

数字逻辑与处理器基础

MIPS汇编编程实验

实验指导书

2021年 春季学期

陈佳煜 黄成宇

目录

- MARS环境安装与基础使用方法
- 实验内容一：基础练习
 - 循环分支
 - 系统调用
 - 数组指针
 - 函数调用
- 实验内容二：综合练习
 - 背包问题
- 参考资料

安装JRE

- 运行JAVA程序包需要运行环境：Java Runtime Environment (JRE)
- Windows系统运行 JavaSetup8u241.exe按提示进行安装。
- 如果安装中又遇到问题，或者其他操作系统可以访问JAVA官网：
https://www.java.com/zh_CN/，下载完成后按提示进行安装。

报告问题

访问包含 Java 应用程序的
页时为什么始终重定向到此
页？

» [了解详细信息](#)

⚠ Oracle Java 许可重要更新

从 2019 年 4 月 16 起的发行版更改了 Oracle Java 许可。

新的适用于 Oracle Java SE 的 Oracle 技术网许可协议 与以前的 Oracle Java 许可有很大差异。新许可允许某些免费使用（例如个人使用和开发使用），而根据以前的 Oracle Java 许可获得授权的其他使用可能会不再支持。请在下载和使用此产品之前认真阅读条款。可在此处查看常见问题解答。

可以通过低成本的 [Java SE 订阅](#) 获得商业许可和技术支持。

Oracle 还在 [jdk.java.net](#) 的开源 [GPL 许可](#) 下提供了最新的 OpenJDK 发行版。

免费 Java 下载

» [什么是 Java?](#) » [我有 Java 吗?](#) » [是否需要帮助?](#)

负责助教：陈佳煜、黄成宇

邮箱：jiayu-ch19@mails.tsinghua.edu.cn

huang-cy20@mails.tsinghua.edu.cn

运行MARS

- 如果JRE正确安装， 双击Mars4_5.jar即可打开MARS仿真器。
- 如果打不开， 先检查JRE是否安装正确， 可以考虑重新安装。
- 如果是软件包的问题可以进入MARS官网下载。
<https://courses.missouristate.edu/KenVollmar/mars/download.htm>
- 接下来将用example_0.asm作为例子演示MARS的用法。

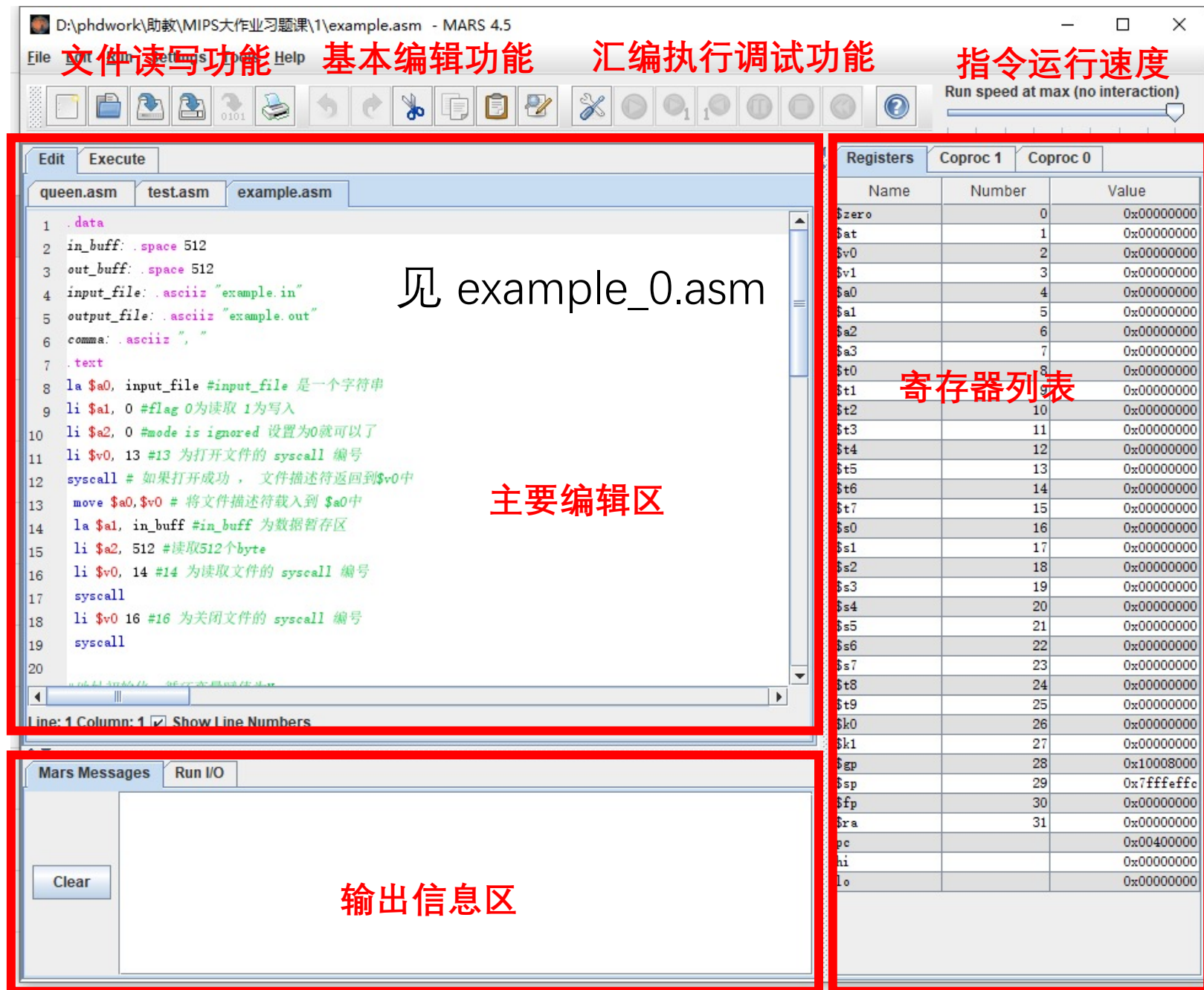
运行MARS

运行MARS后的主要界面如图所示。

主要编辑区用于编写汇编指令。

输出信息区可以查看程序运行过程中的输出和系统报错等。

寄存器列表实时显示当前运行状态下各个寄存器存储的值。



汇编运行

首先打开汇编文件
example_0.asm

点击**汇编**按钮即可切换到
执行页面，源代码汇
编成基础指令和机器码，
PC置为0x00400000，
并等待执行。

执行页面内可以看到汇
编后的**基础指令**和对
应的**机器码**，每条指令的
指令地址。

D:\phdwork\助教\MIPS大作业习题课\1\example.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

指令断点 **地址** **机器码** **基础指令** **源代码**

Bkpt	Address	Code	Basic	Source Code
<input type="checkbox"/>	0x00400000	0x3c011001	lui \$1, 0x00001001	8: la \$a0, input_file #input_file 是...
<input type="checkbox"/>	0x00400004	0x34240400	ori \$4, \$1, 0x00000400	
<input type="checkbox"/>	0x00400008	0x24050000	addiu \$5, \$0, 0x00000000	9: li \$a1, 0 #flag 0为读取 1为写入
<input type="checkbox"/>	0x0040000c	0x24060000	addiu \$6, \$0, 0x00000000	10: li \$a2, 0 #mode is ignored 设置为0就...
<input type="checkbox"/>	0x00400010	0x2402000d	addiu \$2, \$0, 0x0000000d	11: li \$v0, 13 #13 为打开文件的 syscall ...
<input type="checkbox"/>	0x00400014	0x0000000c	syscall	12: syscall # 如果打开成功, 文件描述符...
<input type="checkbox"/>	0x00400018	0x00022021	addu \$4, \$0, \$2	13: move \$a0, \$v0 # 将文件描述符载入到 \$...
<input type="checkbox"/>	0x0040001c	0x3c011001	lui \$1, 0x00001001	14: la \$a1, in_buff #in_buff 为数据暂存区
<input type="checkbox"/>	0x00400020	0x34250000	ori \$5, \$1, 0x00000000	
<input type="checkbox"/>	0x00400024	0x24060200	addiu \$6, \$0, 0x00000200	15: li \$a2, 512 #读取512个byte

Memory查看器

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
0x10010000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) ☒ Hexadecimal Addresses ☒ Hexadecimal Values

0x10000000 (.extern)
0x10010000 (.data)
0x10040000 (heap)
current \$gp
current \$sp
0x00400000 (.text)
0x90000000 (.kdata)
0xffff0000 (MMIO)

可以选择查看哪一段地址

Mars Messages Run I/O

Clear Reset: reset completed.

Registers Coproc 1 Coproc 0

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

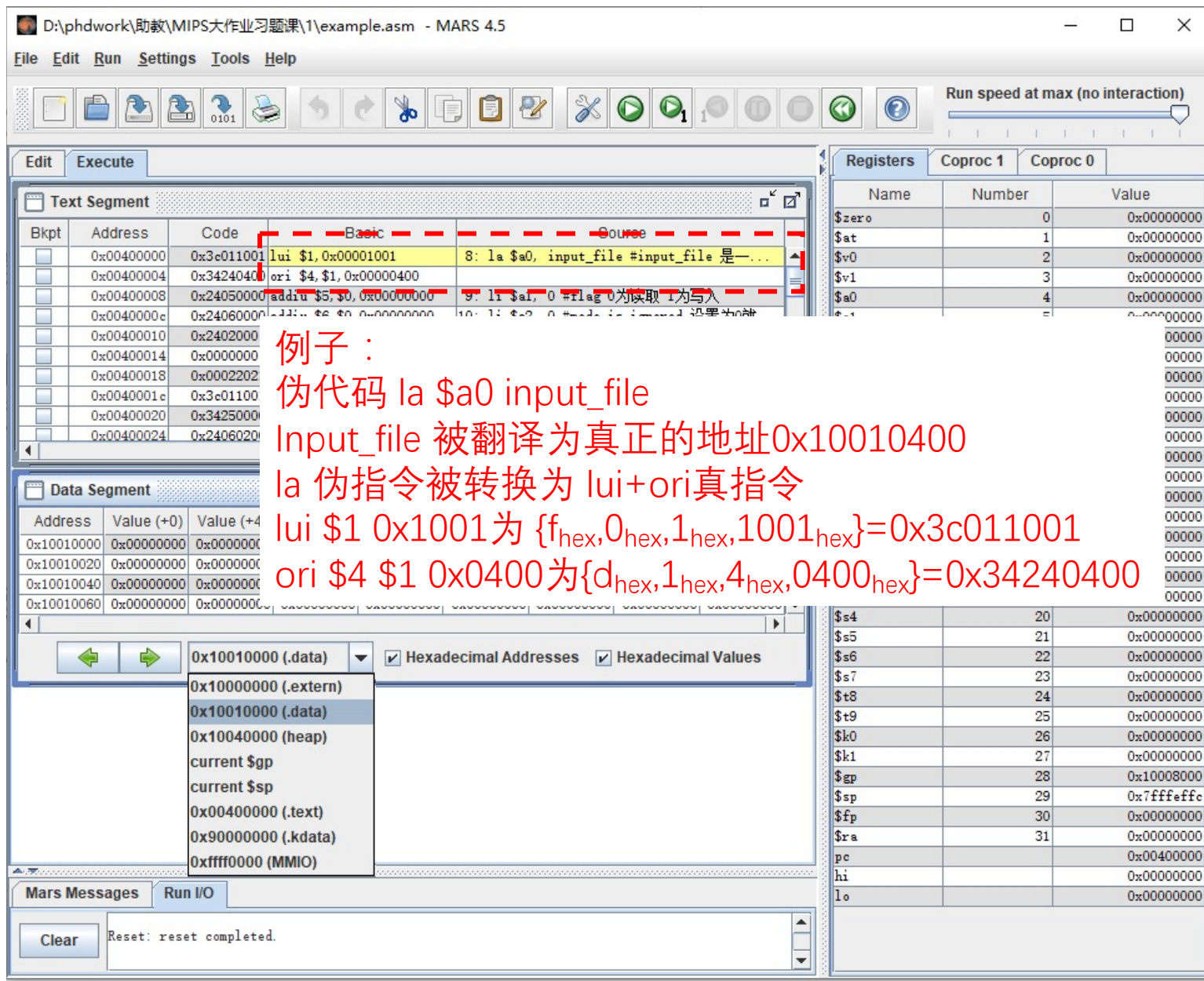
汇编运行

源代码：用户编写的汇编代码，包括标记，伪代码等。

基础指令：汇编后的指令，伪代码被转换，标记被翻译。

地址&机器码：与基础指令一一对应，32bit一条指令，地址依次加四。

断点：调试用，当执行到这一句时暂停。



D:\phdwork\助教\MIPS大作业习题课\1\example.asm - MARS 4.5

File Edit Run Settings Tools Help

Run speed at max (no interaction)

Registers

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$a4	8	0x00000000
\$a5	9	0x00000000
\$a6	10	0x00000000
\$a7	11	0x00000000
\$s0	12	0x00000000
\$s1	13	0x00000000
\$s2	14	0x00000000
\$s3	15	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

Text Segment

Bkpt	Address	Code
	0x00400000	0x3c011001 lui \$1, 0x0001001
	0x00400004	0x34240400 ori \$4, \$1, 0x00000400
	0x00400008	0x24050000 addiu \$5, \$0, 0x00000000
	0x0040000c	0x24060000 addiu \$6, \$0, 0x00000000
	0x00400010	0x24020000 addiu \$7, \$0, 0x00000000
	0x00400014	0x00000000
	0x00400018	0x0002202
	0x0040001c	0x3c01100
	0x00400020	0x3425000
	0x00400024	0x2406020

Data Segment

Address	Value (+0)	Value (+4)
0x10010000	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000

0x10010000 (.data)

- 0x10000000 (.extern)
- 0x10010000 (.data)
- 0x10040000 (heap)
- current \$gp
- current \$sp
- 0x00400000 (.text)
- 0x90000000 (.kdata)
- 0xffff0000 (MMIO)

Hexadecimal Addresses Hexadecimal Values

Mars Messages Run I/O

Clear Reset: reset completed.

例子：
伪代码 la \$a0 input_file
Input_file 被翻译为真正的地址0x10010400
la 伪指令被转换为 lui+ori真指令
lui \$1 0x1001为 {f_{hex}, 0_{hex}, 1_{hex}, 1001_{hex}} = 0x3c011001
ori \$4 \$1 0x0400为 {d_{hex}, 1_{hex}, 4_{hex}, 0400_{hex}} = 0x34240400

汇编运行

执行：从第一条指令开始连续执行直到结束。

单步执行：执行当前指令并跳转到下一条。

单步后退：后退到最后一条指令执行前的状态（包括寄存器和memory）

暂停&停止：在连续执行的时候可以停下来，一般配合较慢的指令运行速度，不用于调试。**调试最好使用断点功能。**

重置：重置所有寄存器和memory。

汇编 执行 单步执行 单步后退 暂停 停止 重置

指令运行速度

黄色指令代表当前指令
是即将执行但尚未执行的指令
也是pc寄存器对应的指令

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

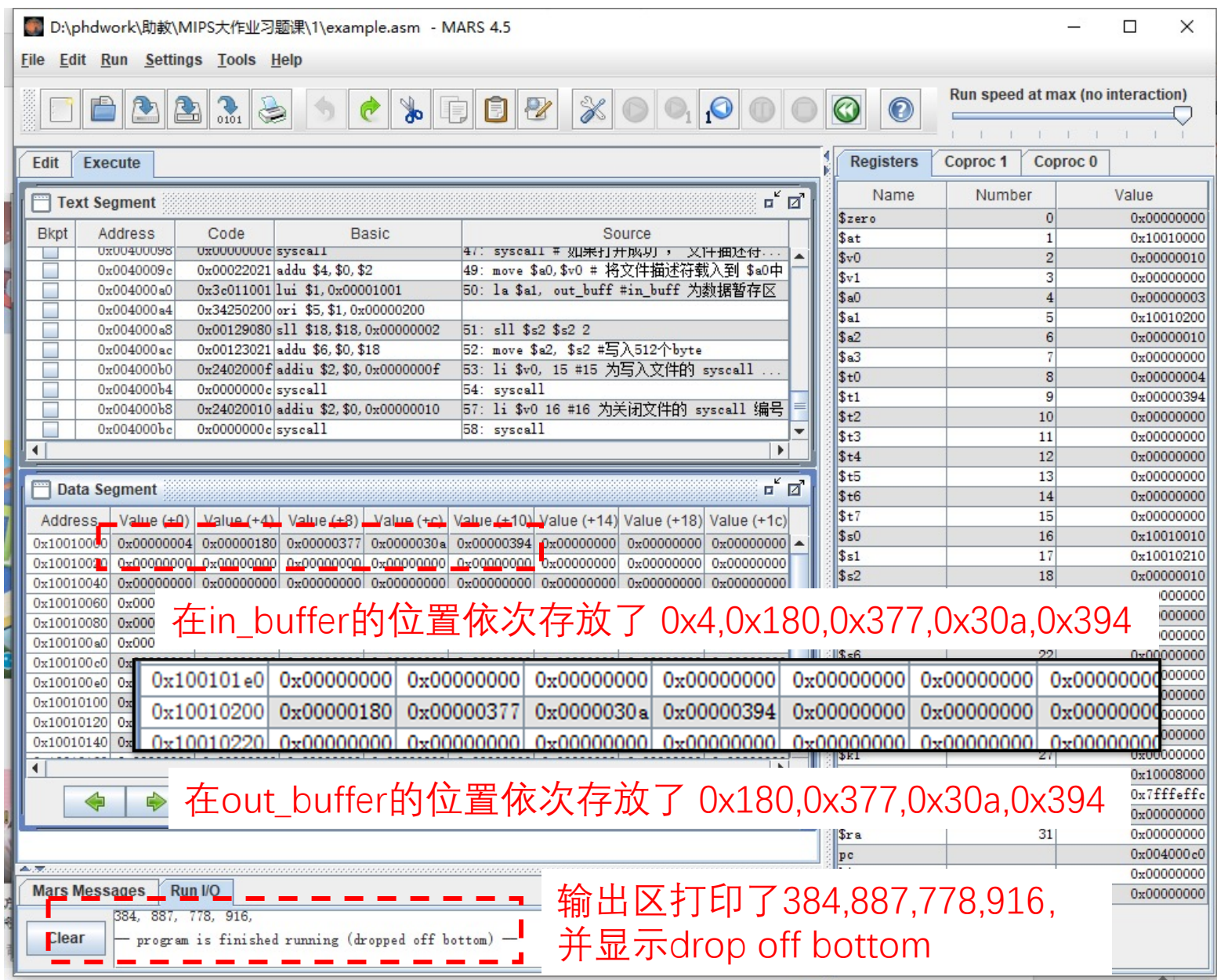
汇编运行

点击执行按钮后，所有指令执行完毕。

可以看到各个寄存器内的值发生了变化。

Memory中in_buffer, out_buffer地址对应的数据发生变化。

输出区正确打印了对应的数据并提示，程序执行完地址最大的指令并且没有后续指令了（drop off bottom）



汇编运行

38 la \$a0, comma

在38行的指令处设置断点。并点击运行按钮两次，程序停在该位置。

可以看到程序向out_buffer中写入两个数，也向输出区打了两个数，各个寄存器也停留在对应状态。

在打印某个整数和打印逗号之间的指令设置断点

在in_buffer的位置依次存放了 0x4,0x180,0x377,0x30a,0x394

在out_buffer的位置依次存放了0x180,0x377

输出区打印了384,887

PC取值为0x00400070

example_0.asm 内包含一个从文件读取数据并写入另一个文件的例子

数据声明，此部分数据存在0x10010000

```
.data
in_buff: .space 512
out_buff: .space 512
input_file: .asciiz "example.in"
output_file: .asciiz "example.out"
comma: .asciiz ", "
```

```
.text
la $a0, input_file #input_file 是一个字符串
li $a1, 0 #flag 0为读取 1为写入
li $a2, 0 #mode is ignored 设置为0就可以了
li $v0, 13 #13 为打开文件的 syscall 编号
syscall # 如果打开成功，文件描述符返回到
move $a0, $v0 # 将文件描述符载入到 $a0中
la $a1, in_buff #in_buff 为数据暂存区
li $a2, 512 #读取512个byte
li $v0, 14 #14 为读取文件的 syscall 编号
syscall
li $v0, 16 #16 为关闭文件的 syscall 编号
syscall
```

打开读取文件，并将数据写入in_buff

初始化变量

```
la $s0, in_buff
la $s1, out_buff
lw $s2, 0($s0)
li $t0, 0
```

循环体

```
#地址加4 循环变量减1
for: addi $s0, $s0, 4
addi $t0, $t0, 1
lw $t1, 0($s0)
sw $t1, 0($s1)
addi $s1, $s1, 4
#打印整数
move $a0, $t1
li $v0, 1
syscall
#打印逗号
la $a0, comma
li $v0, 4
syscall
bne $t0, $s2, for
```

跳转条件

打开文件并将out_buff的数据写入

```
la $a0, output_file #output_file 是一个字符串
li $a1, 1 #flag 0为读取 1为写入
li $a2, 0 #mode is ignored 设置为0就可以了
li $v0, 13 #13 为打开文件的 syscall 编号
syscall # 如果打开成功，文件描述符返回到
move $a0, $v0 # 将文件描述符载入到 $a0中
la $a1, out_buff #in_buff 为数据暂存区
sll $s2, $s2, 2
move $a2, $s2 #写入512个byte
li $v0, 15 #15 为写入文件的 syscall 编号
syscall
#此时$a0 中的文件描述符没变
#直接调用 syscall 16 关闭它
li $v0, 16 #16 为关闭文件的 syscall 编号
syscall
```

汇编基本结构

见 example_1.asm

- 数据段
 - 以 “.data” 记号开头。
 - 包括常量数据和固定数组的声明。
- 代码段
 - 以 “.text” 记号开头。
 - 包括待执行的代码和行标记。
- 注释
 - “#” 为注释标记。
 - 可以出现在任意位置。
 - “#” 及其之后的所有内容均被忽略。

```
.data
    string: .asciiz "Hello World!\n"

.text
main:
    la $a0 string #载入字符串地址
    li $v0 4      #4代表打印字符串
    syscall       #执行系统调用

    li $v0 17     #17代表exit
    syscall       #执行系统调用
```


汇编基本结构

`.align x`

将下一个数据项对齐到特定的byte边界。如果要读取的数据是以2byte, 4byte, 8byte为单位的, 需要对齐到对应的边界上。x取值0表示1byte, 1表示2byte, 2表示4byte, 3表示8byte。

`.ascii, .asciiz`

表示字符串, 其中asciiz会自动在最后补上null字符。

`.byte, .half, .word`

表示数组常量按1byte, 2byte, 4byte存储

`.space`

表示一个以byte计长度的数组

见 example_2.asm

`.data`

```
stringz: .asciiz "Hello World!\n"
string: .ascii "Hello World!\n"
#让array对齐到4byte边界
.align 4 #没有这句话可能会出错
array: .space 512
#以下常数数组会自动对齐到对应边界
barray: .byte 1,2,3,4
harray: .half 1,2,3,4
warray: .word 1,2,3,4
```

代码段基本结构

代码段一般由若干段顺序排列的指令序列构成。从第一条指令开始执行。每执行完一条指令后会顺序执行下一条指令，除非发生跳转

label_name : add \$0 \$1 \$2

字符串+冒号代表标记，可以用在对应指令同一行开头或者对应指令上一行。

一个函数一般以函数名为第一条指令的label（函数入口）。程序内一般包括入栈，程序主体，设置返回值，出栈，返回上一级程序。

见 example_3.asm

主过程

```
.text
main:
    li $v0 5      #5代表读入一个整数
    syscall      #执行系统调用
    move $a0 $v0  #将读入的整数作为第一个参数
    li $v0 5      #5代表读入一个整数
    syscall      #执行系统调用
    move $a1 $v0  #将读入的整数作为第二个参数

    jal product  #跳转到子过程product
    move $a0 $v0  #将返回值赋给$a0
    li $v0 1      #1代表打印一个整数
    syscall      #执行系统调用

    li $v0 17     #17代表exit
    syscall      #执行系统调用
```

代码段基本结构

代码段一般由若干段顺序排列的指令序列构成。从第一条指令开始执行。每执行完一条指令后会顺序执行下一条指令，除非发生跳转

label_name : add \$0 \$1 \$2

字符串+冒号代表标记，可以用在对应指令同一行开头或者对应指令上一行。

一个函数一般以函数名为第一条指令的label（函数入口）。程序内一般包括入栈，程序主体，设置返回值，出栈，返回上一级程序。

见 example_3.asm

子过程，接上页

```
product:
    move $t0 $a0  #将第一个参数赋给t0作为累加值
    move $t1 $a1  #将第二个参数赋给t1作为计数器
    move $t2 $zero #结果清零
loop:   add $t2 $t2 $t0  #结果累加t0
        addi $t1 $t1 -1  #计数器减一
        bnez $t1 loop    #如果计数器不为零循环继续
    move $v0 $t2  #将结果赋给返回值
    jr $ra        #跳转回上一级程序
```

实验内容1

- 用MIPS32汇编指令完成下列任务， 调试代码并获得正确的结果。
- 练习1-1：循环， 分支。
- 练习1-2：系统调用。
- 练习1-3：数组， 指针。
- 练习1-4：函数调用。

练习1-1：循环, 分支

**1、用MIPS语言实现
exp1_1_loop.cpp中的功能并
提交汇编代码，尽量在代码中
添加注释。**

- exp1_1_loop.cpp代码内容主要包括：
- 将输入值取绝对值，存在变量i, j中
- 取i, j中的较小值，存在j中
- 求和sum=0+1+2+……+j

exp1_1_loop.cpp文件内容

```
#include "stdio.h"
int main()
{
    int i,j,temp,sum=0;
    scanf("%d",&i);
    scanf("%d",&j);
    if (i<0){i=-i;}
    if (j<0){j=-j;}
    if (j>i)
    {
        temp = i;
        i = j;
        j = temp;
    }
    for (temp=0;temp<=j;temp++)
    {
        sum += temp;
    }
    printf("%d",sum);
    return 0;
}
```

练习1-2：系统调用

- 练习使用MARS模拟器中的系统调用syscall，使用syscall可以完成包括文件读写，命令行读写（标准输入输出），申请内存等辅助功能。
- 系统调用基本的使用方法是
 1. 向\$a*寄存中写入需要的参数（如果有）
 2. 向\$v0寄存器中写入需要调用的syscall的编号
 3. 使用” syscall” 指令进行调用
 4. 从\$v0中读取调用的返回值（如果有）
- 更多具体的使用方法可以参照MARS模拟器的Help中的相关内容。

练习1-2：系统调用

- 用MIPS汇编指令实现exp1_2_sys_call.cpp 的功能并提交汇编代码，尽量在代码中添加注释。
- exp1_2_sys_call.cpp代码内容主要包括：
 - 申请一个8byte整数的内存空间。
 - 从” a.in” 读取两个整数。
 - 从键盘输入一个整数。
 - 对上面 三个整数求最大值。
 - 向屏幕打印最大值结果。
 - 向” a.out” 写入最大值结果。

```
#include "stdio.h"
int main()
{
    FILE * infile ,*outfile;
    int i,max_num=0;
    int* buffer;
    buffer = new int[2];
    infile = fopen("a.in","rb");
    fread(buffer, 4, 2, infile);
    fclose(infile);
    scanf("%d",&i);
    max_num = i;
    for(id=0,id<2,++id)
    {
        if(max_num < buffer[id])
        {max_num = buffer[id];}
    }
    printf("%d",max_num);
    outfile = fopen("a.out","wb");
    fwrite(max_num, 4, 1, outfile);
    fclose(outfile);
    return 0;
}
```

exp1_2_sys_calll.cpp文件内容 }

练习1-3：数组、指针

- 用MIPS汇编指令实现exp1_3_array.cpp 的功能并提交汇编代码，尽量在代码中添加注释。
- exp1_3_array.cpp代码内容主要包括：
 - 输入数组a的长度n
 - 任意输入n个整数
 - 将数组a逆序，并且仍然存储在a中
 - 打印数组a的值

exp1_3_array.cpp文件内容

```
#include "stdio.h"
#include <stdio.h>
int main()
{
    int *a, n, i, t;
    scanf("%d",&n);
    a = new int [n];
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    for(i=0;i<n/2;i++)
    {
        t = a[i];
        a[i] = a[n-i-1];
        a[n-i-1] = t;
    }
    for(i=0;i<n;i++) printf("%d ",a[i]);
    return 0;
}
```


练习1-4：函数调用

• 调用函数A的流程为：

1. 设置参数寄存器
\$a0~\$a3。
2. 使用jal跳转到被调函数B。
3. 使用被调函数的返回值
\$v0~\$v1执行接下来的内容。

寄存器编号	助记符	用法
0	zero	永远为0
1	at	用做汇编器的临时变量
2-3	v0, v1	用于过程调用时返回结果
4-7	a0-a3	用于过程调用时传递参数
8-15	t0-t7	临时寄存器。在过程调用中被调用者不需要保存与恢复
24-25	t8-t9	
16-23	s0-s7	保存寄存器。在过程调用中被调用者一旦使用这些寄存器时，必须负责保存和恢复这些寄存器的原值
26,27	k0,k1	通常被中断或异常处理程序使用，用来保存一些系统参数
28	gp	全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量
29	sp	堆栈指针
30	fp	帧指针
31	ra	返回地址

练习1-4：函数调用

• 被调函数B的流程为：

1. 分配栈空间（ $\$sp - 4 * n$ ）。
2. 将需要保存的寄存器存入栈。
3. 使用输入参数 $\$a0 \sim \$a3$ 执行函数内容并将结果存入返回寄存器 $\$v0 \sim \$v1$ 。
4. 将入栈的数据恢复到寄存器。
5. `j $ra` 返回上级函数。

寄存器编号	助记符	用法
0	zero	永远为0
1	at	用做汇编器的临时变量
2-3	v0, v1	用于过程调用时返回结果
4-7	a0-a3	用于过程调用时传递参数
8-15	t0-t7	临时寄存器。在过程调用中被调用者不需要保存与恢复
24-25	t8-t9	
16-23	s0-s7	保存寄存器。在过程调用中被调用者一旦使用这些寄存器时，必须负责保存和恢复这些寄存器的原值
26,27	k0,k1	通常被中断或异常处理程序使用，用来保存一些系统参数
28	gp	全局指针。一些运行系统维护这个指针来更方便的存取static和extern变量
29	sp	堆栈指针
30	fp	帧指针
31	ra	返回地址

逐步完成函数调用的编译

```
int Fib(int n) {  
    s0 = n;  
    if(s0<3)  
        return 1;  
    else {  
        s1 = 0;  
        s1 += Fib(s0-1);  
        s1 += Fib(s0-2);  
        return s1;}  
}
```

1. 先编译其他语句。
2. 拆解编译调用函数的语句
3. 在首尾分别补上入栈和出栈

```
Fib(4)  
= Fib(3) + Fib(2)  
= Fib(3) + 1  
= Fib(2) + Fib(1) + 1  
= Fib(2) + 1 + 1  
= 1 + 1 + 1 = 3
```

五次函数调用

```
Main -> Fib(4),  
Fib(4) -> Fib(3) + Fib(2),  
Fib(3) -> Fib(2) + Fib(1)
```

<code>int Fib(int n) {</code>	Fib:#将参数放入\$a0 (保护现场)
<code> s0 = n;</code>	<code>addi \$s0 \$a0 0</code>
<code> if(s0<3)</code>	<code>slti \$t0 \$s0 3</code>
<code> return 1;</code>	<code>beqz \$t0 Next</code> <code>addi \$v0 \$0 1 # 返回1</code> (恢复现场)
	<code>jr \$ra</code>
<code> else{</code>	Next:
<code> s1 = 0</code>	<code>addi \$s1 \$0 0</code>
<code> s1 += Fib(s0-1)</code>	(调用Fib(n-1)) <code>add \$s1 \$v0 \$s1</code>
<code> s1 += Fib(s0-2)</code>	(调用Fib(n-2)) <code>add \$s1 \$v0 \$s1</code>
<code> return s1;}</code>	<code>addi \$v0 \$s1 0</code> (恢复现场)
<code>}</code>	<code>jr \$ra</code>

练习1-4：函数调用

- 继续完成Fib过程的编译，使得其可以完成计算 $Fib(n)$ 的任务。
(在实验报告中完成即可，不需要提交相应汇编程序)

实验内容2-背包问题

- 给定 n 个重量为 w_1, w_2, \dots, w_n , 价值为 v_1, v_2, \dots, v_n 的物品和一个承重量为 W 的背包, 如果选择一些物品放到背包中, 求使得背包中物品价值最大的方案。
- 例: 背包总重量 $W=5$

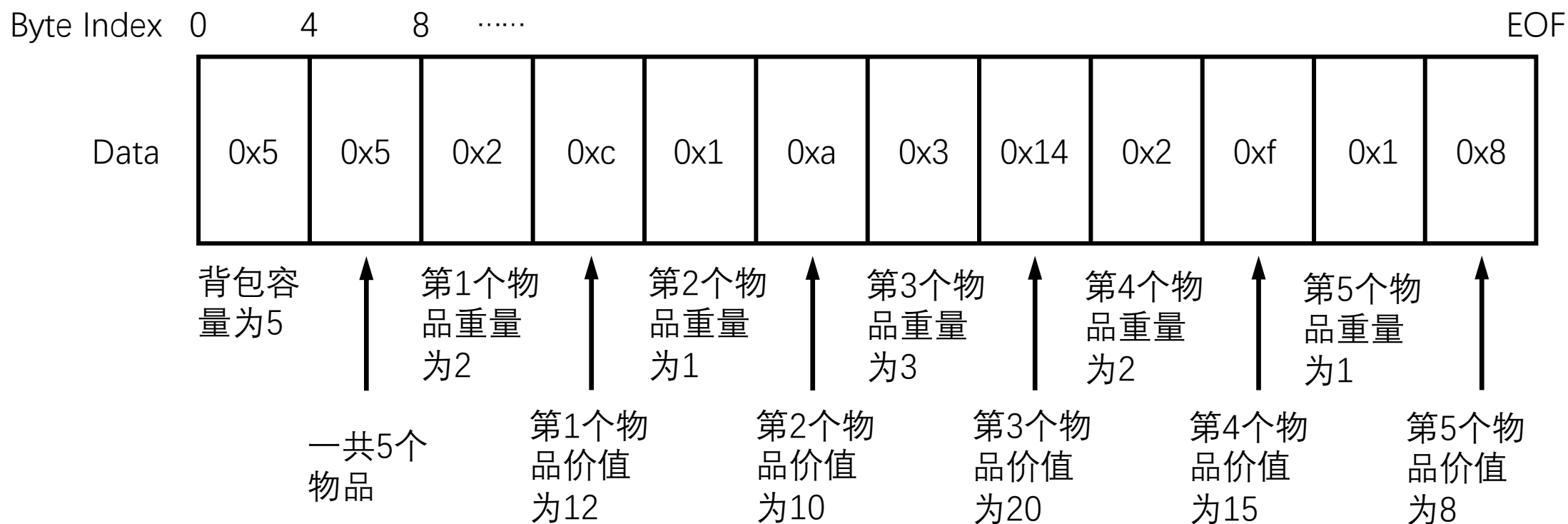
序号	1	2	3	4	5
重量	2	1	3	2	1
价值	12	10	20	15	8

最优方案: 物品2+物品3+物品5, 重量5, 价值38

实验内容2-背包问题

- 输入文件格式

输入文件为二进制文件。文件中第一个4Bytes表示背包的容量，第二个4Bytes表示总共的物品数量，之后的数据物品重量和价值交错排列，如下图。



实验内容2-背包问题

- 用MIPS32汇编指令将以下C语言代码转化为汇编语言执行，调试代码并获得正确的结果。测试样例被限制在最大背包总重量小于63，物品数量小于31。要求汇编程序结束时，计算结果储存在寄存器\$v0中。
- 动态规划-自底向上
 - 熟悉基本操作，练习文件读取写入。
- 遍历搜索
 - 练习位的相关操作。
- 动态规划-自顶向下
 - 练习递归函数调用，入栈出栈等操作。

C代码编译时，注意将测试数据与代码放在同一文件夹下

exp2_1 自底向上动态规划-C代码

```
#include <stdio.h>

typedef struct{
    int weight;  定义Item结构体
    int value;
}Item;

int knapsack_dp_loop(int item_num, Item* item_list, int knapsack_capacity);

int main() {    主函数
    FILE* infile;
    int in_buffer[512], item_num, knapsack_capacity;
    infile = fopen("test.dat", "rb");          读入二进制文件
    fread(in_buffer, sizeof(int), 512, infile);
    fclose(infile);
    knapsack_capacity = in_buffer[0]; 第1个byte存储背包重量
    item_num = in_buffer[1];          第2个byte存储物品数量
    Item* item_list = (Item*)(in_buffer + 2); 第3个byte开始存储物品的重量和价值
    printf("%d\n", knapsack_dp_loop(item_num, item_list, knapsack_capacity)); 调用函数求解问题
    return 0;
}
```

注意C语言实现输出了结果，但是汇编程序要求将结果写在\$V0中

接上页

exp2_1 自底向上动态规划-C代码

```
#define MAX_CAPACITY 63
```

函数实现

```
int knapsack_dp_loop(int item_num, Item* item_list, int knapsack_capacity){  
    int cache_ptr[MAX_CAPACITY + 1] = {0}; 定义cache用于存储子问题结果  
    for(int i = 0; i < item_num; ++i){  
        int weight = item_list[i].weight; 针对前i个物品的子问题进行循环  
        int val = item_list[i].value;  
        for(int j = knapsack_capacity; j >= 0; --j){  
            if(j >= weight){  
                cache_ptr[j] =  
                    (cache_ptr[j] > cache_ptr[j - weight] + val)?  
                    cache_ptr[j]:  
                    cache_ptr[j - weight] + val;  
            }  
        }  
    }  
    return cache_ptr[knapsack_capacity];  
}
```

状态转移：

$$OPT(i, w) = \max\{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\}$$

exp2_2 遍历搜索-C代码

```
#include <stdio.h>
```

```
typedef struct{
```

```
    int weight;  定义Item结构体  
    int value;
```

```
}Item;
```

```
int knapsack_search(int item_num, Item* item_list, int knapsack_capacity);
```

```
int main(){      主函数
```

```
    FILE* infile;
```

```
    int in_buffer[512], item_num, knapsack_capacity;
```

```
    infile = fopen("test.dat", "rb");
```

读入二进制文件

```
    fread(in_buffer, sizeof(int), 512, infile);
```

```
    fclose(infile);
```

```
    knapsack_capacity = in_buffer[0]; 第1个byte存储背包重量
```

```
    item_num = in_buffer[1];          第2个byte存储物品数量
```

```
    Item* item_list = (Item*)(in_buffer + 2); 第3个byte开始存储物品的重量和价值
```

```
    printf("%d\n", knapsack_search(item_num, item_list, knapsack_capacity)); 调用函数求解问题  
    return 0;
```

```
}
```

注意C语言实现输出了结果，但是汇编程序要求将结果写在\$V0中

接上页

exp2_2 遍历搜索-C代码

所有可能的状态可以被编码为一个长度为item_num的二进制码。0/1表示物品不在/在背包中，总共有 $2^{\text{item_num}}$ 种状态。可以用 $0 \sim 2^{\text{item_num}} - 1$ 的数表示每一个状态，进行遍历搜索。

```
int knapsack_search(int item_num, Item* item_list, int knapsack_capacity){  
    int val_max = 0;  初始化最大价值为0  
    for(int state_cnt = 0; state_cnt < (0x1 << item_num); ++state_cnt){  
        int weight = 0;  
        int val = 0;  
        for(int i = 0; i < item_num; ++i){  
            int flag = (state_cnt >> i) & 0x1;  
            weight = flag? (weight + item_list[i].weight): weight;  
            val = flag? (val + item_list[i].value): val;  
        }  
        if(weight <= knapsack_capacity && val > val_max) val_max = val;  
    }  
    return val_max;  
}
```

函数实现

循环遍历 $0 \sim 2^{\text{item_num}} - 1$ 表示的状态

利用位运算判断该状态表示的第i个物品是否在背包中

计算状态对应总的物品重量和价值

如果物品总重量小于背包总重量且价值大于最大价值，更新最大价值

exp2_3 自顶向下动态规划-C代码

主函数

```
#include <stdio.h>
```

```
typedef struct{
```

```
    int weight;    定义Item结构体
```

```
    int value;
```

```
}Item;
```

```
int knapsack_dp_recursion(int item_num, Item* item_list, int knapsack_capacity);
```

```
int main(){
```

```
    FILE* infile;
```

```
    int in_buffer[512], item_num, knapsack_capacity;
```

```
    infile = fopen("test.dat", "rb");    读入二进制文件
```

```
    fread(in_buffer, sizeof(int), 512, infile);
```

```
    fclose(infile);
```

```
    knapsack_capacity = in_buffer[0]; 第1个byte存储背包重量
```

```
    item_num = in_buffer[1];          第2个byte存储物品数量
```

```
    Item* item_list = (Item*)(in_buffer + 2); 第3个byte开始存储物品的重量和价值
```

```
    printf("%d\n", knapsack_dp_recursion(item_num, item_list, knapsack_capacity)); 调用函数求解问题
```

```
    return 0;
```

```
}
```

注意C语言实现输出了结果，但是汇编程序要求将结果写在\$V0中

接上页

exp2_3 自顶向下动态规划-C代码

```
int knapsack_dp_recursion(int item_num, Item* item_list, int knapsack_capacity){
    if(item_num == 0)return 0;
    if(item_num == 1){
        return (knapsack_capacity >= item_list[0].weight)? item_list[0].value: 0;
    }
    int val_out = knapsack_dp_recursion(item_num - 1, item_list + 1, knapsack_capacity);
    int val_in = knapsack_dp_recursion(item_num - 1, item_list + 1, knapsack_capacity -
item_list[0].weight) + item_list[0].value;
    if(knapsack_capacity < item_list[0].weight)return val_out;
    else return (val_out > val_in)? val_out: val_in;
}
```

递归终止条件，当只有一个物品时，判断物品能否添加进背包中

计算 $OPT(i - 1, w)$ 和 $v_i + OPT(i - 1, w - w_i)$

根据转移方程 $OPT(i, w) = \max\{OPT(i - 1, w), v_i + OPT(i - 1, w - w_i)\}$ 返回结果

作业提交要求

- 作业用一个压缩包提交，压缩包名称：“学号_姓名.7z”。推荐用7z格式，其他常见压缩格式也可以。
- 压缩包打开后需要包含：
 - 一个“实验报告.pdf”文件，完成实验内容
 - 一个“exp_1_1.asm”，一个“exp_1_2.asm”，一个“exp_1_3.asm”
 - 一个“exp_2_1.asm”，一个“exp_2_2.asm”，一个“exp_2_3.asm”
- 注意所有的MIPS代码需要和C语言代码对应，不可使用其他C程序！！注意实验2要求的输出格式：最终程序运行结束时结果储存在寄存器\$v0中。没有按要求输出将会酌情扣分。

参考资料

- MIPS32 官方网站资料
<https://www.mips.com/products/architectures/mips32-2/>
- 指令集架构简介 *Introduction to the MIPS32 Architecture.pdf*
- 指令集手册 *MIPS32 Instruction Set Manual.pdf*

附件

- JAVA环境安装包 JavaSetup8u241.exe
- MARS模拟器 Mars4_5.jar
- 二进制文件查看器 pxBinaryViewerSetup.exe
- 随机数生成