

Lab1 指令级 MIPS 模拟器

姓名：王茂增

学号：2113972

代码：<https://github.com/mzwangg/ComputerArchitecture>

sim.c 整体设计

1. `process_instruction()`: 处理当前指令

1. **读取当前指令**: 通过`mem_read_32(CUR_PC)`读取
2. **解析当前指令**: 通过位运算, 解析出`op`、`rs`、`rt`、`rd`、`immediate`、`target`、`shamt`、`funct`对应的值, 并保存在全局变量中
3. **PC 加 4**: 将 `NEXT_PC` 设为 `CUR_PC + 4`
4. **分发并执行指令**: 根据`op`的值, 判断不同指令的类型并分别处理
 1. `op == 0x01`: R 型指令, 调用`r_fmt_ins_exec()`进行处理
 1. `r_fmt_ins_exec()`: 通过`switch(funct)`, 解析出不同指令, 并实现相应功能
 2. `op == 0x02`: I 型分支指令
 1. `i_fmt_br_ins_exec()`: 通过`switch (rt)`解析出不同指令, 并实现相应功能
 3. `op == 0x02 || op == 0x03`: J 型指令
 1. `j_fmt_ins_exec()`: 通过`switch (op)`解析出不同指令, 并实现相应功能
 4. **else**: I 型指令
 1. `i_fmt_ins_exec()`: 通过`switch (op)`解析出不同指令, 并实现相应功能

具体实现

逻辑运算指令

and、or、xor、nor 指令

```
// 可直接调用c++中的逻辑运算符, 将rs和rt寄存器的值进行逻辑运算, 将结果存放在rd中
NEXT_REG(rd) = CUR_REG(rs) & CUR_REG(rt); //AND
NEXT_REG(rd) = CUR_REG(rs) | CUR_REG(rt); //OR
NEXT_REG(rd) = CUR_REG(rs) ^ CUR_REG(rt); //XOR
NEXT_REG(rd) = ~(CUR_REG(rs) | CUR_REG(rt)); //NOR
```

andi、ori、xori 指令指令

```
//可直接调用c++中的逻辑运算符, 将rs寄存器的值与经无符号拓展的立即数进行逻辑运算, 将结果存放在rt中
NEXT_REG(rt) = CUR_REG(rs) & (uint32_t)immediate;; //ANDI
NEXT_REG(rt) = CUR_REG(rs) | (uint32_t)immediate; //ORI
NEXT_REG(rt) = CUR_REG(rs) ^ (uint32_t)immediate;; //XORI
```

lui 指令

```
// 加载立即数的高16位，低16位用0填充，加载到rt寄存器中
NEXT_REG(rt) = (uint32_t)immediate << 16; //LUI
```

移位指令

sll、sllv、sra、srav、srl、srlv 指令

```
// 逻辑左移、右移指令，直接使用<<或>>即可，对rt进行运算，结果储存在rd中
NEXT_REG(rd) = CUR_REG(rt) << shamt; //SLL
NEXT_REG(rd) = CUR_REG(rt) >> shamt; //SRL

// 算数右移指令，先将数据转为有符号整数，再右移即可，对rt进行运算，结果储存在rd中
NEXT_REG(rd) = (int32_t)CUR_REG(rt) >> shamt; //SRA

// 算数左移、右移，移动位数为rs寄存器的后五位，所以通过& 0x1F得到低五位的值，结果保存在rd寄存器中
NEXT_REG(rd) = CUR_REG(rt) << (CUR_REG(rs) & 0x1F); //SLLV
NEXT_REG(rd) = (int32_t)CUR_REG(rt) >> (CUR_REG(rs) & 0x1F); //SRLV

// 类似SRLV指令，不过是算数右移，所以将rt寄存器的值转为有符号整数
NEXT_REG(rd) = (int32_t)CUR_REG(rt) >> (CUR_REG(rs) & 0x1F); // SRAV
```

算数操作指令

add、addu、sub、subu、slt、sltu 指令

```
// add与addu、sub与subu的区别在于是否进行溢出检测，由于Lab1暂时不实现，所以两者的命令一样
// rs寄存器的值 +/- rt寄存器的值，结果保存在rd中
NEXT_REG(rd) = CUR_REG(rs) + CUR_REG(rt); //ADD
NEXT_REG(rd) = CUR_REG(rs) + CUR_REG(rt); //ADDU
NEXT_REG(rd) = CUR_REG(rs) - CUR_REG(rt); //SUB
NEXT_REG(rd) = CUR_REG(rs) - CUR_REG(rt); //SUBU

// SLT和SLTU的区别在于处理的是有符号数还是无符号数
// 所以SLT要是用(int32_t)进行类型转换
// 如果rs寄存器的值小于rt寄存器的值则将rd寄存器置为1，否则置为0
NEXT_REG(rd) = (int32_t)CUR_REG(rs) < (int32_t)CUR_REG(rt); //SLT
NEXT_REG(rd) = CUR_REG(rs) < CUR_REG(rt); //SLTU
```

addi、addiu、slti、sltiu 指令

```
// addi与addiu的区别在于是否进行溢出检测，由于Lab1暂时不实现，所以两者的命令一样
// 对于符号拓展，由于immediate为uint16_t类型，我们要先将其转为int16_t，再转为int32_t
// 将rs寄存器的值加上符号拓展的立即数再保存在rt中
NEXT_REG(rt) = CUR_REG(rs) + (int32_t)(int16_t)immediate; //ADDI
NEXT_REG(rt) = CUR_REG(rs) + (int32_t)(int16_t)immediate; //ADDIU

// slti、sltiu的区别在于处理的是有符号数还是无符号数
// 两者均需要进行符号拓展，所以对于SLTIU，需要进行(uint32_t)(int32_t)(int16_t)的转换，
// 先转为符号拓展的32位有符号整数，再转为无符号
// 如果rs寄存器的值小于符号拓展的立即数，则将rt寄存器置为1，否则置为0
NEXT_REG(rt) = (int32_t)CUR_REG(rs) < (int32_t)(int16_t)immediate; //SLTI
NEXT_REG(rt) = CUR_REG(rs) < (uint32_t)(int32_t)(int16_t)immediate; //SLTIU
```

multu、mult 指令

```
// MULT
// 有符号乘法指令，将结果的低32位保存在NEXT_LO，高32位保存在NEXT_HI
NEXT_HI = ((int64_t)CUR_REG(rs) * (int64_t)CUR_REG(rt)) >> 32;
NEXT_LO = ((int64_t)CUR_REG(rs) * (int64_t)CUR_REG(rt)) & 0xFFFFFFFF;

//MULTU
// 无符号乘法指令，将结果的低32位保存在NEXT_LO，高32位保存在NEXT_HI
NEXT_HI = ((uint64_t)CUR_REG(rs) * (uint64_t)CUR_REG(rt)) >> 32;
NEXT_LO = ((uint64_t)CUR_REG(rs) * (uint64_t)CUR_REG(rt)) & 0xFFFFFFFF;
```

divu、div 指令

```
// DIV为有符号除法指令，NEXT_LO保存除数，NEXT_HI保存余数
if (CUR_REG(rt) != 0) {
    NEXT_LO = (int32_t)CUR_REG(rs) / (int32_t)CUR_REG(rt);
    NEXT_HI = (int32_t)CUR_REG(rs) % (int32_t)CUR_REG(rt);
} else {
    printf("除数不能为0! ");
}

// DIVU为无符号除法指令，NEXT_LO保存除数，NEXT_HI保存余数
if (CUR_REG(rt) != 0) {
    NEXT_LO = CUR_REG(rs) / CUR_REG(rt);
    NEXT_HI = CUR_REG(rs) % CUR_REG(rt);
} else {
    printf("除数不能为0! ");
}
```

转移指令

jr、jalr、j、jal（跳转指令）

```

// JR为跳转指令，将下一个PC设置为rs寄存器的值
NEXT_PC = CUR_REG(rs);

// JALR使用rd保存下一PC的值，然后跳转到rs寄存器指定的地址
NEXT_REG(rd) = NEXT_PC;
NEXT_PC = CUR_REG(rs);

// J为无条件跳转，PC的高四位不变，低28位由target给出，其中NEXT_PC已经PC+4
NEXT_PC = (NEXT_PC & 0xF0000000) | (target << 2);

// 对于JAL指令，首先将PC+4的值保存在$31寄存器
NEXT_REG(31) = CUR_PC + 4;
// 无条件跳转，PC的高四位不变，低28位由target给出，其中NEXT_PC已经PC+4
NEXT_PC = (NEXT_PC & 0xF0000000) | (target << 2);

```

beq、bne、bgez、bgezal、bltz、bltzal、bgtz、blez (分支指令)

```

//分支指令的目标地址均为立即数符号拓展并左移两位的地址+PC+4，其中PC+4已经在其他地方实现

//BEQ和BNE分别在rs和rt寄存器的值相等时分支
if (CUR_REG(rs) == CUR_REG(rt)) { //BEQ
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
}
if (CUR_REG(rs) != CUR_REG(rt)) { //BNE
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
}

// BGEZ指令和BGEZAL指令均在寄存器值大于等于0时分支，不过BGRZAL会同时将下一PC写入寄存器$31
if ((int32_t)CUR_REG(rs) >= 0) //BGEZ
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
if ((int32_t)CUR_REG(rs) >= 0) { //BGEZAL
    NEXT_REG(31) = NEXT_PC;
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
}

// BLTZ和BLTZAL与BGEZ指令和BGEZAL指令类似，不过是在小于0时分支
if ((int32_t)CUR_REG(rs) < 0) //BLTZ
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
if ((int32_t)CUR_REG(rs) < 0) {
    NEXT_REG(31) = NEXT_PC; //BLTZAL
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
}

// BGTZ为大于则分支，立即数符号拓展并左移两位，其中NEXT_PC已经PC+4
if ((int32_t)CUR_REG(rs) > 0)
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
break;

```

```
// BLEZ为小于等于则分支，立即数符号拓展并左移两位，其中NEXT_PC已经PC+4
if ((int32_t)CUR_REG(rs) <= 0)
    NEXT_PC += ((int32_t)(int16_t)immediate << 2);
```

加载存储指令

对于加载存储指令，都需要读取和写入数据到内存中。但是我们要保证读取和写入的地址 4 字节对齐，所以我们首先定义如下变量：

- address: 写入或读取的地址，通过符号拓展的 immediate 再加上 rs 寄存器的值计算
- address_aligned: 在 address 之前且最近的四字节对齐地址，我们读取和写入都要针对这个地址进行
- address_data: 上述对齐地址处的值

```
uint32_t address = (uint32_t)(int32_t)(int16_t)immediate + CUR_REG(rs); // 计算
address
uint32_t address_aligned = address & 0xffffffffc; // address & 0xffffffffc使address4
字节对齐
uint32_t address_data = mem_read_32(address_aligned); // 读取对齐后address的位置
```

lb、lbu、Lh、Lhu、lw 指令

对于上述指令，我着重介绍一下 LB 指令是如何实现的，其他指令的实现方法类似。

对于 LB 指令，可以读取任意地址处的字节，不过我们只能读取四字节对齐地址的值，所以需要进行处理。我们首先通过 $(\text{address} \& 0x3) \ll 3$ 计算得到 LB 指令读取位置相对于四字节对齐地址的偏移量，然后通过 $(\text{address_data} \& (0xff \ll \text{shift_lb}))$ 取出目标地址的数据，并通过 $\gg \text{shift_lb}$ 将该数据移到最低位，最后进行符号拓展得到对应位置的值。

```
// LB指令
int16_t shift_lb = (address & 0x3) << 3;
NEXT_REG(rt) = (int32_t)(int8_t)(((address_data & (0xff << shift_lb)) >>
shift_lb));
```

```
//LBU指令
int16_t shift_lbu = (address & 0x3) << 3;
NEXT_REG(rt) = (uint32_t)(((address_data & (0xff << shift_lbu)) >> shift_lbu));

//LH指令
if (address & 0x1) { // 当地址的最低位不是零时发生地址错误
    printf("地址错误!\n");
} else {
    int16_t shift_lh = (address & 0x2) << 3;
    NEXT_REG(rt) = (int32_t)(int16_t)(((address_data & (0xffff << shift_lh)) >>
shift_lh));
}
```

```
//LHU指令
if (address & 0x1) { // 当地址的最低位不是零时发生地址错误
    printf("地址错误!\n");
} else {
    int16_t shift_lhu = (address & 0x2) << 3;
    NEXT_REG(rt) = (uint32_t)(((address_data & (0xffff << shift_lhu)) >>
shift_lhu));
}

//LW指令
if (address & 0x3) { // 当地址的低两位不是零时发生地址错误
    printf("地址错误!\n");
} else {
    NEXT_REG(rt) = mem_read_32(address);
}
```

sb、sh、sw 指令

同样的，对于上述指令，我们需要写入一个不一定四字节对齐的位置，下面我以 SB 指令为例进行介绍。

对于 SB 地址，我们，仍然通过 $(\text{address} \& 0x3) \ll 3$ 计算得到 SB 指令写入位置相对于四字节对齐地址的偏移量，然后通过 $\text{address_data} \&= \sim(0xFF \ll \text{shift_sb})$ 清除原始数据的相应 8 位位置，然后通过指令 $\text{address_data} |= ((\text{CUR_REG}(\text{rt}) \& 0xff) \ll \text{shift_sb})$ 对这八位地址进行写入，最后 $\text{mem_write_32}(\text{address_aligned}, \text{address_data})$ 写到内存的相应位置。

```
// SB指令
// 计算要加载位置的偏移量
int16_t shift_sb = (address & 0x3) << 3;
// 清除原始数据的相应8位位置
address_data &= ~(0xFF << shift_sb);
// 将rt寄存器的低8位值存储到原始数据的相应位置
address_data |= ((CUR_REG(rt) & 0xff) << shift_sb);
// 将修改后的数据写回到内存
mem_write_32(address_aligned, address_data);
```

```
// SH指令
if (address & 0x1) { // 当地址的最低位不是零时发生地址错误
    printf("地址错误!\n");
} else {
    // 计算要加载位置的偏移量
    int16_t shift_sh = (address & 0x2) << 3;
    // 清除原始数据的相应16位位置
    address_data &= ~(0xFFFF << shift_sh);
    // 将rt寄存器的低16位值存储到原始数据的相应位置
    address_data |= ((CUR_REG(rt) & 0xffff) << shift_sh);
    // 将修改后的数据写回到内存
    mem_write_32(address_aligned, address_data);
}
```

```
// SW指令
if (address & 0x3) { // 当地址的低两位不是零时发生地址错误
    printf("地址错误!\n");
} else {
    mem_write_32(address, CUR_REG(rt));
}
```

实验验证

验证方法

Mars 模拟器是一个用于 MIPS 架构汇编语言和机器码的模拟和调试工具。这个模拟器主要用于教育目的，帮助学生和程序员理解计算机体系结构和汇编语言编程。

在实验中，我使用 Mars 将 `inputs` 文件夹中的 `.s` 文件转为 `.x` 文件，并逐步执行代码与本项目的结果进行对照，本项目逐步执行结果可在 `dumpsim` 文件夹中找到。通过逐步对照，结果表明，本项目的实现完全正确。

增加测试程序

除此之外，本项目还编写了一个测试文件 `my_test.s`，包含了逻辑运算、算术运算、跳转指令、加载存储指令等，从而更全面地验证了程序的正确性。下面是 `my_test.s` 文件内容：

```
.data
array: .space 16 # 用于存储数据的数组
result: .word 0 # 用于存储计算结果的变量

.text
main:
    # 初始化数据
    ori $t0, $zero, 10 # 将寄存器$t0初始化为10
    ori $t1, $zero, 3  # 将寄存器$t1初始化为3
    sw $t0, array      # 存储$t0的值到数组
    sw $t1, array + 4  # 存储$t1的值到数组的下一个位置

    # 加法
    lw $t2, array      # 从数组加载数据到$t2
    add $t3, $t0, $t1   # 计算$t0 + $t1
    add $t4, $t2, $t3   # 计算$t2 + $t3

    # 减法
    sub $t5, $t4, $t2   # 计算$t4 - $t2

    # 乘法
    mult $t5, $t1       # 计算$t5 * $t1

    # 除法
    div $t7, $t5, $t0   # 计算$t6 / $t0

    # 分支跳转
    beq $t7, $t1, equal
```

```
bne $t7, $t1, not_equal

equal:
    ori $t8, $zero, 1
    j end

not_equal:
    ori $t8, $zero, 0

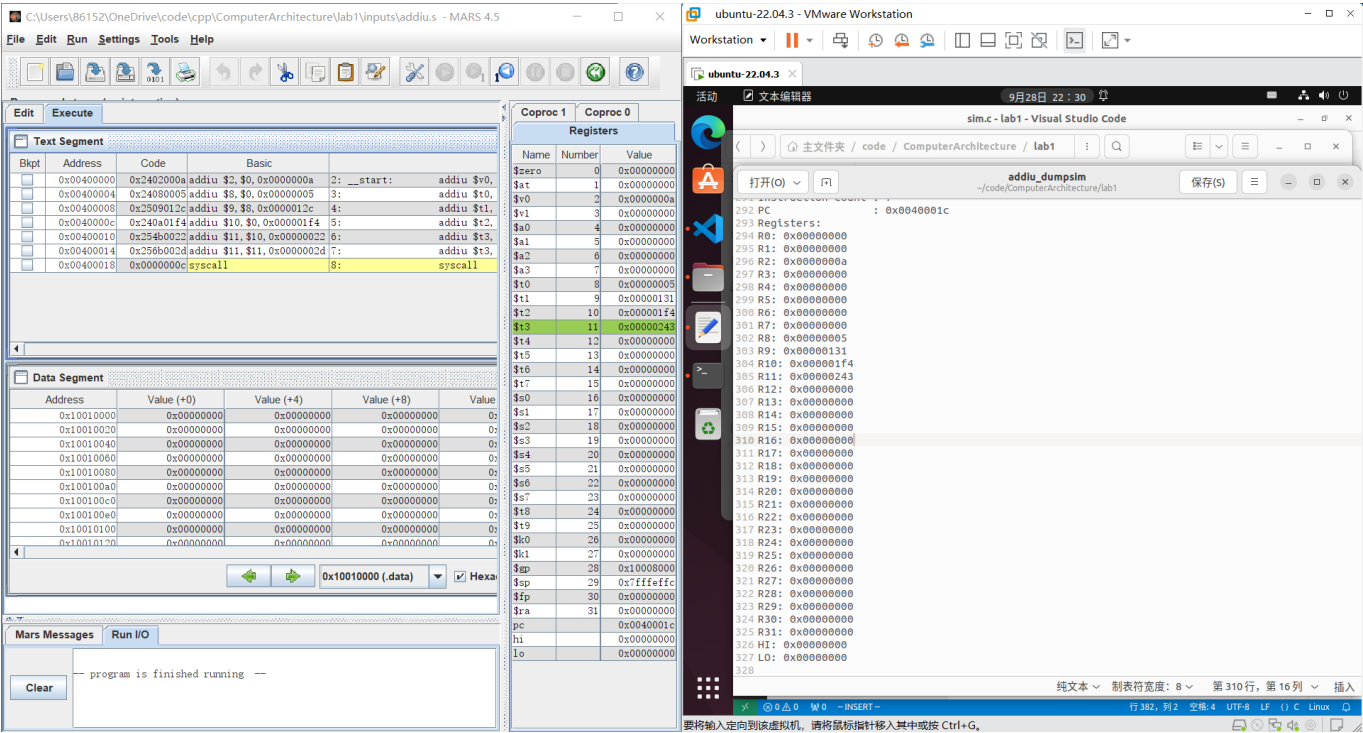
end:
    # 存储计算结果到result变量
    sw $t8, result

    # 终止程序
    addiu $v0, $zero, 10
    syscall
```

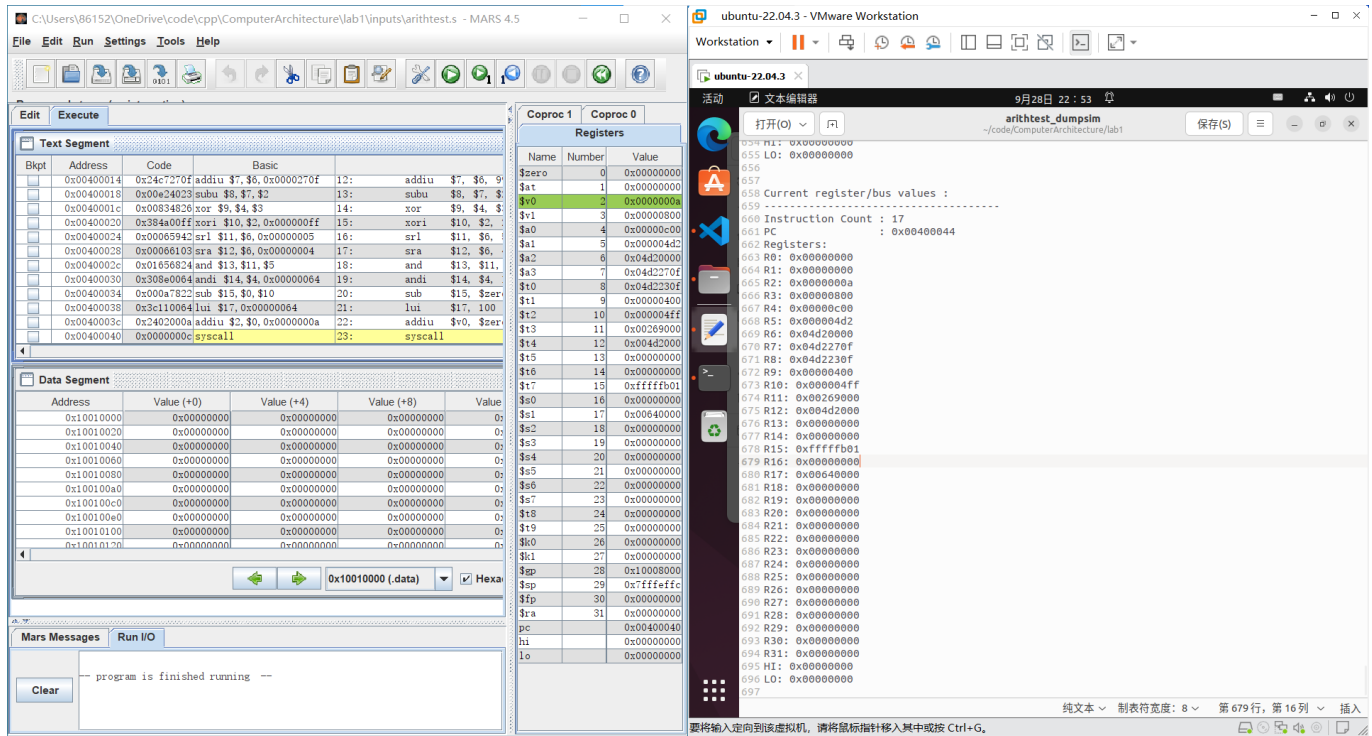
验证结果

最后，我将附上每一个实验最后的寄存器和内存结果与 Mars 运行结果的对照图，以说明程序的正确性。可以看到，寄存器和内存与 Mars 的运行结果一致。

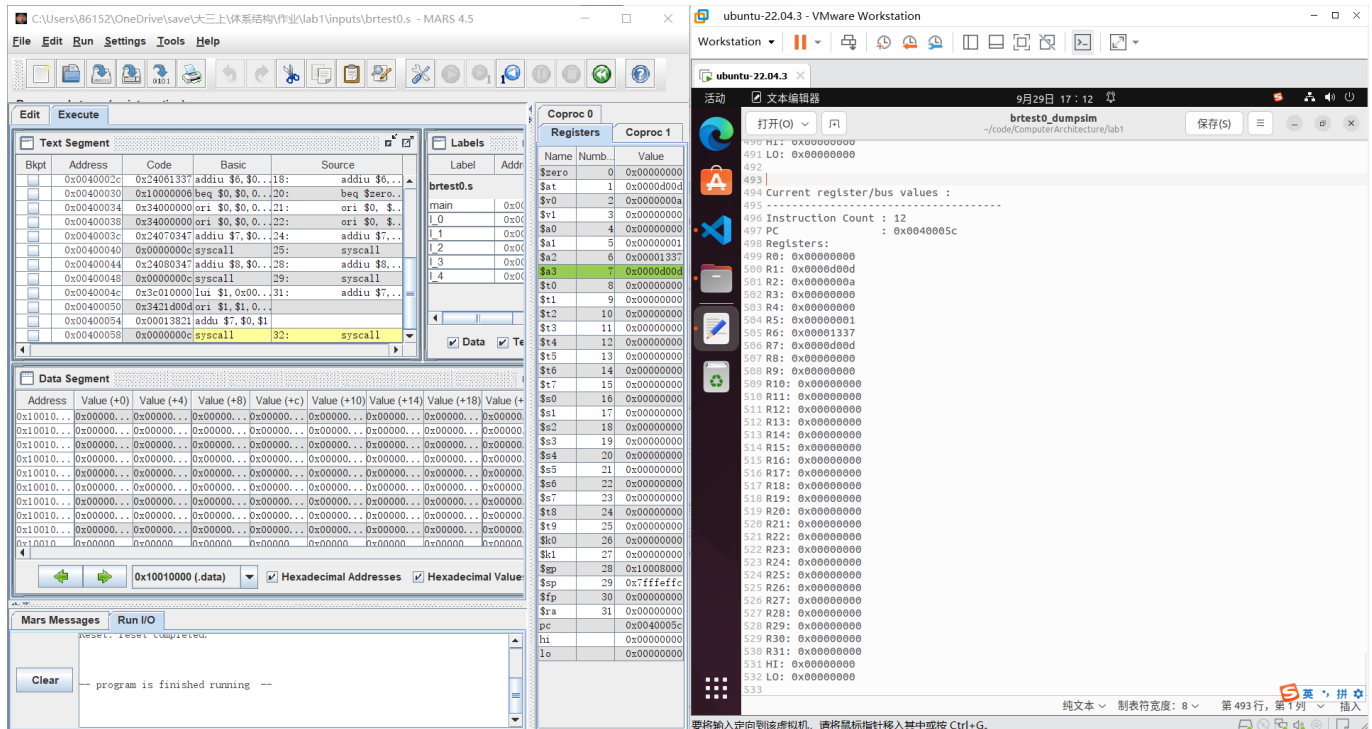
daddiu



arithtest



brtest0



brtest1

The screenshot displays two windows. The left window is MARS 4.5, showing the assembly code for `brtest1.s`. The code includes instructions like `addiu $5, $5, 0x00000001`, `syscall`, `addu $5, $5, $6`, `bltzal $4, 1, 12`, `addiu $5, $5, 400`, `addu $5, $5, $6`, `bgezal $4, 1, 11`, `addiu $5, $5, 0x00000063d`, `addu $5, $5, $1`, and `syscall`. The right window is Ubuntu-22.04.3-VMware Workstation, showing the output of the `brtest1_dumpsim` program. The output displays the current register/bus values, instruction count (30), and the PC value (0x00400090). The registers are listed with their values, and the program is finished running.

brtest2

The screenshot displays two windows. The left window is MARS 4.5, showing the assembly code for `brtest2.s`. The code includes instructions like `addiu $2, $0, 0x00000000a`, `j 1, 1`, `bne $zero, $zero, 1, 3`, `beq $0, $0, 0x000000003`, `bne $zero, $zero, 1, 4`, `syscall`, `addiu $7, $0, 0x0000000347`, `addiu $7, $zero, 0x1337`, `addiu $7, $0, 0x000000000`, `addiu $7, $zero, 0x400`, `addu $7, $0, 0x00000000d`, `addu $7, $0, $1`, and `syscall`. The right window is Ubuntu-22.04.3-VMware Workstation, showing the output of the `brtest2_dumpsim` program. The output displays the current register/bus values, instruction count (8), and the PC value (0x0040002c). The registers are listed with their values, and the program is finished running.

mtest0

The screenshot displays two windows from a VMware Workstation. The left window, titled 'C:\Users\B6152\OneDrive\save\大三上\体系结构\作业\lab1\inputs\memtest0.s - MARS 4.5', shows the Mars assembly editor. The 'Text Segment' is visible, containing assembly instructions for Coprocessor 0. The 'Data Segment' shows memory addresses and their corresponding values. The 'Registers' window for Coprocessor 0 is also open, showing the state of various registers. The right window, titled 'ubuntu-22.04.3 - VMware Workstation', shows a terminal window running the 'mtest0_dumpsim' program. The terminal output displays the memory content of the simulated system, showing addresses and their corresponding values.

Mars Assembly Editor - Text Segment

Bkpt	Address	Code	Basic	Source
0x00400050	0x8c4f0004	lw \$15, 0x00000004(\$3)	31: lw \$15, 4(\$3)	
0x00400054	0x8c700008	lw \$16, 0x00000008(\$3)	32: lw \$16, 8(\$3)	
0x00400058	0x00098820	add \$17, \$0, \$9	35: add \$17, \$zero, \$9	
0x0040005c	0x022a8820	add \$17, \$17, \$10	36: add \$17, \$17, \$10	
0x00400060	0x022b8820	add \$17, \$17, \$11	37: add \$17, \$17, \$11	
0x00400064	0x022c8820	add \$17, \$17, \$12	38: add \$17, \$17, \$12	
0x00400068	0x022d8820	add \$17, \$17, \$13	39: add \$17, \$17, \$13	
0x0040006c	0x022e8820	add \$17, \$17, \$14	40: add \$17, \$17, \$14	
0x00400070	0x022f8820	add \$17, \$17, \$15	41: add \$17, \$17, \$15	
0x00400074	0x02308820	add \$17, \$17, \$16	42: add \$17, \$17, \$16	
0x00400078	0x2402000a	addiu \$2, \$0, 0x0000000a	45: addiu \$2, \$zero, 0xa	
0x0040007c	0x0000000c	syscall	46: syscall	

Mars Assembly Editor - Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10000000	0x000000ff	0x000000ff	0x000001fe	0x000003fc
0x10000020	0x00000000	0x00000000	0x00000000	0x00000000
0x10000040	0x00000000	0x00000000	0x00000000	0x00000000
0x10000060	0x00000000	0x00000000	0x00000000	0x00000000
0x10000080	0x00000000	0x00000000	0x00000000	0x00000000
0x100000a0	0x00000000	0x00000000	0x00000000	0x00000000
0x100000c0	0x00000000	0x00000000	0x00000000	0x00000000
0x100000e0	0x00000000	0x00000000	0x00000000	0x00000000
0x10000100	0x00000000	0x00000000	0x00000000	0x00000000
0x10000120	0x00000000	0x00000000	0x00000000	0x00000000

Registers (Coprocessor 0)

Name	Nu...	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x0000000a
\$v1	3	0x10000004
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0xffffffff
\$t4	12	0xffffffff
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0xffffffff
\$s0	16	0xffffffff
\$s1	17	0x000179ea
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeff
\$fp	30	0x00000000
\$ra	31	0x00000000

Ubuntu VM - mtest0_dumpsim

```
200 R0: 0x00000000
200 R1: 0x00000000
200 R2: 0x0000000a
200 R3: 0x10000004
200 R4: 0x00000000
200 R5: 0x00000000
200 R6: 0x00000000
200 R7: 0x00000000
200 R8: 0x00000000
200 R9: 0x00000000
200 R10: 0x00000000
200 R11: 0x00000000
200 R12: 0x00000000
200 R13: 0x00000000
200 R14: 0x00000000
200 R15: 0x00000000
200 R16: 0x00000000
200 R17: 0x00000000
200 R18: 0x00000000
200 R19: 0x00000000
200 R20: 0x00000000
200 R21: 0x00000000
200 R22: 0x00000000
200 R23: 0x00000000
200 R24: 0x00000000
200 R25: 0x00000000
200 R26: 0x00000000
200 R27: 0x00000000
200 R28: 0x00000000
200 R29: 0x00000000
200 R30: 0x00000000
200 R31: 0x00000000
200 HI: 0x00000000
200 LO: 0x00000000
```

mtest1

The screenshot displays two windows from a VMware Workstation. The left window, titled 'C:\Users\B6152\OneDrive\code\cpp\ComputerArchitecture\lab1\inputs\memtest1.s - M...', shows the Mars assembly editor. The 'Text Segment' is visible, containing assembly instructions for Coprocessor 0. The 'Data Segment' shows memory addresses and their corresponding values. The 'Registers' window for Coprocessor 0 is also open, showing the state of various registers. The right window, titled 'ubuntu-22.04.3 - VMware Workstation', shows a terminal window running the 'mtest1_dumpsim' program. The terminal output displays the memory content of the simulated system, showing addresses and their corresponding values.

Mars Assembly Editor - Text Segment

Bkpt	Address	Code	Basic	Source
0x00400070	0x846f0004	lh \$15, 0x00000004(\$3)	31: lh \$15	
0x00400074	0x84700008	lh \$16, 0x00000008(\$3)	32: lh \$16	
0x00400078	0x00098820	add \$17, \$0, \$9	35: add \$17	
0x0040007c	0x022a8820	add \$17, \$17, \$10	36: add \$17	
0x00400080	0x022b8820	add \$17, \$17, \$11	37: add \$17	
0x00400084	0x022c8820	add \$17, \$17, \$12	38: add \$17	
0x00400088	0x022d8820	add \$17, \$17, \$13	39: add \$17	
0x0040008c	0x022e8820	add \$17, \$17, \$14	40: add \$17	
0x00400090	0x022f8820	add \$17, \$17, \$15	41: add \$17	
0x00400094	0x02308820	add \$17, \$17, \$16	42: add \$17	
0x00400098	0x2402000a	addiu \$2, \$0, 0x0000000a	45: addiu \$2, \$0	
0x0040009c	0x0000000c	syscall	46: syscall	

Mars Assembly Editor - Data Segment

Address	Value (+0)	Value (+4)	Value (+8)
0x10000000	0x000000c0	0xfecacafe	0xfbebeef
0x10000020	0x00000000	0x00000000	0x00000000
0x10000040	0x00000000	0x00000000	0x00000000
0x10000060	0x00000000	0x00000000	0x00000000
0x10000080	0x00000000	0x00000000	0x00000000
0x100000a0	0x00000000	0x00000000	0x00000000
0x100000c0	0x00000000	0x00000000	0x00000000
0x100000e0	0x00000000	0x00000000	0x00000000
0x10000100	0x00000000	0x00000000	0x00000000
0x10000120	0x00000000	0x00000000	0x00000000

Registers (Coprocessor 0)

Name	Nu...	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x0000000a
\$v1	3	0x10000004
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0xffffffff
\$t4	12	0xffffffff
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0xffffffff
\$s0	16	0xffffffff
\$s1	17	0x000179ea
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$s8	24	0x00000000
\$s9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7ffffeff
\$fp	30	0x00000000
\$ra	31	0x00000000

Ubuntu VM - mtest1_dumpsim

```
200 R0: 0x00000000
200 R1: 0x00000000
200 R2: 0x0000000a
200 R3: 0x10000004
200 R4: 0x00000000
200 R5: 0x00000000
200 R6: 0x00000000
200 R7: 0x00000000
200 R8: 0x00000000
200 R9: 0x00000000
200 R10: 0x00000000
200 R11: 0xffffffff
200 R12: 0xffffffff
200 R13: 0x00000000
200 R14: 0x00000000
200 R15: 0xffffffff
200 R16: 0xffffffff
200 R17: 0x000179ea
200 R18: 0x00000000
200 R19: 0x00000000
200 R20: 0x00000000
200 R21: 0x00000000
200 R22: 0x00000000
200 R23: 0x00000000
200 R24: 0x00000000
200 R25: 0x00000000
200 R26: 0x00000000
200 R27: 0x00000000
200 R28: 0x00000000
200 R29: 0x00000000
200 R30: 0x00000000
200 R31: 0x00000000
200 HI: 0x00000000
200 LO: 0x00000000
```

my_test

C:\Users\B6152\OneDrive\code\cpp\ComputerArchitecture\lab1\inputs\my_test.s - MARS 4.5

File Edit Run Settings Tools Help

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400024	0x014b5020	add \$t2, \$t0, \$t1	16: add \$t4, \$t2, \$t3 # \$
	0x00400028	0x018a5022	sub \$t3, \$t2, \$t0	19: sub \$t5, \$t4, \$t2 # \$
	0x0040002c	0x01a90018	mult \$t3, \$9	22: mult \$t5, \$t1 # 堆\$
	0x00400030	0x15000001	bne \$8, \$0, 0x00000001	25: div \$t7, \$t5, \$t0 # \$
	0x00400034	0x0000000d	break	
	0x00400038	0x01a8001a	div \$t3, \$8	
	0x0040003c	0x00007812	mflw \$t5	
	0x00400040	0x011e9001	beg \$t5, \$9, 0x00000001	28: beq \$t7, \$t1, equal
	0x00400044	0x15e90002	bne \$t5, \$9, 0x00000002	29: bne \$t7, \$t1, not_equal
	0x00400048	0x34180001	ori \$t4, \$zero, 1	32: ori \$t8, \$zero, 1
	0x0040004c	0x08100015	j 0x00400054	33: j end
	0x00400050	0x34180000	ori \$t4, \$zero, 0	36: ori \$t8, \$zero, 0
	0x00400054	0x3c011001	lui \$t1, 0x00001001	40: sw \$t8, result
	0x00400058	0xac380010	sw \$t4, 0x00000010(\$t1)	
	0x0040005c	0x2402000a	addiu \$t2, \$0, 0x0000000a	43: addiu \$v0, \$zero, 10
	0x00400060	0x0000000c	syscall	44: syscall

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x10010000	0x0000000a	0x00000003	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x00000000
0x10010040	0x00000000	0x00000000	0x00000000	0x00000000
0x10010060	0x00000000	0x00000000	0x00000000	0x00000000
0x10010080	0x00000000	0x00000000	0x00000000	0x00000000
0x100100a0	0x00000000	0x00000000	0x00000000	0x00000000

Mars Messages Run I/O

Clear

program is finished running

Coproc 0

Registers

Name	Num.	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x0000000a
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x0000000a
\$t1	9	0x00000003
\$t2	10	0x0000000a
\$t3	11	0x0000000d
\$t4	12	0x00000017
\$t5	13	0x00000004
\$t6	14	0x00000000
\$t7	15	0x00000001
\$t8	16	0x00000000
\$t9	17	0x00000000
\$t10	18	0x00000000
\$t11	19	0x00000000
\$t12	20	0x00000000
\$t13	21	0x00000000
\$t14	22	0x00000000
\$t15	23	0x00000000
\$t16	24	0x00000000
\$t17	25	0x00000000
\$t18	26	0x00000000
\$t19	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc0
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400064
hi		0x00000003
lo		0x00000001

ubuntu-22.04.3 - VMware Workstation

Workstation

ubuntu-22.04.3

活动 文本编辑器 9月29日 03:00

my_test_dumpsim

~/code/ComputerArchitecture/lab1

保存(s)

345 R1: 0x10010000

346 R2: 0x0000000a

347 R3: 0x00000000

348 R4: 0x00000000

349 R5: 0x00000000

350 R6: 0x00000000

351 R7: 0x00000000

352 R8: 0x0000000a

353 R9: 0x00000003

354 R10: 0x0000000a

355 R11: 0x0000000d

356 R12: 0x00000017

357 R13: 0x00000004

358 R14: 0x00000000

359 R15: 0x00000001

360 R16: 0x00000000

361 R17: 0x00000000

362 R18: 0x00000000

363 R19: 0x00000000

364 R20: 0x00000000

365 R21: 0x00000000

366 R22: 0x00000000

367 R23: 0x00000000

368 R24: 0x00000000

369 R25: 0x00000000

370 R26: 0x00000000

371 R27: 0x00000000

372 R28: 0x00000000

373 R29: 0x00000000

374 R30: 0x00000000

375 R31: 0x00000000

376 H1: 0x00000003

377 L0: 0x00000001

378

379

380 Memory content [0x10010000..0x10010010] :

381

382 0x10010000 (268500992) : 0x0000000a

383 0x10010004 (268500996) : 0x00000003

384 0x10010008 (268501000) : 0x00000000

385 0x1001000c (268501004) : 0x00000000

386 0x10010010 (268501008) : 0x00000000

387

纯文本 制表符宽度: 8 第 210 行, 第 16 列 插入

要将输入定向到该虚拟机。请将鼠标指针移入其中或按 Ctrl+G.