华中科技大学计算机科学与技术学院

一个用软件实现的模拟指令系统及其汇编语言
——Xmips 指令系统与 ABC 汇编语言

专业：计算机科学与技术

班级：0903

学号：U200914943

姓名：余晨晔

2010 年 12 月 16 日

目录

一、概述 1
1.1 关于 Xmips 1
1.2 最早的 Xmips 指令系统 1
1.3 改进后的 Xmips 指令系统 2

二、Xmips 指令系统 3
2.1 Xmips 中的寄存器 3
2.2 Xmips 指令格式 4
2.3 Xmips 的寻址方式 5
2.4 地址空间说明 6

三、ABC 汇编语言 7
3.1 一个简单的例子——SUM 7
3.2 寻址方式 7
3.3 机器指令语句和伪指令 8
3.4 系统功能调用简介 11

四、汇编过程 13
4.1 从源程序到机器指令 13
4.2 SUM 程序的汇编过程 14

五、执行过程 18
5.1 执行机构 18
5.2 数据通路 18
5.3 程序 SUM 的执行过程 19

六、Xmips 操作实例 22
6.1 路径设置和参数配置 22
6.2 编辑汇编源程序 23
6.3 运行 Xmips 24

七、Xmips 其他功能简介 25

附录 26
A Xmips 指令操作码和 ABC 汇编语言助记符对照表 26
B 系统运行状态信息表 28

一、概述

1.1 关于 Xmips

Xmips 是用 C++编写的一个旨在模拟多进程调度和资源分配的程序。Xmips 系统组成部分目前主要包括汇编程序部分，指令解释和执行部分和进程调度部分。本文主要涉及到前两部分。

目前的 Xmips 是一个 Windows XP 系统下的单线程的控制台程序。要用它模拟多进程并发，必须为其设计一套模拟的指令系统和程序设计语言。为了实现这一目标，我仿照 X86 汇编语言的形式，为 Xmips 系统设计了一套汇编语言指令，命名为 Stimulated Assembly Language By C++（以下简记为”ABC 汇编语言”），并为其编写了一个简单的汇编程序，用于将汇编源程序翻译成可以被系统执行的机器码。运行在 Xmips 系统上的用户程序均是由这种汇编语言编写的，而 Xmips 的系统程序和系统函数库部分用 ABC 汇编语言编写，其余则由 C++编写，对于由 C++编写这部分内容，目前可以解释为这是 Xmips 系统模拟的“硬件功能”。

编写 Xmips 的初衷是为了结合组成原理，操作系统和面向对象程序设计语言课程中学到的知识，设计出一个模拟计算机系统运行的程序。由于最早写出的 C++程序中主要设计了 Memory，Interpreter，Processer 和 Stack 四个类，所以我把由它们扩展而成的程序叫做 Xmips，后来学到指令系统的有关课程时，才知道 MIPS 是一种 RISC CPU，也算是很巧合吧。

为了简化，我把 Xmips 中所有的数据单元都定义为整型，即指令的每一个字段都是一个整型量，模拟的内存以 int 为编址单位，所有寄存器的大小也都是 int。所以目前 Xmips 系统的汇编语言是一种无类型的语言，不支持字符类型和浮点类型，一切操作都针对整型量。

1.2 最早的 Xmips 指令系统

最早的 Xmips 系统中没有设计寄存器，指令的实行实际上是根据代码段的内容直接修改数据段。不设寄存器的原因是因为认为在纯软件模拟的条件下不存在寄存器和主存的访问速度差别。但是在主存中增加了一个用于保存操作码和操作数地址等信息的区域。在后来的设计中，该区域被取消，其功能由系统提供的寄存器组取代。

最初开始设计的时候，由于还没有学到组成原理指令系统课程，我所设计的 ABC 汇编语言和机器指令从格式上是严格对应的，即：

操作码　目的操作数地址　源操作数地址

对于单操作数的指令和无操作数的指令，操作数地址缺省为 0。

这种方法的寻址方式单一，仅仅支持直接寻址。为了支持立即数寻址，只能专门增加一组针对立即数的指令，其操作码助记符形式为“IMXXX”，如 IMADD SUM 9，表示将立即数 9 加到符号地址 SUM 所对应的主存单元中，即 SUM <— (SUM)+9。

由于某些程序中需要间接寻址，我用“@”符号表示实际的地址为@后地址单元中的值，

如 MOV SUM @I，表示 SUM <— ([I])。但是其功能的实现，是在执行时用符号地址为 I 的单元中的值替换掉代码段中的相应字段，然后在执行后在把原来代码段中的内容换回来完成的。为此，我不得不为每一条指令的操作数字段附加一个说明字，用于说明是否采用了间接寻址。

1.3 改进后的 Xmips 指令系统

在学习了组成原理指令系统这一章后，我对原来的 Xmips 指令系统和 ABC 汇编语言做了一些改进。

1）在执行器件中，新增了寄存器

设计了两个寄存器组 MRgst 和 GM，以及一些供系统使用的寄存器，如指令计数器 PC，状态寄存器 flag 等。

2）指令格式和 ABC 汇编语言格式分离，增加了寻址方式字段

ABC 汇编语言格式与原来相同，但是增加了用于偏移寻址和间接寻址的操作符&和@，以及寄存器标识符#0~#14。

指令格式则由四个定长的字段构成，每个字段的长度为一个整型量（4B）：

操作码　　寻址方式　　目的操作数地址　　源操作数地址

支持的寻址方式包括立即数寻址，直接寻址，间接寻址，寄存器寻址，寄存器编址寻址，变址寻址等，该字段的值在汇编过程中计算。对于双操作数的情况，该字段将由目的操作数的寻址方式和源操作数的寻址方式拼接而成。同时在汇编过程中，将计算操作数的形式地址。当指令执行时，将通过寻址方式和形式地址计算操作数的有效地址。

3）细分了解释执行指令的流程

在最早的 Xmips 系统中，每条指令执行时，分别根据自己的数据通路（数据通路将在 5.2 中详细介绍）取操作数，改进后的指令执行部件中，同一类数据通路的指令的取数和存数操作将统一完成。其操作的一般形式为

<1>取指令。由于指令由 4 个整型量构成，故取一条指令的过程包括：

取操作码

取寻址方式

取目的操作数形式地址（缺省为 0）

取源操作数形式地址（缺省为 0）

指令计数器加 1

<2>计算操作数有效地址

<3>取操作数，操作数可能在寄存器中，主存中，立即数或操作数就是地址

<4>执行运算操作

<5>存操作数

<6>取下一条指令

二、Xmips 指令系统

需要再次说明的是，为了简化操作，Xmips 系统中的基本数值单位是整型（int，4Byte），内存单元以 int 编址，所有指令的长度，和寄存器（组）的长度都只能是 int 的整数倍。

2.1 Xmips 中的寄存器

Xmips 中的寄存器分为两类，第一类为内部寄存器，仅供系统使用，用户不可以访问。内部寄存器包括指令计数器 PC，程序状态寄存器 flag，寄存器组 GM 中编号为#15，#16 的寄存器（数据寄存器），以及寄存器组 MRgst 中的所有寄存器。

第二类为通用寄存器。通用寄存器用户可以使用。Xmips 提供了 15 个整型通用寄存器，全部在寄存器组 GM 中，寄存器的编号为#0~#14，其中#1~#9 号寄存器可用于变址寻址。

由于 GM 中的#15 和#16 号寄存器允许直接与通用寄存器交换数据，所以将它们于其他 15 个通用寄存器设置在同一个寄存器组 GM 中。以下简记寄存器组 GM 中的第 i 号寄存器为# i。

Xmips 寄存器汇总，具体功能见表 2.1
寄存器 PC。
寄存器 flag。
寄存器组 MRgst。默认寄存器数量，10。
通用寄存器组 GM。默认寄存器数量，20。

表 2.1　Xmips 寄存器

寄存器组	序号	功能	说明	可使用者
PC	0	指令计数器	寄存器组 PC 中只有一个寄存器	系统
flag	0	状态寄存器	寄存器组 flag 中只有一个寄存器	系统
MRgst	0	保存操作码		系统
MRgst	1	保存寻址方式	对于双操作数，寻址方式分解成目的操作数的寻址方式和源操作数的寻址方式	系统
MRgst	2	保存目的操作数形式地址和有效地址	先保存形式地址，然后结合寻址方式计算有效地址并保存	系统
MRgst	3	保存目的操作数所在的位置信息	0 表示操作数在主存，1 表示操作数在寄存器，2 表示操作数为立即数	系统
MRgst	4	保存源操作数形式地址和有效地址	同目的操作数	系统
MRgst	5	保存源操作数所在的位置信息	同目的操作数	系统
MRgst	6~14	未定	留作扩展	系统
GM	0	通用寄存器		系统，用户
GM	1~9	通用寄存器	可作偏移寻址的变址寄存器，格式：&R	系统，用户

GM	10	通用寄存器	保存中断号和中断返回信息	系统，用户
GM	11~12	通用寄存器	用于系统功能调用时传参	系统，用户
GM	13~14	通用寄存器		系统，用户
GM	15	数据寄存器	保存目的操作数的数值	系统
GM	16	数据寄存器	保存源操作数的数值	系统
GM	17~19	未定	留作扩展	系统

2.2 Xmips 指令格式

Xmips 指令共有 4 个字段，每个字段的大小为一个整型量。

指令格式： 操作码 寻址方式 目的操作数形式地址 源操作数形式地址

1）操作码字段

操作码（opcode）可以分解为三个部分，操作标识（op），数据操作类型标识（d），操作数数目标识（t）。

操作数数目标识 t 为操作码的个位数字，表示操作码对于操作数的格式。双操作数操作码，t=2；单操作数操作码，t=1；无操作数操作码，t=0。

数据操作类型标识（d）为操作码的十位数字，表示执行时存取操作数的方式，d 与指令是否取目的操作数和源操作数的数值，是否在执行完成后保存源操作数有关。

操作标识 op 为操作码的其他数字，op 与操作一一对应。

例如 ADD #2 SUM，ADD 的操作码为 900272。

操作标识 op=9002，唯一标识加法操作。

数据操作类型标识 d=7，表示加法操作要取源操作数和目的操作数的数值，运算结果要保存到目的操作数的数据单元。

操作数数目标识 t=2，表示 ADD 操作有两个操作数。

2）寻址方式字段

用一个整型量表示寻址方式（addressing）。

对于双操作数，其寻址方式分别用 addressing_d 和 addressing_s 表示，由于指令中只有一个寻址方式字段，故将 addressing_s 和 addressing_d 拼接成复合寻址方式 addressing，addressing=addressing_d+addressing_s*1000。

对于单一操作数，寻址方式 addressing 为一个 3 位十进制整数 ABC。在解释指令时，又将 ABC 分解为两个部分分别解释：

<1>R = A = addressing / 100，即 addressing 的百位数表示用于偏移寻址的通用寄存器号（R 的范围是 1~9）。

<2>type = BC = addressing mod 100，即 addressing 的十位和个位表示寻址方式的类型。

type 的范围是 0~15，对应一个二进制数 $(S0S1S2S3)_2$ ，其中：

S0=1 表示偏移寻址
S1=1 表示间接寻址
S2S3=00 表示操作数在主存
S2S3=10 表示操作数在寄存器
S2S3=11 表示操作数为立即数

例如 ADD #1 3。目的操作数#1 表示通用寄存器#1（在寄存器组 GM 中），源操作数 3 为立即数，故

$$addressing\_d = （0010）_2 = 2$$

$$addressing\_s = （0011）_2 = 3$$

$$addressing = addressing\_d + addressing\_s * 1000 = 2 + 3 * 1000 = 3002$$

即该指令的寻址方式字段的值为 3002。

3）操作数形式地址字段
操作数形式地址（opreand address， 简记为 oa）于操作数存储的位置有关。

若操作数在数据段（位于主存）的段内偏移地址为 M，则 oa = M。
若操作数在通用寄存器 R，则 oa = R。
若操作数为立即数 I，则 oa = I 。此时寻址方式字段为 3，表示立即数。

2.3 Xmips 的寻址方式

1）Xmips 指令支持的寻址方式包括
<1>寄存器寻址
<2>寄存器间接寻址
<3>直接寻址
<4>间接寻址
<5>基址或变址寻址。操作数在主存，有效地址是某通用寄存器与指令中某常量之和。
<6>立即数寻址

2）有效地址的计算
ABC 汇编语言地址形式：[&R][@]K。
若包含&R 为偏移寻址，使用通用寄存器#R。
若包含@为间接寻址。
K 应该是主存地址（包括符号地址（如 SUM）和数值地址（如！10）），通用寄存器号（如#2 表示 2 号寄存器）或立即数（如 80 ）中的一种。
多重寻址方式可能混合，如&6@SUM 包含偏移寻址和寄存器间接寻址。

如汇编地址为&6@SUM，若(#6)=2，(SUM)=10，则有效地址 EA=(#6)+(SUM)=2+10=12。

若操作数的形式地址为 D，寻址方式为 addressing，设 EA 为要计算的有效地址，则有效地址的计算见表 2.2。

表 2.2　有效地址的计算

addressing	S0S1S2S3	汇编地址形式示例	有效地址的计算	操作数位置
0	0000	SUM，!0	EA=D	主存
1	0001	无效	无效	无效
2	0010	#2	EA=#D	寄存器
3	0011	6	不存在	立即数
4	0100	@ST，@!10	EA=(D)	主存
5	0101	无效	无效	无效
6	0110	@#2	EA=(#D)	主存
7	0111	无效	无效	无效
8+100R	1000	&3SUM，&9!21	EA=(#R)+D	主存
9+100R	1001	无效	无效	无效
10+100R	1010	无效（&3#4）	无效	无效
11+100R	1011	无效	无效	无效
12+100R	1100	&5@ST	EA=(#R)+(D)	主存
13+100R	1101	无效	无效	无效
14+100R	1110	&6@#5	EA=(#R)+(#D)	主存
15+100R	1111	无效	无效	无效

2.4 地址空间说明

目前的 Xmips 缺少内存管理机构和地址变换机构，实际上相当于每一个进程的代码段，数据段和堆栈段都是一个物理上独立的存储器。关于其地址空间的规定如下：

<1>程序的逻辑地址空间就是存储器的物理地址空间。

<2>程序的代码段，数据段和堆栈段分别对应三个存储器，每个段（存储器）的起始地址为 0。

在 Xmips 系统的源代码中，用 MCode 表示代码段，用 MData 表示数据段，用 S 表示堆栈段。下文中，采用诸如 MCode[5]的方式表示存储器的单元号，如 MCode[5]表示代码段（存储器）的第 5 号单元，（MCode[5]）表示代码段第 5 号单元的值。

下一步，将对 Xmips 的内存表示进行改进，将逻辑地址空间与物理地址空间分离，建立起相应的内存管理机构和地址变换机构。从而使 Xmips 系统中进程的概念更接近与实际情况。（目前 Xmips 系统中进程的概念实际上进程的代码段存储器，数据段存储器和堆栈段存储器和一组进程描述信息的结合，与实际的进程的概念有很大偏差）。

三、ABC 汇编语言

ABC 汇编语言是由一组 Xmips 系统机器指令的助记符和伪指令过程，由 Xmips 系统的汇编程序翻译成机器码后在 Xmips 系统的执行机构上运行。

3.1 一个简单的例子——SUM

下面给出了一个简单的 ABC 汇编语言的例子。

```
0       ~ 计算 SUM=1+2+...+10
1       DIM SUM 0
2       MOV #1 1
3       : L1
4       ADD SUM #1
5       INC #1
6       CMP #1 11
7       JB L1
8       END
```

这个例子计算 1 加到 10 的和。其中第 0 行为注释语句，第 1 行为变量定义语句，第 3 行为标号语句，这 3 行属于伪指令，在源程序被汇编之后，将不会生成这几行的机器码。

源程序汇编之后，生成的机器码为一组整型数序列，将其转换为字符后生成的文件称为为字符码文件。以上程序的字符码文件内容如下：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 4 0 900000 1 0

字符码文件被转换成整型读入内存后，可以直接在 Xmips 执行机构上运行。

3.2 寻址方式

1）寄存器寻址

格式　　#R

功能　　寄存器 #R 的内容即为操作数。

示例　　INC #5

INC 为加 1 指令操作符，#5 <— (#5) + 1。

2）寄存器间接寻址

格式　　@#R

功能　　寄存器 #R 的内容为操作数的偏移地址。

示例　　MOV #0 @#5

MOV 为传送指令，#0 <— ([#5]) 。

3）直接寻址

格式　　S　或　!K

说明　　S 为定义的变量名；K 为一个表示数据段地址的整数，即可以直接指明要使用的地址单元。

功能　　操作数形式地址码字段就是操作数在数据段中的地址。
示例　　MOV SUM #5
SUM <— (#5)

或　ADD　S　!8
S <— (S) + (8)

4）间接寻址
格式　　@S 或　@!K
说明　　S 和 K 的意义同直接寻址。
功能　　形式地址是操作数地址的地址。
示例　　MOV　#0　@SUM
#0 <— ([SUM])

5）偏移寻址（包括变址寻址和基址寻址）
格式　　&RX
说明　　R 为寄存器号（允许 1~9），X 的内容可以用寄存器间接寻址，直接寻址和间接寻址的方式给出。
功能　　有效地址由寄存器#R 中的值和形式地址相加得到。
示例　　ADD　#0　&2SUM　(即 R=2，X=SUM）
源操作数的有效地址 EA = (#2) + SUM
该指令的功能为　#0 <— (#0) + (SUM)

或　　INC &5@#2　（即 R=5，　X=@#2）
目的操作数的有效地址 EA= (#5) + ([#2])
该指令的功能为　(#5) + ([#2])　　<— ((#5) + ([#2])) +1

6）立即数寻址
格式　　n
功能　　指令的形式地址字段就是操作数。
示例　　MOV　SUM 5
SUM <— 5

多重寻址方式有时可以组合使用，如偏移寻址示例中的 INC &5@#2，还包含了寄存器间接寻址的成分。

但是这种组合不可以随意进行，有如下两条限制
<1>立即数寻址方式中不可以有其他成分，如@8 是错误的。
<2>偏移寻址的 X 部分不可以是寄存器寻址，如&5#3，（X=#3）是错误的。

3.3 机器指令语句和伪指令

1）变量，标号与地址表达式
<1>变量定义语句

格式　　DIM　　*S*　[D*n*]　初值 1，初值 2，... ，初值 K　**；**

说明　　*S* 为变量名，变量名要以字母开头。*n* 为定义变量的个数，当定义一组初值时，要以分号结尾。如果 DIM 语句之定义一个变量，则 D*n* 可以省略。如果初值不确定，可用“？”代替。

功能　　定义一个变量或一个数组（使用 D*n*）。

示例　　DIM　SET　D5　1,2,3　**；**

定义了一个大小为 5 的数组，并为数组中前三个元素赋初值。

或　　DIM　　MAX　　100

定义了一个名为 MAX 的变量，初值为 100

<2>标号定义语句

格式　　**：** L

功能　　定义一个标号。

示例　　**：** L2

定义了一个标号 L2。

<3>地址表达式

格式　　3.2 中所定义的所有寻址方式都可以作为地址表达式。

说明　　对于双操作数的指令，立即数不可以作为目的操作数。

2）常用的机器指令语句

ABC 汇编语言的语句形式类似于 X86 汇编，除此之外，增加了一些由于和调度系统交互的语句，但是这些语句只能在系统函数中使用，不能出现在用户自己编写的程序中。用户要实现相应的操作，如访问系统内存共享区，必须通过 INT 软中断的方式进行。具体机器指令语句见表 3.1。

表 3.1 常用的机器指令语句

类型	格式	功能	说明
数据传送	MOV　OPD　OPS	OPD <—（OPS）	
地址传送	LEA　OPD　OPS	OPD <— OPS	
算术运算	ADD　OPD　OPS	OPD <— (OPD) + (OPS)	
	INC　OPD	OPD <— (OPD) + 1	
	SUB　OPD　OPS	OPD <— (OPD) - (OPS)	
	NEG　OPD	OPD <— - (OPD)	
	CMP　OPD　OPS	(OPD) - (OPS)	比价结果保存到状态寄存器 flag
	MUL　OPD　OPS	OPD <— (OPD) * (OPS)	
	DIV　OPD　OPS	OPD <— (OPD) / (OPS) #0 <— (OPD) % (OPS)	
	MOD　OPD　OPS	OPD <— (OPD) % (OPS)	
逻辑运算	ADD　OPD　OPS	OPD <— (OPD) & (OPS)	使用的逻辑运算符号是 C 语言中
	XOR　OPD　OPS	OPD <— (OPD) | (OPS)	

	NOT　OPD	OPD <— ~ (OPD)	的位运算符号
	XOR　OPD　OPS	OPD <— (OPD) ^ (OPS)	
跳转指令	JA　L	flag > 0，PC=L	
	JB　L	flag < 0，PC=L	
	JMP　L	PC=L	
	JE　L	flag == 0，PC=L	
	JNE　L	flag != 0，PC=L	
栈操作	PUSH　OPD	（OPD）压栈	
	POP　OPD	OPD <— 出栈结果	
	PUSHA	所有通用寄存器压栈	通用寄存器指的是#0~#14
	POPA	所有通用进程器出栈	
函数调用	CALL　FUNCTION	调用函数 FUNCTION	
	RET	返回调用函数	
软中断	INT	软中断	中断号在#10 中
	IRET	中断返回指令	返回码在#10 中
	CLI	关中断	
	SLI	开中断	
访问系统共享区	SYSR	读系统共享区	只有系统函数可以使用。#11 为访问系统的起始地址，#12 为访问的长度
	SYSW	写系统共享区	
多进程控制指令	WAKE	唤醒进程	只有系统函数可以使用。#11 为待唤醒进程的通信标识符，#12 为待唤醒的进程所在的等待队列号
	SET	设置进程的通信标识符	#11 为通信类型，#12 为通信标识符的值
结束指令	END	程序结束	汇编时遇到 END 结束汇编过程，遇到 HALT 不结束。HALT 可用于函数定义在主程序之后的情况。
	HALT	停机指令，程序结束	

3）其余的伪指令

<1>注释

格式　　~　注释内容

或

CMNT　注释内容

说明　　注释内容中间不可以有分隔符

<2>函数定义语句
格式　　PROC　FUNCTION
功能　　函数 FUNCTION 从表示下一行的语句开始（至 RET 语句结束）。

注意，ABC 汇编语言的操作符支持纯大写字符或纯小写字符，如“ADD”和“add”都是正确的，但是混合使用是非法的，如“Add”或“aDD”为非法。

所有 ABC 汇编语言助记符和 Xmips 指令操作码对照见附录 A。

3.4 系统功能调用简介

Xmips 系统提供了一组系统函数，用户通过使用软中断语句 INT 可以调用这组函数，调用前要将调用号传送到寄存器#10 中，同时还需要将函数参数传给相应的寄存器。

下面给出一个系统调用的例子。

程序 BUBBLE_INT.cupa 定义了数组 SET，SET 中有 10 个数，程序通过 10 号系统调用对这 10 个数进行排序。寄存器#11，#12 用于传参，寄存器#10 用于提供调用号。

BUBBLE_INT.cupa 源程序如下：

```
~ 定义数组 SET
DIM SET D10　2,4,51,13,17,40,5,22,3,16;
~ 传送数组首地址到#11
MOV #11 0
~ 传送待排序数的个数到#12
MOV #12 10
~ 传送调用号到#10
MOV #10 10
INT
END
```

10 号系统调用的汇编源程序如下：

```
~　INT10
~　sysfun:冒泡排序
~　#11:数组起始地址
~　#12:排序串长度
~　#0:位置 a[i]指针
~　#1:位置 a[i+1]指针
~　#2:交换存储单元
~　#3:每轮终结单元
~　#4:数组结束地址
~　传参区#11,#12
MOV #4 #11
```

```
ADD #4 #12
SUB #4 1
MOV #0 #11
MOV #1 #11
INC #1
MOV #3 #4
: L2
CMP @#0 @#1
JA L1
~ 交换前后单元
MOV #2 @#0
MOV @#0 @#1
MOV @#1 #2
: L1
INC #0
INC #1
CMP #0 #3
JB L2
~ 下一轮排序
MOV #0 #11
MOV #1 #11
INC #1
SUB #3 1
CMP #3 0
JNE L2
~ 排序结束中断返回
MOV #10 0
IRET
END
```

程序的运行结果（保存在进程的数据段中）

```
process ID:51
memory ID:1
0    51
1    40
2    22
3    17
4    16
5    13
6    5
7    4
8    3
9    2
```

四、汇编过程

4.1 从源程序到机器指令

1）汇编器 ASM

ASM 是 Xmips 系统中由于对汇编源程序进行汇编的函数。它将字符串形式的汇编语句转换成数字形式的机器码。

汇编的过程中涉及到四张表：Opcode Table，Variable Table，Tag Table 和 Jump Table。

Opcode Table 中记录汇编操作符对应的机器码（包括伪指令的数字码），其记录的字段如下：

序号 No	操作符 opcode	机器码

Variable Table 中登记 Dim 语句定义的符号地址（变量名）。其记录的字段如下：

序号 No	变量名 var	变量数据段首地址 pos	有效位 valid

Tag Table 中登记标号语句所定义的标号，其记录的字段如下：

序号 No	标号 tag	标记的代码段地址 pos	有效位 valid

Jump Table 中登记跳转语句所引用的标号，其记录的字段如下：

序号 No	标号 tag	跳转语句地址 pos	已找到对应的标号语句 found	有效位 valid

2）ASM 的工作流程。

ASM 首先根据以下过程分析每一条语句。

<1>查找 Opcode Table，将操作助记符（或伪指令）转换成机器码（数字码）。

<2>如果是助记符，则将机器码写入指令的操作码字段，转<6>。否则转<3>

<3>如果该伪指令为变量定义语句，将定义的变量计入 Variable Table，转<9>。

<4>如果该伪指令为标号语句（或函数定义），则将标号（或函数名）登记到 Tag Table，转<9>。

<5>如果该伪指令是注释语句，直接转<9>。

<6>如果操作符是 END 语句，汇编结束。

<7>如果该助记符是跳转指令（或函数调用指令），将跳转标号（函数名）登记到 Jump Table，转<9>。

<8>分析操作数，计算寻址方式和形式地址。

首先判断操作数中有无&，@，#，！等表示各类寻址方式的符号，并根据各类寻址方式在指令寻址方式字段所占的权值，累加到一个整型量 addressing 上。

如果操作数总出现了变量名，则查找该变量名是否已经登记在 Variable Table 中。如果已登记，则用表中的 pos 字段替换变量名。否则报错，汇编终止。

将 addressing 和操作数的形式地址记入指令的寻址方式字段和操作数形式地址字段。

<9>分析下一条语句。

如果程序中有标号语句（或函数定义语句）和跳转语句（函数调用语句），在再分析完所有语句之后，ASM 将根据 Tag Table 和 Jump Table 的内容修改指令的对于内容，即将跳转语句的标号替换成标号语句对应的行号。具体过程如下：

<1>读取完源程序后。读取标号表中的一条标号记录，到跳转表中查找与其标号字段相同的记录，将代码段中跳转语句所在行的目的操作数改为标号所在的行号（由于每条指令由4个整型量构成，每个整型量占一行，实际代码段中目的操作数的位置是标号的行号加2）。并将对应跳转表中“已找到对应的标号语句 found”字段修改为1。如标号表中有标号“L1”，当检查到跳转表序号为0的记录中的标号也为“L1”时，进行上述操作。

<2>反复执行<1>操作直到标号表中有效位字段 valid 为1的记录全部读过。

<3>检测跳转表中是否还有未转换的记录，即 found 字段为的记录，有则报错，说明该跳转语句的标号无效。

经过上述过程，如果源程序正确，所有跳转语句的标号将被替换成对应的行号。

最后，如果该程序定义了变量，将所定义的变量的值写入指令的最后。

4..2 程序 SUM 的汇编过程

以下是3.1中出现过的一个程序 SUM，下面逐条对该程序进行汇编，生成机器指令。

```
0       ~ 计算 SUM=1+2+...+10
1       DIM SUM 0
2       MOV #1 1
3       : L1
4       ADD SUM #1
5       INC #1
6       CMP #1 11
7       JB L1
8       END
```

1）逐行汇编

0 行：

操作符 “~” ，Opcode Table 机器码字段是3，是注释伪指令，跳过。

1 行：

操作符 “DIM”，Opcode Table 机器码字段是910002，是变量定义伪指令。登记 Variable Table 如下：

序号 No	变量名 var	变量数据段首地址 pos	有效位 valid
0	“SUM”	0	1

2 行：

操作符“MOV”，Opcode Table 机器码字段是900152，是传数机器指令。

将900152写入代码段操作码字段，目的操作数#1为寄存器寻址，源操作数1为立即数，计算出：

寻址方式字段为3002；

目的操作数的形式地址为1；

源操作数的形式地址为1。

生成的机器码：

900152 3002 1 1

3 行：

操作符 “**:**” ，Opcode Table 机器码字段是 920001，是标号伪指令语句。登记 Tag Table 如下：

序号 No	标号 tag	标记的代码段地址 pos	有效位 valid
0	“L1”	4	1

pos 的值为 4 是因为由于 1 条指令占 4 个 int 单元，故 0,1,2,3 四个单元已经被前面的指令占据，下面的指令从第 4 行开始。

4 行：

操作符“ADD”，Opcode Table 机器码字段是 900272，是加法机器指令。

将 900272 写入代码段操作码字段，目的操作数 SUM，查找 Variable Table，找到已登记变量名 SUM 故为直接寻址寻址，源操作数#1 为寄存器寻址，计算出：

寻址方式字段为 2000；

目的操作数的形式地址为 0，即 SUM 在 Variable Table 中的 pos 字段的值；

源操作数的形式地址为 1。

生成的机器码：

900152 3002 1 1 **900272 2000 0 1**

5 行：

操作符“INC”，Opcode Table 机器码字段是 900961，是加 1 机器指令。

将 900961 写入代码段操作码字段，目的操作数#1 为寄存器寻址。

寻址方式字段为 2；

目的操作数的形式地址为 1；

没有源操作数，其形式地址缺省为 0。

生成的机器码：

900152 3002 1 1 900272 2000 0 1 **900961 2 1 0**

6 行：

操作符 CMP”，Opcode Table 机器码字段是 900332，是比较机器指令。

将 900332 写入代码段操作码字段，目的操作数#1 为寄存器寻址，源操作数 11 为立即数，计算出：

寻址方式字段为 3002；

目的操作数的形式地址为 1；

源操作数的形式地址为 11。

生成的机器码：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 **900332 3002 1 11**

7 行：

操作符“JB”，Opcode Table 机器码字段是 900521，是跳转机器指令。

将 900521 写入代码段操作码字段，由于是跳转指令，登记 Jump Table 如下：

序号 No	标号 tag	跳转语句地址 pos	已找到对应的标号语句 found	有效位 valid
0	“L1”	16	0	1

寻址方式字段为 3（标号视为立即数）；

暂时不计算目的操作数的形式地址，缺省为 0；

没有源操作数，其形式地址缺省为 0。

生成的机器码：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 **900521 3 0 0**

8 行:

操作符“END”，Opcode Table 机器码字段是 900000，是结束机器指令。

将 900000 写入代码段操作码字段，由于是结束指令，表明程序结束。

生成的机器码：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 0 0 **900000**

2）处理标号和跳转

逐条汇编结束后，生成的 Tag Table 和 Jump Table 如下：

Tag Table

序号 No	标号 tag	标记的代码段地址 pos	有效位 valid
0	“L1”	4	1

Jump Table

序号 No	标号 tag	跳转语句地址 pos	已找到对应的标号语句 found	有效位 valid
0	“L1”	16	0	1

对照两张表，将所有跳转语句的标号替换成对应的行号。如上表中 Jump Table 中 pos 为 16 的跳转语句对应的标号为“L1”，“L1”已经登记在 Tag Table 中，故将代码段中 16+2 的位置替换成 Tag Table 中“L1”记录的 pos 值，4。16+2 的原因是因为在跳转语句中，跳转标号相当于目的操作数，位于这条语句的 2 号字段（0 号字段为操作码，1 号字段为寻址方式）。

处理完标号之后的机器码：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 **4** 0 900000

3）在末尾添加数据信息

由于程序中定义了一个变量 SUM 并赋初值 0，故将变量的个数和初值登记在机器码的末尾（程序执行时将形成数据段）。

添加了数据信息的机器码为：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 4 0 900000 **1 0**

这就是完整的机器码，即：

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 4 0 900000 1 0

至此，汇编过程结束。

汇编得到的机器码将以字符的形式保存到磁盘上，形成的文件称为字符码文件，扩展名为 .co 。

五、执行过程

5.1 执行机构

1）Xmips 的执行机构用于模拟计算机的处理器，其组成部分包括

<1>寄存器（组）

包括指令计数器 PC，状态寄存器 flag，内部寄存器组 MRgst，通用寄存器及数据寄存器组 GM。各个寄存器的功能已在 2.1 中详细介绍。

<2>译码机构

分析操作码，选择相应的数据通路和运算操作。

<3>有效地址计算机构

根据指令的选择方式字段和操作数形式地址字段，计算操作数的有效地址。

<4>访存机构

从主存取数到数据寄存器和将运算结果返回主存。

<5>运算机构

对数据寄存器的内容执行操作码指定的运算操作。

2）一条指令的执行过程如下

<1>取指令，PC+4（每条指令 4 个 int）。

<2>根据指令的寻址方式字段和形式地址字段计算操作数有效地址。

<3>分析指令的操作码字段，选择数据通路。

<4>根据数据通路将要运算的值（可能是数值或地址）传到数据寄存器。

<5>执行运算。

<6>根据数据通路选择是否把运算结果回存。

5.2 数据通路

1）数据通路的表示

在最早的 Xmips 系统中，每条指令执行时，分别根据自己的数据通路取操作数，改进后的指令执行部件中，同一类数据通路的指令的取数和存数操作将统一完成。如所有的双操作数算术指令，数据通路均为 D <— (D) + (S)，即取目的操作数和源操作数的数值，运算结果要传到目的操作数的地址中。

用一个整型量 dataLS 表示指令的数据通路，dataLS 的范围是 0~8，dataLS 是操作码的十位数数字。V0V1V2 为 dataLS 的二进制值。其中：

V0=1 表示计算结果将保存到目的操作数的地址中，用符号“D <—”表示；

V1=1 表示要取目的操作数的数值到数据寄存器，用符号“(D)”表示；

V2=1 表示要取源操作数的数值到数据寄存器，用符号“(S)”表示；

无相应操作时，对应的 V0/V1/V2 为 0。

各类数据通路见表 5.1。

表 5.1 各类数据通路

dataLS	V0V1V2	数据通路	操作符举例	操作码
7	111	D <— (D) + (S)	add	900272
6	110	D <— (D) + 1	inc	900961

5	101	D <— (S)	mov	900152
4	100	D <— S	lea	902042
4	100	D <— 堆栈	pop	902241
3	011	(D) - (S)	cmp	900332
2	010	(D)	jmp	900621
2	010	堆栈 <— (D)	push	902121
0	000	无操作数	其余操作	XXXX00

2）数据操作的流程

<1>如果 V2=1 且源操作数在主存，则将源操作数载入到数据寄存器（位于通用寄存器组，序号 16）。

<2>如果 V1=1 且目的操作数在主存，则将目的操作数载入到数据寄存器（位于通用寄存器组，序号 15）。

<3>如果源/目的操作数在通用寄存器组，则将操作数移动到数据寄存器。

<4>如果源/目的操作数为立即数，则将操作数直接写到数据寄存器。

<5>对数据寄存器中的操作数执行指令对应的操作。（包括无操作数指令）

<6>如果 V0=1 且目的操作数在主存，则将目的操作数从数据寄存器写到主存单元。

<7>如果 V0=1 且目的操作数在寄存器，在将目的操作数从数据寄存器移动到通用寄存器组的相应寄存器中。

5.3 程序 SUM 的执行过程

下面分析程序 SUM 的执行过程。SUM 的汇编源程序和汇编后产生的字符码文件如下：

源程序

```
0       ~ 计算 SUM=1+2+...+10
1       DIM SUM 0
2       MOV #1 1
3       : L1
4       ADD SUM #1
5       INC #1
6       CMP #1 11
7       JB L1
8       END
```

字符码文件

900152 3002 1 1 900272 2000 0 1 900961 2 1 0 900332 3002 1 11 900521 3 4 0 900000 1 0

下文中 MCode，MData 分别表示程序的数据段和代码段（在 2.4 中介绍）

<1>执行之前：PC=0，flag=0，各个寄存器组的值均为 0。

<2>第 1 条指令：900152 3002 1 1　　（MOV #1 1）

具体操作见表 5.2。

执行完后，PC = 4，（#1）= 1。

表 5.2　第 1 条指令的操作

操作	周期	说明
MRgst[0] <— (MCode[PC++])	取指令	取操作码字段
MRgst[1] <— (MCode[PC++])		取寻址方式字段
MRgst[2] <— (MCode[PC++])		取目的操作数形式地址
MRgst[4] <— (MCode[PC++])		取源操作数形式地址
MRgst[2] <— 1	计算有效地址	目的操作数有效地址为 1（寄存器#1）
MRgst[3] <— 1		目的操作数在寄存器
MRgst[4] <— 1		源操作数有效地址为 1（即立即数的值）
MRgst[5] <— 2		源操作数为立即数
GM[16] <— (MRgst[4])	取值	立即数直接传到数据寄存器
GM[15] <— (GM[16])	执行	源操作数传到目的操作数寄存器
GM[MRgst[2]] <—（GM[15]）	回存	将计算结果回存到寄存器#1

<3>第 2 条指令： 900272 2000 0 1　（ADD SUM #1）
取指令操作同第 1 条指令，其余操作见表 5.3。
执行完后，PC = 8，（MCode[0]) = 1。

表 5.3　第 2 条指令的操作

MRgst[2] <— 0	计算有效地址	目的操作数有效地址为 1（主存代码段）
MRgst[3] <— 0		目的操作数在主存
MRgst[4] <— 1		源操作数有效地址为 1（寄存器#1）
MRgst[5] <— 1		源操作数在寄存器
GM[16] <— (MData[[MRgst[4]])	取值	从主存数据段传数到源操作数寄存器
GM[15] <— (MData[[MRgst[2]])		从主存数据段传数到目的操作数寄存器
GM[15] <— (GM[15]) + (GM[16])	执行	源操作数加到目的操作数寄存器
MData[[MRgst[2]] <—（GM[15]）	回存	将计算结果回存到主存数据段

<4>第 3 条指令：900961 2 1 0　（INC #1）
取指令操作同第 1 条指令，其余操作见表 5.4。
执行完后，PC = 12，（#1）= 2。

表 5.4　第 3 条指令的操作

MRgst[2] <— 1	计算有效地址	目的操作数有效地址为 1（寄存器#1）
MRgst[3] <— 1		目的操作数在寄存器

GM[15] <— (GM[[MRgst[2]])	取值	从寄存器#1 传数到目的操作数寄存器
GM[15] <— (GM[15]) +1	执行	目的操作数寄存器加 1
GM[MRgst[2]] <—（GM[15]）	回存	将计算结果回存到寄存器#1

<5>第 4 条指令：900332 3002 1 11 （CMP #1 11）
取指令操作同第 1 条指令，其余操作见表 5.5。

执行完后，PC = 16，flag = -1。

表 5.5 第 4 条指令的操作

MRgst[2] <— 1	计算有效地址	目的操作数有效地址为 1（寄存器#1）
MRgst[3] <— 1		目的操作数在寄存器
MRgst[4] <— 11		源操作数有效地址为 11（即立即数的值）
MRgst[5] <— 2		源操作数为立即数
GM[16] <— (MRgst[4])	取值	立即数直接传到数据寄存器
GM[15] <— (GM[[MRgst[2]])		从寄存器#1 传数到目的操作数寄存器
(GM[15]) - (GM[16])	执行	目的操作数寄存器与源操作数寄存器想减，结果为负数，状态寄存器设为-1。

<5>第 5 条指令： 900521 3 4 0 （JB L1）
取指令操作同第 1 条指令，其余操作见表 5.6。

执行完后，PC = 4（发生跳转）。

表 5.6 第 5 条指令的操作

MRgst[2] <— 4	计算有效地址	目的操作数有效地址为 4（即立即数的值）
MRgst[3] <— 2		目的操作数为立即数
GM[15] <— (MRgst[4])	取值	立即数直接传到数据寄存器
PC <— (GM[15])	执行	因为 flag<0，改变指令计数器 PC 的值

<6>由于发生了跳转，第 2~5 条指令将被反复执行，直到（#1）=11。此时，SUM 单元中的数值已为计算结果 55，即 1 加到 10 的和。

<7>第 6 条指令： 900000 （END）
执行到结束指令，程序执行结束。最终结果 55 保存在 SUM 单元中。

六、Xmips 操作实例

Xmips 系统目录下，主要包括

可执行文件	Xmips.exe	
配置文件	config.ini	
用户程序目录	file	
系统程序目录	sysfun	该目录下存放的系统函数库，用户不可修改。

除此之外，还有所有汇编产生的字符码文件，扩展名为 .co 。

下面将以程序 SUM 的运行为例，说明 Xmips 系统的使用方法。

6.1 路径设置和参数配置

1）路径设置

所有的用户汇编源文件都要保存到 Xmips\file 目录下。

运行列表文件 Xmips\file\run.list 登记了要汇编和运行的程序名。

run.list 文件的格式如下：

```
文件名 1
文件名 2
文件名 3
。。。。。。
文件名 N
end
```

ABC 汇编源程序对扩展名不做要求，但推荐使用 .abc 或 .cupa 。

必须要以“end”或“END”结尾，“end”或“END”之后的内容将不会再被读取。

要运行程序 SUM.cupa，run.list 内容如下

```
SUM.cupa
end
```

2）参数配置

目前 Xmips 系统提供的用户可以配置的参数有 4 个，分别是 delayMode，displayMode，reportLevel，updateSysfun 。

<1>delayMode——访存延时模拟

由于在实际的计算机中，访问主存比访问寄存器慢很多，故 Xmips 系统提供了访存延时模拟的功能。

delayMode = 0　　访存延时模拟关闭。即运行时不体现访存和访问寄存器的速度差异。
delayMode = 1　　访存延时模拟开启。

<2>displayMode——显示汇编源程序。
displayMode = 0　汇编过程中不显示源程序。
displayMode = 1　汇编过程中显示源程序。

<3>reportLevel——系统运行状况报告等级

Xmips 系统在运行的过程中，会显示系统的运行状况信息，如汇编源程序语法错误，中断方式，进程运行完毕等等。这些信息分为 4 个等级。具体信息内容及其等级见附录 B。

0　主要运行状况信息
1　次要运行状况信息
2　警告信息
4　错误信息

reportLevel = 0　　显示警告信息和错误信息。
reportLevel = 1　　显示主要运行状况信息，警告信息和错误信息。
reportLevel = 2　　显示 4 个等级的全部信息。

<4>updateSysfun——重新汇编用 ABC 汇编语言编写的系统函数库

系统函数库提供了一组用户可以通过软中断调用的 API，用户不可以修改。在汇编了系统函数之后，如果用户不删除在 Xmips 系统目录下出现的.co 文件，以后可以不再汇编系统函数。但如果删除了.co 文件，则必须重新汇编。

updateSysfun = 0　不重新汇编系统函数库。
updateSysfun = 1　重新汇编系统函数库。

这些参数信息保存在 Xmips/config.ini 文件中，用户可以修改。
下面是一个合法的 config.ini 文件的内容，运行程序 SUM 时，采用该配置。

```
delayMode=0
displayMode=1
reportLevel=2
updateSysfun=1
end
```

config.ini 文件有两点格式要求：
<1>每一行不能有空格，如“=”和参数的数值之间不可有空格。
<2>必须以 end 结尾。

6.2 编辑汇编源程序

要运行的用户源程序必须建立在 Ximps/file 目录下。
在该目录下建立 SUM.cupa，输入汇编源程序如下

```
DIM SUM 0
MOV #1 1
: L1
ADD SUM #1
INC #1
CMP #1 11
JB L1
END
```

注意，汇编源程序必须以 END 语句结尾。

保存源程序后，在 run.list 文件中登记要运行的源程序名，在 6.1 中已经提过，即：
run.list

```
SUM.cupa
end
```

6.3 运行 Xmips

双击 Xmips/Xmips.exe 运行或者命令行格式下运行 Xmips 系统，正常情况下，系统将很快给出程序的运行结果，如果出现内存访问非法或死循环，请检查相应文件的了路径设置，特别要注意是否相应文件都以 end 或 END 结尾。

如果按照 6.1 中配置 config.int 文件中的参数，系统的输出信息自上而下显示为
<1>系统函数库的汇编源程序
<2>用户要求运行的程序 SUM 的源程序
<3>进程调度信息
<4>运行结果，即 SUM 函数的数据段。
如果运行完全正确，数据段的第一个单元的值将为 55，这部分显示如下：

```
。。。。。。
process ID:50
memory ID:1
0     55
1     0
2     0
3     0
4     0
5     0
。。。。。。
```

七、Xmips 其他功能简介

设计 Xmips 的目标是模拟计算机系统的运行，其核心是实现一个虚拟的支持多道程序设计和可交互式的操作系统。其基本思想是虚拟化：包含 2 个方面：

<1>用一个线程模拟处理机；

<2>用多线程或者多进程程序模拟各个设备的并行工作，主要是处理机和外设的并行。

Xmips 指令可以在模拟的处理机上执行。模拟的各个进程在虚拟操作系统的控制下使用系统资源，包括处理机资源，外设资源和内存资源等等。

目前的 Xmips 系统只是部分实现了虚拟化的第一个方面，即当所有进程只是使用处理机资源时，可以实现并发执行，这种并发只是一种伪并行，实际上在同一时刻还是只有一个进程在运行，只不过通过分时等方式使各个进程轮换使用处理机。但是对于涉及到外设的操作，由于外设和处理机之间的并行是真正的并行，无法用单线程程序进行模拟，所以下一个目标是将当前的 Xmips 系统改造成为一个多线程程序，或者采用多进程的方式。

另一个问题是，目前的 Xmips 没有内存管理的机构（除了一个简单的进程控制块 PCB 管理机构）。所有进程实际直接使用其宿主系统，即 Windows XP 的内存，也就是说 Xmips 系统中实际上拥有一个无限大的内存空间，而每个进程的地址空间相对于对应于一个独立的存储器件，而且 Xmips 系统无法对其进行任何管理。所以完善系统的模拟内存管理机构，也是需要实现的。

除此之外，还应该为 Xmips 系统建立一个模拟的文件系统，以及一个可用于和用户交互的命令行环境，类似于 Linux 下的 shell。

本文主要叙述的 Xmips 指令系统实际上也是不完善的，比如说只支持单一数据类型，即 4B 的整型。同时 ABC 汇编语言的能力也十分有限，只能用于编写十分简单的程序，并且不支持宏扩展功能。所以 Xmips 系统的很多操作，特别是涉及到多进程的操作，都是直接采用 C++编写的程序实现的，这或许可以描述为 Xmips 系统有一个功能强大的模拟硬件系统，可是它的软件机构十分低能。所以，扩展 Xmips 系统的程序设计语言，逐步减少对 C++的依赖，也是改进 Xmips 的一个长远目标。

附录

A　Xmips 指令操作码和 ABC 汇编语言助记符对照表

操作符	功能	机器码
机器码助记符		
MOV	数据传送	900152
LEA	地址传送	902042
ADD	加	900272
INC	加 1	900961
SUB	减	901072
NEG	求相反数	901961
CMP	比较	900332
MUL	乘	902072
DIV	除	903072
MOD	取模	904072
AND	与	905072
OR	或	906072
NOT	非	902961
XOR	异或	907072
JA	大于跳转	900421
JB	小于跳转	900521
JMP	无条件跳转	900621
JE	等于跳转	900721
JNE	不等于跳转	900821
PUSH	压栈	902121
POP	出栈	902241
PUSHA	寄存器组压栈	902300
POPA	寄存器组出栈	902400
CALL	函数调用	904121
RET	函数调用返回	904200
INT	软中断	904000
IRET	中断返回	904300
CLI	关中断	960000
STI	开中断	960100
SYSR	读系统共享区	950000
SYSW	写系统共享区	950100
WAKE	唤醒进程	950200
SET	设置通信标识	950300
END	结束	900000
HALT	停机	800000
伪指令		
DIM	变量定义	910002

LOC	同 DIM，已不用	910002
PROC	函数定义	920001
:	标号	920001
~	注释	3
CMNT	注释	3
$	占位空语句	0

B　系统运行状态信息表

状态信息格式{RES, Level, Message}

RES　　状态信息编号

Level　　状态等级，包括

0　主要运行状况信息

1　次要运行状况信息

2　警告信息

4　错误信息

Message　　提示信息

其中 Class::Function 表示发出该提示信息的函数

{0, 0, "SYSTEM: normal end"},
{10, 3, "memory::read: memory overflow"},
{11, 3, "memory::write: memory overflow"},
{12, 3, "editor::editorData: illegal data"},
{13, 3, "editor::editorCollection: illegal variable name"},
{14, 3, "editor::editorCollection: illegal opcode"},
{15, 3, "editor::editorCollection: too many content"},
{16, 3, "stack::push: stack is full"},
{17, 3, "stack::pop: stack is empty"},
{18, 0, "memory::update: update memory size"},
{19, 3, "editor::editorCollection: error editor num"},
{20, 1, "interpreter::exer: interrupt occured"},
{21, 3, "interpreter::exer: interpreter overflow"},
{22, 1, "interpreter::exer: interpreter reaches cycleTimes"},
{23, 0, "interpreter::exer: process execute completed"},
{24, 3, "interpreter::exer: property set illegal"},
{51, 3, "assembler::trans2: \'@\' use error"},
{52, 3, "assembler::trans2: undeclared indentifier"},
{53, 3, "assembler::trans2: \'&\' use error"},
{54, 3, "assembler::trans2: register used in indirect addressing"},
{55, 3, "assembler::trans2: \'#\' use error"},
{56, 3, "assembler::trans2: other char occur before immediate num"},
{57, 3, "assembler::trans2: \'!\' use error"},
{58, 3, "assembler::trans2: over the max general register number(14)"},
{60, 3, "queue::enQueue: queue is full"},
{61, 2, "queue::deQueue: queue is empty"},
{80, 3, "storage::getFile: visit way error"},
{81, 3, "storage::getFile: file open failed"},
{82, 3, "storage::releaseFile: file pointer is null"},
{90, 1, "sysList::copy: a system function is copied"},
{130, 2, "pcbList::enQueue: list is empty"},
{131, 3, "pcbList::get: list is empty"},

{132, 3, "pcbList::get: pcb not found"},
{170, 0, "dispatcher::swap2: process swaps to run"},
{171, 0, "dispatcher::swap2: process swaps from run to ready"},
{172, 0, "dispatcher::swap2: process swaps to wait"},
{173, 1, "dispatcher::swap2: a system call happened"},
{174, 1, "dispatcher::swap2: a system function completed"},
{175, 0, "dispatcher::swap2: process swaps from wait to ready"},
{176, 0, "dispatcher::swap2: a system function swaps to ready"},
{177, 0, "dispatcher::swap2: a process finished"},
{178, 0, "dispatcher::swap2: a process failed"},
{179, 1, "dispatcher::swap2: swap is completed"},
{180, 3, "dispatcher::swap2: access violation to system share region"},
{181, 1, "dispatcher::swap2: read system share"},
{182, 1, "dispatcher::swap2: write system share"},
{183, 3, "dispatcher::pcbManagement: no pcb can use now"},
{184, 3, "dispatcher::loader: pcb dispatch failed"},
{185, 0, "dispatcher::swap2: a process suspended"},
{186, 0, "dispatcher::swap2: a process woken"},
{187, 3, "dispatcher::swap2: access violation to wake up a process"},
{188, 1, "dispatcher::swap2: insert back to the head of ready queue"},
{189, 1, "dispatcher::swap2: interruption ban checked"},
{190, 3, "dispatcher::swap2: over the range of wait queue numbers"},
{191, 3, "editor::editorFromFile: initial value more than declared"},
{192, 3, "editor::editorFromFile: illegal declaration"},
{193, 3, "editor::editorFromFile: tag repeated"},
{194, 3, "editor::editorFromFile: tag not find"},
{195, 3, "editor::ASM: open file failed"},
{196, 3, "editor::editorFromFile: indentifier repeated"},
{197, 3, "editor::editorFromFile: immediate num cant be 1st operand in double operands instruction"}