

Y86-64 阶段一、二实验报告

姓名：罗旭川

学号：17307130162

实验时间：2018.11.1-2018.12.2

一、实验目的

编写一个 Y86-64 流水线 CPU 模拟器

二、实验具体过程

A、阶段一：模拟器的正确性与基础功能（非 GUI 版）

1. 构思

①语言：考虑用 C++ 实现该 Y86-64 模拟器

②模拟方式：

(1) 用一个 `vector<string>` `instructions` 存储读入的指令。

(2) 用 15 个 `long long` 型变量模拟由 `%rax` 到 `%r14` 共 15 个寄存器。

(3) 为节约内存，考虑用 `map` 模拟内存空间。

(4) 为实现 pipe 型流水线，用若干变量模拟出所有 pipe register 上的数据。

(5) 为了实现 Fetch、Decode、Execute、Memory、Write back 五个阶段同时并行的效果，考虑运用 C++ 的多线程库 `thread`。用 5 个线程分别反复执行五个阶段的操作，并用 5 个全局指针 `point_f`、`point_d`、`point_e`、`point_m`、`point_w` 分别指示每个线程需要执行的语句。接下来只要控制指针的递增和特殊情况下指针的跳跃即可。

③输入和输出：用 `fstream` 库实现从 `.yo` 文件读取指令，最后把运行过程和运行结果的数据写入 `.txt` 文件中，这也是为了方便阶段二 GUI 的实现。

2. 几个重要的 Y86-64 功能及其实现方法

①多线程实现 5 个 stage 并行

```
thread t1(fetch, point_f);
thread t2(decode, point_d);
thread t3(execute, point_e);
thread t4(memory, point_m);
thread t5(write_back, point_w);
t1.join();
t2.join();
t3.join();
t4.join();
t5.join();
```

```
void fetch(int point);
void decode(int point);
void execute(int point);
void memory(int point);
void write_back(int point);
```

通过调用 C++ 的 `thread` 库，将 Fetch、Decode、Execute、Memory、Write back 五个阶段分别放在 5 个线程中，并用 5 个全局的指针 `point_f`、`point_d`、`point_e`、`point_m`、`point_w` 分别指示每个线程需要执行的语句，并将相应指针传入相应的 5 个 stage 函数中

②解决 加载/使用冒险 的问题

```
bool has_repeat(int a)
{
    if (a == E_dstE || a == E_dstM || a == M_dstE || a == M_dstM || a == W_dstE || a == W_dstM) return true;
    return false;
}

bool forward_icode(char c)
{
    if (c == E_icode || c == M_icode || c == W_icode) return true;
    else return false;
}

bool check_forward()
{
    switch (D_icode)
    {
        case '0':
        case '1':
        case '3':
        case '7': return false;
        case '2':
        case '4':
            if (has_repeat(rA)) return true;
            else return false;
        case '5': //mrmovq
            if (has_repeat(rB)) return true;
            else return false;
        case '6': //OPq
            if (has_repeat(rA) || has_repeat(rB)) return true;
            else return false;

        case 'A': //push
        case 'a':
            if (has_repeat(rA)) return true;
        case '8': //call
        case '9': //ret
        case 'B': //pop
        case 'b':
            if (has_repeat(4) || forward_icode(8) || forward_icode('A') || forward_icode('B')) return true; //rsp
            else return false;

        default: return false;
    }
}
```

写了一个 check_forward 函数，通过检查 Fetch 阶段之后的每个阶段中的 dstE 和 dstM 值是否存在与当前 Fetch 阶段的 rA、rB 相同的情况，以此来判断是否出现加载/使用冒险。

注意：%rsp 牵涉到的 push、pop、call、ret 指令要特别判断，因为 push 和 pop 的 rA 不一定是 %rsp，而 call、ret 没有 rA、rB。

```
else if (wait == true) //conditon: forward
{
    point_w = point_m; point_m = point_e; point_e = point_d; point_d = -1; //fetch stage repeat the same.
    wait = false;
}
```

若出现了该冒险，则会控制 Fetch 暂停，直到后续阶段的语句全部执行完后该语句才会继续执行。(point=-1 代表设置了气泡 Bubble)

③完美 处理 ret 语句

```

else if (ret_signal == true)    //condition: ret
{
    if (get_retadd == true)
    {
        point_w = point_m; point_m = point_e; point_e = point_d; point_d = point_f;
        point_f = location_point[predPC]; PC = predPC;
        ret_signal = false;
    }
    else
    {
        point_w = point_m; point_m = point_e; point_e = point_d; point_d = point_f;
        point_f = -1;
    }
}

```

通过设置两个全局的 bool 变量 ret_signal、get_retadd，分别表示 Fetch 阶段读取到 ret 指令 和 Write back 阶段已将 return address 写入 PC。

当 ret_signal 为真而 get_retadd 为假时，表示刚读取到 ret 指令。此时令 ret 指令及其之前的指令继续执行，直至 get_retadd 为真，即 ret 执行完 Write back 阶段为止。而 ret 之后的指令不会再进入流水线（通过设置 Fetch 阶段为气泡来实现）。

④解决 预测分支错误 的问题

```

if (M_cnd == 0)                //condition: jump false
{
    point_f = location_point[predPC]; PC = predPC; point_d = -1; point_w = point_m; point_m = point_e; point_e = -1;
    ret_signal = false;
    m_cnd = 1;
}

```

在执行 jXX 指令时，会按照优先执行分支的原则执行。当执行到 Execute 阶段时，若发现猜测正确，则 cnd=1，程序继续执行；若发现猜测错误，即不应该执行分支，则令 cnd=0，并重置 predPC。当下一周期开始时，Fetch 阶段读取重置的 PredPC 来执行，jXX 及其之前的语句正常执行，而 jXX 之后的语句全部置为气泡 Bubble。这样就重新回到了正确执行的情形。

⑤异常情况的显示和处理

(1)INS

```

switch (instructions[point][7])
{
    case '0':halt(point, stage); break;
    case '1':nop(point, stage); break;
    case '2':rrmovq(point, stage); break;
    case '3':irmovq(point, stage); break;
    case '4':rrmovq(point, stage); break;
    case '5':rrmovq(point, stage); break;
    case '6':OPq(point, stage); break;
    case '7':jXX(point, stage); break;
    case '8':call(point, stage); break;
    case '9':ret(point, stage); break;
    case 'a':
    case 'A':pushq(point, stage); break;
    case 'b':
    case 'B':popq(point, stage); break;
    case 'c':
    case 'C':iaddq(point, stage);break;
    default: nop(point, 'X'); instr_nums[13]++; break;
}

```

```

case 'X':
    //We ignore INS in instructions and pass this wrong instruction.
    output_F(PC);

    d_stat = "INS";
    error = "INS";
    d_icode = '-';
    d_ifun = 0;
    rA = 15;
    rB = 15;
    d_valC = 0;
    d_valP = predPC = PC + (instructions[point].size()-7)/2;

    break;
}

```

在判断语句的类型时，若发现指令的 icode 值超过了 Y86-64 指令集的范围，则将该指令当作 nop 指令执行，并传入标识符 'X' 来表示其为无效指令。考虑到改种 INS 错误对前后语句的影响不大，因此程序不会停止运行。

注意：本程序新增了 iaddq 指令，使得测例 asumi.yo 能得到正确结果，而不被当作无效指令跳过，这与范例 pipe 不同。

```

case 'F':
    output_F(PC);

    if (check_r(point) == false)
    {
        //if the register code is false, we stop the whole simulator.
        instr_nums[13]++;
        error = "INS";
        error = "INS";
        stop = true;
        return;
    }

```

当在 Fetch 阶段时，发现指令的 rA、rB 值为无效值，则判断为 INS 错误指令。因为该种 INS 错误会影响到前后语句的执行和整个程序运行结果的正确性，所以当遇到该种 INS 错误时，程序会直接停止运行。

(2)HLT

```

case 'E':
    output_E("halt", E_valC, E_valA, E_valB, E_srcA, E_srcB, E_dstE, E_dstM, E_stat);
    instr_nums[0]++;
    //HLT makes the afterward instructions stop.
    m_stat = E_stat;
    halt_stop = true;

else if (halt_stop == true) //condition: halt
{
    point_f = -1; point_d = -1; point_w = point_m; point_m = point_e; point_e = -1;
}

```

当 Execute 读取到 halt 指令时，令一个全局 bool 变量 halt_stop=true。这会使程序将 halt 语句之后的语句（在 Fetch 和 Decode 阶段的语句）置为气泡 Bubble，这样就避免了执行 halt 后续的语句。而 halt 及其之前的指令会继续执行，直至全部执行完毕，程序停止。

(3)ADR

```

if (PC>max_location)
{
    error = "ADR"; no_pro = false; stop = true; return;
} //set ADR

```

在 Fetch 阶段时，若 PC 指向的地址值超出了给定的最大内存算出的最大地址 max_location，则判断为 ADR 异常，程序直接停止。

```

if (M_valE >= 0 && M_valE<=max_location)
{
    changed_Memory[M_valE] = Memory[M_valE];
    Memory[M_valE] = M_valA;
    Mem_stat = " Wrote valA to memory D(valE). ";
}
else
{
    error = w_stat = "ADR";
    no_pro = false;
    Mem_stat = " Can't wrote to an invalid address.";
    stop = true;
}

```

```

if (M_valA >= 0 && M_valA <= max_location)
{
    w_valM = Memory[M_valA];
    Mem_stat = " Read valM from memory D(valA). ";
}
else
{
    error = w_stat = "ADR";
    no_pro = false;
    Mem_stat = " Can't read from an invalid address.";
    stop = true;
}

```

在 Memory 阶段时，若 valE 或 valA 的值超出了最大地址值 max_location 或者 valE<0，则无法读取或写入内存，因此判断为 ADR 异常，程序直接停止。

⑥统计每种语句的执行数量

```
extern int instr_nums[14];
```

设置一个专门统计执行的各种语句的数量的数组即可。

⑦性能分析（运行时间、CPI)

```

clock_t start, finish;
double totaltime;
start = clock();

//timer stop.
finish = clock();
totaltime = (double)(finish - start) / CLOCKS_PER_SEC;

```

利用 C++ 的 time.h 头文件，在每次测例运行的开头和结尾分别记录时钟，最后相减即能得到运行时间。

```
//CPI.  
double CPI = ( i+0.0) / instructions.size();
```

运用公式计算 CPI，即每指令周期数。

通过运行时间和 CPI 就能大致衡量出本程序的性能和运行效率。

3. 遇到的难题及其解决方案

①线程冲突问题

由于本程序是通过 5 个线程同时运行来实现 pipe 流水线功能的，因此若不对 5 个线程之间进行一些相互牵制的话，很容易出现一个线程跑得比另一个线程快，最终导致运行混乱。通过网上查阅，这个问题可以通过利用 C++ 的 mutex 库给 5 个线程之间加锁来解决。

```
void fetch(int point)  
{  
    lock_guard<mutex> lck(mt);  
}  
  
void decode(int point)  
{  
    lock_guard<mutex> lck(mt);  
}  
  
void execute(int point)  
{  
    lock_guard<mutex> lck(mt);  
}  
  
void memory(int point)  
{  
    lock_guard<mutex> lck(mt);  
}  
  
void write_back(int point)  
{  
    lock_guard<mutex> lck(mt);  
}
```

通过 lock_gard<mutex> lck(mt)语句来将 5 个 stages 的函数（线程）锁在一起，这样的效果是，对于每一个 stages 函数，必须要等待上一个执行这个函数的线程执行完毕后，另外一个线程才能进入这个函数进行执行。这样就能保证同一时刻至多只有一个线程对每一个 stage 对象执行操作，从而避免了程序的混乱。

②内存问题

>>问题出现原因：

```
0x018: | # Array of 4 elements  
0x018: | .align 8  
0x018: 0d000d000d000000 | array: .quad 0x0000000d000d000d  
0x020: 40ff3fff3fffffff | .quad 0xfffff3fff3fff40 # -0x000000c000c000c0  
0x028: 000b000b000b0000 | .quad 0x00000b000b000b00  
0x030: 0060ff5fff5fffff | .quad 0xffff5fff5fff6000 # -0x0000a000a000a000
```

一开始在跑 abs-asum-cmov.yo 这个测例时，我的结果总是错误，后来发现原因是出在我没有把读入的指令写入内存里，而只是单独地放在 instructions 数组里，这导致上述数据没有读入，而造成结果错误。

>>解决办法：

```
Memory[locate] = shift_valC(instructions.size()-1,7);
```

在读取指令的同时将指令写入相应的地址即可。

③数据类型的转换问题

>>问题出现原因:

由于读入的指令是字符串形式的，而程序运行过程中常常需要进行数值计算，因此需要想办法将字符串转化为数值

>>解决办法:

<code>int</code>	<code>stoi(const std::string& str, std::size_t* pos = 0, int base = 10);</code>	(1) (since C++11)
<code>int</code>	<code>stoi(const std::wstring& str, std::size_t* pos = 0, int base = 10);</code>	
<code>long</code>	<code>stol(const std::string& str, std::size_t* pos = 0, int base = 10);</code>	(2) (since C++11)
<code>long</code>	<code>stol(const std::wstring& str, std::size_t* pos = 0, int base = 10);</code>	
<code>long long</code>	<code>stoll(const std::string& str, std::size_t* pos = 0, int base = 10);</code>	(3) (since C++11)
<code>long long</code>	<code>stoll(const std::wstring& str, std::size_t* pos = 0, int base = 10);</code>	

使用 C++11 中的 `stoi`、`stol`、`stoll` 函数，可以将字符串直接转化为相应的 `int`、`long`、`long long` 类型

```
long long complement(string &str)
{
    long long ans=0;
    for (int i = 2; i < str.size(); i++)
        ans = (ans<<4) + (15-char_int(str[i]));
    return -(ans + 1);
}
```

对于补码形式储存的负数，我自己编写了将补码字符串转化为相应负数数值的函数如上。

④数据大小问题

>>问题出现原因:

在第一次编写完这个 Y86-64 程序时，我的许多变量还是用 `int` 类型储存，最后导致在许多测例上出现结果错误

>>解决办法:

改用 `long long` 类型储存可能较大的数据，转换上述函数也只能用 `stoll` 而不能用 `stoi` 了

4. 本程序的特点

①调试方便，可变动性强

由于本程序中各指令的实现是分开编写的，因此各指令的程序相互独立，当遇见 bug 时也能很快找到。另外，由于各指令的实现是分开的，所以也可以很方便的添加新的指令类型进去而不影响其他指令的正确实现。（比如在本程序中新添了 `iaddq` 指令）

②输出整洁

```
##### This is the beginning of prog1.yo #####
```

```
-----  
*****Circle 0*****  
ZF=1 SF=0 OF=0 Stat=AOK  
F : predPC=0x0  
D : Decode stage is doing nothing.    State=BUB  
E : Execute stage is doing nothing.    State=BUB  
M: Memory stage is doing nothing.    State=BUB  
W: Write-back stage is doing nothing. State=BUB  
    Execute: ALU:  -----  
    Memory:  -----  
    Writeback: -----  
*****  
  
*****Circle 1*****  
ZF=1 SF=0 OF=0 Stat=AOK  
F : predPC=0xa
```

由于输出采用文件输出，因此不会出现数据结果疯狂刷屏的情形。并且输出时每个 circle 用花边隔开，每个 stage 也对齐输出，整体看起来很美观、整洁。

③运行效率较高

由于本程序运用了 C++ 的 thread 库实现了多线程运行，因此运行效率较高。

B、阶段二：模拟器的附加功能与完整性（GUI 版）

1. 构思

①语言：考虑用 Qt 做 GUI 界面

②框架构思：

- (1)为了使得界面整洁易用，考虑用菜单栏的形式填装各种功能。
- (2)为了能同时实现输入和输出，考虑用两个文本框，一个用于输入，一个用于输出。
- (3)为了使界面不过于简单，应该增加一些辅助性器件，如工具栏、进度条、按键等；还可以设计一下背景图片、按钮样式等。
- (4)个人偏好的界面风格为清新脱俗，活泼可爱。

2. 实现的功能及其实现方法

①运行功能

A、描述：全部运行/单个运行/自定义输入运行 三种运行方式可选，结果显示在 TextBrowser 中，过程状态和最终状态分开显示，点击 result 按钮可查看最终状态

B、实现方法：

(1)先在非 GUI 版本中实现这三种运行方式

```
cout << "Print 'test_begin.' to start all the Y86-64 tests one by one," << endl;
cout << "or print 'test_one.' to test the one test you want, or print 'order.' to send an instant code." << endl;
cin >> order;

if (order == "test_begin.")
{
    read_filename(order);

else if (order == "test_one.")
{

else if (order == "order.")
{

else
{
    cout << "It's a wrong order." << endl;
    exit(0);
}
```

>>test_begin. 是全部运行模式。在该模式下，程序将自动分别读入并运行全部 21 个测例（通过事先建立好含有所有测例的名称的 test_list 文件来实现），

在每个测例运行前，会先清空模拟内存和模拟寄存器以及各种变量；在每个测例运行完后，会分别生成 test_process 和 test_result 文本，分别放在两个文件夹中，分别包含了运行中每个周期的状态以及运行结束后的最终状态。

>>tese_one. 是单个运行模式。在该模式下，程序会让用户输入想要运行的测例文件的名称，然后仅运行该测例，然后生成对应的 test_process 和 test_result 文本。

>>order. 是自定义输入运行模式。在该模式下，用户可以输入符合规范的 Y86-64 指令集中的数字编码（不允许空行），以“EOF”作为输入结束的标志。输入完成会立即会打印出

test_process 和 test_result.

(2)在 VS 中实现了上述三种运行方式后，接下来利用 Qt 中的 QProcess 类，来实现与命令行 cmd 通信，从而通过 cmd 发出执行这个源程序的指令,如下图所示：

```
connect(cmd, SIGNAL(readyReadStandardOutput()), this, SLOT(runalloutput()));

get_string(QDir::currentPath());
cmd->write("cd ");
cmd->write(_string);cmd->write("\n");
cmd->write("cd ../../Y86-64_VS/River's_Y86-64/Release\n");
cmd->write(" .\\River's_Y86-64.exe\n");
```

其中 connect(cmd, SIGNAL(readyReadStandardOutput()), this, SLOT(runalloutput()));将 cmd 的输出和自定义的一个 runalloutput()函数连接起来，之后将利用后者将 cmd 的输出打印在 TextBrowser 文本框中；

QDir::currentPath()返回的是当前程序的路径，结合这个路径，就可以利用相对位置来找到阶段一所生成的 Y86-64.exe，再输入打开命令即可。

(3)同时设置 3 个不同的选项（按键），分别对应于发出上述 3 种不同的指令。

指令 1:

```
cmd->write("test_begin.\n");
```

指令 2:

```
cmd->write("test_one.\n");
```

指令 3:

```
cmd->write("order.\n");

QString code=textEdit->document()->toPlainText();
cmd->write(code.toLocal8Bit() + "\n");
cmd->write("EOF\n");
```

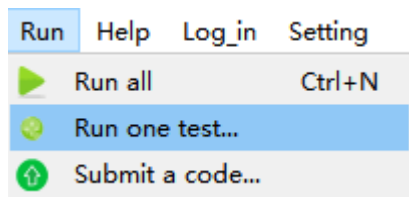
(4)运用 QProcess 的成员函数 readAllStandardOutput()获取命令行的返回内容

```
textBrowser->append(cmd->readAllStandardOutput().data());
```

(5)利用 Qt 中的 QFile 类打开 VS 代码所生成的 test_process 和 test_result 文件，并打印在 TextBrowser 中即可

```
QTextStream in(&file);
textBrowser->setText(in.readAll());
```

C、运行效果



```

open prog7.yo successfully. Loading...
open prog8.yo successfully. Loading...
open prog9.yo successfully. Loading...
open prog10.yo successfully. Loading...
open abs-asum-cmov.yo successfully. Loading...
open abs-asum-jmp.yo successfully. Loading...

```

```

##### This is the beginning of prog7.yo #####

*****Circle 0*****
ZF=1 SF=0 OF=0 Stat=AOK
F : predPC=0x0
D : Decode stage is doing nothing. State=BUB
E : Execute stage is doing nothing. State=BUB
M: Memory stage is doing nothing. State=BUB
W: Write-back stage is doing nothing. State=BUB
Execute: ALU: _____
Memory: _____

```

②调试功能

A、描述

- 允许用户打开测例文件进行编辑
- 用户在测例文件或是自定义的指令代码中加上标识符“break”可以实现添加断点功能
- 用户可以保存更改的测例文件或是自己编写的指令代码

B、实现方法

(1)通过下述两个代码分别实现打开文件和保存文件两种行为

```

void MainWindow::openFile()
{
    QString path = QFileDialog::getOpenFileName(this,
        tr("Open File"),
        "../Y86-64_VS/file",
        tr("Text Files (*.txt *.yo)"),
        0, QFileDialog::ShowDirsOnly);

    if(!path.isEmpty()) {
        QFile file(path);
        if (!file.open(QIODevice::ReadOnly)) {
            QMessageBox::warning(this, tr("Read File"),
                tr("Cannot open file:\n%1").arg(path));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    } else {
        QMessageBox::warning(this, tr("Path"),
            tr("You did not select any file."));
    }
}

void MainWindow::saveFile()
{
    QString path = QFileDialog::getSaveFileName(this,
        tr("Open File"),
        "../Y86-64_VS/file",
        tr("Text Files (*.txt *.yo)"),
        0, QFileDialog::ShowDirsOnly);

    if(!path.isEmpty()) {
        QFile file(path);
        if (!file.open(QIODevice::WriteOnly)) {
            QMessageBox::warning(this, tr("Write File"),
                tr("Cannot save file:\n%1").arg(path));
            return;
        }
        QTextStream out(&file);
        out << textEdit->toPlainText();
        file.close();
    } else {
        QMessageBox::warning(this, tr("Path"),
            tr("You did not select any file."));
    }
}

```

(2)设置断点的功能考虑现在 VS 代码中实现

```

while (getline(cin, instant_code))
{
    if (instant_code == "EOF")
    {
        pipeline(filename);
        break;
    }
    else if (instant_code == "break")
    {
        pipeline(filename);
        instructions.clear();
        isbreak = true;
        continue;
    }
    else
    {
        if (isempty(instant_code)) continue;
        while (instant_code.size() < 27) instant_code.push_back(' ');
        instructions.push_back(instant_code);
        long long locate = stoll(instant_code.substr(0, 5), NULL, 16);
        location_point[locate] = instructions.size() - 1;
        Memory[locate] = shift_valC(instructions.size() - 1, 7);
    }
}

```

break 是我们打在 TextEdit 文本框中的标识符，最终将被 QProcess 输入进 cmd，从而发送给阶段一的.exe 文件，因此要实现断点功能应该从原始代码中接收的字符串入手

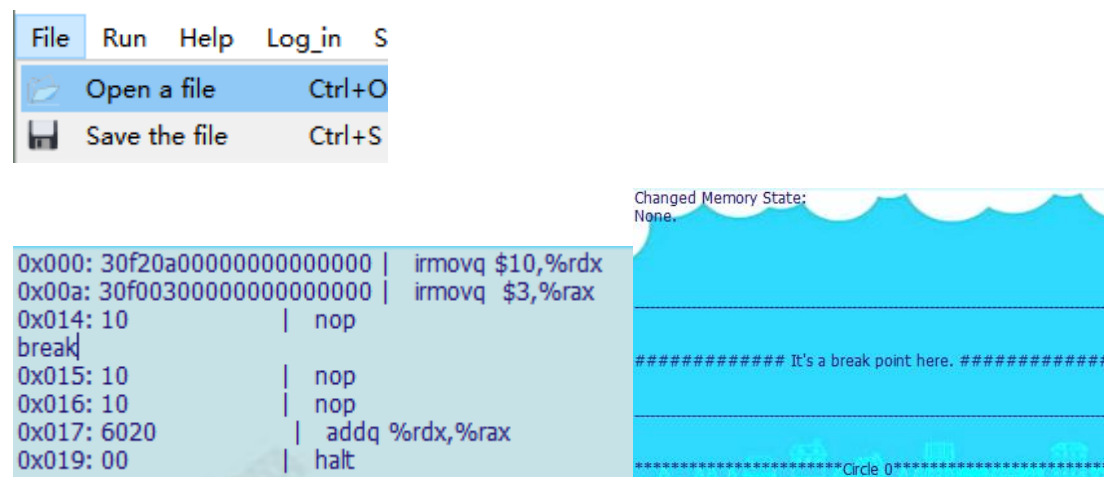
>>如果接收的不是“break”也不是“EOF”标识符，则将指令存入 instructions 中

>>如果接收到“break”，那么不再将指令储存进 instructions 数组中，而是直接执行已经存在 instructions 中的指令（即 break 之前的指令）。到 break 为止的运行状态会被打印出来；

>>如果接收到“EOF”，将剩余在 instructions 中的指令送进流水线执行，执行完后程序结束。

在原始代码中修改完毕后，打印在界面上的运行结果也就成为了调试结果了。

C、运行效果



PS:左图以淡紫色为背景的是 TextEdit 文本框，用于输入；右图以蓝色为背景的是 TextBrowser 文本框，用于输出。并且在本实验报告中总是如此。

③设置功能

A、描述

允许用户设置内存空间的大小以及该 CPU 模拟器运行的速度

B、实现方法

(1)同样新增这个功能需要从原始代码开始改起，通过增加以下两个输入来实现：

```
cout << "Memory Size is:";
cin >> m; cout << m << endl;
max_location = m * 1024 * 1024;
cout << "Speed level is:"<<endl;
cin >> speed; cout << speed << endl;

while (1)
{
    Sleep(20*(5-speed));
```

允许用户输入自定义的内存大小以及期望的运行速度等级。

>>通过输入的内存大小可以得出地址的最大值，因此在程序运行过程中将以该内存地址 max_location 最大值作为判断 ADK 异常的依据。

>>根据输入的运行速度大小，结合 Sleep 函数，模拟出程序运行的快慢区别。运行速度每降一级，每个周期延时 20 毫秒。

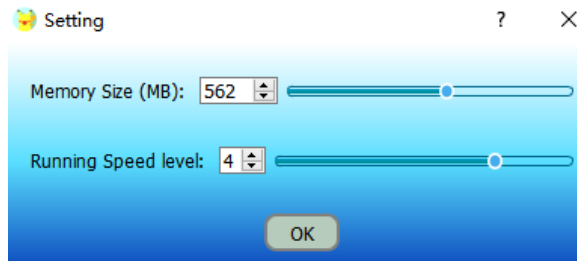
(2)在 Qt 界面中，可以设计两个滑块分别代表用户设置的内存大小和运行速度，然后再将

这两个设置的值通过 QProcess 传送给原始代码作为其输入进行运行即可

```
get_string2(QString::number(max_memory));
cmd->write(_string2);cmd->write("\n");

get_string2(QString::number(run_speed));
cmd->write(_string2);cmd->write("\n");
```

C、运行效果



```
CPI: 1.42857
Runtime: 0.403 s
Memory Size: 1000MB
Speed Level: 5
```

```
CPI: 1.42857
Runtime: 0.471 s
Memory Size: 100MB
Speed Level: 4
```

同一测例在不同的速度等级下得运行时间出现差异

④登陆/注销功能

A、描述

允许用户输入用户名及密码来登陆账号，登陆后可以快速注销

B、实现方法

创建一个 QDialog 类，添加 QLabel 和 QLineEdit 来实现输入用户名和密码的文本框，然后连接到一个判断填写是否正确的槽函数，若填写正确则显示一个写有“Welcome back”的 QLabel。

```
void MainWindow::Login()
{
    logindialog = new QDialog(this);
    logindialog->setMinimumSize(200,200);
    logindialog->setAttribute(Qt::WA_DeleteOnClose);
    logindialog->setWindowTitle(tr("Log in"));
    logindialog->setStatusTip(tr("Log in to get more power.));

    QLabel *input_id = new QLabel(QWidget::tr("Your ID :"));
    id = new QLineEdit();
    id->setPlaceholderText(tr("Please input your ID"));
    QHBoxLayout *h1 = new QHBoxLayout;
    h1->addWidget(input_id);
    h1->addWidget(id);
    QWidget *w1 = new QWidget;
    w1->setLayout(h1);

    QLabel *input_password = new QLabel(QWidget::tr("Password :"));
    password = new QLineEdit();
    password->setEchoMode(QLineEdit::Password);
    password->setPlaceholderText(tr("The password.));
    QHBoxLayout *h2 = new QHBoxLayout;
    h2->addWidget(input_password);
    h2->addWidget(password);
    QWidget *w2 = new QWidget;
    w2->setLayout(h2);

    QPushButton *b = new QPushButton;
    b->setMinimumSize(50,25);
    b->setText(tr("Log in"));
    connect(b,&QPushButton::clicked,this,&MainWindow::judgeID);
}
```

```
void MainWindow::judgeID()
{
    if(id->text() == "River" &&
       password->text() == "17307130162")
    {
        success();
    }
    else {
        QMessageBox::warning(this, tr("Warning !!"),
                              tr("Wrong ID or password."),
                              QMessageBox::Yes);
    }
}
```

C、运行效果



⑤帮助功能

A、描述

点击一个按钮，可以打开该 Y86-64 的使用说明书；或者扫描二维码，用手机查看说明书

B、实现方法

分别将两个按钮连接到两个函数，分别实现打开 README.txt、打开一个 Qt.png，函数的实现代码如下：

```
void MainWindow::Readme()
{
    QFile file("../README.txt");
    if (!file.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("Read File"),
            tr("Cannot open file:\n%1").arg(Path));
        return;
    }
    QTextStream in(&file);
    textBrowser->setText(in.readAll());
    file.close();
}

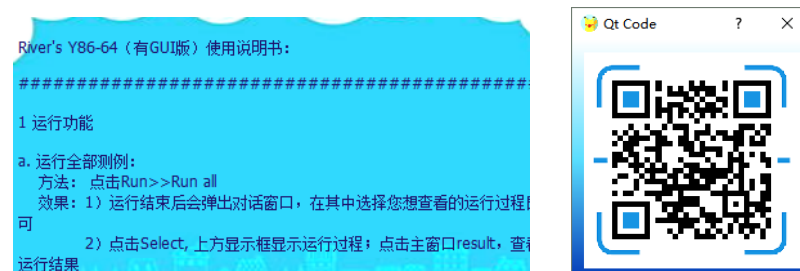
void MainWindow::get_qtcode()
{
    QDialog *dialog = new QDialog(this);
    dialog->setMinimumSize(200,200);
    dialog->setAttribute(Qt::WA_DeleteOnClose);
    dialog->setWindowTitle(tr("Qt Code"));
    dialog->setStatusTip(tr("Scan the qt_code to get a quick help.));

    QPixmap pixmap(":/image/Qt3");
    QPixmap pixmap2 = pixmap.scaled(200, 200);
    QLabel *ww=new QLabel;
    ww->setPixmap(pixmap2);

    QHBoxLayout *ll=new QHBoxLayout;
    ll->addWidget(ww);
    dialog->setLayout(ll);

    dialog->show();
}
```

C、运行效果



⑥其他功能

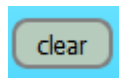
1) clear 功能

效果：清空上下两个文本框，并使程序进入初始状态

实现方法：将一个按钮连接到以下 cleanall 函数即可：

```
void MainWindow::cleanall()
{
    count=0;
    textBrowser->clear();
    textEdit->clear();
    progress->setValue(0);
    progress->setFormat(tr("Current progress : %1%").arg(QString::number(0, 'f', 0)));
}
```

运行效果：



2) 工具栏功能

效果：本程序具有强大的工具栏功能，点击工具栏上的按键，就能实现与图标对应的功能。

实现方法：对于每一个菜单栏中的功能，都找一个合适的图标，让用户能一眼明白它的功能，然后添加到创建的工具栏中

运行效果：



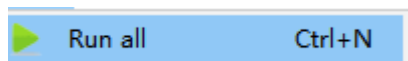
3) 快捷键功能

效果：一些使用频繁的功能设置了快捷键，具体在菜单栏上有明显提示

实现方法：通过 setShortcuts()函数实现，如：

```
runall->setShortcuts(QKeySequence::New);
```

运行效果：



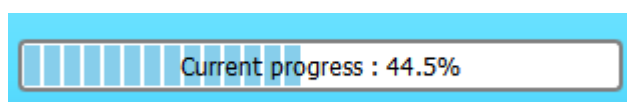
4) 进度显示功能

效果：有进度条实时显示运行的进度

实现方法：增添一个进度条控件，写在打印数据的函数中，当所有语句打印完毕，进度条正好显示到 100%（利用打印的语句的条数来事先计算好）

```
void MainWindow::runalloutput()
{
    count++;
    progress->setRange(0,13488);
    int val=progress->value();
    for(int i=1;i<=500;i++)
    {
        double dProgress = (progress->value() - progress->minimum()) * 100.0
            / (progress->maximum() - progress->minimum());
        progress->setFormat(tr("Current progress : %1%").arg(QString::number(dProgress, 'f', 1)));
        progress->setValue(val+i);
    }
}
```

运行效果：



5) 显示时间的功能


效果：在主窗口左下方实时更新当前的北京时间

实现方法：利用 Qt 中的 QTime 类获取当前时间，写在 QLabel 上显示即可

```
void MainWindow::sl_time()
{
    QTime time=QTime::currentTime();
    QString txtTime=time.toString("hh:mm:ss");

    QDate date=QDate::currentDate();
    QString txtDate=date.toString("yy-MM-dd");
    txtDate = "20"+txtDate;
    timelabel->setText(txtTime);
    datelabel->setText(txtDate);
}
```

运行效果：



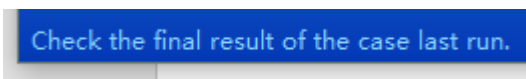
6) 状态条

效果：将鼠标移动至任意按键上时，屏幕最下方会显示简要说明（比查看说明书更方便）

实现方法：添加一个 statusBar()器件，然后其他器件均通过 setStatusTip()写上说明即可

```
clean->setStatusTip(tr("clear the table."))
```

运行效果：



7) 了解作者的功能

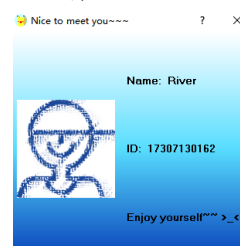
效果：可以打开一个写有作者相关信息的窗口，登陆的用户名和密码就在这里：)

实现方法：写一个 dialog，在上面写上作者信息即可

```
void MainWindow::Author()
{
    QDialog *dialog = new QDialog(this);
    dialog->setMinimumSize(300,300);
    dialog->setAttribute(Qt::WA_DeleteOnClose);
    dialog->setWindowTitle(tr("Nice to meet you~~~"));
    dialog->setStatusTip(tr("River is happy to see you.~"));

    QLabel *name=new QLabel;
    name->setText(tr("Name: River"));
    QLabel *note= new QLabel;
    note->setText(tr("ID: 17307130162"));
    QLabel *picture=new QLabel;
    picture->setBaseSize(30,30);
    picture->setPixmap(QPixmap(":/image/me"));
    QLabel *words=new QLabel;
    words->setText(tr("Enjoy yourself~~~>_<"));
}
```

运行效果：



3. 遇到的难题及其解决方案

①.exe 文件的地址问题

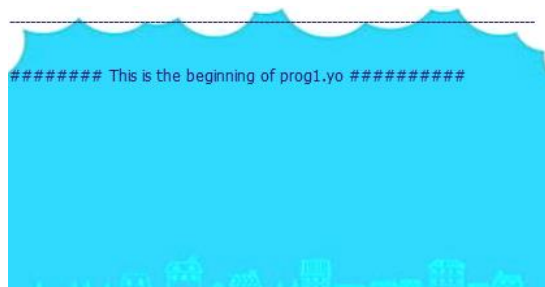
(1)描述：由于是利用 cmd 间接运行 Y86-64.exe 这一原始代码文件来实现调用，因此必须要知道.exe 的地址。若是输入绝对地址，的确能准确找到该文件，但是考虑到绝对地址是很容易变化的（比如说拷贝到 TA 的电脑上就运行不了了），因此需要利用相对地址来运行程序。但是命令行的位置和 Qt 项目的位置不同，所以相对于 Y86-64.exe 原始代码的位置也不同。这就使找地址变得困难了。

(2)解决方案：通过查找资料，发现 Qt 有一个定位函数 QDir::currentPath()，它能直接返回本 Qt 项目或 Qt.exe 的绝对位置，这样一来，再利用它与 Y86-64.exe 这一原始代码文件的相对位置即能准确定位后者的绝对位置了。

```
get_string2(QDir::currentPath());
cmd->write("cd ");
cmd->write(_string2);cmd->write("\n");
cmd->write("cd ../../Y86-64_VS/River's_Y86-64/Release\n");
cmd->write(" .\\River's_Y86-64.exe\n");
```

②打印出的文件不完整的问题

(1)描述：当把程序的运行速度等级调低后，会出现 TextBrowser 上打印出的文件不完整的 Bug，如下图所示：



通过分析，我认为这是由于程序的运行速度慢到一定程度时，会慢于数据打印的速度，即程序还没运行完，TextBrowser 就把整个文件打印完了

(2)解决方案：增加一个延时函数，延时 1 秒后再开始打印

```
QDateTime curDateTime = QDateTime::currentDateTime();//延时1秒
QDateTime nowDateTime;
do
{
    nowDateTime = QDateTime::currentDateTime();
    QApplication->processEvents();
}while(curDateTime.secsTo(nowDateTime) <= 1);
```

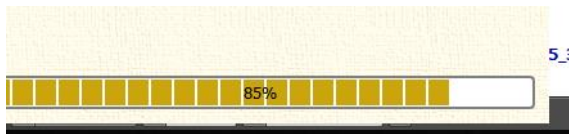
```
*****Circle 10*****
ZF=0 SF=0 OF=0 Stat=AOK
F : Fetch stage is doing nothing.      State=HLT
D : Decode stage is doing nothing.     State=HLT
E : Execute stage is doing nothing.    State=HLT
M : Memory stage is doing nothing.    State=HLT
W: Instr=halt    valE=0x0 valM=0x0 dstE=---- dstM=---- State=HLT
    Execute: ALU: -----
    Memory:  -----
    Writeback: -----
*****
##### The Y86 is stopped. Stat=HLT #####
```

③找不到在启动界面上加进度条的方法

(1)描述：为了不让启动界面过于单调，希望添加一个进度条在启动界面上。在网上搜索的方法都是自行构造一个包含启动界面和进度条的类，再把整个类显示出来，然而它的代码又长又复杂。

(2)解决方案：在纠结了很久后，偶然发现启动界面 SplashScreen 拥有一个 setLayout()的函数。因此突发奇想，类比在窗口 Dialog 上添加进度条的方法，先将进度条放在一个 Layout 中，再 SplashScreen->setLayout(Layout);即可

```
QVBoxLayout *vlay=new QVBoxLayout;  
vlay->addStretch();  
vlay->addWidget(p);  
SplashScreen->setLayout(vlay);
```



4. 本程序的特色

①功能丰富

本程序拥有运行、调试、设置、登陆、帮助 5 大功能，另外还有 7 个小功能控件，功能十分丰富

②创新性高

>>运行功能上：自定义输入运行是创新之处。这一功能是为了不让该 Y86-64 模拟器仅局限于那几个测例上，而应该像一个真的 CPU 一样运行各种可能的指令。

>>调试功能上：在该程序中，设置的断点不是一般的断点，而是不会停止运行的断点。因此可以一次性在所有想查看的位置都写上一行“break”标识符，这样一来就能只跑一遍程序而同时看到多个断点处的运行状态。这样大幅提高了调试效率。

>>登陆/注销功能：这是一个创新的功能，深度开发这一功能的话能提高该 CPU 的安全性，而不容易被非法利用。

>>帮助功能：扫二维码获取快速帮助是一个创新点。考虑到帮助的显示和程序的运行共用一个 TextBrowser，这样不利于在运行程序是快速查看帮助，因此有了这个二维码的设计。

③人机交互友好度（易用、整洁、合理）

>>扫码功能：使用户能边看说明，边运行程序了

>>工具栏：在熟悉了本程序的功能后，按工具栏能方便不少，使得本程序变得易用；并且工具栏还使得程序界面变得整洁

>>快捷键：使用频繁的功能运用快捷键能大幅提高工作效率，这样的设计使本程序变得便捷易用

>>进度条：可以缓解用户等待程序运行时的无聊和郁闷

>>时间：方便用户快速了解时间（再也不用担心不知道今天几号了）

>>状态条：能比查看说明书更快速的让用户了解每一个按钮的大致用途

>>了解作者：让用户能够有机会了解作者的信息，既能满足用户的好奇心，又能让用户和作者交上朋友，大幅提升了交互的友好度

④鲁棒性强

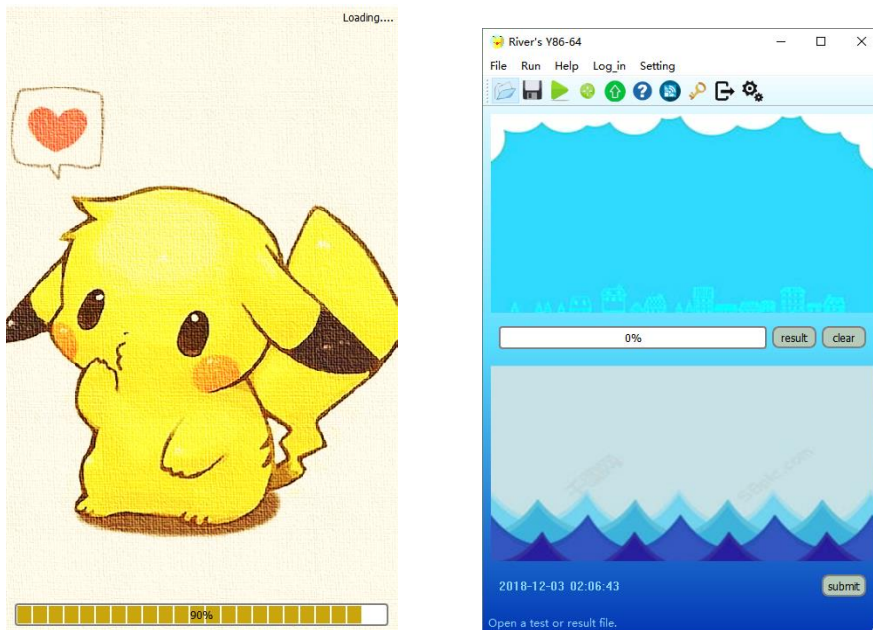
>>**clear 按钮**：当出现由于不当操作造成的界面混乱时，此按钮可以快速复原

⑤风格清新，小巧便捷，灵活多变

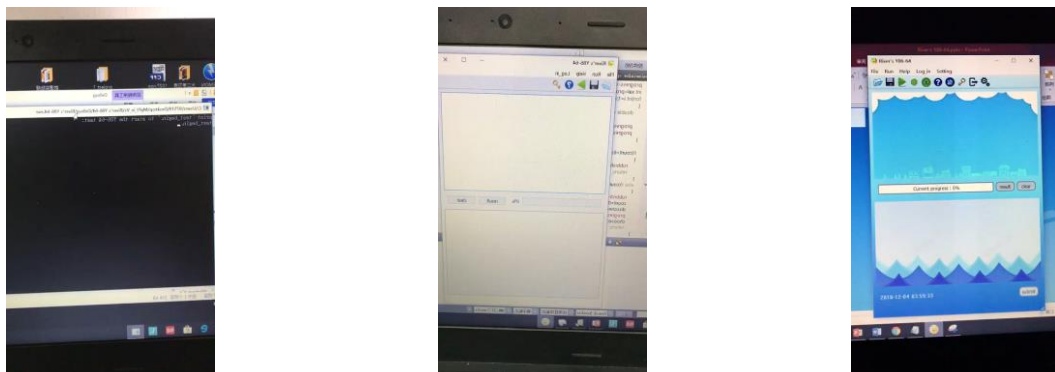
本程序以天蓝和海蓝色调为主，配以**皮卡丘**的主图标以及皮卡丘的启动界面，风格活泼可爱，清新脱俗

三、 实验结果

本程序能正确通过所有 21 个测例，且 GUI 版性能稳定，功能众多。符合实验要求。



四、 心路历程及实验总结



本次 Y86-64PJ 的阶段一、二到这里就结束了。从一开始啥都没有，到能够运行所有测例，再到把所有的 bug 调试完（能够运行出正确结果），这是阶段一的全部；然后再从黑底白字开始，到出现一个像样的标准界面，再到一个色彩丰富，精细完美的漂亮界面，从工具栏上两三个图标，到现在的几乎塞满了图标，这是阶段二的全部。虽然 debug 很痛苦，很费时，但当我把一个又一个充满趣味的功能填进我的工具栏内时，我心里是充满快感的；当我看着这样一个像宝贝一样的第一个有 GUI 的程序时，我觉得值了。由游戏得来的灵感，也让我全程像玩游戏一样沉迷。