



南開大學
Nankai University

计算机学院
体系结构仿真实验一报告

Lab s1: Instruction-Level MIPS Simulator

姓名：曹嘉悦

学号：2113881

专业：计算机科学与技术

2023 年 9 月 30 日

目录

1	实验环境搭建	2
2	实验过程	2
2.1	sim.c 程序的设计	2
2.2	译码的具体实现	4
2.3	结果的具体输出	11
2.3.1	addiu.x	11
2.3.2	arithtest.x	12
2.3.3	brtext.x	14
2.3.4	memtext.x	18

1 实验环境搭建

由于所提供的 asm2hex 不能直接运行，我们需要下载最新版本的 spim764, 并重写 python 脚本，从而实现 16 进制文件输出，完成汇编文件的转化。

名称	修改日期	类型	大小
spimsimulator-code-r764-CPU	2023/9/29 18:15	文件夹	
spimsimulator-code-r764-Document...	2023/9/29 18:15	文件夹	
spimsimulator-code-r764-Setup	2023/9/29 18:15	文件夹	
spimsimulator-code-r764-spim	2023/9/29 18:15	文件夹	
spimsimulator-code-r764-Tests	2023/9/29 18:15	文件夹	

图 1.1: prime 运行结果

2 实验过程

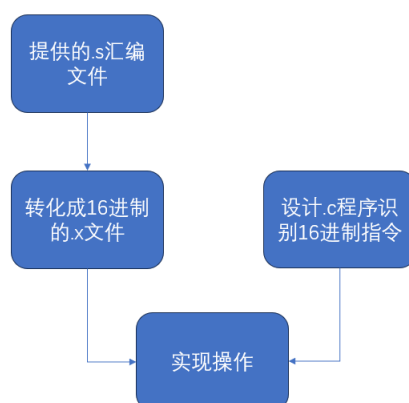


图 2.2: 实验主要核心过程

2.1 sim.c 程序的设计

MIPS 将指令分为了三类：I-Type, J-type, R-type。我们可以按照 op 值把他们分开并划分子集。

I-Type I-Type 指令通常用于执行带有立即数（常数值）的操作，例如加载常数、加载内存中的数据、分支和跳转等。大部分 I-Type 指令可以按照 op 区分开来，但是其中 **BLTZ**、**BGEZ**、**BLTZAL**、**BGEZAL** 是按照 rt 部分分开的。

J-Type J-Type 指令通常用于无条件跳转到程序的其他部分，例如跳转到子程序或跳转到特定地址。我们会按照 op 部分分开。

R-Type R-Type 指令通常用于执行寄存器之间的操作，例如加法、减法、逻辑运算等。我们会按照 funct 部分把它们分开。

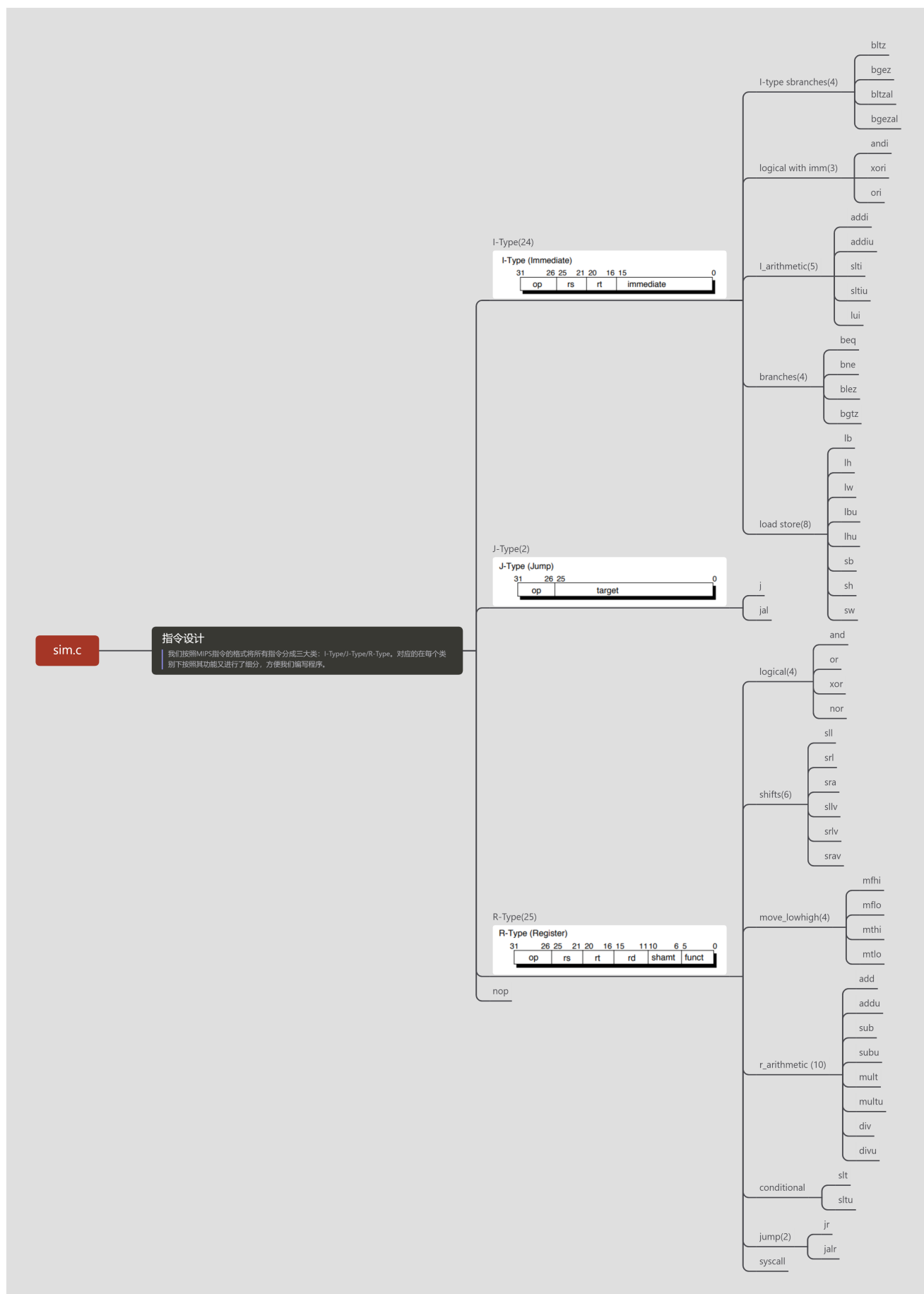


图 2.3: 指令分类

2.2 译码的具体实现

我们把所有的代码已经上交并上传至 github 仓库。我们以一个经典的指令来解释译码过程。

```

1 void jal(uint32_t target)
2 {
3     uint32_t id_low = target << 2;
4     uint32_t id_high = (CURRENT_STATE.PC + 4) & (~0xffffffff);
5     NEXT_STATE.PC = (id_low | id_high) - 4;
6     NEXT_STATE.REGS[31] = CURRENT_STATE.PC + 4;
7 }

```

jal 指令作用为: $pc \leftarrow (pc+4) [31, 28] || \text{target} || '00'$, 转移到新的指令地址。其中新指令地址的低 28 位是指令中的 target 左移两位的值, 新指令地址的高 4 位是跳转指令后面延迟槽指令的地址高 4 位。指令 jal 还要将跳转指令后面第 2 条指令的地址作为返回地址保存到寄存器 \$31。

我们下面逐行直接代码的意思:

设置一个无符号 32 位整数, 保存 target 左移两位后的结果, target 只有 26 位, 高位会由 0 进行填充。移位后, id_low 会有 28 位, 会是新地址的低地址部分, 高位用 0 填充。高地址部分, 我们想要得到延迟槽指令 (当前 PC 紧接的下一条指令) 的高四位。我们所采用的方法是让它和 0xf0000000 进行按位与操作从而只有高四位有意义, 剩余 28 位全部为 0。从而我们将两部分实现按位或的操作, 得到新的 PC 值。**这里由于下面展示的部分, 我们在主函数把 PC 统一更新了一次, 所以为了避免重复操作, 我们需要先进行一次-4 操作。**与此同时, 我们需要把当前跳转指令的下一条指令的地址存储在 31 号寄存器中, 从而作为返回地址返回。

```

1 void process_instruction()
2 {
3     /* execute one instruction here. You should use CURRENT_STATE and modify
4      * values in NEXT_STATE. You can call mem_read_32() and mem_write_32() to
5      * access memory. */
6     // 初始化一个指令
7     uint32_t instruction = 0x0;
8     instruction = mem_read_32(CURRENT_STATE.PC); // 读入当前指令
9     // 初始化指令的各部分
10    uint8_t op = 0;
11    uint8_t rs = 0, rt = 0;
12    uint8_t rd = 0, shamt = 0, funct = 0;
13    uint16_t imm = 0; // 立即数有符号, 先保存后续会进行转化
14    uint32_t target = 0;
15    op = ((instruction >> 26) & 0x3F);
16    rs = ((instruction >> 21) & 0x1F);
17    rt = ((instruction >> 16) & 0x1F);
18    rd = ((instruction >> 11) & 0x1F);
19    shamt = ((instruction >> 6) & 0x1F);
20    funct = (instruction & 0x3F);
21    imm = (instruction & 0xFFFF);
22    target = (instruction & 0x3FFFFFFF);
23    if (instruction == 0) { nop(); }
24    else if (op == 0) { rtype(rs, rt, rd, shamt, funct); }

```

```

25     else if (op == 2 || op == 3) { jtype(op, target); }
26     else { itype(op, rs, rt, imm); }
27 //进入下一条指令
28     NEXT_STATE.PC += 4;
29 }

```

和上述的 jal() 函数类似，其他指令解析的过程都基于其作用，通过读、取、算数运算、位运算等进行了相对应的实现。我们将所有函数声明（分类函数展示所有内容）展示如下：

```

1 void nop()
2 void shift(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t shamt, uint8_t funct)
3 {
4     switch (funct) {
5         case SLL:
6             sll(rt, rd, shamt);
7             return;
8         case SRL:
9             srl(rt, rd, shamt);
10            return;
11        case SRA:
12            sra(rt, rd, shamt);
13            return;
14        case SLLV:
15            sllv(rs, rt, rd);
16            return;
17        case SRLV:
18            srlv(rs, rt, rd);
19            return;
20        case SRAV:
21            srav(rs, rt, rd);
22            return;
23    }
24 }
25 void sll(uint8_t rt, uint8_t rd, uint8_t shamt)
26 void srl(uint8_t rt, uint8_t rd, uint8_t shamt)
27 void sra(uint8_t rt, uint8_t rd, uint8_t shamt)
28 void sllv(uint8_t rs, uint8_t rt, uint8_t rd)
29 void srlv(uint8_t rs, uint8_t rt, uint8_t rd)
30 void srav(uint8_t rs, uint8_t rt, uint8_t rd)
31 void jump(uint8_t rs, uint8_t rd, uint8_t funct)
32 {
33     switch (funct) {
34         case JR:
35             jr(rs);
36             return;
37         case JALR:
38             jalr(rs, rd);
39             return;
40     }

```

```
41 }
42 void jr(uint8_t rs)
43 void jalr(uint8_t rs, uint8_t rd)
44 void syscall()
45 void move_low_high(uint8_t rs, uint8_t rd, uint8_t funct)
46 {
47     switch (funct) {
48     case MFHI:
49         mfhi(rd);
50         return;
51     case MFLO:
52         mflo(rd);
53         return;
54     case MTHI:
55         mthi(rs);
56         return;
57     case MILO:
58         mtlo(rs);
59         return;
60     }
61 }
62 void mfhi(uint8_t rd)
63 void mfLO(uint8_t rd)
64 void mthi(uint8_t rs)
65 void mtlo(uint8_t rs)
66 void arithmetic(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct)
67 {
68     switch (funct) {
69     case ADD:
70         add(rs, rt, rd);
71         return;
72     case ADDU:
73         addu(rs, rt, rd);
74         return;
75     case SUB:
76         sub(rs, rt, rd);
77         return;
78     case SUBU:
79         subu(rs, rt, rd);
80         return;
81     case MULT:
82         mult(rs, rt);
83         return;
84     case MULTU:
85         multu(rs, rt);
86         return;
87     case DIV:
88         div(rs, rt);
89         return;
```

```

90     case DIVU:
91         divu(rs, rt);
92         return;
93     }
94 }
95 void add(uint8_t rs, uint8_t rt, uint8_t rd)
96 void addu(uint8_t rs, uint8_t rt, uint8_t rd)
97 void sub(uint8_t rs, uint8_t rt, uint8_t rd)
98 void subu(uint8_t rs, uint8_t rt, uint8_t rd)
99 void mult(uint8_t rs, uint8_t rt)
100 void multu(uint8_t rs, uint8_t rt)
101 void div(uint8_t rs, uint8_t rt)
102 void divu(uint8_t rs, uint8_t rt)
103 void logical(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct) {
104     switch (funct) {
105     case AND:
106         and (rs, rt, rd);
107         return;
108     case OR:
109         or (rs, rt, rd);
110         return;
111     case XOR:
112         xor (rs, rt, rd);
113         return;
114     case NOR:
115         nor(rs, rt, rd);
116         return;
117     }
118 }
119
120 void and(uint8_t rs, uint8_t rt, uint8_t rd)
121 void or(uint8_t rs, uint8_t rt, uint8_t rd)
122 void xor (uint8_t rs, uint8_t rt, uint8_t rd)
123 void nor (uint8_t rs, uint8_t rt, uint8_t rd)
124 void compare(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t funct)
125 {
126     switch (funct) {
127     case SLT:
128         slt(rs, rt, rd);
129         return;
130     case SLTU:
131         sltu(rs, rt, rd);
132         return;
133     }
134 }
135 void slt(uint8_t rs, uint8_t rt, uint8_t rd)
136 void sltu(uint8_t rs, uint8_t rt, uint8_t rd)
137 void itype_branches_special(uint8_t rs, uint8_t rt, uint16_t imm)
138 {

```



```

139  switch (rt) {
140  case BLTZ:
141      bltz(rs, imm);
142      return;
143  case BGEZ:
144      bgez(rs, imm);
145      return;
146  case BLTZAL:
147      bltzal(rs, imm);
148      return;
149  case BGEZAL:
150      bgezal(rs, imm);
151      return;
152  }
153 }
154 void bltz(uint8_t rs, uint16_t imm)
155 void bgez(uint8_t rs, uint16_t imm)
156 void bltzal(uint8_t rs, uint16_t imm)
157 void bgezal(uint8_t rs, uint16_t imm)
158 void itype_branches(uint8_t op, uint8_t rs, uint8_t rt, uint16_t imm)
159 {
160     switch (op) {
161     case BEQ:
162         beq(rs, rt, imm);
163         return;
164     case BNE:
165         bne(rs, rt, imm);
166         return;
167     case BLEZ:
168         blez(rs, imm);
169         return;
170     case BGTZ:
171         bgtz(rs, imm);
172         return;
173     }
174 }
175 void beq(uint8_t rs, uint8_t rt, int16_t imm)
176 void bne(uint8_t rs, uint8_t rt, int16_t imm)
177 void blez(uint8_t rs, uint8_t rt, int16_t imm)
178 void bgtz(uint8_t rs, uint8_t rt, int16_t imm)
179 void itype_arithmetic_logical(uint8_t op, uint8_t rs, uint8_t rt, uint16_t imm)
180 {
181     switch (op) {
182     case ADDI:
183         addi(rs, rt, imm);
184         return;
185     case ADDIU:
186         addiu(rs, rt, imm);
187         return;

```

```

188     case SLTI:
189         slti(rs, rt, imm);
190         return;
191     case SLTIU:
192         sltiu(rs, rt, imm);
193         return;
194     case ANDI:
195         andi(rs, rt, imm);
196         return;
197     case ORI:
198         ori(rs, rt, imm);
199         return;
200     case XORI:
201         xori(rs, rt, imm);
202         return;
203     case LUI:
204         lui(rt, imm);
205         return;
206     }
207 }
208 void addi(uint8_t rs, uint8_t rt, int16_t imm)
209 void addiu(uint8_t rs, uint8_t rt, uint16_t imm)
210 void slti(uint8_t rs, uint8_t rt, uint16_t imm)
211 void sltiu(uint8_t rs, uint8_t rt, uint16_t imm)
212 void andi(uint8_t rs, uint8_t rt, uint16_t imm)
213 void ori(uint8_t rs, uint8_t rt, uint16_t imm)
214 void xori(uint8_t rs, uint8_t rt, uint16_t imm)
215 void lui(uint8_t rt, uint16_t imm)
216 void load_store(uint8_t op, uint8_t base, uint8_t rt, int16_t imm)
217 {
218     switch (op) {
219     case LB:
220         lb(base, rt, imm);
221         return;
222     case LH:
223         lh(base, rt, imm);
224         return;
225     case LW:
226         lw(base, rt, imm);
227         return;
228     case LBU:
229         lbu(base, rt, imm);
230         return;
231     case LHU:
232         lhu(base, rt, imm);
233         return;
234     case SB:
235         sb(base, rt, imm);
236         return;

```

```

237     case SH:
238         sh(base, rt, imm);
239         return;
240     case SW:
241         sw(base, rt, imm);
242         return;
243     }
244 }
245 void lb(uint8_t base, uint8_t rt, int16_t imm)
246 void lh(uint8_t base, uint8_t rt, int16_t imm)
247 void lw(uint8_t base, uint8_t rt, int16_t imm)
248 void lbu(uint8_t base, uint8_t rt, int16_t imm)
249 void lhu(uint8_t base, uint8_t rt, int16_t imm)
250 void sb(uint8_t base, uint8_t rt, int16_t imm)
251 void sh(uint8_t base, uint8_t rt, int16_t imm)
252 void sw(uint8_t base, uint8_t rt, int16_t imm)
253 void j(uint32_t target)
254 void jal(uint32_t target)
255 //把要求指令按照三种类型分开
256 void rtype(uint8_t rs, uint8_t rt, uint8_t rd, uint8_t shamt, uint8_t funct)
257 {
258     if (funct <= 7) { shift(rs, rt, rd, shamt, funct); }
259     else if (funct == 8 || funct == 9) { jump(rs, rd, funct); }
260     else if (funct == 12) { syscall(); }
261     else if (funct >= 16 && funct <= 19) { move_low_high(rs, rd, funct); }
262     else if ((funct >= 24 && funct <= 27) || (funct >= 32 && funct <= 35)) {
263         arithmetic(rs, rt, rd, funct); }
264     else if (funct >= 36 && funct <= 39) { logical(rs, rt, rd, funct); }
265     else if (funct == 42 || funct == 43) { compare(rs, rt, rd, funct); }
266     else { ; }
267 }
268 void itype(uint8_t op, uint8_t rs, uint8_t rt, uint16_t imm)
269 {
270     if (op == 1) { itype_branches_special(rs, rt, imm); }
271     if (op >= 4 && op <= 7) { itype_branches(op, rs, rt, imm); }
272     else if (op >= 8 && op <= 15) { itype_arithmetic_logical(op, rs, rt, imm); }
273     else if (op >= 32 && op <= 43) { load_store(op, rs, rt, imm); }
274     else { ; }
275 }
276 void jtype(uint8_t op, uint32_t target)
277 {
278     if (op == 2) { j(target); }
279     else if (op == 3) { jal(target); }
280     else { ; }
281 }
282 void process_instruction()

```

其中包含了三种指令的划分，以及其下子集的划分，和所有指令的具体实现。

2.3 结果的具体输出

make 之后，我们通过以下指令开始调试。

```
1 src/sim inputs/addiu.x
```

2.3.1 addiu.x

“go” 之后寄存器内的结果如下图所示：

```
Current register/bus values :
-----
Instruction Count : 7
PC                  : 0x0040001c
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000000
R6: 0x00000000
R7: 0x00000000
R8: 0x00000005
R9: 0x00000131
R10: 0x000001f4
R11: 0x00000243
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000
```

图 2.4: addiu.x 运行结果

我们根据汇编指令对应验证：

```
1      .text
2  __start:  addiu $v0, $zero, 10
3      addiu $t0, $zero, 5
4      addiu $t1, $t0, 300
```

```

5      addiu $t2, $zero, 500
6      addiu $t3, $t2, 34
7      addiu $t3, $t3, 45
8      syscall

```

1. v0 (2 号) 寄存器被赋值为 10, 我们根据结果可以看到 R2 寄存器内部结果变成 10。
2. t0 寄存器被赋值为 5, 我们看到 R8 寄存器内结果变成了 5。
3. t1 寄存器被赋值为 305, 我们看到 R9 寄存器内结果变成了 0x00000131(305)。
4. t2 寄存器被赋值为 500, 我们看到 R10 寄存器内结果变成了 0x000001f4(500)。
5. t3 寄存器在最后两步最终被赋值为 579, 我们看到 R11 寄存器内结果变成了 0x00000243(579)。

2.3.2 arithtest.x

```

Current register/bus values :
-----
Instruction Count : 17
PC                : 0x00400044
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000800
R4: 0x00000c00
R5: 0x000004d2
R6: 0x04d20000
R7: 0x04d2270f
R8: 0x04d2230f
R9: 0x00000400
R10: 0x000004ff
R11: 0x00269000
R12: 0x004d2000
R13: 0x00000000
R14: 0x00000000
R15: 0xfffffb01
R16: 0x00000000
R17: 0x00640000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000

```

图 2.5: arithtest.x 运行结果

```

1 main:
2     addiu    $2, $zero, 1024
3     addu     $3, $2, $2
4     or       $4, $3, $2
5     add      $5, $zero, 1234
6     sll      $6, $5, 16
7     addiu    $7, $6, 9999
8     subu     $8, $7, $2
9     xor      $9, $4, $3
10    xori     $10, $2, 255
11    srl      $11, $6, 5
12    sra      $12, $6, 4
13    and      $13, $11, $5
14    andi     $14, $4, 100
15    sub      $15, $zero, $10
16    lui      $17, 100
17    addiu    $v0, $zero, 0xa
18    syscall

```

1. R2 寄存器对应 \$2, 寄存器内部值为 1024。
2. R3 寄存器对应 \$3, 寄存器内部值为 2048。
3. R4 寄存器对应 \$4, 寄存器内部值为 $(1000\ 0000\ 0000) \mid (0100\ 0000\ 0000) = 1100\ 0000\ 0000 = 0xc00$ 。
4. R5 寄存器对应 \$5, 寄存器内部值为 1234 (0x4d2)。
5. R12 寄存器对应 \$6, 将 R5 寄存器的值左移了 16 进制的 4 位。
6. R7 寄存器对应 \$7, 结果为 $(0x4d20000 + 0x270f) = 0x4d2270f$, 对应保存其中。
7. R8 寄存器对应 \$8, 结果为 $0x4d2270f - 0x400 = 0x4d2230f$ 。
8. 0xc00 异或 0x800 结果为 0x400, 存在 R9 中。
9. 0100 0000 0000 和 0000 1111 1111 异或结果 0100 1111 1111 = 0x4ff 存储在 10 号寄存器中。
10. 0100 1101 0010 0000 0000 0000 0000 右移 5 位得到结果。
11. 算数右移 4 位, 16 进制右移一位, 结果正确。
12. $0x00269000 \& 0x000004d2$ 结果对应归 0。
13. $0xc00 \& 0x64$ 结果对应归 0。
14. 负数通常以补码的形式进行保存。我们对 0x0000004ff 取反加一, 即可得到 0xffffb01。
15. 我们将 0x64 填充在高位, 低位补 0。
16. v0 寄存器被赋值为 10, 我们根据结果可以看到 R2 寄存器内部结果变成 10。

2.3.3 brtext.x

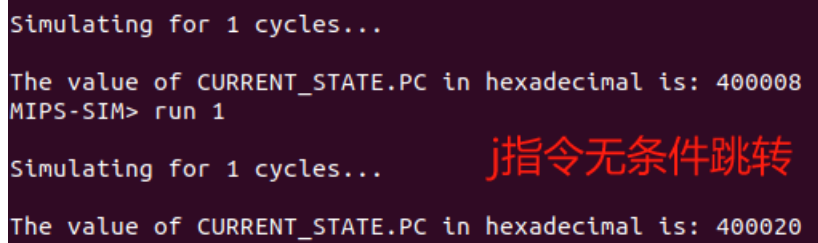
后面每种类型我们只取其中一个进行详细验证，其他文件仅展示结果。

brtext0.x 在 brtext0.x 中有一个 j 指令。

```

1 l_0:
2     addiu $5, $zero, 1
3     j l_1//对应的机器码位08100008
4     addiu $10, $10, 0xf00
5     ori $0, $0, 0
6     ori $0, $0, 0
7     addiu $5, $zero, 100
8     syscall

```



```

Simulating for 1 cycles...
The value of CURRENT_STATE.PC in hexadecimal is: 400008
MIPS-SIM> run 1
Simulating for 1 cycles...
The value of CURRENT_STATE.PC in hexadecimal is: 400020

```

j指令无条件跳转

图 2.6: 地址跳转结果

我们通过运行发现地址由 0x0040 0008 跳转到 0x0040 0020。我们分析其这么跳转的原因：

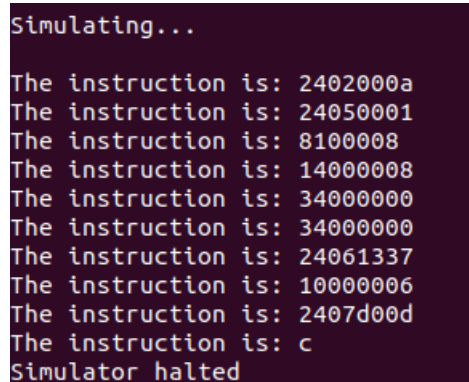
指令的二进制对应为 0000 1000 0001 0000 0000 0000 0000 1000。我们取低 26 位，左移两位后高位补 0（延迟槽指令的高四位为 0）得到 0000 0000 0100 0000 0000 0000 0010 0000。结果对应正确，转去执行 L_1 部分。然后我们顺序执行指令，直到遇到以下指令，再次实现跳转：

```

1 l_2:
2     beq $zero, $zero, l_4

```

最终由于跳转，我们只运行了以下机器码：



```

Simulating...
The instruction is: 2402000a
The instruction is: 24050001
The instruction is: 81000008
The instruction is: 14000008
The instruction is: 34000000
The instruction is: 34000000
The instruction is: 24061337
The instruction is: 10000006
The instruction is: 2407d00d
The instruction is: c
Simulator halted

```

图 2.7: 所有运行指令

我们参考输出和.x 文件。发现实现跳转后直接从 0x1000 0006 转去执行 2407 d00d(addiu \$7, \$zero, 0xd00d)。最终我们输出所有寄存器的信息，证明所有结果都和逻辑一致！

```

Current register/bus values :
-----
Instruction Count : 10
PC                : 0x00400054
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000001
R6: 0x00001337
R7: 0x0000d00d
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000

```

图 2.8: 寄存器结果

我们可以看到：

1. `addiu $v0, $zero, 0xa` 2 寄存器结果为 0xa。
2. `addiu $5, $zero, 1` 使得 5 寄存器结果为 1。
3. `addiu $6, $zero, 0x1337` 使得寄存器结果为 0x1337。
4. `addiu $7, $zero, 0xd00d` 使得寄存器结果为 0xd00d。

我们将剩余两个部分也进行了逻辑验证，结果正确，这里只展示结果。

brtext1.x 下面两张图片分别展示了 `brtext1.x` 对应执行的指令和最终的寄存器结果。这部分的核心在于实现多种跳转，尤其是 `jal` 指令的使用，因为它在跳转的同时还要保存返回地址。


```

MIPS-SIM> go

Simulating...

The instruction is: 2402000a
The instruction is: 24030001
The instruction is: 2404ffff
The instruction is: 24051234
The instruction is: 81000007
The instruction is: 24a50007
The instruction is: c1000005
The instruction is: bf2821
The instruction is: 10000003
The instruction is: 24a50009
The instruction is: 14640002
The instruction is: 24a5000b
The instruction is: 1860fffc
The instruction is: 24a50063
The instruction is: 1c60fffa
the num is 1!
we are going to branch!The instruction is: 24a5006f
The instruction is: 3e000008
The instruction is: 81000116
The instruction is: 24a500d7
The instruction is: c10001a
The instruction is: a62821
The instruction is: 49000002
The instruction is: a62821
The instruction is: 491fffc
The instruction is: 3c01beb0
The instruction is: 3421063d
The instruction is: a12821
The instruction is: c
Simulator halted

```

图 2.9: 指令执行结果 (bretext1.x)

```

Current register/bus values :
-----
Instruction Count : 28
PC                : 0x00400090
Registers:
R0: 0x00000000
R1: 0xbef0063d
R2: 0x0000000a
R3: 0x00000001
R4: 0xffffffff
R5: 0xbef01a59
R6: 0x00000000
R7: 0x00000000
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00400070
HI: 0x00000000
LO: 0x00000000

```

图 2.10: 寄存器结果 (bretext1.x)

brtext2.x 下面两张图片分别展示了 bretext2.x 对应执行的指令和最终的寄存器结果。

Machine Code	Assembly	Description
3c010000	lui \$at, 0x0000	This instruction loads the upper 16 bits of the immediate value into the \$at register
3421d00d	ori \$at, \$at, 0xd00d	This instruction performs an OR operation on the \$at register with the immediate value
00013821	addu \$7, \$at, \$zero	This adds the value in \$at to zero and stores the result in \$7

表 1: Machine Code to Assembly Mapping

在本系列的最后一个文件中，我们调试结果意外发现 1 号寄存器也有了值，但是原汇编文件并没有使用这个寄存器，为了发现其中的原因，我们打开机器码发现在翻译过程中，7 号寄存器的加法过程出现了使用中间变量的过程，虽然笔者认为非常没有必要。

```

Simulating...

The instruction is: 2402000a
The instruction is: 8100002
The instruction is: 14000003
The instruction is: 10000003
The instruction is: 3c010000
The instruction is: 3421d00d
The instruction is: 13821
The instruction is: c
Simulator halted

```

图 2.11: 指令执行结果 (bretext2.x)

```

Instruction Count : 8
PC                : 0x0040002c
Registers:
R0: 0x00000000
R1: 0x0000d00d
R2: 0x0000000a
R3: 0x00000000
R4: 0x00000000
R5: 0x00000000
R6: 0x00000000
R7: 0x0000d00d
R8: 0x00000000
R9: 0x00000000
R10: 0x00000000
R11: 0x00000000
R12: 0x00000000
R13: 0x00000000
R14: 0x00000000
R15: 0x00000000
R16: 0x00000000
R17: 0x00000000
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000

```

图 2.12: 寄存器结果 (bretext2.x)

2.3.4 memtext.x

memtext0.x 这里主要测试的是读取功能的指令。我们核心看这部分代码：

```

1      sw      $5, 0($3)
2      sw      $6, 4($3)
3      sw      $7, 8($3)
4      sw      $8, 12($3)
5
6      lw      $9, 0($3)
7      lw      $10, 4($3)
8      lw      $11, 8($3)
9      lw      $12, 12($3)
10
11     addiu   $3, $3, 4
12     sw      $5, 0($3)
13     sw      $6, 4($3)
14     sw      $7, 8($3)
15     sw      $8, 12($3)
16
17     lw      $13, -4($3)
18     lw      $14, 0($3)
19     lw      $15, 4($3)
20     lw      $16, 8($3)

```

我们绘制流程图如下：

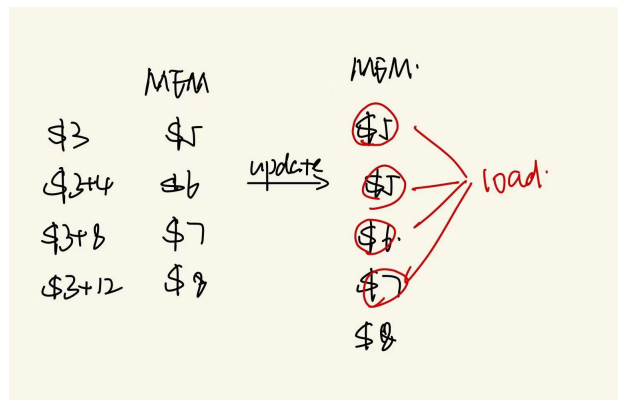


图 2.13: 读存过程

由于我们中间存在对于 3 寄存器中的值加 4 的情况，所有最后内存两块部分的内容实际上是相同，从而我们解释了为什么结果寄存器 13 和 14 中结果一样。

```
Current register/bus values :
-----
Instruction Count : 32
PC                : 0x00400080
Registers:
R0: 0x00000000
R1: 0x00000000
R2: 0x0000000a
R3: 0x10000004
R4: 0x00000000
R5: 0x000000ff
R6: 0x000001fe
R7: 0x000003fc
R8: 0x0000792c
R9: 0x000000ff
R10: 0x000001fe
R11: 0x000003fc
R12: 0x0000792c
R13: 0x000000ff
R14: 0x000000ff
R15: 0x000001fe
R16: 0x000003fc
R17: 0x0000881d
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000
```

图 2.14: 读存寄存器结果 (memtext1.x)

memtext1.x 最后一部分的核心作者认为在于读存字节和半字，尤其是对应的无符号和有符号拓展回 32 位以存储于寄存器中。对应结果如下：

```
Instruction Count : 40
PC                : 0x004000a0
Registers:
R0: 0x00000000
R1: 0x0000efbe
R2: 0x0000000a
R3: 0x10000004
R4: 0x00000000
R5: 0x0000cafe
R6: 0x0000feca
R7: 0x0000beef
R8: 0x0000efbe
R9: 0x000000fe
R10: 0x000000ca
R11: 0xffffffff
R12: 0xffffffff
R13: 0x0000cafe
R14: 0x0000feca
R15: 0xffffbeef
R16: 0xffffefbe
R17: 0x000179ea
R18: 0x00000000
R19: 0x00000000
R20: 0x00000000
R21: 0x00000000
R22: 0x00000000
R23: 0x00000000
R24: 0x00000000
R25: 0x00000000
R26: 0x00000000
R27: 0x00000000
R28: 0x00000000
R29: 0x00000000
R30: 0x00000000
R31: 0x00000000
HI: 0x00000000
LO: 0x00000000

MIPS-SIM>
```

图 2.15: 读存寄存器结果 (memtext1.x)