Y86模拟器实验报告

陈洋 19307130211 张开元 19307130227

目录

一、设计思路

我们的Y86模拟器姑且命名为CZ_CPU, 它的总体框架参考了CSAPP上第四章的内容，通过将指令的执行分为5个不同的阶段，使用不同的流水线寄存器储存各个阶段的运行结果。然后完善每个阶段的执行逻辑以及对流水线大冒险的处理，完成流水线CPU的模拟。

CZ_CPU的核心是易用, 清晰, 高鲁棒性. 在加载测试文件之后, 查看, 运行, 断点调试都十分简单, 图形化断点调试更是大大提升了调试体验. 对于错误代码也能即时识别输出异常.

二、实现的功能

1. 实现了Y86-Simulator指令集，完成了样例的测试。
2. 实现了额外的`iopq`指令（`iaddq, isubq, iandq, ixorq`），提供样例`123.yo`供测试
3. 增加了CPU的鲁棒性，对于荣誉样例都能够通过。
4. 查看CPU状态

   16个寄存器的值

条件码

5个流水线寄存器

5. 个性化运行

单步运行

调速运行

6. 基于行的调试操作

按代码行查看运行状态

可视化添加断点

单断点运行

三、模拟器的使用方法

1. 模拟器概览

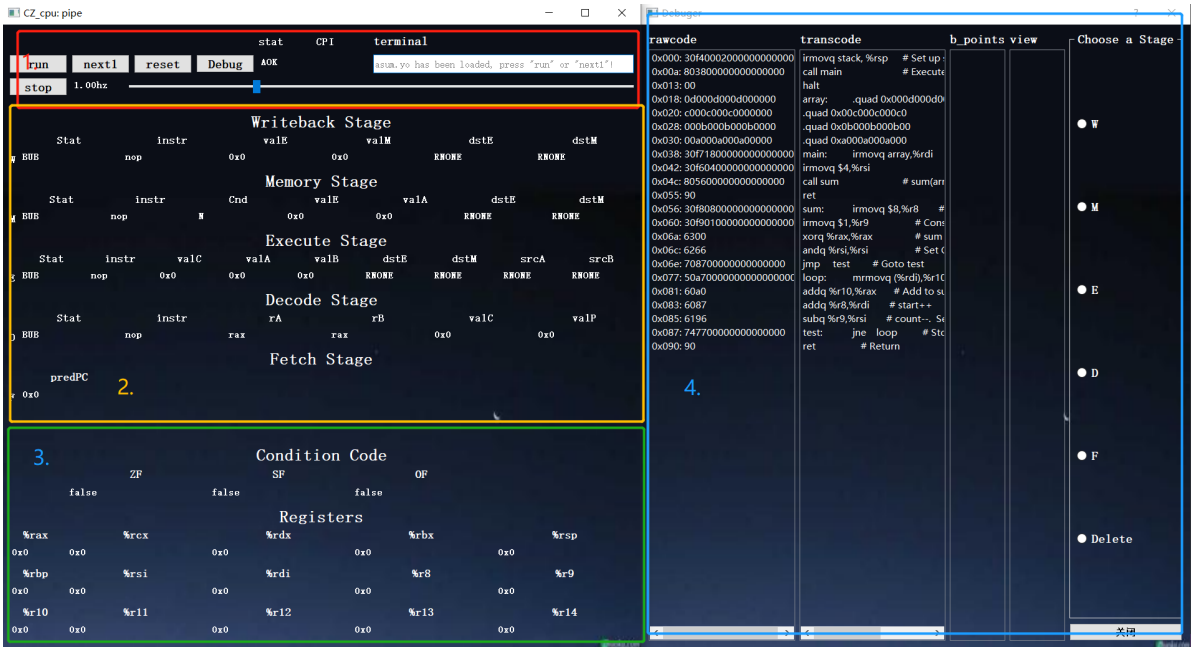
模拟器可以分为4个部分

① 交互控制部分——在这里与UI界面进行交互

② 流水线监视部分——显示各阶段的值

③ 条件码和寄存器

④ debug窗体——在这里可视化添加断点。



2. 基本使用

打开Demo.exe程序后，输入测试样列的名称，（测试样列放在test文件夹下），然后按下回车键.

run按钮: 程序将运行只至结束。如果设置了断点, 会运行至下一断点.

next1按钮: 程序将一个时钟周期一个时钟周期的运行。

stop按钮: 程序停止运行。

速度条: 通过拖动速度条上的光标改变程序的运行速度。

reset按钮: 重置数据，可以再次运行测试。

Debug按钮：点击后，会出现调试界面，出现机器代码，汇编代码以及断点信息和程序各阶段运行到的位置。通过选择阶段后点击代码所在行，可以在该行设置对应阶段的断点（再点一次取消）。此时点击run按钮将会在该行该阶段停下，与主界面的流水线寄存器数据进行观察，完成调试。

四、对提交的代码说明

后端部分代码

代码概述

1. 在`MyISA.h`和`MyISA.cpp`中声明和实现了CPU运行的一些基础变量，完成了一些基础的功能，是流水线运行的基础。
2. `initializing.h`和`initializing.cpp`则是完成了对流水线寄存器中使用的所有全局变量的声明和各个流水线寄存器的数据更新函数。
3. `pipeline.h`和`pipeline.cpp`则是本次CPU模拟的主体，期间实现了各个阶段的具体控制逻辑和操作，完成了 Y86-Simulator 的实现。

具体内容

`MyISA.h`:

对于各个指令以及寄存器(`%rax`,`%rbx`等)的命名，变量类型的命名(Byte,Word)，

```
typedef enum {
    I_HALT, I_NOP, I_RRMOVQ, I_IRMOVQ, I_RMMOVQ, I_MRMOVQ,
    I_ALU, I_JMP, I_CALL, I_RET, I_PUSHQ, I_POPQ,
    I_IOPQ, I_POP2
} itype_t;

typedef enum {
    RAX, RCX, RDX, RBX,
    RSP, RBP, RSI, RDI,
    R8, R9, R10, R11,
    R12, R13, R14, R_NONE = 0xF, ERR
} reg_id_t;

typedef unsigned char Byte;
typedef long long int Word;
```

内存结构的定义以及内存进行初始化，加载，读取，修改，释放等函数的声明：

```
typedef struct {
    int len;
    Word maxaddr;
    Byte* contents;
    bool* if_can_write;
    std::map<Word, int> ref_table;
    std::vector<std::string> insts;
} Mem, *Mem_p;

int load_mem(Mem_p m,std::istream& filename);
inline Byte number(Byte c) { return c - '0'; }
Mem_p init_mem(int);
void clear_mem(Mem_p);
```

```
void free_mem(Mem_p);
bool get_Byte(Mem_p, Word, Byte*);
bool get_Word(Mem_p, Word, Word*);
bool set_Byte(Mem_p,Word,Byte);
bool set_Word(Mem_p,Word,Word);
```

对流水线寄存器结构的定义：

```
struct reg_F {
    Word predPC;
};
struct reg_D {
    Byte stat;
    Byte icode, ifun, rA, rB;
    Word valC, valP;
};
/*...*/
```

对于寄存器读写函数的声明：

```
Word get_rg_val(Byte x);
void set_rg_val(Byte x,Word val);
```

其他函数的声明：ALU计算逻辑的函数，条件码的设置，和对条件是否成立的判断函数：

```
Word compute_alu(alu_t op, Word argA, Word argB);
void setcc(alu_t op, Word argA, Word argB);
bool get_Cnd(cond_t bcond);
```

`MyISA.cpp`

`MyISA.h`文件中声明函数的实现。

`initializing.h`：

其中声明了初始化函数（对需要初始化的数据进行初始化）：

```
void init_all();
```

流水线寄存器数据更新的控制函数函数:

```
void F_updata(bool);
void D_updata(bool,bool);
void E_updata(bool);
void M_updata(bool);
void W_updata(bool);
```

`initializing.cpp`

其中包含了本次PJ模拟中使用到的所有全局变量的声明。以及对应.h文件中声明函数。同时设置了各个寄存器阶段对应的bubble常量，来完成流水线寄存器的初始化和流水线大冒险的控制。

`pipelinefun.h`:

对5个不同流水线阶段模拟的函数声明

```
void fetch();

void decode();

void execute();

void access_memory();

void write_back();
```

对于CPU最终状态的设置函数的声明

```
void set_final_stat();
```

对于流水线大冒险控制函数的声明

```
bool is_F_stall();

bool is_D_stall();

bool is_W_stall();

bool is_D_bubble();

bool is_E_bubble();

bool is_M_bubble();
```

对流水线寄存器数据更新函数的声明(将`initializing.h`中实现的函数封装为一个)

```
void updata();
```

`pipelinefun.cpp`

其中包含了对应.h文件中函数的实现，以及每个阶段进行执行的逻辑控制函数等的实现。

前端部分代码

概览

1. `ui_dialog.h`与`ui_mainwindow.h'`是用`Qtcreator`生成的交互界面代码, 因不了解具体实现, 此处从略.
2. `mainwindow.h`与`mainwindow.cpp`定义了主交互界面`mainwindow`
3. `dialog.h`与`dialog.cpp`定义了调试界面`dialog.h`
4. `decode.h`与`decode.cpp`定义机器码到语义语言的译码函数

具体内容

`mainwindow`类

1. `mainwindow.h`定义mainwindow类的成员, `debuger`为交互界面对话框, `runOK`指示模拟器是否出于非异常状态可被运行, `fTimer`是用于周期跳转计时器, 另外`slots`中的信号槽, 对应各个按钮的交互逻辑, 计时逻辑.

```
class Dialog{/*...*/};
class MainWindow//...
private slots:
    void on_next1_clicked(bool checked);//next1 clicked
    void on_run_clicked(bool checked);//...
    void on_inname_returnPressed();//输入框中回车确认载入测试文件
    void on_reset_clicked(bool checked);//重置, 可以再次运行
    void on_Debug_clicked();//打开调试窗口
    void on_timer_timeout();//每次时钟周期计时满触发功能
    void on_speed_valueChanged(int value);//调整时钟快慢
    void on_hlt_clicked();//stop pressed
private:
    Ui::MainWindow *ui;
    Dialog debuger;
    bool runOK;
    QTimer *fTimer;
```

2. 在`mainwindow.cpp`中首先把一周期的CPU运行实现在函数`on_next1_clicked()`中, 然后其他函数都关联在这一个函数下, 他们或是触发这个函数, 或是设置触发它的速度和条件

`dialog`类

0. 在`dialog.h`中我们可以看到调试器的大体架构, 调试器的实现通过维护`QStringListModel`来实现, 我们在读代码时, 把每行代码处理为一个`QString`, 然后放入`QStringListModel`, 这样可以很好的按照行号管理各行代码, 方便用户调试.

```
//from :dialog.h
#include <QDialog>
#include <QStringListModel>
class Dialog :public QDialog{/*...*/
private:
    QStringListModel *strm;//按行盛放机器可识别的码(rawcode)
    QStringListModel *strm1;//按行盛放程序员友善的汇编码(transcode)
    QStringListModel *locm;//用于展示各阶段索出位置
    QStringListModel *pointsm;//用于让用户单击添加断点(points)
private slots:
    void setDebug_stage();//选择在哪个阶段添加断点
    void on_points_clicked(const QModelIndex &index);//单击对应行设置断点
    void on_rawcode_clicked(const QModelIndex &index);
    void on_transcode_clicked(const QModelIndex &index);
}
```

1. 可视化断点实现. `on_points_clicked()`
用户通过单击StringList, 获取他单击的行号, 将StringList对象`pointsm`对应位置更改为断点.

```
//选择器setDebug_stage()通过用户选择设置Debug_stage为"D", "E", "F", "W", "M", "
"()
void Dialog::on_points_clicked(const QModelIndex &index)//单击更改断点
{
   /*...*/ //为了更好的用户体验, 我们在这里实现了再次单机取消断点的功能
   pointsm->setData(index, QString(Debug_stage));
}
void Dialog::on_rawcode_clicked(const QModelIndex &index)
{
   on_points_clicked(index);//关联,单机代码位置也可以设置断点槽,操作会体验更佳
}
void Dialog::on_rawcode_clicked...//同样地关联
```

2. 运行状态展示实现. `statenew()`
   底层运行时, 把每条指令的首地址存储起来(如`pipe_F.curpc`), 维护一个首地址到行号的查找表`map<Word> ref_table`, 每周期运行后将这个Stringlist对应位置字符更改. 同时可以查看是否到达了断点应该暂停.

```
//from mainwindow.cpp
void MainWindow::on_next1_clicked(bool checked)//每周期运行
{
    /*...*/
    debuger.statenew();
}
```

```
//from dialog.cpp
void Dialog::statenew()
{
    if(memory->ref_table.find(pipe_F.curpc) != memory->ref_table.end())
        locm->setData(locm->index(memory->ref_table[pipe_F.curpc]),
QString("F"));
    //当展示StringList locm与断点StringList locm有重合时, 表示到达断点应该暂停
    QStringList locsl = locm->stringList();
    QStringList pointsl = pointsm->stringList();
    for(int i = 0, sz = locsl.size();i != sz;i++)
    {
        if(locsl[i][0] != ' ' && locsl[i][0] == pointsl[i][0])
        {
            pausepoint = 1;//bool pausepoint表示应该暂停
            break;
        }
    }
}
```

```
//from mainwindow.cpp, 每次run点下, cpu会周期运行直至pausepoint为1
void MainWindow::on_run_clicked(bool checked)
{
    debuger.pausepoint = 0;
    if(!fTimer->isActive())
    {
        fTimer->start(runtime_speed);
    }
}
void MainWindow::on_timer_timeout()
{
```

```
    if(!debuger.pausepoint)
        on_next1_clicked(true);
}
```

`decode`函数

1. decode提供的译码功能包括条件码, 指令码, 状态码, 迭代器码

```
QString de_Cnd(bool);
QString de_instr(Byte, Byte=0);
QString de_stat(Byte);
QString de_reg(Byte);
```

2. 用数组来实现译码, 错误的指令会发生数组越界, 故实现时加入了判断

```
QString de_instr(Byte icode, Byte ifun) {
    if(icode > 12)
        return "NO INS";
    if (icode == 2) {
        if(ifun >= 7)
            return "NO INS";
        return cmov[ifun];
    }
    /*...*/
}
```

五、实现的细节

后端：

1. 对于基本操作的实现细节不加赘述。
2. 对于额外的`iopq`指令，在现行的流水线框架下，经过观察期实现逻辑与`irmovq`和之前的`opq`指令类似，在加入指令时可以参考之前这两种指令的实现。
3. 对于CPU的鲁棒性：

   首先在fetch阶段进行各种判断，通过对一些指令的特判，使得可以知道该指令是否有效。

   对于一些其他情况，如`irmovq`指令中没有`valc`的情况，开始想法是在加载内存时进行判断，后来 觉得认为如果`irmovq`指令如果后面还有指令的话，那么按照CPU的处理逻辑会将后续指令的一部分当作`valc`，所以这里不应该简单的在指令的加载阶段就报错。我的解决方法是将指令集在内存中使用的长度算出来，如果需要的`valc`的储存小于这个长度应该报错。
4. 对于一些指令存放的地址不应该被修改的问题，我的解决较为生硬，偏向于面向样例的解决：将已经读过的指令都设置为不可读的状态。存在的问题是，对于修改未执行的指令集，并不能做出判断。
5. 对于前段实现调试功能，我们在内存中添加了一个ref_table，将地址和行号对应起来，之后再将地址通过流水线寄存器传递下去，就可以通过对每个流水线处理指令的首地址和对应的指令行号联系起来，方便调试和展示功能。

前端:

1. 测试样例的读取有很高的开放性, 输入错误, 未载入样例便开始运行, 都会使程序崩溃, 我们实现了对每个情况的报错, 在输入框中对用户展示.
2. 在测试的过程中我们发现了很多机器友好, 但和用户习惯相悖的操作特性, 我们都进行了优化:

   在已经加入的断点上再次点击, 会取消这个断点而不是做一次值不变的修改.

   reset按钮按下时, 断点会保留而不是一起删去.

   当各阶段指令无法在`ref_table`上查到值时, 会隐藏该指令而不是在第零行输出

六、队员分工情况

工作分工：后端基本由陈洋完成，前端由张开元完成。在完成基本的操作后，再合力完成附加功能的实现。