

Assembler and Simulator Porject

3180106061 混合1801 朱帅龙

Assembler and Simulator Porject

- 1 Pivotal project requirements
- 2 Some special and interesting C functions
- 3 Code analysis
 - 3.1 asm.c
 - 3.1.1 distinguish code and data
 - 3.1.2 .fill distinguish label and number
 - 3.2 riscv-small.c
- 4 Code implementation
 - 4.1 Assembler part
 - 4.1.1 R-type instruction
 - 4.1.2 I-type instruction(addi , lw)
 - 4.1.3 sw instruction
 - 4.1.4 beq instruction
 - 4.2 Simulator part
 - 4.2.1 decode
 - 4.2.2 excution
- 5 Test bench outcome
 - 5.1 simple.s
 - 5.2 my_test.s

1 Pivotal project requirements

我们主要关注以下三个要求，也是很可能会出现问题的地方：

- 1. beq 指令执行的是相对地址;
- 2. lw, sw, beq指令的立即数域既可能是value也可能是label;
- 3. imm数是有符号数。

2 Some special and interesting C functions

- 1. void rewind(FILE* stream)

重新定位文件指针的文件流的开始位置；

- 2. int atoi(const char* str)

把参数str所指向的字符串转换为一个整数;

- 3. char *strtok(char *str, const *delim)

分解字符串 str 为一组字符串， delim 为分隔符; 特别注意其用法，调用顺序一定要保持如下：

```
1 token = strtok(str, s);
2 token1=strtok(NULL, s);
3 token2=strtok(NULL, s);
4 ...
5
```

3 Code analysis

3.1 asm.c

思路为通过两遍扫描完成指令汇编成机器码。第一遍扫描生成label-address对应表，通过rewind()函数回流，第二遍扫描根据相应的label-address table进行扫描分析，然后划分区域进行解码分析：

- (1) char* readAndParse()函数将读入的汇编指令切分为相应的字符指针对应的字段，分别labelPtr, opcodePtr, arg0Ptr, arg1Ptr, arg2Ptr;
- (2) 第一步扫描会确定如果labelPtr非空，则将其映射到label-imm table中去，通过两个数组记录，Labels数组记录对应的label,Address数组记录address，一一对应；
- (3) 回流，通过得到的操作数和操作符进行解析。

3.1.1 distinguish code and data

代码通过char * readAndParse()函数，判断第一个字符是否是tab,如果是tab,则为普通指令，如果不是tab,则根据指南，其为labelptr 指向的字符串——label。

```
1 char *readAndParse(FILE *inFilePtr, char *lineString, char **labelPtr,  
2 char **opcodePtr, char **arg0Ptr, char **arg1Ptr,  
3 char **arg2Ptr)  
4 {
```

3.1.2 .fill distinguish label and number

特别有意思的是，注意到当操作数为.fill时，程序提供了相关的一些有意思的处理方式，对我们后面的程序编写有帮助,其实这里表明的是，当arg0是number的时候，那我们就默认存储的是数值，如果arg0是label 说明其中的内容还存储在另一片地址中；

```
1 else if (!strcmp(opcode, ".fill")) {  
2     if (!isNumber(arg0)) {  
3         num = get_label_address(arg0);  
4     }  
5     else {  
6         num = atoi(arg0);  
7     }  
8 }
```

3.2 riscv-small.c

simulator部分代码逻辑相对简单，通过mask获得一系列字段的位置，然后根据opcode进行相应的寄存器，存储器访问等操作，同时注意PC的转移即可。主要也是分为两步：

- (1) 根据opcode分别提取对应字段；
- (2) 根据opcode和相应的字段完成计算；

4 Code implementation

4.1 Assembler part

4.1.1 R-type instruction

此部分分析add, sub, sll, srl, or, and, 我们以srl 为例分析（考虑到逻辑右移的特殊性）。

```
1  else if(!strcmp(opcode, "srl")){
2      num = (SRL_FUNC7 << FUNCT7_SHIFT) | (atoi(arg2) << RS2_SHIFT) | (atoi(arg1)
      << RS1_SHIFT) | SRL_FUNC3<<FUNCT3_SHIFT | (atoi(arg0) << RD_SHIFT) |
      REG_REG_OP;
3  }
```

通过opcode 判断为 srl 指令，因此采用逻辑右移。注意相应的数据域分布,在此对应R型指令我们必须强调以下：在汇编指令中add rd rs1 rs2, 但是机器码中的分布: opcode rs2 rs1 func3 rd func7。这一点一定要注意。所以通过数据的左移我们就能得到concat 我们所需要的32位机器码，对应的左移位数也是通过宏定义好的。

4.1.2 I-type instruction(addi , lw)

addi指令相对简单，只是需要特殊处理立即数的12位问题而已；

```
1  else if(!strcmp(opcode, "addi")){
2      num = (atoi(arg2) << ADDI_IMM_SHIFT) | (atoi(arg1) << RS1_SHIFT) |
      ADDI_FUNC3<<FUNCT3_SHIFT | (atoi(arg0) << RD_SHIFT) | ADDI_OP;
3  }
```

lw指令则需要小心，这就是得注意前面提到的判断是否为label的问题；

```
1  else if(!strcmp(opcode, "lw")){
2      int tmp;
3      if (!isNumber(arg2)) {
4          tmp = get_label_address(arg2);
5      }
6      else {
7          tmp = atoi(arg2);
8      }
9
10     num = (tmp << LW_IMM_SHIFT) | (atoi(arg1) << RS1_SHIFT) |
      LW_FUNC3<<FUNCT3_SHIFT | (atoi(arg0) << RD_SHIFT) | LW_OP;
11 }
```

4.1.3 sw instruction

这条指令除开和之前lw指令同样需要注意label 和value的问题外，还要特别注意它的汇编指令对于寄存器位置的改变：

lw rd rs1 imm Reg[rd] <- Mem[Reg[rs1] + imm]

sw rs2 rs1 imm Mem[Reg[rs1] + imm] <- Reg[rs2]

同时通过左移和右移得到我所需要的高位[11:5]和低位[4:0]，高位由于已经移出去了，所以无需担心，但是低位特别小心右移时会出现意向不到的符号位扩展；所以我们最好引入一个mask；

```

1  #include<stdio.h>
2  int main() {
3      int a;
4      a = 10;
5      printf("%d", (a << 30) >> 30);
6  }
7  //output -2

```

所以我们只需要对低位进行一个mask就好;

```

1  #define SW_LOW_MASK      0xF80

```

```

1  else if(!strcmp(opcode, "sw")){
2      //转换成2进制后分别取出7位和5位即可
3      //注意 sw的 rs1,rs2的位置的改变
4      //arg0-----rs2(原来的rd)
5      //arg1-----rs1
6      //arg2-----imm(原来的rs2)
7      int tmp;
8      if (!isNumber(arg2)) {
9          tmp = get_label_address(arg2);
10     }
11     else {
12         tmp = atoi(arg2);
13     }
14     printf("%d",tmp);
15     num = ((tmp>>5)<<25)|((tmp<<27)>>20)&SW_LOW_MASK| atoi(arg1) <<
RS1_SHIFT | SW_FUNC3<<FUNCT3_SHIFT | (atoi(arg0) << RS2_SHIFT) | SW_OP;
16 }

```

4.1.4 beq instruction

beq这部分又是sw的加强版，注意以下三点：

- (1) 立即数的分布被改变，特别小心所有的右移操作；
- (2) beq汇编指令为 beq rs1 rs2 imm，顺序又被改变；
- (3) PC为相对地址，注意减去当前地址；

为了避免太多的细节问题，保证移位一步到位，这里我们直接采用四个mask。

```

1  #define BEQ_MASK1      0x00001000//取[12]
2  #define BEQ_MASK2      0x00000800//取[11]
3  #define BEQ_MASK3      0x000007e0//取[10:5]
4  #define BEQ_MASK4      0x0000001e//取[4:1]

```

```

1  else if (!strcmp(opcode, "beq")){
2      //arg0-----rs1(原来的rd)
3      //arg1-----rs2
4      //arg2-----imm(原来的rs2)
5      int tmpt;
6      int tmp[5];
7      if (!isNumber(arg2)) {
8          tmpt = get_label_address(arg2);
9          tmp[0]=tmpt-address;

```

```

10     }
11     else {
12         tmp = atoi(arg2);
13         tmp[0]=tmp;
14     }
15     tmp[1]=tmp[0]& BEQ_MASK1 <<19;
16     tmp[2]=tmp[0]&BEQ_MASK2 >>4;
17     tmp[3]=tmp[0]&BEQ_MASK3 <<20;
18     tmp[4]=(tmp[0]&BEQ_MASK4)<<7;
19     num=tmp[1]|tmp[2]|tmp[3]|tmp[4]|atoi(arg1) << RS2_SHIFT |
    BEQ_FUNC3<<FUNC3_SHIFT | (atoi(arg0) << RS1_SHIFT) | BEQ_OP;
20 }

```

4.2 Simulator part

4.2.1 decode

我们认为opcode, func3, func7是共有字段（尽管不一定合理）；

```

1 // public part
2 opcode = (state->mem[word_pc] ) & OP_MASK;
3 func3=(state->mem[word_pc]>>FUNC3_SHIFT ) & FUNC3_MASK;
4 func7=(state->mem[word_pc]>>FUNC7_SHIFT ) & FUNC7_MASK;

```

针对R型指令,解出rd, rs1, rs2;

```

1 if(opcode==REG_REG_OP){
2     rd=(state->mem[word_pc]>>RD_SHIFT ) &REG_MASK ;
3     rs1=(state->mem[word_pc]>>RS1_SHIFT ) &REG_MASK ;
4     rs2=(state->mem[word_pc]>>RS2_SHIFT ) &REG_MASK ;
5 }

```

针对l型指令 addi,得到rs1,rs2,immField, 千万小心immFiled不要再与ADDI_IMM_MASK相与, 否则会抹去其有符号数特征; 针对lw指令也一样, 在此就不赘述了。

```

1 else if(opcode==ADDI_OP){
2     rd=(state->mem[word_pc]>>RD_SHIFT ) &REG_MASK ;
3     rs1=(state->mem[word_pc]>>RS1_SHIFT ) &REG_MASK ;
4     immField=(state->mem[word_pc]>>ADDI_IMM_SHIFT );
5 }

```

针对sw指令,注意这时的移位仍然保持了有符号扩展;

```

1 else if(opcode==SW_OP){
2     rs1=(state->mem[word_pc]>>RS1_SHIFT ) &REG_MASK ;
3     rs2=(state->mem[word_pc]>>RS2_SHIFT ) &REG_MASK ;
4     rd=(state->mem[word_pc]>>RD_SHIFT ) &REG_MASK ;
5     immField=rd|(((signed int)(func7)<<25)>>20);
6 }

```

针对beq指令, 我们依然采用mask的方法保证正确性, 同时保证对最高位[12]拓展, 通过组合得到immField;

```

1  else if(opcode==BEQ_OP){
2      rs1=(state->mem[word_pc]>>RS1_SHIFT ) &REG_MASK ;
3      rs2=(state->mem[word_pc]>>RS2_SHIFT ) &REG_MASK ;
4      int tmp[5];
5      tmp[1]=((signed int)(state->mem[word_pc]&BEQ_MASK1))>>19;
6      tmp[2]=(state->mem[word_pc]&BEQ_MASK2)>>20;
7      tmp[3]=(state->mem[word_pc]&BEQ_MASK3)>>7;
8      tmp[4]=(state->mem[word_pc]&BEQ_MASK4)<<4;
9      immField=tmp[1]|tmp[2]|tmp[3]|tmp[4];

```

4.2.2 excution

相应的R型指令中srl指令具有特殊性, 注意是逻辑右移, 所以我们强制转成无符号数;

```

1  else if(func3==SRL_FUNC3){
2      state->reg[rd] = ((unsigned int)state->reg[rs1]) >>state->reg[rs2];
3  }

```

我们简单仿真一下这样的操作:

```

1  #include<stdio.h>
2  int main() {
3      int a;
4      a = 0xF0000000;
5      printf("%d", (unsigned int)a >> 30) ;
6  }
7  //output 3

```

lw和sw 就是对存储进行存取;

```

1  else if(opcode == LW_OP) {
2      state->reg[rd] = state->mem[immField>>2];
3  }
4  else if(opcode == SW_OP) {
5      state->mem[(immField+state->reg[rs1])>>2]=state->reg[rs2];
6  }

```

beq指令: 若两寄存器值相等, 则发生地址转移, 注意是相对地址转移, 通过将状态输出, 利用continue跳过下面的代码块, 进入对应的PC地址执行下一条指令;

```

1  else if(opcode == BEQ_OP) {
2      if(state->reg[rs1]==state->reg[rs2]){
3          state->pc=state->pc+immField;
4          printState(state);
5          state->reg[0] = 0;
6          continue;
7      }
8  }

```

5 Test bench outcome

5.1 simple.s

```

1      lw 21 0 op1 reg[21] <- op1
2      lw 22 0 op2 reg[22] <- op2
3      lw 23 0 op3 reg[23] <- op3
4      add 24 21 22 reg[24] <- reg[21] + reg[22]
5      sub 25 24 23 reg[25] <- reg[24] - reg[23]
6      sw 25 0 answer reg[25] -> answer
7  done      halt
8  op1 .fill 50
9  op2 .fill 30
10 op3 .fill 20
11 answer .fill 0

```

此条输出为如下,对比参考程序,一致:

<code>≡ simp.txt</code>	<code>×</code>	<code>...</code>	<code>≡ simple.out</code>	<code>×</code>
<code>≡ simp.txt</code>			<code>≡ simple.out</code>	
1 01c02a83			1 01c02a83	
2 02002b03			2 02002b03	
3 02402b83			3 02402b83	
4 016a8c33			4 016a8c33	
5 417c0cb3			5 417c0cb3	
6 03902423			6 03902423	
7 0000003f			7 0000003f	
8 00000032			8 00000032	
9 0000001e			9 0000001e	
10 00000014			10 00000014	
11 00000000			11 00000000	
12			12	

同时逐行对比memory状态和寄存器状态, simulator仿真结果也正常。

5.2 my_test.s

在自己的测试中观测以下几个方面:

- (1) 对于srl的逻辑右移, 是否对寄存器中负数右移后能变成正数;
- (2) sw,lw,beq对于label 和value的双重支持性;
- (3) 相关立即数为负数时候是否能表现符号扩展, 运行正确;
- (4) 注: 前面的运行表明对于lw 和 sw立即数模式已经正常, 那么主要来尝试value模式;

然而自己果然遇到了一个神奇的问题, 当无符号数与int类型相与后, 成为无符号类型, 此时右移就是逻辑右移, 并不符合我们在指令跳转和装载立即数时候的要求, 所以强制转换成为signed int 必须牢记。

```

1  #include<stdio.h>
2
3  int main(){
4      int i=-100;
5      printf("%d\n",((signed int)(0xffffffff&i))>>1);
6      printf("%d",((0xffffffff&i))>>1);
7  }
8  //output:
9  //-50
10 //2147483598

```

汇编程序如下:

```

1      lw 21 0 op1 reg[21] <- op1
2      lw 22 0 op2 reg[22] <- op2
3      lw 23 22 76 reg[23] <- op3
4      addi 21 21 -20
5      beq 22 21 -4 back to line4
6      add 25 21 22 reg[25] <- reg[21] + reg[22] --> -60
7      sub 24 22 21 reg[25] <- reg[22] - reg[21] --> 20
8      srl 25 25 24 reg[25] <- reg[25] >> 20 --> 0xfff=4095
9      sw 25 0 answer reg[25] -> answer
10     beq 22 23 done
11     sub 24 22 21
12     done halt
13     op1 .fill 0
14     op2 .fill -20
15     op3 .fill -20
16     answer .fill 0

```

- (1) 第3行考察lw Reg[rd] <- Mem[Reg[rs1] + imm] 其中对于rs1寄存器的处理;
- (2) 第4行考察addi 加负数的功能;
- (3) 第5行考察 beq的value模式, 同时也是看是否能保证imm 为负数不出问题;
- (4) 第8行考察 srl的逻辑移位功能, 我们保证25寄存器为负数, 观察最后的正负;
- (5) 第10行考察beq的 跳转到label功能;

```

1  machine halted
2  total of 13 instructions executed
3  state after cycle 12:
4      pc=48
5      memory:
6          mem[0] 0x3002a83 (50342531)
7          mem[1] 0x3402b03 (54536963)
8          mem[2] 0x4cb2b83 (80423811)
9          mem[3] 0xfeca8a93 (-20280685)
10         mem[4] 0xff5b0ee3 (-10809629)
11         mem[5] 0x16a8cb3 (23760051)
12         mem[6] 0x415b0c33 (1096485939)
13         mem[7] 0x18cdcb3 (26008755)
14         mem[8] 0x3902e23 (59780643)
15         mem[9] 0x17b0463 (24839267)
16         mem[10] 0x415b0c33 (1096485939)
17         mem[11] 0x3f (63)

```



```

18      mem[12] 0x0 (0)
19      mem[13] 0xffffffffec (-20)
20      mem[14] 0xffffffffec (-20)
21      mem[15] 0xfff (4095)
22      registers:
23      reg[0] 0x0 (0)
24      reg[1] 0x0 (0)
25      reg[2] 0x0 (0)
26      reg[3] 0x0 (0)
27      reg[4] 0x0 (0)
28      reg[5] 0x0 (0)
29      reg[6] 0x0 (0)
30      reg[7] 0x0 (0)
31      reg[8] 0x0 (0)
32      reg[9] 0x0 (0)
33      reg[10] 0x0 (0)
34      reg[11] 0x0 (0)
35      reg[12] 0x0 (0)
36      reg[13] 0x0 (0)
37      reg[14] 0x0 (0)
38      reg[15] 0x0 (0)
39      reg[16] 0x0 (0)
40      reg[17] 0x0 (0)
41      reg[18] 0x0 (0)
42      reg[19] 0x0 (0)
43      reg[20] 0x0 (0)
44      reg[21] 0xffffffffd8 (-40)
45      reg[22] 0xffffffffec (-20)
46      reg[23] 0xffffffffec (-20)
47      reg[24] 0x14 (20)
48      reg[25] 0xfff (4095)
49      reg[26] 0x0 (0)
50      reg[27] 0x0 (0)
51      reg[28] 0x0 (0)
52      reg[29] 0x0 (0)
53      reg[30] 0x0 (0)
54      reg[31] 0x0 (0)

```

在这里我们仅仅通过分析比对最后结果来看正确性：

- (1) lw通过相对地址找到op3，成功load;
- (2) 由于beq的存在，第4行指令会执行两遍，因此21寄存器变成-40;
- (3) 25寄存器在中间变成-60;
- (4) 24寄存器变成20;
- (5) 通过无符号右移后25寄存器从-60则变成0xfff 4095 满足要求;
- (6) sw指令实现存储;
- (7) beq通过label跳转，跳过一行sub指令，因此24寄存器仍未改变，保持20;
- (8) halt成功; 我们可以看到指令4, 5执行两次，指令11被跳过，因此一共执行12-1+2=13次，也正确;