

计算机体系结构-仿真实验1

2112495魏靖轩

[源码GitHub仓库](#)

实验目的

使用c语言编写一个对于MIPS指令集的仿真程序。

实验准备

实验环境配置有点难搞。。。ams2hex年久失修（课程ece447也好难找），随后选择使用MARS手动转换文件（需要JAVA环境）：

下载和官方wiki：[点我](#)，使用的教程：[点我](#)

实验指令的中文解析：[点我](#)

恳求老师能够下次修一下环境

因为实验指导书指出并不需要考虑运算中发生的溢出现象，因此程序没有相关的考虑与处理

实验依赖文件分析

首先我们要先看一下我们整个sim的流程是怎样的，根据指导书，我们首先来shell看一下go()和run()函数的运行：

我们发现：运行的本质就是调用一次 `cycle()` 函数。

那么我们看 `cycle()` 函数：

```

1 | void cycle() {
2 |     process_instruction();
3 |     CURRENT_STATE = NEXT_STATE;
4 |     INSTRUCTION_COUNT++;
5 | }

```

可以看到，它调用一次我们的 `process_instruction()` 函数来执行，然后实现状态的切换，然后指令计数加一。

那么我们的执行过程就很有思路了，就是依据命令，进行操作，被操作数是 `CURRENT_STATE`，随后将结果保存到 `NEXT_STATE` 中，然后返回即可。

现在问题来到了如何获取指令上面。我们翻一下 `shell.c` 文件，发现了如下的代码：

```

1 | /* Read in the program. */
2 |
3 |     ii = 0;
4 |     while (fscanf(prog, "%x\n", &word) != EOF) {
5 |         mem_write_32(MEM_TEXT_START + ii, word);
6 |         ii += 4;
7 |     }
8 |
9 |     CURRENT_STATE.PC = MEM_TEXT_START;

```

上面节选自 `load_program` 函数，至此我们明白了指令应该如何去获得，在初始化的时候我们将 `.x` 文件读入了我们设定的内存的 `text` 段，上面的代码是写入的部分，因此我们只需要使用 `mem_read_32()` 函数去读 `CURRENT_STATE.PC` 就可以获得需要执行的指令了。

实验设计

前期准备

首先我们要设计读指令的部分，很简单，思路上面已经给出。

```

1 | uint32_t mypc=mem_read_32(CURRENT_STATE.PC);

```

随后是另外一个问题，我们的指令是 `uint32_t` 类型的，并不能满足我们直接获得操作码、功能码等功能，因此我们希望能将这个数转为二进制，每一位都存在一个位置中，整体为一个32大小的 `int` 数组，因此我们给出转换代码如下：

```

1  int mpc[32];
2  char *pt = (char*)&mypc;
3  for(int i=0;i<32;i++)
4  {
5      mpc[31-i]=mypc&1;
6      mypc/=2;
7  }

```

这样我们就实现了指令转换为二进制，访问数组即可得到其各位的二进制值。

说明：数组的存储形式为从0~31，依次从高位开始存储（这也与我们常规学习中和参考手册中的从左至右的顺序相同，便于我们理解与操作），例如'3232299786'，十六进制为'0xC0A8FB0A'，二进制为'1100 0000 1010 1000.....'（'C0A8.....'），数组的0~15依次为1100 0000 1010 1000。

随后每次执行完一条指令，我们要执行数组清0的操作：

```

1  memset(mpc,0,sizeof(mpc));

```

然后我们需要对指令进行分类，以便我们设计其执行，使得我们的程序更加模块化。

依据上文中的中文解析，将指令分为如下的几类，总共有53条指令需要实现：

因为指令太多较杂，传统分类并不能涵盖所有指令，故在此按照其功能分类，并不按照R型、I型等分类进行，对应处理函数按照op码进行分块

同时我们还需要一个辅助函数，分别计算数组中第i位到第j位（ $i < j$ ）的二进制数值所对应的十进制值：

```

1  uint32_t getdec(int start,int end)
2  {
3      uint32_t res=0;
4      int str=1;
5      for(int i=end;i>=start;i--)
6      {
7          res+=mpc[i]*str;
8          str*=2;
9      }
10     return res;
11 }

```

逻辑运算指令

逻辑运算指令共8个，分别为AND、OR、XOR、NOR、ANDI、ORI、XORI、LUI。

AND、OR、XOR、NOR

四条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs) : *****
- 20-16 (rt) : *****
- 15-11 (rd) : *****
- 10-6: 00000
- 5-0 (func) : 100100为AND、100101为OR、100110为XOR、100111为NOR

代码如下：

```
1  switch (getdec(26,31))
2      {
3          case 36:
4              //AND
5              NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]&CURRENT_STATE
6              .REGS[rt];
7              break;
8          case 37:
9              //OR
10             NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]|CURRENT_STATE
11             .REGS[rt];
12             case 38:
13                 //XOR
14                 NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]^CURRENT_STATE
15                 .REGS[rt];
16                 case 39:
17                     //NOR
18                     NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]|CURRENT_STATE
19                     .REGS[rt];
20                     NEXT_STATE.REGS[rd]=~NEXT_STATE.REGS[rd];
```

ANDI、ORI、XORI

三条指令的按照op区分：001100为ANDI、001101为ORI、001110为XORI

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 0011**

- 25-21 (rs) : *****
- 20-16 (rt) : *****
- 15-0 (imm) : *****

代码如下:

```

1  void op12()
2  {
3      //ANDI
4      int rs=getdec(6,10);
5      int rt=getdec(11,15);
6      int imm=getdec(16,31);
7      uint32_t high=CURRENT_STATE.REGS[rs]&0xFFFF0000;
8      uint32_t low=CURRENT_STATE.REGS[rs]&0xFFFF;
9      uint32_t ans=low&imm;
10     NEXT_STATE.REGS[rt]=high+ans;
11 }
12
13 void op13()
14 {
15     //ORI
16     int rs=getdec(6,10);
17     int rt=getdec(11,15);
18     int imm=getdec(16,31);
19     uint32_t high=CURRENT_STATE.REGS[rs]&0xFFFF0000;
20     uint32_t low=CURRENT_STATE.REGS[rs]&0xFFFF;
21     uint32_t ans=low|imm;
22     NEXT_STATE.REGS[rt]=high+ans;
23 }
24
25 void op14()
26 {
27     //XORI
28     int rs=getdec(6,10);
29     int rt=getdec(11,15);
30     int imm=getdec(16,31);
31     uint32_t high=CURRENT_STATE.REGS[rs]&0xFFFF0000;
32     uint32_t low=CURRENT_STATE.REGS[rs]&0xFFFF;
33     uint32_t ans=low^imm;
34     NEXT_STATE.REGS[rt]=high+ans;
35 }

```

LUI

指令作用为: $rt \leftarrow \text{immediate} \ll 0$, 将指令中的16bit立即数保存到地址为rt的通用寄存器的高16位。另外, 地址为rt的通用寄存器的低16位使用0填充。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 001111
- 25-21: 00000
- 20-16 (rt) : *****
- 15-0 (imm) : *****

代码如下：

```
1 void op15()  
2 {  
3     //LUI  
4     int rt=getdec(11,15);  
5     uint32_t imm=getdec(16,31);  
6     imm=imm<<16;  
7     NEXT_STATE.REGS[rt]=imm;  
8 }
```

移位指令

移位指令共6个，分别为SLL、SRL、SRA、SLLV、SRLV、SRAV。

SLL、SRL、SRA、SLLV、SRLV、SRAV

六条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs/00000) : *****/00000
- 20-16 (rt) : *****
- 15-11 (rd) : *****
- 10-6 (sa/00000) : *****/00000
- 5-0 (func) : 000000为SLL、000010为SRL、000011为SRA、000100为SLLV、000110为SRLV、000111为SRAV

代码如下：

```
1 switch(getdec(26,31))  
2 case 0:  
3     //SLL  
4     NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]<<sa;  
5     break;
```

```

6   case 2:
7       //SRL
8       NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]>>sa;
9       break;
10  case 3:
11      //SRA
12      uint32_t hi=CURRENT_STATE.REGS[rt]&0x80000000;
13      if(hi==1)
14          NEXT_STATE.REGS[rd]=(0xFFFFFFFF<<(32-sa))+(CURRENT_STATE.REG
S[rt]>>sa);
15      else
16          NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]>>sa;
17      break;
18  case 4:
19      //SLLV
20      uint32_t saa=CURRENT_STATE.REGS[rs]&0x1F;
21      NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]<<saa;
22      break;
23  case 6:
24      //SRLV
25      uint32_t saa=CURRENT_STATE.REGS[rs]&0x1F;
26      NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]>>saa;
27      break;
28  case 7:
29      //SRAV
30      uint32_t saa=CURRENT_STATE.REGS[rs]&0x1F;
31      uint32_t hi=CURRENT_STATE.REGS[rt]&0x80000000;
32      if(hi==1)
33          NEXT_STATE.REGS[rd]=(0xFFFFFFFF<<(32-saa))+(CURRENT_STATE.RE
GS[rt]>>saa);
34      else
35          NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rt]>>saa;
36      break;

```

移动操作指令

移动操作指令共4个，分别为MFHI、MFLO、MTHI、MTLO。

MFHI、MFLO、MTHI、MTLO

四条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs/00000) : *****/00000

- 20-16: 00000
- 15-11 (rd/00000) : ***/00000
- 10-6 (00000) : 00000
- 5-0 (func) : 010000为MFHI、010001为MTHI、010010为MFLO、010011为MTLO

代码如下:

```

1  case 16:
2      //MFHI
3      NEXT_STATE.REGS[rd]=CURRENT_STATE.HI;
4      break;
5  case 17:
6      //MTHI
7      NEXT_STATE.HI=CURRENT_STATE.REGS[rs];
8      break;
9  case 18:
10     //MFLO
11     NEXT_STATE.REGS[rd]=CURRENT_STATE.LO;
12     break;
13 case 19:
14     //MTLO
15     NEXT_STATE.LO=CURRENT_STATE.REGS[rs];
16     break;

```

算术操作指令

算术操作指令共14个，分别为ADD、ADDU、SUB、SUBU、SLT、SLTU、ADDI、ADDIU、SLTI、SLTIU、MULT、MULTU、DIV、DIVU

ADD、ADDU、SUB、SUBU、SLT、SLTU

六条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs) : ***/
- 20-16 (rt) : ***/
- 15-11 (rd) : ***/
- 10-6 (00000) : 00000
- 5-0 (func) : 100000为ADD、100001为ADDU、100010为SUB、100011为SUBU、101010为SLT、101011为SLTU

根据实验指导书，在此不考虑处理数据溢出后的异常（因为实验未给出相关接口，因此不作抛出异常的处理），仅遵守溢出后不修改寄存器等规定

代码如下：

```
1  case 32:
2      //ADD
3      uint32_t ans=CURRENT_STATE.REGS[rs]+CURRENT_STATE.REGS[rt];
4      if(!(ans<CURRENT_STATE.REGS[rs]||ans<CURRENT_STATE.REGS[rt]))
5  )
6      NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]+CURRENT_STATE.REG
7  S[rt];
8      break;
9  case 33:
10     //ADDU
11     NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]+CURRENT_STATE.REG
12     S[rt];
13     break;
14 case 34:
15     //SUB
16     if(CURRENT_STATE.REGS[rs]>=CURRENT_STATE.REGS[rt])
17     NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]-CURRENT_STATE.REG
18     S[rt];
19     break;
20 case 35:
21     //SUBU
22     NEXT_STATE.REGS[rd]=CURRENT_STATE.REGS[rs]-CURRENT_STATE.REG
23     S[rt];
24     break;
25 case 42:
26     //SLT
27     int rrs=CURRENT_STATE.REGS[rs];
28     int rrt=CURRENT_STATE.REGS[rt];
29     if(rrs<rrt)
30     NEXT_STATE.REGS[rd]=(uint32_t)1;
31     else
32     NEXT_STATE.REGS[rd]=(uint32_t)0;
33     break;
34 case 43:
35     //SLTU
36     if(CURRENT_STATE.REGS[rs]<CURRENT_STATE.REGS[rt])
37     NEXT_STATE.REGS[rd]=(uint32_t)1;
38     else
39     NEXT_STATE.REGS[rd]=(uint32_t)0;
40     break;
```

ADDI、ADDIU、SLTI、SLTIU

四条指令的按照op区分：001000为ADDI、001001为ADDIU、001010为SLTI、001011为SLTIU

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : *****
- 25-21 (rs) : *****
- 20-16 (rt) : *****
- 15-0 (imm) : *****

代码如下：

```
1  void op8()
2  {
3      //ADDI
4      int rs=getdec(6,10);
5      int rt=getdec(11,15);
6      uint32_t imm=getdec(16,31);
7      if(mpc[16]==1)
8          imm+=0xFFFF0000;
9      uint32_t ans=CURRENT_STATE.REGS[rs]+imm;
10     if(!(ans<CURRENT_STATE.REGS[rs]||ans<imm))
11         NEXT_STATE.REGS[rt]=CURRENT_STATE.REGS[rs]+imm;
12 }
13
14 void op9()
15 {
16     //ADDIU
17     int rs=getdec(6,10);
18     int rt=getdec(11,15);
19     uint32_t imm=getdec(16,31);
20     if(mpc[16]==1)
21         imm+=0xFFFF0000;
22     NEXT_STATE.REGS[rt]=CURRENT_STATE.REGS[rs]+imm;
23 }
24
25 void op10()
26 {
27     //SLTI
28     int rs=getdec(6,10);
29     int rt=getdec(11,15);
30     uint32_t imm=getdec(16,31);
31     if(mpc[16]==1)
32         imm+=0xFFFF0000;
33     int rrs=CURRENT_STATE.REGS[rs];
```

```

34     int rimm=imm;
35     if(rrs<rimm)
36     NEXT_STATE.REGS[rt]=(uint32_t)1;
37     else
38     NEXT_STATE.REGS[rt]=(uint32_t)0;
39 }
40
41 void op11()
42 {
43     //SLTIU
44     int rs=getdec(6,10);
45     int rt=getdec(11,15);
46     uint32_t imm=getdec(16,31);
47     if(mpc[16]==1)
48     imm+=0xFFFF0000;
49     if(CURRENT_STATE.REGS[rs]<imm)
50     NEXT_STATE.REGS[rt]=(uint32_t)1;
51     else
52     NEXT_STATE.REGS[rt]=(uint32_t)0;
53 }

```

MULT、MULTU

两条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs) : *****
- 20-16 (rt) : *****
- 15-11 (00000) : 00000
- 10-6 (00000) : 00000
- 5-0 (func) : 011000为MULT、011001为MULTU

代码如下：

```

1  case 24:
2      //MULT
3      int rrs=CURRENT_STATE.REGS[rs];
4      int rrt=CURRENT_STATE.REGS[rt];
5      long long ans=rrs*rrt;
6      uint64_t uans=ans;
7      uint32_t anshigh=(uans&0xFFFFFFFF00000000)>>32;
8      uint64_t myt=0xFFFFFFFF;
9      uint32_t anslow=uans&myt;

```

```

10     NEXT_STATE.HI=anshigh;
11     NEXT_STATE.LO=anslow;
12     break;
13 case 25:
14     //MULTU
15     uint64_t ans=CURRENT_STATE.REGS[rs]*CURRENT_STATE.REGS[rt];
16     uint32_t anshigh=(ans&0xFFFFFFFF00000000)>>32;
17     uint64_t myt=0xFFFFFFFF;
18     uint32_t anslow=ans&myt;
19     NEXT_STATE.HI=anshigh;
20     NEXT_STATE.LO=anslow;
21     break;

```

DIV、DIVU

两条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs) : *****
- 20-16 (rt) : *****
- 15-11 (00000) : 00000
- 10-6 (00000) : 00000
- 5-0 (func) : 011010为DIV、011011为DIVU

代码如下：

```

1 case 26:
2     //DIV
3     int rrs=CURRENT_STATE.REGS[rs];
4     int rrt=CURRENT_STATE.REGS[rt];
5     int ans=rrs/rrt;
6     int aans=rrs%rrt;
7     uint32_t uans=ans;
8     uint32_t uaans=aans;
9     NEXT_STATE.HI=uaans;
10    NEXT_STATE.LO=uans;
11    break;
12 case 27:
13     //DIVU
14     uint32_t uans=CURRENT_STATE.REGS[rs]/CURRENT_STATE.REGS[rt];
15     uint32_t uaans=CURRENT_STATE.REGS[rs]%CURRENT_STATE.REGS[rt]
16 ;
17     NEXT_STATE.HI=uaans;

```

```
18 | NEXT_STATE.LO=uans;  
    | break;
```

跳转指令

跳转指令共4个，分别为JR、JALR、J、JAL

JR、JALR

JR、JALR两条指令的op一致，不同的为func。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-21 (rs) : *****
- 20-16 (00000) : 00000
- 15-11 (rd/00000) : *****/00000
- 10-6 (00000) : 00000
- 5-0 (func) : 001000为JR、001001为JALR

代码如下：

```
1 | case 8:  
2 |     //JR  
3 |     NEXT_STATE.PC=CURRENT_STATE.REGS[rs];  
4 |     break;  
5 | case 9:  
6 |     //JALR  
7 |     NEXT_STATE.PC=CURRENT_STATE.REGS[rs];  
8 |     NEXT_STATE.REGS[rd]=CURRENT_STATE.PC+4;  
9 |     break;
```

J、JAL

J、JAL两条指令以op区分：000010为J、000011为JAL

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : *****
- 25-0 (rs) : *****

代码如下：

```

1  void op2()
2  {
3      //J
4      uint32_t instr=getdec(6,31);
5      instr=instr<<2;
6      uint32_t npc=CURRENT_STATE.PC+4;
7      uint32_t unpc=npc&0xF0000000;
8      uint32_t instr_index=instr+unpc;
9      NEXT_STATE.PC=instr_index;
10 }
11
12 void op3()
13 {
14     //JAL
15     uint32_t instr=getdec(6,31);
16     instr=instr<<2;
17     uint32_t npc=CURRENT_STATE.PC+4;
18     uint32_t unpc=npc&0xF0000000;
19     uint32_t instr_index=instr+unpc;
20     NEXT_STATE.PC=instr_index;
21     NEXT_STATE.REGS[31]=CURRENT_STATE.PC+4;
22 }

```

分支指令

分支指令共有8个，分别为BEQ、BNE、BLEZ、BGTZ、BLTZ、BGEZ、BLTZAL、BGEZAL

BEQ、BNE、BLEZ、BGTZ

四条指令以op区分：000100为BEQ、000101为BNE、000110为BLEZ、000111为BGTZ

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : *****
- 25-21 (rs) : *****
- 20-16 (rt/00000) : *****/00000
- 15-0 (offset) : *****

代码如下：

```

1  void op4()
2  {
3      //BEQ
4      int rs=getdec(6,10);
5      int rt=getdec(11,15);

```

```

6      uint32_t off=getdec(16,31);
7      off=off<<2;
8      if(mpc[16]==1)
9      off+=0xFFFC0000;
10     if(CURRENT_STATE.REGS[rs]==CURRENT_STATE.REGS[rt])
11     NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
12 }
13
14 void op5()
15 {
16     //BNE
17     int rs=getdec(6,10);
18     int rt=getdec(11,15);
19     uint32_t off=getdec(16,31);
20     off=off<<2;
21     if(mpc[16]==1)
22     off+=0xFFFC0000;
23     if(CURRENT_STATE.REGS[rs]!=CURRENT_STATE.REGS[rt])
24     NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
25 }
26
27 void op6()
28 {
29     //BLEZ
30     int rs=getdec(6,10);
31     uint32_t off=getdec(16,31);
32     off=off<<2;
33     if(mpc[16]==1)
34     off+=0xFFFC0000;
35     int rrs=CURRENT_STATE.REGS[rs];
36     if(CURRENT_STATE.REGS[rs]==0x0 || rrs<0)
37     NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
38 }
39
40 void op7()
41 {
42     //BGTZ
43     int rs=getdec(6,10);
44     uint32_t off=getdec(16,31);
45     off=off<<2;
46     if(mpc[16]==1)
47     off+=0xFFFC0000;
48     int rrs=CURRENT_STATE.REGS[rs];
49     if(rrs>0)
50     NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
51 }

```

BLTZ、BGEZ、BLTZAL、BGEZAL

四条指令的op一致，不同的为rt处的值。

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000001
- 25-21 (rs) : *****
- 20-16: 00000为BLTZ、00001为BGEZ、10000为BLTZAL、10001为BGEZAL
- 15-0 (offset) : *****

代码如下：

```
1  void op1()
2  {
3      int rs=getdec(6,10);
4      uint32_t off=getdec(16,31);
5      off=off<<2;
6      if(mpc[16]==1)
7          off+=0xFFFC0000;
8      switch (getdec(11,15))
9      {
10         case 0:
11             //BLTZ
12             int rrs=CURRENT_STATE.REGS[rs];
13             if(rrs<0)
14                 NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
15             break;
16         case 1:
17             //BGEZ
18             int rrs=CURRENT_STATE.REGS[rs];
19             if(rrs>=0)
20                 NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
21             break;
22         case 16:
23             //BLTZAL
24             int rrs=CURRENT_STATE.REGS[rs];
25             if(rrs<0)
26             {
27                 NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
28                 NEXT_STATE.REGS[31]=CURRENT_STATE.PC+4;
29             }
30             break;
31         case 17:
32             //BGEZAL
33             int rrs=CURRENT_STATE.REGS[rs];
```



```

34         if(rrs>=0)
35         {
36             NEXT_STATE.PC=CURRENT_STATE.PC+off+4;
37             NEXT_STATE.REGS[31]=CURRENT_STATE.PC+4;
38         }
39         break;
40     default:
41         break;
42     }
43 }

```

加载指令

加载指令共5个，分别为LB、LBU、LH、LHU、LW。

LB、LH、LW、LBU、LHU

五条指令以op区分：100000为LB、100001为LH、100011为LW、100100为LBU、100101为LHU

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : *****
- 25-21 (base) : *****
- 20-16 (rt) : *****
- 15-0 (offset) : *****

代码如下：

```

1  void op32()
2  {
3      //LB
4      int base=getdec(6,10);
5      int rt=getdec(11,15);
6      uint32_t off=getdec(16,31);
7      if(mpc[16]==1)
8          off+=0xFFFF0000;
9      uint32_t add=CURRENT_STATE.REGS[base]+off;
10     uint32_t mem=mem_read_32(add);
11     uint32_t data=mem&0xFF;
12     if(data>=128)
13         data+=0xFFFF00;
14     NEXT_STATE.REGS[rt]=data;
15 }
16

```

```

17 void op33()
18 {
19     //LH
20     int base=getdec(6,10);
21     int rt=getdec(11,15);
22     uint32_t off=getdec(16,31);
23     if(mpc[16]==1)
24         off+=0xFFFF0000;
25     uint32_t add=CURRENT_STATE.REGS[base]+off;
26     uint32_t mem=mem_read_32(add);
27     uint32_t data=mem&0xFFFF;
28     if(data>=0x8000)
29         data+=0xFFFF0000;
30     NEXT_STATE.REGS[rt]=data;
31 }
32
33 void op35()
34 {
35     //LW
36     int base=getdec(6,10);
37     int rt=getdec(11,15);
38     uint32_t off=getdec(16,31);
39     if(mpc[16]==1)
40         off+=0xFFFF0000;
41     NEXT_STATE.REGS[rt]=mem_read_32(CURRENT_STATE.REGS[base]+off
42 );
43 }
44
45 void op36()
46 {
47     //LBU
48     int base=getdec(6,10);
49     int rt=getdec(11,15);
50     uint32_t off=getdec(16,31);
51     if(mpc[16]==1)
52         off+=0xFFFF0000;
53     uint32_t add=CURRENT_STATE.REGS[base]+off;
54     uint32_t mem=mem_read_32(add);
55     uint32_t data=mem&0xFF;
56     NEXT_STATE.REGS[rt]=data;
57 }
58
59 void op37()
60 {
61     //LHU
62     int base=getdec(6,10);
63     int rt=getdec(11,15);
64     uint32_t off=getdec(16,31);

```

```

65     if(mpc[16]==1)
66         off+=0xFFFF0000;
67     uint32_t add=CURRENT_STATE.REGS[base]+off;
68     uint32_t mem=mem_read_32(add);
69     uint32_t data=mem&0xFFFF;
70     NEXT_STATE.REGS[rt]=data;
    }

```

存储指令

存储指令共3个，分别为SB、SH、SW。

SB、SH、SW

三条指令以op区分：101000为SB、101001为SH、101011为SW

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : *****
- 25-21 (base) : *****
- 20-16 (rt) : *****
- 15-0 (offset) : *****

代码如下：

```

1  void op40()
2  {
3      //SB
4      int base=getdec(6,10);
5      int rt=getdec(11,15);
6      uint32_t off=getdec(16,31);
7      if(mpc[16]==1)
8          off+=0xFFFF0000;
9      uint32_t add=CURRENT_STATE.REGS[base]+off;
10     uint32_t data=mem_read_32(add);
11     data=data&0xFFFFFFF0;
12     uint32_t mydata=CURRENT_STATE.REGS[rt];
13     mydata=mydata&0xFF;
14     mydata+=data;
15     mem_write_32(add,mydata);
16 }
17
18 void op41()
19 {
20     //SH

```

```

21     int base=getdec(6,10);
22     int rt=getdec(11,15);
23     uint32_t off=getdec(16,31);
24     if(mpc[16]==1)
25         off+=0xFFFF0000;
26     uint32_t add=CURRENT_STATE.REGS[base]+off;
27     uint32_t data=mem_read_32(add);
28     data=data&0xFFFF0000;
29     uint32_t mydata=CURRENT_STATE.REGS[rt];
30     mydata=mydata&0xFFFF;
31     mydata+=data;
32     mem_write_32(add,mydata);
33 }
34
35 void op43()
36 {
37     //SW
38     int base=getdec(6,10);
39     int rt=getdec(11,15);
40     uint32_t off=getdec(16,31);
41     if(mpc[16]==1)
42         off+=0xFFFF0000;
43     uint32_t add=CURRENT_STATE.REGS[base]+off;
44     mem_write_32(add,CURRENT_STATE.REGS[rt]);
45 }

```

特殊指令

特殊指令为SYSCALL

SYSCALL

其各位的分布如下（此处31指最高位，对应数组的最低下标0）：

- 31-26 (op) : 000000
- 25-6 (code) : *****
- 5-0 (func) : 001100

代码如下：

```

1     case 12:
2         //SYSCALL
3         if(CURRENT_STATE.REGS[2]==10)RUN_BIT=0;
4         break;

```

实验验证

.s转换为.x文件

在前期准备中已经说明，不再赘述。

编译

进入目录：

```
1 | cd src/
```

编译make：

```
1 | make
```

随后在根目录下执行：

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

验证

addiu

执行整个程序，随后保存最后的结果：

```
1 | $ src/sim inputs/addiu.x
2 | $ go
3 | $ rdump
```

SIM程序执行结果：

```
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
```

我们使用MARS验证，使用MARS执行汇编程序（.s文件），结果如下：

Registers	Coproc 1	Coproc 0	
Name	Number	Value	
\$zero	0	0x00000000	
\$at	1	0x00000000	
\$v0	2	0x0000000a	
\$v1	3	0x00000000	
\$a0	4	0x00000000	
\$a1	5	0x00000000	
\$a2	6	0x00000000	
\$a3	7	0x00000000	
\$t0	8	0x00000005	
\$t1	9	0x00000131	
\$t2	10	0x000001f4	
\$t3	11	0x00000243	
\$t4	12	0x00000000	
\$t5	13	0x00000000	
\$t6	14	0x00000000	
\$t7	15	0x00000000	
\$s0	16	0x00000000	
\$s1	17	0x00000000	
\$s2	18	0x00000000	
\$s3	19	0x00000000	
\$s4	20	0x00000000	
\$s5	21	0x00000000	
\$s6	22	0x00000000	
\$s7	23	0x00000000	
\$t8	24	0x00000000	
\$t9	25	0x00000000	
\$k0	26	0x00000000	
\$k1	27	0x00000000	
\$gp	28	0x10008000	
\$sp	29	0x7fffffc	
\$fp	30	0x00000000	
\$ra	31	0x00000000	
pc		0x0040001c	
hi		0x00000000	
lo		0x00000000	

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

arithtest

执行整个程序，随后保存最后的结果：

```

1 | $ src/sim inputs/arithtest.x
2 | $ go
3 | $ rdump

```

SIM程序执行结果：

```
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: 0x00000000
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
```

我们使用MARS验证，使用MARS执行汇编程序（.s文件），结果如下：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x0000000a
\$v1	3	0x00000800
\$a0	4	0x00000c00
\$a1	5	0x000004d2
\$a2	6	0x04d20000
\$a3	7	0x04d2270f
\$t0	8	0x04d2230f
\$t1	9	0x00000400
\$t2	10	0x000004ff
\$t3	11	0x00269000
\$t4	12	0x004d2000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0xfffffb01
\$s0	16	0x00000000
\$s1	17	0x00640000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400044
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

brtest0

执行整个程序，随后保存最后的结果：

```
1 | $ src/sim inputs/brtest0.x
2 | $ go
3 | $ rdump
```

SIM程序执行结果：

```
Current register/bus values :
-----
Instruction Count : 12
PC                : 0x0040005c
Registers:
R0: 0x00000000
R1: 0x0000d00d
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
```

我们使用MARS验证，使用MARS执行汇编程序（.s文件），结果如下：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x0000d00d
\$v0	2	0x0000000a
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000001
\$a2	6	0x00001337
\$a3	7	0x0000d00d
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040005c
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

有一点需要说明的是，在原本的程序中，1号寄存器不应该被赋值，但在此却出现了不是0的值，其原因在于对于命令的解析：我们所假定的是直接执行命令；而MARS在对可能溢出的addiu执行时，有如下的解释方式：

lui \$1, 0x00000000	31:	addiu \$7, \$zero, 0xd00d
ori \$1, \$1, 0x0000d00d		
addu \$7, \$0, \$1		
syscall	32:	syscall

可以看到，addiu被转换为了左边的3个指令来执行，因此1号寄存器作为中间值而被更改，但是只看开始和结果，7号寄存器的值是没有问题的（同时在将.s文件转换为.x文件时转换出的命令也会变成3条，不过lui、ori、addu等命令我们也进行了实现，这还能验证我们这三条命令的正确性）。

brtest1

执行整个程序，随后保存最后的结果：

```
1 | $ src/sim inputs/brtest1.x
2 | $ go
3 | $ rdump
```

SIM程序执行结果：

```
Current register/bus values :
-----
Instruction Count : 30
PC                  : 0x00400090
Registers:
R0: 0x00000000
R1: 0xbeb0063d
R2: 0x0000000a
R3: 0x00000001
R4: 0xffffffff
R5: 0xbef01a5e
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
```

随后使用MARS执行汇编程序，结果如下：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0xbab0063d
\$v0	2	0x0000000a
\$v1	3	0x00000001
\$a0	4	0xffffffff
\$a1	5	0xbef01a5e
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00400070
pc		0x00400090
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

brtest2

执行整个程序，随后保存最后的结果：

```
1 | $ src/sim inputs/brtest2.x
2 | $ go
3 | $ rdump
```

SIM程序执行结果：

```
Current register/bus values :
-----
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
```

随后使用MARS执行汇编程序，结果如下：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x0000d00d
\$v0	2	0x0000000a
\$v1	3	0x00000000
\$a0	4	0x00000000
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x0000d00d
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000000
\$t3	11	0x00000000
\$t4	12	0x00000000
\$t5	13	0x00000000
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040002c
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

memtest0

执行整个程序，随后保存最后的结果：

```

1 | $ src/sim inputs/memtest0.x
2 | $ go
3 | $ rdump

```

SIM程序执行结果：


```
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
```

随后使用MARS执行汇编程序，结果如下：

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x0000000a
\$v1	3	0x10000004
\$a0	4	0x00000000
\$a1	5	0x000000ff
\$a2	6	0x000001fe
\$a3	7	0x000003fc
\$t0	8	0x0000792c
\$t1	9	0x000000ff
\$t2	10	0x000001fe
\$t3	11	0x000003fc
\$t4	12	0x0000792c
\$t5	13	0x000000ff
\$t6	14	0x000000ff
\$t7	15	0x000001fe
\$s0	16	0x000003fc
\$s1	17	0x0000881d
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400080
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同，证明我们仿真平台的正确性。

memtest1

执行整个程序，随后保存最后的结果：

```
1 | $ src/sim inputs/memtest1.x
2 | $ go
3 | $ rdump
```

SIM程序执行结果：

```
Current register/bus values :
-----
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: 0x0001cb90
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
```

随后使用MARS执行汇编程序，结果如下：

\$zero	0	0x00000000
\$at	1	0x0000efbe
\$v0	2	0x0000000a
\$v1	3	0x10000004
\$a0	4	0x00000000
\$a1	5	0x0000cafe
\$a2	6	0x0000feca
\$a3	7	0x0000beef
\$t0	8	0x0000efbe
\$t1	9	0x000000fe
\$t2	10	0x000000ca
\$t3	11	0xffffffff
\$t4	12	0xffffffff
\$t5	13	0x0000cafe
\$t6	14	0x0000feca
\$t7	15	0xffffbeef
\$s0	16	0xffffefbe
\$s1	17	0x000179ea
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10008000
\$sp	29	0x7fffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x004000a0
hi		0x00000000
lo		0x00000000

对比结果，可以验证各个寄存器的值全部相同（除了17号），证明我们仿真平台的正确性。

在此说明：17号寄存器为9-16号寄存器的值全部加起来的结果，因为实验指导书中明确说明不需要处理溢出，因此其结果与MARS有所不同

结论与结果

经过验证，在不考虑一些条件（根据实验指导书）时，完成了相关指令的sim仿真，通过了所有的test测试。

额外的工作

在上面的验证中，有两个寄存器（gp、sp）的值值得注意，但我们的软件仅仅对.text段的代码进行仿真，对于这两个寄存器的值的确定，并不在我们的程序的功能内，因此不作相关的处理。