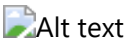
MIPS指令集模拟器设计思路

2113301 朱霞洋

本次实验用C语言完成了一个包含53条指令的MIPS指令模拟器，该模拟器可以读取32位MIPS指令对应的机器码，并执行相应的操作，大致的设计思路和流程如下：

针对每一条指令的标识性字段进行宏定义

 Alt text

对于R型指令，其op字段都是000000，需要进一步通过funct字段对每一条指令唯一识别，因此要对每一条R型指令的funct字段进行宏定义:

```
#define op_R 0x00  //R type instruction

#define funct_add 0x20 // add 10000
#define funct_addu 0x21 //addu 100001
#define funct_sub 0x22 //sub 100010
...
```

对于I型指令和J型指令，其op字段各不相同，因此可以对这两种类型的指令的op字段进行宏定义，用于对其的识别:

```
#define op_addi 0x08 //addi 001000
#define op_addiu 0x09 //addiu 001001
#define op_slti 0x0a  //slti 001010
...
#define op_j 0x02 //J 000010
#define op_jal 0x03 //jal 000011
```

此外，此次实现的指令中 BGEZ,BGEZAL,BLTZ,BLTZAL 这四条指令较为特殊，其指令的高六位（即op字段所在位置）都是000001，需要进一步通过指令的16-20位（即rt字段所在位置）进行识别，因此对这四条指令的rt字段进行了宏定义：

```
#define op_Branch 0x01 // REGIMM 000001
#define BGEZ 0x01 //BGEZ 00001
#define BGEZAL 0x11 // BGEZAL 10001
#define BLTZ 0x00 //bltz 00000
#define BLTZAL 0x10 //BLTZAL 10000
```

指令的读取和译码

在宏定义部分，将此次要实现的指令大致分为**R型**（op=000000，通过funct唯一识别），**I和J型**（通过op唯一识别），以及**四条特殊跳转指令**（op=000001，通过rt字段唯一识别），因此在指令的译码和执行时，也按照这样的分类进行。

首先通过`mem_read_32(CURRENT_STATE.PC)`可以实现在PC记录的地址处获取指令，然后通过一系列位运算，获取各个字段的值，并且注意此处对于I型的imm立即数字段进行了0拓展和符号拓展，并统一保存为`uint32_t`类型，在必要的时候将符号拓展后的立即数转换为有符号的`int32_t`类型：

```
    uint32_t ins = mem_read_32(CURRENT_STATE.PC);  //read the instruction to be
processed via PC register


    uint32_t op = (ins >> 26) & 0x3F; // ins[31:26]
    uint32_t rs = (ins >> 21) & 0x1F; // ins[25:21]
    uint32_t rt = (ins >> 16) & 0x1F; //ins [20:16]
    uint32_t rd = (ins >> 11) & 0x1F; //ins [15:11]
    uint32_t shamt = (ins >> 6) & 0x1F; //ins[10:6]
    uint32_t funct = ins  & 0x3F; //ins[5:0]
    uint32_t uimm = ins & 0xFFFF; //ins[15:0]   //zero-extended
    uint32_t imm = (uimm & 0x8000) ? (uimm | 0xffff0000) : uimm;  //sign-extended
    uint32_t target = ins & 0x3ffffff;  // ins[25:0]
```

对于上文大致分出的三种类型的指令，分别编写了`excute_R()`,`excute_I_and_J()`,`excute_branch()`函数，对op字段进行初步判断后，即可转到对应的指令的处理函数中，这样一来对于特殊的分支跳转指令可以更快的进行判断和执行：

```
if(op == op_R){ //是R型的指令
           excute_R(rs,rt,rd,shamt,funct);
   }
   else if (op == op_Branch){    //是四条特殊跳转指令之一
           excute_branch(rs,rt,rd,imm,uimm);
   }
   else{  //其他情况，I和J型
       excute_I_and_J(op,rs,rt,rd,imm,uimm,target);
   }
```

指令的执行

在每一个excute函数中，都会先进行PC寄存器的自增4，对其进行更新，这样做的目的是，一些跳转指令会二次改变PC的值，需要放在PC+4之后，否则其作用会被PC寄存器的自增所覆盖。

紧接着，通过switch语句对每一条指令进行唯一识别，分别编写执行对应操作的代码，并将结果更新到NEXT_STATE中，因此执行部分的框架大致都如下：

```
    NEXT_STATE.PC = CURRENT_STATE.PC + 4;

    switch (funct) //or op(I type and J type),or rt(BGEZ,BGEZAL,BLTZ,BLTZAL)
```

```
    {
    case 1: //每个case是一条指令
            ...  //指令的详细实现

    case 2:
            ...

    case 3:
            ...
    ...
    }
```

由于每条指令执行的具体代码都不相同，在此处不做赘述，详细代码可见sim.c。