



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

体系结构仿真实验报告

Lab1

冯思程 2112213

年级：2021 级

专业：计算机科学与技术

指导教师：李雨森

2023 年 9 月 28 日

摘要

本篇报告主要针对指令级 MIPS 模拟器的实现进行了探索，根据 MIPS 的指令集字段特征实现对 53 条 MIPS 指令进行执行，并根据测试文件对实现的 MIPS 模拟器进行测试，结果准确无误，成功实现了模拟器。

关键字：MIPS、模拟器、指令、测试文件

目录

一、 实验环境配置	1
(一) 实验环境与说明	1
二、 实验过程	1
(一) 实验流程总览	1
(二) sim.c 文件的编写	1
1. 划分字段函数	1
2. 扩展函数	2
3. process_instruction 函数实现	3
(三) 测试文件的编写和生成	13
1. 测试文件编写	13
2. .s 文件到.x 文件的转换	14
(四) 测试	15
三、 总结	17
(一) 工作与结果	17

一、实验环境配置

根据实验内容，我首先对实验环境进行了配置，并对必备的一些工具进行安装和测试。

(一) 实验环境与说明

本次实验我在 wsl 中完成，其中具体的环境配置如下：

Windows 版本	wsl 版本	vs code 版本	系统版本	OS 类型	qtspim 版本
windows11	wsl2	1.82.0	Ubuntu 22.04.2 LTS	Linux 64 位	9.1.24

表 1: 实验环境说明表

实验用到的代码可以见 GitHub 这个[链接](#)，工具目前主要用到了 make 命令，这个工具已经在当前环境中配置好。

综上，感谢老师与助教的审查批阅与指正，辛苦！

二、实验过程

(一) 实验流程总览

首先在完成 sim.c 文件 (实现 MIPS 53 条指令) 的基础上，我利用在 /src/ 路径下已经写好的 makefile，用命令 1 编译并链接生成一个目标文件 sim。然后，我再利用 /inputs/ 路径下的一些用于测试的 .s 文件生成的 .x 文件对 sim 进行测试，用到的命令是命令 1。

make

src/sim inputs/addiu.x (以 addiu.x 文件为例)

补充说明：这里从 .s 测试文件到 .x 文件 (16 进制机器码) 的转换是本来是通过 asm2hex 文件实现的，但是由于这里 asm2hex 文件用到的 spim 没有给出，于是利用了 qtspim 进行 .s 文件到 .x 文件的转换。

(二) sim.c 文件的编写

这里的 sim.c 文件中应该对全部的 53 条指令进行实现，这里的输入就是 instruction，即指令。我需要完成的是根据输入的 32 位指令的划分出字段，并根据字段判断指令的类型与具体操作，然后根据操作进行指令的执行。最终我完成了整个 MIPS 模拟器的实现。sim.c 文件的编写主要分为以下三个部分。

1. 划分字段函数

MIPS 指令集包括三种类型的指令：

1. I 型指令：与立即数操作相关的指令，这类指令通常有一个寄存器操作数和一个立即数（即固定的数值）。其字段划分如下：

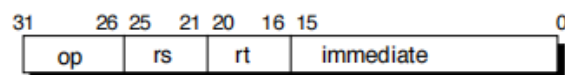


图 1: I 型指令字段组成

2. J 型指令：与跳转相关的指令，用于无条件跳转。它们指定一个跳转目标地址。其字段划分如下：

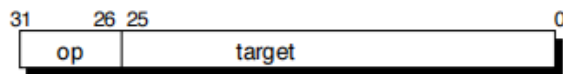


图 2: J 型指令字段组成

3. R 型指令：主要用于寄存器之间的操作的指令，它们通常有三个寄存器操作数：源寄存器 1、源寄存器 2 和目标寄存器。其字段划分如下：

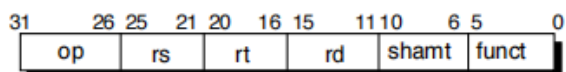


图 3: R 型指令字段组成

根据上文不同类型指令的字段划分，我在 `sim.c` 文件中编写函数进行不同字段的获取，函数的输入都是 32 位指令，然后根据不同字段对应的不同位置，利用**位与**运算的特点进行字段划分，代码如下：

获取字段函数

```
1 //获取instruction的指定字段，其中inst代表32位指令
2
3 uint32_t get_op(uint32_t inst) { return inst >> 26; }
4
5 uint32_t get_rs(uint32_t inst) { return (inst >> 21) & 0x1f; }
6
7 uint32_t get_rt(uint32_t inst) { return (inst >> 16) & 0x1f; }
8
9 uint32_t get_rd(uint32_t inst) { return (inst >> 11) & 0x1f; }
10
11 uint32_t get_targetadr(uint32_t inst) { return inst & 0x3ffffff; } //获取跳转
    目标地址-26位
12
13 uint32_t get_imm(uint32_t inst) { return inst & 0xffff; } //获取立即数字段-16
    位
14
15 uint32_t get_shamt(uint32_t inst) { return (inst >> 6) & 0x1f; }
16
17 uint32_t get_func(uint32_t inst) { return inst & 0x3f; }
```

2. 扩展函数

在指令执行的时候，需要实现 16 位立即数符号扩展、字节符号扩展、半字节符号扩展、32 位立即数零扩展、字节零扩展、半字节零扩展，扩展的目标都是 32 位。其中扩展方法是利用 C 语言的指针和类型转换实现的，代码如下：

扩展函数

```

1 // 将16位立即数符号扩展到32位
2 uint32_t sign_ext(uint32_t imm) {
3     int32_t signed_imm = *((int16_t*)&imm); // 将16位值视为有符号数
4     uint32_t extended_imm = *((uint32_t*)&signed_imm); // 转换为32位数
5     return extended_imm;
6 }
7
8 // 将字节（8位）符号扩展到32位
9 uint32_t sign_ext_byte(uint8_t imm) {
10    int32_t signed_imm = *((int8_t*)&imm); // 将8位值视为有符号数
11    uint32_t extended_imm = *((uint32_t*)&signed_imm); // 转换为32位数
12    return extended_imm;
13 }
14
15 // 将半字（16位）符号扩展到32位，实现与16位立即数的符号扩展相同
16 uint32_t sign_ext_half(uint16_t imm) {
17    int32_t signed_imm = *((int16_t*)&imm); // 将16位值视为有符号数
18    uint32_t extended_imm = *((uint32_t*)&signed_imm); // 转换为32位数
19    return extended_imm;
20 }
21
22 // 将32位立即数零扩展（实际上，它保持不变，只需要转一下类型就可以了），
23 uint32_t zero_ext(uint32_t imm) {
24    return imm; // 32位值不变
25 }
26
27 // 将字节（8位）零扩展到32位，实现类似zero_ext
28 uint32_t zero_ext_byte(uint8_t imm) {
29    return imm; // 在C中，8位无符号值会自动零扩展到32位
30 }
31
32 // 将半字（16位）零扩展到32位，实现类似zero_ext
33 uint32_t zero_ext_half(uint16_t imm) {
34    return imm; // 在C中，16位无符号值会自动零扩展到32位
35 }

```

3. process_instruction 函数实现

这里利用 switch-case 结构根据不同字段值区分不同指令，并针对 53 条指令一一进行实现，展示代码如下（适当注释，标明了每个 case 实现的指令）。

process_instruction 函数

```

1 switch (op) {
2     case 0x0: { // R型指令
3         switch (func) {
4             case 0x0: {
5                 // SLL逻辑左移

```

```

6         NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] << shamt;
7         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
8         break;
9     }
10    case 0x2: {
11        // SRL逻辑右移
12        NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] >> shamt;
13        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
14        break;
15    }
16    case 0x3: {
17        // SRA算术右移
18        int32_t val = *((int32_t*)&CURRENT_STATE.REGS[rt]);
19        val = val >> shamt;
20        NEXT_STATE.REGS[rd] = val;
21        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
22        break;
23    }
24    case 0x4: {
25        // SLLV变量逻辑左移
26        uint32_t shamt = CURRENT_STATE.REGS[rs] & 0x1f;
27        NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] << shamt;
28        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
29        break;
30    }
31    case 0x6: {
32        // SRLV变量逻辑右移
33        uint32_t shamt = CURRENT_STATE.REGS[rs] & 0x1f;
34        NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rt] >> shamt;
35        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
36        break;
37    }
38    case 0x7: {
39        // SRAV变量算术右移
40        int32_t val = *((int32_t*)&CURRENT_STATE.REGS[rt]);
41        uint32_t shamt = CURRENT_STATE.REGS[rs] & 0x1f;
42        val = val >> shamt;
43        NEXT_STATE.REGS[rd] = val;
44        NEXT_STATE.PC = CURRENT_STATE.PC + 4;
45        break;
46    }
47    case 0x8: {
48        // JR无条件跳转
49        NEXT_STATE.PC = CURRENT_STATE.REGS[rs];
50        break;
51    }
52    case 0x9: {
53        // JALR跳转并链接

```

```

54     NEXT_STATE.REGS[rd] = CURRENT_STATE.PC + 4;
55     NEXT_STATE.PC = CURRENT_STATE.REGS[rs];
56     break;
57 }
58 case 0xc: {
59     // SYSCALL系统调用, 这里是按照实验手册的限制性定义进行编写
60     if (CURRENT_STATE.REGS[2] == 0x0a) { // v0寄存器对应reg[2]
61         RUN_BIT = FALSE;
62     }
63     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
64
65     break;
66 }
67 case 0x10: {
68     // MFHI从HI寄存器移动值
69     NEXT_STATE.REGS[rd] = CURRENT_STATE.HI;
70     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
71     break;
72 }
73 case 0x11: {
74     // MTHI将值移动到 HI 寄存器
75     NEXT_STATE.HI = CURRENT_STATE.REGS[rs];
76     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
77     break;
78 }
79 case 0x12: {
80     // MFLO从 LO 寄存器移动值
81     NEXT_STATE.REGS[rd] = CURRENT_STATE.LO;
82     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
83     break;
84 }
85 case 0x13: {
86     // MILO 将值移动到 LO 寄存器
87     NEXT_STATE.LO = CURRENT_STATE.REGS[rs];
88     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
89     break;
90 }
91 case 0x18: {
92     // MULT乘法
93     int64_t lhs = *((int32_t*)&CURRENT_STATE.REGS[rs]);
94     int64_t rhs = *((int32_t*)&CURRENT_STATE.REGS[rt]);
95     int64_t product = lhs * rhs;
96     uint64_t uint_product = *((uint32_t*)&product);
97     NEXT_STATE.HI =
98         (uint32_t)((uint_product >> 32) & 0xffffffff);
99     NEXT_STATE.LO = (uint32_t)(uint_product & 0xffffffff);
100    NEXT_STATE.PC = CURRENT_STATE.PC + 4;

```

```

101         break;
102     }
103     case 0x19: {
104         // MULTH无符号乘法
105         uint64_t lhs = CURRENT_STATE.REGS[rs];
106         uint64_t rhs = CURRENT_STATE.REGS[rt];
107         uint64_t product = lhs * rhs;
108
109         NEXT_STATE.HI = (uint32_t)((product >> 32) & 0xffffffff);
110         NEXT_STATE.LO = (uint32_t)(product & 0xffffffff);
111         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
112         break;
113     }
114     case 0x1a: {
115         // DIV 除法
116         int32_t lhs = *((int32_t*)&CURRENT_STATE.REGS[rs]);
117         int32_t rhs = *((int32_t*)&CURRENT_STATE.REGS[rt]);
118         NEXT_STATE.LO = lhs / rhs;
119         NEXT_STATE.HI = lhs % rhs;
120         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
121         break;
122     }
123     case 0x1b: {
124         // DIVU无符号除法
125         uint32_t lhs = CURRENT_STATE.REGS[rs];
126         uint32_t rhs = CURRENT_STATE.REGS[rt];
127         NEXT_STATE.LO = lhs / rhs;
128         NEXT_STATE.HI = lhs % rhs;
129         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
130         break;
131     }
132     case 0x20: {
133         // ADD加法
134         NEXT_STATE.REGS[rd] =
135             CURRENT_STATE.REGS[rs] + CURRENT_STATE.REGS[rt];
136         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
137         break;
138     }
139     case 0x21: {
140         // ADDU无符号加法
141         NEXT_STATE.REGS[rd] =
142             CURRENT_STATE.REGS[rs] + CURRENT_STATE.REGS[rt];
143         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
144         break;
145     }
146     case 0x22: {
147         // SUB减法
148         NEXT_STATE.REGS[rd] =

```



```

149         CURRENT_STATE.REGS[rs] - CURRENT_STATE.REGS[rt];
150     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
151     break;
152 }
153 case 0x23: {
154     // SUBU无符号减法
155     NEXT_STATE.REGS[rd] =
156         CURRENT_STATE.REGS[rs] - CURRENT_STATE.REGS[rt];
157     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
158     break;
159 }
160 case 0x24: {
161     // AND位与
162     NEXT_STATE.REGS[rd] =
163         CURRENT_STATE.REGS[rs] & CURRENT_STATE.REGS[rt];
164     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
165     break;
166 }
167 case 0x25: {
168     // OR 位或
169     NEXT_STATE.REGS[rd] =
170         CURRENT_STATE.REGS[rs] | CURRENT_STATE.REGS[rt];
171     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
172     break;
173 }
174 case 0x26: {
175     // XOR位异或
176     NEXT_STATE.REGS[rd] =
177         CURRENT_STATE.REGS[rs] ^ CURRENT_STATE.REGS[rt];
178     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
179     break;
180 }
181 case 0x27: {
182     // NOR位非或
183     NEXT_STATE.REGS[rd] =
184         ~(CURRENT_STATE.REGS[rs] | CURRENT_STATE.REGS[rt]);
185     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
186     break;
187 }
188 case 0x2a: {
189     // SLT设置小于
190     int32_t lhs = *((int32_t*)&CURRENT_STATE.REGS[rs]);
191     int32_t rhs = *((int32_t*)&CURRENT_STATE.REGS[rt]);
192     NEXT_STATE.REGS[rd] = (lhs < rhs) ? 1 : 0;
193     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
194     break;
195 }
196 case 0x2b: {

```

```

197         // SLTU无符号设置小于
198         NEXT_STATE.REGS[rd] =
199             CURRENT_STATE.REGS[rs] < CURRENT_STATE.REGS[rt] ? 1 :
200             0;
201         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
202         break;
203     }
204     default: {
205         printf("Unknown instruction: 0x%x\n", inst);
206         break;
207     }
208     break;
209 }
210 case 0x8: {
211     // ADDI立即数加法
212     NEXT_STATE.REGS[rt] = CURRENT_STATE.REGS[rs] + sign_ext(imm);
213     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
214     break;
215 }
216 case 0x9: {
217     // ADDIU无符号立即数加法
218     NEXT_STATE.REGS[rt] = CURRENT_STATE.REGS[rs] + sign_ext(imm);
219     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
220     break;
221 }
222 case 0xc: {
223     // ANDI立即数位与
224     NEXT_STATE.REGS[rt] = CURRENT_STATE.REGS[rs] & zero_ext(imm);
225     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
226     break;
227 }
228 case 0xd: {
229     // ORI立即数位或
230     NEXT_STATE.REGS[rt] = CURRENT_STATE.REGS[rs] | zero_ext(imm);
231     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
232     break;
233 }
234 case 0xe: {
235     // XORI立即数位异或
236     NEXT_STATE.REGS[rt] = CURRENT_STATE.REGS[rs] ^ zero_ext(imm);
237     NEXT_STATE.PC = CURRENT_STATE.PC + 4;
238     break;
239 }
240 case 0x4: {
241     // BEQ相等跳转
242
243     uint32_t offset = sign_ext(imm) << 2;

```

```

244
245     if (CURRENT_STATE.REGS[rs] == CURRENT_STATE.REGS[rt]) {
246         NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
247     } else {
248         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
249     }
250     break;
251 }
252 case 0x1: {
253     uint32_t offset = sign_ext(imm) << 2;
254
255     switch (rt) {
256         case 0x0: {
257             // BLTZ小于0分支
258             if ((CURRENT_STATE.REGS[rs] & 0x80000000) != 0) {
259                 NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
260             } else {
261                 NEXT_STATE.PC = CURRENT_STATE.PC + 4;
262             }
263             break;
264         }
265         case 0x10: {
266             // BLTZAL小于0跳转并链接
267             NEXT_STATE.REGS[31] = CURRENT_STATE.PC + 4;
268             if ((CURRENT_STATE.REGS[rs] & 0x80000000) != 0) {
269                 NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
270             } else {
271                 NEXT_STATE.PC = CURRENT_STATE.PC + 4;
272             }
273             break;
274         }
275         case 0x11: {
276             // BGEZ大于等于0分支
277             if ((CURRENT_STATE.REGS[rs] & 0x80000000) == 0) {
278                 NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
279             } else {
280                 NEXT_STATE.PC = CURRENT_STATE.PC + 4;
281             }
282             break;
283         }
284         case 0x11: {
285             // BGEZAL大于等于0跳转并链接
286             NEXT_STATE.REGS[31] = CURRENT_STATE.PC + 4;
287             if ((CURRENT_STATE.REGS[rs] & 0x80000000) == 0) {
288                 NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
289             } else {
290                 NEXT_STATE.PC = CURRENT_STATE.PC + 4;
291             }

```

```

292         break;
293     }
294 }
295 break;
296 }
297 case 0x5: {
298     // BNE不相等分支
299
300     uint32_t offset = sign_ext(imm) << 2;
301
302     printf("BNE: offset: %d, rs: %d, rt: %d\n", offset, rs, rt);
303
304     printf("rs: 0x%08x\n", CURRENT_STATE.REGS[rs]);
305     printf("rt: 0x%08x\n", CURRENT_STATE.REGS[rt]);
306
307     if (CURRENT_STATE.REGS[rs] != CURRENT_STATE.REGS[rt]) {
308         NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
309     } else {
310         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
311     }
312     break;
313 }
314 case 0x6: {
315     // BLEZ小于等于0分支
316
317     uint32_t offset = sign_ext(imm) << 2;
318
319     if (rt == 0) {
320         if ((CURRENT_STATE.REGS[rs] & 0x80000000) != 0 ||
321             CURRENT_STATE.REGS[rs] == 0) {
322             NEXT_STATE.PC = CURRENT_STATE.PC + offset + 4;
323         } else {
324             NEXT_STATE.PC = CURRENT_STATE.PC + 4;
325         }
326     } else {
327         // Illegal instruction
328         printf("Illegal rt in BLEZ.\n");
329     }
330     break;
331 }
332 case 0x7: {
333     // BGTZ大于0分支
334     uint32_t offset = sign_ext(imm) << 2;
335
336     printf("BGTZ: offset: 0x%08x, rs: %d, rt: %d, pc: 0x%08x\n",
337           offset,
338           rs, rt, CURRENT_STATE.PC);
339 }

```

```

339         if (rt == 0) {
340             if ((CURRENT_STATE.REGS[rs] & 0x80000000) == 0 &&
341                 CURRENT_STATE.REGS[rs] != 0) {
342                 NEXT_STATE.PC = CURRENT_STATE.PC + offset + (uint32_t)4;
343                 printf("PC: 0x%08x\n", NEXT_STATE.PC);
344             } else {
345                 NEXT_STATE.PC = CURRENT_STATE.PC + 4;
346             }
347         } else {
348             // Illegal instruction
349             printf("Illegal rt in BGTZ.\n");
350         }
351         break;
352     }
353     case 0x2: {
354         // J无条件跳转
355         NEXT_STATE.PC = (CURRENT_STATE.PC & 0xf0000000) | (targetadr <<
356             2);
357         break;
358     }
359     case 0x3: {
360         // JAL跳转并链接
361         NEXT_STATE.REGS[31] = CURRENT_STATE.PC + 4;
362         NEXT_STATE.PC = (CURRENT_STATE.PC & 0xf0000000) | (targetadr <<
363             2);
364         break;
365     }
366     case 0xf: {
367         // LUI将立即数加载到寄存器高16位
368         if (rs == 0) {
369             NEXT_STATE.REGS[rt] = imm << 16;
370             NEXT_STATE.PC = CURRENT_STATE.PC + 4;
371         } else {
372             // Illegal instruction
373         }
374         break;
375     }
376     case 0x20: {
377         // LB从内存加载字节
378         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
379
380         uint8_t byte = mem_read_32(addr) & 0xff;
381
382         NEXT_STATE.REGS[rt] = sign_ext_byte(byte);
383         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
384         break;

```

```
385     }
386     case 0x24: {
387         // LBU从内存加载无符号字节
388
389         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
390
391         uint8_t byte = mem_read_32(addr) & 0xff;
392
393         NEXT_STATE.REGS[rt] = zero_ext_byte(byte);
394         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
395         break;
396     }
397     case 0x21: {
398         // LH从内存加载半字
399
400         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
401
402         uint16_t half = mem_read_32(addr) & 0xffff;
403
404         NEXT_STATE.REGS[rt] = sign_ext_half(half);
405         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
406         break;
407     }
408     case 0x25: {
409         // LHU从内存加载无符号半字
410
411         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
412
413         uint16_t half = mem_read_32(addr) & 0xffff;
414
415         NEXT_STATE.REGS[rt] = zero_ext_half(half);
416         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
417         break;
418     }
419     case 0x23: {
420         // LW从内存加载字
421
422         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
423
424         NEXT_STATE.REGS[rt] = mem_read_32(addr);
425         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
426         break;
427     }
428     case 0x28: {
429         // SB将字节存储到内存
430
431         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
432
```

```

433         uint32_t val = (mem_read_32(addr) & 0xfffff00) |
434                         (CURRENT_STATE.REGS[rt] & 0xff);
435
436         mem_write_32(addr, val);
437         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
438         break;
439     }
440     case 0x29: {
441         // SH将半字存储到内存
442
443         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
444
445         uint32_t val = (mem_read_32(addr) & 0xffff0000) |
446                         (CURRENT_STATE.REGS[rt] & 0xffff);
447         mem_write_32(addr, val);
448         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
449         break;
450     }
451     case 0x2b: {
452         // SW 将字存储到内存
453         uint32_t addr = sign_ext(imm) + CURRENT_STATE.REGS[rs];
454
455         mem_write_32(addr, CURRENT_STATE.REGS[rt]);
456         NEXT_STATE.PC = CURRENT_STATE.PC + 4;
457         break;
458     }
459     default: {
460         printf("wrong_inst! at: 0x%08x\n", inst);
461         break;
462     }
463 }

```

(三) 测试文件的编写和生成

1. 测试文件编写

在

addtest1.s

```

1      .text
2      test1:
3          addiu $a0, $zero, 20          # $a0 = 20
4          addiu $a1, $a0, 30           # $a1 = $a0 + 30 = 50
5          addiu $a2, $a1, 40           # $a2 = $a1 + 40 = 90
6          addu   $a3, $a2, $a0         # $a3 = $a2 + $a0 = 110
7          or     $t0, $a3, $a1         # $t0 = $a3 | $a1
8          andi   $t1, $t0, 255         # $t1 = $t0 & 255

```

```

9          sll      $t2, $t1, 3           # $t2 = $t1 << 3
10         sub      $t3, $zero, $t2       # $t3 = -$t2
11         xor      $t4, $t3, $t0         # $t4 = $t3 ^ $t0
12         addiu    $v0, $zero, 0x4       # $v0 = 4
13         syscall

```

addtest2.s

```

1      .text
2      test2:
3          addiu    $t0, $zero, 500       # $t0 = 500
4          addu     $t1, $t0, $t0         # $t1 = $t0 + $t0 = 1000
5          or       $t2, $t1, $t0        # $t2 = $t1 | $t0 = 1500
6          sll      $t3, $t2, 1          # $t3 = $t2 << 1 = 3000
7          subu     $t4, $t3, $t1        # $t4 = $t3 - $t1 = 2000
8          xor      $t5, $t4, $t2       # $t5 = $t4 ^ $t2
9          xori     $t6, $t5, 100        # $t6 = $t5 ^ 100
10         srl      $t7, $t6, 2          # $t7 = $t6 >> 2
11         sra      $t8, $t7, 1          # $t8 = $t7 >> 1 (arithmetic shift)
12         and      $t9, $t8, $t6        # $t9 = $t8 & $t6
13         lui      $s0, 50              # $s0 = 50 << 16
14         addiu    $v0, $zero, 0x5     # $v0 = 5
15         syscall

```

2. .s 文件到.x 文件的转换

这里由于原来的 asm2hex 可执行文件不可用，这里我下载了 9.1.24 版本的 qtspim 进行转换，在[链接](#)安装.deb 文件，然后在 wsl 中进行安装配置。

安装成功后，在命令行输入 qtspim 命令进入其图形化界面，然后通过 load file 的选项加载一个.s 文件，下面以 addiu.s 文件为例进行操作，加载文件后结果如下：

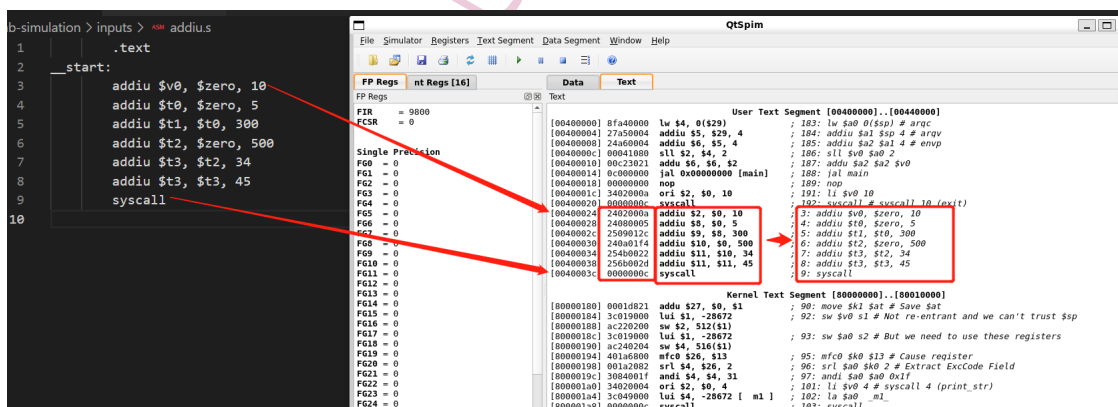


图 4: qtspim 结果

观察发现，.s 文件中的 asm 汇编代码，正好可以与 qtspim 中的 16 进制机器码一一对应，如上图中的红色箭头，然后复制 16 进制机器码并放入一个 addiu.x 文件，完成转换。其他的.s 文件用同样的操作进行转换。

展示 addiu.x 文件的代码如下：


```
addiu.x
1 2402000a
2 24080005
3 2509012c
4 240a01f4
5 254b0022
6 256b002d
7 0000000c
```

(四) 测试

现在我已经完成了 `sim.c` 文件的编写和 `.x` 测试文件的生成，现在只需要进行测试并检验 MIPS 模拟器是否可以正确执行指令。

首先通过 `make` 命令编译链接生成 `sim` 可执行文件，结果如下：

```
fsc@FSC:~/lab-simulation$ cd src
fsc@FSC:~/lab-simulation/src$ ls
Makefile  shell.c  shell.h  sim.c
fsc@FSC:~/lab-simulation/src$ make
gcc -g -O2 shell.c sim.c -o sim
fsc@FSC:~/lab-simulation/src$ ls
Makefile  shell.c  shell.h  sim  sim.c
fsc@FSC:~/lab-simulation/src$
```

图 5: `sim` 可执行文件生成

然后通过 `cd` 命令回到 `src` 上级路径中，实行命令1进行测试，这里以 `addiu.x` 文件为例进行测试并说明，命令执行结果如下，成功进行内置的 `shell` 界面：

`src/sim inputs/addiu.x`(`addiu.x` 为例)

```
fsc@FSC:~/lab-simulation/src$ cd ..
fsc@FSC:~/lab-simulation$ src/sim inputs/addiu.x
MIPS Simulator

Read 7 words from program into memory.

MIPS-SIM>
```

图 6: `addiu.x` 测试

运行 `shell` 中已经内置好的命令 `go` 和 `rdump` 进行检验，结果如下：

```

MIPS-SIM> go
Simulating...
Simulator halted
MIPS-SIM> rdump

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: 0x00001f4
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 7: 检验结果

展示出 addiu.s 文件的代码，如下：

```

addiu.s
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

```

根据代码进行分析 pc 是 0x0040001c, v0 寄存器值为 10, t0 寄存器值为 5, t1 寄存器值为 305, t2 寄存器值为 500, t3 寄存器值为 579。

观察寄存器值, R2 寄存器是 0x0000 000a, R8 寄存器值是 0x0000 0005, R9 寄存器值是 0x0000 0131, R10 寄存器值是 0x0000 01f4, R11 寄存器值是 0x0000 0243。

v0 寄存器对应 R2 寄存器, t0 寄存器对应 R8 寄存器, t1 寄存器对应 R9 寄存器, t2 寄存器对应 R10 寄存器, t3 寄存器对应 R11 寄存器, 经过进制转换后检验发现, 程序正确执行, MIPS 模拟器成功执行这个程序, 而且注意到 pc 的值符合在实验手册中对系统调用命令的限制性要求。

遍历测试 对其他的用于测试的.x 文件一一进行测试，发现均可以正确执行，证明了 MIPS 模拟器的准确性。

三、 总结

(一) 工作与结果

在这次实验中，我主要根据 MIPS 指令的字段特性进行 sim.c 文件的编写，并额外编写了几个测试文件，然后用生成的.x 文件进行 MIPS 模拟器的测试，最后测试结果均正确，证明了我成功的实现了一个 MIPS 模拟器。

NIJUB

参考文献

- [1] README.pdf
- [2] lab s1.pdf
- [3] MIPSISA.pdf

NIKU