timing simulation of all modules is done in Modelsim simulation software.

3.1 Implementation of Master module:

The PCI Master interface connects the Master module with the Local Bus. This module is written in behavioral style architecture using FSM (Finite State Machine) It consists of Master State Machine, AD/CBE Buffer, Parity Generator/Checker, FIFO IN & FIFO OUT.

Master state Machine: Master state machine used FSM technique for executing sequential operation as shown in Fig.3.

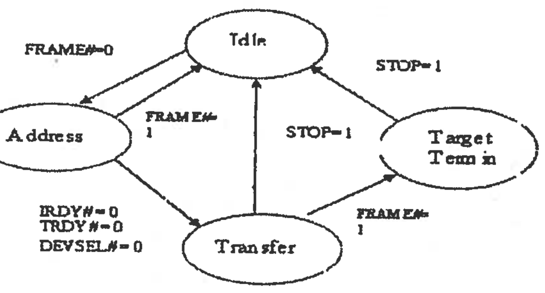


Fig. 3 PCI Master State diagram

i) When PCI-to-PCI Bridge is reset, PCI Master state machine is at IDLE state.

ii) It asserts REQ# and detects whether GNT# is active. If it is active, the master state machine enters into ADRESS state. It asserts FRAME# and IRDY#.

iii) When Master is in ADDRESS state it delivers Address, Bus command, byte enable. If IRDY#, TRDY# are in asserted form it enters in the TRANSFER state, else transaction is terminated.

iv) PCI read or write is carried on, if no suspension termination detected. The master state machine will return to IDLE state after transferring last data.

3.2 Implementation of the PCI Target module

The Internal Target Interface links the Target block of the PCI Bridge with the PCI bus and is a complete FIFO (First In First Out) interface. This module is written in behavioral style architecture using FSM (Finite State Machine). It consists of the Target State Machine, AD/CBE Buffer, Parity Generator/Checker, FIFO IN & FIFO OUT.

The Target state Machine: Target state machine transactions are performed using the Master state machine with different state transition conditions and operations.

Sequential operations are as follows.

i) When the PCI-to-PCI Bridge is reset, the PCI Target state machine is at IDLE state.

ii) When the Target state machine is in IDLE state and FRAME# is asserted it enters in the ADDRESS state.

iii) When the Target state machine is in ADDRESS state, it compares accepted address with its base address and if it does not match it returns into the IDLE state. Otherwise, it enters into the TRANSFER state.

iv) The TRANSFER state is carried on by asserting DEVSEL# and TRDY# signals, according to the command operation. If desired conditions are not matching, the Target state machine returns to the IDLE state. Otherwise, transaction termination is done with the last data transfer.

3.3 Implementation of the Configuration Space

The configuration space has the configuration registers and handles read/write accesses to them. It also manages the address decoding for Memory and I/O cycles. The PCI configuration space is divided into two parts- the configuration header and the device specific configuration register. In the configuration space format the first 16 dwords are referred to as configuration header from 64 dwords.

3.4 Implementation parity generation / checking

The parity block generates the parity and checks possible errors for the Master side of the PCI Bridge. The agent that is driving the PCI AD bus has to calculate and manage the parity line of the Bridge, which is called PAR. The master drives PAR during the address phase of each transaction and the data phase of write operations. The number of "1"s on AD, CBE# and PAR equals an even number. This block also manages the checking of possible parity errors and drives PERR# signal during read data phase; it works during write, as well, to report error information to the FIFO OUT block [4].

4. Simulation and Synthesis Result Of PCI Bridge Module

All modules such as the Master, Target, Configuration Space and parity are simulated with Modelsim Simulation software version 5.4 and implemented with Xilinx synthesis software version 9.1 on FPGA.

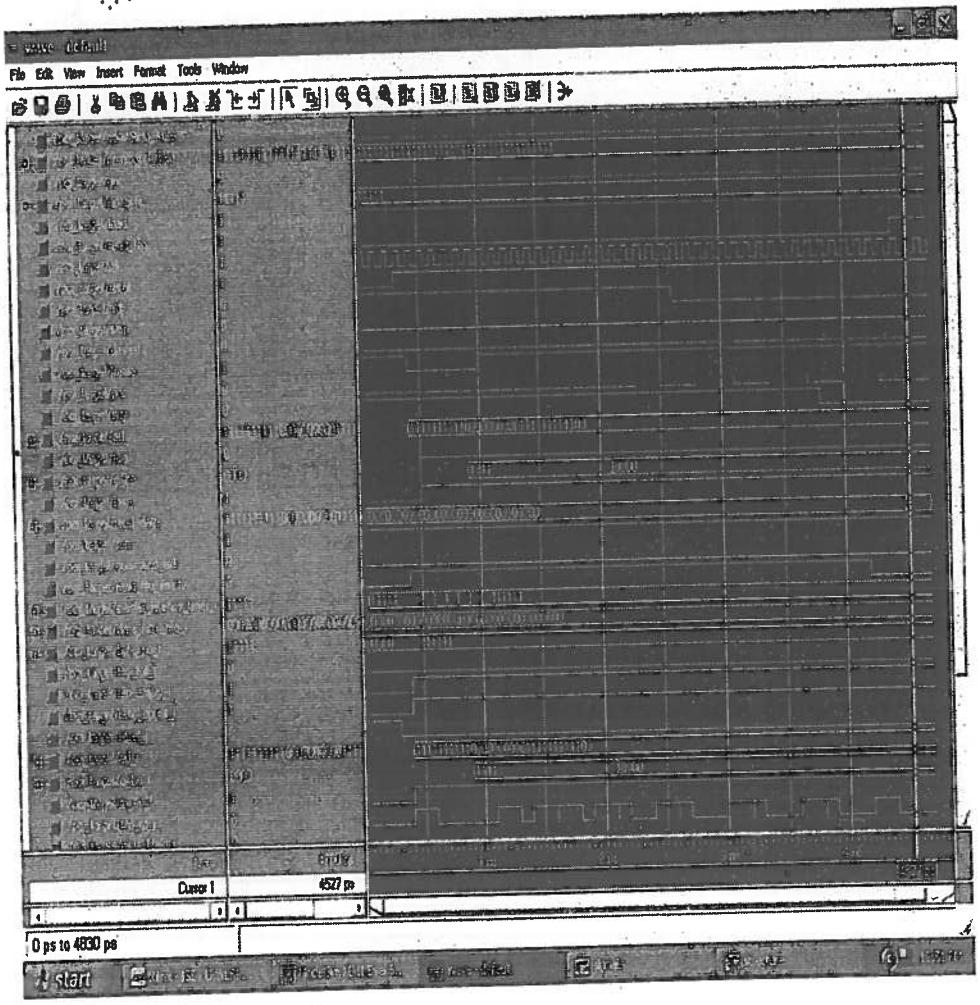


Fig. 5 Simulation Result

4.1 Synthesis results

	Master Module	Target Module	Configuration Space Module	Parity Module
No of Flipflops	110	104	20	1
No of LUTs	160	152	45	12
No. of IOs	97	97	50	41

5. Discussion

The simulation result (Fig.5) shows the write transaction of PCI interface Module. The 32 bit address and data is transmitted with 33 MHz frequency. The 4 bit command/ byte enable indicates the write transaction. The transaction starts when the control signal 'FRAME' is asserted. The control signals 'IRDY' (Initiator Ready) and 'TRDY' (Target Ready) should be low to indicate that the initiator and target are ready to transmit and receive. The Initiator transmits the address, command/byte enable and data that are received by the Target. The Parity Generator/Checker generates and checks even parity and indicates it on the PERR (Parity Error) signal.

The synthesis results show that the number of flipflops, LUTs, IOs required for all of the above-mentioned modules is reduced due to the optimization of the code.

5. Conclusion:

This paper presents the implementation of functionality of the PCI 32-bit Bridge using VERILOG. It is implemented on a Spartan-III FPGA device. The PCI bridge described in this paper is limited to 32 bits and 33 MHz operating frequency but both the number of bits and the operating frequency can be extended. The implementation of PCI Bridge on FPGA has led to easy re-configurability, improved performance, less time to market and cost effective designs. The synthesis report shows that the area of utilization on FPGA is about 45 % that saves about 30% power.

Acknowledgment

We thank the University of Pune for funding our project titled PCI Bridge.

Reference:

[1] Karl Wang, Chris Brynt," Designing MPC105 PCI Bridge Memory Controller", IEEE transaction, April 95, vol.15.pp.44-49.

[2] Zang Chunha, Shen Changli,"Non Transparent PCI-to-PCI Bridge Based on Verilog And FPGA",IEEE conference on communications, circuits & systems ,2006,vol. 4,pp. 2282-2285.

[3] Tom Shanely , Don Anderson, PCI System Architecture (fourth edition)

[4] "Useful Tips for the PCI System Designer" A white paper at Techonline.com, June 2004.

[5] http://www.plxtech.com

[6] http://www.Eurekatech.com

Design and Implementation of a Cross Compiler

Pinaki Chakraborty*

Abstract

This paper describes the design and the implementation of a cross compiler. The cross compiler has been developed to serve as a teaching tool capable of demonstrating the process of compilation to students in a stepwise manner. The cross compiler has been implemented in the C++ programming language. It uses a subset of the C++ programming language as the source language and the Intel 8085 assembly language as its object language. The cross compiler comprises of five phases, viz., lexical analyzer, syntax analyzer, intermediate code generator, code optimizer and code generator. The bookkeeping module and the error handler module have been also implemented. The parser used here is nondeterministic in nature where the order of application of the production rules is determined only at the runtime. The nondeterminism has been realized by associating priorities with the production rules. Three simple heuristics have been used to improve the performance of the parser. Apart from the cross compiler, the system includes an editor and a help subsystem.

Keywords: Compiler, cross compiler, parser, shift-reduce parser, nondeterministic parser, heuristic.

*School of Computer and Systems Sciences, Jawaharlal Nehru University, New Delhi 110067, E-mail: pinaki_chakraborty_163@yahoo.com

1. Introduction

A course on compiler construction is essential in any curriculum of computer engineering. The importance of the subject is well established. Every subject needs some simple experiments that can be used for teaching the basic concepts of the subject to the students. Unfortunately, there are not many such suitable experiments in compiler construction. Consequently, a simple compiler that can demonstrate the process of compilation in a stepwise manner will be immensely helpful to the students [1, 2]. The current paper presents a simple cross compiler developed to serve such a purpose. Special care has been taken while designing the compiler so that intermediate results are available to the users. These intermediate results can be useful in illustrating the process of compilation.

It can be recalled that a compiler is a program that takes as input a program written in a high level language, called the source language, and produces as output a functionally equivalent program in a low level language, called the object language or the target language [3 5]. A compiler that runs on one machine and produces object programs for another machine is known as a cross compiler. The current paper presents the design and the implementation of a cross compiler called the iXC85. The word iXC85 is an acronym for Intelligent Cross Compiler for Intel 8085. The cross compiler generates object programs for Intel 8085 processors and uses artificial intelligence techniques to do so. Hence the name "iXC85".

This paper does not present any new technique for compiler construction. Instead, the focus of the paper is on employing and thus verifying the standard concepts of compiler construction to design and implement a novel cross compiler. Section 2 presents the overall system structure. Section 3 discusses the implementation details of the various phases and modules of the iXC85 cross compiler. Section 4 provides brief accounts of the accessories of the cross compiler and Section 5 concludes the paper.

2. System Organization

The system has been developed as an Integrated Development Environment (IDE). The IDE consists of three components, viz., the iXC85 cross compiler, the Editor 85 editor and the help subsystem (Fig. 1). The iXC85 cross compiler has been written in the C++ programming language and compiled using the Turbo C++ version 3.0 compiler from Borland International, Inc. The executable file is named iXC85.EXE. The cross compiler accesses a file named RULES.TXT for purposes explained later in Section 3.3. The Editor85 has been implemented using Microsoft Visual Basic version 6.0. The executable file is named E85.EXE. It uses a data file named iXC85.DAT to store the environmental settings of the editor between two consecutive sessions. The help subsystem has been developed using the Microsoft Help Workshop version 4.03. The help topics written in rich text format (.RTF) files have been compiled to obtain a help file named iXC85.HLP.

When a typical source file *filename*.CPP is compiled using the iXC85 cross compiler the object program is obtained as an executable file *filename*.EXE. The iXC85 cross compiler may also produce eight text files, OUT1.TXT through OUT8.TXT, containing the outputs of the various phases and components of the cross compiler in a human readable format.

3. The Cross Compiler

Compilers are complex programs [5]. The entire task of compilation is generally divided into a number of phases. Logically similar tasks are clubbed into one phase. To some extent, the number of phases and the tasks handled by each phase depend on the developers. The iXC85 cross compiler consists of five phases that work sequentially (Fig. 2). The phases are the lexical analyzer, the syntax analyzer, the intermediate code generator, the code optimizer and the final code generator. Apart from these phases, a preprocessor, a bookkeeping module and an error handler module have been also implemented in the cross compiler.

In the iXC85 cross compiler, each phase generates its output in two forms (Fig. 2). One form is used by the next phase and the other form is meant for the user. While the first form contains the information in a compressed binary format, the latter contains the same information in a textual format intelligible to the user. However, the generation of the latter may be controlled by providing proper options as command line argument to the cross compiler.

The iXC85 cross compiler has been implemented in the C++ programming language [6, 7]. A subset of the C++ programming language, defined as C85++, acts as the source language of the cross compiler while the Intel 8085 assembly language is the object language. Therefore, in compiler construction terminology the iXC85 cross compiler can be represented as.

C85++, C++ Intel 8085 Assembly Language

There are two ways to use the iXC85 cross compiler. It can be invoked to compile a program from the Editor85. Alternatively, the iXC85 cross compiler may be called from the command line with the name of the source file and other arguments. The source directory, the include directory, the output directory, the base address of the object program, the base address of the runtime stack, etc. may be provided optionally with the call.

The structure and the working of the preprocessor, the five phases and the two modules of the iXC85 cross compiler have been explained in the rest of this section.

3.1. The Preprocessor

The preprocessor reads the source program one character at a time. It replaces all macro statements in the source program with proper source language statements. The iXC85 cross compiler supports the '#include' macro. The '#include' macro statement is handled by including the mentioned file. There are two modes for including files. If the name of the file is mentioned within a pair of '<' and '>', then the include directory is searched for the file. Otherwise, if the name of the file is mentioned within a pair of double quotes, then first the source directory is searched for the file. And, only if the file is not found there, the include directory is searched. Using this macro, one or more files can be included. After handling the macro statements, the preprocessor deletes the comments and strips out the redundant white spaces. Both single line comments, preceded by a '//', and multiple line comments, enclosed between a pair of '/*' and '*/', are supported in the C85++ language.

The preprocessor reports errors on encountering an unknown macro, an erroneous macro syntax, or if the file mentioned with the '#include' macro is not found. On success, the preprocessor generates a version of the source program without macro statements, redundant white spaces and comments.

3.2. The Lexical Analyzer

To develop the lexical analyzer, four finite automata have been first constructed. They have been designed to accept the keywords, identifiers, constants, and operator and punctuation symbols, respectively. The lexical analyzer takes as input the output of the preprocessor and tries to group the characters into tokens using the four finite automata. The finite automata are always used in the order to recognize keywords, identifiers, constants, operators and punctuators. If one of them fails to recognize a token, the next one is invoked at the same position of the source program. The four finite automata work independently of each other. If the lexical analyzer recognizes a token of the identifier type, then a bookkeeping module is called to install the identifier in the symbol table if it is not there already.

A lexical error is reported when the sequence of characters in the source program cannot be grouped into any legal token. On success, the lexical analyzer generates a stream of tokens as its output. In this stream, each token is represented by a pair of a type and a value field (Fig. 3b).

3.3. The Syntax Analyzer

The output of the lexical analyzer is fed to the syntax analyzer. To develop the syntax analyzer or the parser, a context free grammar for the source language has been first constructed. The grammar contains 17 nonterminals, 50 terminals and 66 production rules. The nonterminal 'program' is the start symbol of the grammar. According to their usage, the production rules can be broadly classified into three categories as follows.

1. *Program related production rules.* They specify the overall structure of a program.

2. *Expression related production rules.* They are used to construct expressions using the different operators.

3. *Statement related production rules.* They define different types of statements in a program.

The parser used in the iXC85 cross compiler is a nondeterministic shift-reduce parser. It tries to reduce the source program to the start symbol. The term nondeterministic signifies that the order in which the production rules are applied is dynamic instead of being predetermined. The nondeterminism comes from associating priorities to the production rules in a way similar to the one used by Humenik and Pinkham [8]. The production rules are matched in the descending order of their priorities. To add to the nondeterministic behavior, the priorities of the production rules change dynamically. The priorities of the production rules are whole numbers between 0 and 899. The initial value of the priorities of each production rule is predetermined. These initial values are assigned to the production rules according to their probability of being used. The priority of a production rule increases on being used recently or if its use in near future is predicted by some other production rule. However, the priority of a production rule cannot be allowed to increase infinitely. To ensure this, the production rules are divided into 18 disjoint classes, each with a distinct priority range. A production rule in a class can have up to 50 values for its priority. Since, the priority of a production rule can only increase and there is no mechanism to decrease its value, it may easily reach the maximum limit of its class. Consequently, this prioritization approach works very well for small programs but becomes less effective with the increase in the program size.

The entire parsing process ~~revolves~~ around applying production rules iteratively to parse the source program. Since the number of production rules is not small, there may be too many permutations. But if studied carefully, it can be noticed that all production rules are not equally likely to be applied at a particular instant of parsing the program. To narrow the search some heuristics can be used. Three simple heuristics [9], as follows, have been realized in the parser.

- ***Heuristic 1.*** Some production rules inherently have more probability of being used than other production rules. So each production rule is assigned an initial priority according to its chance of being used. The production rules are applied in the descending order of their priorities. Therefore, while applying the production rules, a production rule with a greater priority is given preference over another with a lower priority.

- ***Heuristic 2.*** If a particular production rule is applied to parse a program, it is generally done so for more than once. So, every time a production rule is applied, its priority is incremented by one. Thus the most frequently used production rules gain priority over the others.

- ***Heuristic 3.*** It can be observed that some production rules are generally used in cascade. The application of a production rule in such a cascade can predict the use of the next production rule in the sequence. In such a case, the priority of the predicted production rule is increased by two.

These heuristics speed up the parsing of a large number of source programs and they can be expected to work well for an average source program. Hence, it can be said that the use of these heuristics makes the cross compiler intelligent. These heuristics also take care of the precedence and the associativity of operators defined in the source language.

The production rules have been encoded so that the parser can easily read them and modify their priorities. Accordingly, each type of terminals and nonterminals is assigned a unique 3-digit numeric code that appears in the sentential forms while parsing. Each production rule is represented as a sequence of its rule number, initial priority, predicted next production rule, nonterminal in left hand side of the production rule and the terminals and nonterminals on the right hand side of the production rule. Each of these values is denoted as a 3-digit code. The production rules thus encoded have been stored in a file named RULES.TXT and are accessed by the parser before the start of the parsing process. The production rules have been provided separately from the main program so that a user can modify the initial priorities and the predicted next production rules of the production rules to parse their programs more effectively. Thus, the assortment of production rules can be considered analogous to a knowledgebase of an expert system.

The parser starts its operation by reading the production rules form the file RULES.TXT and sorts them according to their initial priorities. Then the parser reads a stream of tokens from the output of the lexical analyzer and stores them in a buffer. Next the parser selects the production rule with the highest priority and tries to match its right hand side with a portion of the content of the buffer. If they match, then that portion of the buffer is replaced by the left hand side of the production rule. If they do not match, then the same procedure is repeated with the next production rule. Finally, the priorities of the production rules are updated and sorted. This process is carried out iteratively until no more reduction is possible. Then the parser declares success if and only if the buffer contains only the start symbol. Otherwise, the error handler module is called to handle a parsing error.

The syntax analyzer reports an error if the file named RULES.TXT is not found. Another type of error that may be reported by the syntax analyzer is a parsing error. On success, the parser generates the parse tree as its output (Fig. 4). The parse tree is represented by a list of nodes in which each node is denoted by a type, production rule used to derive it, number of children nodes and a list of children nodes, if any (Fig. 3c).

3.4. The Intermediate Code Generator

The intermediate code generator uses the information stored in the nodes of the parse tree to produce three address instructions. The three address instructions are realized as quadruples using a record structure of four fields, viz., `op`, `operand1`, `operand2` and `result`. Such a record typically stands for an operation of the form `result := operand1 op operand2`. An important benefit of using quadruples is that they allow the statements to be moved around in the code optimization phase. The intermediate code generator creates a new temporary object to store an intermediate result whenever required. The temporary objects thus created are inserted in the symbol table. However, there may arise situations where the symbol table gets cluttered up with a large number of temporary objects.

On completion, the intermediate code generator generates a sequence of simple machine independent instructions as its output. Each instruction is represented by a quadruple (Fig. 3d). In these instructions, a temporary name is denoted by a '$' symbol followed by a natural number. And the position of an identifier in the symbol table is enclosed between a pair of '['and']'.

3.5. The Code Optimizer

The code optimizer sequentially reads the instructions produced by the intermediate code generator. It improves an instruction or a few consecutive instructions if there is a scope to do so. In this cross compiler only some trivial machine independent optimization techniques have been implemented.

On completion, the code optimizer generates another sequence of simple instructions as its output (Fig. 3e). This sequence of instruction is an improved version of the output of the intermediate code generator. However, on some occasions there may not be any scope of improvement and the two sequences of instructions are identical.

3.6. The Code Generator

The final code generator sequentially reads the instructions produced by the code optimizer and produces their Intel 8085 assembly language equivalents. A single intermediate language instruction may be translated into one or more Intel 8085 assembly language instructions. Unlike the instructions produced by the intermediate code generator, these instructions are machine specific and deal with the actual registers of the target processor.

On completion, the code generator generates the aforesaid sequence of Intel 8085 assembly language instructions as its output (Fig. 3f). This sequence of instruction is functionally equivalent to the source program written in the C85++ language.

3.7. The Book keeping Module

The bookkeeping module includes a symbol table which is implemented as a linear list of records. Each record consists of a known number of consecutive words in the memory. In this approach, the cost of entering a new name or making an inquiry is $O(n)$, where n is the number of entries already in the symbol table.

Whenever the lexical analyzer recognizes an identifier it calls a bookkeeping routine to install the identifier in the symbol table if it is not already installed. The identifiers recognized by the lexical analyzer are permanent objects. The intermediate code generator may call the same bookkeeping routine to install some temporary objects in the symbol table. These temporary objects are generated by the compiler and do not appear anywhere in the source program. The names of these temporary objects consists of a '$' symbol followed by a natural number. The temporary objects are similar to the permanent objects in all other respects. The bookkeeping module generates a copy of the symbol table at the end of the compilation process as its output (Fig. 3g).

3.8. The Error Handler Module

Whenever any phase or module of the cross compiler finds itself in an out of order situation, which it cannot deal with, it calls the error handler module with proper arguments. The error handler used in the iXC85 cross compiler is a trivial one. On detecting an error, it just generates an error message and stops all activities of the cross compiler. The error messages refer to the source program only and not any internal representation of it. To help the user in debugging the programs, the error messages are kept simple and easy to understand.

4. The Accessories

As mentioned in Section 2, the IDE consists of two utility programs apart from the iXC85 cross compiler. The editor and the help subsystem aid the use of the iXC85 cross compiler. Brief accounts about them are given next.

4.1. The Editor

The Editor85 is a simple text editor program intended to facilitate the use of the iXC85 cross compiler. It assists in the development of efficient programs that may be compiled using the iXC85 cross compiler. It supports activities like creating new programs, editing programs, compiling programs and printing them. Editor85 is able to explicitly specify the base addresses of the object program and the runtime stack. It can also display the output of each phase individually whenever asked by the user.

The Editor85 is a GUI based editor which acts as an interface between the user and the iXC85 cross compiler. It is a menu-driven program and it is easy to use. It also supports a toolbar that can be customized at the runtime. The Editor85 supports its own set of shortcut keys and some standard Windows shortcut keys. It supports Multiple Document Interface (MDI) so that the user can work with more than one source file at a time.

Although Editor85 is specially developed to support the use of the iXC85 cross compiler, other editors or IDEs can also be used to develop programs for the iXC85 cross compiler. The performance of the iXC85 cross compiler is not affected by the use of such tools.

4.2. The Help Subsystem

The help subsystem provides help and support on 55 topics covering different aspects of the iXC85 cross compiler that can be navigated using 220 internal links. Whenever a topic is viewed, links to topics related to it are also displayed. It also supports indexing and searching activities. Hall and Zeck compressions have been used to reduce ~~the size~~ of the help file ~~by~~ about 36.82%.

5. Concluding Remarks

The development of the iXC85 cross compiler has been an interdisciplinary venture requiring the knowledge of compiler construction, artificial intelligence and microprocessor systems. It has been an academic endeavor that can immensely help students and researcher to understand the design and the working of a simple compiler. The iXC85 cross compiler also helps the students to learn to write programs in Intel 8085 assembly language and determine the difference between machine generated programs and hand written programs. Accordingly, the iXC85 cross compiler can be applied as a pedagogical tool for teaching several courses ranging from microprocessor systems to compiler construction and system programming. Apart from its academic applications, the iXC85 cross compiler can be used to effortlessly develop assembly language programs for commercial microprocessor applications. The iXC85 cross compiler can be beneficial to naive as well as expert programmers.

To further enhance the capabilities of the iXC85 cross compiler, the following tasks may be undertaken.

1. The source language may be extended to include different data types and high level language constructs like structures, classes and templates.
2. The grammar may be enhanced by adjusting the priorities of the production rules.
3. Extensive code optimization techniques may be included.
4. The compiler can be improved to generate more efficient object programs by removing redundant load

and store instructions. This will require addressing the register allocation and register assignment problems in the code generator.

Acknowledgements

The author would like to acknowledge S. Chandana-Taneja and late Prof. R. G. Gupta for supervising this research.

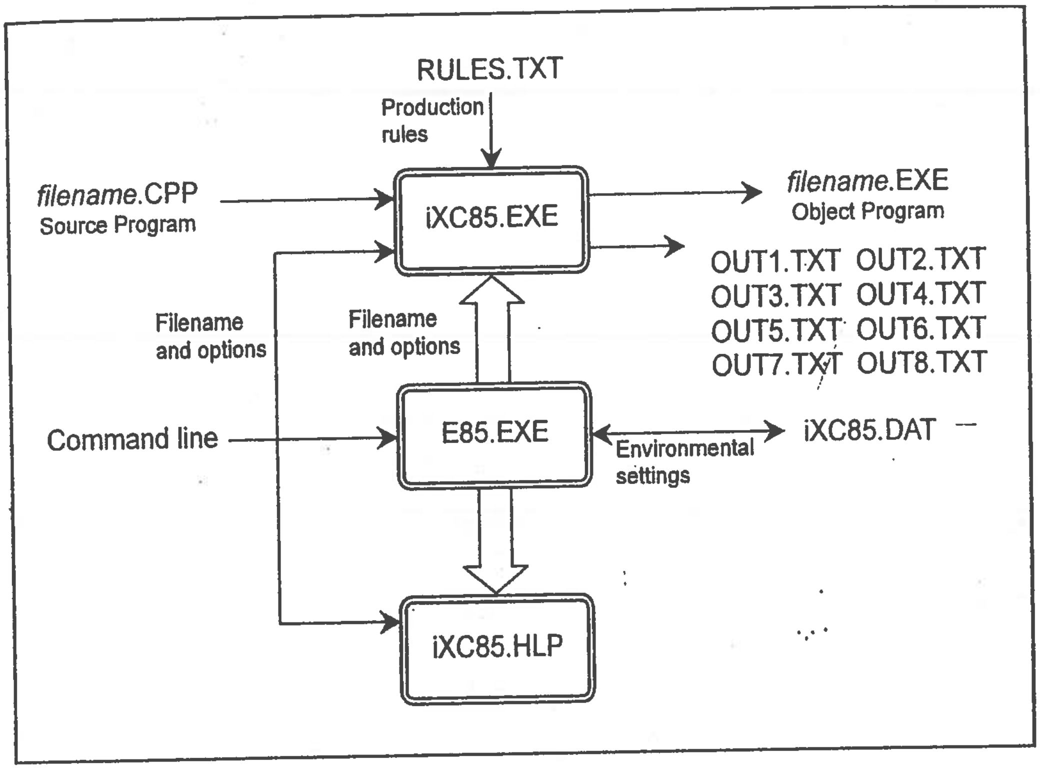


Fig. 1. The overall system organization.

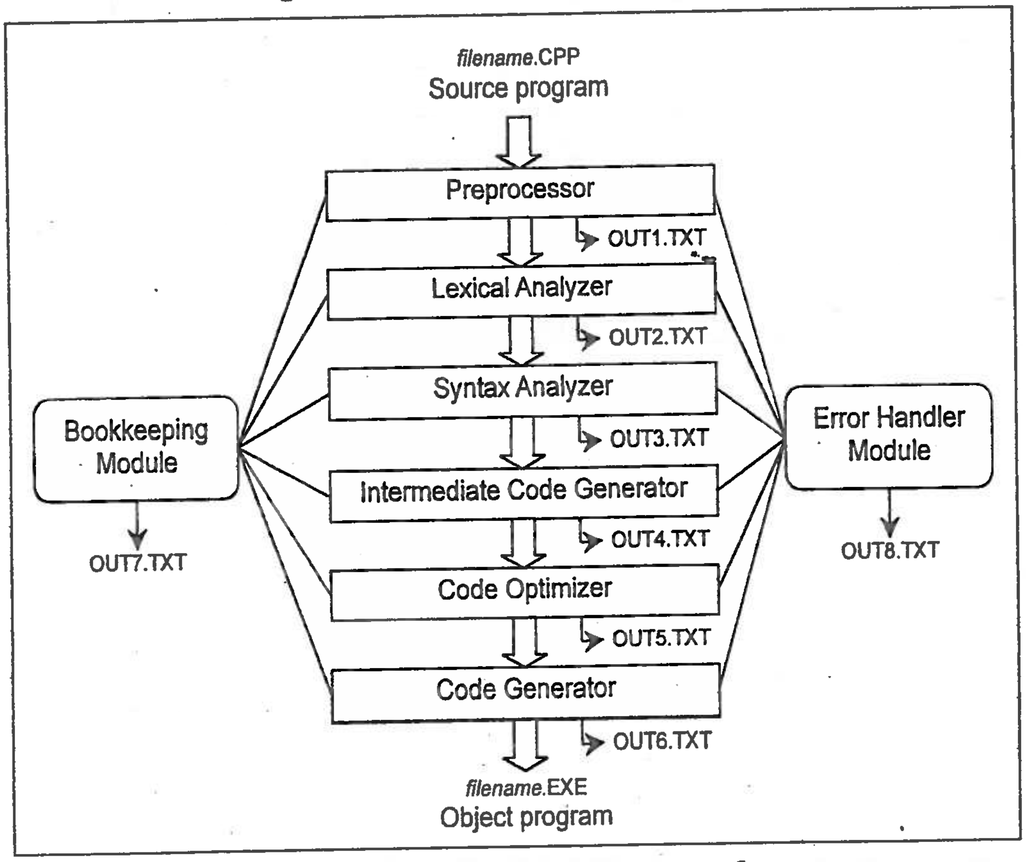
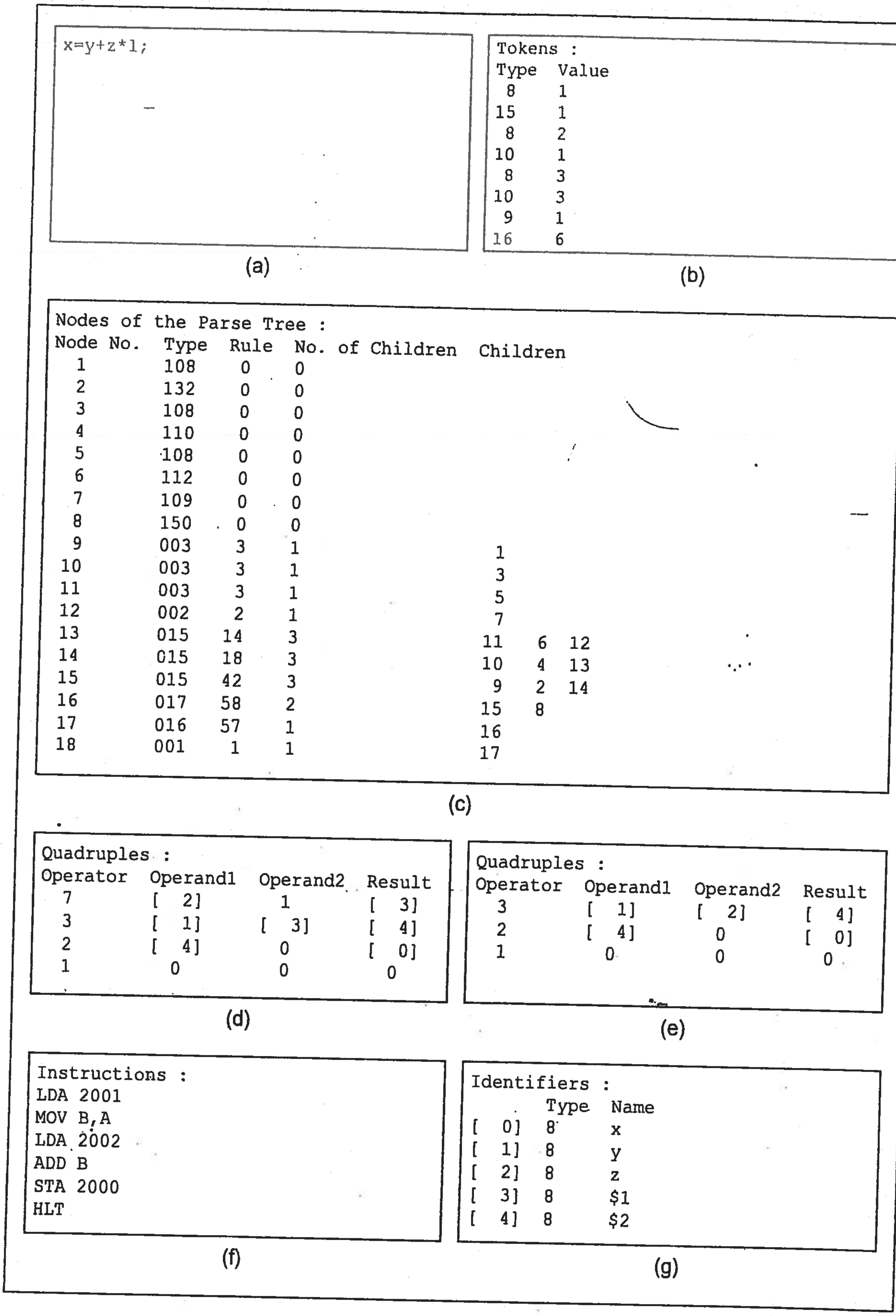


Fig. 2. Phases and modules of the iXC85 cross compiler.



```
x=y+z*1;
```

```
Tokens :
Type  Value
 8     1
15     1
 8     2
10     1
 8     3
10     3
 9     1
16     6
```

```
Nodes of the Parse Tree :
Node No.  Type  Rule  No. of Children  Children
  1       108    0     0
  2       132    0     0
  3       108    0     0
  4       110    0     0
  5       108    0     0
  6       112    0     0
  7       109    0     0
  8       150    0     0
  9       003    3     1                 1
 10       003    3     1                 3
 11       003    3     1                 5
 12       002    2     1                 7
 13       015   14     3                11   6  12
 14       015   18     3                10   4  13
 15       015   42     3                 9   2  14
 16       017   58     2                15   8
 17       016   57     1                16
 18       001    1     1                17
```

```
Quadruples :
Operator   Operand1   Operand2   Result
  7        [   2]        1       [   3]
  3        [   1]     [   3]     [   4]
  2        [   4]        0       [   0]
  1           0          0          0
```

```
Quadruples :
Operator   Operand1   Operand2   Result
  3        [   1]     [   2]     [   4]
  2        [   4]        0       [   0]
  1           0          0          0
```

```
Instructions :
LDA 2001
MOV B,A
LDA 2002
ADD B
STA 2000
HLT
```

```
Identifiers :
         Type   Name
[   0]   8      x
[   1]   8      y
[   2]   8      z
[   3]   8      $1
[   4]   8      $2
```

Fig. 3. Compilation of (a) a sample fragment of source code to obtain the outputs of (b) lexical analyzer, (c) syntax analyzer, (d) intermediate code generator, (e) code optimizer, (f) code generator and (g) bookkeeping module.

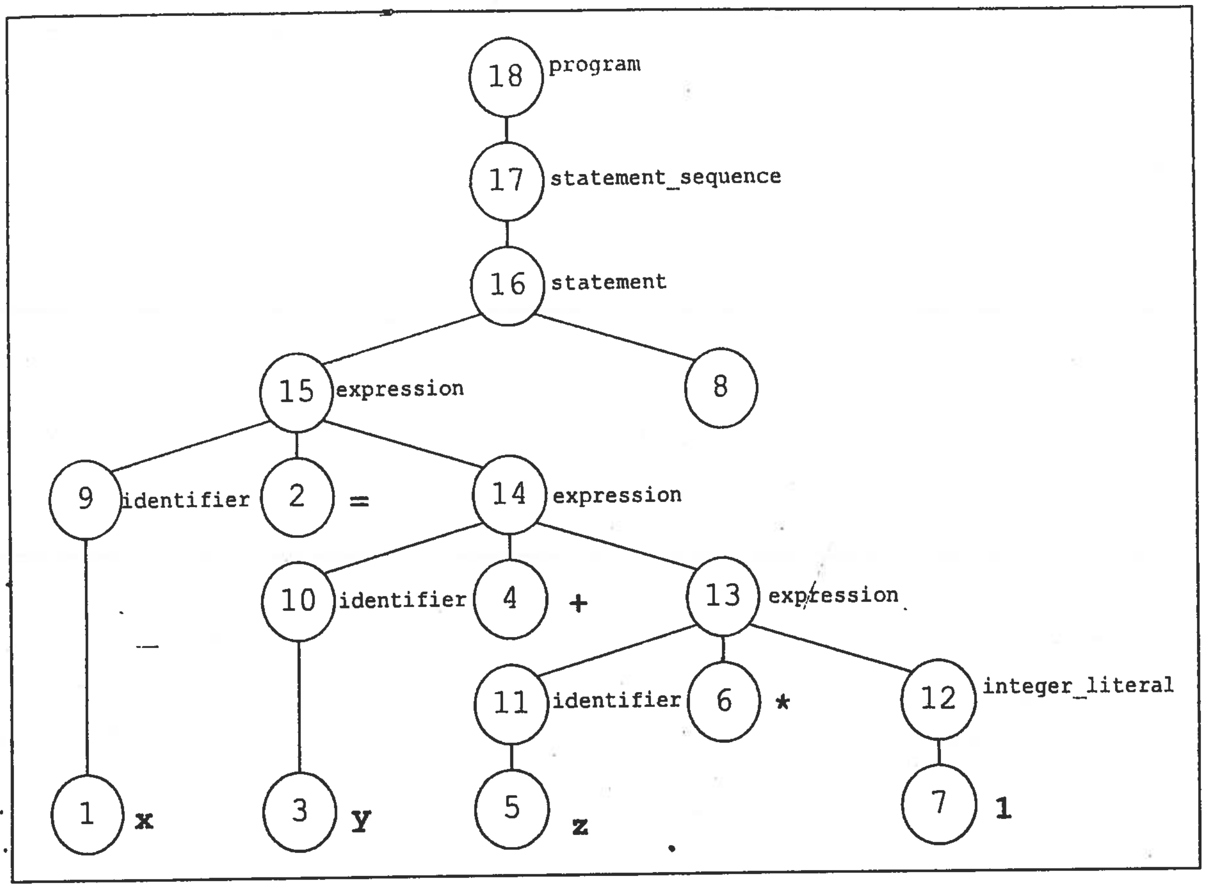


Fig. 4. Parse tree generated by the syntax analyzer for the given source code fragment.

References

[1] Chakraborty, P. 2007. A language for easy and efficient modeling of Turing machines. *Progress in Natural Science*, **17**(7): 867-871.

[2] Chakraborty, P. and Gupta, R. G. 2008. A simple object oriented compiler, *Proceedings of the National Conference on Information Technology and Competitive Dynamics*, pp. 203-215.

[3] Aho, A. V. and Ullman, J. D. 1977. *Principles of Compiler Design*. Addison-Wesley.

[4] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D. 2007. *Compilers: Principles, Techniques and Tools*. Addison-Wesley.

[5] Holub, A. I. 1990. *Compiler Design in C*. Prentice-Hall.

[6] Stroustrup, B. 1986. *The C++ Programming Language*. Addison-Wesley.

[7] ISO. 2003. *Programming Languages C++*. ISO/IEC 14882:2003.

[8] Humenik, K. and Pinkham, R. S. 1990. Production probability estimators for context free grammars. *Journal of Systems and Software*, **12**(1): 43-53.

[9] Chakraborty, P. 2008. Use of heuristics in shift-reduce parsers. *Proceedings of the International Conference on Data Management, pp 103-109.*