中国矿业大学计算机学院

2019 级本科生课程设计报告

课程名称 系统软件开发实践

报告时间 2022.4.14

学生姓名 王杰永

学　　号 03190886

专　　业 计算机科学与技术

任课教师 张博

成绩考核

编号	课程教学目标	占比	得分
1	**目标 1：**针对编译器中词法分析器软件要求，能够分析系统需求，并采用 FLEX 脚本语言描述单词结构。	15%	
2	**目标 2：**针对编译器中语法分析器软件要求，能够分析系统需求，并采用 Bison 脚本语言描述语法结构。	15%	
3	**目标 3：**针对计算器需求描述，采用 Flex/Bison 设计实现高级解释器，进行系统设计，形成结构化设计方案。	30%	
4	**目标 4：**针对编译器软件前端与后端的需求描述，采用软件工程进行系统分析、设计和实现，形成工程方案。	30%	
5	**目标 5：**培养独立解决问题的能力，理解并遵守计算机职业道德和规范，具有良好的法律意识、社会公德和社会责任感。	10%	
总成绩			
指导教师		评阅日期	

目录

目标代码生成

1 实验目的

本实验是系统软件课程设计最后一次实验，其任务是在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将 C 自己源代码翻译为 MIPS32 指令序列（可以包含伪指令），并在 SPIM Simulator 或翻译为 x86 指令序列并在 x86 Simulator 上运行。

2 实验内容

1) 确定目标机的体系结构，选择一款模拟器进行安装学习。学习与具体体系结构相关的指令选择、寄存器选择以及存储器分配管理技术。

2) 编写目标代码的汇编生成程序可以将在此之前已经得到的中间代码映射为相应的汇编代码。

3) 实现错误处理程序，包括词法错误、语法错误、编译警告和运行异常。

4) 生成可执行文件，最终获得一个 C 自己的编译器，包含基本的前端和后端功能。

5) 实现链接器，生成可运行程序。

3 x86 体系及模拟器学习

X86 架构（The X86 architecture）是微处理器执行的计算机语言指令集，指 intel 通用计算机系列的标准编号缩写，也标识一套通用的计算机指令集合。

由于之前学习的《微机原理与接口技术》课程是建立在 x86 指令系统上授课的，因此本项目选择 X86 的汇编指令集作为目标代码生成的目标。

3.1 EMU8086 模拟器的安装与使用

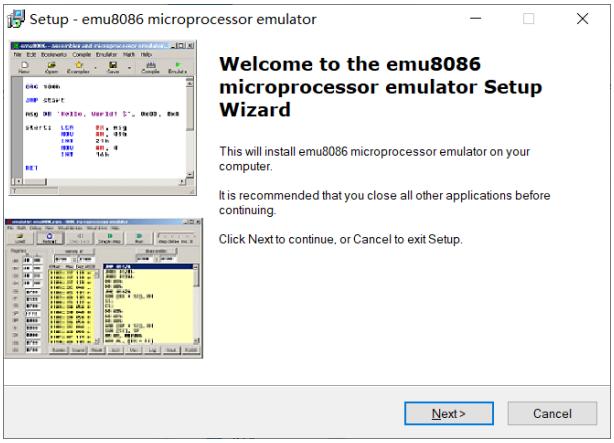


图 1 EMU8086 模拟器的安装

首先，将资料中的 EMU8086 压缩包解压，执行其中可执行文件，安装 EMU8086 模拟器。

EMU8086 是一个结合了原始编辑器、组译器、反组译器、具除错功能的软件模拟工具（虚拟 PC），可以在 EMU8086 中进行 x86 汇编语言的编译和运行。

EMU8086 的整体界面如图 2 所示。左上侧工具栏中的工具为 EMU8086 的主要功能。

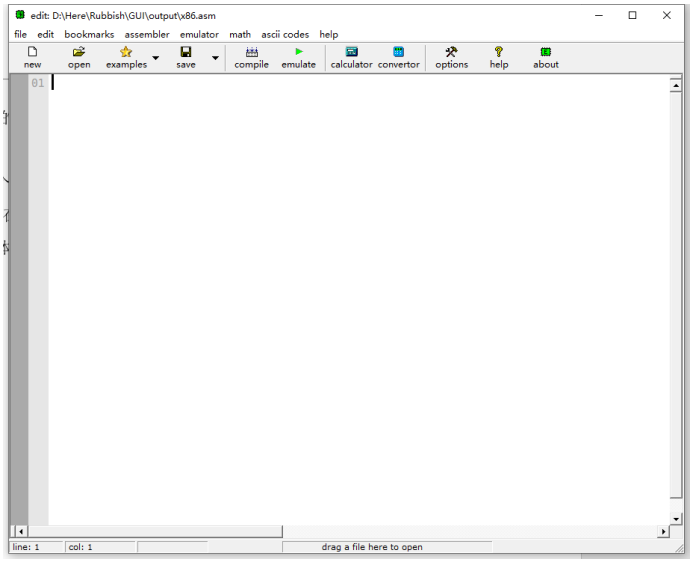


图 2 EMU8086 整体界面

在界面中间的代码区域，写下测试代码。通过模拟器编译，运行结果如图 3 所示。可以看到，在左侧的寄存器区域标明了 AH、AL 寄存器最后的值。

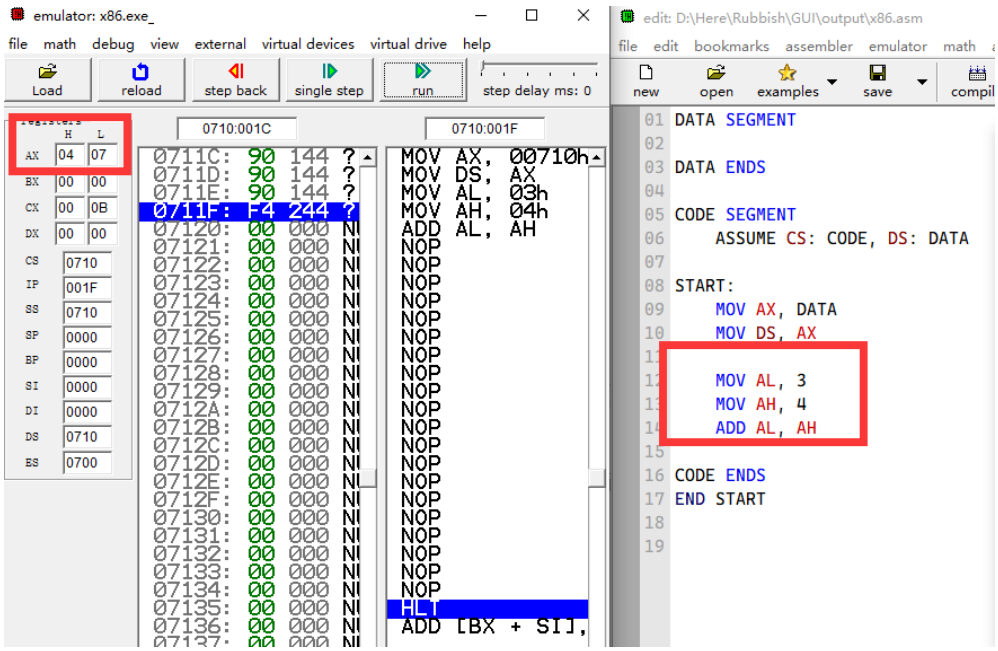


图 3 运行结果

4 中间代码生成

中间代码是一种易于翻译成目标程序的源程序的等效内部表示代码。生成中间代码时可以不考虑具体目标机的特性，因此使得产生中间代码的编译程序更容易一些。而且，中间代码的形式与具体的目标机无关，编译程序便于移植到其他机器上。

编译程序使用的中间代码有多种形式，我选择四元式来表达中间代码。

4.1 抽象语法树的生成

完整的编译程序是通过词法分析、语法分析产生了抽象语法树。在抽象语法树的基础上，遍历树节点，进而生成中间代码。因此，首先要完成的任务是扫描源文件，生成对应的抽象语法树。

在 bison 实验二中，已经分析了 C 语言子集的词法、语法分析，并对生成抽象语法树的代码进行了详细的阅读与理解。该过程在前面的实验中已经叙述，此处不再赘述。

bison 实验二涉及到的代码文件及其作用如下表：

表 1 抽象语法树相关代码

文件名	作用
input.lex	词法分析 flex 文件
cgrammar-new.y	语法分析 bison 文件
nconst.h	定义了若干宏
parser.h/parser.c	声明并实现了若干与符号表、抽象语法树相关的结构与函数
main.h/main.c	主函数

也就是说，以上代码文件完成了词法分析、语法分析，并最终生成了抽象语法树。

4.2 四元式的生成

我们在 parser.h 中新增加了如下函数声明，这些函数用于生成中间代码。

```
1. /**
2.  * 在 tac.c 中实现
3.  * 与产生中间代码有关的方法
4. */
5. char *op_string(int op);
6. void tac();
7. static void start_tac(int n);
8. static char *deal_expk(int n);
9. static void MAIN_K(int n);
10.static void DEFINEPARA_K(int n);
```

```
11. static void ASSIGN_K(int n);
12. static fourvarcode* get_tac(int op,char a[],char b[],char c[]);
13. static char * newtemp();
14.
15. static void EXP_K(int n);
16. static void IF_K(int n);
17. static void WHILE_K(int n);
18. static char* GO_EXP_TAC(int op, int n);
```

以上函数均在 tac.c 文件中实现。

start_tac 函数是生成四元式的入口函数。在 start_tac 函数中，函数参数是待分析的 AST 节点编号。若节点 n 的类型是我们的编译程序可以识别的(行 12, 17, 23, 29)，那么就转向相应类型节点的处理程序(行 14, 19, 25, 30)，随后处理节点 n 的下一个兄弟节点。否则，我们递归的进入节点 n 的子节点。

```
1.  void start_tac(int n)
2.  {
3.   /**
4.    * 匹配到某一已知节点，则去处理它以及其兄弟节点（广度优先）
5.    * 否则
6.    * 对其孩子节点递归处理（深度优先）
7.   */
8.     while(n!=0)
9.     {
10.      char *start=node_name[node[n].id];  //取出结点 name
11.      printf("now parser node: %s\n", start);
12.        if(strcmp(start,"funcdecl_")==0)    //main 方法体
13.        {
14.            MAIN_K(n);
15.          break;
16.        }
17.        if(strcmp(start,"declarations_")==0)  //变量定义
18.        {
19.            DEFINEPARA_K(n);
20.          n = node[n].next;    //下一个兄弟结点
21.          continue;
22.        }
23.        if(strcmp(start,"statements_")==0)  //算术表达式
24.        {
25.            ASSIGN_K(n);
26.          n = node[n].next;  //下一个兄弟结点
27.          continue;
```

```
28.         }
29.       if(strcmp(start, "block_") == 0){
30.           start_tac(node[n].child);
31.           break;
32.       }
33.
34.       /**
35.        * 由于只做了部分节点，因此当处理到不识别的节点时，跳过它
36.        *
37.        * 直接进入其子节点的处理
38.       */
39.       n = node[n].child;  //下一个子结点
40.     }
41. }
```

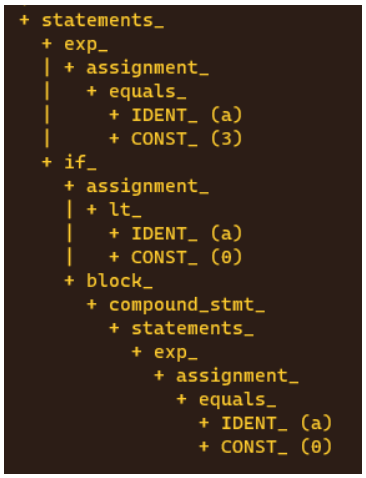
该项目所实现的编译器可以处理的节点类型有变量定义，算术表达式、逻辑表达式、if 语句与 while 语句。不同表达式对应的 AST 节点类型不同，采用函数 MAIN_K、DEFINEPARA_K、ASSIGN_K、EXP_K、IF_K、WHILE_K 处理不同的 AST 节点。

ASSIGN_K 函数用于处理 AST 中的赋值节点。词法、语法规则使得赋值节点有三种类型：表达式 EXP，IF 语句的条件部分、WHILE 语句的条件部分，因此在 ASSIGN_K 中根据当前节点类型，分别调用不同的子函数去处理。

```
1. void ASSIGN_K(int n)
2. {
3.  char t1[10],t2[10];
4.  n = node[n].child;
5.  while(n!=0)
6.  {
7.      char* currentNode = node_name[node[n].id];  //取出结点 name
8.
9.      if(strcmp(currentNode, "exp_")==0)
10.         EXP_K(n);
11.     else if(strcmp(currentNode, "if_") == 0)
12.         IF_K(n);
13.     else if(strcmp(currentNode, "while_") == 0)
14.         WHILE_K(n);
15.
16.     n = node[n].next;
17. }
18. }
```

以下面的测试代码来说明 IF_K 函数。

```
1.  int main()
2.  {
3.      int a;
4.      a = 3;
5.      if(a < 0) {
6.          a = 0;
7.      }
8.  }
9.
```



这一段代码生成的**部分** AST 如上面右图所示。IF_K 函数的代码如下。从部分 AST 中可以发现，IF 节点下通常跟两个子节点——assignment 节点与 block 节点。行 8 将 assignment 节点赋值给 n，行 9 将 assigment 节点的子节点 lt 赋值给 n，行 10 调用 deal_expk 函数处理 lt 小于表达式节点，行 11 调用 get_tac 函数将产生的四元式添加到四元式链表尾部。

行 14 则代表开始处理 block 节点。由于 if 语句中可以跟随的语句类型是任意的，因此行 15 重新调用 start_tac 函数**递归的处理** block 中的节点。

```
1.  /**
2.   * IF 下通常有两个子结点——assignment_、block_(empty_stmt_)
3.  */
4.  void IF_K(int n){
5.   char t1[10],t2[10];
6.
7.   //处理 assignment_
8.   n = node[n].child; //assignment_
9.   int i = node[n].child;      // gt_
10.  strcpy(t1, deal_expk(i));
11.  fourvarcode* f =  get_tac(IF_IF, t1, adr, adr);
12.
13.  //处理 bolck_
14.  n = node[n].next;   //bolck_
15.  start_tac(n);
16.  //每一个块结束意味着前面有一个条件语句。
17.  //此处为条件语句的 False 出口
18.  char thisLabel[8];
19.  sprintf(thisLabel, "label%d", labelNum++);
20.  get_tac(LABEL, thisLabel, adr, adr);
21.
```

```
22.
23.  //sprintf(f->addr2.name, "%d", f->line + 1);
24.  sprintf(f->addr3.name, "%s", thisLabel);
25. }
```

程序生成的中间代码（四元式）如下：

input > test.c > main()

```
1  int main()
2  {
3    int sum;
4    int i;
5    i = 1;
6    while(i <= 10){
7      sum = sum + i;
8      i = i + 1;
9    }
10 }
11
12
13
```

output > tac_before_optimization.txt

```
1   100 (main , _ , _ , _ , )
2   101 (define_variable , int_ , sum , _ , )
3   102 (define_variable , int_ , i , _ , )
4   103 (= , 1 , t#0 , _ , )
5   104 (= , t#0 , i , _ , )
6   105 (label , label1 , _ , _ , )
7   106 (= , i , t#1 , _ , )
8   107 (<= , t#1 , 10 , t#2)
9   108 (if , t#2 , _ , label2)
10  109 (= , sum , t#4 , _ , )
11  110 (= , i , t#5 , _ , )
12  111 (+ , t#4 , t#5 , t#6)
13  112 (= , t#6 , t#3 , _ , )
14  113 (= , t#3 , sum , _ , )
15  114 (= , i , t#8 , _ , )
16  115 (+ , t#8 , 1 , t#9)
17  116 (= , t#9 , t#7 , _ , )
18  117 (= , t#7 , i , _ , )
19  118 (jmp , label1 , _ , _ , )
20  119 (label , label2 , _ , _ , )
21  120 (main_end , _ , _ , _ , )
22
```

图 4 四元式

5 中间代码优化

大部分编译程序会在中间代码或目标代码生成之后对生成的代码进行优化。所谓优化，实质上是对代码进行等价变换，使得变换后的代码运行结果与变换前的运行结果相同，而运行速度加快或占用的存储空间减少。

图 4 的四元式显然存在许多冗余表达式，如行 109~113，将变量 sum 的值赋值给中间变量 t#4，将变量 i 的值赋值给中间变量 t#5，计算两中间变量 t#4 与 t#5 的和并赋值给中间变量 t#6。随后将 t#6 赋值给中间变量 t#3，又将 t#3 的值赋值给变量 sum，从而完成了 sum=sum+i 的操作。

显然，我们可以对上述操作进行优化，直接将 t#4 与 t#5 的和赋值给变量 sum 即可。从而减少了编译程序对内存空间的消耗。

5.1 优化原则

生成的四元式以链表的形式存储在内存中。由于只实现了 C 语言的部分功能，因此优化的原则具有明显的针对性。四元式的形式可以使用(op, addr1, addr2, addr3)表示。

使用窗口大小为 3 的滑动窗口遍历整个四元式链表。当遇到以下情况时采取相应的优化措施：

(1) 连续的两个四元式均属于赋值语句，且前者的 addr2 与后者的 addr1 相同，此时可以将这两个四元式优化为同一个。例如四元式：

$$103\ (=, 1, t\#0, \_)$$

$$104\ (=, t\#0, i, \_)$$

优化后：

$$103\ (=, 1, i, \_)$$

(2) 连续的两个四元式，如果前者不是赋值而后者是赋值语句，且前者的 addr3 与后者的 addr1 相同，此时可以进行优化。例如，四元式：

$$112\ (/, t\#6, 3, t\#7,)$$

$$113\ (=, t\#7, t\#2, \_)$$

优化后：

$$112\ (/, t\#6, 3, t\#2,)$$

5.2 代码

同样，在 parser.h 中增加中间代码优化相关函数的声明。中间代码优化相关函数的实现在 optimizetac.c 文件中：

```
1. /**
2.  * 在 optimizetac.c 中实现
3.  * 与中间代码优化有关的方法
4. */
5. void start_optimize(fourvarcode* tac_head);
6. void optimer_tac(fourvarcode* tac_head);
7. void print_optimize_tac(fourvarcode* tac_head);
```

optimer_tac 函数接受的参数是未优化的四元式链表头节点指针，在函数体中实现了优化。print_optimize_tac 函数则将优化后的四元式链表打印并以文件的形式存储。start_optimize 则是对外提供的优化四元式功能的接口函数，在函数体中分别调用

optimzer_tac 与 print_optimize_tac 完成优化。

```
1. void optimer_tac(fourvarcode* tac_head)
2. {
3.     /**
4.      * t p q 指向四元式链表的三个连续节点
5.      * 即使用窗口大小为 3 的滑动窗口遍历整个链表
6.      * 遇到待优化四元式序列则优化
7.     */
8.     fourvarcode* t=tac_head->next;
9.     fourvarcode* p;
10.    fourvarcode* q;
11.    p=t->next;
12.    q=p->next;
13.    while(q!=NULL)
14.    {
15.        if(p->op==1)
16.        {
17.            if(q->op==1)
18.            {
19.                char a[20];
20.                strcpy(a,p->addr2.name);
21.
    if(strcmp(p->addr2.name,q->addr1.name)==0&&a[0]=='t'&&a[1]=='#')
22.                {
23.                    strcpy(p->addr2.name,q->addr2.name);
24.                    p->next=q->next;
25.                    free(q);
26.                    q=p->next;
27.                }
28.                else
29.                {
30.                    t=t->next;
31.                    p=p->next;
32.                    q=q->next;
33.                }
34.            }
35.            else if(q->op!=1)
36.            {
37.                char a[10],b[20],c[20];
38.                strcpy(a,p->addr1.name);
39.                strcpy(b,p->addr2.name);
40.                strcpy(c,q->addr3.name);
```

```
41.
    if(strcmp(p->addr2.name,q->addr1.name)==0&&b[0]=='t'&&b[1]=='#'&&(c[0]=='\0'||a[1]=='#'))
42.                 {
43.                     strcpy(q->addr1.name,p->addr1.name);
44.                     t->next=p->next;
45.                     free(p);
46.                     p=t->next;
47.                     q=q->next;
48.                 }
49.                 else
50.                 {
51.                     t=t->next;
52.                     p=p->next;
53.                     q=q->next;
54.                 }
55.             }
56.         }
57.         else if(p->op!=1)
58.         {
59.             if(q->op==1)
60.            {
61.                 if(p->addr3.kind!=emptys)
62.                 {
63.                     char a[20];
64.                    strcpy(a,p->addr3.name);
65.                     if(strcmp(p->addr3.name,q->addr1.name)==0&&a[0]=='t'&&a[1]=='#')
66.                     {
67.                         strcpy(p->addr3.name,q->addr2.name);
68.                         p->next=q->next;
69.                         free(q);
70.                         q=p->next;
71.                     }
72.                     else
73.                     {
74.                         t=t->next;
75.                         p=p->next;
76.                         q=q->next;
77.                     }
78.                 }
79.                 else
80.                 {
81.                     t=t->next;
82.                    p=p->next;
```

```
                q=q->next;
            }
        }
        else
        {
            t=t->next;
            p=p->next;
            q=q->next;
        }
    }
}
```

5.3 优化后的四元式

同样使用 4.2 中的测试代码，对其产生的四元式进行优化，优化后的结果如图：

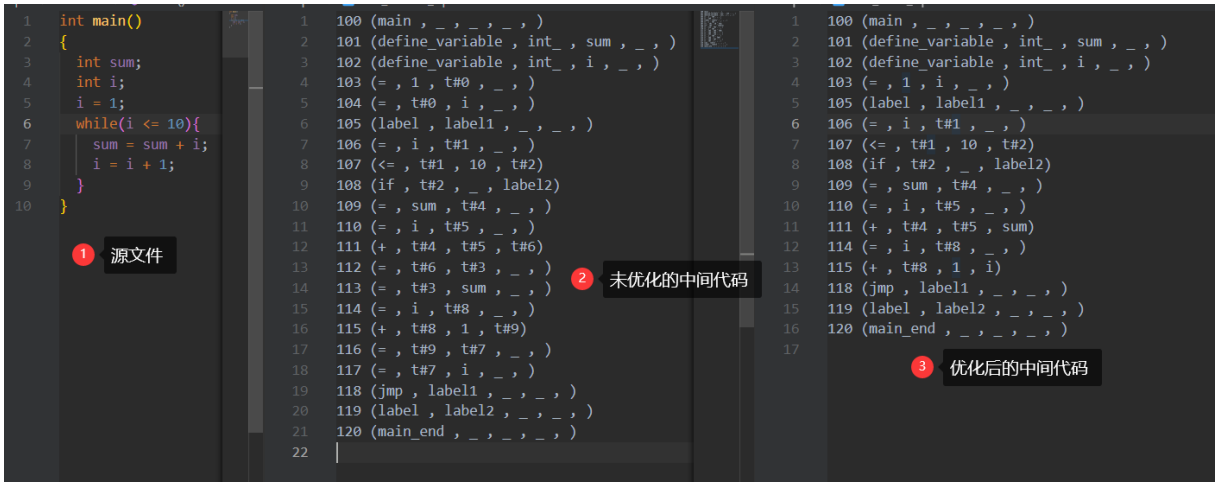


图 5 中间代码优化结果

6 目标代码生成

编译程序的最终目标是将源程序翻译成目标代码，若在语义分析阶段没有完成目标代码的生成，则必须单独完成此项工作。目标代码生成是将编译生成的经过优化得到的中间代码变换成目标代码的过程。目标代码常常分为机器语言和汇编语言两大类。我们采用 x86 汇编语言指令作为目标代码。

6.1 代码结构

同样，在 parser.h 中增加了函数声明：

```
/**
 * 在 asm.c 中实现
```

```
3.   * 与目标代码生成有关的方法
4.  */
5.  void start_asm(fourvarcode* );
```

start_asm 的参数是四元式链表头指针，函数实现了将中间代码生成目标代码的相关逻辑。该函数在 asm.c 文件中实现。

6.2 变量存储空间的分配

由于在词法、语法分析以及中间代码生成的过程中，已经将全部的变量存储在符号表中，那么在对变量进行存储空间分配时就很简单了——只需要遍历符号表，为每一个变量在存储器中开辟一个新的存储空间即可。代码如下：

```
1.  void writeAllocStatement(){
2.      symboltable* p = local_table->next;
3.      int base = 0;
4.      while(p != NULL){
5.          char* type = p->type;
6.          char* name = p->name;
7.          int location = p->location;
8.
9.          // 写入如下格式:
10.         // MOV AX, num
11.         // MOV [location], AX
12.         char text[64];          // 缓冲区
13.         sprintf(text, "MOV AL, %s", "0");       //这里暂时写入 0 吧，有点没搞懂为什么
    要设置一个 ALLOC 标记
14.         doPrintWithOneTab(text);
15.         sprintf(text, "MOV [%d], AL", location);
16.         doPrintWithOneTab(text);
17.
18.         p = p->next;
19.     }
20.     doPrint("");
21.     doPrint("MAIN:");
22. }
```

6.3 寄存器的分配

寄存器的使用并不是随意的。8086 为我们提供了 AX、BX、CX、DX 四个 16 位寄存器，每一个寄存器又可以拆分成两个 8 位寄存器如 AH、AL。在每次需要寄存器运算时，判断对应寄存器的标志位，若占用则顺次申请新的寄存器。寄存器的分配代码如下：

```
1. char *registerGet()
2. {
3.     if(AL_FLAG == 0){
4.         AL_FLAG = 1;
5.         return "AL";
6.     } else if(AH_FLAG == 0){
7.         AH_FLAG = 1;
8.         return "AH";
9.     } else if(BL_FLAG == 0){
10.        BL_FLAG = 1;
11.        return "BL";
12.    } else if(BH_FLAG == 0){
13.        BH_FLAG = 1;
14.        return "BH";
15.    } else if(CL_FLAG == 0){
16.        CL_FLAG = 1;
17.        return "CL";
18.    } else if(CH_FLAG == 0){
19.        CH_FLAG = 1;
20.        return "CH";
21.    } else if(DL_FLAG == 0){
22.        DL_FLAG = 1;
23.        return "DL";
24.    } else if(DH_FLAG == 0){
25.        DH_FLAG = 1;
26.        return "DH";
27.    }
28.    return "-1";
29.}
```

6.4 具体运算

对于类似 (+ , t#4 , t#5 , sum)这样的加法四元式而言，其处理逻辑如下。

```
1. void build_adds(fourvarcode* t)
2. {
3.     int isTemp1 = 0, isTemp2 = 0;
4.     char reg_1[64], reg_2[64];
5.     int location1, location2;
6.     getVal(&(t->addr1), reg_1, &isTemp1);
7.     getVal(&(t->addr2), reg_2, &isTemp2);
8.
9.     //相加后， 赋值给变量
10.    char reg[8];
11.    get_copy_object(&(t->addr3), reg);
```

```
12.     //int location = searchlocal(t->addr3.name);
13.     char text[64];
14.     //写入
15.     sprintf(text, "ADD %s, %s", reg_1, reg_2);
16.     doPrintWithOneTab(text);
17.     sprintf(text, "MOV %s, %s", reg, reg_1);
18.     doPrintWithOneTab(text);
19.
20.     registerFree(reg_1);
21.     registerFree(reg_2);
22. }
```

通过调用 getVal 函数，将 addr1 与 addr2 中的变量值存放申请到的寄存器中。调用 get_copy_object 函数，得到相加后结果应该存储的变量地址。最后行 15~18 完成了 ADD 语句的生成。

6.5 模拟器运行结果

EMU8086 模拟器可以在运行中实时查看各寄存器、各存储单元的状态。对于在 main 函数开始处声明的若干变量，按顺序依次存放在基地址为 0710，偏移量从 1 逐次递增的存储单元。对于图 7 左侧所示的源代码，变量 a 存放在地址 0710:0001 处，变量 b 存放在地址 0710:0002 处。

执行完全部汇编代码后，从图中可以看出，a 的值为 0x0F，即十进制的 15；b 的值为 0x05，即十进制的 05。

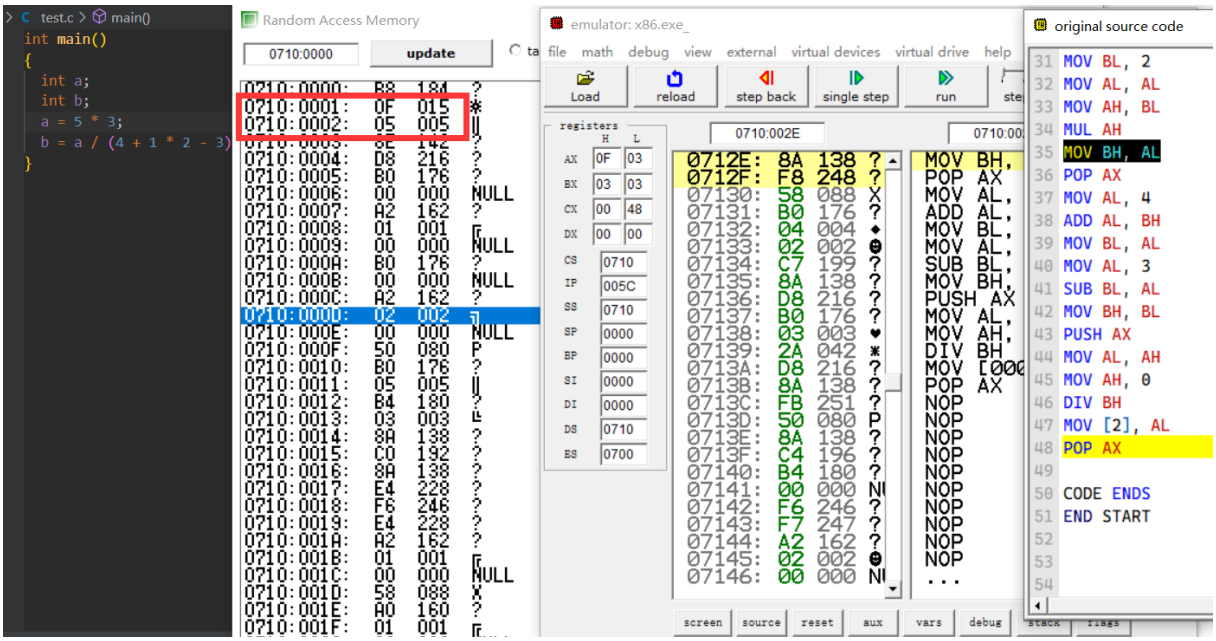


图 7 运行结果 1

对于更复杂一些的例子，如图 xxx。首先将 a 的初始值赋值为 5，由于 a 的值大于 0，因此进入 if 的语句块中。最终 a 的值应为 3（存储单元 0710:0001）。

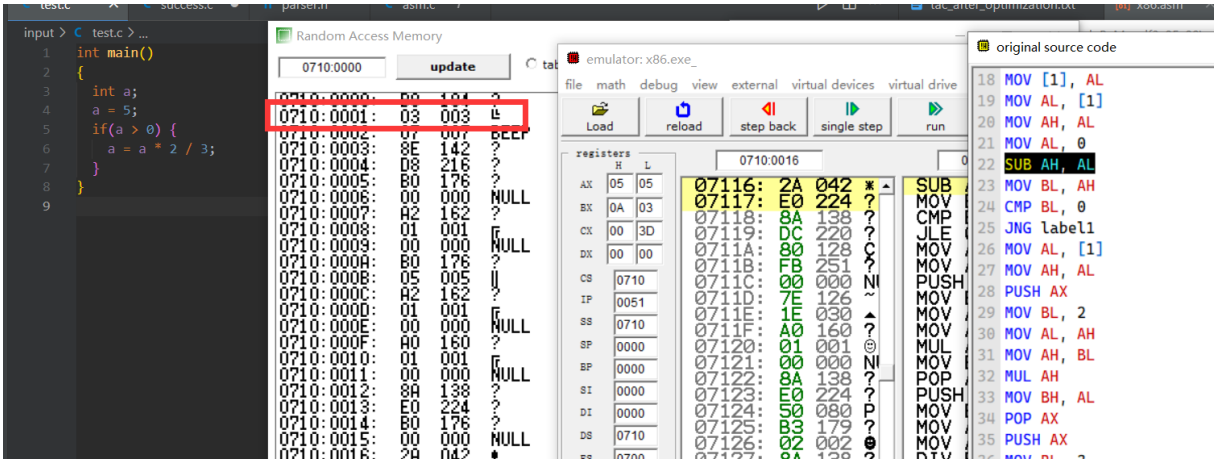


图 8 执行结果 2

而在图 xx 所示的例子中，因为初始化的方式使得未经赋值的 sum 初值为 0。该段程序计算了自然数 1~10 的和。最终存储在 sum 中。可以看到，存储单元 0710:0001 的值为 0x37 即十进制的 55。

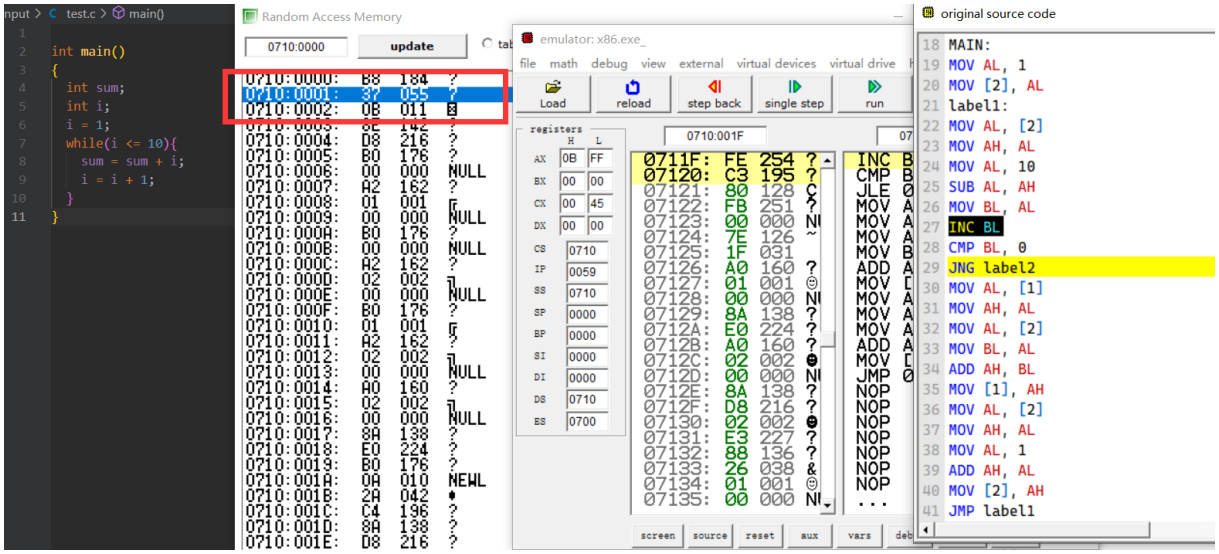


图 9 执行结果 3

7 GUI 程序

将本项目的全部 flex、bison 以及 c 语言源代码打包成动态链接库文件，该动态库包含了词法、语法分析、中间代码生成、目标代码生成功能。通过 JNA jar 包调用动态链接库，并使用 Swing 制作简单的可视化界面，如下：

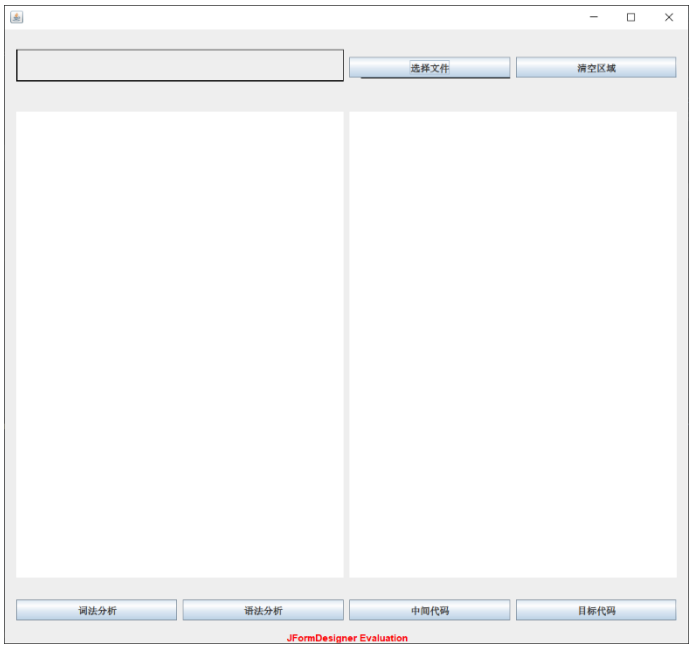


图 10 GUI 程序

通过选择文件按钮，选择源代码文件，加载到左侧代码框中。点击右下角目标代码按钮，在右侧文本框中可以显示生成的目标代码。

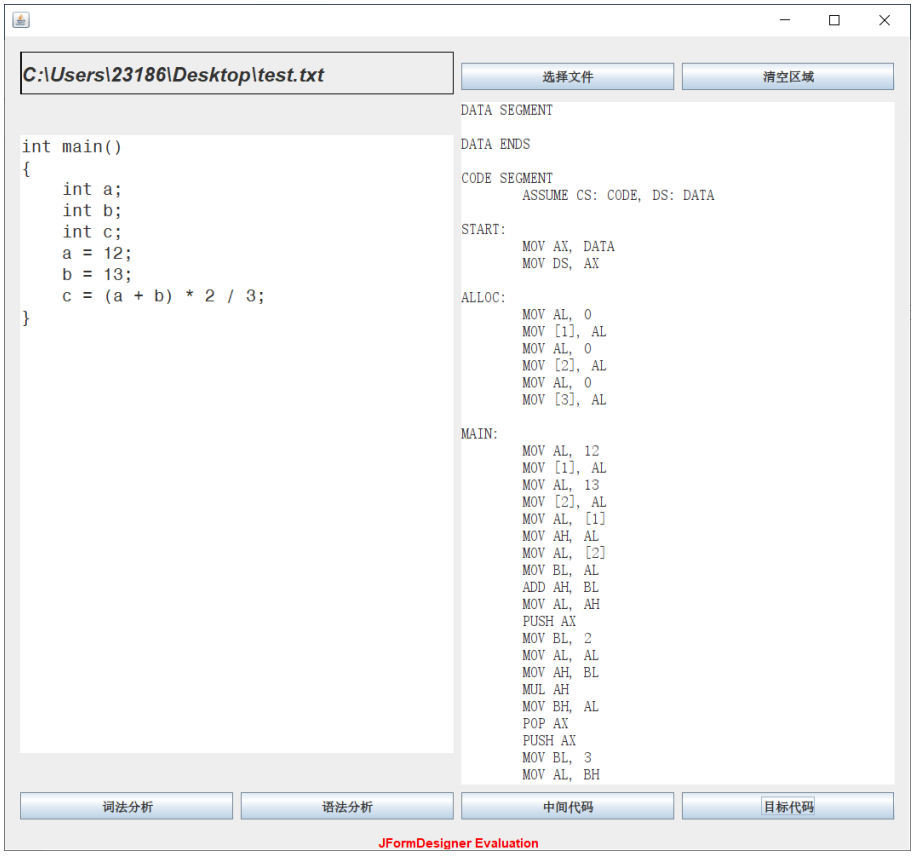


图 11 GUI 示例

8 实验总结

本次实验难度较大。

最初拿到题目时感到无从下手。在编译课程上也仅仅是简单的讲了讲中间代码生成，对于后续的优化、生成目标代码相关知识则需要自己重新学习才可理解。在开源源代码的基础上，为了实现 if 语句与 while 语句，增加了近一倍的代码。完成项目时，深感计算机体系结构相关技术的难度之大。尽管完成的编译器仅仅是 C 语言的冰山一角，尽管受限于时间与精力，项目存在着诸多不足之处，但通过本项目，我对编译器的整个工作流程有了更深的体会，进一步巩固了《编译技术》课程的知识。

受益匪浅。