体系结构

lab1 MIPS模拟器

本次实验是一个环境初始搭建的实验，代码部分并没有特别困难的问题。因为老师也没要求实验报告有什么太高的要求，所以实验报告我就偷偷懒了，简单介绍一下我的实验流程和思路吧。

本次实验代码及相关文件已上传到github上：155chb/archlab (github.com)

姓名：蔡鸿博 学号：2112157

第一步 —— .s文件转.x文件

这一步几乎是用时最长的了，但最终也依然没有得到很好的解决。

开始时遇到的问题

首先是老师发的文件中，inputs文件夹下的asm2hex，按实验手册所说应该就是使用这个文件和SPIM模拟器对文件进行转换。但是这一步骤出现了不少问题，比如一开始看不懂老师给的下载链接，下载错成了qtspim（但也没有浪费，最后SPIM没有走通，qtspim帮了我很多）。

看群里有同学说是版本问题，我既下载了老版本的SPIM447，还下载了最新版的SPIM764，但最后其实都有问题。脚本文件中所写的vasm那个参数根本不是SPIM的可识别参数，不清楚它是什么意思，SPIM就自动将其忽略了，这条命令本身大抵就是有问题的，尝试了很多办法，也在网络上搜索了一下解决方法，但都没有实际的进展。即使最后实验做完了，我依然没有办法使用SPIM打开.s文件并导出成.x文件。

第一个解决方法，SPIM暴力复制粘贴

不过还好，我在配置环境前就阅读过shell文件，知道其实.x文件的格式要求就是16进制机器码的文本形式。因此，不论如何，都有最简单的一个办法，我一开始就是使用的这个方法进行了前几个验证。那就是，使用qtspim打开.s文件，这时就可以看到每一条命令对应的机器码，直接将其复制粘贴到一个.x文件就好了，至于.x文件，Windows直接用.txt文件改扩展名就好了，或者Ubuntu直接vim创建一个就行。

第二个解决方法，MARS导出

后来写完实验验证了一半之后，我感觉这样的效率还是太低，因此继续在网上冲浪搜索一些可用的方法。然后我找到了一个应用MARS（http://courses.missouristate.edu/kenvollmar/mars/），这个应用就可以在打开.s文件后直接dump16进制机器码到一个文件中。因此后几个使用的MARS导出.x文件，格式也是正确的（基本正确）。

虽然MARS导出的文件和SPIM中复制粘贴下来的机器码，大致是相同的，但还是有一点不同。在MARS的i型分支跳转指令，其机器码offset的数值总是比SPIM的机器码的offset小1。我也不清楚这是什么原因，一开始以为是延迟槽的问题，发现有没有延迟槽都是那个数值。我也没有继续深究下去，直接将导出的数值-1继续进行验证实验，以后如果有时间可能会继续研究一下。

第二步 —— 统计实验要求完成的指令格式

这一步就是有点耗时，内容比较枯燥。为了保证我编写程序的准确性，以及能够尽快捋清楚思路，我首先想到的是，统计好实验要求的每一条指令的格式，根据指令格式设置宏定义，对指令格式进行分类，更好更快的设计程序框架区分每一条指令。

统计指令格式，只需要参照老师发的文件中的MIPSISA.pdf文件就好，就是过程比较枯燥，但也是为了保证不出错。创建了一个表格文件，按顺序记录下每一条要求的指令，分别对指令的固定数值进行记录，如（接下来是从我编辑的xlsx文件复制来的一小段，实验相关内容上传到github上了，这个文件也在本次实验的文件夹下）

		6	**5**	**5**	**5**	**5**	**6**	
35	ADD	000.000				00.000	100.000	add
7	ADDI	001.000						add immediate
8	ADDIU	001.001						add immediate unsigned

正如表格中的格式，将本次实验要求的每一条指令统计了下来，通过对内容的统计，我很快也写好了整个process_instruction()函数的框架，如下

```
# 宏定义开始
...
# 宏定义结束
void process_instruction()
{
    # 取指令并拆分
    ...
    # 判断指令
    if (OP == 0)
    {
        switch(OP)
        {
        ...
        }
    }
    else
    {
        switch(FUNC)
        {
        ...
        }
    }
}
```

之后根据统计的数值，设置宏定义，将宏定义应用于switch语句中，整体脉络还是十分清晰的。

第三步 —— 取指令以及实现所有指令功能

取指令的部分比较好写，而且内容不多，比较清晰，因此我就直接把代码列在这里了，不多做分析。这里还是比较简单的。我在这里把PC数值一开始就修改了一次，如果后面有跳转语句再修改一次，不冲突。

```
NEXT_STATE.PC = CURRENT_STATE.PC + 4;
uint32_t INST = mem_read_32(CURRENT_STATE.PC);
uint8_t
    OP = INST >> 26,
    RS = (INST >> 21) & 0x1F,
    RT = (INST >> 16) & 0x1F,
    RD = (INST >> 11) & 0x1F,
    SHAMT = (INST >> 6) & 0x1F,
    FUNC = INST & 0x3F;
uint16_t IMM = INST & 0xFFFF;
uint32_t TARGET = INST & 0x03FFFFFF;
```

而对于所有指令的功能，我依然是沿着MIPSISA中的介绍，一个一个进行实现的，但毕竟有53条指令呢，我这里也不全介绍了，简单分类介绍一下好了。介绍一下，编写代码时所想到的一些问题，以及遇到的一些问题，以及解决方法。

有符号和无符号

我遇到的第一个让我花了心思考虑的问题就是有符号和无符号的区别。直接说我最后得到的结论，注意大小比较和乘除时就好。

有符号和无符号的影响，我大致将其分为三类：

- 在不考虑溢出的情况下，完全没有影响的。如加减运算
- 需要考虑，会对结果有影响的。如乘除运算，两个有符号数的大小比较
- 符号扩展。所有立即数是否需要进行符号扩展，这也是一个问题

先说**第一类**，由于数值在计算机中是按照补码的形式保存的，所以加减运算直接算就好了，不需要考虑有符号和无符号的问题（我是这么想的，简单测试了一下没发现什么问题，也有可能我测试的不全面吧）。总之，对于如addi和addiu，add和addu，sub和subu我都是没有做区分的，它们两者的代码是相同的。

第二类，这一点其实还是比较明确的，因为在程序中，-5*4和5*4的结果区别还是蛮大的。我想了一段时间，确实想到比较好的处理方案，所以就直接使用了类型强转。如除法DIV的运算我就直接写作了这样，如下

```
case FUNC_DIV:
    NEXT_STATE.LO = (int32_t)CURRENT_STATE.REGS[RS] /
(int32_t)CURRENT_STATE.REGS[RT];
    NEXT_STATE.HI = (int32_t)CURRENT_STATE.REGS[RS] %
(int32_t)CURRENT_STATE.REGS[RT];
```

这个方法比较直接，因为REGS的类型是uint32_t嘛，所以给他强制转换成int32_t就好了不是吗，经过测试结果是准确的。（这里不举MULT的例子，是因为MULT还有一些其他的小问题，会在后面提到的）同理，SLT等两个数的比较指令也是这样处理的，跟0作比较的指令就直接判断符号位就行了。

第三类，符号扩展是个麻烦事，我也是在看那个MPISISA的手册时才发现忘记了符号扩展的问题。符号扩展倒还算好做，一个宏定义就解决了，如下

```
/* 定义符号扩展 */
#define SIGN_EXTEND(V, W) ((V) | (((V) & (1 << ((W)-1))) ? 0xFFFFFFFF << (W) :
0))
```

主要问题还是需要考虑，哪些立即数需要进行符号扩展，哪些不需要进行符号扩展。

可能还是理论课学的不够扎实，这里确实是没记起来老师上课是怎么介绍的了。一开始我以为，带u的就不需要进行符号扩展，而不带u的就需要进行符号扩展。后来写一半拿SPIM运行的时候发现和一开始想的不太一样。因为我尝试了以下汇编指令（直接在这里说了一下我的验证过程，因为我懒得给它单独写一段了）

```
addiu $7, $zero, -5
```

我惊奇的发现，这一条指令在SPIM中，是先进行了符号扩展后，再进行的add操作。想一想确实能想清楚，进不进行符号扩展对于这条指令的结果是由很大影响的，一个可以理解为加0xFFFFFFFB，而一个是加0x0000FFFB，因此我意识到了我一开始的错误理解。

之后又进行了简单的摸索，我发现，其实对于立即数，MPIS是说明或者说规定了每条指令的立即数的范围。对于andi，ori，xori这三个位操作，规定了立即数为非负数，因此这三个不需要进行符号扩展；而对于其他addi，subi等操作，就需要先将立即数进行符号扩展，然后才能进行运算。举个例子，如下

```
case OP_ADDI:
    NEXT_STATE.REGS[RT] = CURRENT_STATE.REGS[RS] + SIGN_EXTEND(IMM, 16);
    break;
case OP_ANDI:   /* 注意：andi的立即数要求是非负整数 */
    NEXT_STATE.REGS[RT] = CURRENT_STATE.REGS[RS] & IMM;
    break;
```

内存读写

本次实验中，实验手册中就已经说明了是按照小端的形式存储数据。我这个问题主要还是在于SB和SH指令。一开始不是很清楚，其实就是脑子一抽，自己把自己绕糊涂了。总之看了一眼mem_read_32()函数的结构，其实也就大致清楚应该修改哪个数值了。

因为LB和LH指令，一个按字节读数据，一个按半字读数据，而mem_read_32()函数每次都是按照一个字进行数据的读取，而mem_write_32()函数也是按照一个字进行数据的写访问。为了确保其他三个字节或者高半字不会被修改，就需要现读取目标地址的一字数据，将目标位置修改好后，再写回到内存中。

就是在这个步骤，想不清楚读取之后的数据到底应该修改低n位还是修改高n位。所以看了一眼mem_read_32()函数，如下

```
uint32_t mem_read_32(uint32_t address)
{
    int i;
    for (i = 0; i < MEM_NREGIONS; i++) {
        if (address >= MEM_REGIONS[i].start &&
                address < (MEM_REGIONS[i].start + MEM_REGIONS[i].size)) {
            uint32_t offset = address - MEM_REGIONS[i].start;

            return
                (MEM_REGIONS[i].mem[offset+3] << 24) |
                (MEM_REGIONS[i].mem[offset+2] << 16) |
                (MEM_REGIONS[i].mem[offset+1] <<  8) |
                (MEM_REGIONS[i].mem[offset+0] <<  0);
        }
    }

    return 0;
```

```
}
```

很清楚的能看到，mem中最低地址（同时也是目标地址），被写在了返回值的最低8个有效位上，这也符合小端的定义。因此，我意识到了，最低几位就是我需要进行修改的数据。可惜的是，我又开始糊涂了，SB很好弄，就是最低的8位就了。但SH呢，则需要写两个8位，这两个8位修改的时候，哪个写上哪个的数值又搞不清楚了。

后来我又想到，具体数值和内存数值大小端的区别，只在内存读写的才发生。我既然是要Store Halfword，那就只需要修改那Halfword就好了，它原本在什么位置就给它放到什么位置就好，完全不需要考虑那么多。例如，REG3假如调用的是同一个地址的LW指令，也许它得到了0x12345678，那么低16位其实就是我需要的半字，因此就是用掩码得到0x00005678就好了，这样我就得到了关于内存读的代码，如下

```
case OP_SB:
    ADDR = CURRENT_STATE.REGS[RS] + SIGN_EXTEND(IMM, 16);
    TARGET = mem_read_32(ADDR);
    TARGET = (TARGET & 0xFFFFFF00) | (NEXT_STATE.REGS[RT] & 0x000000FF);
    mem_write_32(ADDR, TARGET);
    break;
case OP_SH:
    ADDR = CURRENT_STATE.REGS[RS] + SIGN_EXTEND(IMM, 16);
    TARGET = mem_read_32(ADDR);
    TARGET = (TARGET & 0xFFFF0000) | (NEXT_STATE.REGS[RT] & 0x0000FFFF);
    mem_write_32(ADDR, TARGET);
    break;
case OP_SW:
    ADDR = CURRENT_STATE.REGS[RS] + SIGN_EXTEND(IMM, 16);
    mem_write_32(ADDR, NEXT_STATE.REGS[RT]);
    break;
```

这里只是给出了Store相关的代码，但是Load的代码也是殊途同归，大差不差的。只是需要考虑保存哪几位就好，甚至比Store的命令更容易翻译和执行。

一些需要考虑考虑的指令

JALR：

这条指令挺奇怪的，我一开始以为他就是和从J到JR类似，只需要把立即数改为寄存器的数值就行。后来突然发现，它居然是本次实验中唯一一个有两副面孔的指令！

```
jalr rs
jalr rd, rs
```

以上两条指令都属于它的汇编指令形式，不过好在也就这样。其实它的机器码格式是一样的，汇编器会自动把上一条默认成rd是寄存器31，也就是ra寄存器。因此其实也没有影响，最后使用一套就可以了，不需要多做判断。它的指令还是蛮简单的

```
case FUNC_JR:
    NEXT_STATE.PC = CURRENT_STATE.REGS[RS];
    break;
```

MULT和MULTU：

这两条指令我是后来验证的时候发现存在的问题。C语言中，无论是int32_t还是uint32_t，两个同类型数值进行程序计算式，它都默认将结果存储在一个临时的int32_t或uint32_t中，这就导致我一开始的这几行代码其实是错误的，如下

```
case FUNC_MULT:
    TEMP = (int32_t)CURRENT_STATE.REGS[RS] *
(int32_t)CURRENT_STATE.REGS[RT];
    NEXT_STATE.LO = TEMP & 0xFFFFFFFF;
    NEXT_STATE.HI = TEMP >> 32;
    break;
case FUNC_MULTU:
    TEMP = CURRENT_STATE.REGS[RS] * CURRENT_STATE.REGS[RT];
    NEXT_STATE.LO = TEMP & 0xFFFFFFFF;
    NEXT_STATE.HI = TEMP >> 32;
    break;
```

这样计算结果其实是错误的，比如我测试了下面两种结果比较大的情况

```
# 这里就简单写一下，实际的指令肯定是要改改，就是说明一下是这两个数值
mult    0x0fff0000, 0x00000fff
multu   0x00000009, 0xfffffffb
```

正常理解上，这两条指令应该分别会得到结果（TEMP的数值）0x 0000 00ff e001 0000和0x 0000 0008 ffff ffd3，但是实际运算的结果确是0x ffff ffff e001 0000和0x 0000 0000 ffff ffd3。分析一下原因，认识到了，实际上无论是int32_t还是uint32_t，计算后高于32位的结果都被丢弃掉了。

这样分析的结果就是，实际上右边算式得到的是0x e001 0000和0x ffff ffd3。然后将等式右侧的计算结果赋值给左侧的TEMP时，发生了数据的类型转换。从int32_t到uint64_t，它是先进行了符号扩展得到数据类型int64_t，让后将数值赋给了uint64_t；而从uint32_t到uint64_t，就没有发生符号扩展，直接高位补零，得到了最后的结果。

虽然分析出了问题可能的原因，但也没想出特别的好的解决方法，所以就还是老一套的，类型强制转换，将数据类型扩展到64位再进行计算，由此得到了以下代码

```
case FUNC_MULT: /* 这里不知道是什么原因，计算结果是当作int32_t进行存储,所以先扩展一下再
计算 */
    TEMP = (int64_t)CURRENT_STATE.REGS[RS] *
(int64_t)CURRENT_STATE.REGS[RT];
    NEXT_STATE.LO = TEMP & 0xFFFFFFFF;
    NEXT_STATE.HI = TEMP >> 32;
    break;
case FUNC_MULTU:    /* 同mult */
    TEMP = (uint64_t)CURRENT_STATE.REGS[RS] *
(uint64_t)CURRENT_STATE.REGS[RT];
    NEXT_STATE.LO = TEMP & 0xFFFFFFFF;
    NEXT_STATE.HI = TEMP >> 32;
    break;
```

修改过后的代码，也成功计算出了正确的数值。

SYSCALL:

这条指令其实没什么好说的。按照实验手册的弄就完了，判断$v0寄存器的数值，然后修改RUN_BIT就可以了，代码如下

```
case FUNC_SYSCALL:
    if (CURRENT_STATE.REGS[2] == 0x0000000A)
        RUN_BIT = FALSE;
    else ;
    break;
```

错误处理：

如果发现识别到了无法识别的代码，我认为还是需要报告一下的，因此简单写了一下，让它报个错（真的很简单）

```
default:
    printf("Unknown instruction: 0x%08x\n", INST);
    RUN_BIT = FALSE;
    break;
```

第四步 —— 验证程序正确性

之前就说过了，这里不想多做介绍，所以就这样吧。我其实就运行了一遍老师给的代码，自己写了一点alu.s的代码，包含了乘除的操作（我看老师给的没有）。简单看了一下计算的正确性，能改的都改完了，上面也都提到了，发现了什么问题，有一部分就是在这一步骤发现的。