Aim:-

A Register Allocation algorithm that translates the given code into one with a fixed number

of registers.

Theory:-

In compiler optimization, register allocation is the process of assigning a large number of target program variables onto a small number of CPU registers.

Register allocation can happen over a basic block (*local register allocation*), over a whole function/procedure (*global register allocation*), or across function boundaries traversed via call-graph (*interprocedural register allocation*). When done per function/procedure the calling convention may require insertion of save/restore around each call-site.

In many programming languages, the programmer may use any number of variables. The computer can quickly read and write registers in the CPU, so the computer program runs faster when more variables can be in the CPU's registers.Also, sometimes code accessing registers is more compact, so the code is smaller, and can be fetched faster if it uses registers rather than memory. However, the number of registers is limited. Therefore, when the compiler is translating code to machine-language, it must decide how to allocate variables to the limited number of registers in the CPU.

Not all variables are in use (or "live") at the same time, so, over the lifetime of a program, a given register may be used to hold different variables. However, two variables in use at the *same* time cannot be assigned to the same register without corrupting one of the variables. If there are not enough registers to hold all the variables, some variables may be moved to and from RAM. This process is called "spilling" the registers.

### Components of register allocation:-

* Move insertion

This action consists in increasing the number of move instructions between registers, i.e. make a variable live in different registers during its lifetime, instead of one. This occurs in the split live range approach.

* Spilling

This action consists of storing a variable into memory instead of registers.

* Assignment

This action consists of assigning a register to a variable.

* Coalescing

This action consists of limiting the number of moves between registers, thus limiting the total number of instructions. For instance, by identifying a variable live across different methods, and storing it into one register during its whole lifetime.

Graph-coloring allocation is the predominant approach to solve register allocation. It was first proposed by Chaitin et al. In this approach, nodes in the graph represent live ranges (variables, temporaries, virtual/symbolic registers) that are candidates for register allocation. Edges connect live ranges that interfere , i.e., live ranges that are simultaneously live at at least one program point. Register allocation then reduces to the graph coloring problem in which colors (registers) are assigned to the nodes such that two nodes connected by an edge do not receive the same color.

Using liveness analysis, an interference graph can be built. The interference graph which is an undirected graph where the nodes are the program's variables is used to model which variables cannot be allocated to the same register.

Code:-

1. Code Text file input

We are using a .txt format file as an input code. The language to be executed is written line by line in the text file, and consists of mainly assignment commands, with while loops, for loops, and if-else conditional assignments

2. Scanning the file

We are using python as the programming language to perform the High-Level Synthesis. So, we scan the text file using python commands, and using string library commands to eliminate and analyze all assignments, we make a list of sets of variables for each line of the program, where the first element of each of the tuples correspond to the variable to which assignment is made, and rest are variables used in assignment. This ordered list of tuples for variables is given as the output.

3. Computing Live Variable

We iteratively analyze the live variables for a particular line of the code. With respect to the current line, we check the variables in the ordered set for the line, and keep that variable live which is to be used in assignment in some of the further code, if there is no assignment made to this variable prior to this usage.

4. Construction of RIG

Based on the set of tuples of live variables, we first compute the list of variables used in the netlist. Then we are constructing a graph, given as an Adjacency matrix, forwhich edges are set between nodes i and j, if the variables corresponding to node iand j are present together in a tuple in live variable list.

5. Graph Colouring Block

This block takes the Adjacency Matrix as an input. We have a set restriction for the number of registers, so we cannot assign more than that number of colours. We first initialize the colours of all the variables as 0. Then we start assigning a colour to a node in order, and check if it violates the colouring criteria. If not, the colour is assigned to the node. If yes, we check for the next available colour. If no possible colour satisfies the condition, we let the colour remain as 0. Now, if we have obtained a proper colouring sequence for all the nodes, we give this to output. Otherwise, we first compute the node with maximum degree, and allot this variable to directly store and load from the memory. Then we modify our adjacency matrix and recompute the colouring algorithm. This is performed recursively to remove all the nodes, without which we can provide a satisfactory colour sequence for the graph. This is known as spilling.

6. Display Register Allocation

The colour sequence corresponding to each variable, obtained after previous block, is our output. However, we have displayed this coloured graph in a graphical image form. This is our final output, which gives us the registers allocated corresponding to the given vaariables.

Input:-

a = 1

b = 2

c = a + b

d = c + a

d = d + 1

b = d\*2

d = c - d

return d

Output:-

Variable list for given code : ['f', 'b', 'c', 'a', 'd', 'e']

Adjacency Matrix of RIG : [[0, 1, 1, 1, 1, 1], [1, 0, 1, 0, 0, 0], [1, 1, 0, 1, 1, 1], [1, 0, 1, 0, 0, 0], [1, 0, 1, 0, 0, 1], [1, 0, 1, 0, 1, 0]]

Following are the assigned colours and independent nodes:

Colour of f node is 1

Colour of b node is 2

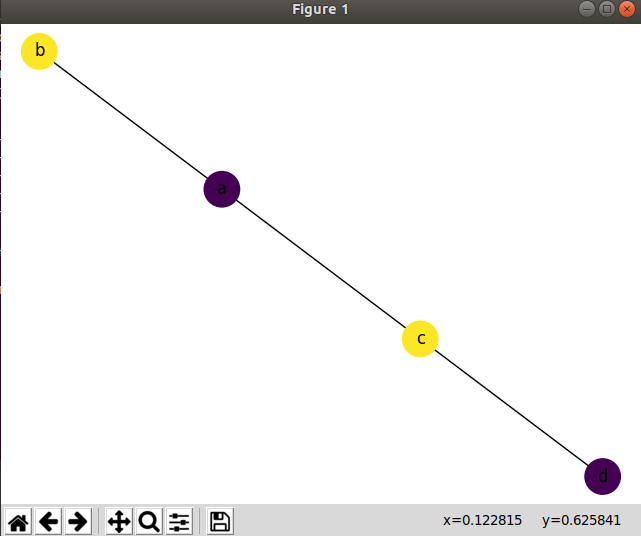
Colour of c node is 3

Colour of a node is 2

Colour of d node is 2

Colour of e node is 4

Total number of registers used : 4



Conclusion:-

The graph colouring algorithm used is of high efficiency, but better colouring techniques do exist. However, for most of the cases with lesser number of variables, the graph colouring obtained is minimized. Register allocation in compilers can be implemented using graph coloring algorithm and is NP complete.