Assignment 6

Problem Statement: A Register allocation algorithm that translates the given code into one with a fixed number of registers.

Software Requirements: Lex,Yacc,GCC

Hardware Requirements: 4 GB RAM,500 GB HDD

Theory:
Register Allocation: 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 while function/procedure(global register allocation) or across function boundaries traversed via call graph(inerprocedural register allocation).

Principle: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 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.

In three address code there are unlimited no. of variables.
There are two important activities done while using registers:
1.Register allocation:During register allocation select appropriate set of variables that will reside in registers.
2.Register assignment:During regiter assignment pick up the specific register in which corresponding variable will reside.

Obtaining the optimal(minimum) assignment of registers to variable is difficult.Certain machine require register pairs such as even odd numbered register for some operands and results.
For eg:
Consider three address code:
t1=a+b
t1=t1/c
t1=t1/d
The efficient machine code sequence will be
mov a,R0;
add b,R0
mul c,R0
div d,R0
mov R0,t1

Various strategies used in register allocation and assignment:

1.Global register allocation:

Generating the code the registers are used to hold the value for the duration of single block.

All the live variables are stored at the end of each block.

For the variables that are used consistently we can allocate specific set of registers.

Hence allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

Some strategies:

The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.

Another strategy is to assign some fixed number of global registers to hold the most active value in each inner loop.

The registers not already allocated may be used to hold values local to one block.

In certain language like C or Bliss programmer can do the register allocation by using register declaration.

Usage count:

The usage count is the count for the use of some variable x in some register used in any basic block.

The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.

The approximate formula for usage count for the loop L in some basic block B can be given as,

$\sum$ (use(x , B)+2*live(x , B)) Block B in L

Where use (x, B) is number of times x used in block B prior to any definition of x and live (x, B)=1 is live on exit from B; otherwise live(x)=0.

To evaluate for x=a we observe that a is live on exit from B1 and is assigned a value there, but is not live on exit from B2, B3, or B4.

$\sum$ 2*live(a , B)=2.Also, use(a,B1)=0,since a is defined in B1 B in L before any use. Also, use(a,B2)=use(a,B3)=1and use(a,B4)=0. Thus, $\sum$ use(a , B)=2. Hence the value of(9.4) for x=a is 4. B in L

That is, four of cost can be saved by selecting a for one of the global registers.

The values of(9.4) for b , c , d , e and f are 6,3,6,4, and 4, respectively.

Register assignment for outer loop

Having assigned registers and generated code for inner loops, we may apply the same idea to progressively loops.

If an outer loop L1, contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1- L2.

However, if name x is allocated a register in loop L1 but not L2, we must store x on entrance to L2 and load x if we leave L2, and enter a block of L1-L2.

Similarly, if we choose to allocate x a register in L2, but not L1 must load x on entrance to L2 and store x on exit from L2.We leave as an exercise the derivation of a criterion for selecting names to be allocated registers in an outer loop L, given that choices have already been made for all loops nested within L.

Register allocation by graph coloring

A register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (Spilled)into a memory location in order to free up register.

Graph coloring is a simple systematic technique for allocating registers and managing register spills.

In this method, two passes are used:

In first , target-machine instruction are selected as though there were as infinite number of symbolic registers; in effect, names used un the intermediate code become names of registers and the three-address statements become machinelanguage statements.

In the second pass, for each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.

An attempt is made to color the register-interference graph using k colors, where k is the number of assignable registers.

Conclusion:

Hence we studied and implemented register allocation algorithm that translates given code into one with a fixed no. Of registers.

Code

assign6.l

```
%{
#include"stdio.h"
#include"y.tab.h"
%}
%%
[a-z][a-zA-Z0-9]*|[0-9]+ { strcpy(yylval.vname,yytext); return NAME; }
[ |\t] ;
.|\n { return yytext[0]; }
%%
```

assign6.y

```
%{
#include<stdio.h>
#include<ctype.h>
#include<string.h>
FILE *fpOut;
%}
%union
{
char vname[10];
int val;
}
```

```
%left '+' '-'
%left '*' '/'
%token <vname> NAME
%type <vname> expression
%%
input : line '\n' input
| '\n' input
| ;
line : NAME '=' expression { fprintf(fpOut,"MOV %s, AX\n",$1); }
;
expression: NAME '+' NAME
{
fprintf(fpOut,"MOV AX, %s\n",$1);
fprintf(fpOut,"ADD AX, %s\n",$3);
}
| NAME '-' NAME
{
fprintf(fpOut,"MOV AX, %s\n",$1);
fprintf(fpOut,"SUB AX, %s\n",$3);
}
| NAME '*' NAME
{
fprintf(fpOut,"MOV AX, %s\n",$1);
fprintf(fpOut,"MUL AX, %s\n",$3);
}
| NAME '/' NAME
{
```

```
fprintf(fpOut,"MOV AX, %s\n",$1);
fprintf(fpOut,"DIV AX, %s\n",$3);
}| NAME
{
fprintf(fpOut,"MOV AX, %s\n",$1);
strcpy($$, $1);
}
;
%%
FILE *yyin;
main()
{
FILE *fpInput;
fpInput = fopen("input1.c","r");
if(fpInput=='\0')
{
printf("file read error");
//exit(0);
}
fpOut = fopen("output.c","w");
if(fpOut=='\0')
{
printf("file creation error");
//exit(0);
}
yyin = fpInput;
yyparse();
```

```
fcloseall();
}
yyerror(char *msg)
{
printf("%s",msg);
}
```

Output:

```
(base) dell@dell-Inspiron-15-3567:~/Downloads/Compiler/Assignment6$ lex assign6.l
(base) dell@dell-Inspiron-15-3567:~/Downloads/Compiler/Assignment6$ yacc -d assign6.y
(base) dell@dell-Inspiron-15-3567:~/Downloads/Compiler/Assignment6$ gcc y.tab.c lex.yy.c
-ll -ly -lm
(base) dell@dell-Inspiron-15-3567:~/Downloads/Compiler/Assignment6$ ./a.out
```

input1.c:

```
t1 = c * d
t2 = b +t1
t3 = e /f
t4 = t2 -t3
a = t4
```

output.c:

```
MOV AX, c
MUL AX, d
MOV t1, AX
```

```
MOV AX, b
ADD AX, t1
MOV t2, AX
MOV AX, e
DIV AX, f
MOV t3, AX
MOV AX, t2
SUB AX, t3
MOV t4, AX
MOV AX, t4
MOV a, AX
```