CONTROL FLOW CHECKING USING SOFTWARE SIGNATURE (CFCSS)

A Dissertation Submitted in partial fulfillment of
the requirements for the award of the degree of

MASTER OF ENGINEERING
(Computer Technology & Applications)

By

Ankur Gupta
College Roll No. 01/CTA/04
Delhi University Roll No. 8503

Under the guidance of
Dr. Goldie Gabrani

Department Of Computer Engineering
Delhi College of Engineering
Bawana Road, Delhi-110042
(University of Delhi)

June 2006

CERTIFICATE

This is to certify that the dissertation entitled "**Control Flow Checking Using Software Signature (CFCSS)**" submitted by **Ankur Gupta** in the partial fulfillment of the requirement for the award of degree of **Master of Engineering** in Computer Technology and Application, Delhi College of Engineering is an account of his work carried out by him.

June, 2006

Dr. Goldie Gabrani
(Project Guide)
Head of the Department
Department of Computer Engineering
Delhi College of Engineering, Delhi

ACKNOWLEDGEMENT

It is a great pleasure to have the opportunity to extent my heartiest felt gratitude to everybody who helped me throughout the course of this project.

I would like to express my heartiest felt regards to **Dr. Goldie Gabrani,** Head of the Department, Department of Computer Engineering for the constant motivation and support during the duration of this project. It is my privilege and owner to have worked under his supervision. His invaluable guidance and helpful discussions in every stage of this project really helped me in materializing this project. It is indeed difficult to put his contribution in few words.

I would also like to take this opportunity to present my sincere regards to my teachers viz. Professor D. Roy Choudhury, Mr. Rajeev Kumar, Dr. S. K. Saxena and Mrs. Rajni Jindal for their support and encouragement.

I am thankful to my friends and classmates for their unconditional support and motivation during this project.

Ankur Gupta
M.E. (Computer Technology & Applications)
College Roll No. 01/CTA/04
Delhi University Roll No. 8503

CONTENTS

LIST OF FIGURES

ABSTRACT

CFCSS is a pure software method that checks the control flow of a program using assigned signatures. While special hardware for error checking is required in other signature monitoring techniques, CFCSS does not need the help of extra hardware for error detection; this is the advantage of CFCSS.

The distinctive feature of the CFCSS over previous signature monitoring techniques is that CFCSS needs no dedicated hardware such as a watchdog processor for control flow checking because it is a pure software method. A watchdog task in multitasking environment also needs no extra hardware, but the advantage of the CFCSS over it is that CFCSS can be used even when the operating system does not support multitasking.

This dissertation is presented to implement the CFCSS algorithm for the 586 architecture, for any fan in value.

The runtime provides a visual basic interface as a text editor in which a person can write the 586 code. After he finishes writing the code he can run it normally using MASM or he can run using the CFCSS option, in which the code is passed through the CFCSS algorithm where the signatures are appended to the code accordingly. And when the program is run the errors if any are logged.

The length of the code increases due to the presence of signatures on each node but it accounts for the cost saved in having hardware to solve the purpose.

This dissertation is written in as friendly and explanatory manner as was possible to make even the people with little or no background understand the project.

1. INTRODUCTION

1.1 Problem Statement

Control flow errors are long existent but are very rare, and are generally handled with the help of a hardware known as the watch dog processors. Control flow errors have to be checked in an environment where we need high accuracy and any error results in fatal errors like in the case of supercomputers where working needs highly efficient and accurate systems.

Control flow errors are errors that exist when the program does not behave as It was intended to do mainly due to the presence of false jumps that is for example a program may reach a wrong position due to a voltage fluctuation or some other malfunction. In this case the hardware program terminates abnormally, or may give false results. To check this condition we can either use hardware or a software technique.

This project is the software point of view of solving the problem using a signature based method. The signatures prove to be easy and highly efficient in problem solving. It is portable as you need only the software and costs less than the existing hardware.

The CFCSS technique defines a software based technique to solve control flow errors. It divides the program into parts known as nodes and assigns a signature to each node if the signature of the node matches the signature to which it was supposed to go, the program proceeds. If the jump is wrong that is the signatures doesn't match then the program reports an error and does not end abnormally.

1.2 Approach

The CFCSS algorithm has been implemented for the Pentium 586 architecture that is the code written for the 586 environment will be handled for control flow errors.

The front end consists of a text editor implemented in Visual Basic. We have made the use of the C environment to append the signatures to the existing files and get the appropriate CFCSS version. The code transformation also makes the use of 'batch scripts' to run the 'C' code in the Visual Basic environment.

1.3 Purpose

1.3.1 Project Perspective

1.3.1.1 Why CFCSS?

CFCSS has certain advantages over the other techniques used to find control flow errors.

First and the foremost being a software technique it's portable, easy to carry from one machine to the other, which is not generally the case with hardware. It is cheaper than hardware and is highly cost efficient. The accuracy achieved by CFCSS is also greater than the accuracy of hardware.

The distinctive feature of the CFCSS over previous signature monitoring techniques is that CFCSS needs no dedicated hardware such as a watchdog processor for control flow checking because it is a pure software method. A watchdog task in multitasking environment also needs no extra hardware, but the advantage of the CFCSS over it is that CFCSS can be used even when the operating system does not support multitasking.

1.3.2 Developers Perspective

1.3.2.1 Languages Used

Different languages have been used in the project. In the heart of the project lies the assembly language of 586 for which we are building the project, that is it the code which is going to be transformed using CFCSS. We need not be fully aware on the functioning of the 586 assembly code, and all we need to know is, what are the different jump instructions in 586 instruction set, so that we can write the code accordingly? A little detailed description of the 586 environment is covered in section 3.

The text editor used to write the code is also custom made in Visual Basic so that we can have our own environment of writing the code. We can directly compile the code and run it from the front end developed in Visual Basic.

The use of the C code is mainly to transform the assembly code into the relevant CFCSS code. The C code first breaks the assembly program into nodes and then makes a graph of the total control flow in the program. After this it appends the relevant signatures to each node and then adds on a checking procedure to match the signatures when the program is run.

The use of batch scripts has also been made, so that we can execute more than one command directly on the command prompt through the visual basic interface. This was not otherwise possible.

1.3.2.2 Issues

CFCSS is a signature based software technique so the first issue raised is the amount of code added to each node that is by how much amount the length of code is increased if this technique is used. Does the code become too big after the identification of nodes and adding the signatures?

Further is the time taken to do the checking is not too much, that is the code for signature checking should be fast and efficient, and should not cause the program to slow up.

The signature checking technique should be fully efficient and address to all the problems relating to control flow errors.

1.3.3 Others Perspective

1.3.3.1 Graphs and Errors

Form the point of view of a general person the project helps in finding the errors. In an intermediate file the project draws up a complete graph depicting the overall flow of the system, so firstly a person can check whether their program is actually correct or not, that is flow wise is the system, leading to where it was intended to.

Then the graph can also be used for further error detection as to where an error has occurred

1.3.3.2 Integration

The system provides an overall integration to different types of working environments, including the very basic C environment to Visual Basic as the front end, to batch files to run the commands in the background.

1.4 Dissertation Organization

The organization of this dissertation is as follows

Section1 is introduction to the topic with minor details.

Section 2 explains CFCSS, the general approach and the procedure used in the project.

Section 3 gives an overview of the Intel architecture and the instruction set used.

Section 4 discusses the design of the project dealing with the system design to the detailed design.

Section 5 deals with the implementation exemplifying how to use CFCSS on any system.

2. Control Flow Checking using Software Signatures

2.1 Introduction

Transient or permanent faults introduced in a computer system during runtime can cause an incorrect sequence of instruction execution in the program and may cause control flow errors. If the system does not perform some run-time checking, the erroneous output may not be detected and serious damage may result. Therefore, it is important to monitor the program to detect any abnormality in the control flow or other error, and to take appropriate actions to avoid any incorrect output.

This report presents a new assigned signature monitoring technique called *Control Flow Checking by Software Signatures* (CFCSS), which monitors assigned signatures for inter-block control flow checking using instructions without using any special hardware. The program is divided into basic blocks. A *basic block* is a branch-free sequence of instructions: A node in the program graph (explained in detail in following section) represents each block, and there are no jumps into or out of the block except for the first and last instruction of the block. All nodes in the program graph are assigned different arbitrary numbers (signatures), which are embedded into the program during reprocessing or compile time. During program execution, a run-time signature *G* is stored in one of the general purpose registers called the *global signature register* (GSR), and compared with the stored signature of the node whenever control is transferred to a new node*.* For multiple branching cases, a run-time adjusting signature *D* is combined with *G.* The complete algorithm and an example program are presented in following sections.

2.2 Preliminaries

Before we present our approach, we define the terminology that will be used later. A *basic block* is a maximal set of ordered instructions in which its execution starts from the first instruction and terminates at the last instruction. There is no branching instruction in a basic block except possibly for the last one. A basic block terminates at either an instruction branching to another basic block or an instruction receiving transfer of control flow from two or more places in the program. By defining V = {v_1, v_2, ..., v_n} as the set of vertices denoting basic blocks, and E = {br_{ij}| br_{ij} is a branch from vi to vj} as the set of edges denoting possible flows of control between the basic blocks, a program can be represented by a *program graph*, P = {V, E}. These br(i,j)s are not necessarily explicit branch instructions. They also represent fall through execution paths, jumps, subroutine calls and returns. An example is shown in Fig. 3.

Vertex v_j is in the set *suc*(v_i) if and only if br_{ij} is included in E. Similarly, vertex v_i is in the set *pred*(v_j) if and only if br_{ij} is included in E. If a program is represented by its program graph P = {V, E}, br_{ij} (a branch from v_i to v_j during the execution of P) is *illegal* if br_{ij} is not included in E [3]. This illegal branch indicates a control flow error, which can be caused by transient or permanent faults in hardware such as the program counter, address circuits, or memory system [1].

If a node receives more than two transfers of control flow, it is said to be a *branch-fan-in node*, meaning that the number of nodes in *pred*(v) is greater than one. A 6 *branch insertion* occurs when one of the instructions in the node is changed to a branch instruction as the result of an error. A *branch deletion* occurs when an error causes the branch instruction of a node to change to a non-branch instruction. As a result, the node without the branch instruction merges with the node that is adjacent to it in the memory address space.

The *xor-difference* of a and b is the result of performing the bitwise XOR operation of a and b, i.e., xor-difference = a ⊕ b, where a and b are binary numbers.

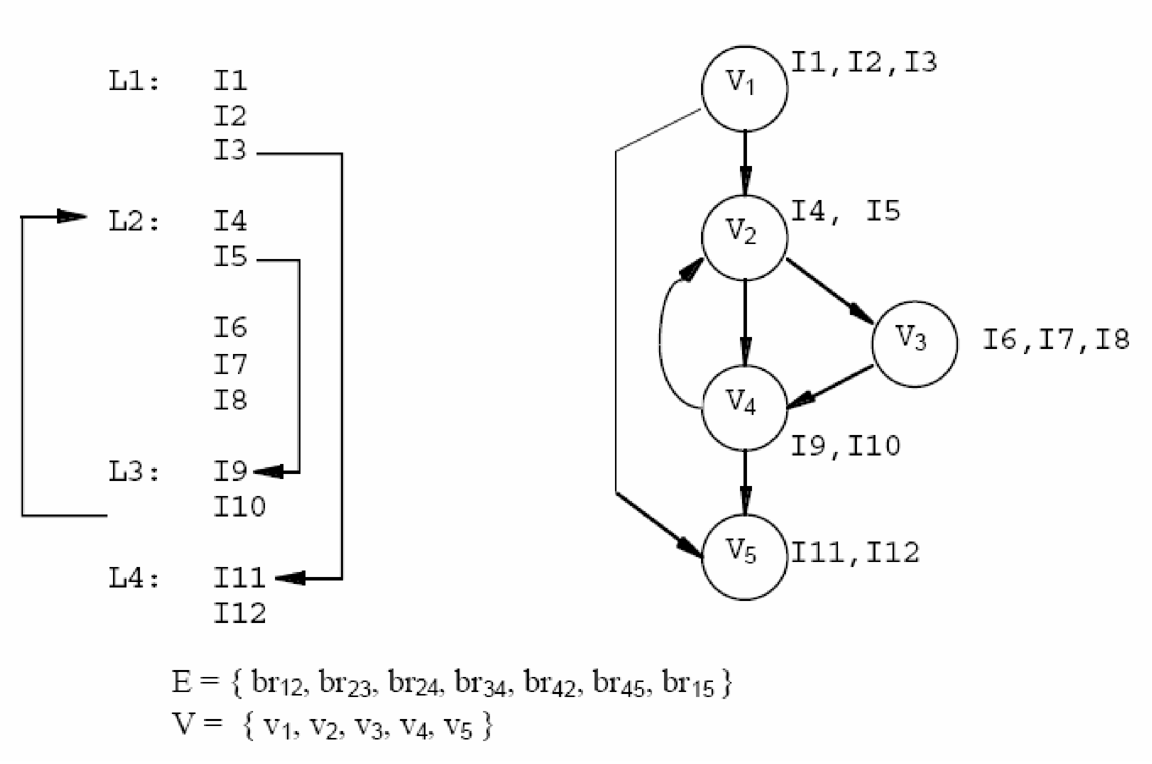


Figure 1: A sequence of instructions and its graph

2.3 Description Of CFCSS

2.3.1 The run-time signature G

Control Flow Checking by Software Signatures (CFCSS) checks the control flow of the program by using a dedicated register called the global signature register (GSR), which contains the run-time signature *G* associated with the current node (the node that contains the instruction currently executed) in the program flow graph. Every basic block (represented by a node vi in the program flow graph) is identified and assigned a unique signature s_i when the program is compiled. Let G_i be the run-time value of *G* when the program flow is in node $_{vi}$. Under normal execution of the program (no errors), G_i should be equal to s_i. If *G* contains a number different from the signature associated with the current node, it means an error has occurred in the program.

When control is transferred from one basic block to another, a new run-time signature *G* is generated by a signature function *f* at the destination node of the branch. A *signature function f* is a function that updates *G* for the current node by using two values: the signature of the previous node (source node of the branch) and the signature of the current node (destination node of the branch). We use these two values since the source and destination nodes of the branch uniquely determine each branch in E.

Suppose that the signature function *f* is defined as $\boldsymbol{f(G,\mathrm{d}_i) = G \oplus \mathrm{d}_i}$, and s_s and s_d are the signatures of the source node v_s and the destination node v_d of branch br_{sd}. The signature difference $\mathbf{d_d}$ $\mathbf{(d_d = s_s \oplus s_d)}$ is calculated in advance at compile-time and stored in the destination node v_d. Before the branch br_{sd} is taken, *G* contains *G*s (the signature s_s of the source node v_s). After the branch is taken, *G* is updated with a new value, $\boldsymbol{G_d = f(G_s, \mathrm{d}_d)}$, based on the previous value *G*s and the signature difference dd. If G_d is equal to the signature s_d of the destination node v_d, it means there is no control flow error. On the other hand, if G_d is different from s_d, it tells us that a control flow error has occurred.

We chose the XOR operation as the signature function because the XOR operation is better than other ALU operations for the purpose of checking or generating signatures. As AND, OR and XOR operations use fewer gates in the ALU than addition and multiplication, they have less chance of having an error in the ALU than addition and multiplication. We want to check the correct control flow in the original program and minimize the probability of error in the signature function. The fewer gates the signature function uses, the lower the probability of an error is in calculation of the signature functions. Furthermore, AND and OR operations may result in aliasing between one input and output; thus, the second input is not unique. Therefore, the best candidate for the signature function is the XOR operation.

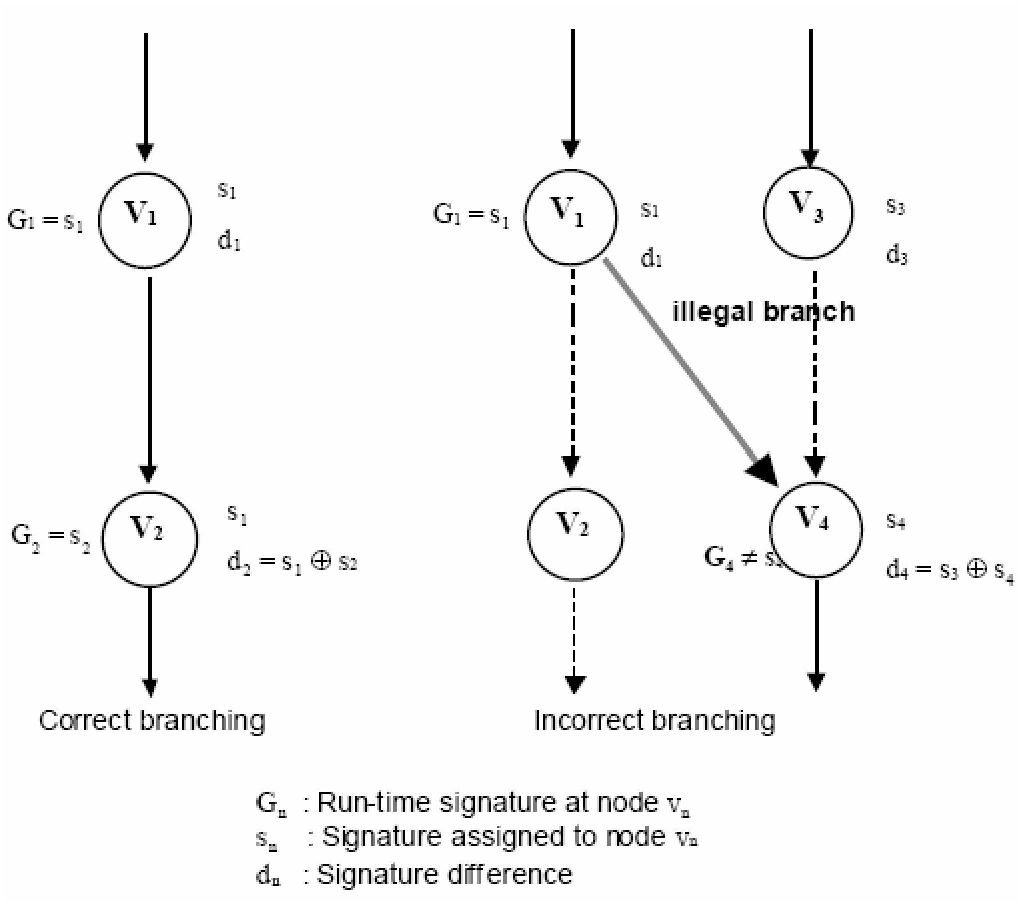


Figure 2: Detection of an illegal branch

For example, in Fig. shown above, the signature function *f* is defined as

$$\boldsymbol{f\,(G,\, \mathrm{d_i}) = G \oplus \mathrm{d_i}} \qquad \textbf{(1)}$$

and s_1 and s_2 are the signatures of the source node v_1 and destination node v_2 of the branch br_{12}, respectively. Note that nodes are assigned unique numbers as their signatures. Before a branch is taken, *G* is equal to G_1 that is the same as s_1, the signature of the source node of the branch. After the branch is taken, *G* is updated with a new run-time signature G_2, $G = G_2 = f(G_1, d_2) = G_1 \oplus d_2$. Since the signature difference d_2 is $d_2 = s_1 \oplus s_2$ and *G1* was s_1, the new run-time signature G_2 is $G_2 = f(G_1, d_2) = G_1 \oplus d_2 = s_1 \oplus (s_1 \oplus s_2) = s_2$, i.e., the updated run-time signature G_2 is the same as the signature s_2 of the current node v_2; therefore, no error has occurred. On the other hand, suppose that an illegal branch from node v_1 to node v_4 is taken. In other words, the control should have moved from node v_1 to node v_2, but an error causes an illegal branch from v_1 to v_4. Before the illegal branch is taken, G_1 is equal to s_1 as before. However, after the branch is taken, at node v_4, the new updated run-time signature G_4 is different from the signature s_4 of the new node v_4 because **$G_4 = f(G_1, d_4) = G_1 \oplus d_4 = G_1 \oplus (s_3 \oplus s_4) = s_1 \oplus (s_3 \oplus s_4) \neq s_4$**, i.e., the control flow error can be detected by observing that the run-time signature is different from the signature of the new node.

Before going into the exact algorithm, it will be helpful to describe the outline of the algorithm adding CFCSS to a program.

All basic blocks (nodes) in the program are identified and numbered. Each basic block is assigned a unique signature. The signature difference (XOR-difference between the source and destination node) of all branches is also calculated and stored in the destination nodes of all the branches. Whenever the control enters a new node, the runtime signature is updated to a new value G_{new} by the signature function f that uses the previous run-time signature G_{prev} and the signature difference d_{new} as the arguments. If the new run-time signature G_{new} is the same as the signature of the new node, the instructions in the node are executed. If G_{new} is different from the signature of the new 10 node, it means a control flow error has occurred and control is transferred to the error handling routine.

Now, we can begin the detail of the algorithm. To check the control flow, checking instructions are located at the top of each basic block, in other words, checking instructions are executed prior to the execution of the original instructions in the basic block. In Fig. 4.2, the basic block B_k consists of instructions I_1, I_2, ..., I_n and additional checking instructions located at its beginning. The checking instructions consist of two parts: the signature function that generates the run-time signature ($G = G \oplus d_k$), and the branch instruction, 'br ($G \neq s_k$) error', that compares the run-time signature with the signature of basic block B_k. In this way, a node v_k represents a basic block B_k with the checking instructions associated with B_k.

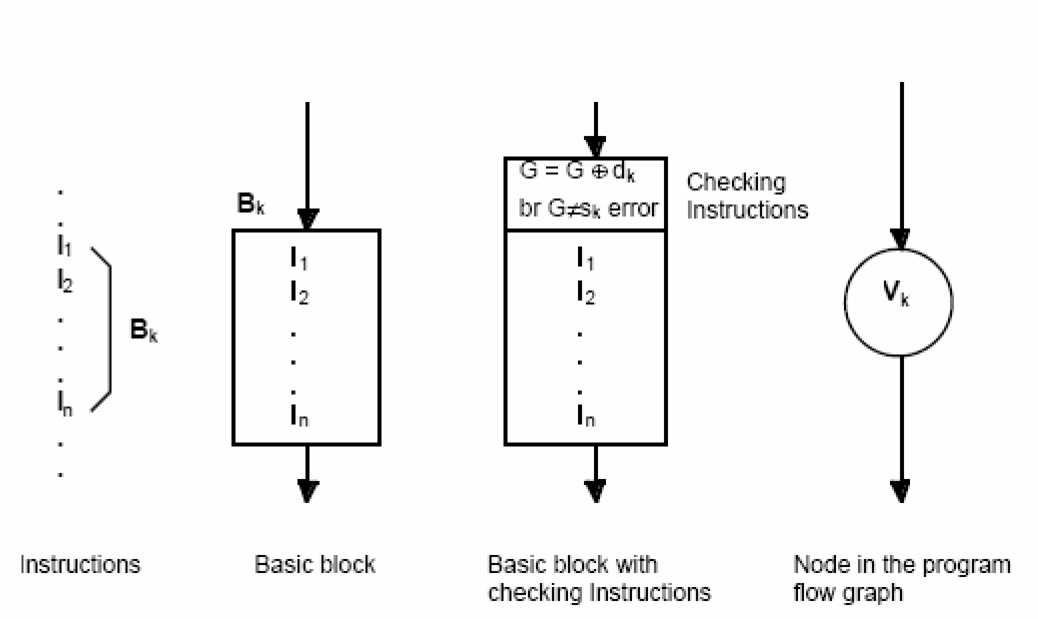


Figure 3: A basic block with checking instructions

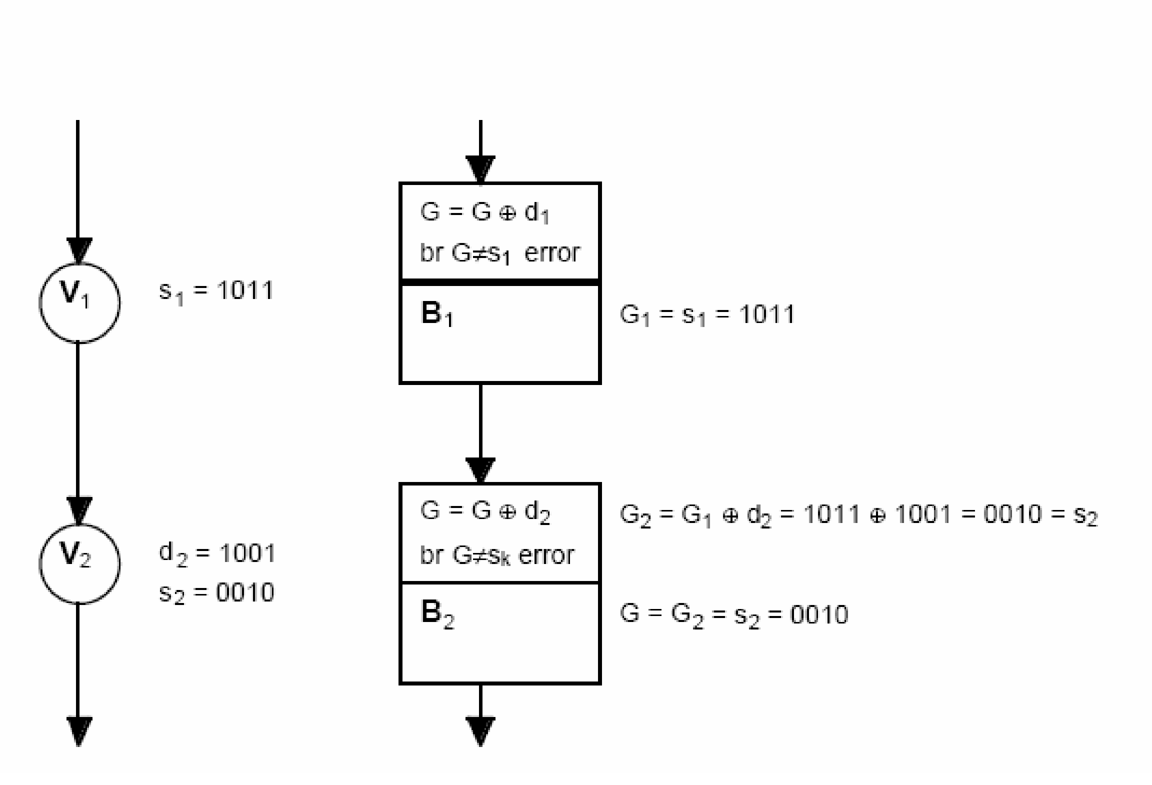


Figure 4: The checking instruction in a correct control flow

Figure shown above shows how the checking instructions work to detect errors. The control is going to be transferred from node v_1 to node v_2. G is equal to G_1 = s_1 = 1011, the signature of the current node v_1. After the branch br_{12} is taken, the signature function *f* generates the new run-time signature $G = G_2 = G_1 \oplus d_2 = 1011 \oplus 1001 = 0010$, and *G* is compared with the signature s_2 by the ‘br (G ≠ s) error’ instruction. The conditional branch instruction ‘br (G $\oplus$ s) error’ branches to the error handler if *G* and s_2 are different. In contrast, Fig. 4.4 shows the case where an illegal branch is taken and how it is detected by the checking instructions. Before an illegal branch br_{14} is taken, *G* has the value s_1. However, at node v_4, the new run-time signature $G = G_4$ is different from s_4 since *G* is 0101 and s_4 is 0110 ($G_4 = G_1 \oplus d_4 = 1011 \oplus 1110 = 0101 \neq s_4 = 0110$). This mismatch causes the following instruction ‘br (G ≠ s) error’ to transfer the control to the error handling routine.

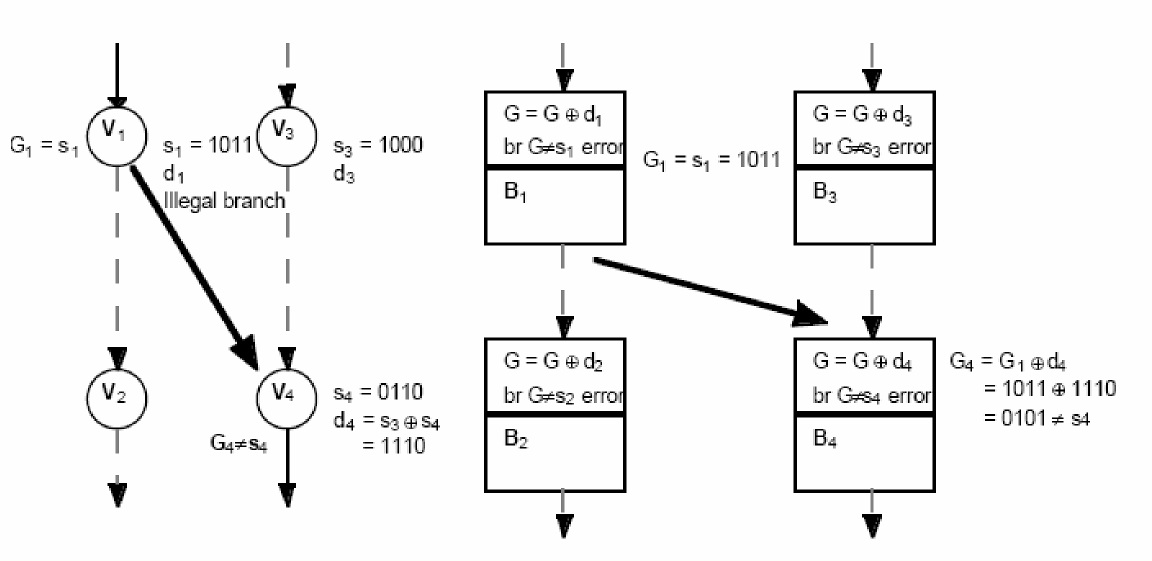


Figure 5: The detection of an illegal branch

Figure above shows the case where the illegal branch is taken to the location where the signature function instruction is located, i.e., the first instruction of the node. Following figures illustrates the case in which an error occurring in the branch instruction, for example, a bit flip in the destination field, causes an unpredictable jump to any place in the entire program, i.e., any basic block in the program and any place in that basic block. *G* has the value s_1 at node v_1. The illegal branch from node v_1 to node v_4 is taken and the control is transferred to one of the instructions in the basic block B_4, not the checking instructions. As a new run-time signature is not generated at v_4, *G* is still equal to the previous value G_1*. G* is not updated to s_4 although control is transferred to node v_4. After the instructions in node v_4 are executed, the branch from node v_4 to node v_5 is taken. At node v_5, G is updated to a new value G_5, but it is not equal to the signature of node v_5 because the previous G before the branch br_{45} was G_1(= s_1), not the correct value G_4(= s_4). Thus an illegal branch to any instruction in the node will also be caught. The detailed calculation of G is shown in the figure. Figure shows the case where an illegal branch br_{14} lands at the second instruction of the node, i.e., 'br (G ≠ s) error'. In a similar way, as the new run-time signature is not generated at v_4, G is still equal to the previous value G_1 that is not equal to s_4. Therefore, 'br (G ≠ s) error' catches this mismatch and the error is detected.

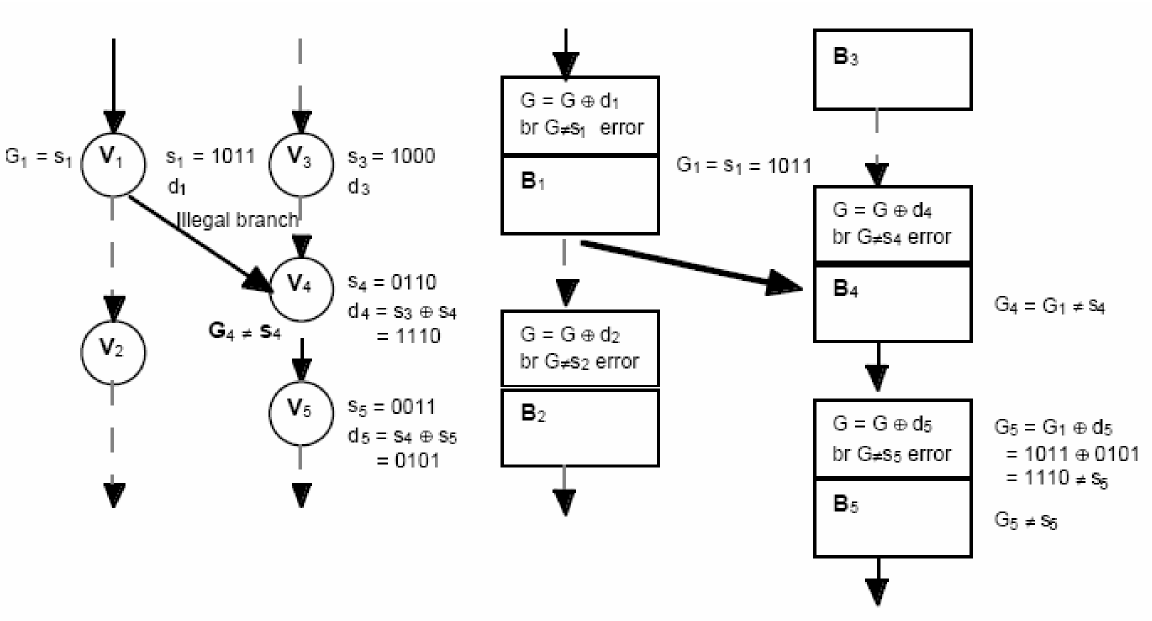


Figure 6: The detection of a branch illegally jumping to the middle of a basic block

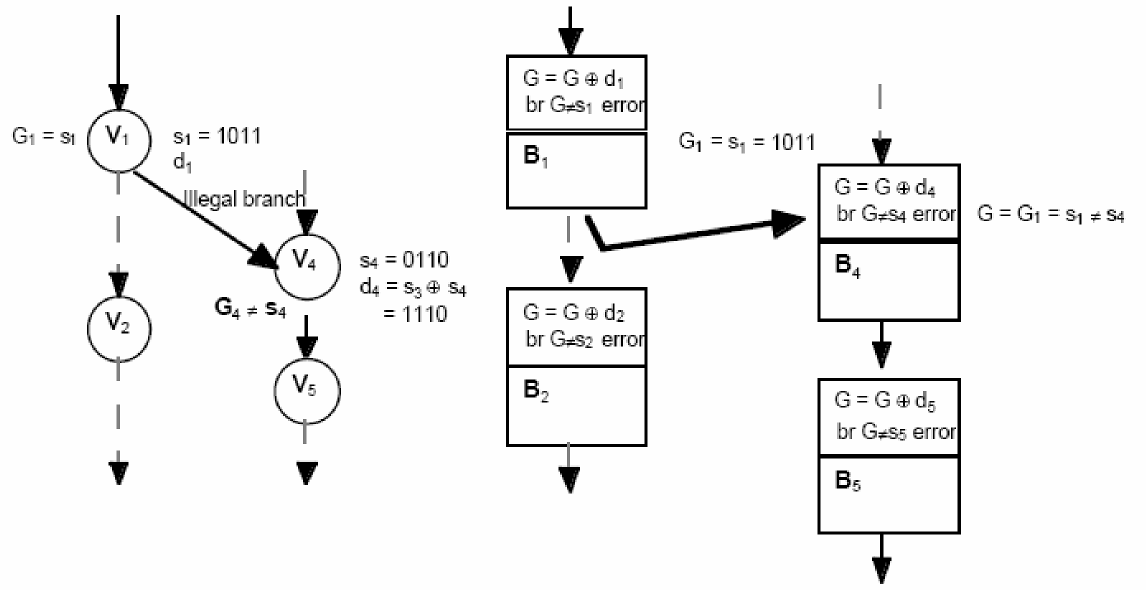


Figure 7: The detection of a branch illegally
jumping to the second instruction of a basic block

2.4 The Run-Time Adjusting Signature D

It was shown that illegal branches violating the control flow can be detected by assigning unique signatures to each of the nodes in the program graph and adding signature checking instructions to them. However, there are cases where the same signature has to be assigned to multiple nodes, for example, a branch-fan-in node. In figure shown below, the two nodes, v_1 and v_3 have branches to the same node, a branch-fan-in node v_5. If d_5 is the signature difference between nodes v_1 and v_5 as stated before ($d_5 = s_1 \oplus s_5$), there is no problem when the branch br_{15} is taken because $\boldsymbol{G_5 = G_1 \oplus d_5 = s_1 \oplus s_1 \oplus s_5 = s_5}$, which is the signature of node v_5. If the branch br_{35} is taken, however, the run-time signature *G* at node v_5 is not equal to s_5 as $\boldsymbol{G_5 = G_3 \oplus d_5 = s_3 \oplus s_1 \oplus s_5 \neq s_5}$**, if** $\boldsymbol{s_3 \neq s_1}$.

However, if we use s_1 □ s_3 as the signatures, then an illegal branch from v_1 to v_4, or from v_3 to v_2, will not be detected. In order to solve the problem of assigning the same signature to multiple predecessors of a branch-fan-in node, a run-time adjusting signature *D* is introduced. After the run-time signature *G* is generated by the signature generation function, *G* is XORed with *D* to get the signature of the branch-fan-in node; thus, at the source node, *D* has to be set to the value which makes *G* equal to the signature of the destination node. Figure below illustrates an example where *D* is used in the branch-fan-in node. At node v_5, one more checking instruction $G_5 = G_3 \oplus D$ is added. After the signature generation function $G = G \oplus d_5$, *G* is XORed with *D* that should be determined at the source nodes v_1 and v_3. Since d_5 is initially set to the XOR-difference between s_1 and s_5 ($d_5 = s_1 \oplus s_5$), when the branch br_{15} is taken, the updated run-time signature *G* is already the same as s_5; we do not need to change *G*, thus, *D* is set to zero at v_1

$$\boldsymbol{G_5 = G_5 \oplus D = s_5 \oplus 0000 = s_5} \qquad \textbf{(2)}$$

When the branch br_{35} is taken, the updated *G* at the first line of v_5 is $\boldsymbol{G_5 = G_3 \oplus d_5 = s_3 \oplus (s_1 \oplus s_5)}$. To make *G* equal to s_5, *G* should be XORed with $s_1 \oplus s_3$ at the second line, i.e.,

$$\boldsymbol{G = G_5 \oplus D = s_3 \oplus (s_1 \square s_5) \oplus (s_1 \oplus s_3) = s_5} \qquad \textbf{(3)}$$

Therefore, at the source node v_3, *D* should be set to $D = s_1 \oplus s_3$.

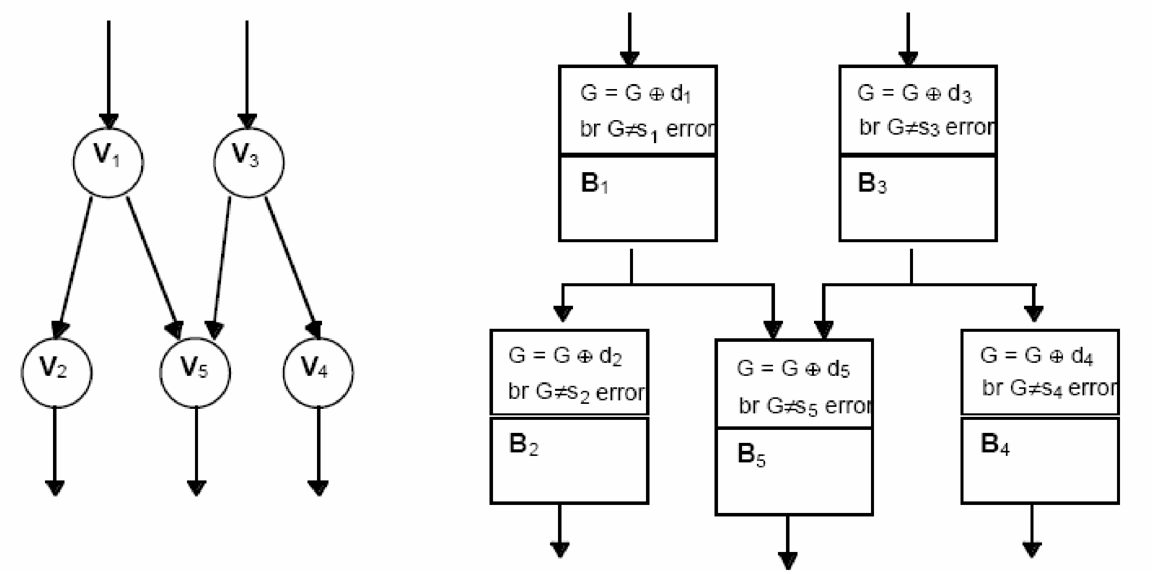


Figure 8: Node V_1 and V_3 have same signatures

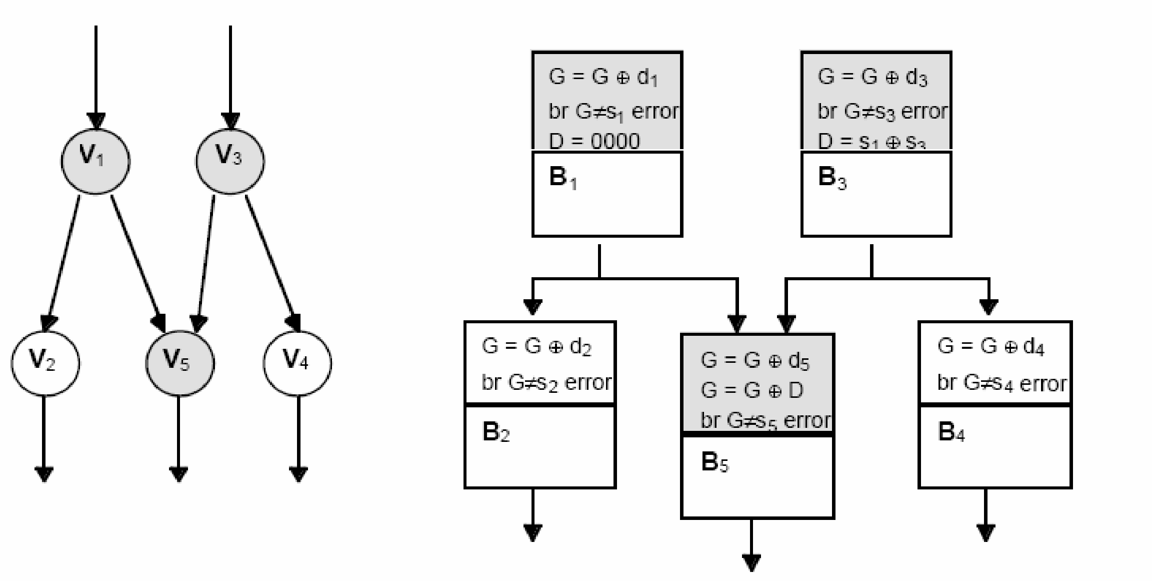


Figure 9: Node V_1 and V_3 have different signatures

For the branch br_{12}, *D* is not necessary as node v_2 is not a branch-fan-in node; only one branch is coming into v_2 and d_2 is equal to $s_1 \oplus s_2$. Thus, the updated *G* at node v_2 is equal to s_2 as in the previous case in figure shown below. In summary, if one source node has a branch to the branch-fan-in node, the node has to have one extra instruction for *D* in the checking instructions to set *D* to the appropriate value before branching. If the branch to the branch-fan-in node is taken, *D* is XORed with *G* at the destination node. If not, *D* is just ignored. In this way, we can assign arbitrary different numbers to all the nodes in the program graph.

2.5 Algorithm

The following is the complete description of *Algorithm A* that assigns signatures to each node in a program flow graph when a program is compiled.

Algorithm A

1. Identify all basic blocks, build program flow graph and number all nodes in the program flow graph.

2. Assign a signature s_i to node v_i in which $s_i \neq s_j$ if $i \neq j$, i, j = 1, 2, ..., N, N is the total number of nodes in the program.

3. For each node v_j, j = 1, 2, ……, N.

3.1 For node v_j whose *pred* (v_j) is only one node v_i, the signature difference d_j is calculated as $d_j = s_i \oplus s_j$.

3.2 For node v_j whose *pred*(v_j) is a set of nodes $v_{i1}, v_{i2}, ..., v_{im}$ therefore, v_j is a branch-fan-in node -- the signature difference is determined by one of the nodes (picked arbitrarily) as $d_j = s_{i1} \oplus s_j$. For node v_{im}, m = 1,2,… M, insert 17 an instruction $D_{im} = s_{i1} \oplus s_{im}$ into node v_{im}. This instruction should be located after 'br ($G \neq s_j$) error' instruction in v_{im}.

3.3 Insert an instruction $G = G \oplus d_j$ at the beginning of node v_j.

3.4 If v_j is a branch-fan-in node, insert an instruction $G = G \oplus D$ after $G = G \oplus d_j$ in node v_j.

3.5 Insert an instruction 'br ($G \neq s_j$) error' after the instructions placed in step 3.3 or 3.4

When a branch br_{ij} is taken, if the destination node v_j is not a branch-fan-in node, the run-time signature G_j is generated by the signature function $f(G_i, d_j) = G_i \oplus d_j$ and compared with the signature s_j of node v_j. If they match, it means no control flow error has occurred in taking branch br_{ij}.

In addition, when a branch br_{ij} is taken, if the destination node v_j is a branch-fan-in node, the run-time signature G_j is generated by the signature function and D, i.e.,

$$\boldsymbol{G_j = f(\ f(G_i, d_j), D_j)}$$

If they match, it means no control flow error has occurred in taking branch br_{ij}.

CFCSS will detect the following types of control flow errors. They are presented as *Corollary 1-5* and accompanied by their proofs.

***Corollary 1*.** An illegal branch taken to the signature function instruction -- the first line of the node -- will be detected.

***Proof*.** Suppose that br_{ij} is an illegal branch, thus $v_i \notin pred(v_j)$. At node v_i, G is equal to s_i. After the branch is taken, the new run-time signature is generated, $G = G_j = G_i \oplus d_j = s_i \oplus s_k \oplus s_j$ in which s_k is the signature of node v_k, where $v_k = pred(v_j)$. Since s_i, s_k, and s_j are all different numbers, $G = s_i \oplus s_k \oplus s_j \neq s_j$. Mismatch occurs and the error is detected.

***Corollary 2*.** An illegal branch taken to the instruction br (G ≠ s) error -- the second line of the node -- will be detected.

***Proof*.** Suppose that br_{ij} is an illegal branch and the branch is taken to the second line of the node, i.e., skipped the signature function. Since the new G was not generated, G is still equal to s_i, not s_j. Therefore, 'br (G≠ s) error' instruction sees the mismatch and detects the error.

***Corollary 3*.** An illegal branch to the body of the node where the original basic block sits will be detected.

***Proof*.** Suppose that br_{ij} is an illegal branch and the branch is taken to a place where one of the instructions in the original basic block is located, i.e., skipped the checking instructions and landed at one of the instructions in the original basic block. Since the new *G* is not generated at node v_j, *G* is equal to s_i, not s_j although the control is transferred to the node v_j. After the instructions in node v_j are executed, br_{jk} is taken, where v_k = *suc*(v_j). The checking instructions in node v_k generate the updated $G = G_k = G_j \oplus d_k = G_i \oplus (s_j \oplus s_k) = s_i \oplus s_j \oplus s_k$. Since s_i, s_k and s_j are all different numbers, $G = s_i \oplus s_k \oplus s_j \neq s_j$. Mismatch occurs and the error is detected.

***Corollary 4*.** A branch insertion inside a node will be detected if it is an illegal branch.

***Proof*.** Suppose that br_{ik} is inserted at node v_i, $br_{ik} \notin E$ (br_{ik} is an illegal branch). At node v_i, *G* is equal to s_i. After br_{ik} is taken to the first instruction of node v_k, the new updated *G* is $G = G_k = G_i \oplus d_k = s_i \oplus (s_k \oplus s_l)$ in which s_l is the signature of node v_l, where v_l = *pred*(v_k). Since s_i, s_k, and s_l are all different numbers, $G = s_i \oplus s_k \oplus s_l \oplus s_j$ unless aliasing occurs (discussed later). Mismatch occurs and the error is detected. By *Corollary 3*, a branch to other instructions of the node will also be detected.

***Corollary 5*.** The deletion of an unconditional branch instruction from the node will be detected.

***Proof*.** Suppose that the branch instruction br_{ij} at node v_i is changed to another instruction; therefore, br_{ij} is removed from E and an adjacent node v_k is merged into the node vi. Then, the signature of this node is changed from s_i to s_k in the middle of the node where vi and v_k are merging; thus, the *G* should be updated to s_k. However, since $v_i \notin$ *pred*(v_k), *G* will not match with s_k. Therefore, the error is detected. This is similar to the case where an illegal branch $br_{ik} \notin E$ occurs.

2.6 Aliasing

If multiple nodes share multiple branch-fan-in nodes as their destination nodes, aliasing may occur between legal and illegal branches, and cause an undetectable control flow error. In Figure below, node v_5 is a branch-fan-in node that has three source nodes v_1, v_2 and v_3 (*pred*(v_5) = {v_1, v_2, v_3}). Node v_6 is also a branch-fan-in node but it has only two source nodes v_2 and v_3, not node v1 (*pred*(v_6) = {v_2, v_3}). According the step 3.4 of *Algorithm A*, first, the signature difference d_5 of node v_5 is determined as $d_5 = s_2 \oplus s_5$. The runtime signature D_2 of node v_2 is $D_2 = s_2 \oplus s_2 = 0000$, D_3 is $D_3 = s_2 \oplus s_3$, and D_1 is $D_1 = s_2 \oplus s_1$. Furthermore, for the branch-fan-in node v_6, the signature difference d_6 should also be calculated with node v_2 ($d_6 = s_2 \oplus s_6$) since {v_2, v_3} is a subset of both *pred*(v_5) and *pred*(v_6). In other words, d_6 can be either $d_6 = s_2 \oplus s_6$ or $d_6 = s_3 \oplus s_6$. However, since *pred*(v_5) ={v_1, v_2, v_3} and *pred*(v_6) = {v_2, v_3} have the same subset {v_2, v_3} and d_5 of v_5 is already calculated with the signature of v_2, d_6 of v_6 should also be calculated with the signature of v_2 ,i.e., $d_6 = s_2 \oplus s_6$. As a result, both nodes v_5 and v_6 have the difference signature calculated with the same s_2 ($d_5 = s_2 \oplus s_5$, $d_6 = s_2 \oplus s_6$).

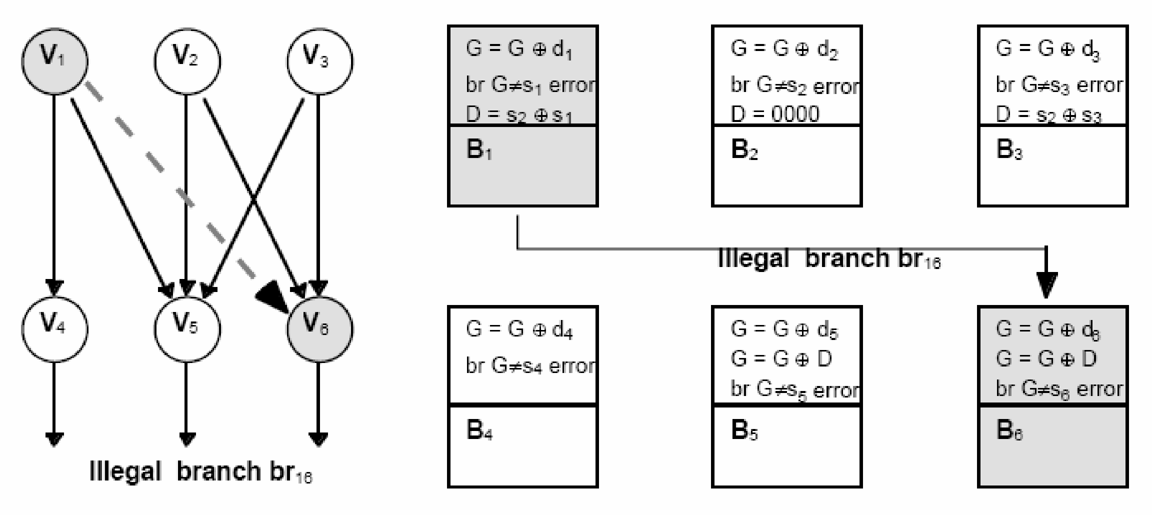


Figure 10: Aliasing causing an undetectable control flow error

Suppose that an illegal branch br_{16} occurs and lands at the first line of node v_6, where the instruction of signature function $f(G_{\text{prev}}, d_6)$ is located. G_{prev} is $G_1 = s_1$ and updated G is equal to $\boldsymbol{G = G_6 = f(G_{\text{prev}}, d_6) \oplus D = (G_1 \oplus d_6) \oplus (s_2 \oplus s_1) = (s_1 \oplus (s_2 \oplus s_6)) \oplus (s_2 \oplus s_1) = s_6}$. The updated *G6* is equal to s_6; therefore, this illegal branch is not detected. This aliasing error is caused by the fact that more than two branch-fan-in nodes have their signature differences calculated with the signature of the same node, and their predecessor sets are not equal. More specifically, the condition for aliasing error is:

Aliasing error occurs when $d_i = s_s \oplus s_i$, $d_j = s_s \oplus s_j$, but *pred*(v_i) $\oplus$ *pred*(v_j). If *pred*(v_i) - *pred*(v_j), an illegal branch from a node in *pred*(v_i) - *pred*(v_j) (assuming *pred*(v_j) - *pred*(v_i)) to node v_j is undetectable when that branch is taken to the location of the instruction for the signature function.

If the illegal branch is taken to any location except for the first line of the node -- the instruction for the signature function -- the control flow error is detected because the new run-time signature associated with the destination node is not generated.

In other words, the illegal branch is detected unless it lands at the first line of the destination node that satisfies the condition described above. With this observation, we can avoid the undetectable illegal branch if we assign signatures to the nodes in the following way.

If we assume one bit error, and the Hamming distance between the addresses of the first instructions in nodes v_5 and v_6 is greater than one, this undetectable illegal branch is avoided; one bit error in the destination field of the branch instruction at node v_1 cannot cause an illegal branch to the location of the first line of node v_6. Similarly, if we assume *m* bit error and the addresses of the first instructions of all successor nodes are different by Hamming distance greater than *m*, undetectable illegal branches caused by aliasing will be avoided.

.

3. INTRODUCTION TO THE INTEL PENTIUM PROCESSOR

In 1985, Intel introduced the first in a line of 32-bit microprocessors compatible with the already broad base of existing Intel architecture software. That was the Intel386 microprocessor. The Intel 32-bit architecture has since grown to become the standard for cost-effective, high performance computing with an installed base of over 40 million units. Intel has continued to evolve and improve the basic implementation by incorporating the most advanced computer design and silicon technology. The Intel Pentium family is the most recent product of that effort.

The Intel Pentium processor, like its predecessor the Intel486 microprocessor, is fully software compatible with the installed base of over 100 million compatible Intel architecture systems. In addition, the Intel Pentium processor provides new levels of performance to new and existing software through a reimplementation of the Intel 32-bit instruction set architecture using the latest, most advanced, design techniques. Optimized, dual execution units provide one-clock execution for "core" instructions, while advanced technology, such as superscalar architecture, branch prediction, and execution pipelining, enables multiple instructions to execute in parallel with high efficiency. Separate code and data caches combined with wide 128-bit and 256-bit internal data paths and a 64-bit, burstable, external bus allow these performance levels to be sustained in cost-effective systems. The application of this advanced technology in the Intel Pentium processor brings "state of the art" performance and capability to existing Intel architecture software as well as new and advanced applications.

3.1 Modes of Operation

The Pentium processor has two primary operating modes and a "system management mode." The operating mode determines which instructions and architectural features are accessible. These modes are:

3.1.1 Protected Mode

This is the native state of the microprocessor. In this mode all instructions and architectural features are available, providing the highest performance and capability. This is the recommended mode that all new applications and operating systems should target.

Among the capabilities of protected mode is the ability to directly execute "real-address mode" 8086 software in a protected, multi-tasking environment. This feature is known as Virtual-8086 "mode" (or "V86 mode"). Virtual-8086 "mode" however, is not actually a processor "mode," it is in fact an attribute which can be enabled for any task (with appropriate software) while in protected mode.

3.1.2 Real-Address Mode (also called "real mode")

This mode provides the programming environment of the Intel 8086 processor, with a few extensions (such as the ability to break out of this mode). Reset initialization places the processor in real mode where, with a single instruction, it can switch to protected mode.

3.1.3 System Management Mode

The Pentium microprocessor also provides support for System Management Mode (SMM). SMM is a standard architectural feature unique to all new Intel microprocessors, beginning with the Intel386 SL processor, which provides an operating-system and application independent and transparent mechanism to implement system power management and OEM differentiation features. SMM is entered through activation of an external interrupt pin (SMI#), which switches the CPU to a separate address space while saving the entire context of the CPU. SMM-specific code may then be executed transparently. The operation is reversed upon returning.

3.2 The Pentium Architecture

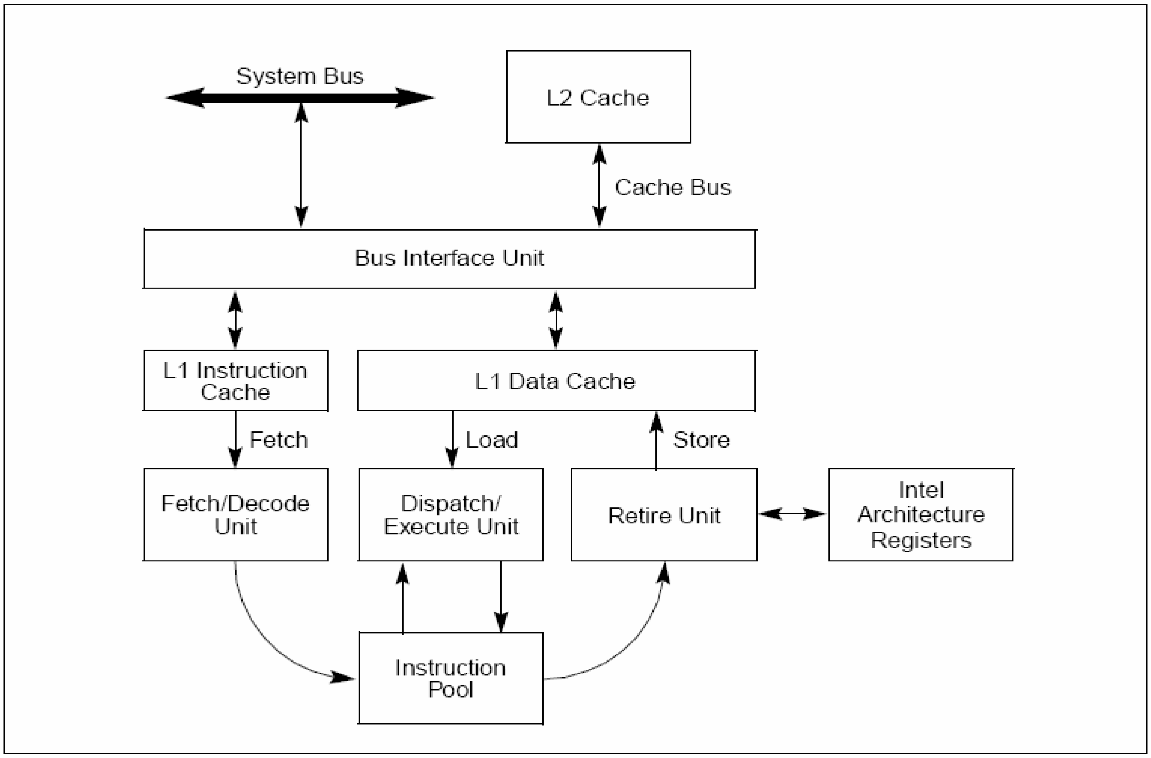


Figure 11: The Basic Pentium Architecture

Figure shows a functional block diagram of the Pentium Pro processor micro architecture. In this diagram, the following blocks make up the four processing units and the memory subsystem shown in the first figure:

- *Memory subsystem*—System bus, L2 cache, bus interface unit, instruction cache (L1), data cache unit (L1), memory interface unit, and memory reorder buffer.
- *Fetch/decode unit*—Instruction fetch unit, branch target buffer, instruction decoder, microcode sequencer, and register alias table.
- *Instruction pool*—Reorder buffer
- *Dispatch/execute unit*—Reservation station, two integer units, two floating-point units, and two address generation units.
- *Retire unit*—Retire unit and retirement register file.

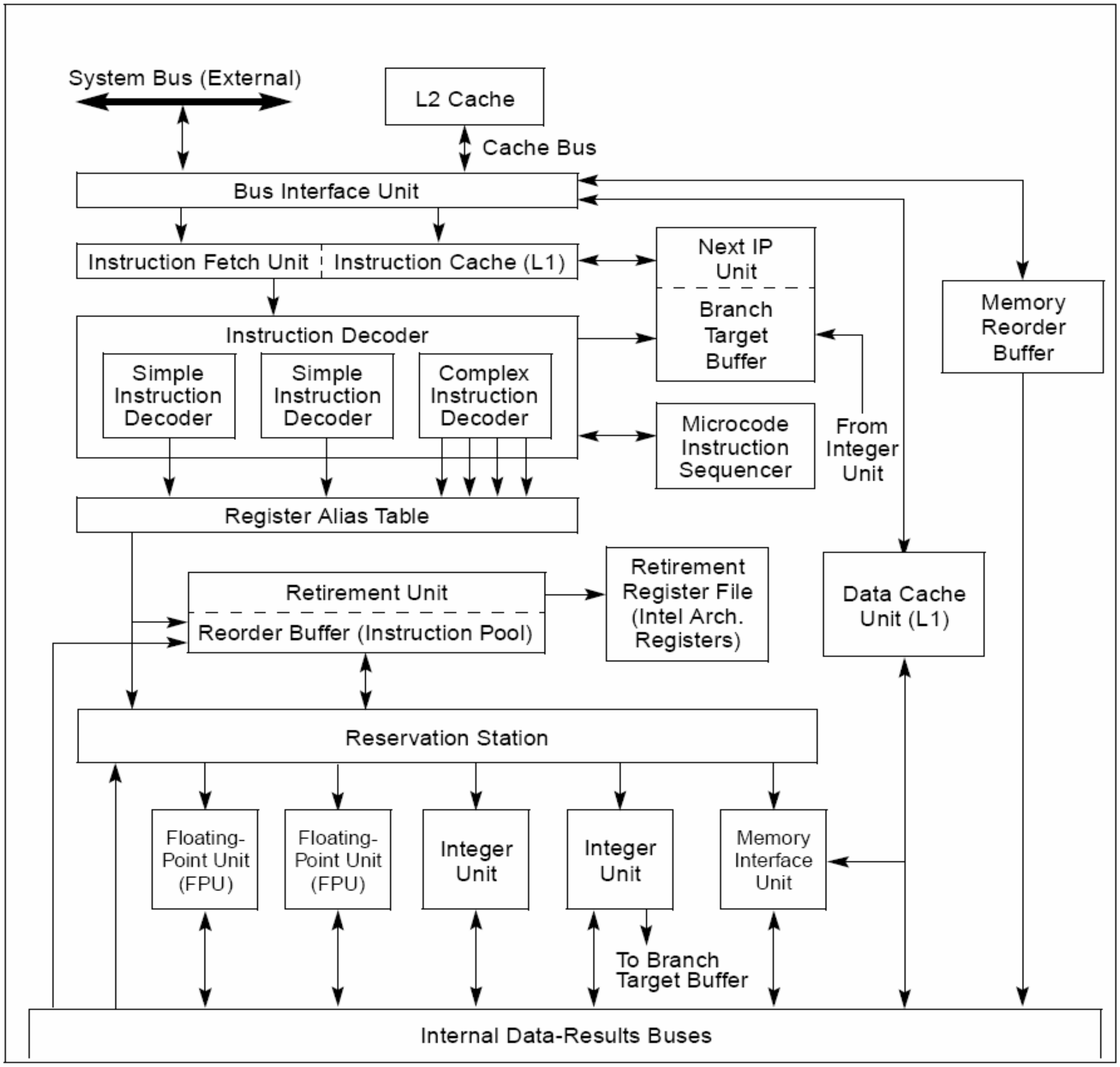


Figure 12: The Functional Block Diagram of the Pentium Architecture

3.3 The Instruction Set

3.3.1 The General Instruction Format

All Intel Architecture instruction encodings are subsets of the general instruction format shown in Figure 2-1. Instructions consist of optional instruction prefixes (in any order), one or two primary opcode bytes, an addressing-form specifier (if required) consisting of the ModR/M byte and sometimes the SIB (Scale-Index-Base) byte, a displacement (if required), and an immediate data field (if required).

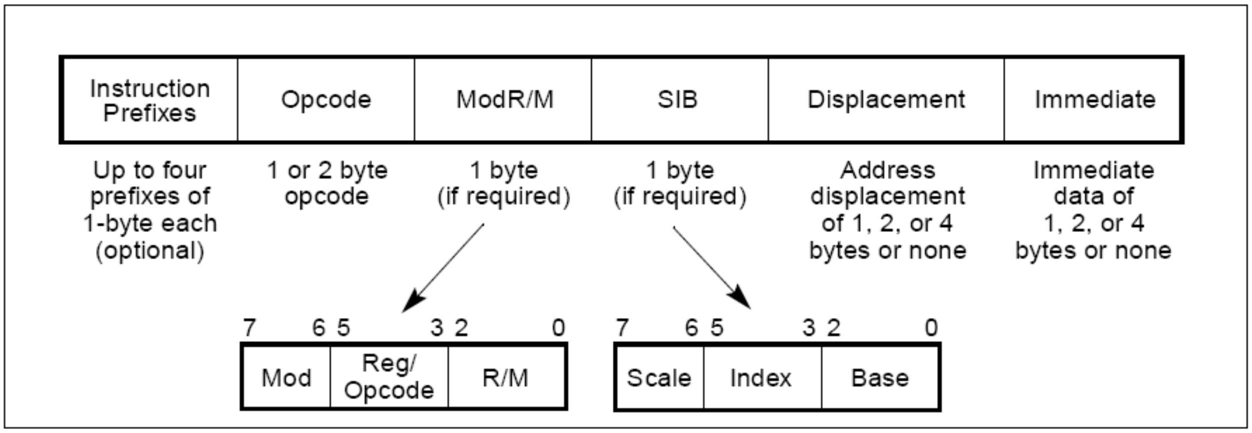


Figure 13: Intel Architecture Instruction Format

3.3.2 Instruction Set

There are different types of instruction in the whole instruction set and can be divided into further categories :-

1. Data Transfer instructions
2. Binary arithmetic instructions
3. Decimal arithmetic instructions
4. Logic instructions
5. Shift and rotate instructions
6. Bit and byte instructions
7. Control and jump instructions

8. Flag control instructions
9. Segment register instructions
10. Miscellaneous instructions

Out of these the most important instructions for us are the “Control and jump instructions”, because these are the only instructions that cause control flow errors.

These are the instructions that have to be checked for dividing the nodes. Apart from these instructions the nodes are also divided when a label starts. So, these are the two primary conditions in dividing the code into nodes.

Of the jump instructions there is one such instruction that has to be taken special care of, it is the unconditional jump statement “JMP”, which does not take any condition as a parameter so it jumps the whole program to a new location, but the overall fan-in of the node is just one.

The whole set of instructions of our interest are shown in Appendix B.

4. DESIGN

The purpose of the design phase is to plan a solution of the problem specified by the requirement document. The phase is the first step in moving form the problem domain to the solution domain. In other words, starting with what is needed; design takes us toward how to satisfy the needs. The design of a system is perhaps the most critical factor affecting the quality of the software; it has a major impact on the later phases, particularly testing and maintenance. The output of this phase is the design document. This document is similar to blueprint or plan for the solution and is used later during implementation, testing, and maintenance.

The design activity is often divided into two separate phases:

- *System design*
- *Detailed design*

4.1 System Design

System design, which is sometimes also called top-level design, aims to identify the modules that should be in the system, the specification of these modules, and how they interact with each other to produce the desired results. Design is a much more creative process than analysis. It involves working with the unknown new system rather than analyzing the known system. Thus involves working with the unknown new system rather than analyzing the known system.

The first process in system design will be to divide the whole requirement into major modules as follows:

1. ***Translator:*** This is the main module which will convert the assembly code to CFCSS equivalent assembly code.

2. ***IDE:*** This is the user interaction module. This will help the user to work easily with the assembler and CFCSS code level. This will help the user to easily write the code and will help the person to compile it properly.

3. ***Output File:*** This will be the one which will give the actual output in a log file and will help the user to check if actually every thing is gone fine while executing or not.

4. ***Interaction Mechanism:*** This is the one which will help us in communicating between various code and modules.

4.2 Detailed Design

4.2.1 Translator

The translator is mainly made up of 3 C files. The first one used to identify all the nodes along with the line numbers. The second file used to assign signatures to the nodes, find the difference, the run time adjustment signatures and making the graph. The third file is used to insert the appropriate code in the assembly code at proper positions so that the signatures could be matched at runtime, ensuring the proper functioning of the program.

4.2.1.1 CFCSS.C

This file is basically used to identify the nodes in the program, along with the initial and ending line numbers. These nodes are extracted and given node numbers and stored in a text file, from where they are further processed.

The intermediate file which will be building after running the program will have the number of nodes in the first line. The second line will consist of the node's name, along with the node's starting and ending line number. These values will be separated using a delimiter '|'.

Nodes are found with these things considering in mind. Any node will be starting with line number having some label at beginning of line 0r immediately after the jump instruction.

End line of any node can be the line number just before the new label starts or the line which have conditional or unconditional jump.

Node structure made in this file will be something like as follows.

Example: *Node name* |*start line number* |*end line number*

Start |28 |30

4.2.1.2 CFCSSG.C

The identified nodes from the intermediate file produced in 'CFCSSC.C' are read and signatures are provided to each node.

Each node is provided with a random signature such that no two nodes have the same signature. Even after providing random signatures the signatures are double checked, because they should be unique to each node. These values are written to another intermediate file. This intermediate file also contains the signature difference and the run time adjustment signature. So the bulk of the processing takes place in this file.

The structure of each line of the intermediate file should be:

Node name | *node number* | *signature* | *S line no* | *E line no* | *difference* | *RAS count* | *comma separated RAS values*

Example *again* | *1* | *14344* | *30* | *35* | *17804* | *3* |*17804, 20217, 30462*

The intermediate file also provides with a graph of the nodes, with the proper flow of the program, so a programmer can also check the overall flow of the program and can check for any mistakes which he has made while writing the code. The flow graph is very helpful in checking the working of the program. It is also useful in finding wrong jumps and finding areas which can never be reached, thus making the task of the programmer easier.

The result is an intermediate file containing all the required information about the node. The node name, its number, signature, difference, fan in of the node and the different runtime adjustment signatures along with the graph.

4.2.1.3 CFCSSNF.C

This file is used to add the entire extra code to the original file, i.e. it appends the signature, the difference and the RAS values in the starting of each node in the program. It also has a routine to check the signatures at runtime. This routine extracts all the information from the node and checks for the validity of the signature before moving in to the next node.

The file first extracts all the information from the intermediate file created by CFCSSG.C and stores them in a structure which is the data structure used for storing the nodes information.

The code adds in a function prototype just above the data section for the function which will be used to check for the signatures at run time. The data section is introduced with a lot of variables; it has the signature to the first node and its difference. There is a RAS array for each node having the RAS values stored for each of its node.

Then the main changes come in the code section, the code section starts with calling the function for the first node checking for the signatures and putting the result of a match in the log file which will tell about the correct or wrong execution. Each node is assigned with its signature, difference and the count of RAS for that node.

4.2.2 User Interface Design

VB is used as Interface between **User** and **Assembler**. In VB the IDE for assembly code is developed which has the following attributes:

a. File Menu
b. Edit Menu
c. View Menu
d. Tools Menu
e. Project Menu
f. Widow Menu
g. Help

These all Menus further have options in it.
Following are the detailed description over all these Menus.

4.2.2.1 File Menu

File Menu contains the following options:

1. New
2. Open
3. Save
4. Save As
5. Close
6. Save & Close All
7. Command Prompt
8. Set Current Directory
9. Exit

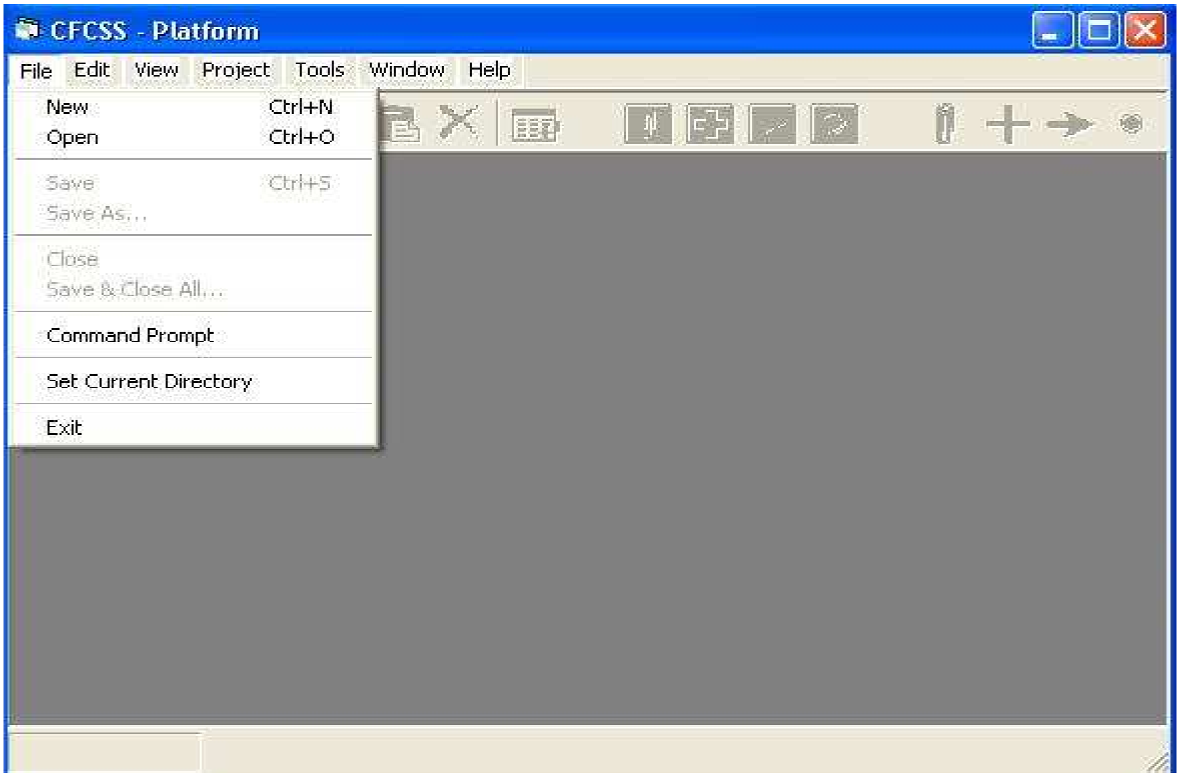


Figure 14: File Menu

New

The New option is present here for making of New File of Assembly i.e. here instead of opening a file present at any other location we can make our own file also.

Open

Open option is there in case we already have a program made in Assembly and we need to pick up its code from that location.

Save

Save option is present for saving any of the changes made in any program or to save any new file.

Save As

Save As option is provided in case we want to save any particular program with any other name.

Close

Close option exists in case we want to close any particular file opened.

Save & Close All

This option closes all the opened files after saving them with their respective names.

Command Prompt

There may be need at times when we want to manually work with assembler, here this command prompt helps us indirectly by providing a link with shell.

Set Current Directory

Set Current Directory is there for setting the directory where we want to work i.e. where we want to save our all files etc.

Exit

Exit options as name suggests is for getting out of IDE.

4.2.2.2 Edit Menu

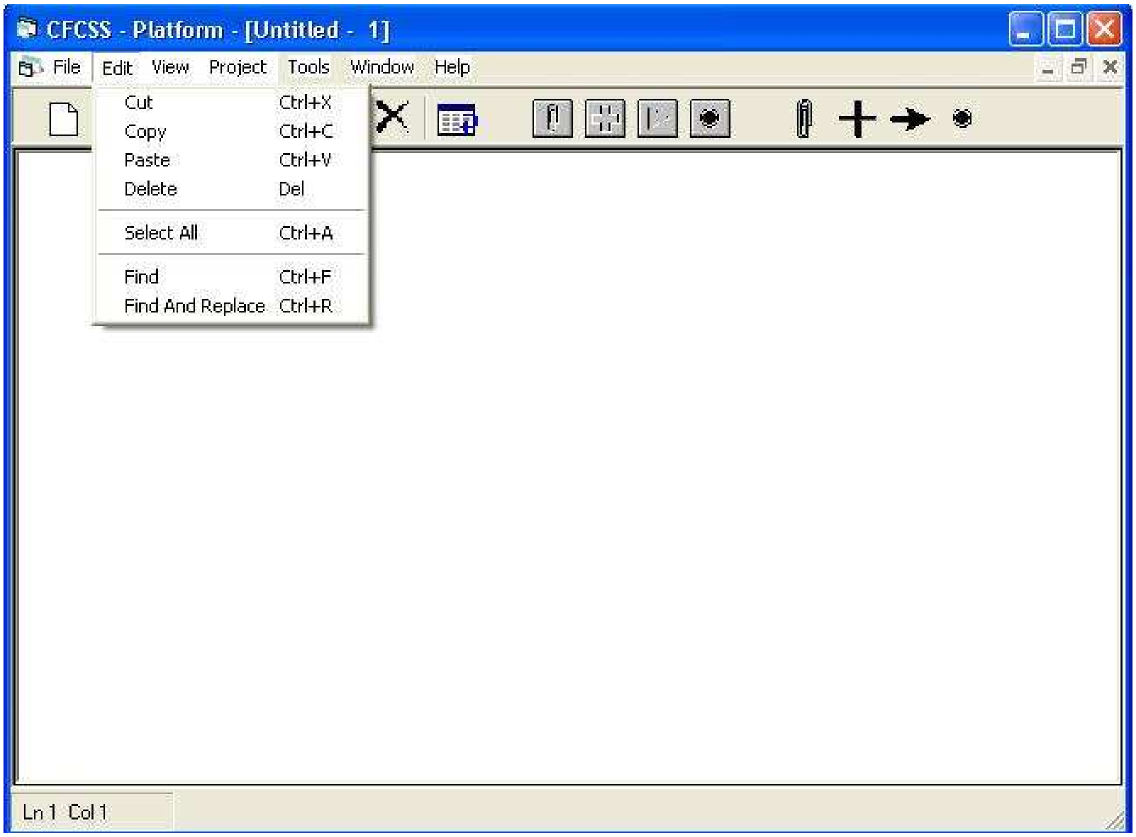


Figure 15: Edit Menu

The Edit Menu consists of following sub Menus or Items:

1. Cut
2. Copy
3. Paste
4. Delete
5. Select All
6. Find
7. Find and Replace

Cut

Cut option is provided for cutting some text from a particular position and inserting it into any other location with the help of paste command.

Copy

Sometimes we need to copy some particular text from one location to another in that case copy command comes into play. Copy is provided for copying the contents from one location to another.

Paste

Paste command is used in conjunction with copy/cut. With the help of cut and copy we can only cut or copy the contents from that location but we can't insert it to our required location. Insertion of text to required location is done with the help of Paste Command.

Delete

Sometimes some unwanted or erroneous data is entered by mistake, and we want to delete it from our file. In that case Delete command is used. It is there for deleting the selected text contents.

Select All

Select All is used for selecting all the contents of opened file.

Find

Find is used for searching first location of some particular text in file. Next is used for finding the next location of that text location. It may/may not be case sensitive.

Find and Replace

Find and Replace is similar to Find Command. It actually replaces the text found by the find command.

4.2.2.3 View Menu :

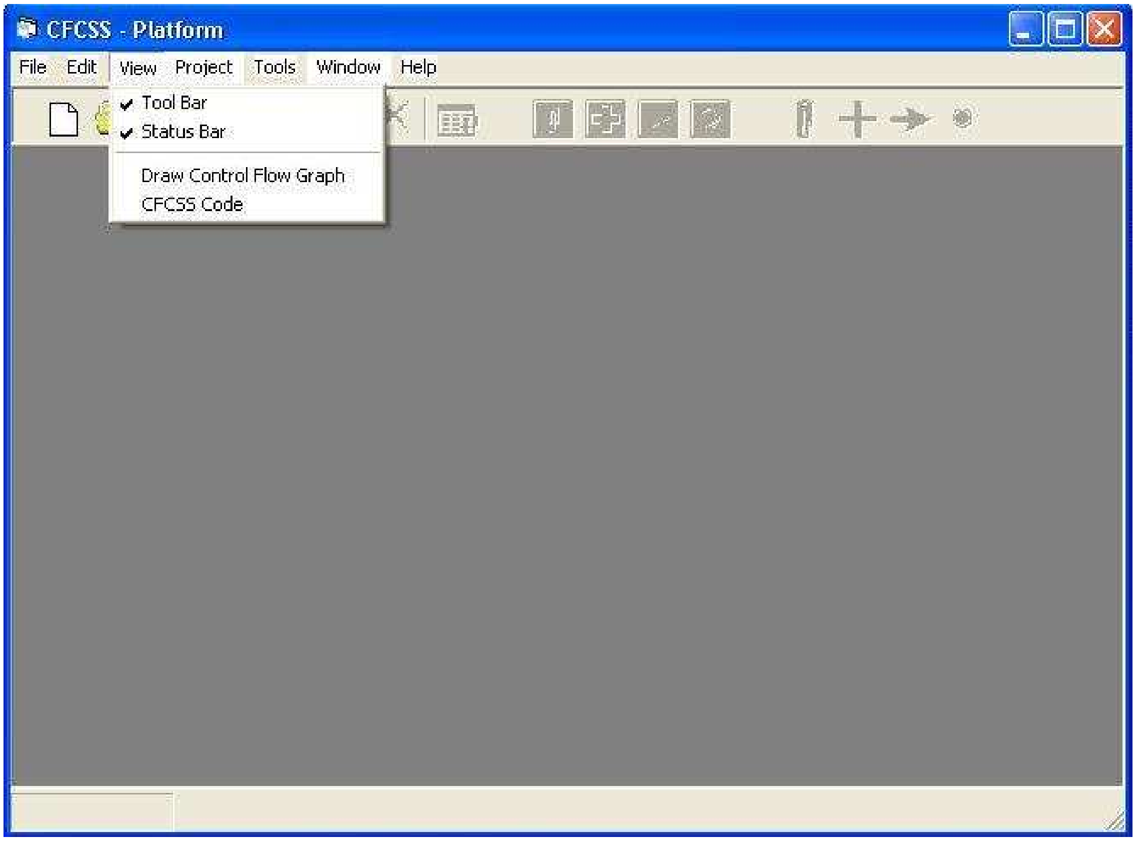


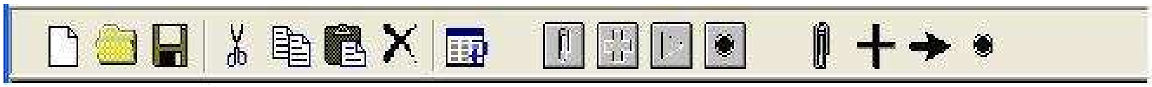
Figure 16: View Menu

The View Menu comprises of following options:

1. Tool Bar
2. Status Bar
3. Draw Control Flow Graph
4. CFCSS Code

Tool Bar

Tool Bar is option which is either enabled or disabled. If it is enabled then the toolbar is visible to us otherwise not. The toolbar is shown below:



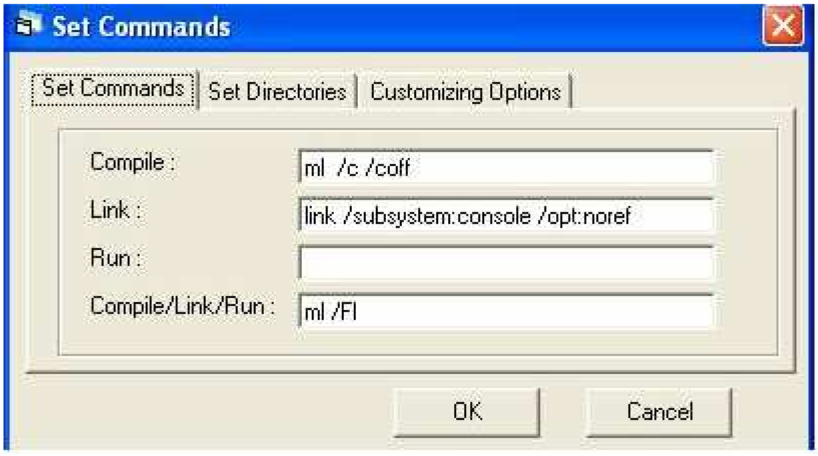
Status Bar

Status Bar is used for showing the current position of cursor in terms of line & column number.

CFCSS Code

This option is available for the user if he wants to see the code which is to be finally executed by the assembler. It contains all the codes entered by the programmer in the assembly coding.

4.2.2.3 Tools Menu



Compile : ml /c /coff

Link : link /subsystem:console /opt:noref

Run :

Compile/Link/Run : ml /Fl

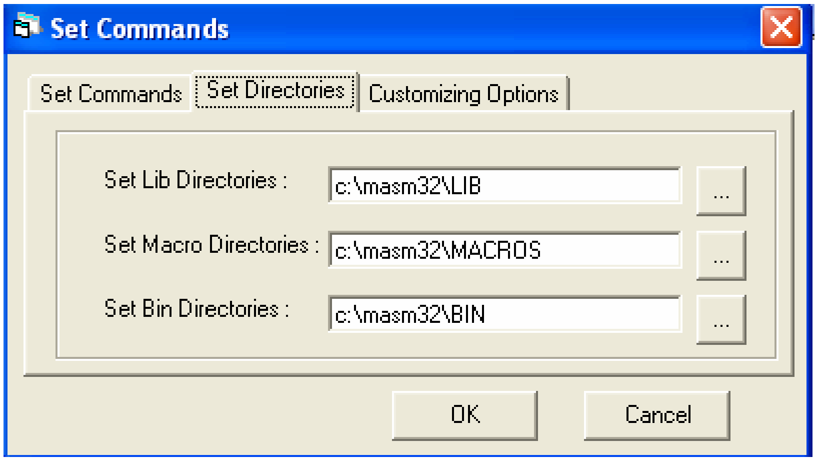
Figure 17: Set Commands Menu

The tools menu consists of further options which internally contains:

1. Set Commands
2. Set Directories
3. Customizing Options

Set Commands

It contains the options for changing the commands for compiling, linking and running as per the requirements of assembler. These commands are based on Assembler used. e.g. The commands for compiling, linking and running shown above refer to MASM32 assembler.



Set Lib Directories : c:\masm32\LIB

Set Macro Directories : c:\masm32\MACROS

Set Bin Directories : c:\masm32\BIN

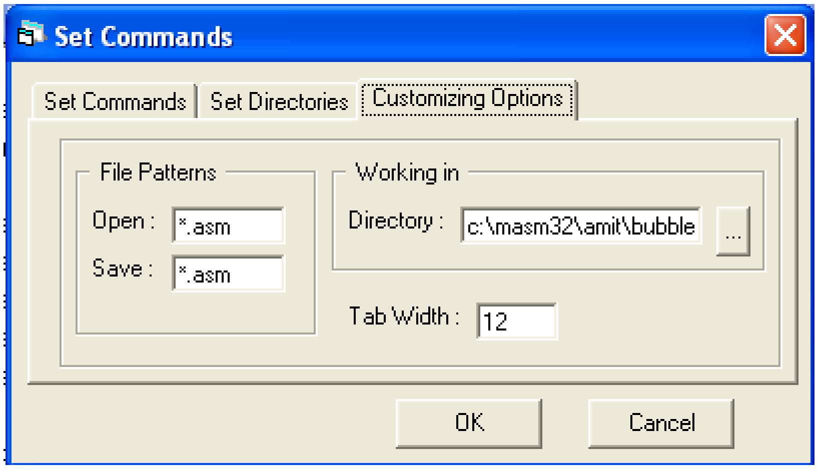
Set Directories

It helps us in setting various directories required for running our assembly code. These options include:

1. Set Lib Directory
2. Set Macro Directory
3. Set Bin Directory

These are required for linking, compiling and running of our program.

Customizing Options



Open : *.asm
Save : *.asm
Directory : c:\masm32\amit\bubble
Tab Width : 12

These options include:

1. Different file Patterns (which can be opened and saved)
2. Current Working Directory
3. Tab Width

Here file patterns chosen for saving and opening of file is *.asm which is actually the extension of Assembly program.

Current Working Directory is there which is default for opening and saving files. i.e. whenever we want to open a file or save it this particular folder will be opened.

And last of all is Tab Width. This is for setting the tab size.

4.2.2.1 Project Menu

Project Menu contains the following options:

1. Compile
2. Compile and Run
3. Build All
4. CFCSS compile
5. CFCSS compile and run
6. CFCSS build all

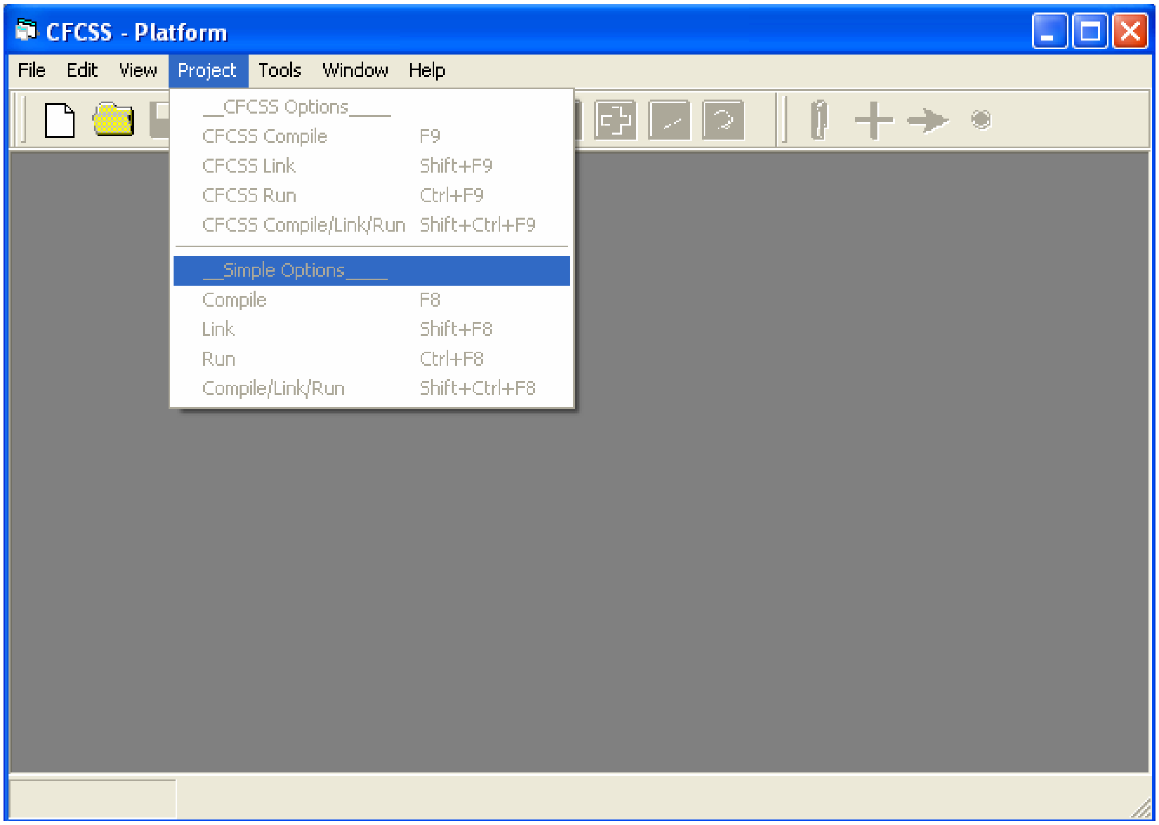


Figure 18: Project Menu

Compile

This option is used to directly compiler the program with the help of MASM. It traditionally compiles the code.

Compile and Run

This is also the traditional command used to directly compile and run the program immediately after compilation.

Build all

This command is used to build the current assembly code.

CFCSS Compile

This option is used to compile the code using CFCSS that is it applies the translator code and converts the existing code into CFCSS code that will be able to check for errors.

CFCSS Compile and Run

This option is used to compile the program using CFCSS and run directly after the compilation.

CFCSS Build All

This command is used to build the program with the CFFCSS code.

4.2.3 Output File

The output file is a log file that is achieved after the program has executed. The log file contains the whole path traveled by the program, checking signatures for each node and explaining whether the jump was valid or not.

The log file has three types of entries

1. In the first type it just prints the signature of the source node followed by the signature of the destination node telling whether the signatures matched. If the signatures are matched it gives **‘s+ d fine**’ as output.

2. In the second type, the signatures may also be checked with the RAS values. So the checking goes on one step further checking the RAS values along with the signature and the difference. If the signature matches then it gives the output as ‘**s + d + r fine**’

3. The third type of entry is when the signatures don’t match and the jump is illegal. This implies that the control flow error has occurred and the program

has terminated. It gives the output as a message along with the signatures '**not a legal jump**'.

4.2.4 Interaction Mechanism

The interaction mechanism consists of the visual basic platform interacting with the C code. The main thing responsible for the interaction here is the use of batch files which are used to run a number of C files synchronously through visual basic.

The procedure taken to build batch files is as follows.

Depending on the source file the currents code as placed at a proper location. Then the batch file starts its work. First of all it runs the three translator file one by one and generates the proper CFCSS translated code. Then this code is fed to assembler and then the assembler takes the command.

The work of assembler is to build the object file and then link it to have an executable files. This executable file is then calls from VB interface itself to run the code. This whole procedure is done as a batch process. This means with the help of VB only one batch file is made at runtime. Then the rest of work is done by this batch file itself. It have all the options of copying one file from one directory to another and then to compile all the code.

5. RESULTS

Here an example is provided to show the proper working of the project. Firstly a sample assembly code is taken. Then the intermediate files are displayed and finally the file with the additional CFCSS code is shown. The output of the program with a sample run is also presented.

5.1 The Assembly Code

A sample assembly code is presented on which the CFCSS algorithm will be tested.

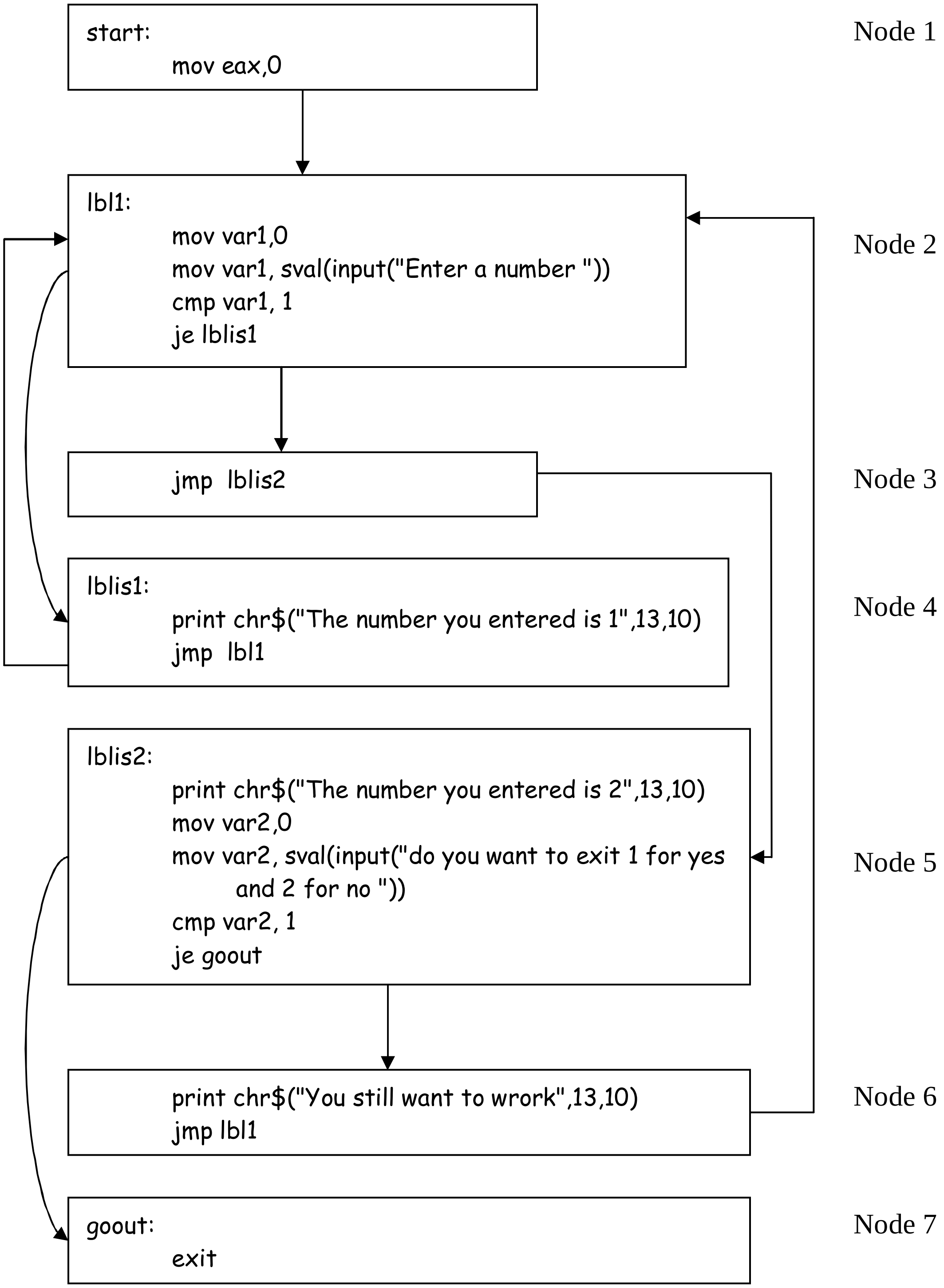
Assembly Code

```
.586
.model flat,stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\masm32.inc
include \masm32\macros\macros.asm

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\masm32.lib

.data

            var1      dd     ?
```

```
           var2     dd    ?
.code
start:

                mov eax,0
lbl1:   mov var1,0
    mov var1, sval(input("Enter a number "))

        cmp var1, 1          ; compare the variable to the immediate number 100
je lblis1            ; jump if var1 is equal to 100 to "equal"
    jmp  lblis2

lblis1:
                print chr$("The number you entered is 1",13,10)
                jmp lbl1

lblis2:
    print chr$("The number you entered is 2",13,10)
        mov var2,0
    mov var2, sval(input("do you want to exit 1 for yes and 2 for no "))
        cmp var2, 1
    je goout
        print chr$("You still want to wrork",13,10)
    jmp lbl1

goout:
    exit

end start
```

This file has all the essential code from the declarations to the end but the part of the program that is of interest is between ‘.code’ and ‘end start’.

We divide the code into different nodes and show the proper flow of the program.



```
start:
        mov eax,0

lbl1:
        mov var1,0
        mov var1, sval(input("Enter a number "))
        cmp var1, 1
        je lblis1

        jmp  lblis2

lblis1:
        print chr$("The number you entered is 1",13,10)
        jmp  lbl1

lblis2:
        print chr$("The number you entered is 2",13,10)
        mov var2,0
        mov var2, sval(input("do you want to exit 1 for yes
                and 2 for no "))
        cmp var2, 1
        je goout

        print chr$("You still want to wrork",13,10)
        jmp lbl1

goout:
        exit
```

Figure 19: The node structure and flow of the program

5.2 The Intermediate Files

When we compile the program through CFCSS the following tasks take place.

Firstly the CFCSS.C file is run which makes an intermediate file named ‘CFCSS3.txt’ giving the following output

CFCSS3.txt

```
7
start|1|20|22
lbl1|2|23|27
jmp|3|28|28
lblis1|4|32|34
lblis2|5|36|40
print|6|43|44
goout|7|46|54
```

This file simply identifies the number of nodes their name, their node number and their starting and ending line numbers.

Next the CFCSSG.c file is run which makes an intermediate file named ‘CFCSS4.txt’ having these values.

CFCSS4.txt

```
7
start|1|15280|20|22|0|0|
lbl1|2|15650|23|27|1682|2|29752,6442
jmp|3|18195|28|28|31281|0|
lblis1|4|20360|32|34|29354|0|
lblis2|5|21112|36|40|5483|0|
print|6|8858|43|44|28898|0|
goout|7|14972|46|46|26628|0|

0 1 0 0 0 0 0 0
0 0 1 1 0 0 0 0
```

```
0 0 0 0 1 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
```

This file gives as output a matrix which shows the complete flow of the graph. It does some specific work of assigning the signatures to the nodes, calculating the differences and the most specific task of assigning run time signatures.

The fan in of each node is calculated and the corresponding ras values are calculated along with the count of ras values.

Finally, CFCSSNF.c file is run which gives CFCSS6.txt as the output. This file contains the complete assembly code, along with the modifications i.e. all the additional code added to the file to check for signatures at runtime.

CFCSS6.txt

```
.586
.model flat,stdcall
option casemap:none

include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
include \masm32\include\masm32.inc
include \masm32\macros\macros.asm

includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
includelib \masm32\lib\masm32.lib


                cfcssrtfunc PROTO :dword,:dword,:dword,:dword

.data

                cfcsssig      dd     15280
                cfcssdiff      dd     0
                cfcssras       dd     0
                cfcsshndl      dd     ?
                cfcssrtf       db     "cfcss7.log"
                cfcsssizef     dd     ?
                cfcssbw        dd     ?
                cfcssras1        dd       0
                cfcssras2        dd       29752,6442
                cfcssras3        dd       0
                cfcssras4        dd       0
                cfcssras5        dd       0
                cfcssras6        dd       0
                cfcssras7        dd       0

           var1       dd     ?
           var2       dd     ?
.code
start:
        invoke CreateFile,addr cfcssrtf,GENERIC_WRITE,NULL,NULL,
CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL
        mov cfcsshndl, eax

                pusha
                pushf
                mov eax ,   15280
                mov ebx ,   0
                mov ecx ,   0
                mov edx ,   offset cfcssras1
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa


        mov eax,0
   lbl1:
                pusha
                pushf
                mov eax ,   15650
                mov ebx ,   1682
                mov ecx ,   2
                mov edx ,   offset cfcssras2
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa
                      mov var1,0
            mov var1, sval(input("Enter a number "))

            cmp var1, 1              ; compare the variable to the immediate number 100
            je lblis1            ; jump if var1 is equal to 100 to "equal"

                pusha
                pushf
                mov eax ,   18195
                mov ebx ,   31281
                mov ecx ,   0
                mov edx ,   offset cfcssras3
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa
                jmp  lblis2



   lblis1:
                pusha
                pushf
                mov eax ,   20360
                mov ebx ,   29354
                mov ecx ,   0
                mov edx ,   offset cfcssras4
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa

                 print chr$("The number you entered is 1",13,10)
            jmp lbl1

   lblis2:
                pusha
                pushf
                mov eax ,   21112
                mov ebx ,   5483
                mov ecx ,   0
                mov edx ,   offset cfcssras5
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa
                    print chr$("The number you entered is 2",13,10)
            mov var2,0
            mov var2, sval(input("do you want to exit 1 for yes and 2 for no "))
            cmp var2, 1
            je goout



                pusha
```

```
                pushf
                mov eax ,   8858
                mov ebx ,   28898
                mov ecx ,   0
                mov edx ,   offset cfcssras6
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa
                print chr$("You still want to wrork",13,10)
            jmp lbl1

    goout:
                pusha
                pushf
                mov eax ,   14972
                mov ebx ,   26628
                mov ecx ,   0
                mov edx ,   offset cfcssras7
                invoke cfcssrtfunc, eax, ebx, ecx, edx
                popf
                popa

             exit


          cfcssrtfunc proc sig:DWORD,diff:DWORD,rascount:dword,ras:dword
                                  local cfcsststr:dword
                                  local tempras:dword
                                  local tempcheck:dword
                                  mov cfcsststr,alloc(100)
                                  mov ecx,cfcsssig
                                  xor ecx,diff
                                  .if ecx==sig
                                        strcat
cfcsststr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$("   s+d  fine     "),chr$(13),chr$(10)
                                        mov cfcsssizef,len(cfcsststr)
                                        invoke WriteFile,cfcsshndl,cfcsststr,cfcsssizef,ADDR
cfcssbw,NULL
                                  .else
                                        mov tempras,ecx
                                        mov edx,0
                                        mov tempcheck,0
                                        .while edx < rascount
                                              mov ecx,tempras
                                              xor ecx,[cfcssras+edx*4]
                                              .if ecx==sig
                                                    mov tempcheck,1
                                                    .break;
```

```
                        .endif
                        inc edx
                    .endw
                    .if tempcheck==1
                        strcat
cfcsststr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$("   s+d+r  fine     "),chr$(13),chr$(10)
                        mov cfcsssizef,len(cfcsststr)
                        invoke
WriteFile,cfcsshndl,cfcsststr,cfcsssizef,ADDR cfcssbw,NULL
                    .else
                        strcat
cfcsststr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$("   not a legal jump
"),chr$(13),chr$(10)
                        mov cfcsssizef,len(cfcsststr)
                        invoke
WriteFile,cfcsshndl,cfcsststr,cfcsssizef,ADDR cfcssbw,NULL
                    .endif
                .endif
                mov eax,sig
                mov cfcsssig,eax
                mov eax,ras
                mov cfcssras,eax
                free cfcsststr
                ret
            cfcssrtfunc endp
end start
```

5.3 Log File

After the file is run with the additional code a log file 'CFCSS7.txt' is created to check whether the jumps were taken according to the signatures or not.

CFCSS7.txt

```
15280 15280   s+d  fine
15280 15650   s+d  fine
15650 20360   s+d  fine
20360 15650   s+d+r   fine
15650 18195   s+d  fine
18195  21112   s+d  fine
21112  14972   s+d  fine
```

RELATED WORKS

While special hardware for error checking is required in other signature monitoring techniques, CFCSS does not need the help of extra hardware for error detection; this is the advantage of CFCSS. The distinctive feature of the CFCSS over previous signature monitoring techniques is that CFCSS needs no dedicated hardware such as a watchdog processor for control flow checking because it is a pure software method. A watchdog task in multitasking environment also needs no extra hardware, but the advantage of the CFCSS over it is that CFCSS can be used even when the operating system does not support multitasking.

CFCSS uses an assigned signature technique similar to Structural Integrity Checking (SIC), but does not need to send *check-labels* to a watchdog processor since it checks the signatures using instructions. Block Signature Self-Checking (BSSC) is also an assigned signature technique that uses a subroutine to replace the watchdog processor. However, its drawback is that the code depends on the location of the code because the signature consists of an absolute address. The control flow checking scheme presented in is a pure software method but it constructs a database containing information about concurrent control flow checking, thus it may require significant memory overhead.

CONCLUSION

Merits

The CFCSS technique to check the control flow of the program is cheaper and faster than the traditional hardware which is being used to solve the problems of the control flow.

With the log being maintained we can actually check the flow and check where the jump went wrong and the code does not terminate abnormally.

The code goes to the basic commands currently just checking for the jumps and the labels.

Another advantage we add with the signatures and not just using labels is that we can match the constructs faster and more efficiently

Demerits

The main problem with this technique is that the code after the compilation becomes big and we have to call the checking function each time we find a node.

The constructs currently dealt with the system are simple and we can further enhance the system using the programming language constructs.

REFERENCES

[1] Lu, D. J., “Watchdog Processor and Structural Integrity Checking”, *IEEE Transactions on Computers*, vol. C-31, No. 7, pp. 681-685, July 1982.

[2] Yau, S. S. and Fu-Chung Chen, “An Approach to Concurrent Control Flow Checking,” *IEEE Trans. on Software Engineering*, Vol. SE-6, No. 2, March 1980

[3] Ersoz, A., D. M. Andrews, and E. J. McCluskey, “The Watchdog Task: Concurrent Error Detection Using Assertions,” Stanford University, Center for Reliable Computing, TR 85-8.

[4] Eifert, J. B. and J. P. Shen, “Processor Monitoring Using Asynchronous Signatured Instruction Streams,” *Digest of Papers, 14th Annual IntÕl Conf. on Fault-Tolerant Computing*, pp. 394-399, June 1984.

[5] Wilken, K., and J.P. Shen, “Concurrent Error Detection Using Signature Monitoring and Encryption: Low-Cost Concurrent-Detection of Processor Control Errors,” *Dependable Computing for Critical Applications, Springer-Verlag, A. Avizienis, J.C. Laprie (eds)*, Vol. 4, pp. 365-384, 1989.

[6] Wilken, K. and J. P. Shen, “Continuous Signature Monitoring: Low-Cost Concurrent- Detection of Processor Control Errors,” *IEEE Trans. on Computer Aided Design*, Vol. 9, No. 6, pp. 629-641, June 1990.

[7] Saxena, N. R., and E. J. McCluskey, “Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums,Ó *IEEE Trans. on Computers*, Vol. 39, No. 4, pp. 554-559, April 1990.

[8] J. Ohlsson and M Rimen, “Implicit Signature Checking,” *Digest of Papers, Twenty-Fifth International symposium on Fault-Tolerant Computing,* pp. 218-227, June 1995. 30

[9] Shirvani, P.P. and E.J.McCluskey, “Fault-Tolerant Systems in a Space Environment: The CRC ARGOS Project,Ó CRC-TR 98-2, Stanford University, Dec. 1998

[10] Lu, D. J., “Watchdog Processors and VLSI,Ó *Proceedings of the National Electronics conference*, Vol. 34, pp. 240-245, Chicago, Illinois, October 27-28, June 1980.

[11] Mahmood, Aamer and E. J. McCluskey, “Watchdog Processor: Error Coverage and Overhead,” *Digest, The Fifteenth Annual IntÕl Symposium on Fault-Tolerant Computing (FTCS-15),* pp. 214-219, Ann Arbor, Michigan, June 19-21, 1985.

[12] Hennessy, J. L. and D. A. Patterson*, Computer Architecture: A Quantitative Approach*, Second edition, 1996.

[13] C. H. Tung and C. W. McCarron, ÒConcurrent Control Flow Checking in Sequential and Parallel Program,Ó *Twenty-Fourth Asilomar Conference on Signals, Systems and Computers*, Maple Press, Vol. 2, pp. 851-855, Nov. 1990.

[14] Furtado, P., H. Madeira, ÒFault Injection Evaluation of Assigned Signatures in RISC Processors,Ó *Proc., Second European Dependable Computing Conference,* Taormina, Italy, pp. 55-72, October 1996.

APPENDIX A

THE 'C' CODE

CFCSS.c

```
#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
#define DELIMITER '|'
FILE *fp,*cmd,*ft,*fln;

void showfile();   // to print whole of the file
char *token(FILE *,char *);   // to get next token in a file and character which breaks it
void  shownode();
int checklabel(char *);
int checkjump(char *);


void main()
{
        int i;
        fp=fopen("cfcss1.txt","r");

        if(fp==NULL)
        {
                printf("file cant be opened:sort");
                getch();
                exit(0);
        }
        cmd=fopen("cfcss2.txt","r");
        if(cmd==NULL)
        {
                printf("file cant be opened:command");
                getch();
                exit(0);
        }
        ft=fopen("cfcss3.txt","w");
        if(ft==NULL)
        {
                printf("file cant be opened:Sample");
                getch();
                exit(0);
        }
        fputs("   \n",ft);

        clrscr();
        shownode();
        fclose(fp);
        fclose(cmd);
        fclose(ft);

}

void showfile()
{
        char str[30],ch;
        while(!feof(fp))
        {
                strcpy(str,token(fp,&ch));
                printf("%s",str);
                if (ch!=EOF)
                        printf("%c",ch);
                //getch();
        }
}
char *token(FILE *src,char *retch)
{
        char temp[30],temp1[30],ch;
        int i=0,j=0;
        while (1)
        {
                ch=fgetc(src);
                if (ch=='\n'||ch=='\t'||ch==' '||ch==':'||ch==';'||ch=='+'||ch=='-
'||ch=='['||ch==']'||ch==','||ch=='.'||ch==EOF)
                {
                        *retch=ch;
                        temp[i]='\0';
                        return (temp);
                }
                else
                {
                        temp[i]=ch;
                }
                i++;
        }
}
void shownode()
{
        //skipping all the code till .code prefix
        char temp[30],temp1[30],ch,node_no[5],ch1;
        int lastno,pline=0,lineno=1,start=0,chk1=0, chk2=0,bline=0,chk3=0,nodestart=0; //pline points to the
lastnon-empty line.
        int nodeno=0;
        rewind(fp);
        while(!feof(fp)) //searching till .code comes
        {
                strcpy(temp,token(fp,&ch));
                if (ch=='\n')
                {
                        lineno++;
                }
                else if (ch=='.')
                {
                        strcpy(temp,token(fp,&ch));
                        if(!strcmp(temp,"code"))
                        {
                                if (ch=='\n')
                                        lineno++;
```

```
            break;
        }
        pline=lineno;
        if (ch=='\n')
            lineno++;

    }
    if (ch==';')
    {
        while(ch!='\n')
            strcpy(temp,token(fp,&ch));
        lineno++;
    }
} //// end of .code search
// now finding nodes

while (!feof(fp))
{
    strcpy(temp,token(fp,&ch));
    if (temp[0]!='\0'&&chk1==1&&chk2==1)
    {
        chk1=0;
        chk2=0;
        nodeno++;
        fputs(temp,ft);
        fputc(DELIMITER,ft);
        itoa(nodeno,node_no,10);
        fputs(node_no,ft);
        fputc(DELIMITER,ft);
        nodestart=lineno;
        itoa(lineno,node_no,10);
        fputs(node_no,ft);
        fputc(DELIMITER,ft);
        pline=lineno;
    }
    if (ch==';')
    {
        while(ch!='\n')
            strcpy(temp,token(fp,&ch));
        if (chk1==1)
            chk2=1;
    }

    if(ch=='\n')
    {
        if (chk3==1)
            bline++;
        lineno++;
        chk3=1;

        if (chk1==1)
            chk2=1;
    }
    if(ch==':')
    {
        if (checklabel(temp))
        {
            if(start==1)
            {
                if((pline-1)<nodestart)
```

```
                        continue;
                    itoa(pline-1,node_no,10);
                    fputs(node_no,ft);
                    fputs("\n",ft);
                    start=0;
                }
                start=1;
                //code to insert into the sample file.
                nodeno++;
                fputs(temp,ft);
                fputc(DELIMITER,ft);
                itoa(nodeno,node_no,10);
                fputs(node_no,ft);
                fputc(DELIMITER,ft);
                itoa(lineno,node_no,10);
                nodestart=lineno;
                fputs(node_no,ft);
                fputc(DELIMITER,ft);
                //code ends
            }
            pline=lineno;
        }
        if (temp[0]!='\0'&& checkjump(temp) )
        {
            itoa(lineno-bline,node_no,10);
            fputs(node_no,ft);
            fputc('\n',ft);
            chk1=1;
        }
        if(temp[0]!='\0')
        {
            chk3=0;
            bline=0;
            pline=lineno;
        }

    }
    itoa(lineno-1,node_no,10);
    fputs(node_no,ft);
    rewind(ft);
    itoa(nodeno,node_no,10);
    fputs(node_no,ft);
    fclose(ft);

}

checklabel(char *str)
{
    if(!strcmp(str,"ss"))
        return(0);
    if(!strcmp(str,"cs"))
        return(0);
    if(!strcmp(str,"ds"))
        return(0);
    if(!strcmp(str,"es"))
        return(0);
    if(!strcmp(str,"fs"))
        return(0);
    if(!strcmp(str,"gs"))
        return(0);
```

```
		return(1);
}
checkjump(char *str)
{
		char str1[30],ch;
		int chk=0;
		rewind(cmd);
//		puts("hi");
		while(!feof(cmd))
		{
				strcpy(str1,token(cmd,&ch));
				if(!strcmpi(str,str1))
						chk=1;
		}
		if(chk==1)
				return 1;
		else
				return 0;
}
```

CFCSSG.c

```
#include "stdio.h"
#include "conio.h"
#include "math.h"
#include "stdlib.h"
#include "alloc.h"
#include "string.h"
#include "ctype.h"
#include "time.h"
#define DELIMITER '|'
#define MAX_NODES 200

void getNewSignature();             //to get random signatures
int checkSignature(int signToCheck); //to check whether the random signatures donot match previous signatures

int signatures[MAX_NODES];
int usedSignature=0;

FILE *fp,*cmd,*ft,*op,*ftab;

void showfile();   // to print whole of the file
char * token(FILE *,char *);   // to get next token in a file and character which breaks it
int checkjump(char *);
void graph();
void getftom(int );
char* gotoline(int );

struct sampl
{
            char nodename[MAX_NODES];
            int nodeno;
            int sign;
            int strt;
            int last;
            double diff;
            int ras[MAX_NODES];
}link[20];

void main()
{
            int i;
            fp=fopen("cfcss1.txt","r");
            if(fp==NULL)
            {
                        printf("file cant be opened:sort");
                        getch();
                        exit(0);
            }

            ftab=fopen("cfcss4.txt","w+");

            if(ftab==NULL)
            {
                        printf("file cant be opened:table");
                        getch();
                        exit(0);
```

```
        }

        cmd=fopen("cfcss2.txt","r");
        if(cmd==NULL)
        {
                printf("file cant be opened:command");
                getch();
                exit(0);
        }
        ft=fopen("cfcss3.txt","r");
        if(ft==NULL)
        {
                printf("file cant be opened:Sample");
                getch();
                exit(0);
        }
        op=fopen("cfcss5.txt","w+");
        if(op==NULL)
        {
                printf("file cant be opened:Sample1");
                getch();
                exit(0);
        }

        clrscr();
        graph();

        fclose(op);
        fclose(fp);
        fclose(cmd);
        fclose(ftab);

}
void graph()
{
        char tabl[30][30],tabl1[30][30],jmp1,ch,str1[512];
        int lineno,node,i,j,k,l,snode,lnode;

        strcpy(str1,token(ft,&ch));
        node=atoi(str1);

        while(ch!='\n')
                strcpy(str1,token(ft,&ch));

        getftom(node);
        fclose(ft);


        for(i=0;i<node+1;i++)
        for(j=0;j<node+1;j++)
        {
                if(j==i+1)
                {
                        tabl[i][j]='1';
                        link[j].diff=link[i].sign^link[j].sign;
                }
                else
                        tabl[i][j]='0';
        }
```

```
for(i=0;i<node;i++)
{
        rewind(fp);
        fputs("\n",ftab);
        itoa(link[i].last-1,str1,10);
        fputs(str1,ftab);                       // to input line no in the file.
        fputc('|',ftab);

        for(k=0;k<link[i].last-1;k++)
        {
                ch='a';
                while(ch!='\n')
                        strcpy(str1,token(fp,&ch));
        }

        strcpy(str1,token(fp,&ch));
        while(str1[0]=='\0')
                strcpy(str1,token(fp,&ch));

        fputs(str1,ftab);
        if(!strcmpi(str1,"EXIT"))
                tabl[i][node]='1';

        if(checkjump(str1) )
        {

                if(!strcmpi(str1,"JMP"))
                        tabl[i][i+1]='0';

                strcpy(str1,token(fp,&ch));

                while(str1[0]=='\0')
                        strcpy(str1,token(fp,&ch));

                for(j=0;j<node;j++)
                 if(!strcmpi(link[j].nodename,str1))
                 {
                        tabl[i][j]='1';
                        //link[i].ras=link[i].sign^link[j-1].sign;
                 }
        }
}
for (i=0;i<node;i++)
        for (j=0;j<node;j++)
                tabl1[i][j]=tabl[j][i];

//getting RAS values
for (i=0;i<node;i++)
{
        k=1;
        link[i].ras[0]=0;

        for (j=0;j<node;j++)
        {
                if(tabl1[i][j]=='1')
                {

                        link[j].diff=link[i].sign^link[j].sign;
                        l=j;
                        j++;
```

```
            break;
        }
    }
    for(j;j<node;j++)
    {
        if (tabl1[i][j]=='1')
        {
            link[i].ras[0]+=1;
            link[i].ras[k++]=link[l].sign^link[j].sign;
        }

    }

}

fputc('\n',ftab);
for(i=0;i<node+1;i++)
{
    for(j=0;j<node+1;j++)
    {
        fputc(tabl[i][j],ftab);
        if(j!=node)
        fputc(' ',ftab);
    }
    fputc('\n',ftab);
}

//To write into the Sample1 file.
itoa(node,str1,10);
fputs(str1,op);
fputc('\n',op);
for(i=0;i<node;i++)
{
    fputs(link[i].nodename,op);
    fputc(DELIMITER,op);

    itoa(link[i].nodeno,str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);

    itoa(link[i].sign,str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);

    itoa(link[i].strt,str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);

    itoa(link[i].last,str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);

    itoa(link[i].diff,str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);

    itoa(link[i].ras[0],str1,10);
    fputs(str1,op);
    fputc(DELIMITER,op);
```

```
                for (j=1;j<=link[i].ras[0];j++)
                {
                        itoa(link[i].ras[j],str1,10);
                        fputs(str1,op);
                        if(j!=link[i].ras[0])
                                fputs(",",op);
                }
                fputs("\n",op);
        }

        //to write table into the sample1...
        //required code BHUPESH
        for(i=0;i<node+1;i++)
        {
                for(j=0;j<node+1;j++)
                {
                        fputc(tabl[i][j],op);
                        fputc(' ',op);
                }
                fputc('\n',op);
        }
}

void getftom(int node)
{
        float lineno,i;
        char ch,str[30];
        randomize();
        for(i=0;i<node;i++)
        {
                strcpy(str,token(ft,&ch));
                strcpy(link[i].nodename,str);//TO put label

                strcpy(str,token(ft,&ch));
                lineno=atoi(str);
                link[i].nodeno=lineno;//to put node no

                getNewSignature();
                link[i].sign=signatures[i];

                strcpy(str,token(ft,&ch));
                lineno=atoi(str);
                link[i].strt=lineno;//to put start line no.

                strcpy(str,token(ft,&ch));
                lineno=atoi(str);
                link[i].last=lineno;//to put end line no.
                link[i].diff=link[i].ras[0]=0;
        }
        link[i-1].last=link[i-1].strt;
}

char *token(FILE *src,char *retch)
{
        char temp[30],temp1[30],ch;
        int i=0;
        while (1)
        {
                ch=fgetc(src);
                if (ch=='\n'||ch==' '||ch==':'||ch==';'||ch=='+'||ch=='-
'||ch=='['||ch==']'||ch==','||ch=='.'||ch==EOF||ch=='|'||ch=='\t')
                {
                        *retch=ch;
                        temp[i]='\0';
                        return (temp);
                }
                else
                        temp[i]=ch;
                i++;
        }
}

checklabel(char str[])
{
        if((!strcmpi(str,"ss")) ||(!strcmpi(str,"SS")))
                return(0);
        if((!strcmpi(str,"cs")) ||(!strcmpi(str,"CS")))
                return(0);
        if((!strcmpi(str,"ds")) ||(!strcmpi(str,"DS")))
                return(0);
        if((!strcmpi(str,"es")) ||(!strcmpi(str,"ES")))
                return(0);
        if((!strcmpi(str,"fs")) ||(!strcmpi(str,"FS")))
                return(0);
        if((!strcmpi(str,"gs")) ||(!strcmpi(str,"GS")))
                return(0);

        return(1);
}
checkjump(char str[])
{
        char str1[30],ch;
        int chk=0;
        rewind(cmd);
        while(!feof(cmd))
        {
                strcpy(str1,token(cmd,&ch));
                if(!strcmpi(str,str1) )
                        chk=1;
        }
        if(chk==1)
                return 1;
        else
                return 0;
}

void getNewSignature()
{
        int newSign=rand();
        while(1)
        {
          if(checkSignature(newSign))
          {
                        break;
          }
          newSign=rand();
        }
        signatures[usedSignature++]=newSign;
}
```

```
int checkSignature(int signToCheck)
{
        int i=0;
        for(i=0;i<usedSignature;i++)
        {
                if(signatures[i]==signToCheck)
                {
                        return 0;
                }
        }
        return 1;
}
```

CFCSSNF.c

```
#include "stdio.h"
#include "conio.h"
#include "stdlib.h"
#include "string.h"
#include "ctype.h"
#define DELIMITER '|'
#define MAX_NODES 200
FILE *fp,*ft,*nf,*cmd;

void showfile();   // to print whole of the file
char * token(FILE *,char *);   // to get next token in a file and character which breaks it
int checklabel(char *);
int checkjump(char *);
void getftom(int a);
void graph();
void newfile();
void printNode(int );



struct sampl
{
            char nodename[20];
            int nodeno;
            int sign;
            int strt;
            int last;
            int diff;
            int ras[MAX_NODES];
}link[20];



void main()
{
            clrscr();
            fp=fopen("cfcss1.txt","r");

            if(fp==NULL)
            {
                        printf("file cant be opened:sort");
                        getch();
                        exit(0);
            }
            ft=fopen("cfcss5.txt","r");
            if(ft==NULL)
            {
                        printf("file cant be opened:Sample1");
                        getch();
                        exit(0);
            }

            nf=fopen("cfcss6.txt","w");
            if(nf==NULL)
            {
                        printf("file cant be opened:table");
```

```
                getch();
                exit(0);
        }
        cmd=fopen("cfcss2.txt","r");
        if(cmd==NULL)
        {
                printf("file cant be opened:command ");
                getch();
                exit(0);
        }

        newfile();

        fclose(fp);
//      fclose(ft);
        fclose(nf);
        fclose(cmd);
}

char *token(FILE *src,char *retch)
{
        char temp[30],temp1[30],ch;
        int i=0;
        while (1)
        {
                ch=fgetc(src);
                if (ch=='\n'||ch==' '||ch==':'||ch==';'||ch=='+'||ch=='-
'||ch=='['||ch==']'||ch==','||ch=='.'||ch==EOF||ch==DELIMITER)
                {
                        *retch=ch;
                        temp[i]='\0';
                        return (temp);
                }
                else
                        temp[i]=ch;

                i++;
        }
}
checklabel(char str[])
{
        if((!strcmpi(str,"ss")) ||(!strcmpi(str,"SS")))
                return(0);
        if((!strcmpi(str,"cs")) ||(!strcmpi(str,"CS")))
                return(0);
        if((!strcmpi(str,"ds")) ||(!strcmpi(str,"DS")))
                return(0);
        if((!strcmpi(str,"es")) ||(!strcmpi(str,"ES")))
                return(0);
        if((!strcmpi(str,"fs")) ||(!strcmpi(str,"FS")))
                return(0);
        if((!strcmpi(str,"gs")) ||(!strcmpi(str,"GS")))
                return(0);

        return(1);
}
checkjump(char str[])
{
        char str1[30],ch;
        int chk=0;
```

```
		rewind(cmd);
		while(!feof(cmd))
		{
			strcpy(str1,token(cmd,&ch));
			if(!strcmpi(str,str1))
				chk=1;
		}
		if(chk==1)
			return 1;
		else
			return 0;
	}

	void getftom(int node)
	{
		int lineno,i,j;
		char ch,str[30];
		for(i=0;i<node;i++)
		{
			strcpy(str,token(ft,&ch));
			strcpy(link[i].nodename,str);//TO put label

			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].nodeno=lineno;//to put node no

			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].sign=lineno;//to put signatue no

			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].strt=lineno;//to put start line no.


			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].last=lineno;//to put end line no.

			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].diff=lineno;//to put end line no.

			strcpy(str,token(ft,&ch));
			lineno=atoi(str);
			link[i].ras[0]=lineno; //to put end line no.

			if(link[i].ras[0]!=0)
			{
			for(j=0;j<link[i].ras[0];j++)
				{
					strcpy(str,token(ft,&ch));
					lineno=atoi(str);
					link[i].ras[j+1]=lineno; //to put end line no
				}
			}
			else
			{
			  strcpy(str,token(ft,&ch));
			}
```

```
        }
}


void newfile()
{
        char str[30],str1[30],ch,str2[30],str3[20],temp[30],temp1[10];
        int i,j,chk=0,node,sign,strt,last,lineno=1,diff,ras,nodeno;

        //to extract sample.txt in a structure.
        strcpy(str,token(ft,&ch));
        node=atoi(str);
        while(ch!='\n')
                strcpy(str,token(ft,&ch));

        getftom(node);
        printNode(node);
        fclose(ft);


        while(!feof(fp)) //searching till .data comes
        {
                strcpy(temp,token(fp,&ch));
                if (ch=='\n')
                {
                        lineno++;
                }
                else if (ch=='.')
                {
                        fputs(temp,nf);
                        strcpy(temp,token(fp,&ch));
                        if(!strcmpi(temp,"data"))
                        {
                                fputs("\n\t\tcfcssrtfunc PROTO :dword,:dword,:dword,:dword\n",nf);
                                fputs("\n.data\n",nf);
                                fputs("\n\t\tcfcsssig        dd      ",nf);
                                itoa(link[1].sign,temp1,10);
                                fputs(temp1,nf);
                                fputs("\n\t\tcfcssdiff       dd      0",nf);
                                fputs("\n\t\tcfcssras        dd      0",nf);
                                fputs("\n\t\tcfcsshndl       dd      ?",nf);
                                fputs("\n\t\tcfcssrtf        db      \"cfcss7.log\"",nf);
                                fputs("\n\t\tcfcsssizef      dd      ?",nf);
                                fputs("\n\t\tcfcssbw         dd      ?\n",nf);

                                for (i=1;i<=node;i++)
                                {
                                        fputs("\t\tcfcssras",nf);
                                        itoa(i,temp1,10);
                                        fputs(temp1,nf);
                                        fputs("\tdd\t",nf);
                                        if(link[i-1].ras[0]==0)
                                        {
                                                fputs("0\n",nf);
                                        }
                                        else
                                        {
                                                for(j=1;j<=link[i-1].ras[0];j++)
                                                {
                                                        itoa(link[i-1].ras[j],temp1,10);
```

```
                            fputs(temp1,nf);
                            if(j!=link[i-1].ras[0])
                            {
                                fputs(",",nf);
                            }
                            else
                            {
                                fputs("\n",nf);
                            }
                        }
                    }
                }

                if (ch=='\n')
                    lineno++;
                break;
            }
            else
            {
                fputc('.',nf);
            }
            if (ch=='\n')
                lineno++;

        }

        fputs(temp,nf);
        fputc(ch,nf);
        if (ch==';')
        {
            while(ch!='\n')
            {
                strcpy(temp,token(fp,&ch));
                fputs(temp,nf);
                fputc(ch,nf);
            }
            lineno++;
        }

    } //// end of .data search
    // now finding nodes

    //upto .CODE part
    strt=link[0].strt;
    while(lineno<strt)
    {
        strcpy(temp,token(fp,&ch));
        if(ch=='\n')
            lineno++;
        //if(temp[0]!='\0')
        fputs(temp,nf);
        fputc(ch,nf);
    }


//printf("lno=%d",lineno);
    for(i=0;i<node;i++)
    {
        strcpy(str,link[i].nodename);
        sign=link[i].sign;
```

```
strt=link[i].strt;
last=link[i].last;
diff=link[i].diff;
ras=link[i].ras[0];
nodeno=link[i].nodeno;

chk=0;
if(!feof(fp))
{   //go to the strt line no.
        ch='a';
        while(lineno<strt)
        {
                strcpy(str1,token(fp,&ch));
                fputs(str1,nf);
                fputc(ch,nf);
                if(ch=='\n')
                        lineno++;
        }
        strcpy(str1,token(fp,&ch));

        while(str1[0]=='\0')
        {
                fputs(str1,nf);
                fputc(ch,nf);
                strcpy(str1,token(fp,&ch));
        }
        if(ch!=':')
        {
                fputs("\n\t\tpusha",nf);
                fputs("\n\t\tpushf",nf);
                fputs("\n\t\tmov eax ,   ",nf);
                itoa(sign,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov ebx ,   ",nf);
                itoa(diff,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov ecx ,   ",nf);
                itoa(ras,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov edx ,   offset cfcssras",nf);
                itoa(nodeno,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tinvoke cfcssrtfunc, eax, ebx, ecx, edx",nf);
                fputs("\n\t\tpopf",nf);
                fputs("\n\t\tpopa\n\t\t",nf);
                if(atoi(str1)==ras)
                        fputs(link[i].nodename,nf);
                else
                        fputs(str1,nf);
                fputc(ch,nf);
        }
        else
        {
                fputs(str1,nf);

                if(!strcmpi(str1,"START"))
                {
                        if(str1==NULL)
                        {           fputs(link[i].nodename,nf);
                                    fputc(':',nf);
                                    if(!strcmpi(link[i].nodename,"START"))
                                    {
                                        fputs("\n\tinvoke                    CreateFile,addr
cfcssrtf,GENERIC_WRITE,NULL,NULL, CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL",nf);
                                        fputs("\n\tmov cfcsshndl, eax\n",nf);
                                    }
                        }
                        else
                                    fputc(ch,nf);
                        if(!strcmpi(str1,"START"))
                        {
                                    fputs("\n\tinvoke                         CreateFile,addr
cfcssrtf,GENERIC_WRITE,NULL,NULL, CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL",nf);
                                    fputs("\n\tmov cfcsshndl, eax\n",nf);
                        }
                }
                else
                        fputc(ch,nf);
                fputs("\n\t\tpusha",nf);
                fputs("\n\t\tpushf",nf);
                fputs("\n\t\tmov eax ,   ",nf);
                itoa(sign,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov ebx ,   ",nf);
                itoa(diff,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov ecx ,   ",nf);
                itoa(ras,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tmov edx ,   offset cfcssras",nf);
                itoa(nodeno,temp,10);
                fputs(temp,nf);
                fputs("\n\t\tinvoke cfcssrtfunc, eax, ebx, ecx, edx",nf);
                fputs("\n\t\tpopf",nf);
                fputs("\n\t\tpopa\n\t\t",nf);
            }

         }//end if.
    }//end of for(node)
    while(!feof(fp))
    {
        strcpy(str,token(fp,&ch));
        strcpy(str3,str);
        if (!strcmpi(str,"END"))
        {

            fputs("\n\tcfcssrtfunc     proc     sig:DWORD,diff:DWORD,rascount:dword,ras:dword
",nf);
            fputs("\n\t                              local cfcssstr:dword ",nf);
            fputs("\n\t                              local tempras:dword ",nf);
            fputs("\n\t                              local tempcheck:dword ",nf);
            fputs("\n\t                              mov cfcssstr,alloc(100) ",nf);
            fputs("\n\t                              mov ecx,cfcsssig ",nf);
            fputs("\n\t                              xor ecx,diff ",nf);
            fputs("\n\t                              .if ecx==sig ",nf);
            fputs("\n\t                                        strcat
cfcssstr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$(\"   s+d  fine     \"),chr$(13),chr$(10) ",nf);
            fputs("\n\t                                        mov cfcsssizef,len(cfcssstr) ",nf);
            fputs("\n\t                                        invoke
WriteFile,cfcsshndl,cfcssstr,cfcsssizef,ADDR cfcssbw,NULL ",nf);
```

```
            fputs("\n\t     .else ",nf);
            fputs("\n\t         mov tempras,ecx ",nf);
            fputs("\n\t         mov edx,0 ",nf);
            fputs("\n\t         mov tempcheck,0 ",nf);
            fputs("\n\t         .while edx < rascount ",nf);
            fputs("\n\t             mov ecx,tempras ",nf);
            fputs("\n\t             xor     ecx,[cfcssras+edx*4]
",nf);
            fputs("\n\t             .if ecx==sig ",nf);
            fputs("\n\t                 mov     tempcheck,1
",nf);
            fputs("\n\t                 .break; ",nf);
            fputs("\n\t             .endif ",nf);
            fputs("\n\t             inc edx ",nf);
            fputs("\n\t         .endw ",nf);
            fputs("\n\t         .if tempcheck==1 ",nf);
            fputs("\n\t             strcat
cfcsststr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$(\"   s+d+r  fine      \"),chr$(13),chr$(10)  ",nf);
            fputs("\n\t             mov
cfcsssizef,len(cfcsststr)  ",nf);
            fputs("\n\t             invoke
WriteFile,cfcsshndl,cfcsststr,cfcsssizef,ADDR cfcssbw,NULL  ",nf);
            fputs("\n\t         .else  ",nf);
            fputs("\n\t             strcat
cfcsststr,ustr$(cfcsssig),chr$(9),ustr$(sig),chr$(9),chr$(\"   not a legal jump      \"),chr$(13),chr$(10)  ",nf);
            fputs("\n\t             mov
cfcsssizef,len(cfcsststr)  ",nf);
            fputs("\n\t             invoke
WriteFile,cfcsshndl,cfcsststr,cfcsssizef,ADDR cfcssbw,NULL  ",nf);
            fputs("\n\t         .endif  ",nf);
            fputs("\n\t     .endif  ",nf);
            fputs("\n\t     mov eax,sig  ",nf);
            fputs("\n\t     mov cfcsssig,eax  ",nf);
            fputs("\n\t     mov eax,ras  ",nf);
            fputs("\n\t     mov cfcssras,eax  ",nf);
            fputs("\n\t     free cfcsststr  ",nf);
            fputs("\n\t     ret  ",nf);
            fputs("\n\t cfcssrtfunc endp  \n",nf);
        }
        fputs(str3,nf);
            if(ch!=EOF)
                fputc(ch,nf);

    }
}//end new file().

void printNode(int node)
{
    int j,i;
    //strcpy(str,token(ft,&ch));
    //node=atoi(str);

    for(i=0;i<node;i++)
    {
        printf("%s\t",link[i].nodename);
        printf("%d\t",link[i].nodeno);
        printf("%d\t",link[i].sign);
        printf("%d\t",link[i].strt);
        printf("%d\t",link[i].last);
        printf("%d\t",link[i].diff);
				printf("%d\t",link[i].ras[0]);
				for(j=0;j<link[i].ras[0];j++)
				{
						printf("%d\t",link[i].ras[j+1]);
				}

				printf("\n");
		}
		getch();
}
```

APPENDIX B

INSTRUCTION SET

Instruction	Description
JA rel8	Jump short if above (CF=0 and ZF=0)
JAE rel8	Jump short if above or equal (CF=0)
JB rel8	Jump short if below (CF=1)
JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
JC rel8	Jump short if carry (CF=1)
JCXZ rel8	Jump short if CX register is 0
JECXZ rel8	Jump short if ECX register is 0
JE rel8	Jump short if equal (ZF=1)
JG rel8	Jump short if greater (ZF=0 and SF=OF)
JGE rel8	Jump short if greater or equal (SF=OF)
JL rel8	Jump short if less (SF<>OF)
JLE rel8	Jump short if less or equal (ZF=1 or SF<>OF)
JNA rel8	Jump short if not above (CF=1 or ZF=1)
JNAE rel8	Jump short if not above or equal (CF=1)
JNB rel8	Jump short if not below (CF=0)
JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
JNC rel8	Jump short if not carry (CF=0)
JNE rel8	Jump short if not equal (ZF=0)
JNG rel8	Jump short if not greater (ZF=1 or SF<>OF)
JNGE rel8	Jump short if not greater or equal (SF<>OF)
JNL rel8	Jump short if not less (SF=OF)
JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
JNO rel8	Jump short if not overflow (OF=0)
JNP rel8	Jump short if not parity (PF=0)
JNS rel8	Jump short if not sign (SF=0)
JNZ rel8	Jump short if not zero (ZF=0)

JO rel8	Jump short if overflow (OF=1)
JP rel8	Jump short if parity (PF=1)
JPE rel8	Jump short if parity even (PF=1)
JPO rel8	Jump short if parity odd (PF=0)
JS rel8	Jump short if sign (SF=1)
JZ rel8	Jump short if zero (ZF = 1)
JA rel16/32	Jump near if above (CF=0 and ZF=0)
JAE rel16/32	Jump near if above or equal (CF=0)
JB rel16/32	Jump near if below (CF=1)
JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
JC rel16/32	Jump near if carry (CF=1)
JE rel16/32	Jump near if equal (ZF=1)
JZ rel16/32	Jump near if 0 (ZF=1)
JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
JGE rel16/32	Jump near if greater or equal (SF=OF)
JL rel16/32	Jump near if less (SF<>OF)
JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
JNA rel16/32	Jump near if not above (CF=1 or ZF=1)
JNAE rel16/32	Jump near if not above or equal (CF=1)
JNB rel16/32	Jump near if not below (CF=0)
JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
JNC rel16/32	Jump near if not carry (CF=0)
JNE rel16/32	Jump near if not equal (ZF=0)
JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
JNL rel16/32	Jump near if not less (SF=OF)
JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=OF)
JNO rel16/32	Jump near if not overflow (OF=0)
JNP rel16/32	Jump near if not parity (PF=0)
JNS rel16/32	Jump near if not sign (SF=0)
JNZ rel16/32	Jump near if not zero (ZF=0)
JO rel16/32	Jump near if overflow (OF=1)

JP rel16/32	Jump near if parity (PF=1)
JPE rel16/32	Jump near if parity even (PF=1)
JPO rel16/32	Jump near if parity odd (PF=0)
JS rel16/32	Jump near if sign (SF=1)
JZ rel16/32	Jump near if 0 (ZF=1)
LOOP	Decrement count; jump short if count ≠ 0
LOOPZ	Decrement count; jump short if count ≠ 0 and ZF=1
LOOPE	Decrement count; jump short if count ≠ 0 and ZF=1
LOOPNZ	Decrement count; jump short if count ≠ 0 and ZF=0
LOOPNE	Decrement count; jump short if count ≠ 0 and ZF=0
JMP rel16/32	Unconditional Jump, to a label