



南開大學
Nankai University

计算机学院
体系结构第一次仿真实验

实现 MIPS 指令模拟器

姓名：张铭徐
学号：2113615
专业：计算机科学与技术

2023 年 9 月 26 日

目录

1	输入文件处理部分	2
2	<i>shell.c</i> 修改部分	3
3	<i>sim.c</i> 设计思路	3
4	仿真结果	5
5	链接及实验总结	7
5.1	实验总结	7

1 输入文件处理部分

本次实验拟实现一个 MIPS 仿真软件，能够接收 MIPS 的机器码，然后通过模拟寄存器/内存的读取操作实现基本指令。

实验提供了一个外壳 *shell.c*，是整个 *simulator* 的框架，我们需要实现的是 *sim.c*，是整体 *simulator* 的实现函数，我们需要在 *process_instruction* 函数中实现所有的指令以及对应的输出。

整体的仿真实验的流程如下所示：

- 实现 *process_instruction* 函数
- 使用 *asm2hex* 工具将 */inputs* 中的样例测试输入文件从 MIPS 汇编转换为二进制的机器码，即实现从后缀为.s 到后缀为.x 的过程
- 调用对应的文件，测试 *simulator* 的功能

但是我们可以发现，在 *asm2hex* 工具中，需要配置 *spim* 的工具，*spim* 是一个开源的 MIPS 指令模拟器，实际上我们的目的就是使用该工具实现汇编到机器码的转变。我们可以配置对应的 *spim* 的路径，下载对应工具，但是给出的路径名为 *spim447*，无论是我们使用 *sudo apt install spim* 还是使用给出的链接下载配置，实际上使用的都是已经安装写好的 *simulator*，所以在这次实验中，我们并不使用 *asm2hex*，而是自行使用命令行工具，使用编译器将.s 转换为.x 的文件。

编译命令

```
1 mips-linux-gnu-as -march=mips32 -o addiu.o addiu.s
2 mips-linux-gnu-objcopy -O binary -j .text addiu.o addiu.bin
3 mv addiu.bin addiu.x
```

以 *addiu* 文件为例，我们使用上述命令提取机器码，首先对于第一条指令，我们使用了 GNU 汇编工具，将汇编文件变为.o 文件，使用第二条指令提取.o 文件中特定的段，也即.text 段，该部分存储了指令对应的机器码，最后使用第三个命令，将.bin 文件转存为.x 的文件，该文件并非可执行文件，而是一个.data 类型的文件，里面仅仅存储了机器码，而不存储一些链接信息。

我们还可以使用 *obj* 反汇编工具，对于完整的 *elf* 可执行文件，我们可以使用以下指令得到反汇编：

```
1 mips-linux-gnu-objdump -D addiu.x > addiu\_disassembled.s
```

而对于我们的例子该方法是失效的，因为我们的.x 文件仅仅存储了对应的机器码，所以并不会得到一个完整的反汇编的代码。

在得到 *addiu.x* 后，我们可以使用 *hexdump* 工具，这个工具可以将文件内容以十六进制的形式显示出来。我们使用以下指令来查看文件内容：

```
1 hexdump -C addiu.x
```

在得到该部分后，我们将 *addiu.x* 移动到 */src/sim inputs* 文件夹下方，然后执行以下命令：

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

就可以进入 MIPS Simulator 界面，对于所有的输入文件我们都可以按照这样的流程进行操作，最终得到我们想要的.x 后缀文件。接下来我们考虑修改的 *shell.c* 部分。

2 *shell.c* 修改部分

在这个部分我们主要介绍 *shell.c* 修改的部分。我们可以使用原本的 *shell.c* 文件，但是由于其 *load program* 函数是根据 *spim447* 设计的，我们使用自定义的方法得到的 .x 文件无法被读入进来的，所以我们在这里自定义了一个读入程序码的读入函数。具体函数部分请见 *shell.c* 文件，我们在这里仅仅介绍一下设计的思路。

实际上，对于读入部分，我们仅仅需要将读入部分的二进制码存储到我们内存中的某个区域（通过 *mem_write_32*），然后通过 *mem_read_32* 读取内存中的对应指令。我们使用原本的代码进行读入，读的指令与我们通过 *hexdump* 看到的指令并不相同，这是因为字节顺序不同导致的。*hexdump* 默认以大端字节序显示数据，而我们显示的可能是小端字节序。如果字节序不一致，我们需要将读取到的字的字节顺序进行转换。可以使用 *ntohl* 函数来将网络字节序转换为主机字节序。

ntohl 函数用于将一个 32 位数从网络字节序转换为主机字节序。网络字节序是大端字节序，而主机字节序取决于操作系统。如果系统是小端字节序，那么这个函数会重新排列字节的顺序。如果系统是大端字节序，那么这个函数不会改变任何东西。

此外，我们在代码中添加了部分调试信息，添加了写入的地址和写入的值以便于测试我们上述的读入函数是否正确。对于整体的 *shell.c* 的执行流是：首先初始化内存并读入对应的数据流，然后读取 *simulator* 所需要的指令，我们在这里输入的是 *go*，那么就会调用 *go* 函数，其作用是进行 CPU 的循环 (*cycle* 函数)，根据一个 *bool* 类型的变量 *RUN_BIT* 来看到底当前文件是否执行完，在原本的 *shell.c* 中，我没有看到 *RUN_BIT* 这一变量被更新，所以就会导致一直在执行 *cycle* 函数，即无法完成模拟仿真，所以我们在 *mem_read_32* 中加入了一个判断：如果我们读取到的指令是 *nop* (全 0)，那么就说明我们的文件执行完毕，此时更新 *RUN_BIT* 为 *FALSE*。

这么做实际上是有一些问题的，如果在某一个 MIPS 代码中存在了一些其他的 *nop* 指令，会导致程序过早终止，所以我们还有一个额外的实现思路：我们记录一下写入地址的最高值是多少，然后判断当前在 *mem_read_32* 函数传递的 *address* 参数是否大于了最高地址，如果大于了最高地址，那么就说明当前出现了问题，程序需要终止。

3 *sim.c* 设计思路

事实上，我们本次实验的重头戏是在 *sim.c* 中，上述内容仅仅是配置环境以及搞清楚整体 *simulator* 的执行流，我们实现 *sim.c* 的过程实际上是对 MIPS 指令集进行一系列的解析操作。我们考虑 MIPS 的指令：由于 *simulator* 实际上是接受 32 位的二进制的机器码，这 32 位机器码就决定了我们的模拟器会执行什么样的操作。所以在进行解析之前，我们首先需要熟悉 MIPS32 位的指令的各个字段：指令在计算机中以二进制机器码的形式存储在内存中。同样是 32 位的长度，根据不同的功能，我们将指令分为三种类型：

- I 型指令：I 型指令通常用于实现立即数运算和数据传输等操作，其格式为 *op \$t, \$s, imm*，其中 *op* 表示操作码，*\$t* 和 *\$s* 表示目标寄存器和源寄存器，*imm* 表示一个 16 位的立即数。常见的 I 型指令包括 *addi*、*lw*、*sw*、*andi*、*ori* 等。例如，*addi \$t0, \$s0, 100* 的机器码为 *0x21080064*。
- R 型指令：R 型指令通常用于实现寄存器之间的运算和移位等操作，其格式为 *op \$d, \$s, \$t*，其中 *op* 表示操作码，*\$d*、*\$s* 和 *\$t* 表示目标寄存器、源寄存器和第二个源寄存器。常见的 R 型指令包括 *add*、*sub*、*and*、*or*、*sll*、*srl* 等。例如，*add \$t0, \$s0, \$s1* 的机器码为 *0x02118020*。

- J 型指令：J 型指令通常用于实现无条件跳转操作，其格式为 *j target*，其中 *target* 表示跳转目标地址。常见的 J 型指令包括 *j*、*jal* 等。例如，*j 0x00400014* 的机器码为 *0x0800000d*。

我们整理了如下表格，如??所示：

指令格式	字段	描述	作用和功能	位数范围
I 型指令	op	操作码	指定该指令的具体操作类型	31-26
	rs	源寄存器号	指定源寄存器的编号	25-21
	rt	目标寄存器号	指定目标寄存器的编号	20-16
	imm	立即数	一个 16 位的立即数，用于运算或传输等操作	15-0
J 型指令	op	操作码	指定该指令的具体操作类型	31-26
	addr	跳转地址	一个 26 位的跳转地址，用于实现无条件跳转等操作	25-0
R 型指令	op	操作码	指定该指令的具体操作类型	31-26
	rs	源寄存器号	指定源寄存器的编号	25-21
	rt	第二个源寄存器号	指定第二个源寄存器的编号	20-16
	rd	目标寄存器号	指定目标寄存器的编号	15-11
	shamt	移位位数	一个 5 位的位移量，用于实现移位操作等	10-6
	funct	函数码	一个 6 位的函数码，用于指定具体的操作类型	5-0

表 1: MIPS 指令格式 (使用 `tabularx`)

那么基于我们上面讨论的内容，我们可以将指令先做解析，得到对应的 `opcode`，由于对于所有的指令，其 `opcode` 都是 6 位的，所以我们可以先根据 `opcode`，将指令划分为三种类型，然后在每条指令的内部分别用 `switch - case` 语句模拟对应的指令。

由于我们所要模拟的是 MIPS 程序的执行过程，所以我们仍然需要考虑如何读取指令以及如何更新：我们在读入操作时，将所有的指令的机器码都写入了内存的相应区域，并且使用了一个长度为 32 位的数组来模拟 MIPS 的 32 个通用寄存器的值，在 MIPS 的内部，有一个程序计数器 `program counter` 用于指向当前执行到的指令对应的内存区域，所以我们可以通过 `mem_read_32(pc)` 来读取对应的指令，在每次执行完当前的指令后，可以通过更改 `pc` 的值来达到访问不同的指令区域的效果。

那么问题接踵而至，我们该如何更新 PC 的值呢？我们知道，MIPS 的指令都是 32 位的，也就是 4 字节，且 MIPS 是按照字节寻址的，所以对于非跳转分支指令，也即如果程序是顺序执行，那么我们更新 `pc` 时只需要 `pc = pc + 4` 即可；而对于分支跳转指令，则需要将 `pc` 的值更新为跳转后的目的地地址。

对于指令的模拟过程，只需要直接访问寄存器对应的数组，就可以模拟 MIPS 体系结构计算机对于寄存器的一些操作，所以，以 `add` 指令为例，我们可以写出一下的代码：

add 模拟示例

```

1  uint32_t instruction = mem_read_32(CURRENT_STATE.PC);
2  uint32_t opcode = (instruction & 0xFC000000) >> 26;
3  uint32_t funct = instruction & 0x3F;
4  uint32_t rs = (instruction & 0x03E00000) >> 21;
5  uint32_t rt = (instruction & 0x001F0000) >> 16;
6  uint32_t rd = (instruction & 0x0000F800) >> 11;
7  uint32_t shamt = (instruction & 0x000007C0) >> 6;
8  NEXT_STATE.REGS[rd] = CURRENT_STATE.REGS[rs] + CURRENT_STATE.REGS[rt];

```

```

9     printf("There is ADD instructions!");
10    printf("The operands is:%d %d\n",CURRENT_STATE.REGS[rs],CURRENT_STATE.REGS[rt]);
11    printf("The answer is :%d\n",NEXT_STATE.REGS[rd]);

```

在进行模拟时，同样是更新 PC 的问题，对于跳转指令（即 J 型指令），我们在计算 *targetaddress* 的时候，假设跳转的目的地址是第 10 条指令，那计算得到的 *targetaddress* 的值是 40，但是事实上，我们不可以直接让 $pc = targetaddress$ ，因为我们的内存的起始地址并不是 0，所以我们需要令 $pc = basic + targetaddress$ 。在更新 PC 时，我们引入了一个变量 *update*，对于跳转和分支指令，如果跳转了地址，我们就令 $update = 1$ ，否则 $update = 0$ ，在执行完当前指令后，我们在末尾进行判断，如果 $update = 1$ 则不需要 $pc = pc + 4$ ，否则则需要执行该操作：

```

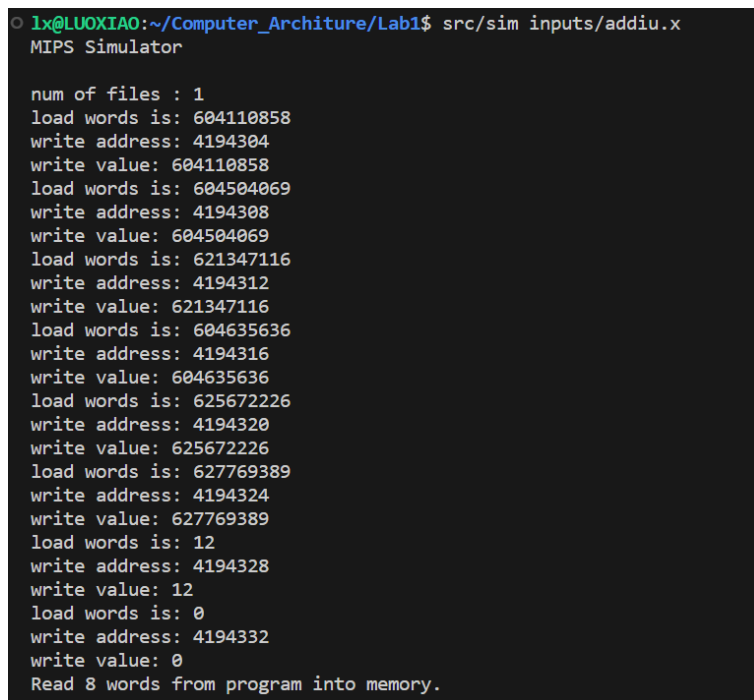
1    if(!update) NEXT_STATE.PC = CURRENT_STATE.PC + 4 ;

```

以上便是我们本次实验的基本设计思想，具体的代码请参考附带的文件内的内容，我们添加了一些中间变量输出以确保我们可以检验代码的正确性。

4 仿真结果

我们在这里，以 *addiu.x* 为例，给出仿真的对应的运行结果图4.1和图4.2：



```

lx@LUOXIAO:~/Computer_Architue/Lab1$ src/sim inputs/addiu.x
MIPS Simulator

num of files : 1
load words is: 604110858
write address: 4194304
write value: 604110858
load words is: 604504069
write address: 4194308
write value: 604504069
load words is: 621347116
write address: 4194312
write value: 621347116
load words is: 604635636
write address: 4194316
write value: 604635636
load words is: 625672226
write address: 4194320
write value: 625672226
load words is: 627769389
write address: 4194324
write value: 627769389
load words is: 12
write address: 4194328
write value: 12
load words is: 0
write address: 4194332
write value: 0
Read 8 words from program into memory.

```

图 4.1: 初始化部分

对于初始化部分，我们可以看到，能够成功的读取到对应的机器码，这需要归功于我们使用了 *ntohl* 函数来将网络字节序转换为主机字节序，同时，我们也可以将对应的机器码写入到内存的对应区域中，最后一共有 8 条指令被载入内存中（最后一条指令为空指令）。我们在读入函数部分使用了如下代码：

读入部分

```

1    while (fread(&word, sizeof(uint32_t), 1, prog) == 1) {

```

```

2     word = ntohl(word); // 转换字节顺序
3     printf("load words is: %u\n",word);
4     mem_write_32(MEM_TEXT_START + ii , word);
5     ii += 4;
6 }

```

```

call for go
Simulating...

Now the address is :4194304
instruction is : 604110858
opcode is : 9
the imm is : 10
now is: 0
the answer is : 10
Now the address is :4194308
instruction is : 604504069
opcode is : 9
the imm is : 5
now is: 0
the answer is : 5
Now the address is :4194312
instruction is : 621347116
opcode is : 9
the imm is : 300
now is: 5
the answer is : 305
Now the address is :4194316
instruction is : 604635636
opcode is : 9
the imm is : 500
now is: 0
the answer is : 500
Now the address is :4194320
instruction is : 625672226
opcode is : 9
the imm is : 34
now is: 500
the answer is : 534
Now the address is :4194324
instruction is : 627769389
opcode is : 9
the imm is : 45
now is: 534
the answer is : 579
Now the address is :4194328
instruction is : 12
opcode is : 0
Now the address is :4194332
instruction is : 0
opcode is : 0
Simulator halted

```

图 4.2: 模拟部分

我们可以看到，对于模拟过程，我们可以从对应的内存区域中读取到正确的指令并正确的解析其指令类型，我们 *addiu.s* 的内容如下：

```

1     .text
2     .globl main
3 main:
4     addiu $v0, $zero, 10
5     addiu $t0, $zero, 5
6     addiu $t1, $t0, 300

```

```
7      addiu $t2, $zero, 500
8      addiu $t3, $t2, 34
9      addiu $t3, $t3, 45
10     syscall
```

那么我们可以看到，确实是模拟出来的若干条指令，说明模拟器的功能是完备的。由于篇幅限制，我们在这里不演示剩余的几组测试样例，在调试中我们的代码是可以正确运行完附带的所有样例文件。

5 链接及实验总结

我们遵循程老师的 DOCX 倡议，将本次实验代码开源，本次实验所有的代码都可以在[体系结构实验仓库](#)中找到。

5.1 实验总结

本次实验我们实现了一个 MIPS32 位指令集的 C 语言模拟器，该模拟器读入一个数据文件 *.x* 即对应指令的机器码，自动存储到模拟出来的内存的对应区域中，然后可以根据 *pc* 的值读取到对应的机器指令，可以根据实现的 *sim.c* 解析对应的指令并模拟运行过程。我们在本次实验报告中给出了较为详细的设计思路和整体项目的概述。

对于我们实现的 *simulator*，与已经实现的 *spim* 相比，功能相差不大，但是其可以可视化的展示出地址区域存储的指令以及存储的变量变化的情况，要比我们实现的模拟器的可读性高一些，我们后续可以基于此对已经实现的项目进行持续的迭代 *gen*