Visual Studio

DTMF

Project Report

DEVELOPER STUDIO

Dual Tone Multiple Frequency Detector

Dissertation for Partial Fulfillment of requirements of the National University of Sciences and Technology for award of B.E degree in Software Engineering

Syndicate

Capt Muhammad Mukarram
Capt Mohsin Abbas
Capt Muhammad Haider Zaidi

Acknowledgments

We would like to extend comprehensive gratitude for Almighty Allah who bestowed upon us the great will. The inquisitive essence granted by Him has facilitated the resolve to consummate this work.

We wish to appreciate the valuable exhortation, persistent encouragement, and sustained support as regards supervision to Dr Noman. Without his guidance and help the completion of this work would have been impossible. We would also like to mention here his patience to bear our numerous visits and telephone calls.

Finally we would extend our appreciation to our session-mates who have always given unending support, in the past three and a half years that we have been together. It was only by the desire to excel of the entire class that today by the grace of God we have attained enough knowledge to step into the challenging world of information technology. We would also extend our gratitude to our families who tolerated our extreme commitments through out our degree.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract

The objective of this project was to develop and implement software simulator for a specific Micro engine. Working on a simulator for Dual Tone Multiple Frequency Detector, gave an immense opportunity to understand the working and the architecture of micro engines, as well as that ofcompiler development.

The Simulator is supposed to be testing software, enabling the designer to verify the performance and optimality of the design, as well as check the clock cycles for instructions executed as per the design constraints. The simulator software emulates the exact behavior of the micro engine. Instructions are read into the Instruction register enabling the different modules to work on the control signals, and values in the register file and memory updated accordingly.

The Input to the simulator is to be in the form of binary values as the machine only understands its machine code. This requirement imposed the development of compiler and assembler software for the specific dual tone multiple frequency detector. The purpose of these two is primarily to convert the source program which shall be in the form of a C file to the machine code, which could be fed into the simulator.

Chapter

1

1 INTRODUCTION

A total computer system includes both hardware and software .Hardware consists of the physical components and all associated equipment. Software refers to the programs that are written for the specific micro engine. Although the possibility of not being aware of the hardware and writing the software or vice versa exits, but when the process of validation of the hardware and of generating the machine language is concerned, knowledge of both the aspects is overriding.

The scope of this project is primarily to exercise the knowledge of computer architecture for the understanding and development of a simulator software for a micro engine, recently designed and applying the knowledge of compiler design, to produce a compiler and assembler for this new designed micro engine. A complete integrated environment, of these three modules, to provide the user with the combined tools of incorporating a Simulator and a Compiler plus assembler for the micro engine, is the goal of this project. Subsequent chapters identify the approach to the problem and the design parameters.

The need for a new micro engine for which these tools are being developed was underlined due to the need of a high performance hardware for the detection of DTMF (Dual Tone Multiple frequency). Dual Tone Multiple Frequency signaling is used in telephone dialing, digital answering machine, computers and even in interactive banking system. DTMF signaling represents each symbol on a telephone touchtone keypad (0-

9,A,B,C,D,*,#) using two sinusoidal tones, as shown in the figure on the next page . When a key is pressed, a DTMF signal consisting of a row frequency tone plus a column frequency is generated and transmitted.

The process of decoding this signal is performed at the exchange. The Micro processors installed at the exchange performs number of operation to detect this signal. The usually employed micro engines work on extensive floating point operation and thereby require heavy computing power. The new design would work on an algorithm which operates on fixed point operation written by Amey A.Deosthali , Shawn R. McCaslin and Brial L.Evans, which was presented to IEEE in October 1999 for an optimized LOW – COMPLEXITY ITU COMPLIANT DUAL TONE MULTIPLE FREQUENCY DETECTOR.

D T M F T O U C H T O N E S C H E M E

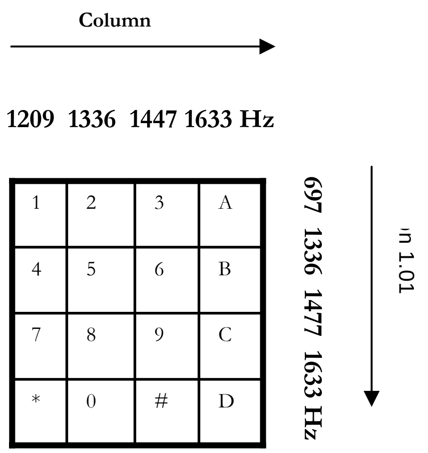


Figure 1.DTMF SCHHEME

1.1 OBJECTIVEs.

The objectives, defined by the hardware design team were to develop software which would be able to check the design and working of the new micro engine. Checking of the hardware design regarding it's operations, are of utmost importance, as the price of the first fabrication is extremely high. It could easily climb to figures like 2 million dollars. The expenditure entails the thorough and detailed working analysis, of any new design. For this the Designs of new micro engines or hardware chips go through extensive checks.

Simulator is also one of the methods to check the working of the hardware design. This enables to check the various modules of the hardware design against the input parameters passed to the simulator. It is to perform as per the design and comply with any constraints the design may have. Each module of the micro engine enabled and selected according to the control bits and different operands selected. Simulator software would actually emulate the behavior of the designed micro engine.

Micro engines as known, operate on machine instructions. Machine instruction inside the computer or the micro engine form a binary pattern, which if not impossible, is extremely difficult to hand code. The requirement of emulating the new design required that exact binary pattern be given as input to the simulator and each module would work according to the machine instruction. It is preferable to write programs with the more familiar symbols of alphanumeric character set , rather than hand coding binary value. As a consequence, the need for translating user-oriented symbolic programs into binary programs recognized by the hardware (based of the hardware design) necessitated the development of an assembler.

The requirement of the assembler was still short of providing a complete integrated environment to complete the package. Today, when high level language are available, it is preferred working on them. Assembly code is also relevantly difficult as these languages provide the user a platform to write sequence of statements in a form that people prefer to think in, when solving a solution. However, each statement must be translated into a sequence of binary instruction before the program can be executed in a computer. Therefore the need of a compiler working on a subset of C (as per the design constraints) was felt, in order to complete a package.

The task as defined above led to the selection of C and Visual C for the implementation of the software. All three modules of the integrated environment use the MFC classes of Visual C. For the purpose of carrying out analytic study of the problem and to come up with a detailed design, Unified modeling language was selected and Rational Rose software was used.

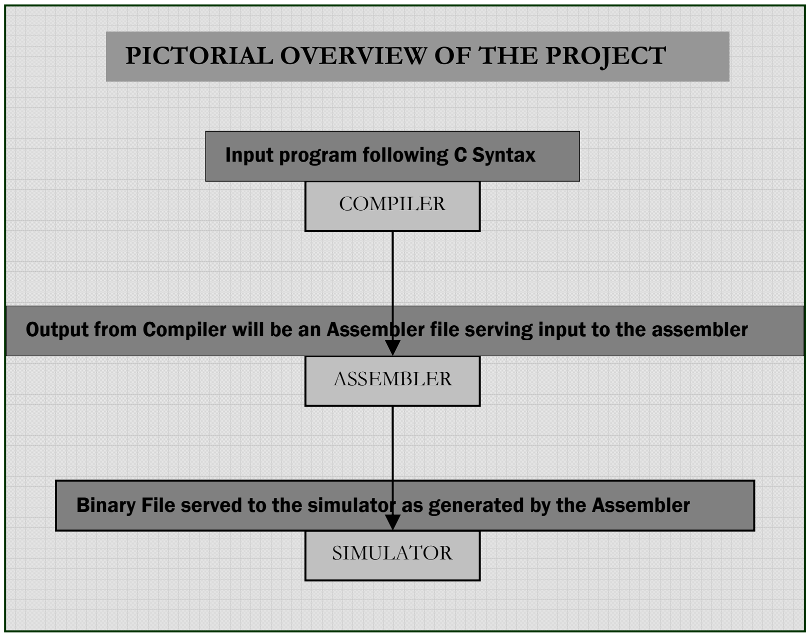

PICTORIAL OVERVIEW OF THE PROJECT
Input program following C Syntax
COMPILER
Output from Compiler will be an Assembler file serving input to the assembler
ASSEMBLER
Binary File served to the simulator as generated by the Assembler
SIMULATOR

Figure 2:- PICTORIAL VEIW OF THE PROJECT

1.2 Report Layout.

This report has been divided into three chapters. The first chapter gives brief introduction of the subject.

The second chapter of the report primarily covers the Simulator. It explains the brief design of the micro engine for which this simulator has been designed. Explaining the functional units in brief and the class diagram of the simulator, which was the foundation for the development of the software for the simulator.

The third chapter of the report addresses issues related to the development of the compiler and assembler. The instruction set of the micro engine and the technique adopted to implement the design. It also describes the Grammar of the assembler and that of the compiler.

The forth chapter primarily describes the way to operate the software. The user who wishes to use the software will find the relevant details of the operating, all the three modules in this chapter.

Chapter

2

2 SIMULATOR

The primary objective of the Project involved development of a simulator softawre. This program would receive an input of a binary file format as generated by the assembler, and would simulate the working of each instruction on the design parameters of the DTMF engine. The value in the Register file and Address register including those of the data memory and program memory being updated, and displayed at each clock cycle.

Simulator would actually operate on each instruction as it would handle each instruction. The addition, multiplication, shifting and loop control is handled as these instructions are being run on DTMF engine. This micro engine is designed to have one fetch stage, followed by first execute and later the second execute stage. The simulator is to run in two modes of operation, a debugger mode and a full simulation mode. The first one is a step by step process in which each instruction is processed and the result displayed after each clock cycle. In the full simulation mode, it operates on the entire input giving the result, in its register file or as it may be.

The subsequent pages would be explaining the design based on different procedures adopted..

2.1 Micro Engine Architectures and Working.

Numerous conventions or designed are followed as regards the architecture of the micro engines. The first evolutionary design parameter were developed by Von Neumann in the early forties of the twentieth century, know as the Von Neumann Architecture.

Von Neumann architecture takes approx eight cycles to process a complete instruction. These cycles or steps actually involve the steps to read the instruction from the memory and execute the instruction. In the simplest fashion, it could be best explained as fetch and execute . First fetch the instruction, as it would tell the micro engine what exactly to do and next steps involve "what to do". The Instruction format therefore is also extremely important as this format should be of the underline machine on which the instruction are actually to be performed.[ACH]

The usual steps proceed by first reading the instruction into the instruction register (A register inside micro engines which holds the instruction to be processed). This is fetched from the memory, another register is therefore used to keep track of which instruction to be fetched next, and this register is termed as the PC or the program counter. Unless told otherwise the Program counter is incremented in a sequential manner in order to read the next instruction by default. This although may not be the requirement each time, as the instruction of Jump or Conditional Jump could be encountered. In that case the controller (a module) of the micro engine works to identify where to jump and read the instruction for. Controller is also used to identify the number of loops as given in loop statements.

The next stage is use to enable the working of the different modules/ components of the computer. This may involve the following action to be performed.

- CPU-Memory : Data may be transferred from CPU to memory or from memory to CPU.
- CPU-I/O : Data may be transferred to or from the outside world by transferring between the CPU and an Input/Output module.
- Data Processing : The CPU may perform some arithmetic or logic operation on data.
- Control : An Instruction may specify that the sequence of the instruction be altered as explained in the para above.
- Or a Combination of these actions.

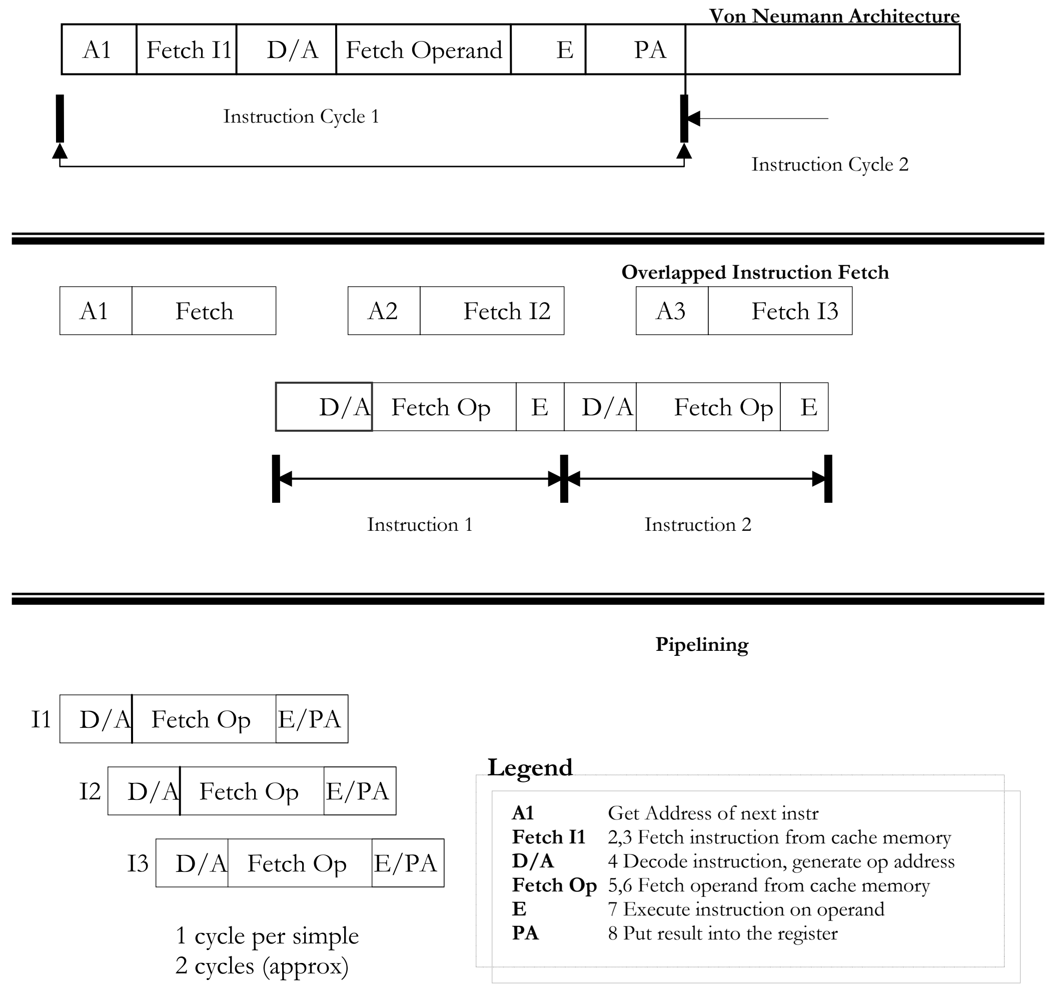


Figure 3: CYLES PER INSTRUCTION [a]

The figure above shows the different clock cycles per instruction taken by the different architecture. The first one is the Von Neumann Architecture, which takes 8 cycles per simple instruction. Followed by the Overlapped instruction fetch which takes 4 cycles per simple instruction.. The DTMF Micro Engine has been based on the pipelining architecture. The first instruction would take three clock cycles to be processed by subsequently each instruction would require only one clock cycle. This achieves immense speed and performance over the Von Neumann architecture. [PDP]

The architecture of a micro processor in general consists of modules like arithmetic unit, shifter and controllers. A summary of the major units used in the DTMF micro engine are given below.

Register File. Used to store the operands on which the different operations are to be performed.

ALU. The Arithmetic and logic unit is that part of the computer which actually performs the arithmetic and logical operations on the data. All other components of the computer system are there to mainly bring data into the ALU for it to process and then to take the result back. Operations like Addition, subtraction, multiplication , division are performed. The Logic Unit performs binary operations like OR, XOR, AND. Shifter operation like shifting the binary bits are performed by the Shifter logic Unit.

Comparator. Used as part of the controller primarily to execute the Conditional Jump statements.

Controller. The specific job of determining the address of the next instruction.

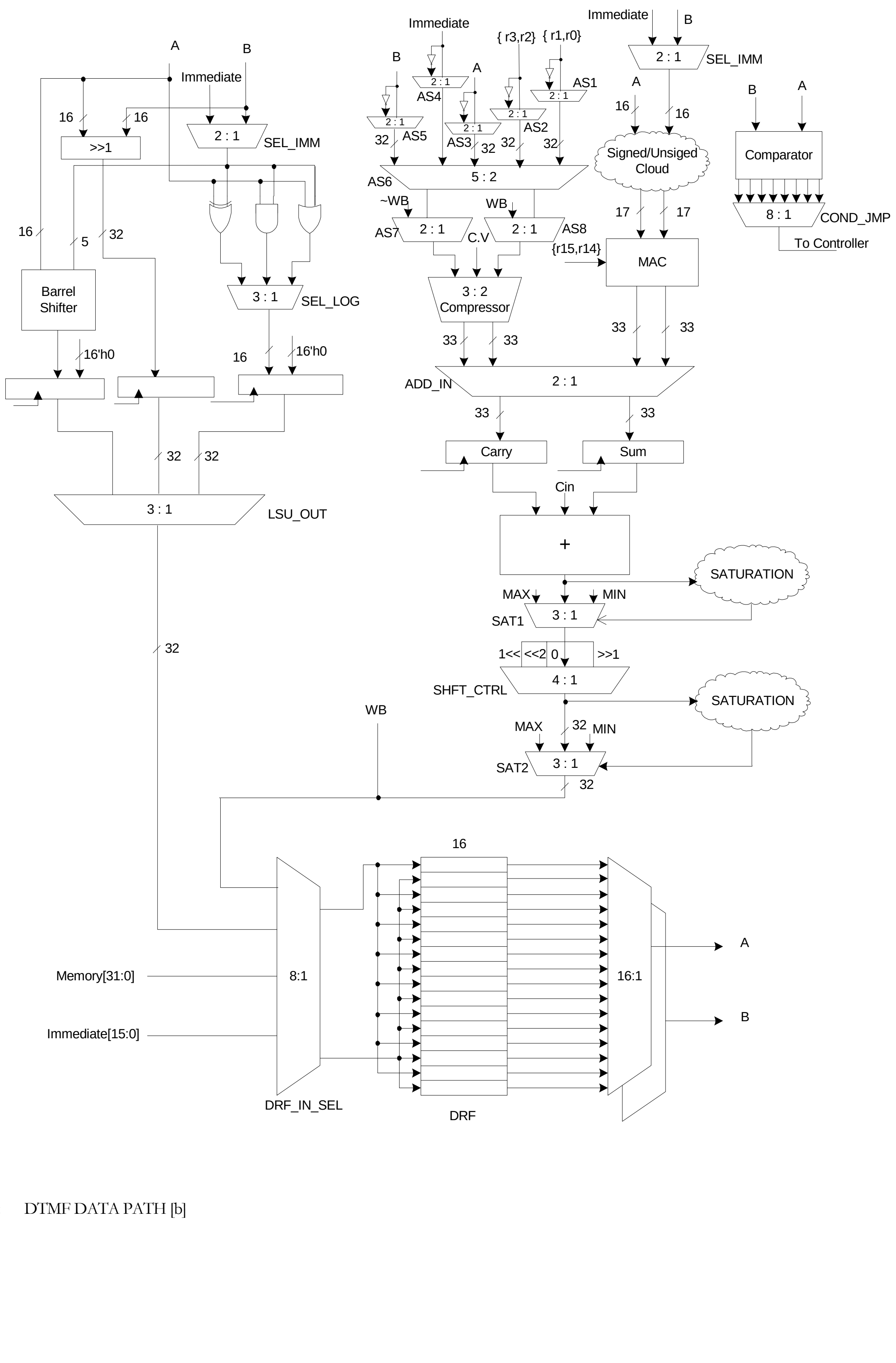


Figure 4: DTMF DATA PATH [b]

2.2 Functional Units and Data Path

The entire design of the DTMF micro engine is explained below. This design was the base for the development of the Simulator software. [DTMF]

The design of the DTMF micro engine has three main functional units as listed below.

- Addition/Subtraction and MAC unit.
- Logic and shifter unit.
- Comparator.

2.2.1. Addition / Subtraction and MAC Unit

This is a unit which is connected in parallel and is a combination of a MAC unit and an add./Sub unit connected in parallel. The carry and sum outputs of these units are selected as per the operation.

MAC Unit. The MAC unit is capable of 16 bit signed and unsigned multiplication and accumulation. The two operands are fetched from the DRF (Data Register File) through mux A and mux B. A 6 bit immediate operand can also be fed as an operand to the Mac Unit instead of operand B. For accumulator purpose the last two registers of the register file, register 14 and 15 are used.

Add/Sub unit. The add/sub unit with an adder can perform the following operations:

- Add with Saturation
- Add with carry and Saturation
- Subtract with Saturation
- Subtract with Borrow and Saturation

The selection of these operations are selected by Correction vector(CV). The add/sub unit is capable of handling either 32 bit or 16 bits operand. The first four registers of the data register file are reserved for the two 32 but operands. These operands can be directly to the add/sub unit. If 16 bit addition is required the two operands are loaded from the register file directly or a 6 bit immediate can also be fed instead of operand B.

Adder. In the Execution phase 2, the 33 bits carry and sum from the respective registers are added together in a 33 bit adder. The output of the adder is input to the saturation unit, which checks the overflow of the addition output. The adder has an option of carry , along with the provision of carry out. This can be used when carry after the addition of two numbers is again used in another operation

Saturation unit. The responsibility of this logic is to control the maximum and the minim range values. As the operation of Adder or that of the shifter may yield a result outside the allowed range. This can be checked and compensated by the saturation unit.

Shift unit. This instruction is used to load a 32 bit memory data in two consecutive register of DRF. Here i= 0,2,4 to 14 and j= 0 to 7. The LSB of the address from the ARF verifies whether to take the upper 32 bits or lower 32 bits.

2.2.2. LOGIC UNIT

This unit is connected in parallel with the Add/Sub and MAC unit. It consists of the logical unit and a Barrel shifter connected in parallel. The inputs to this unit are two 16 bits operands A and B fetched from Mux A and B or a 5 bit immediate value instead of operand B.

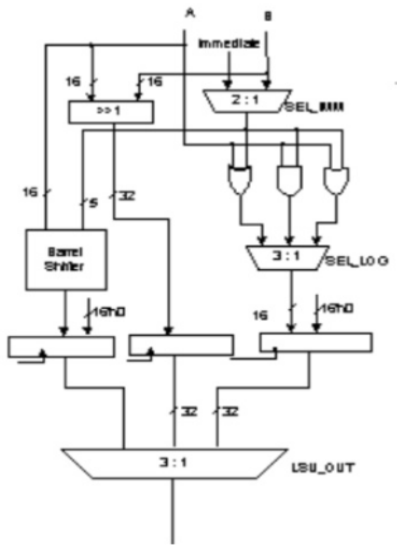


Figure 5: LOGIC UNIT[d]

Logic Unit. This unit primarily performs simple logical operations. Three logical gates XOR,AND, and OR , are connected in parallel. The selection of which is done through a multiplexer.

2.2.3. SHIFTER UNIT

The shifter unit consists of two types of shifters connected in parallel. The first one is a simple 32-bit right shifter and the other is barrel shifter .

32-bit Shifter. This unit shifts the input (32 bit) by one bit to the right. Both operands are combined to make a 32 bit input data. By this shift operation the LSB of operand A replaces the MSB of operand B and the sign of the operand one is extended.

Barrel Shifter. The barrel shifter is placed in parallel with the logical unit. The input to the Barrel shifter is a 16 bit signed/unsigned value which is called shift operand. The barrel shifter can perform a maximum of 16-bit left or 16 bit right arithmetic/logical shifts on 16-bit signed/unsigned operand. The control signal for the shifter consists:

- **Shift Value :** Encoded in 5 bit 2,s complement and selects how much shift is to be performed
- **Shift direction :** A single bit control to select whether to shift in right or left direction
- **Shift type :** A single bit control to perform logical shift or arithmetic shift.
- **Saturation on/off :** A single bit control to select whether to use the saturation check or not on the shifted operand.

2.2.4. COMPARATOR

A comparator is connected in parallel with the above two units and is used for conditional jump instruction. The output of the comparator is sent to the controller for the execution of the jump instruction. The inputs are two 16 bits operands A and B fetched from the register file. The comparator is connected to an 8:1 multiplexer which is used to specify the jump condition for selection of comparator output to be sent to the controller.

2.2.5. ADDRESS GENERATION UNIT (AGU)

The AGU of the DTMF micro engine has a simple design consisting of eight address register. An increment / decrement unit and output multiplexers and an input de-multiplexer. Each register is 10 bits wide.

2.2.6. DATA REGISTER FILE

The micro engine has one data register file which constitutes of 16 Registers each having a width of 16 bits(Each register can hold a word of 16 bits). A 16 bit word and a 32 bit word although can both be written on these registers. For 32 bit operation two register (consecutive) are used, like register r0,r1 could

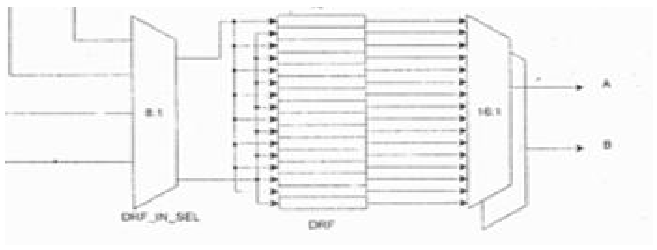


Figure 6: DATA REGISTER FILE [e]

both be used to store a 32 bit word. However for read out only the first two pair are reserved i.e. register r0 through to register r4. The two 16 bit operations can be read out from any one of 16 registers of DRF through MUX A and MUX B. The truth table for the two MUX is attached as Annex B.

2.3 Simulator Execution phases

The simulator has three Stages according to the design. These three stages are discussed in detail below.

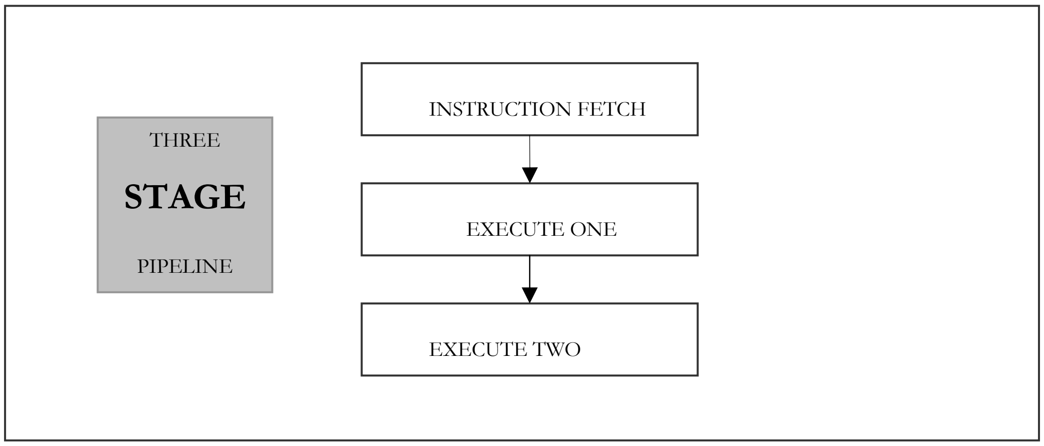


Figure 7: STGES OF DTMF

2.3.1. Instruction Fetch

This process is the first in the simulator which enables the Instruction Register to be loaded with the instruction. The program memory is loaded with the binary file generated by the assembler. As the instruction is fetched into the Instruction register the next phase is set ready to be executed

2.3.2. Execute Phase 1 (First)

The second stage involves the following steps:

- **Operand select signals for Data path include**
    - Data Register File read port select signals (for MUX A and MUX B)

    - Adder/Subtractor mode signal
    - MAC Signal
    - Logical unit signal

- **Control signal to Shifter unit**
    - 6-bit immediate value and select immediate signal.
    - Control Signals for Loop Setup (including loop setup signal, end address and the loop count value to be stored in the respective registers).
    - Control signal for Loop start (repeat bit)
    - Control signals to next address generation and the 11 bit absolute value for jump (conditional/unconditional)
- **AGU control signals including**
    - Address register file read port select signals
    - Address registers write enables in case of increment/decrement
    - Inc/dec signal for AGU Increment/Decrement unit
- **Memory**
    - Read and write signals
    - Address to memory for load/store
    - Data to memory for storing

2.3.3 Execution Phase 2 (EX 2)

The Third stage of Execution 2 involved the following steps.

- Data Register File Write enabled
- Write enable signals
- Input (from memory, add/sub and MAC unit, Logic and Shifter unit, immediate) select signals
- Adder signal
- Select Logic/Shifter signal .)
- ARF write enable in case of load from memory or move from ARF.

2.4 Design And Implementation

2.4.1. Requirements Necessitated

The necessary requirements for the simulator as envisaged by the design and requirement criteria are defined below:

- Data register File values to be viewed at each clock cycle.
- Address register value to be available at each clock cycle, and checking the working of the AGU.
- Relating the instruction handling by each functional unit of the design against desired results
- Cycle count for instruction
- Depict entire working of DTMF Design giving the values of Program counter, Stack pointer and Instruction Register at each cycle.

2.4.2. Instruction Splitting

The next step requires that the instruction must be read and the value of each subset be read out, and stores separately. Value of the instruction register is temporarily saved in order to read all the separate control signals and values of operands or immediate value as the case may be. As the IR is read it is also required to identify which type of instruction it is .Therefore it is imperative that all different combinations of instructions are read. The subject micro instruction has four major formats for the instruction (combinations are shown below)

2 bits [31:30]	2 bits [29:28]	1 bit [27]	3 bits [26:24]	4 bits [23:20]	9 bits [19:11]		2 bits [10:9]	4 bits [8:5]	5 bits [4:0]
Normal	Normal	Read/Write	Selection of DRF input	Selection bits of Op 1	Control signals to Add/Subtract and MAC units,AGU Comparator			Selection bits of Op 2	Storing Operands into DRF
Set Status	AGU Enable					6bit Immediate Operand			
	Acc Reset				[20:5]16 bit Immediate				
Repeat	Loop initial			9[24:16] Loop Count		[15:5] Loop end Address			
Jump			Jump Address MSB		[19:17] Jump Cond	8[16:9] Jump Address LSBs			

Table 1: Instruction Breakdown

The values of each subset like the instruction decoder 1 and 2 etc are all read into there respective register by masking the IR register value with the hexadecimal numbers. The method adopted is described below by extracting out the value of the instruction decoder 1 from the IR register.

- Store the value of the IR temporarily.
- Shift this value by a factor(times) of 30 to the right.

  IR >> 30

  Previous Value

  1000 0111 0000 0000 0000 0000 0010 0001

  After Shift

  0000 0000 0000 0000 0000 0000 0000 0010

- Store this value in the Instruction Decoder after anding it with (binary 3)

  0000 0000 0000 0000 0000 0000 0000 0011.

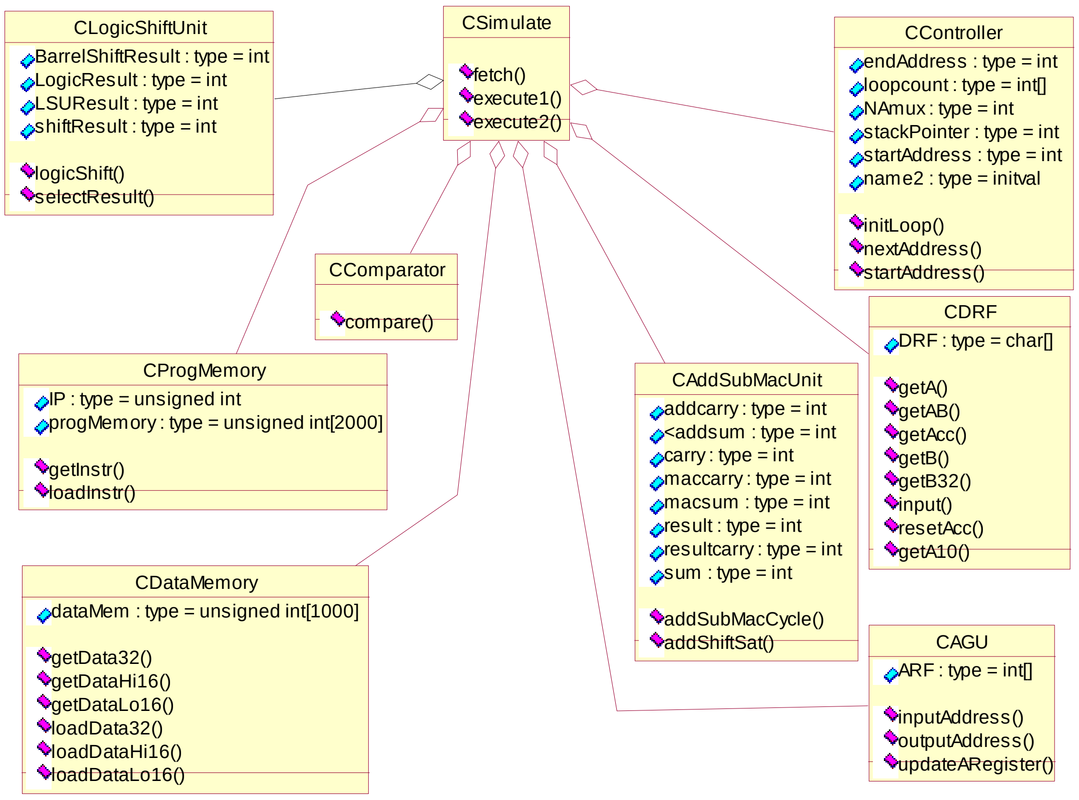
- The value of the Instruction Decoder 1 (first 2 bits) are extracted out of IR.

```
0000 0000 0000 0000 0000 0000 0000 0010
0000 0000 0000 0000 0000 0000 0000  0011
----------------------------------------
0000 0000 0000 0000 0000 0000 0000 0010
```

In this way all the possible value all extracted out. After analyzing the different parameter the relevant process required are activated. Like an entry of all zeros in the control field (9 bits) enable the simulator to understand that the last instruction has been reached and the program must be terminated. Refer to Annex O.

The instruction is decoded to enable the different modules of the micro engine, to operate and select the preferred input and output lines. To make this procedure simpler and follow the lines of the hardware design, for each component a class was made, which behaves similar to the hardware module. The next pages depicts the class diagram of the Simulator with brief explanation of each.

2.5 Class Diagram (UML) Simulator



Classes

- CSimulate
- CLogicShiftUnit
- CController
- CDataMemory
- CProgMemory
- CComparator
- CDRF
- CAddSubMacUnit
- CAGU

2.5.1. CLASS NAME :- CSimulate:-

Class Operation:-

Fetch
Execute1
Execute

Class Description:-

The entire operation of the simulator is controlled by this class. Principally this is the class which enforces the pipelining operation of the Simulator, that is the repetitive cycle of Fetch, and two execute cycles.

Fetch is responsible for reading the instruction into the instruction register from the program memory. Whereas all operations of Execute One, which are reading operands from the Data register, enabling Subtractor, Adder, or the multiplication unit. The operation of the Shifters and the comparator is also initialized in this cycle.

The Execute two is responsible for the carry and sum output of Add/Sub unit or Mac unit, which are added in adder and saturation unit works in this cycle. The output of the functional units is written to the Register File or memory.

2.5.2. CLASS NAME: - CDRF:-

Class Operation:-

getA()	getAB()	getAcc()	getB()	get32()	input()
resetAcc()	getA10()				

Class Description:-

This Class represents the Data Register File of the Simulator. The DRF has 16 register and this register file is depicted by the character array of 16. The entire array is equivalent to the DRF and is used to store the value of the Data Registers. These operations refer to the different requirement of the procedure involve in extracting this data out of the DRF

getA(), simply return the value of the operand 1, as does the operation **getB**() on the operand 2. **getAB**() return the concatenated value of operand one and operand two. Operand takes the last 16 LSB where as operand two takes first sixteen operands, returning a 32 bit value. **getA10**() concatenates the value of the first two registers and returns the concatenated value.**get32**() operation is performed to concatenate register two and three.

Accumulator is by default the last two registers of the DRF, these are register 14 and 15. Function **getAcc**() concatenates the result of the two operand of DRF 14 and 15. Whereas the **resetAcc**(), reset the accumulator to zero.

The input(), operation is required to input the values in the data register, that is to store the result in the data register. This operation is primarily performing the operation of the truth table given in the Annex N.

2.5.3. CLASS NAME: - CCONTROLLER:-

Class Operation:-

initLoop()
nextAddress()
startAddress()

Class Description:-

The controller of DTMF Micro engine does the job of a program sequencer. It is responsible for computing the address and then fetching the next instruction from the program memory. The next address instruction computation can use the decision from a conditional branch or from the zero-overhead looping logic or simply the next sequential instruction.

The Operations above are responsible for keeping track of the next instruction to be fetched the start and the end address including the number of loop counts that need to be processed. The function of the Controller could be best understood after studying the architecture of the controller.

The Controller has three main functions to perform. Fetch the next instruction, process a

conditional jump and or process a unconditional jump. For the later two cases it needs to keep track of the start and the end address as well as decrement the loop count as it is performed. The architecture is designed to support four nested loops therefore, it has the capacity to store these values in three stack register files as marked on the figure on the controller on the next page.

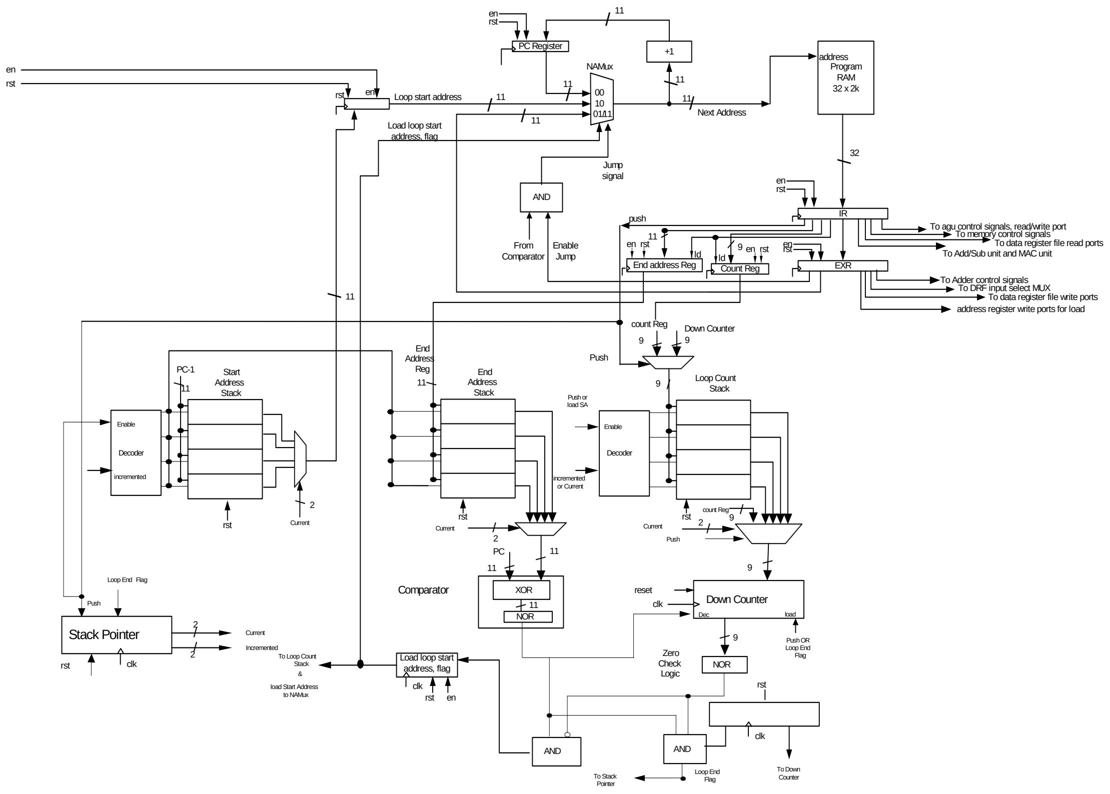


Figure 8: CONTROLLER OF DTMF MICRO ENGINE[f]

2.5.4. CLASS NAME: - CAddSubMacUnit:-

Class Operation:-

addSubMacCycle()
addShiftSat()

Class Description:-

The primary function of class CAddSubMacUnit is to exhibit the behavior of the Adder, the MAC unit. The operation **addSubMacCycle**(), select the required input, either from the MAC Unit or the adder unit. This also enable the Write Back Option if required. The write option is cycle stealing, instead of the result being fed back into the DRF or the Memory, it is directly feed back into the MAC/Adder.

This **addShiftSat**() operation is responsible for the shifting of the operands feed to it if required, and is also responsible for keeping a check on the range of the operation. If the range exceed the maximum allowed value or is less than the least allowed value, the operands are automatically assigned the highest or the least number. The Max allowed range is:-

Positive saturation (0x7fffffff)
Negative saturation (0x80000000)

2.5.5. CLASS NAME: - CAGU

Class Operation:-

inputAddress ()
outputAddress ()
updateARegister ()

Class Description:-

The AGU of the DTMF micro engine has a simple design consisting of an Address Register File (ARF), An Increment/Decrement unit, input and output multiplexers and an input demultiplexer.

The ARF consists of 8 registers of 10 bits width to store address of the data memory of 1k words. The feedback gives the provision of increment or decrement of an address from ARF. The AGU shown in the following diagram:-

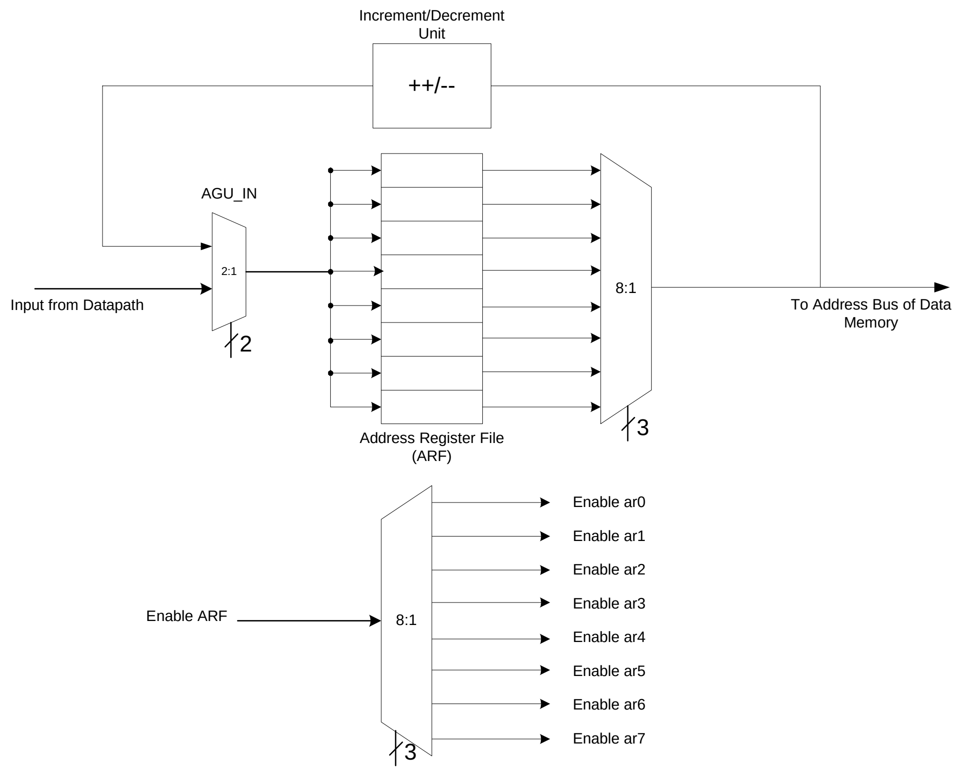


Figure 9:- ADDRESS GENERATION UNIT DTMF MICRO ENGINE

I

nputAdress() is required to store the value in the Address Register File. Whereas the operation **outputAddress**() is primarily responsible for reading the value stored in any of the Address Register. The last operation of **updateARegister**() is used to store the incremented or the decremented vale of the address register as the requirement may be.

2.5.6. CLASS NAME: - CDataMemory

Class Operation:-

getData32()	getDataHi16()	getDataLo16()
loadData32()	loadDataHi16()	loadDataLo16()

Class Description:-

Data Memory consists of single ported Synchronous RAM 4kB in size with 64-bit data bus. The operations of Class Data Memory are responsible for either reading from the memory 32 bit data or 16 bit data, Hi Word or Low Word, first sixteen bits or the last sixteen bits. Similarly the load operation loads the Data Memory with required data, as 32 bit or in case of High 16 or Low 16 bit, it is loaded to the desired address inside the data memory.

2.5.7. CLASS NAME: - CProgMemory

Class Operation:-

getInstru ()
loadInstr ()

Class Description:-

The function of this class is to read from the program memory or two write back to the program memory. These two operations are performed by the **getInstr**() and the **loadinstruction**() methods of this class.

2.5.8. CLASS NAME: - CLogicShiftUnit

Class Operation:-

logicShift ()
selectResult ()

Class Description:-

The Logic, shift unit is connected in parallel with the Add/Sub and MAC unit. This unit consists of a logical unit and a Barrel Shifter connected in Parallel. The inputs to this unit are two 16 bits operands A and B fetch through MUX A and B or a 5 bit immediate value sign extended to 16 bits which can be selected instead of operand B.

2.5.9. CLASS NAME: - CComparator

Class Operation:-

Compare()

Class Description:-

A comparator is connected in parallel with Register File location and Data Register File Input Select and is used for the conditional jump instruction. The output of the comparator is sent to the controller for execution of jump instruction, which is performed by the **compare**() operation. The inputs are two 16 bits operands A and B fetch from the register file. The comparator is connected to an 8:1 multiplexer which is used to specify the jump condition for selection of comparator outputs to be sent to the controller. The truth table of the conditional jump is as follows:

Bits	**Operation**
000	0 (No Jump)
001	A < B
010	A > B
011	A = = B
100	A < = B
101	A > = B
110	A ! = B
111	1 (Unconditional Jump)

Table 2: Comparator Truth Table

Chapter

3

3. COMPILER & ASSEMBLER

To feed any micro engine, the machine language is used. Strictly speaking machine language is a binary program of category one. Because of the simple equivalency between binary and octal or hexadecimal representation, it is customary to refer to category 2 as machine language. The task of writing machine language is extremely difficult and could be termed as next to impossible.[ACH]

In order to provide the programmers a better working environment, a programming language is defined by a set of rules. Users must conform with all format rules of the language if they are desirous of translating the programs correctly to the machine language of the micro engine they want to work on.

In today's environment the availability of high level languages and their use facilitate the working environment of the programmers. These are special languages developed to reflect the procedures used in the solution of a problem rather than be concerned with the computer hardware behavior. Examples of high level language include, BASIC, C/C++, and FORTRAN etc. However each statement must be translated into a sequence of binary instructions before the program can be executed on a micro processor. The job of converting the high level languages into machine languages is performed by compilers. A typical hierarchy of this process of conversion is shown in the diagram below, incorporating compiler and assembler together.

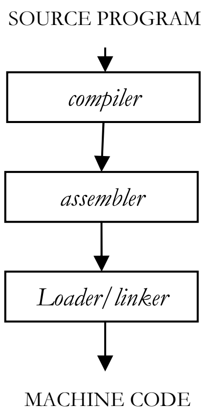


Figure : Language Processing Hierarchy

3.1 Phases of Compiler Development

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation another. A typical decomposition is shown in the diagram below.[CMP]

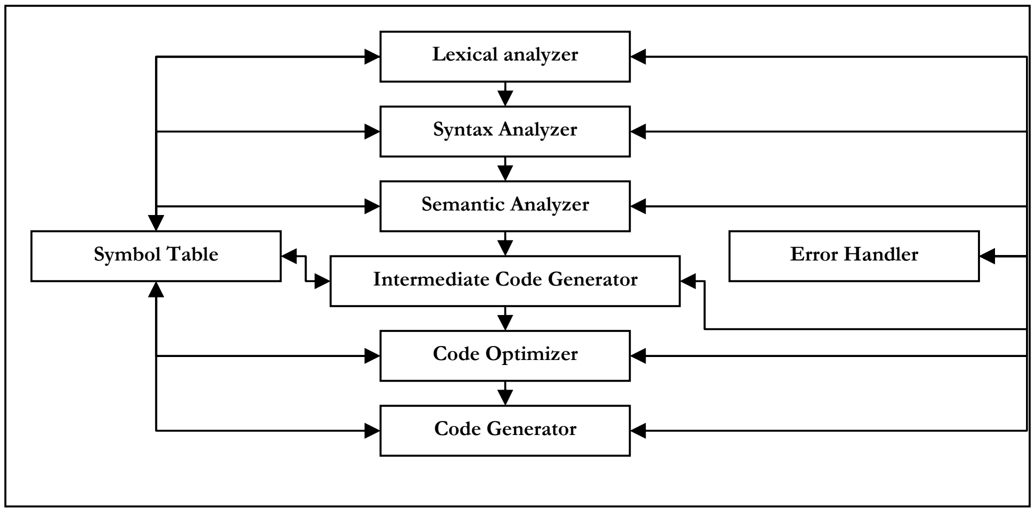


Figure 11: STAGES OF COMPILER DEVELOPMEWNT – *COMPILERS BY ALFERED V.AHO*

The first three phases, deal with the analysis portion of the compiler are discussed below.

3.1.1. Lexical Analysis

Linear Analysis is called lexical Analysis. In a compiler or even in an assembler the lexical analyzer converts the stream of input characters, which are given as the source program into a stream of tokens, which are given to the next phase for the syntax analysis. While talking about the lexical analysis, the terms, "token" and "pattern" have specific meaning . In most of the programming languages the following constructs are treated as **tokens** : keywords, operators, identifiers, constants, literal strings and punctuation strings. A **Pattern** is a rule describing the set of lexemes that can represent a particular token in the source program. A pattern for the keyword "if" would be the same keyword if , but for the "relational operators" a **pattern** would be all the relational operators i.e. <,>,<>,<=,>=,!=.

3.1.1.1. Role of the Lexical Analyzer

Lexical Analyzer's role primarily revolves along the question of identifying the tokens as depicted or as allowed by the Grammar of the language. Lexical Analyzer being the first phase of the design for assembler as well as the compiler primarily depends upon the Grammar of the language as defined. In the case of the assembler for the DTMF micro engine this is the instruction set as per Annex A.

Since the lexical analyzer reads the source text it also clears white spaces or comments (it ignores them). It also keeps tracks of the line numbers and errors that are reported by the lexical analyzer with the corresponding line number. Upon receiving a "get token" command from the parser the lexical analyzer reads the input characters till it can identify the sent token.

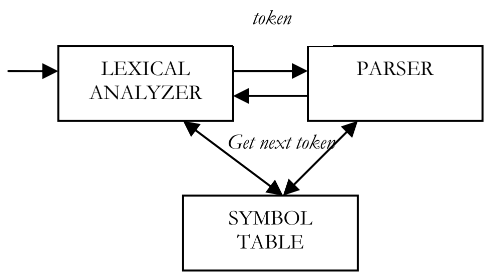


Figure 12: BLOCK DIAGRAM OF LEXICAL ANALYZER

In the statement :

If (rm < rn) goto label1

The lexical analyzer would analyze this statement and would in turn make different tokens for the parser as given below.

Keyword **If**
Register **rm.**
Register **rn.**
Keyword **goto.**
Label **label1.**

3.1.2. Syntax Analysis

Every programming language has rules that prescribe the syntactic structure of well formed programs. The syntax of the programming language constructs can be defined in different notation. The usual technique used is addressed as the context free grammars, or BNF notation. Grammars are capable of producing most of the syntax of programming languages, but not all there is a requirement of other types of check also. The syntax of the language can be checked by the grammar but for semantic rules different types of check like type check, name related checks etc are performed.[CMP]

3.1.2.1. Context Free Grammar

A grammar describes the hierarchical structure of many programming languages. For example an **if-else** statement in C has the form

If (expression) statement else statement

This implies , that the statement is the concatenation of the keyword if, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword else, and another statement. Using the variable *expr* to denote expression and *stmt* to denote statement , this structuring rule can be expressed as

Stmt → **if** (*expr*) *stmt* **else** *stmt*

In this statement the arrow can be read as can have the form. Such a rule is called a production. In a production lexical elements keyword if and parenthesis are called tokens, variables like *expr* and *stmt* represent sequence of token called the non terminal. A context free grammar could be defined as having four components

- A set of tokens , know as the terminal.
- A set of non terminals .
- A set of productions where each production consists of a non terminal called the left side of the production and an arrow, and a sequence of tokens and or non terminals, called the right side of production.

The above expression can be expressed in a parse tree as

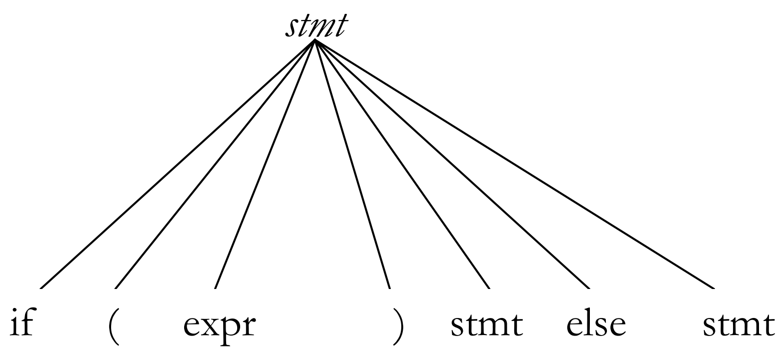


Figure 13:-Parse Tree

3.1.2.2. Ambiguity in Grammar

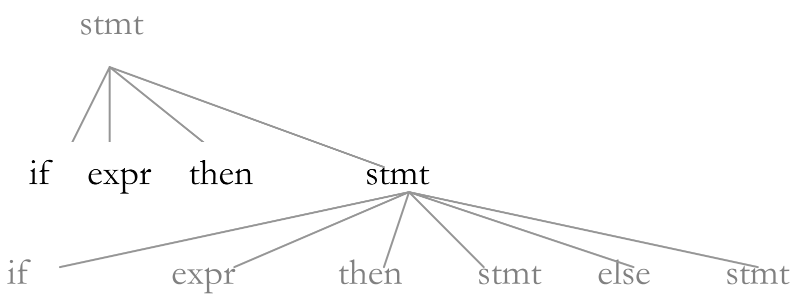
A grammar that produces more than one parse tree for some production is said to be ambiguous. For certain type of parsers this ambiguity needs to removed. The parse used in the assembler and the compiler requires that the ambiguity be removed.As an example the ambiguity from the following grammar would be removed.

Stmt → **if** *expr* **then** *stmt*
| **if** *exor* **then** *stmt* **else** *stmt*
| **if** *exor* **then** *stmt*
| other

Here "**other**" stands for any other statement. According to this grammar, the compound conditional statement is allowed

if *expr* **then** *stmt* **else if** *expr* **then** *stmt* **else** *stmt*

This statement yields two parse tree as shown below.



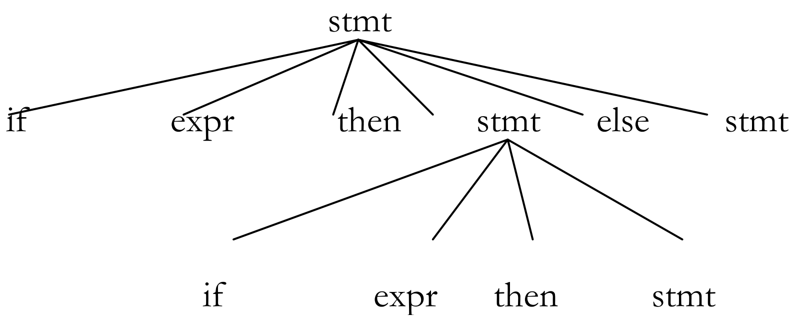


Figure14:- Two parse trees for an ambiguous sentence

This ambiguity can be removed by rearranging the above defined grammar as.

Stmt → matched _ stmt

| unmatched statement

matched_stmt → if expr the matched_ stmt else matched_stmt

| other

unmatched_stmt → if expr then stmt

| if expr then matched_stmt else unmatched_stmt

3.1.2.3. Role of The Parser

Every programming language has rules that prescribe the syntactic structure of programs that are well formed. With the help of the Grammar in certain cases we can very easily and efficiently develop a parser for the subject language. Parser would obtain the strings from the lexical analyzer as shown in diagram below and verifies that the string can be generated by the grammar of the source program. Parser is also responsible for reporting most of the errors checking and recovery.

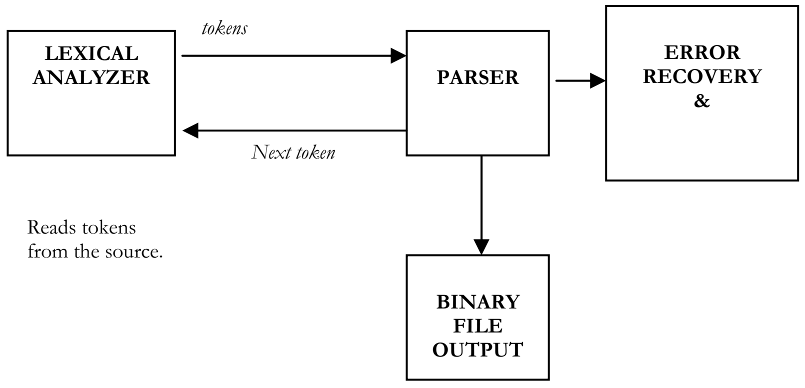


Figure15: BLOCK DIAGRAM OF ASSEMBLER

There are a number of techniques involved as regards the development of parser. The one used in the development of the assembler for the DTMF micro engine will only be described.

Predictive parser follows the scheme of Recursive – descent parsing, which is a top down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each non terminal of the grammar. (stmt) as given in the grammar of the language above. A procedure match is used to match input token with that as defined in the procedure of that non- terminal. If a match occurs the next token is fetched and matched against the next argument. Thus this smooth process of fetching a token form the lexical analyzer and matching this token with the syntax of the Grammar as defined in the procedure of the respective non terminals; allows the process of syntax validation. In predictive parsing methodology the careful elimination of left recursion from the grammar is of paramount importance. Parser thereby enforces the syntactic rules on the source language.

Besides the error recovery routines performed by the parser the predictive parser also generates the binary file. The method of generating the binary code for the instruction set is defined in detailed as below.

3.1.3. Semantic Analyzer

This phase checks the program for the semantic errors and gathers type information. An important component of semantic analysis is type checking. Here the compiler checks that each operator has operands that are permitted by the source language specification. A compiler may allow two operand to be only integer for multiplication , such checks are ensured by type checking.

3.1.4. Intermediate code Generation

After syntax and semantic analysis some compiler generate an explicit intermediate representation of the source program. In order to make a compiler relocatable, that is to be used for some other micro engine, this practice is used. As it is used in the case of compiler for the DTMF micro engine. The intermediate code generation is explained in the compiler development section of this report.

3.1.5. Code Optimization

The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result. There is a great amount of code optimization that few compiler perform. In case of the DTMF micro engine, there is no code optimization incorporated.

3.1.6. Code Generation

The last phase incorporate the generation of the target code. Memory locations are selected for each of variables, and the code for the subject machine is generated.

3.2 Instruction Set for Assembler

Instructions are directly fetched and there is no decoding involved. The signals fetched from the instruction set in the first cycle are described below. For the format of 32 bit instruction set please refer to Annex A. This instruction set was the base for the development of the assembler. It provided the source language and the target machine code. The Grammar for the assembler was based on this instruction set.[IDTMF]

NOP INSTRUCTION

3.2.1. NOP means no operation. In case when no operation is required NOP instruction is given. For NOP instruction the 32 bits of the instruction are divided into the following fields as show in Annex A.

LOAD INSTRUCTIONS

3.2.2. Load Instructions are used for loading the operand in the Data Register File from the Data Memory.

ri = HighWord (*arj[++/--]). In this instruction "ri" (where I = o to 15) represents the location of the register of the DRF in which the data is to be stored. "arj" (where j = 0 to 7) represents the location of the address register of ARF which stores the address of memory from where the data word is to be fetched. The option inside square brackets ++/-- are used when an incremented or decremented address is to be stored back in the ARF. The LSB of the address from ARF shows whether to take upper 32 bits of the 64 bit data from memory or lower 32 bits. "HighWord" means that out of these 32 bits upper 16 are to be stored in the DRF.

ri = **LowWord (*arj [++/--]).** This instruction is the same as the previous one the only difference being that here the command LowWord is used which means take the lower 16 bit of the 32 bits memory data. The division of 32 bit instruction is the same as the previous instruction, only the input select bits are 000 (Annex a)

{ri,r$_{i+1}$ } = * arrj [++/--]. This instruction is used to load a 32 bit memory data in two consecutive register of DRF. Here i= 0,2,4 to 14 and j= 0 to 7. The LSB of the address from the ARF verifies whether to take the upper 32 bits or lower 32 bits.

ri=OxYYYY. This instruction is for storing a 16 bit immediate data to the DRF. Here I = 0 to 15 and YYYY is a 16 bit hexagonal number. The 32 bit instruction is divided into the field shown as per Annex A.

ri=ri+1 = OxYYYY. This instruction is used to store the 16 bit immediate data to two consecutive registers of DRF. Here i=0,2,4,6 to 14 and YYYY is a 16 bit hexagonal number.

STORE INSTRUCTIONS

3.2.3. This instruction is meant to store 32 bit data from the specified address of DRF into a specified address of data memory.

***arj [++/--] = {rm,rn}.** Here "arj" (j= 0 to 7) represents the location of the address register which stores the address of the memory location where the data is to be stored from the DRF. The LSB of the address shows where to store the 32 bit word, i.e. whether to store in the upper 32 bits of the 64 bit memory word or the lower 32 bit "rm","rn" (m= 0 to 15 and n=0 to 15) represent the location of the two particular 16 bit registers from where the data is to be stored. As in the earlier cases ++/-- is used in the case when an incremented or decremented address is to be used and loaded in the address register. Refer to Annex A.

ADDRER/SUBTRACTOR INSTRUCTIONS

3.2.4. These include the instructions for the addition and subtraction of 32 bit or 16 bits operands and also the shifting commands of the result.

{ri,ri+1} = ± {r1,r0} ± {r3,r2} || >>/<< k. This instruction is used for addition/subtraction of two unsigned 32 bit words taken directly from DRF and storing the result in to consecutive registers of DRF. The shifting option is also given {ri, ri+1} (where i=0, 2, 4 to 14) represents the locations of the two registers where 32 bit result is to be stored. The first operand is taken from the first register r0 and the second register r1 and the second operand is taken from r3 and r2. The ± sign before two operands decide which of the following operations are to be performed.

- **A+B**
- **A-B**
- **-A+B**
- **-A-B**

The sign “<<” and “>>” represent shift right and shift left respectively. These options are used only when shifting of the result is required. K represents the shift value. Since the shifter can only shift up to 2 bits to left and 1 bit to the right , k can be 0,1,2, for << and 0, 1 for >>. The 32 bit instruction is divided into the fields as per Annex A.

ri = ± rm ± rn (c) ||>>/<< k. This instruction is used for 16 bit addition / subtraction operation. “ri” (where I = 0 to 15) represents the location of the register in the DRF where the 16 bit result is to be stored. The first operand is taken from the mth register rm (m = 0 to 15) and the second operand is taken from the nth register rn (n =) to 15). The option C is for specification whether to save the carry out of the add/sub operation or not. Other options are same as previous instruction. Refer to Annex A for sub division of the 32 bits.

ri= ±rm ± Y || >>/ << k. Used to perform addition/subtraction of a 6 bit immediate value with an operand fetched from the DRF. “ri” (where I = 0 to 15) represents the destination of the 16 bit result and “rm” (where m=0 to 15) represents the location of the register in DRF from where the operand is to be taken. “Y” is a 6 bit immediate operand. Other options are same as in the earlier instruction

ri = ± rm || >>/<< K. This instruction is used for negation and shift operations on a single operand. “ri” (where i = 0 to 15) and “rm” (m= 0 to 15) are the destination and the source registers of the DRF respectively. K can be 0, 1, 2 for << and 0, 1 for >>. For this instruction, the 32 bit instruction is divided into the fields as given in Annex A.

ri= WB ± rm || >>/<< k. This instruction is used for the addition/subtraction or MAC from the previous cycle with an operand from the DRF. WB is the write back signal which is the result of the previous Add/sub or MAC operation, written back as an operand. Refer to annex A for the sub division of the 32 bit instruction.

MAC Instructions

3.2.5. Instructions for the MAC Unit are as follows:

acc= rm[s/u] * Y [s/u] || >>/<< k. this instruction is for multiplying a 16 bit signed/unsigned operand from DRF with 6-bit signed/unsigned immediate value. “s/u” stands for signed/unsigned. “ri” and “rm” are the destination and source registers respectively. Y is 6 bit immediate operand. Where k can be 0,1,2 for << and 0,1 for >>. Refer to the Annex A attached for the break down of the 32 bit instruction.

acc + = rm [s/u] * Y || >>/<< k. This instruction is similar to the earlier instruction only in this case the value in the registers of the DRF which are reserved for accumulation input {r14, r15} will be added to the result of the above multiplication. Other options are the same. Refer to the Annex A attached for the format of the 32 bit instruction.

acc = rm [s/u] * rn [s/u] || >> /<<.This instruction is for multiplication of two signed unsigned operands from the DRF.”ri” is the destination register and “rm” and “rn” are the source registers of the DRF for the two operands. No accumulation is requires. Other options are the same as in the earlier instruction.

acc += rm [s/u] * rn [s/u] || >>/<< k. This instruction is similar to the earlier instruction only in this case the value in {r14, r15} will be added to the result of the above multiplication. Other options are the same. The 32-bit instruction is divided into the fields as per the Annex A.

|| Reset Accumulator. This instruction can be used with any instruction in parallel. It clears the value in the accumulator register DRF {r14, r15}. The 32 bit instruction is divided into the fields as per Annex A.

AGU Instruction

3.2.6. These instructions are used for saving an address to a specified register of ARF from a specified source register DRF.

arj= ri. Here "arj" (where j=0 to 7) represents the destination register in the ARF and "ri" (i) represents the location of the register in the DRF from where the address is to be taken. Refer to Annex A for the distribution of the 32 bits.

ri = rm & rn. This instruction is to be used for bit wise AND operation on two operands. Here "ri" is the destination register of the DRF and "rm" and "rn" are the source register of the DRF for the two operands. The 32-bits of the instruction are divided into the fields as per Annex A.

ri = rm | rn. This instruction is used for bitwise XOR operation on two operands. Refer to Annex A for bits division.

ri=rm ^ rn. This instruction is used for bitwise XOR operation on two operands. The division of 32 bits of the instructions as shown in the Annex

A, attached to this document.

ri=rm & Y. This instruction is used for bitwise AND operation between an operand from the DRF and an immediate 6-bit value which is given in the instruction. Y is 6 bit immediate operand. Following is the division of the 32 bit instruction set.

ri= rm |Y. This instruction is used for bitwise OR operation between an operand from DRF and an immediate 6-bit value which is given in the instruction. Refer to Annex A for 32-bit division of the instruction.

ri= rm^ y. This instruction is used for the bitwise XOR operation between an operand from the DRF and an immediate 6-bit value which is given in the instruction. 32-bit division is given in the Annex A to this document.

Instructions for Shifter Unit

3.2.7. The include the instruction to 32-bit right shifter and the Barrel shifter as described below. Please refer to Annex A for the division of the 32-bits for each instruction given below.

{ri,ri+1}= {rm, rn} >>1. This instruction is for the 32-bit right shift. {ri,ri+1} (I = 0, 2,4 to 14) show the destination registers of the 32-bit output of the shifter. "rm" and "rn" are the source registers of the DRF for the two operands. These two operands are combined as a single 32-bit word and the right shifted by 1 bit. This means the LSB of the first operand becomes the sign bit of the second operand. First operand is sign extended.

ri=rm.L/A <</>> k. This is the instruction of the Barrel shifter. There is only one 16-bit shift operand which is to be shifted . "ri represents the destination of the 16-bit result and "rm" represents the location of the register in the DRF where the shift operand is to be taken. L/A shows whether logic shift is to be required or the arithmetic shift. << is for left shift and >> is for right shift. K is 5-bit shift value in 2's complement which shows how much shift is required.

Controller Instructions

3.2.8. These instructions include the jump and the conditional jump instructions including the loop instructions. Refer to the following paragraphs for the details of each instruction. Refer to the Annex A for the 32-bit division of these instructions.

goto OxYYYY. This instruction is for an unconditional jump instruction where OxYYYY is an 11-bit hexadecimal jump address of program memory.

If (rm "relational operator" rn) goto OxYYYY | Label. A conditional jump instruction which means that jump if operand 1 which is taken from register rm of the DRF is as per relational operator than operand2 taken from the register rn of DRF. OxYYYY is a 11-bit hexadecimal jump address of program memory or label. Refer to the following table for the combination of the relational operators.

rm < rn	**Operand 1 is less than Operand 2**
rm > rn	
rm > = rn	**Operand 1 is greater than or equal to Operand 2**
rm <= rn	**Operand 1 is less than or equal to Operand 2**
rm = = rn	**Operand 1 is equal to Operand 2**
rm !=rn	**Operand 1 is not equal to Operand 2**

Table3 : Allowed Operand Relationships

LOOOPCOUNT = number || LASTADDRESS = number. This is the loop initialization instruction containing loop count and the loop end address. Here OxYYY is the 7-bit hexadecimal loop count and 0xZZZZ is the 11-bit End Address. The 32-bit instruction is divided as per Annex A.

||REPEAT. This instruction is used with the instruction inside the loop. This instruction can be used in parallel with any other instruction. The 32 bit instruction is divided as per Annex A.

3.3 Assembler Development

The instructions set explained above provided a platform for the development of the Assembler. In order to carry out the analysis of the source program, which in this case was to be based on the instruction set defined above a Grammar was to be defined for the Lexical analyzer. Lexical Analyzer is basically the front hand technique used to identify the tokens, its main task could be stated as " **to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis".**

In order to have the assembler of the DTMF micro engine to be autonomous, as the requirement necessitated the assembler development followed a different suite. Assembler has been developed in a fashion that it operates on it's own as well.

- Lexical Analyzer (with error detection)
- Parser (with error detection)
- Code Generation embedded in Predictive Parser (with error detection)

3.3.1. Lexical Analyzer for Assembler

Lexical Analyzer's role primarily rotates along the question of identifying the tokens as depicted or as allowed by the Grammar of the language. Lexical Analyzer being the first phases of the design for assembler as well as the compiler primarily depends upon the Grammar of the language as defined. In the case of the assembler for the DTMF micro engine this is the instruction set as per Annex A.

Since the lexical analyzer reads the source text it also clears the white spaces or the comments (it ignores them). It also keeps tracks of the line numbers and those errors that are responsible to be reported by the lexical analyzer are reported with the corresponding line number. Upon receiving a "get token" command from the parser the lexical analyzer reads the input characters till it can identify the sent token.

In the statement :

If (rm < rn) goto label1

The lexical analyzer would analyze this statement and would in turn make different tokens for the parser as given below.

Keyword **If**
Register **rm.**
Register **rn.**
Keyword **goto.**
Label **label1.**

3.3.2. Grammar for the Assembler

Grammar for the Assembler is defined below with the help of Regular Expressions.

```
stmts →                     stmt stmts | €

stmt   →                          nop;

     |     ri                     = HighWord (* arj [++/- -])

     |     ri                     = LowWord (* arj  [++/- -])

     |     {ri , ri+1}            =*arj [++/- -]

     |     ri                     =OxYYYY

     |     ri                     = ri+1  =OxYYYY

     |     *arj[++/- -]           ={rm,rn}

     |     {ri , ri + 1}          = ± {r1, r0} ± {r3 ,r3} | | >> /<< k

     |     ri                     = ± rm ± rn [c] | | >> /<< k

     |     ri                     = ± rm ± Y | | >>?<< k

     |     ri                     =± rm | | >>/<< k

     |     ri                     =WB ± rm | | >>/ << k

     |     acc +                  = rm [s/u] * Y | | >>/<< k

     |     acc                    = rm [s/u] * rn [s/u] | | >> /<<

     |     acc                  += rm [s/u] * rn [s/u] | | >>/<< k

     |     arj                    = ri

     |     ri                     = rm & rn

     |     ri                      = rm | rn

     |     ri                     =rm ^ rn

     |     ri                     =rm & Y

     |     ri                     = rm |Y
```

```
        |       ri                      = rm^ y
        |       {ri,ri+1}               ={rm,rn} >>1
        |       ri                      =rm.L/A <</>> k
        |       goto OxYYYY
        |if (rm   "relational operator"  rn) goto OxYYYY | Label.
        |LOOOPCOUNT = number || LASTADDRESS = number
        |        ||REPEAT
relational operator →           <
                        |       >
                        |       > =
                        |       < =
                        |       = =
                        |       ! =
number             →  1 |2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
```

3.3.3. Recognition of Token

The problem of how to specify the language has been explained above. In order to recognize the tokens we use the technique of transition diagrams. This is so done by moving from position to position in the diagram as characters are read.

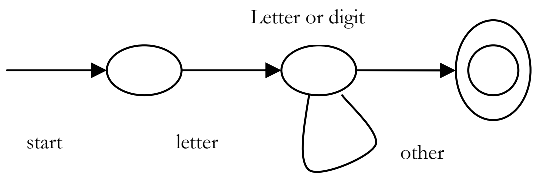
Positions in the transition diagram are drawn as circles and are called states. The states are connected by arrows, called edges. Edges leaving state have labels indicating the input characters that can appear next. The first state is labeled as the start state. This could be best explained by the previously given example of identifiers.

letter → A|B|C|D|E|F……..|Z|a|b|c|d|e|f|……|z

digit → 0|1|2|3|4|5|6|7|8|9

id → letter (letter | digit)*

The transition diagram for the above Regular definition for identifier would be :



The diagram above simply depicts that as a letter is encountered the transition to the next phase takes place and that the first alphabet is valid and satisfies the grammar. As more or any combination of letter and words are encountered they would all be valid and would be accepted as per the Grammar defined. In case a non alphabet or numeric entry is encountered and shall not be accepted and the transition to the final accepting state take place there by depicting that a identifier has been dully recognized. This technique of advancing to the next transition or that of pivoting around a particular state can be easily implemented with the help of **switch** statement.[CMP]

Algorithm for implementation of the above described technique in C would be as given below.

```
Switch (state)
Case 0 : c= next char();
if c *is equal to* **blank, new line, tab** state is equal to 0;
else if c *is equal to* **alphanumeric** state is equal to 1;
Case 1: c= next char();
if c *is equal to* **alphanumeric**  state is equal to 1
else state = 2
Case 2: retract c;
accept  the string before reaching state 2 as a valid string
```

3.3.4. Lexical Interface

Lexical Analyzer is inserted between the parser and the input streams , this input stream may be in the form of input from the keyboard or a text file. It reads characters and recognizes them according to the transition diagram, implemented (most usual case using switch) as shown above. As the characters are read theses characters along with there value and or attribute are passed on to the parser.

Error Recovery by Lexical Analyzer. A few error are also identified by the lexical analyzer. These usually include the recognition of unrecognized characters, invalid entries. The scope of these error is extremely negligible but never the less there importance is vital to the correct performance of the software.

Symbol Table. A data structure called a symbol table is generally used to store information about various source language constructs. The information is collected by lexical analyzer and is used for subsequent code generation. Refer to Annex A1 for Symbol table entries of the Assembler. The symbol table routines are concerned primarily with saving and retrieving lexemes. Two operations are of overriding importance to the symbol table

- Insert function. To insert entries into the symbol table. Symbol table would insert automatically identifiers defined in the construct of the program. In case of the assembler these identifier are label. These are inserted into the symbol table as encountered.

- Lookup function. To view the contents of the Symbol table and match them with the a particular string.

Note: Symbol table handles reserved words of the language by initializing itself with the reserved words entries as the program is initialized. Please refer to Annex A1 for the details of reserved words for the Assembler.

Transition diagram for the DTMF lexical analyzer (assembler only) as derived from the regular expressions , is given below.

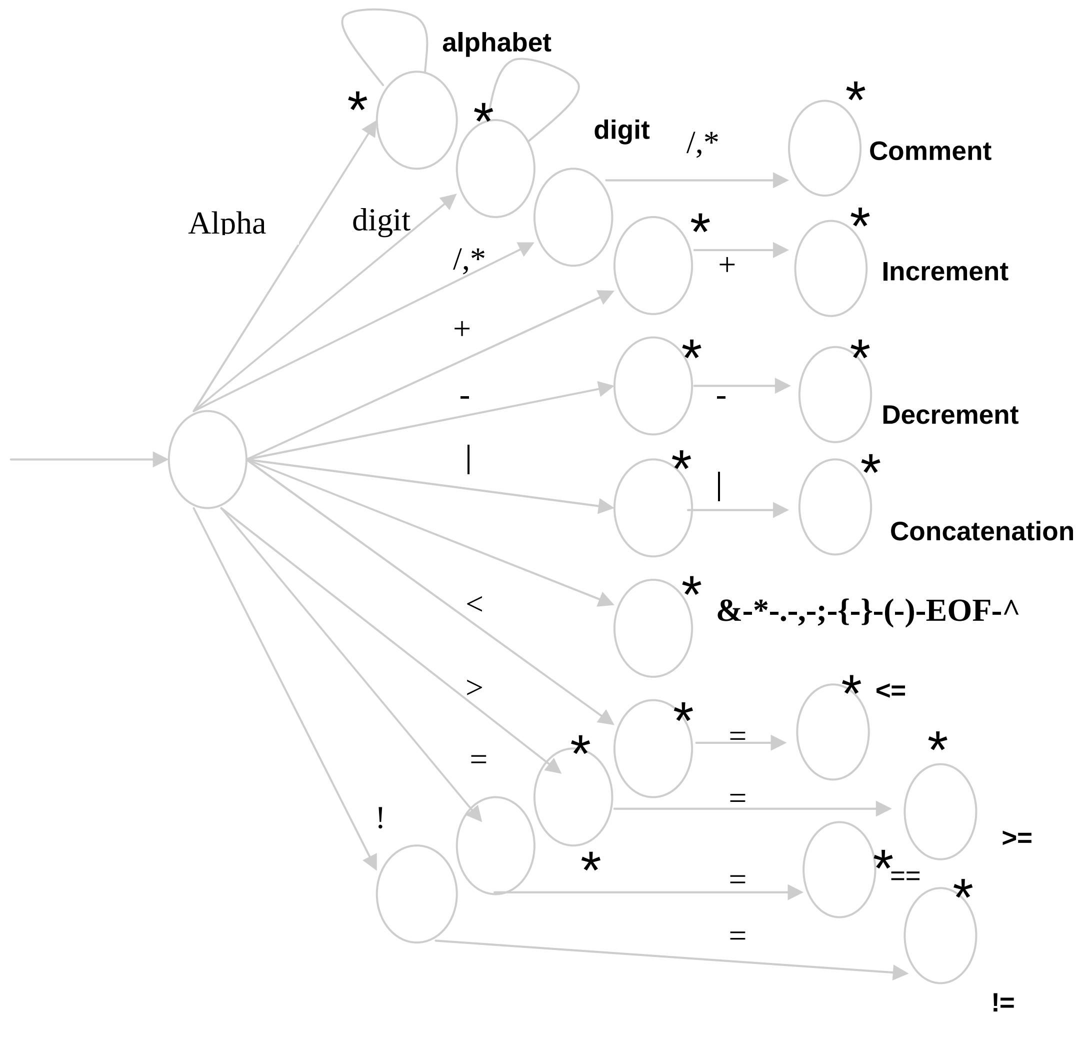


Figure 14 TRANSITION DIAGRAM FOR THE LEXICAL ANALYZER OF THE ASSEMBLER

3.3.5. Parser for Assembler

The parser developed for the assembler is a recursive parser. The ambiguity removed from the grammar and without recursion. The parser is a two pass, parser, in the first go, it reads the label and makes a note of all label entries. In the second parse it generates the binary code. Each non terminal of the Grammar represents a function and the statement of those non terminals are matched against the allowed words, as per the grammar. As the match occurs, that is the word which should occur at the subject place exits , the next token is fetched by the parser from the lexical analyzer and matched against the next allowed word. In case of the Parser for the assembler the code generation was also performed with in the function of each non terminal. This procedure of generating the

Besides the error recovery routines performed by the parser the predictive parser also generates the binary file. The method of generating the binary code for the instruction set is defined in detailed as below.

Code Generation. In order to generate binary code, primarily four functions have been defined according to the instruction format. These function are :-

- GetNormalCode ( ).
- getNormalimm16Code ( ).
- getJumpCode( ).
- getLoopCode ( ).

The reason for defining these function was necessitated primarily because the instruction format for these instruction varies and could be categorized in four sub group (Each function stated above is primarily used for the generation of code, as the instruction format varies). As each instruction is identified the corresponding function as per the format f the instruction is passed the relevant parameters for the generation of the binary number, which is inserted sequentially into an array of unsigned integers.

The total instruction length inclusive of the control and the data field is fixed and is of 32 bits. The instructions categorized as Normal are broken down into the following sub divisions:-

o	**Instruction decoder 1**	**-**	**2 Bits**
o	**Instruction decoder 2**	**-**	**2 Bits**
o	**Memory Access**	**-**	**1 Bit**
o	**Input Select**	**-**	**3 Bits**
o	**Operand 1**	**-**	**4 Bits**
o	**Control**	**-**	**9 Bits**
o	**Immediate**	**-**	**2 Bits**
o	**Operand 2**	**-**	**4 Bits**
o	**Destination**	**-**	**5 Bits**

The instruction set for the jump instruction has sub division identifying the signal for the two decoder, memory access , four bit for the MSB of Branch address followed by operand 1, 8 bit for conditional select , followed by 4 bit for LSB and 4 bits reserved for Operand 2, terminating with 5 unused bits of the Destination sub field:-

o	**Instruction decoder 1**	**-**	**2 Bits**
o	**Instruction decoder 2-**	**-**	**2 Bits**
o	**Memory Access**	**-**	**1 Bit**
o	**Branch address MSB**	**-**	**3 Bits**
o	**Operand 1**	**-**	**4 Bits**
o	**Conditional Select**	**-**	**9 Bits**
o	**Branch address LSB**	**-**	**2 Bits**
o	**Operand 2**	**-**	**4 Bits**
o	**Destination(Unused)**	**-**	**5 Bits**

The requirement for defining the function getNormalimm16 was entailed by a different format of instruction dealing with 16 bit Immediate operand . There are two instruction of this format used to store a 16 bit immediate value to a Data Register. Refer to Annex A serial 4, and 5 for instruction format. The forth function getJumpcode() is evoked for the instructions entailing jump.

o	**Instruction decoder 1**	**-**	**2 Bits**

o	**Instruction decoder 2-**	**-**	**2 Bits**
o	**Input Select**	**-**	**3 Bits**
o	**Immediate**	**-**	**16 Bits**
o	**Destination(Unused)**	**-**	**5 Bits**

3.3.6. Error Recovery.

Error recovery strategies adopted in the development of parser ensure smooth progress of the syntax validation. There are a number of strategies used today for the error recovery. (Non of these are universally accepted but only provide a platform)

1. **Panic mode**

2. **Phrase level**

3. **Error Production**

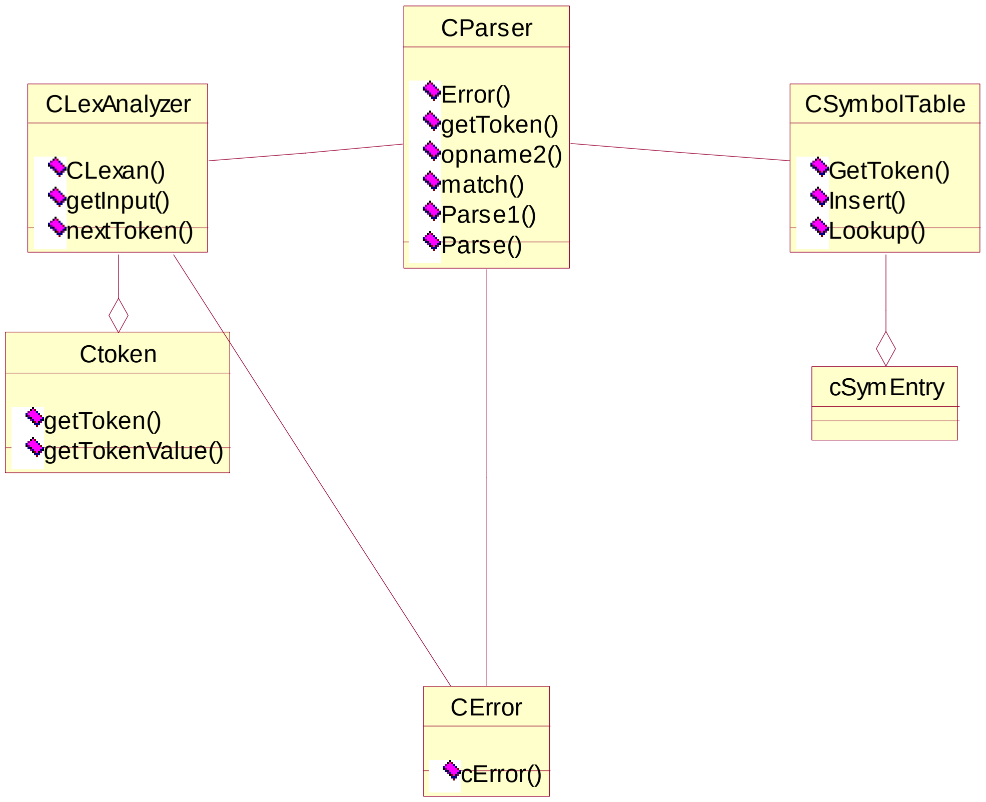
Panic Mode. This is the simplest method to implement and can be used by most parsing methods. On discovering an error the parser discards all the input symbols one at a time until one of the designated synchronization token is found. This synchronization token is usually the delimiter.

Panic Mode recovery technique has been employed in the parser of the assembler. The parser looks forward at few productions also before jumping to the terminator as this technique suggests.

Phrase Level._The parser on discovering an error may be able to perform local correction on the remaining input. A typical correction would be to replace a comma by a semi colon, delete a semi colon and or add one.

Error Production. If the errors to be encountered are well known at the time of the development of the software the grammar for the language at hand can be augmented with the productions that generate erroneous constructs. If an error production is used by the parser, appropriate error diagnostic could be generated to indicate the erroneous construct that has been recognized in the input.

3.4 Class Diagram (UML) Assembler



3.4.1. CLASS NAME:- CLexAnalyzer

Class Operation.

getInput()

NextToken()

CLexan()

Class Description.

This class is primarily responsible for reading the input sequence from the text editor. It **Nextoken()** operation is responsible for reading the tokes in a sequence. Reading each token it forwards the input to the Parser. The operation of getting the input from the text editors environment is performed by the operation getInput().

3.4.2. CLASS NAME:- CParse

Class Operation:- getNormalCode() getNormalimm16Code()

getJumpCode() getLoopCode() error()

Class Description:-

This class is primarily responsible for the syntax analysis of the Assembler. In this case it also generates the target binary code using four functions as discussed above. For the code generation it uses the functions GetNormalCode ( ),getNormalimm16Code ( ),getJumpCode( ),getLoopCode ( ). Error messages are sent to the error classes which are displayed on to the screen.

The Error recover procedure as governed by panic mode error recovery are also handled, by this class.

3.4.3. CLASS NAME:-CSymbolTable

Class Operation:- init() lookup() insert()

Class Description:-

The operation of the symbol table are controlled by this class. The symbol table is first initialized with the keyword. These are stored in a data structure. The function init() initializes the symbol table, with its entries.

3.4.4. CLASS NAME:-CSymEntry

Class Description:-

Stores information regarding the entries in the symbol table. The function get address is primarily used for the replacing the labels with the address of the instruction where that label is used. Moreover it return the parameters as assigned and requires of the tokens from the symbol table. Used to obtain token, lexeme type and address form the symbol table for each specified entry.

3.4.5. CLASS NAME :-CError

Class Description:-

Used to display the error as reported and passed by the Parser and the lexical analyzer.

3.5 Compiler Development

The Compiler developed has been based on a subset of C and it performs the main functions as per the requirement of the hardware for which it has been written. There by a number of aspects as in case of C have not been completed in totality. This compiler supports all the major functions of C. Moreover this is a two front compiler; it has a front end and back end logic. The front end of the compiler generates an intermediate code for the source program and the details of the target program are confined to the back end, as far as possible. The benefits of using a machine-independent intermediate form are :

- Retargeting is facilitated. A compiler for a different machine can be created by attaching a back end for the new machine.
- A machine independent code optimizer could be used.

3.5.1. Lexical Analyzer for the Compiler

The technique adopted fort he development of the lexical analyzer is the same as defined in the assembler section of the report. The difference comes in regards to the keywords allowed by this lexical interface . The following table gives details of allowed key words which are also the words which are initialized in to the symbol table of the compiler.

Reserved Words	Type
if	Keyword
while	Keyword
for	Keyword
switch	Keyword
break	Keyword
case	Keyword
default	Keyword
void	Keyword
main	Keyword
else	Keyword
goto	Keyword
do	Keyword
int	Keyword
Char	Keyword

3.5.2. Grammar of the Compiler

The Grammar for the compiler on which the entire working of the parser is based is given below.[WPU]

Prog →	**void main (void) {** *funcA funcB* **}**
funcA→	*type id arrayDecl func AA*; *funcA*
	| E
type →	*char* | *int* |
arrayDecl →	[*exp*] *multiDimArrayDecl* | E
multiDimArrayDecl →	[*exp*] *multiDimArrayDecl* | E
funcAA →	, id arrayDecl *funcA*A | E
funcB →	*stmt func B* | E
stmts →	**if** (*boolExp*) *stmt stmtifA*
	| **switch** (*exp*) {*caseStmt default stmt*}
	| **while** (*boolexpr*) *stmt*
	| **for** (id=*intcon*; id *Relop* [*intcon*|id] ; id [inc|dec]
	|*label* ;
	| **goto** *label*;
	| *exp assgt*;
	| {*stmt stmt A*}
	|;
stmtifA →	**else** *stmt* | E
caseStmt→	**case** ***exp***: ***stmt*** [break;] **case** *stmt* | E
defaultstmt →	**default** : *stmt* **break** ;
stmtstmtA →	*stmtstmt stmtstmtA* | E

boolexp →	*id* Relop *boolexpr*
	| ! *boolexpr boolexpr1*
	|(*boolexp*) *boolexpr1*
boolexpr1→	*logop boolexp*
boolexp2→	*intcon*| *id*
assgt →	= exp assgt | E
exp →	*term moreterm*
term →	*factor morefactor*
moreterm →	*binopo term moreterm* | E
morefactor →	*binop1 factor morefactor* | E
factor →	*binopo exp*
	|(*exp*)
	| *id id Assoc*
	| *int con*
	| *char con*
	| inc id
	| decide
idAssoc →	[in/dec]
	| arrayelm
	| (parameter list)
	| E
arrayElm →	[*exp*] *more arrayElms*
moreArrayElms→	[*exp*] | E
parameterlist→	*exp moreparameterlist* | E

3.5.3. Type Check

A compiler is suppose to report an error if an operator is applied to an incompatible operand ; for example if an array variable and a int variable are added together. There are two of checking performed . One which is performed by the compiler is termed as the ***static type checking***, whereas the one performed when the target program runs is don ***dynamically***.

The most simplest way of applying type checking is with the help of the symbol table. This is the kind of check incorporated in the Compiler developed. There are two symbol tables for the compiler, one that stores the keywords and identifiers and the on that deals with numeric values. As the parser scans the production for the identifiers, where the type of variable to be declared is given by the user, the symbol table is updates with the type of operand the numeric value has. As the same value or operand is applied or used in any mathematical computation the symbol table scan for the type of the two operands to be checked and generates an error if the two types are incompatible. The overhead of this operation is the additional entry in the symbol table, which keeps track of the type of the numeric or the operand declared, if not initialized.

3.5.4. Code Generation

The basic techniques of the Lexical Analyzer are the same in the case of the compiler as they were followed and described in the Assembler development section. The major difference between the implementation of the compiler and that of the assembler are in the phase of Code Generation, type check, register allocation procedure. Code Generation itself is a two phase process. From the source program first a target three address code is generated, which in term is converted to the target assembly syntax and feed to the assembler. This section defines the three address code and final code generation

3.5.5. Three Address Code

Three address code is a sequence of statements of the general form

Z = a operator b

Where a,b and z are names, constants, or compiler generated temporaries. Operator stands for different arithmetic and logical operator applied on integer, floating point or logical expression as the case may be. In three address code generation there is no build up of arithmetic expression allowed, meaning that a expression like **a+b+c** , would be translated into a sequence

Temp1 = a + b

Temp2= c + Temp1

Here the two names Temp 1 and Temp 2 are actually compiler generated names. A **function** is implemented in the compiler which sequentially generates such variables.

The reason for term " three address code" is that each statement usually contains three addresses, two operands and one for the result.

3.5.5.1. Syntax Directed Definitions. A syntax directed definition is the generalization of a context free grammar in which each grammar symbol has an associated set of attributes. These are addressed as the synthesized and the inherited attributes. The value of the synthesized attribute at the node is computed from the values of the attributes at the children of that node in the parse tree, the value of the inherited attribute is computed from the values of attributes at the siblings and parent of that node. Consider the example below, which is based on the grammar.

E → **T** { *R.i* = *T.val*}

R {*E.val* = *R.s*}

R → **+ T** { *R1.i*=*R.i* + *T.val*} **R1** {*R.s* = *R1.s*}

R → **- T** {*R1.i* = *R.i* – *T.val*} **R1** {*R.s* = *R1.s*}

R → E { *R.s* = *R.i*}

T → (E) { *T.val = E.val*}

T → num { *t.val = num.val*}

Basing on the above translation scheme the parse tree for the expression "**9 – 5+ 2"** which is allowed by the parameters defined above could be easily constructed as

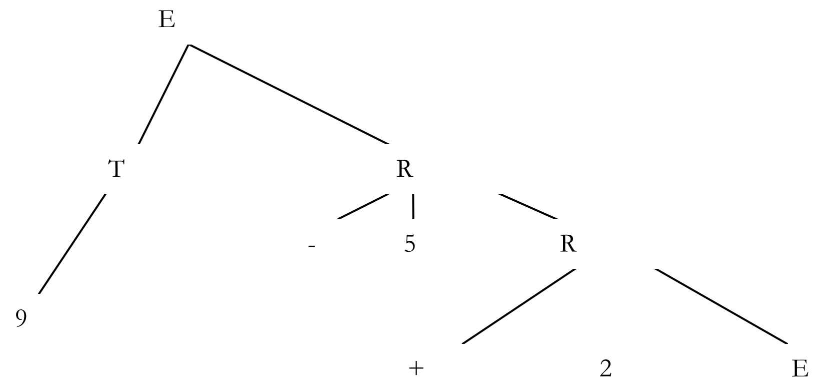


Figure 2 Parse Tree for Expression 9 – 5 +2

This step of creating the parse tree for the expression is quite simple but for computing the value of the expression the help from the semantic rules provide a platform. The synthesized attributes for the above defines grammar are given with the prefix of ".S" and inherited attributes ".I".

Step1:- T.val is assigned 9

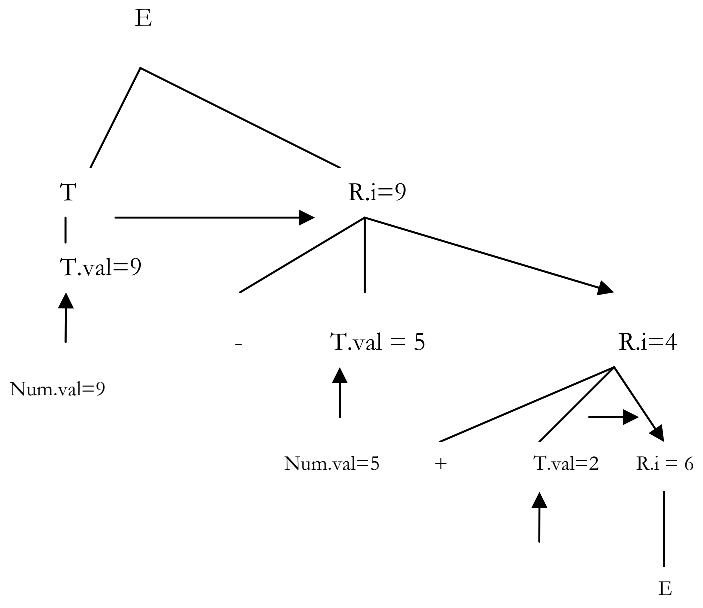
Step 2:-Value of T.val is passed to the inherited attribute of next node

Step 3:-R proceeds to the production

R → - T R T.val is assigned 5 and R1.i that is the inherited attribute of level 1 is computed based on the translation scheme where the value of the inherited attribute of the parent is subtracted from the T.val to yield (9 – 5) 4.

Step 4:-The value computed is passed on as the inherited attribute of level 2 node. The Production next to be processed is R → + T R

First T.val is assigned 2 and the inherited attribute computed as the sum of the synthesized and the inherited attribute 2 + 4=6.This is assigned to the synthesized attribute of second level and as the terminator sign is reached this value is finally assigned to the Evaluation expression. Yielding the desired results.



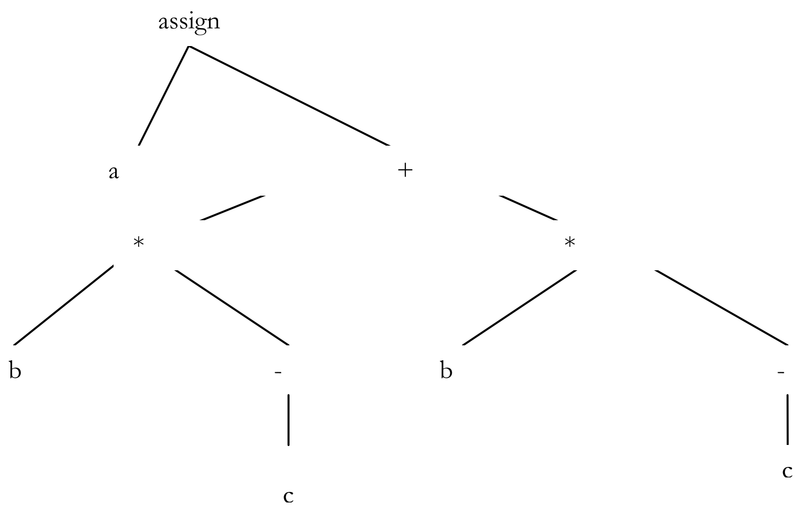
3.5.5.2. Syntax Directed Translation. When three address code is generated temporary names are made up for the interior modes of a syntax tree. This procedure could be best understood with the help of an example. Consider the production as given below

S →	**id = : E**
E →	**E1 + E2**
|	**E1 * E2**
|	**- E1**

| **(E1)**

| **id**

Basing on above Grammar if we make the parse tree for an allowed word such as "**a = b * - c + b * - c",** the procedure for the generation of the three address code will be clear.



Graphical Representation of a = b * - c + b * - c

Using the technique of synthesized and the inherited attribute the generation fort he three address code is followed. The first temporarily generated variable will be assigned the value –c .

Temp 1= - c

Temp 2 = b * Temp1

Temp 3 = - C

Temp 4 = b * temp 3

Temp 5 = Temp 2 + Temp 4

a = Temp 5

Each node is represented by a temporary variable. In terms of implementation of this step a simple function is declared which sequentially generates the temporary variable/names. More over from the above statement it can be seen that there are a number of redundant statement and the optimal generation would involve fewer statement s than the ones generated from the discussed technique. The DAG implementation procedure achieves better result than the ones explained below. The procedure adopted for the generation of code for three address, in the compiler development has been based on Syntax tree and not DAG therefore DAG has not been discussed here. The reason for the implementation of syntax tree method was the time constraint.

3.5.6. Target Code Generation.

The last phase of compiler implementation is the code generation of the target machine. The final phase takes the input form the intermediate representation scheme and generates the equivalent target program. The requirements of the code generator are severe. The output code must be correct and of high quality, meaning that it should use the resources machine. It should preferably be using machine heuristic.

This is also the back end logic, which operates on the generated three address code only. By this stage of the compiler all necessary checks like parsing, syntax evaluation and type checks, type conversions are performed. The output of this generator could be, absolute machine code, assembly code or relocatable machine code. In the case of the compiler for the DTMF micro engine, the target program is the assembly language program as defined in the assembler phase. This micro engine has assembly like constructs but with few difference. As the assembler was developed before the compiler therefore the target code for the compiler was chosen to be assembly (of the DTMF micro engine), so that it could be feed to the assembler which in turn will generate the machine code for the simulator.

3.5.6.1. Instruction Selection. The nature of the instruction set of the target machine determines the difficulty of the instruction section. For each type of three address statement a code skeleton can be designed that outlines the target code to be generated. A very simple type of instruction a=b+c, cn be translated into the assembly code as:-

```
MOV b,RO
ADD z,RO
MOV RO,x
```

In the above example it is assumed that MOV is a recognized assembly construct and the first operand represents the Source where as the second operand represents the destination.

The quality of the code generated is determined by the speed and the size. A target machine with a rich instruction set may provide several ways of implementing a given operation. Since the cost difference between different implementations could be significant, a native translation of the intermediate code may lead to correct but inefficient code.

3.5.6.2. Register Allocation. Instructions involving the register operands are usually shorter and faster than those involving operand in the memory. Therefore, efficient utilization of the registers is particularly important in code generation. Thus the use of registers is divided into the following sub categories.

- **Register Allocation.** The set of variables that will reside in the registers at a point in the program.
- **Register Assignment.** Selection of specific register that a variable will reside in.

For the purpose of tracking the values contained in the register, a symbol table data structure was implemented. This table keeps track of the register stored in the Data Register File and also there value. If the same value are to be used these are looked up in the symbol table and returned. This could be termed as the code descriptor.

The implementation scheme or approach adopted for this problem could be best explained with the help of the following example. Consider the assignment statement d=(a-b) + (a-c) + (a-c). The three address code generated could be.

```
t = a –b
u = a-c
v = t +u
d= v +u
```

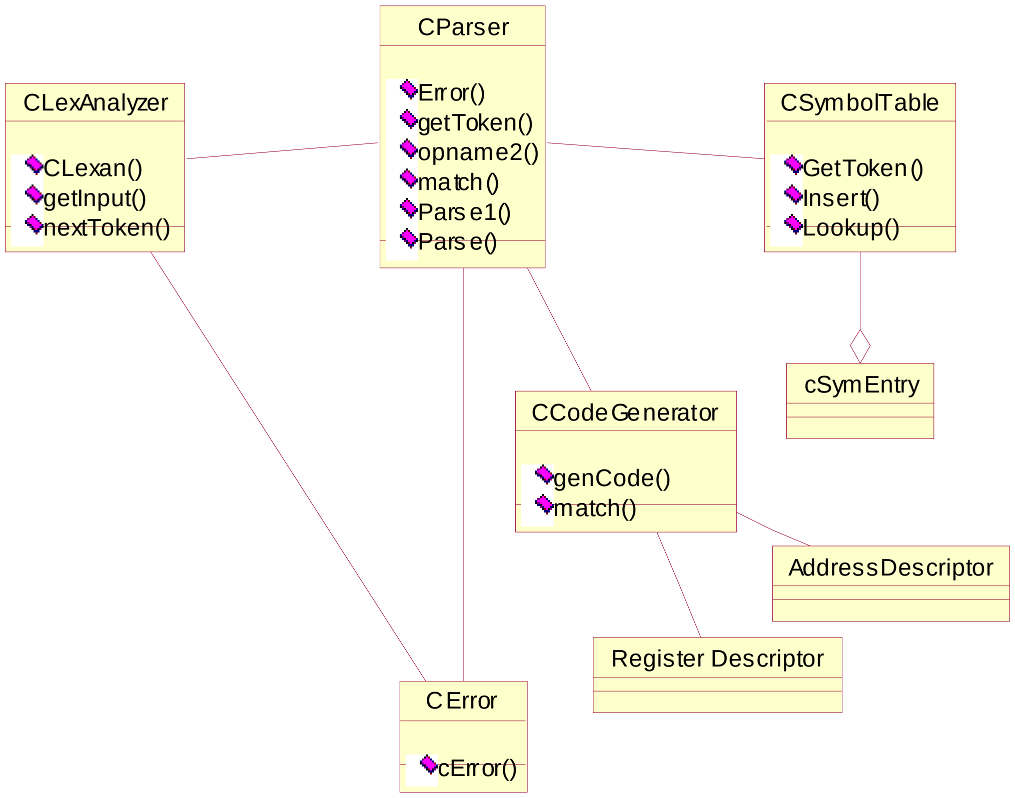
For the first instruction the Code generated would be

```
Mov a, R0
Sub b, R0
```

Initially the register file is empty. After the generation of this code the register the register descriptor is updates to indicate that the computed sum is contained in the register R0. Code Generation Proceeds in the usual manner as depicted in the table below.

STATEMENTS	CODE GENERATION	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
t= a –b	Mov a,RO	R0 contains t	t in R0
	Sub b,R0		
u= a –c	Mov a,R1		t in R0
	Sub c,R1		u in R1
v= t + u	Add R1,Ro		u in R1
d= v +u	Add R1,R0	R0 contains d	
	Mov R0,d		d in R0
			And Memory

3.5.7. CLASS DIAGRAM (UML) COMPILER



3.5.7.1. CLASS NAME:- Address Descriptor

Information regarding every variable, including the data register allocated, address register allocated, memory location in the data memory, and the lexeme of the identifier, is stored in the Address Descriptor. The functionality of Address Descriptor is managed by this Class.

As discussed above in order to keep track of the allocation of the registers to different variables generated, during the process of three address code generation, an address descriptor is mandatory. This adopted procedure would usually not be found in case of the Intel 86 family as there are no separate address registers.

3.5.7.2. CLASS NAME:- Register Descriptor

Information regarding the variable, it's lexeme and the data register allocated. This class is primarily used for the implementation of the register allocation procedure. As a address descriptor is required, a register descriptor is used to track the register allocation of the 16 registers available in the DTMF micro engine.

3.5.7.3. CLASS NAME:- CodeGenerator

The three address code generated from the parser is passed to the Code generator. This class is responsible for generating the assembly code which in turn is fed to the assembler. The Code Generator has its own parser which is based on the three address code.

3.5.7.4. CLASS NAME:-CParser

Cparser is responsible for the basic functions of parser, where it gets token form the Lexical analyzer in this case Class CLexAnalyzer and processes each received token through allowed grammar of the subject language. The Parser is based on the Grammar of C language, in this case.

3.5.7.5. CLASS NAME:-CLexAnalyzer

Lexical Analyzer is responsible for tokenization of the input stream and passing these tokens to the Parser. Lexical Analyzer has been implemented more or less in the same fashion as in case of Assembler

3.5.7.6. CLASS NAME:-CError

An Error recovery procedure is of extreme importance, as it not just reports the error but it also allows the parser to recover after encountering errors. The Panic mode error recovery scheme has been implemented by this class with few modifications.

3.5.7.7. CLASS NAME :-CSymbolTable

Symbol table stores and updates itself at run time. All the reserved words are stored in the symbol table and all identifiers variables and label during the run time are updated. The two important operations controlled by this class are of looking up the symbol table and of inserting a label, identifier or variable. CSybolTable.

3.6 FUTURE RECOMMENDATIONS

The work on the Compiler can be carry forwarded, and a complete compiler for C could be developed. The presently developed compiler has many limitations. The reasons for which were the design specification and meeting the minimum criteria which satisfies the working of the DTMF Micro Engine.

Considering the design parameters and the given requirements, option of including floating point operations, division by odd numbers, structures, pointers and function calls were not catered for in the compiler. In order to take this project further ,compiler may be embedded with all above facilities and may also be given the extra advantage of including C header files.

The Future work on the assembler and the simulator can not be taken up as the requirement for these tools was specific to a problem. No faults or future recommendations have been forwarded by the design team, as regards these two modules of the project.

Chapter

4

4. Software Manual

The Developer Studio for the DTMF detector is an integrated developed environment, offering extremely friendly user interface. It is quite similar to Standard windows application. To run the simulator, first the binary file needs to be prepared which would be loaded into the program memory of the simulator. The generation of the file could be done either by the compiler, or the assembler. The end of the chapter has few test programs to assist the user.

4.1 Menus

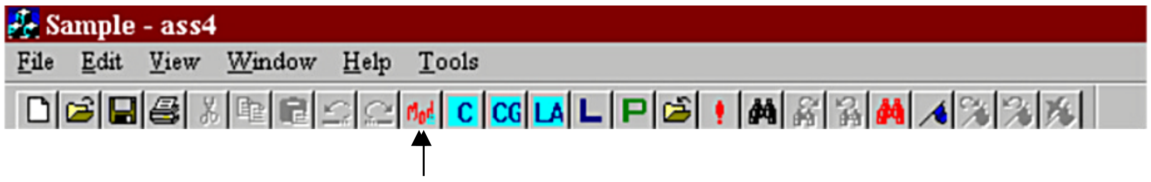
The main frame of the software has the following Menus

File ►	New	► Opens new window for Text Editor
	Open	► Open a saved file for the Text Editor
	Save	► Save the text editor window as a .cpp extension
	Save As	► Save as different name
	Print	► Print the Text Editor window
	Exit	►Exit the software

Edit ▶	Undo	▶ Undo the last step
	Redo	▶ Redo Undo
	Cut	▶ Cut the selection to the Clipboard
	Paste	▶ Paste the selection from the clipboard at the specified place
	Delete	▶ Delete the selection
	Select All	▶ Select the entire editor entries
	Find	▶ Find dialog
	Find Next	▶ Find the next entry as per the Find dialog
	Find Previous	▶Find Previous entry
	Replace	▶Replace the entries against the specified word
	Read Only	▶ Make the document Read only
	Bookmarks	▶ Bookmark the place
	Goto Book marks	▶ Go to the specified bookmark
	More Bookmarks	▶ Toggle Bookmarks
View ▶		
	Toolbar	▶ Enable/disable Visibility of Toolbar
	Status Bar	▶ Enable/disable Visibility of Status Bar
Window▶		
	New Window	▶ Disabled
	Cascade	▶ Display multiple cascaded window
	Tile	▶ Display multiple window tiled
	Arrange Icons	▶ Disabled
Tools▶		
	Lex Analyzer	▶ Lexical Analyzer Mode
	Parser	▶ Enable the Parser

4.2 Compiler mode

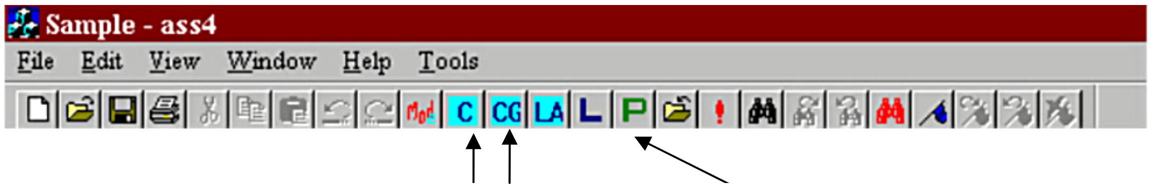
The mode for the compiler and assembler can be changed through the button marked on the toolbar.



Compiler / Assembler mode

After setting the mode to Compiler a Message will appear in the Message window. The user is supposed to enter a C program in the text editor or open a already exiting C program. An attempt to open and parse the assembly file will result in errors. The program must not include any header file and should only start with the statement **void main (void),** followed by parenthesis **{** and terminating parenthesis **}.**

The program needs to Parsed through so that three address code could be generated. Press Marked button for parsing. If there are no errors in the program a Message window display "Parsing OK". And displays the three address code in the Message window.



Three Address Code Generator Code Generator Generates the binary file for simulator

Press the CG button on the tool bar, this generated the assembly code for the assembler. Press the button marked P for the generation of the binary file.. This file is also saved with the extension of asm in the default folder.

4.3 Assembler mode

The mode to the software can be switched from assembler to compiler or vice versa. In order to make the software run in assembler mode, press the Assembler/Compiler switch mode switch. The user at this stage is suppose to enter the program in the assembly language of the DTMF micro engine. Refer to the Annex A attached for the brief on the instruction set in assembly or to section 3.2.

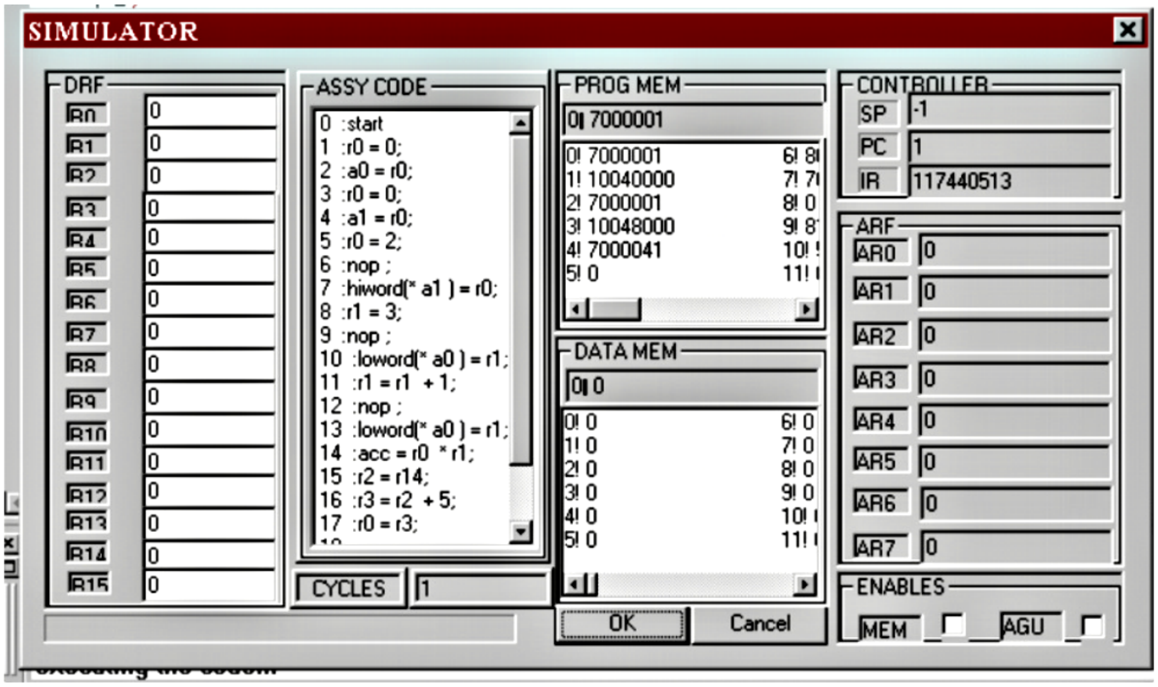
4.4 Simulator

The simulator is a self descriptive user interface. After the binary file has been generated. It is to be loaded into the program memory of the simulator. This is done by pressing the loader button on the toolbar. As the file is loaded the simulator is ready to process this input. Pressing the button for the simulator which is marked as"!" enable the simulator. The Register File of 16 registers, shows the values of the register and the so does the address register file. The Program memory and the data memory also display it's entries. As the program is processed these values are subsequently changed and updated. The clock cycles are displayed in window as the instructions progress through the simulator.

The simulator has two modes to Operate

- o Debugger mode
- o Simulation mode

In the first case the simulator stops after each clock cycle, whereas in the other case it progresses through to the end of the program with updated vales of the Register file, from where the result could be seen.


SIMULATOR

DRF

R0	0
R1	0
R2	0
R3	0
R4	0
R5	0
R6	0
R7	0
R8	0
R9	0
R10	0
R11	0
R12	0
R13	0
R14	0
R15	0

ASSY CODE

```
0  :start
1  :r0 = 0;
2  :a0 = r0;
3  :r0 = 0;
4  :a1 = r0;
5  :r0 = 2;
6  :nop ;
7  :hiword(* a1 ) = r0;
8  :r1 = 3;
9  :nop ;
10 :loword(* a0 ) = r1;
11 :r1 = r1  + 1;
12 :nop ;
13 :loword(* a0 ) = r1;
14 :acc = r0  * r1;
15 :r2 = r14;
16 :r3 = r2  + 5;
17 :r0 = r3;
```

CYCLES 1

PROG MEM

```
0| 7000001

0! 7000001          6! 8
1! 10040000         7! 7
2! 7000001          8! 0
3! 10048000         9! 8
4! 7000041          10! 
5! 0                11! 
```

DATA MEM

```
0| 0

0! 0                6! 0
1! 0                7! 0
2! 0                8! 0
3! 0                9! 0
4! 0                10! 
5! 0                11! 
```

CONTROLLER

SP	-1
PC	1
IR	117440513

ARF

AR0	0
AR1	0
AR2	0
AR3	0
AR4	0
AR5	0
AR6	0
AR7	0

ENABLES

MEM ☐ AGU ☐

OK Cancel

4.5 Test Programs

The following test program of simple mathematical operation was given

```
void main (void)
{
        int b;
        int c;
        c = 2;
        b = 3;
        b ++;
        c = c * b +5;
}
```

The manual result of this program yields C=13. Executing this program on the simulator yields the result 13 in the r1 register. This is the register which stores the value of the C.

REFERENCES

[ACH]	*Computer Architecture and Organization by William Stalling*
[PDP]	*Parallel and Distributed computing by Jeffry D Ulman*
[DTMF]	*DUAL TOME MULTIPLE FREQUENCY DETECTOR BY Enabling Technologies, under the supervision of Dr Noman*
[IDTMF]	*INSTRUCTION SET FOR THE DUAL TONE MULTIPLE FREQUENCY DETECTOR, by Dr Noman*
[CMP]	*Compilers Alfred V.Aho*
[WPU]	*Princeton University WebSite*