Application of Graph Coloring to Create Efficient Register Allocation in Program

Erick Wijaya 13515057[1]
Informatics Undergraduate Program
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Ganesha Street 10 Bandung 40132, Indonesia
[1]13515057@std.stei.itb.ac.id
wijaya.erick52@gmail.com

***Abstract*—Graph coloring is a method to color each vertex in a graph so that no two adjacent vertices have same color. The smallest number of color needed to do graph coloring for any arbitrary graph is called chromatic number. Graph coloring is a powerful tool for compilers to optimize the code so that register allocation can be done effectively. The process consists of creating a control flow graph, computing alive variables at each point, constructing interference graph, and coloring graph.**

***Keywords*—chromatic number, interference graph, graph coloring, register allocation.**

I. Introduction

Register allocation is one of the most important optimizations a compiler performs and is becoming increasingly important as the gap between processor speed and memory access time widens. As the memory access time is much slower than processor, optimizations are needed to avoid using too many memory by using registers in processor. However, the compiler should not simply declare all existing variables in program to available registers because some variables may be “dead” or temporary and not used further after executing specific code segment. That means the variable can be recycled to minimalize usage of memory or registers so more variables can be stored in registers and less variables may be stored in memory. To recycle the “dead” variable, the compiler checks every points of program segment to see which variables are “alive” and will be used further in the program, and which variables are temporary and will “die” in the middle of program. In this case graph coloring rise to be the solution. By using graph coloring method, the compiler can find whether the variables are “dead” or “alive” at specific points of program, then create an interference graph that represents variables and their relationship. After creating the graph, the graph will be colored based on the chromatic number of that particular graph, and the chromatic number represents registers needed to store variables.

II. Basic Graph Theory

2.1. Graph Theory

Graphs are discrete structures consisting of vertices and edges that connect these vertices. There are different kinds of graphs, depending on whether edges have directions, whether multiple edges can connect the same pair of vertices, and whether loops are allowed. Graph models are used in many application in any discipline.

2.1.1. Definition

A graph G = (*V*, *E*) consists of *V*, a nonempty set of *vertices* (or nodes) and *E*, a set of *edges*. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints. With this definition, we know that a graph can at least contains one vertex and zero edge.

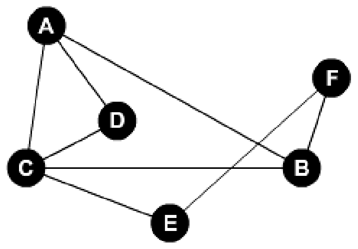


Figure 1. A graph that consists of 6 vertices and 8 edges

2.1.2 Terminology

In graph theory, we will use several terms to describe properties of graph, its behavior, or anything related to graph. Here is a list of important terms that will be used to describe a graph or its properties.

1. Null Graph
   A graph having no edges is called a Null Graph.
2. Trivial Graph
   A graph with only one vertex is called a Trivial Graph.
3. Adjacent
   Two vertices in a graph are said to be adjacent if both vertices are connected by an edge mutually.

4. Incident
   Edge e is incident if it connects two vertices.
5. Degree
   For a non-directed graph, degree of a vertex is the sum of edges that are incident with that vertex.

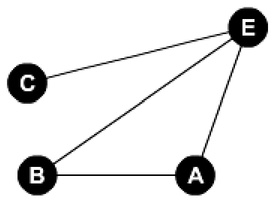


Figure 2. The degree of vertex E is 3

6. Path
   Path with length *n* from initial vertex v_o to final vertex v_n in a graph is an array of pair of edge and vertex that, if the path is exist, will contain v_o, e_1, v_1, e_2, … , v_{n-1}, e_n, v_n so that $e_1 = (v_o,v_1)$, $e_2 = (v_1,v_2)$, … , $e_n = (v_{n-1},v_n)$.
7. Subgraph
   For a graph $G = (V, E)$, G1 = (V_1, E_1) is a subgraph from graph G if $V_1 \subseteq V$ and $E_1 \subseteq E$.
8. Weight
   A weight is a value that represents attributes of edges from weighted-graph. The weight may represent distance, time, cost, or anything. Weighted-graphs can be useful to describe transportation problems and are usually supported by Dijkstra or Prim's Algorithm.

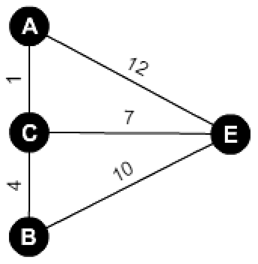


Figure 3. Each edges has a value in weighted-graph

2.1.3. Types of Graph

Graphs are classified to some categories based on their properties. There are various types of graphs depending upon the number of vertices, number of edges, interconnectivity, and their overall structure.

Based on the edge types, we can differ graph into three types listed below:

1. Simple Graph
   A graph with no loops and no parallel edges is called a simple graph.
2. Multigraph
   Graphs that may have multiple edges connecting the same vertices are called multigraphs.
3. Pseudograph
   Pseudograph is a graph which can include loops as well as multiple edges connecting the same pair of vertices.

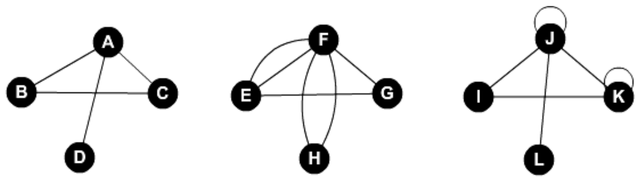


Figure 4. Simple graph (left), multigraph (middle), and pseudograph (right).

Graphs can also be classified based on their edges characteristic, specifically whether the edges have direction or not. We can differ the graph into these types:

1. Non-Directed Graph
   A non-directed graph contains edges, however the edges do not have direction.
2. Directed Graph
   On the other hand, each edges in a directed graph has a direction.

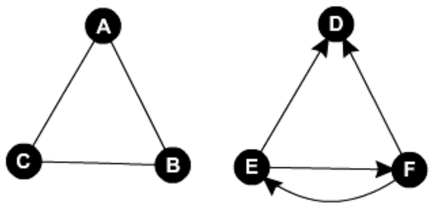


Figure 5. Each edges in directed graph has its direction (right), unlike non-directed graph which do not have direction in the edges (left).

Some simple graphs have their own characteristics which distinguish them from the others. These graphs are called *specific graphs*. There are several specific graphs which are mainly used in computer science:

1. Regular Graph
   A graph is said to be regular, if all its vertices have the same degree. In a graph, if the degree of each vertex is *k*, then the graph is called a *k-regular graph*.
2. Complete Graph
   A simple graph with n mutual vertices is called a complete graph and it is denoted by K_n. In the graph, a vertex should have edges with all other vertices, then it called a complete graph.

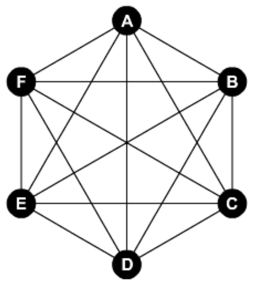


Figure 6. A complete graph denoted by K_6

3. Cyclic Graph

A graph with at least one cycle is called a cyclic graph. Figure 6 is also an example of cyclic graph because it has cycles, path E – F – A – E is the example of cycle in graph.

2.2. Graph Coloring

Graph coloring is nothing but a simple way of labelling graph components such as vertices, edges, and regions under some constraints. In a graph, no two adjacent vertices, adjacent edges, or adjacent regions are colored with minimum number of colors. This number is called the chromatic number and the graph is called a properly colored graph. In this case we will only talk about vertex coloring.

2.2.1. Chromatic Number

The chromatic number of a graph *G* is the smallest number of colors needed to color the vertices of *G* so that no two adjacent vertices share the same color, i.e., the smallest value of *k* possible to obtain a *k*-coloring [5].

The chromatic number of a graph G is most commonly denoted $\chi(G)$.

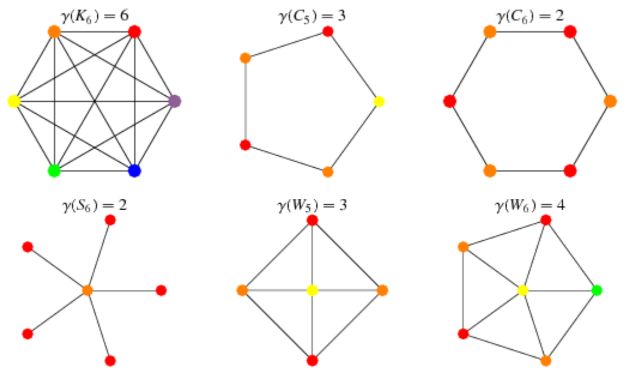


Figure 7. Some graphs with their chromatic numbers. Source:http://mathworld.wolfram.com/ChromaticNumber.html

2.2.2. Coloring Method

Until now, there is no known fast algorithm to find an optimal coloring for an arbitrary graph (NP-hard). However, in this paper the author only cover coloring smaller graphs so coloring can be done manually. To do graph coloring, we can use some lemmas which come in handy when trying to show that a graph has a certain chromatic number, these lemmas are:

- **Lemma 1.** A graph *G* has chromatic number $\chi(G)$ = 2 if and only if it is *bipartite*.
- **Lemma 2.** If *H* is a subgraph of *G* and *G* is *k*-colorable, then so is *H*.
- **Lemma 3.** If *H* is a subgraph of *G* then $\chi(H) \leq \chi(G)$.

III. Assembly and Registers

Assembly language is a low-level programming language for a computer or other programmable device specific to a particular computer architecture in contrast to most high-level programming languages, which are usually portable across multiple systems. Assembly language is converted into executable machine code by a utility program referred to as an assembler like NASM and MASM.

```
plus:
    pushl %ebp
    movl %esp,%ebp
    movl 8(%ebp),%eax
    movl 12(%ebp),%ecx
    sall $2,%ecx
    leal 0(,%eax,8),%edx
    subl %eax,%edx
    leal (%eax,%eax,4),%eax
    movl mat2(%ecx,%eax,4),%eax
    addl mat1(%ecx,%edx,4),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Figure 8. An example of assembly code

Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon. However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus and into the memory storage unit and getting the data through the same channel.

To speed up the processor operations, the processor includes some internal memory storage locations, called registers. The registers store data elements for processing without having to access the memory. A limited number of registers are built into the processor chip. The general registers are further divided into the following groups:

- Data registers.
- Pointer registers.
- Index registers.

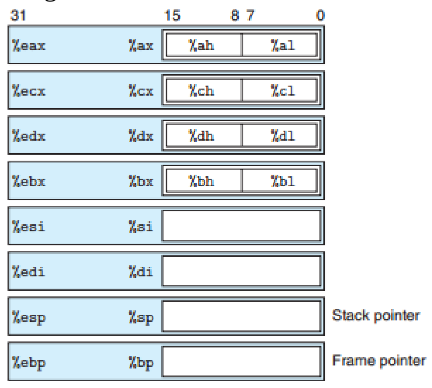


Figure 9. IA32 integer registers. These registers can be accessed as either 8 bit (%al or %ah), 16 bit (%ax), and 32 bit (%eax)
Source: Computer Systems - A Programmer's Perspective.

General registers are the registers used most of the time. Most of the instructions perform on these registers. They all can be broken down into 16 and 8 bit registers.

- 32 bits : %eax, %edx, %ecx, %ebx, %esi, %edi, %esp, %ebp
- 16 bits : %ax, %dx, %cx, %bx, %si, %di, %sp, %bp
- 8 bits : %ah %al %bh %bl %ch %cl %dh %dl

Data Registers are registers that are used mainly for arithmetic, logical, and other operations. In 32 bit operations, the data registers are %eax, %edx, %ecx, and %ebx.

- %eax is the primary accumulator; it is used in input/output and most arithmetic instructions. It is also used mainly to store return value in function calls.
- %ebx is known as the base register, as it could be used in indexed addressing.
- %ecx is known as the count register, %ecx registers store the loop count in iterative operations.
- %edx is known as the data register. It is also used in input/output operations. It is also used with AX register along with DX for multiply and divide operations involving large values.

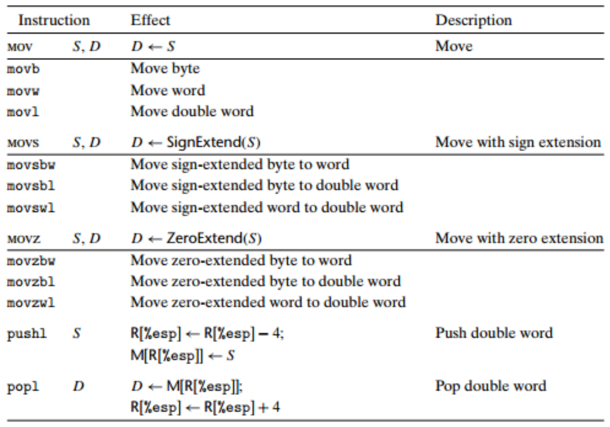
Pointer Registers are %esp, and %ebp registers and corresponding 16-bit right portions %sp and %bp. There are two main categories of pointer registers:

- Stack Pointer (%esp)
  The 16-bit SP register provides the offset value within the program stack. %esp refers to be current position of data or address within the program stack.
- Base Pointer (%ebp)
  Base pointer register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with %edi and %esi as base register for special addressing.

The 32-bit index registers, %esi and %edi, and their 16-bit rightmost portions. %si and %di, are used for indexed addressing and sometimes used in subtraction and addition. There are two sets of index pointers:

- Source Index (%esi)
  It is used as source index for string operations.
- Destination Index (%edi)
  It is used as destination index for string operations.

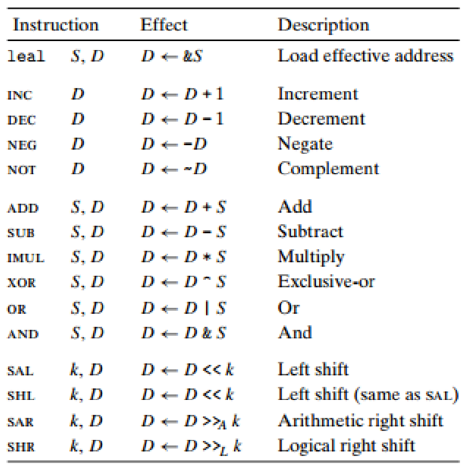
Among the most heavily used instructions are those that copy data from one location to another. The generality of the operand notation allows a simple data movement instruction to perform what in many machines would require a number of instructions [2]. Figure 9 lists the important data movement instructions. As can be seen, we group the many different instructions into instruction classes, where the instructions in a class perform the same operation, but with different operand sizes. For example, the mov class consists of four instructions: movb, movw, movl, and movq. All three of these instructions perform the same operation; they differ only in that they operate on data of size 1, 2, 4, and 8 bytes, respectively.



Instruction		Effect	Description
MOV	S, D	$D \leftarrow S$	Move
movb		Move byte	
movw		Move word	
movl		Move double word	
MOVS	S, D	$D \leftarrow \text{SignExtend}(S)$	Move with sign extension
movsbw		Move sign-extended byte to word	
movsbl		Move sign-extended byte to double word	
movswl		Move sign-extended word to double word	
MOVZ	S, D	$D \leftarrow \text{ZeroExtend}(S)$	Move with zero extension
movzbw		Move zero-extended byte to word	
movzbl		Move zero-extended byte to double word	
movzwl		Move zero-extended word to double word	
pushl	S	R[%esp] ← R[%esp] − 4; M[R[%esp]] ← S	Push double word
popl	D	D ← M[R[%esp]]; R[%esp] ← R[%esp] + 4	Pop double word

Figure 10. Some important data movement instructions in assembly language
Source: Computer Systems - A Programmer's Perspective.

Figure 11 shows some of the integer and logic operations. Most of the operations are given as instruction classes, as they can have different variants with different operand sizes. (Only leal has no other size variants.) For example, the instruction class add consists of several addition instructions: addb, addw, and addl, adding bytes, words, and double words, respectively. Indeed, each of the instruction classes shown has instructions for operating on byte, word, and double-word data. The operations are divided into four groups: load effective address, unary, binary, and shifts [2]. Binary operations have two operands, while unary operations have one operand.



Instruction		Effect	Description
leal	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \,\hat{}\, S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \,\&\, S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

Figure 11. Integer arithmetic operations
Source: Computer Systems - A Programmer's Perspective.

IV. Implementation of Graph Coloring in Register Allocation

Register allocation is one of the most important optimizations a compiler performs and is becoming increasingly important as the gap between processor speed and memory access time widens. If there are k registers, register allocators attempt to solve the NP-complete problem of finding a k-coloring of a graph. If not all the variables can be colored with a register assignment, some variables are spilled to memory and the process is repeated.

Consider this program segment with six variables:

```
a ← c + d
e ← a + b
f ← e - 1
```

with assuming that variable *a* and *e* die after use while variable *f*, *c*, *d*, and *b* will be need in the other part of the program. In this code segment, we can analyze two main points:

- Variable *a* can be reused after *e* ← *a* + *b*
- It also applies to variable *e*

That means we can allocate all the variables *a*, *e*, and *f* to one register (for example, %ecx). We do not need to allocate three different registers for variables *a*, *e*, and *f*. The program segment will be somewhat like this:

```
%ecx ← %ebx + %eax
%ecx ← %ecx + %edx
%ecx ← %ecx - 1
```

From this example, we notice that arbitrary variables t_1 and t_2 can share same register if at any point in the program at most one of t_1 or t_2 is alive. To allocate registers effectively, compilers compute alive variables for each program segment, construct an interference graph (IG), and color that graph. The register allocation will be based on graph colors. Below is the process to optimize code so register allocations can be done effectively. Consider this pseudocode segment below, we will create a control flow graph, compute alive variables for each point, construct an interference graph, and finally do graph coloring.

```
repeat
  a ← b + c
  d ← - a
  e ← d + f
  if (condition1) then
    f ← 2 * e
  else
    b ← d + e
    e ← e - 1
  endif
  b ← f + c
until (condition2)
```

Figure 12. A pseudocode segment, assume that **variable b is alive** after the pseudocode segment is executed
Source:https://www.cs.clemson.edu/course/cpsc827/material/Optimization/Register%20Allocation%20Example.pdf (with some modifications)

1.Create a control flow graph

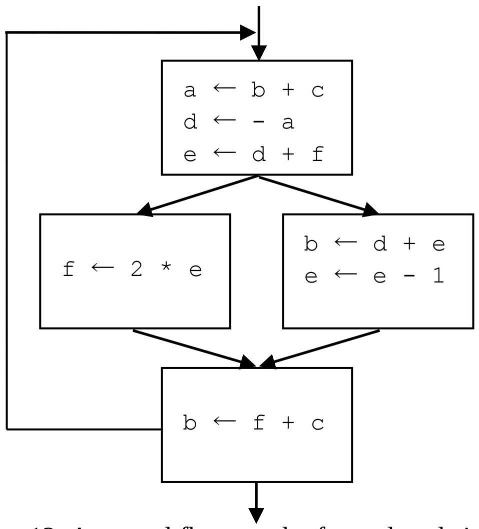


Figure 13. A control flow graph of pseudocode in fig. 12

2.Compute alive variables for each point

The next step in our allocation approach is to determine which variables are alive at each point in the program segment. This is call live range analysis. Initially we will assume that only b is alive on the exit from the program segment. The graph below shows which variables are alive at each point in the program segment.

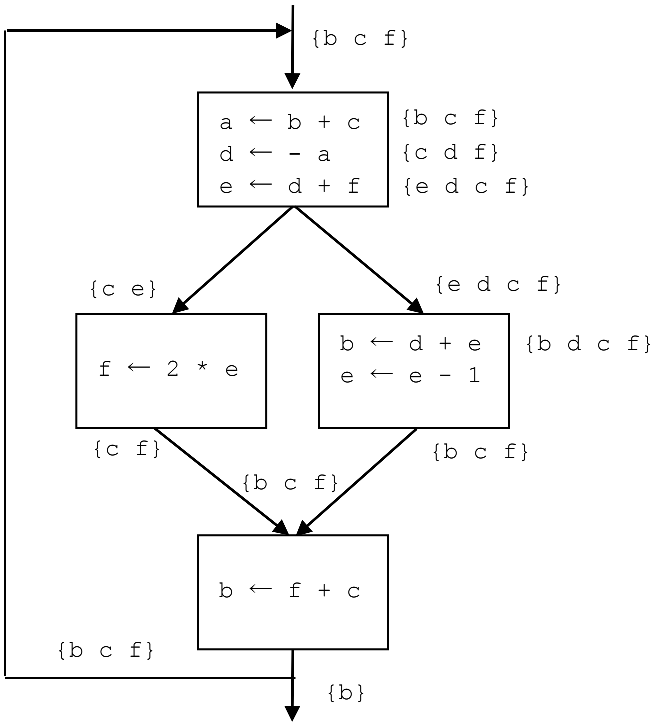


Figure 14. Alive variables from each points

3.Construct an interference graph

Considering figure 15, the vertices represent the variable (A for variable *a*, C for variable *c*, D for variable *d*, and so on). The edges represent that adjacent vertices cannot be in the same register

(variable *f* and *b* cannot be in the same register, however variable *a* and *e* can be in the same register).
To sum up the points:
- Vertices : variables
- Edges : cannot be in the same register

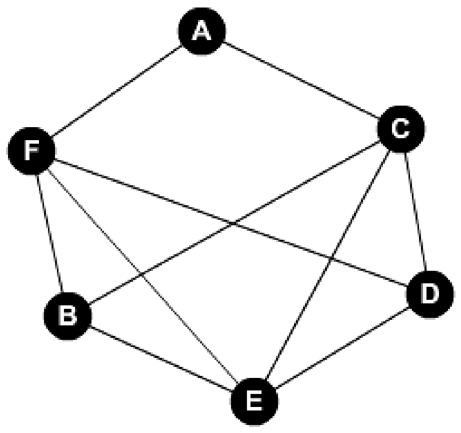


Figure 15. Interference graph *IG* from graph in figure 14.

4.Graph coloring

We then try to color the vertices, and found out that the interference graph *IG* has $\chi(IG) = 4$. Vertex A and B are colored red, C and F colored blue, E colored green, and D colored yellow.

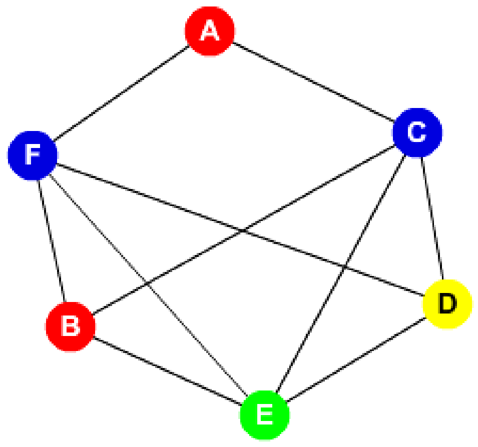


Figure 16. Colored interference graph *IG*.

Hence, we see that if we assign *a* and *b* to register R_1, *f* and *c* to R_2, *d* to R_3, and *e* to R_4, we can keep those registers in the same registers over the lifetime of the program segment. This assumes that there are sufficient registers for the intermediate results. When that assumption is not correct, then we will need to spill registers. That topic is not covered here.

V. CONCLUSION

In conclusion, compilers optimize register allocation by graph coloring so that register allocation can be done effectively. A code that uses many temporary variables can be optimized so that those variables can be stored in fewer registers. Graph coloring is a powerful register allocation scheme, however in the real practice, optimizing register allocation require more than graph coloring. Graph coloring with interference graph is NP-hard and for given *k* of registers, coloring may not be exist. There are some advanced solutions, such as heuristic and linear scan allocation, but these solutions will not be discussed in this paper.

VI. ACKNOWLEDGMENT

The first and foremost thanks from the author is to the God for giving the author inspiration and time to be able to write and finish this paper successfully. Special acknowledgment belongs to Rinaldi Munir, discrete mathematic lecturer in Bandung Institute of Technology, to be tender and nice in teaching, as well as giving the author task to write this paper so that the author can practice to write a good paper and to learn about computer science deeper. Last but not least, other thanks will the author give to the author's parents as they continuously support the author to undergo a college life in Bandung Institute of Technology.

REFERENCES

[1] Rosen, Kenneth H. *Discrete Mathematics and Its Applications*. New York: McGraw-Hill. 2012.
[2] Bryant, Randal. *Computer Systems - A Programmer's Perspective*. Massachusetts: Prentice Hall. 2003.
[3] Munir, Rinaldi. *Diktat Kuliah IF2120 Matematika Diskrit*. STEI, ITB. 2006
[4] http://web.cecs.pdx.edu/~mperkows/temp/register-allocation.pdf (last accessed in December 8th 2016 on 23.36 WIB)
[5] https://www.eecis.udel.edu/~cavazos/cisc672/lectures/Lecture-22.pdf (last accessed in December 8th 2016 on 22.42 WIB)
[6] http://www.cs.cmu.edu/~dkoes/research/graphraTR.pdf (last accessed in December 8th 2016 on 23.01 WIB)
[7] http://mathworld.wolfram.com/ChromaticNumber.html (last accessed in December 6th 2016 on 21.22 WIB)
[8] https://www.tutorialspoint.com/graph_theory/ (last accessed in December 6th 2016 on 18.53 WIB)

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Desember 2016

[illegible]

Erick Wijaya - 13515057