

# 数字逻辑与处理器基础

## MIPS汇编编程实验

2024年 春季学期

杜禧瑞 谢童欣

{dxr22, xtx23}@mails.tsinghua.edu.cn

# 目录

- 汇编程序设计基础
- MARS环境安装与基础使用方法
- 实验内容简介
- 参考资料

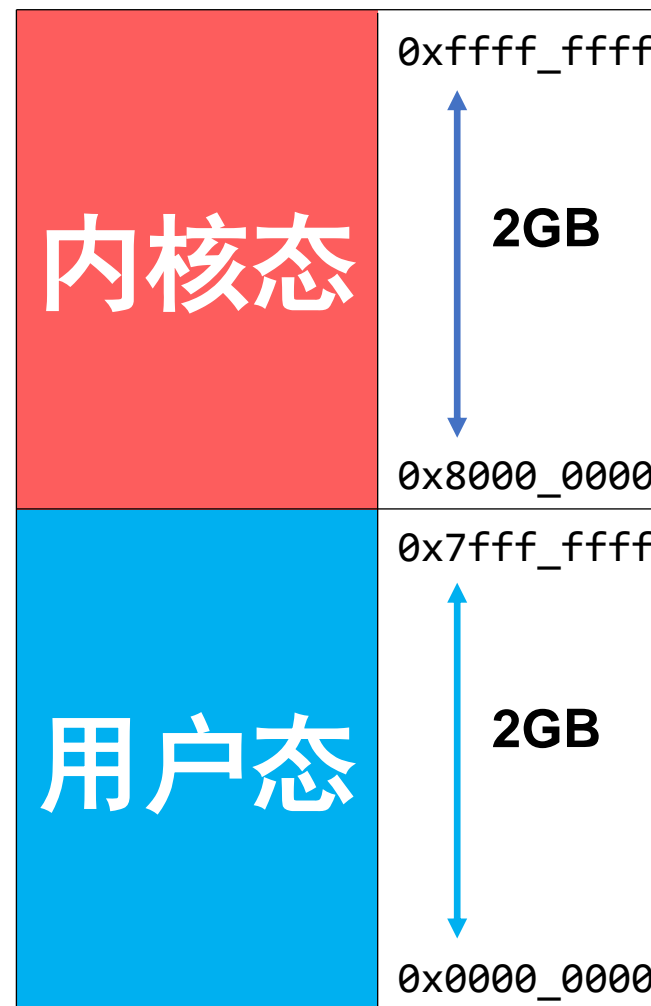
# 目录

- 汇编程序设计基础
  - MIPS内存分配
  - 汇编的语法
  - 变量
  - 分支
  - 数组
  - 系统调用
  - 过程调用

# 32位MIPS的内存分配

- 内存空间分配

- 32位的地址决定了能管理的内存最大为 $2^{32}=4\text{GB}$ 的大小
- 一般我们将低2GB规定为**用户态**空间，即一般的应用程序可以控制的空间
- 高2GB的空间属于**内核态**空间，这部分空间是由操作系统控制的，用户态程序不能控制



# 32位MIPS的内存分配

- 应用程序中常见的数据

```
int global;

int main()
{
    int local;
    char array0[10];
    int * array1;
    array1=(int *) malloc(10*sizeof(int));
    free(array1);
}
```

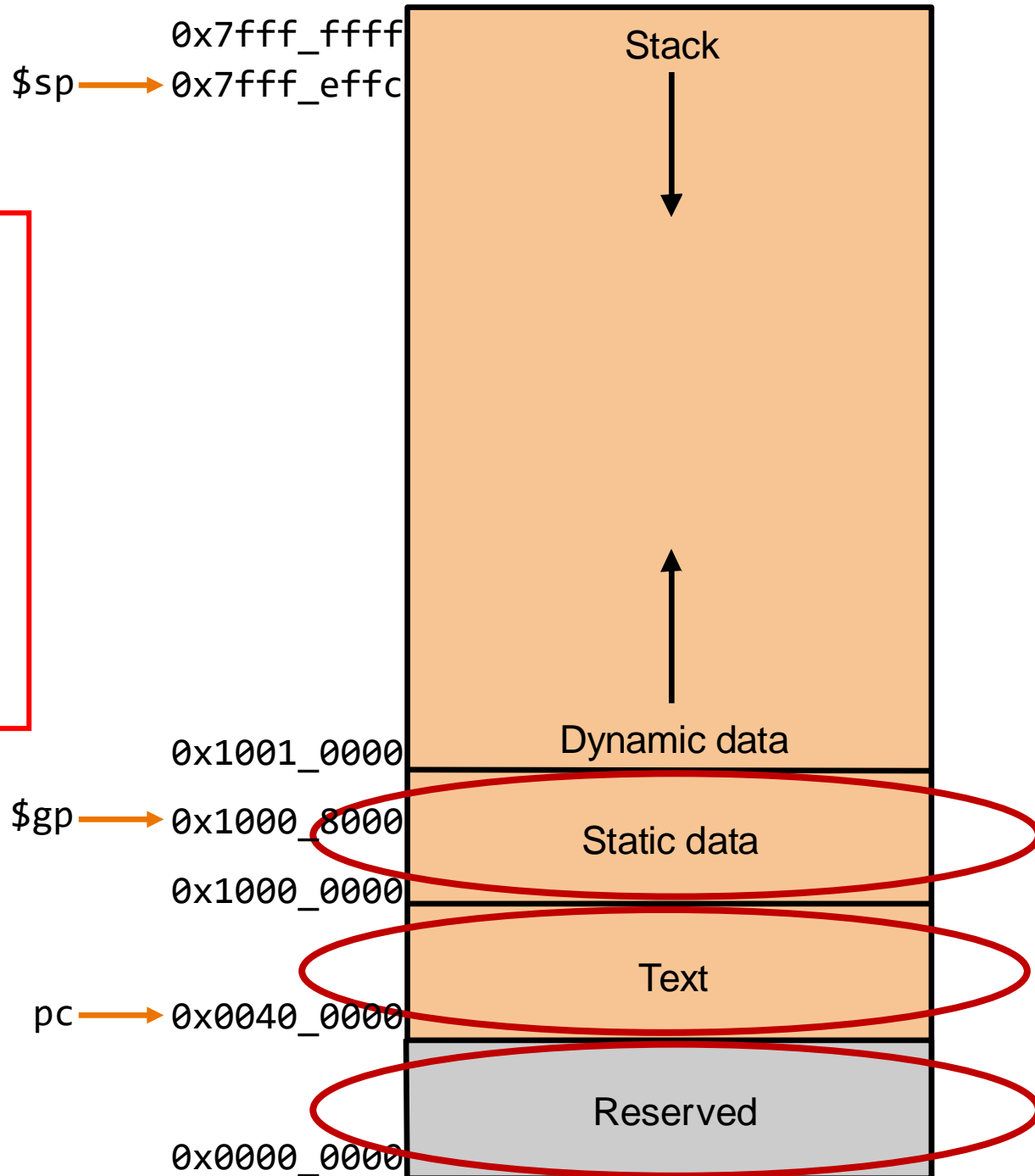
程序运行时这些数据都被放在内存空间的什么地方呢？

# 32位MIPS的内存分配

- 从顶端开始，对栈指针初始化为0x7ffffeffc，并向向下向数据段增长；
- 在底端，程序代码（文本）开始于0x00400000；
- 静态数据开始于0x10000000；
- 紧接着是由C中malloc进行存储器分配的动态数据，朝堆栈段向上增长

全局指针被设定为易于访问数据的地址，以便使用相对于\$gp的±16位偏移量

0x1000\_0000 - 0x1000\_ffff



# MIPS汇编：语法

- 注释以“#” 开始；
- **标签 (label)** 由字母、下划线 ( \_ )、点 ( . ) 构成，但不能以数字开头；  
标签大小写敏感；指令操作码是一些保留字，不能用作标签；
- 标签放在行首，后跟冒号 ( : ) ， 例如

```
.data                # 将子数据项，存放到数据段中
Item: .word 1,2      # 将2个32位数值送入地址连续的内存字中
.text               # 将子串，即指令或字送入用户文件段
.global main        # 必须为全局变量
Main: lw $t0, item
```

# MIPS汇编：语法

## MIPS汇编语言语句格式

- 指令与伪指令语句

**[label:]** <op> Arg1, [Arg2], [Arg3]    **[#comment]**

例如 **AddFunc:** **add** \$a1 \$a2 \$a3 **# a1=a2+a3**

- 汇编命令(directive)语句

**[label:]** **.directive** [arg1], [arg2], ...    **[#comment]**

例如 **.word** 0xa3



# MIPS汇编：语法

- 指令与伪指令（Pseudo Instructions）
  - 有一些MIPS指令是和机器码一一对应，可以直接翻译成机器码
    - `add $s0,$s1,$s2`      `lw $t0, 4($t1)`
  - 还有一些没有对应的机器码，不能直接翻译成机器码，需要先翻译成真的指令
    - `li $s1 0x7f` <==> `addi $s1 $zero 0x7f`
  - 编写程序时使用伪指令有利于提高效率并增加可读性
- 伪指令可在MARS仿真器（稍后介绍）的“帮助”下的Extended (pseudo) Instructions页面查询

# MIPS汇编：语法

## 汇编命令

### - 汇编器用来定义数据段、代码段以及为数据分配存储空间

<code>.data [address]</code>	# 定义数据段, [address]为可选的地址
<code>.text [address]</code>	# 定义正文段(即代码段), [address]为可选的地址
<code>.align n</code>	# 以 $2^n$ 字节边界对齐数据, 只能用于数据段
<code>.ascii &lt;string&gt;</code>	# 在内存中存放字符串
<code>.asciiz &lt;string&gt;</code>	# 在内存中存放NULL结束的字符串
<code>.word w1, w2, . . . , wn</code>	# 在内存中存放n个字
<code>.half h1, h2, . . . , hn</code>	# 在内存中存放n个半字
<code>.byte b1, b2, . . . , bn</code>	# 在内存中存放n个字节
<code>.space n</code>	# 在内存中存放n字节的数组

# MIPS汇编：语法

```
--          .data 0x10010000 # 以下部分定义数据段（可省略地址）
0x10010000  n:      .word 5      # 0x10010000处存1个字，值为5。后面程序可用n来访问此地址
0x10010008  str:    .asciiz "Hello!" # 0x10010008处存放以'\0'结尾字符串。str用来访问首地址
--          .align 2          # 对齐到4字节。否则下一个地址为0x1001000f，会出错！
0x10010010  arr:    .space 8     # 0x10010010处存8个字节（2个字）。arr用来访问首地址
--          .text 0x00400000 # 以下部分定义正文段（可省略地址）
0x00400000  la $s0, n            # 伪指令la：用于加载相应label的地址到相应寄存器
0x00400004  la $s1, arr
0x00400008  lw $t0, 0($s0)      # 使用访存指令去读写数据段的数据
0x0040000c  sw $t0, 0($s1)
0x00400010  addi $t0, $t0, 1
0x00400014  sw $t0, 4($s1)
```

# MIPS汇编： 变量

- **大部分变量存储在主存储器内**（而不是寄存器内）
  - 因为我们通常有很多的变量要存，不止32个
  - 用`.word`等汇编命令声明在数据段；为了实现功能，用`lw`语句将变量加载到寄存器中，对寄存器进行操作，然后再把结果`sw`回去
- **常访问的变量**
  - 让变量在寄存器中保留时间越长越好（`lw`和`sw`比寄存器操作要慢得多得多！）
  - 编译器往往可以自动将常访问的变量作为寄存器变量（如C的`register`存储类型）
  - 本次实验中可以自行决定是否将变量作为`register`存储类型，不放入主存
- **临时寄存器**
  - 由于一条指令只能采用两个输入，所以必须采用临时寄存器计算复杂的问题，如 $z=(x+y)+(x-y)$ ， $a[i]=a[n-i-1]$

# MIPS汇编：分支

符号汇编写法使用绝对位置	汇编器实际使用偏移量
beq \$zero, \$zero, foo	beq \$zero, \$zero, 0x??

- **偏移量**：从下一条指令对应的PC开始到标号位置还有多少条指令
  - beq \$zero, \$zero, foo如果位于地址0x00400000，foo位于地址0x00400100的话，
$$\begin{aligned}\text{偏移量} &= (\text{target} - (\text{PC} + 4)) / 4 = (0x00400100 - 0x00400004) / 4 \\ &= 0xfc / 4 = \text{0x3f}\end{aligned}$$
  - 偏移量为0则表示执行下一条指令不产生任何跳转
  - 原因：*relocatable* (可重新定位的)，分支语句可以在每次被加载到内存不同位置的情况下正常工作

# MIPS汇编：分支

- 分支

- 寄存器之间的比较，可以使用beq, ble, bge, blt, bgt, bne
- 如果和0比较，可以使用beqz, blez, bgez, bltz, bgtz, bnez
- 更复杂的比较，采用比较指令（如slt），然后再用与0比较

- 例子：

```
if (x >= 0)
    y = x;
else
    y = -x;
```

# MIPS汇编：分支

功能：求绝对值

```
.data 0x10000000
.word -6, 0                                # x: -6, y: 0
.text

main:
    lui $s6, 0x1000                        # $s6存放x的地址
    addiu $s5, $s6, 4                      # $s5存放y的地址
    lw $s0, 0($s6)                         # s0=-6
    slt $s2, $zero, $s0                    # 因为0<x不成立，所以这里$s2=0
    beqz $s2, else                         # $s2=0时直接跳到else
    move $s1, $s0                          # $s2=1时赋值后跳到done
    j done
else: sub $s1, $zero, $s0
done: sw $s1, 0($s5)
```

# MIPS汇编：数组

- 用.space来给数组开辟空间，或者.word开辟给定初始值的数组
  - 在编译时静态地开辟 $n \times 4$  bytes, (n个32-bit 字)

- 使用lw和sw访问数组

```
lw $temp, 0($A)           # temp = A[0];
```

```
sw $temp, 8($B)           # B[2] = temp;
```

- 将常数0, 8作为地址偏移量
- 将寄存器\$A和\$B作为数组中的开始地址(A[], B[])

注意这里\$A, \$B存的是地址，\$temp存的是具体的值



# MIPS汇编： 数组

.data

arr1: .word 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17

arr2: .space 68

.text

main: addu \$s7, \$ra, \$zero # 保存返回地址到\$s7

la \$s5, arr1 # \$s5=&A

la \$s6, arr2 # \$s6=&B

addiu \$s0, \$zero, 0x11 # Size(A)=Size(B)=0x11

loop: subu \$s0, \$s0, 1 # 计数

sll \$s1, \$s0, 2 # 换算地址

addu \$s2, \$s1, \$s5 # 计算A[]偏移量,送到\$s2

lw \$s3, 0(\$s2) # 读出A[]中的值

addu \$s2, \$s1, \$s6 # 计算B[]偏移量,送到\$s2

sw \$s3, 0(\$s2) # 写到B[]中去

bnez \$s0, loop

addu \$ra, \$zero, \$s7 # 返回主调过程

jr \$ra

每个数4Bytes, 1 word

功能： 将数组A的值依次拷贝到数组B中

# MIPS汇编：数组

使用移位操作代替mul和div: 因为mul和div一般都比sll和sr1慢

- sllv by k 等价于 mul by  $2^k$
  - sr1v by k 等价于 div by  $2^k$
- 只对无符号数成立，  
且没有超出数据表示范围

- 对于有符号数用 sra

- 高位用符号位填充(在2的补码表示情况下)

- e.g.,

# \$s0 = -6 = 0b11...11010

sr1 \$s0, \$s0, 1      # 0b01...11101    ×

sra \$s0, \$s0, 1      # 0b11...11101 = -3    ✓

# MIPS汇编：系统调用

- **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）
- 系统调用的参数与编号可在MARS仿真器（稍后介绍）的“帮助”下的Syscall页面查询

# MIPS汇编：系统调用

.text

```
addiu $v0, $zero, 5    # 系统调用编号5（整数输入）放入$v0
syscall                # 系统调用进行整数输入，结果会进入$v0
addu $a0, $zero, $v0   # 将系统调用参数（需要打印的整数）放到$a0
addiu $v0, $zero, 1    # 系统调用编号1（整数输出）放入$v0
syscall                # 系统调用进行整数输出，打印$a0内的数
```

# MIPS汇编：过程调用

- 过程调用寄存器保持情况（从主调过程的视角）
  - 主调过程和被调过程都需要维护一些数据

被保持（需被调过程维护）	不被保持（需主调过程维护）
保存寄存器：\$s0-\$s7	临时寄存器：\$t0-\$t9
栈指针：\$sp	参数寄存器：\$a0-\$a3
返回地址寄存器：\$ra	返回值寄存器：\$v0-\$v1

- 被调过程不改变这些寄存器数据
- 如果被调过程要用，需要被调过程维护好
- 进入被调过程后存储，离开被调过程前读取
- 被调过程可以改变这些寄存器数据
- 如果主调过程需要调用结束后继续使用，需主调过程提前备份好
- 过程调用前存储，过程调用后读取

# 目录

- 汇编程序设计基础
- **MARS环境安装与基础使用方法**
- 实验内容简介
- 参考资料

# 安装JRE

- 运行JAVA程序包需要运行环境：Java Runtime Environment (JRE)
- 64位Windows系统可运行attachment中jre\_8u401.exe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/](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?](#)

# 运行MARS

- 如果JRE正确安装，双击Mars4\_5.jar即可打开MARS仿真器。此时的**工作路径**为Mars4\_5.jar文件所处路径
- 如果打不开，先检查JRE是否正确安装，可以考虑重新安装
  - 方法：打开命令行，输入java -version并回车，能正确输出版本号即安装正确
  - Java安装正确，可以尝试用java -jar [相应路径/]Mars4\_5.jar打开MARS。若成功打开，此时**工作路径**为命令行当前所在路径
- 如果是MARS软件包的问题可以进入MARS官网下载  
<https://courses.missouristate.edu/KenVollmar/mars/download.htm>
- 如果仍无法解决，请尽早联系助教



# 运行MARS

- **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入输出（Run I/O选项卡）和系统报错（Mars Messages选项卡）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

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

File Edit Run Options Help **文件读写功能** **基本编辑功能** **汇编执行调试功能** **指令运行速度**

Run speed at max (no interaction)

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	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

见 example\_0.asm

**主要编辑区**

```
1 .data
2 in_buff: .space 512
3 out_buff: .space 512
4 input_file: .asciiz "example.in"
5 output_file: .asciiz "example.out"
6 comma: .asciiz ","
7 .text
8 la $a0, input_file #input_file 是一个字符串
9 li $a1, 0 #flag 0为读取 1为写入
10 li $a2, 0 #mode is ignored 设置为0就可以了
11 li $v0, 13 #13 为打开文件的 syscall 编号
12 syscall # 如果打开成功, 文件描述符返回到$v0中
13 move $a0,$v0 # 将文件描述符载入到 $a0中
14 la $a1, in_buff #in_buff 为数据暂存区
15 li $a2, 512 #读取512个byte
16 li $v0, 14 #14 为读取文件的 syscall 编号
17 syscall
18 li $v0 16 #16 为关闭文件的 syscall 编号
19 syscall
20
```

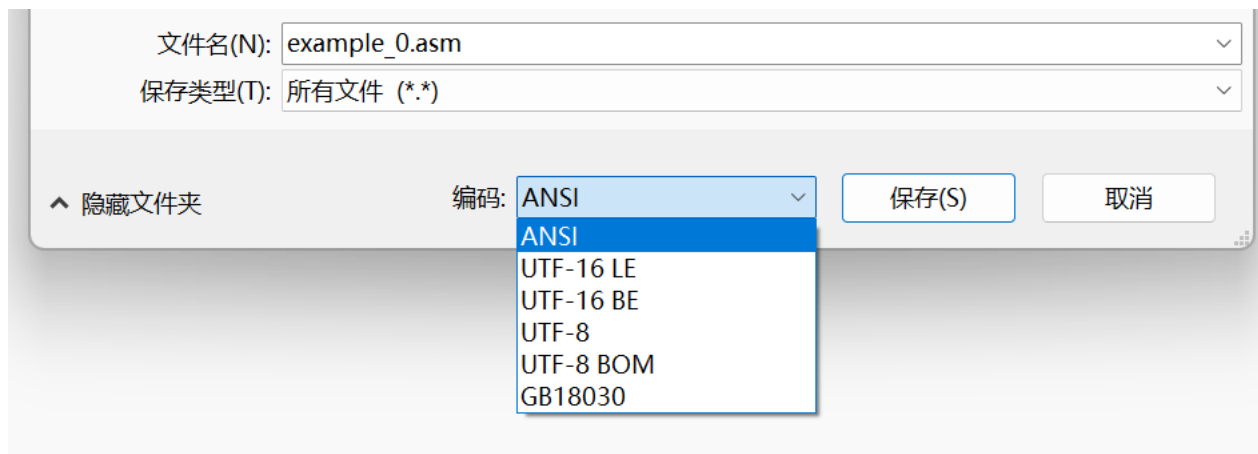
Mars Messages Run I/O

Clear

**输出信息区**

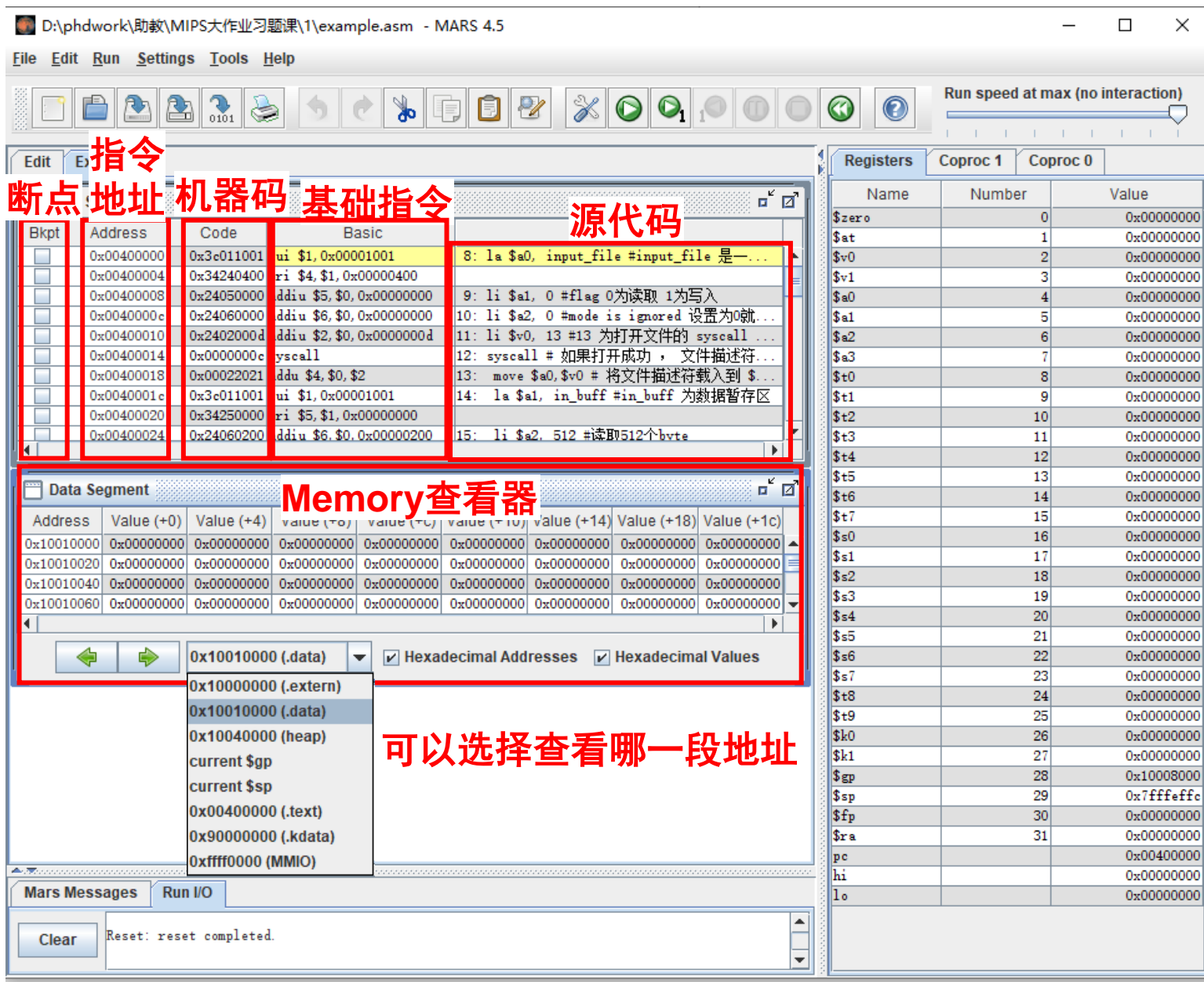
# 汇编运行

- 首先打开汇编文件  
example\_0.asm，并将输入文件  
example.in放在**工作路径**下
- Windows下中文乱码解决方法：  
用记事本打开后“另存为”，下  
面编码改为“ANSI”（如右图）



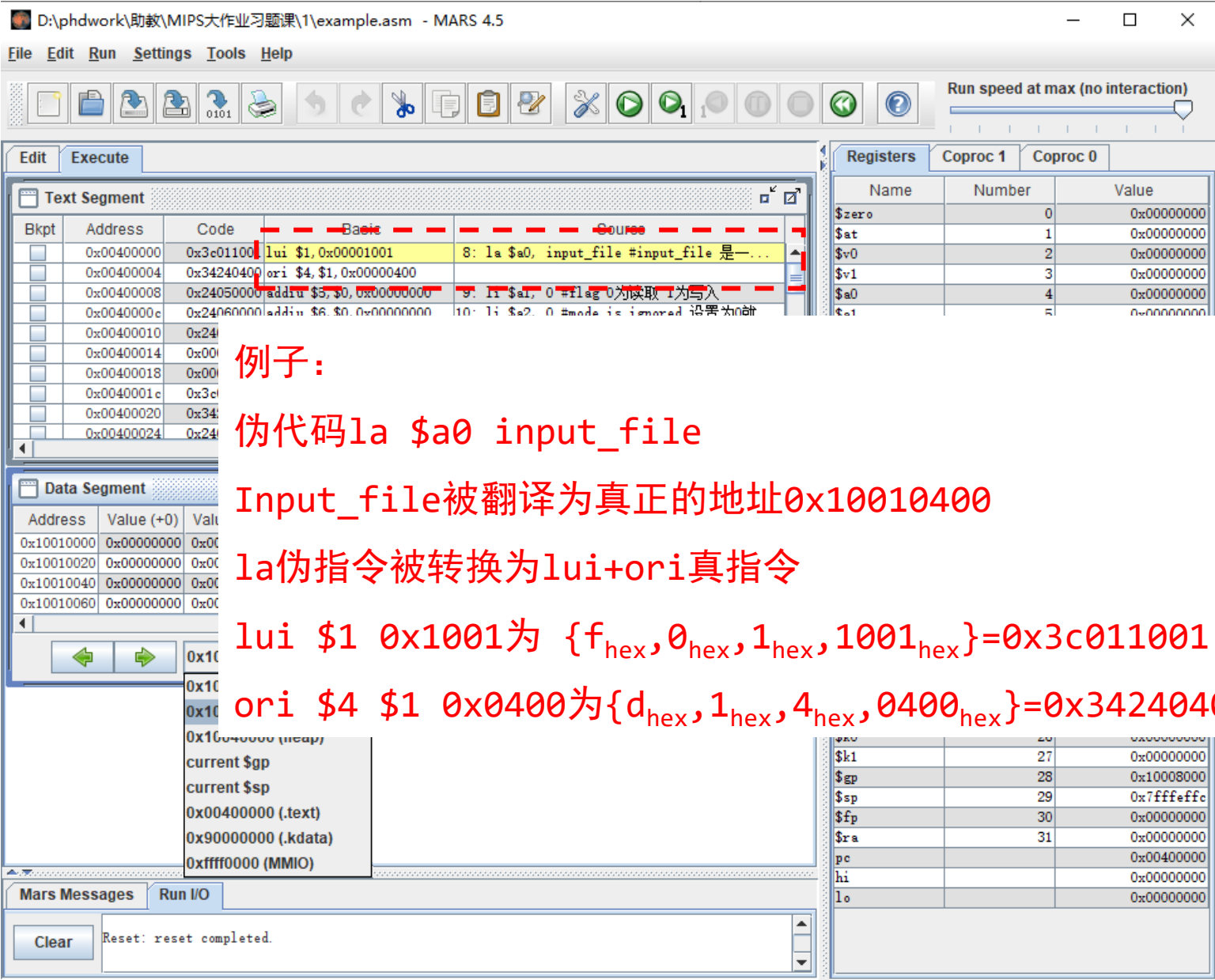
# 汇编运行

- 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**



# 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



Example assembly code snippet from the MARS simulator:

Bkpt	Address	Code	Comment
	0x00400000	lui \$1, 0x00001001	8: la \$a0, input_file #input_file 是一...
	0x00400004	ori \$4, \$1, 0x00000400	
	0x00400008	addiu \$8, \$0, 0x00000000	9: li \$a1, 0 #flag 0为读取 1为写入
	0x0040000c	addiu \$6, \$0, 0x00000000	10: li \$a2, 0 #mode is ignored 设置为0

Registers panel (Coproc 1):

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000

Mars Messages panel:

Reset: reset completed.

Examples of pseudo-instructions and their translation:

- 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

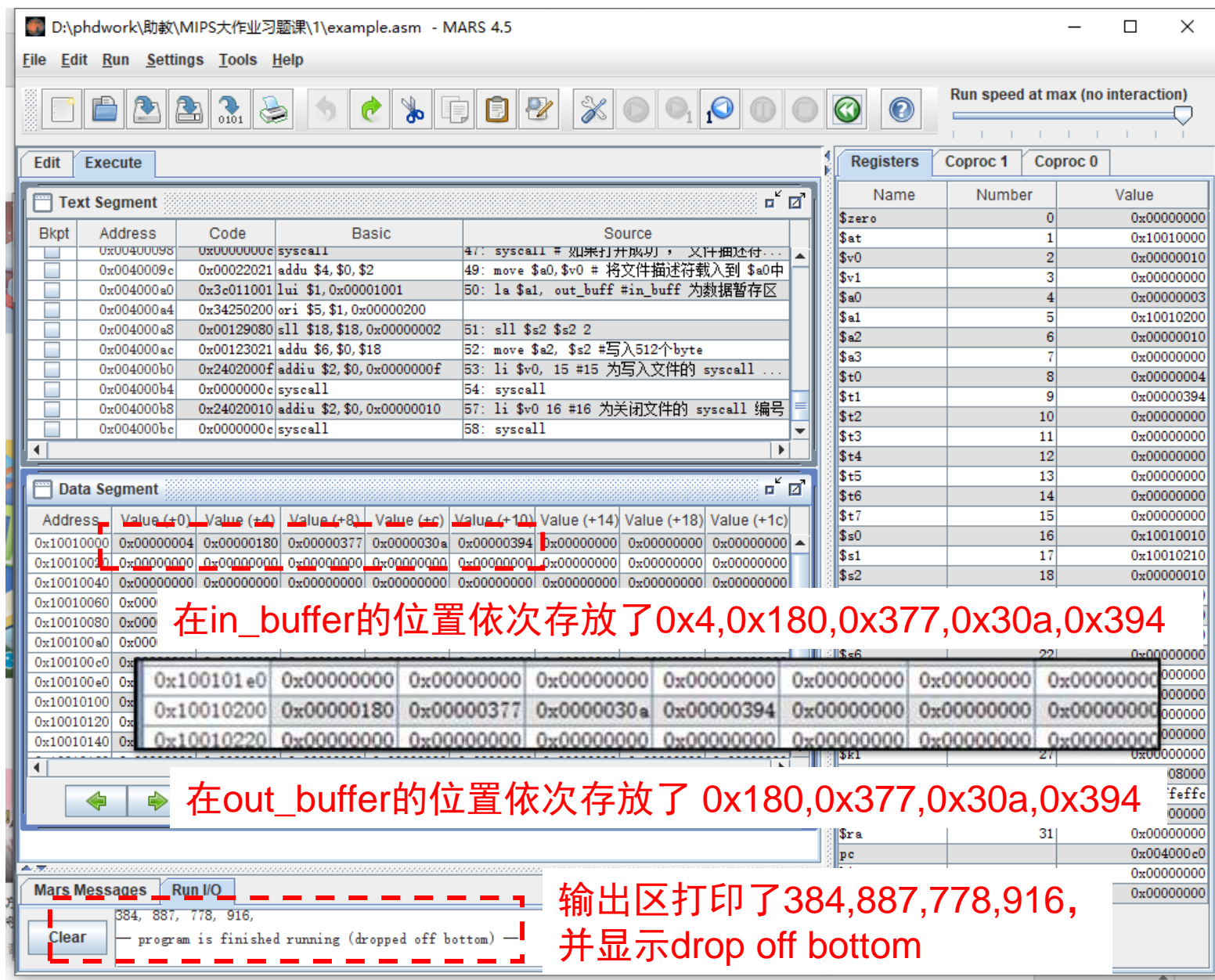
黄色指令代表当前指令  
是即将执行但尚未执行的指令  
也是pc寄存器对应的指令

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	0x7fffffc
\$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
- 各个寄存器内的值发生了变化，内存中in\_buffer和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（dropped 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，程序停  
在该位置
- 运行两次后，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

The screenshot shows the MARS 4.5 MIPS assembler simulator. The assembly code window displays the following instructions:

Bkpt	Address	Code	Basic	Source
	0x00400054	0x21080001	addi \$8,\$8,0x00000001	29: addi \$t0, \$t0, 1
	0x00400064	0x00092021	addu \$4,\$0,\$9	34: move \$a0, \$t1
	0x00400068	0x24020001	addiu \$2,\$0,0x00000001	35: li \$v0, 1
	0x0040006c	0x0000000c	syscall	36: syscall
<input checked="" type="checkbox"/>	0x00400070	0x3e011001	lui \$1,0x00001001	38: la \$a0, comma
	0x00400074	0x34210417	ori \$4,\$1,0x0000117	
	0x00400078	0x24020004	addiu \$2,\$0,0x00000004	39: li \$v0, 4

The registers window shows the following values:

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000001
	3	0x00000000
	4	0x00000377
	5	0x10010000
\$a2	6	0x00000200
\$a3	7	0x00000000
\$t0	8	0x00000002
\$t1	9	0x00000377
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x10010008
\$s1	17	0x10010208
\$s2	18	0x00000004

The data segment window shows the following values:

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0x00000004	0x00000180	0x00000377	0x0000030a	0x00000394	0x00000000	0x00000000	0x00000000
0x10010002	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010004	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x0000							
0x10010080	0x0000							
0x100100a0	0x0000							
0x100100c0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x100100e0	0x100101e0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010100	0x10010200	0x00000180	0x00000377	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010120	0x10010220	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x10010140								

The output buffer window shows the following values:

Address	Value
0x10010000	0x00000000
0x10010002	0x00000000
0x10010004	0x00000000
0x10010006	0x00000000
0x10010008	0x00000000
0x1001000a	0x00000000
0x1001000c	0x00000000
0x1001000e	0x00000000
0x10010010	0x00000000
0x10010012	0x00000000
0x10010014	0x00000000
0x10010016	0x00000000
0x10010018	0x00000000
0x1001001a	0x00000000
0x1001001c	0x00000000
0x1001001e	0x00000000
0x10010020	0x00000000
0x10010022	0x00000000
0x10010024	0x00000000
0x10010026	0x00000000
0x10010028	0x00000000
0x1001002a	0x00000000
0x1001002c	0x00000000
0x1001002e	0x00000000
0x10010030	0x00000000
0x10010032	0x00000000
0x10010034	0x00000000
0x10010036	0x00000000
0x10010038	0x00000000
0x1001003a	0x00000000
0x1001003c	0x00000000
0x1001003e	0x00000000
0x10010040	0x00000000
0x10010042	0x00000000
0x10010044	0x00000000
0x10010046	0x00000000
0x10010048	0x00000000
0x1001004a	0x00000000
0x1001004c	0x00000000
0x1001004e	0x00000000
0x10010050	0x00000000
0x10010052	0x00000000
0x10010054	0x00000000
0x10010056	0x00000000
0x10010058	0x00000000
0x1001005a	0x00000000
0x1001005c	0x00000000
0x1001005e	0x00000000
0x10010060	0x00000000
0x10010062	0x00000000
0x10010064	0x00000000
0x10010066	0x00000000
0x10010068	0x00000000
0x1001006a	0x00000000
0x1001006c	0x00000000
0x1001006e	0x00000000
0x10010070	0x00000000
0x10010072	0x00000000
0x10010074	0x00000000
0x10010076	0x00000000
0x10010078	0x00000000
0x1001007a	0x00000000
0x1001007c	0x00000000
0x1001007e	0x00000000
0x10010080	0x00000000
0x10010082	0x00000000
0x10010084	0x00000000
0x10010086	0x00000000
0x10010088	0x00000000
0x1001008a	0x00000000
0x1001008c	0x00000000
0x1001008e	0x00000000
0x10010090	0x00000000
0x10010092	0x00000000
0x10010094	0x00000000
0x10010096	0x00000000
0x10010098	0x00000000
0x1001009a	0x00000000
0x1001009c	0x00000000
0x1001009e	0x00000000
0x100100a0	0x00000000
0x100100a2	0x00000000
0x100100a4	0x00000000
0x100100a6	0x00000000
0x100100a8	0x00000000
0x100100aa	0x00000000
0x100100ac	0x00000000
0x100100ae	0x00000000
0x100100b0	0x00000000
0x100100b2	0x00000000
0x100100b4	0x00000000
0x100100b6	0x00000000
0x100100b8	0x00000000
0x100100ba	0x00000000
0x100100bc	0x00000000
0x100100be	0x00000000
0x100100c0	0x00000000
0x100100c2	0x00000000
0x100100c4	0x00000000
0x100100c6	0x00000000
0x100100c8	0x00000000
0x100100ca	0x00000000
0x100100cc	0x00000000
0x100100ce	0x00000000
0x100100d0	0x00000000
0x100100d2	0x00000000
0x100100d4	0x00000000
0x100100d6	0x00000000
0x100100d8	0x00000000
0x100100da	0x00000000
0x100100dc	0x00000000
0x100100de	0x00000000
0x100100e0	0x00000000
0x100100e2	0x00000000
0x100100e4	0x00000000
0x100100e6	0x00000000
0x100100e8	0x00000000
0x100100ea	0x00000000
0x100100ec	0x00000000
0x100100ee	0x00000000
0x100100f0	0x00000000
0x100100f2	0x00000000
0x100100f4	0x00000000
0x100100f6	0x00000000
0x100100f8	0x00000000
0x100100fa	0x00000000
0x100100fc	0x00000000
0x100100fe	0x00000000
0x10010100	0x00000000

The PC register shows the value 0x00400070.

Output buffer window shows the following values:

Address	Value
0x10010000	0x00000000
0x10010002	0x00000000
0x10010004	0x00000000
0x10010006	0x00000000
0x10010008	0x00000000
0x1001000a	0x00000000
0x1001000c	0x00000000
0x1001000e	0x00000000
0x10010010	0x00000000
0x10010012	0x00000000
0x10010014	0x00000000
0x10010016	0x00000000
0x10010018	0x00000000
0x1001001a	0x00000000
0x1001001c	0x00000000
0x1001001e	0x00000000
0x10010020	0x00000000
0x10010022	0x00000000
0x10010024	0x00000000
0x10010026	0x00000000
0x10010028	0x00000000
0x1001002a	0x00000000
0x1001002c	0x00000000
0x1001002e	0x00000000
0x10010030	0x00000000
0x10010032	0x00000000
0x10010034	0x00000000
0x10010036	0x00000000
0x10010038	0x00000000
0x1001003a	0x00000000
0x1001003c	0x00000000
0x1001003e	0x00000000
0x10010040	0x00000000
0x10010042	0x00000000
0x10010044	0x00000000
0x10010046	0x00000000
0x10010048	0x00000000
0x1001004a	0x00000000
0x1001004c	0x00000000
0x1001004e	0x00000000
0x10010050	0x00000000
0x10010052	0x00000000
0x10010054	0x00000000
0x10010056	0x00000000
0x10010058	0x00000000
0x1001005a	0x00000000
0x1001005c	0x00000000
0x1001005e	0x00000000
0x10010060	0x00000000
0x10010062	0x00000000
0x10010064	0x00000000
0x10010066	0x00000000
0x10010068	0x00000000
0x1001006a	0x00000000
0x1001006c	0x00000000
0x1001006e	0x00000000
0x10010070	0x00000000
0x10010072	0x00000000
0x10010074	0x00000000
0x10010076	0x00000000
0x10010078	0x00000000
0x1001007a	0x00000000
0x1001007c	0x00000000
0x1001007e	0x00000000
0x10010080	0x00000000
0x10010082	0x00000000
0x10010084	0x00000000
0x10010086	0x00000000
0x10010088	0x00000000
0x1001008a	0x00000000
0x1001008c	0x00000000
0x1001008e	0x00000000
0x10010090	0x00000000
0x10010092	0x00000000
0x10010094	0x00000000
0x10010096	0x00000000
0x10010098	0x00000000
0x1001009a	0x00000000
0x1001009c	0x00000000
0x1001009e	0x00000000
0x100100a0	0x00000000
0x100100a2	0x00000000
0x100100a4	0x00000000
0x100100a6	0x00000000
0x100100a8	0x00000000
0x100100aa	0x00000000
0x100100ac	0x00000000
0x100100ae	0x00000000
0x100100b0	0x00000000
0x100100b2	0x00000000
0x100100b4	0x00000000
0x100100b6	0x00000000
0x100100b8	0x00000000
0x100100ba	0x00000000
0x100100bc	0x00000000
0x100100be	0x00000000
0x100100c0	0x00000000
0x100100c2	0x00000000
0x100100c4	0x00000000
0x100100c6	0x00000000
0x100100c8	0x00000000
0x100100ca	0x00000000
0x100100cc	0x00000000
0x100100ce	0x00000000
0x100100d0	0x00000000
0x100100d2	0x00000000
0x100100d4	0x00000000
0x100100d6	0x00000000
0x100100d8	0x00000000
0x100100da	0x00000000
0x100100dc	0x00000000
0x100100de	0x00000000
0x100100e0	0x00000000
0x100100e2	0x00000000
0x100100e4	0x00000000
0x100100e6	0x00000000
0x100100e8	0x00000000
0x100100ea	0x00000000
0x100100ec	0x00000000
0x100100ee	0x00000000
0x100100f0	0x00000000
0x100100f2	0x00000000
0x100100f4	0x00000000
0x100100f6	0x00000000
0x100100f8	0x00000000
0x100100fa	0x00000000
0x100100fc	0x00000000
0x100100fe	0x00000000
0x10010100	0x00000000

The output buffer window shows the following values:

Address	Value
0x10010000	0x00000000
0x10010002	0x00000000
0x10010004	0x00000000
0x10010006	0x00000000
0x10010008	0x00000000
0x1001000a	0x00000000
0x1001000c	0x00000000
0x1001000e	0x00000000
0x10010010	0x00000000
0x10010012	0x00000000
0x10010014	0x00000000
0x10010016	0x00000000
0x10010018	0x00000000
0x1001001a	0x00000000
0x1001001c	0x00000000
0x1001001e	0x00000000
0x10010020	0x00000000
0x10010022	0x00000000
0x10010024	0x00000000
0x10010026	0x00000000
0x10010028	0x00000000
0x1001002a	0x0000000

# 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

注意被读取文件需要放在工作路径下这个例程才能正常读取

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
```



# 其他样例代码简介

- example\_1.asm
  - 基本代码格式
- example\_2.asm
  - 汇编命令使用
- example\_3.asm
  - 函数使用（无出入栈）

见 example\_3.asm

```
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
```

# 目录

- 汇编程序设计基础
- MARS环境安装与基础使用方法
- 实验内容简介
- 参考资料

# 实验部分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：系统调用
  - 练习1-2：循环，分支
  - 练习1-3：数组，指针
  - 练习1-4：函数调用

# 实验部分2

- 用MIPS32汇编指令翻译给定的3段排序算法代码
  - 插入排序：基本汇编操作
  - 二分插入排序：递归调用
  - 归并排序：链表操作
- 翻译要求：不要求逐句对应，但程序执行整体流程应当和所给出的代码**完全一致**

# 作业提交要求

- 详细提交要求见说明文档
  - 需要使用 **MIPS32** 汇编对 **给定C或C++程序** 进行翻译，不可使用其他汇编语言，不可自行写程序
  - 未按要求进行输入输出/提交可能会导致脚本评测失分！
  - 实验报告：只需要给出运行结果&让助教能结合注释看懂代码
- **不要造假，不要抄袭！！！！**
  - 会查重（同届和往届代码，网络代码，多种AI助手生成代码等）

# 目录

- 汇编程序设计基础
- MARS环境安装与基础使用方法
- 实验内容简介
- 参考资料

# 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*
- 课程教材第V章
- 指令集课件附录

# 附件

- JAVA运行环境安装包jre\_8u401.exe
- MARS模拟器 Mars4\_5.jar
- 二进制文件查看器 pxBinaryViewerSetup.exe