计算机体系结构

实验报告

（2022学年秋季学期）

教学班级	计科二班	专业（方向）	计算机科学与技术
学号	20337263	姓名	俞泽斌

一、实验环境

visual studio2019

C++环境 ISO C++14

C环境 旧MSVC

windows10

二、实验原理

本次实验是关于tomasulo算法的实现，主要涉及的是以下几个方面

ISSUE：从挂起的指令队列中获取指令。

向空闲保留站发出指令。
所选RS标记为忙。
控制将可用的指令操作数值发送到分配的RS。
尚未可用的操作数被重命名为将产生该操作数的RSs（寄存器重命名）。

Execution:对操作数进行操作。

当两个操作数都准备好后，就开始在分配的FU上执行。
如果所有操作数都没有准备好，等待并观察公共数据总线以获得所需的结果。

Write result:完成执行。

将公共数据总线上的结果写入所有等待单元
将保留站标记为可用。

三、实验过程

由实验原理可知，所要实现的即为上述几个步骤，先开始进行数据的预处理

```
vector<vector<string>> spilt_input(int a)
{
    vector<vector<string>> spilt_instr;
    ifstream infile;    //流处理输入
    if (a == 1) {
        infile.open("input1.txt", ios::in);
    }
    else {
```

```
        infile.open("input2.txt", ios::in);
    }
    string line;
    string sing;
    stringstream ss;
    int index1 = 0;
    while (getline(infile, line)) {   //先分行
        spilt_instr.push_back(vector<string>(4));
        ss << line;
        int index2 = 0;
        while (getline(ss, sing, ' ')) {
            spilt_instr[index1][index2] = sing;  //分列
            index2++;
        }
        ss.clear();
        index1++;
    }
    infile.close();//关闭输入流，防止与输出冲突
```

数据预处理方面，采用的是流的文件输入，并且通过vector对数据进行了行和列的分开，相当于采用二维数组存储。

```
void file_print(int a) {//流输出
    if (a == 1) {
        outfile.open("output1.txt", ios::out);
    }
    else {
        outfile.open("output2.txt", ios::out);
    }
}
```

输出到文件采用了简单的流输出

```
void issue(int index)
{
    int now_cycle = cycle;
    cout << "issue" << index << " " << now_cycle << endl;
    instr_cycle[index][0] = now_cycle;
    int RS_No = 0;
    if (instr[index][0] == "ADDD" || instr[index][0] == "SUBD") {
        for (RS_No = 0; RS_No < 3; RS_No++) {   //寻找一个空闲的加法器
            if (Add[RS_No].busy == false) {
                break;
            }
        }
        Add[RS_No].busy = true;//将此时选中的加法器改成busy状态
        Add[RS_No].op = instr[index][0];
        Register_condition[Register_index(instr[index][1])].busy = true;
        Register_condition[Register_index(instr[index][1])].status = "Add" +
to_string(RS_No + 1);//以下开始更新每个RS
        if (Register_condition[Register_index(instr[index][2])].busy) {
            Add[RS_No].vj = "";
            Add[RS_No].qj = Register_condition[Register_index(instr[index]
[2])].status;
```

```
        }
        else {
            Add[RS_No].vj = Register_condition[Register_index(instr[index]
[2])].value;
            Add[RS_No].qj = "";
        }
        if (Register_condition[Register_index(instr[index][3])].busy) {
            Add[RS_No].vk = "";
            Add[RS_No].qk = Register_condition[Register_index(instr[index]
[3])].status;
        }
        else {
            Add[RS_No].vk = Register_condition[Register_index(instr[index]
[3])].value;
            Add[RS_No].qk = "";
        }
    }
    if (instr[index][0] == "MULTD" || instr[index][0] == "DIVD") {
        for (RS_No = 0; RS_No < 2; RS_No++) {
            if (Mult[RS_No].busy == false) {  //寻找一个空闲的乘法器
                break;
            }
        }
        Mult[RS_No].busy = true;
        Mult[RS_No].op = instr[index][0];
        //更新寄存器状态
        Register_condition[Register_index(instr[index][1])].busy = true;
        Register_condition[Register_index(instr[index][1])].status = "Mult" +
to_string(RS_No + 1);
        if (Register_condition[Register_index(instr[index][2])].busy) {
            Mult[RS_No].vj = "";
            Mult[RS_No].qj = Register_condition[Register_index(instr[index]
[2])].status;
        }
        else {
            Mult[RS_No].vj = Register_condition[Register_index(instr[index]
[2])].value;
            Mult[RS_No].qj = "";
        }
        if (Register_condition[Register_index(instr[index][3])].busy) {
            Mult[RS_No].vk = "";
            Mult[RS_No].qk = Register_condition[Register_index(instr[index]
[3])].status;
        }
        else {
            Mult[RS_No].vk = Register_condition[Register_index(instr[index]
[3])].value;
            Mult[RS_No].qk = "";
        }
    }
    if (instr[index][0] == "LD") {
        for (RS_No = 0; RS_No < 3; RS_No++) {
            if (Load[RS_No].busy == false) {//寻找一个空闲的load
                break;
            }
```

```
        }
        Load[RS_No].busy = true;
        if (instr[index][2] == "0") {
            Load[RS_No].address = "M(" + instr[index][3] + ")";//判断是否有偏移量，
方便之后直接输出位置
        }
        else {
            Load[RS_No].address = "M(" + instr[index][2] + instr[index][3] +
")";
        }
        //更新寄存器状态
        Register_condition[Register_index(instr[index][1])].status = "Load" +
to_string(RS_No + 1);
        Register_condition[Register_index(instr[index][1])].busy = true;
    }
    if (instr[index][0] == "SD") {
        for (RS_No = 0; RS_No < 3; RS_No++) {
            if (Store[RS_No].busy == false) {//寻找一个空闲的store
                break;
            }
        }
        Store[RS_No].busy = true;
        if (instr[index][1] == "0") {
            Store[RS_No].address = "M(" + instr[index][3] + ")";
        }
        else {
            Store[RS_No].address = "M(" + instr[index][2] + instr[index][3] +
")";
        }
        if (Register_condition[Register_index(instr[index][1])].busy) {
            Store[RS_No].vj = "";
            Store[RS_No].qj = Register_condition[Register_index(instr[index]
[1])].status;
        }
        else {
            Store[RS_No].vj = Register_condition[Register_index(instr[index]
[1])].value;
            Store[RS_No].qj = "";
        }
    }

    thread t = thread(execute, index, RS_No);
    t.join();
}
```

issue阶段，主要的操作是对于每一条来的指令，首先找到对应的op，去RS中寻找有没有空闲的对应的加法器乘法器等，如果有空闲的就把空闲的占有（标记为busy状态），如果没有就进行等待。然后对于不同的指令，找到对应的位置后判断这条指令的后面的参数寄存器是否有空，如果没空就把对应的寄存器的状态置入此时占有的RS中的qj中，如果有空就将对应的值放入占有的RS中的Vj中，第二个操作寄存器就也进行上述操作只是改成Qk和Vk，而对于Store和Load因为只有一个操作寄存器地址，只需要qj和vj即可，加入address表示计算后的地址

```
void execute(int index, int RS_No)
{
```

```
    int now_cycle = cycle;
    Sleep(10);
    now_cycle = cycle;
    //execution latency
    if (instr[index][0] == "LD") {
        while (cycle != now_cycle + 2);
    }
    if (instr[index][0] == "SD") {
        while (Store[RS_No].qj != "");//先等qj为空
        now_cycle = cycle;
        while (cycle != now_cycle + 2);
    }
    if (instr[index][0] == "ADDD" || instr[index][0] == "SUBD") {
        while (Add[RS_No].qj != "" || Add[RS_No].qk != "");//先等qj和qk均为空
        now_cycle = cycle;
        while (cycle != now_cycle + 2);
    }
    if (instr[index][0] == "MULTD" || instr[index][0] == "DIVD") {
        while (Mult[RS_No].qj != "" || Mult[RS_No].qk != "");//等qj和qk均为空
        now_cycle = cycle;
        if (instr[index][0] == "MULTD") {
            while (cycle != now_cycle + 10);
        }
        else {
            while (cycle != now_cycle + 20);
        }
    }

    cout << "execute" << index << " " << cycle << endl;
    instr_cycle[index][1] = cycle;
    thread t = thread(write, index, RS_No);
    t.join();
}
```

execute阶段，上面说到了如果寄存器在issue阶段忙的时候就会把对应的状态信息存到qj或者qk中，所以在execute阶段，所需要进行的第一部分等待就是把所有的qj和qk全部等待成空值，第二部分就是execution latency 的等待，也就是pdf中所给出的那张表格，关于每个操作需要进行的延迟

```
void write(int index, int RS_No)
{
    int now_cycle = cycle;
    string t_address;
    string t_status;
    while (cycle != now_cycle + 1);
    Sleep(20);
    cout << "write" << index << " " << cycle << endl;
    instr_cycle[index][2] = cycle;   //最终每条指令输出的write周期
    while (cycle != now_cycle + 1);  //等待一周期
    if (instr[index][0] == "LD") {
        t_status = "Load" + to_string(RS_No + 1);
        t_address = Load[RS_No].address;
        for (int i = 0; i < 16; i++) {
            if (Register_condition[i].busy && Register_condition[i].status ==
t_status) {  //以寄存器状态作为判断条件。然后添加寄存器的值
                Register_condition[i].busy = false;
```

```
                Register_condition[i].status = "";
                Register_condition[i].value = t_address;
            }
        }
        Load[RS_No].address = "";
        Load[RS_No].busy = false;
    }
    if (instr[index][0] == "SD") {
        t_status = "Store" + to_string(RS_No + 1);//SD指令不需要改变寄存器
        Store[RS_No].address = "";
        Store[RS_No].busy = false;
        Store[RS_No].vj = "";
    }
    if (instr[index][0] == "ADDD" || instr[index][0] == "SUBD") {
        t_status = "Add" + to_string(RS_No + 1);
        if (instr[index][0] == "ADDD") { //加法or减法
            t_address = Add[RS_No].vj + "+" + Add[RS_No].vk;
        }
        else {
            t_address = Add[RS_No].vj + "-" + Add[RS_No].vk;
        }
        for (int i = 0; i < 16; i++) {//以寄存器状态作为判断条件。然后添加寄存器的值
            if (Register_condition[i].busy && Register_condition[i].status ==
t_status) {
                Register_condition[i].busy = false;
                Register_condition[i].status = "";
                Register_condition[i].value = t_address;
            }
        }
        Add[RS_No].busy = false;  //重新将寄存器的值归0，表明空闲
        Add[RS_No].op = "";
        Add[RS_No].vj = "";
        Add[RS_No].vk = "";
    }
    if (instr[index][0] == "MULTD" || instr[index][0] == "DIVD") {
        t_status = "Mult" + to_string(RS_No + 1);
        if (instr[index][0] == "MULTD") {  //乘法 or 除法
            t_address = Mult[RS_No].vj + "*" + Mult[RS_No].vk;
        }
        else {
            t_address = Mult[RS_No].vj + "/" + Mult[RS_No].vk;
        }
        for (int i = 0; i < 16; i++) {
            if (Register_condition[i].busy && Register_condition[i].status ==
t_status) {
                Register_condition[i].busy = false;
                Register_condition[i].status = "";
                Register_condition[i].value = t_address;
            }
        }
        Mult[RS_No].busy = false;
        Mult[RS_No].op = "";
        Mult[RS_No].vj = "";
        Mult[RS_No].vk = "";
    }
```

```
    for (int i = 0; i < 3; i++) {
        if (Add[i].qj == t_status) {    //更新加法器的状态
            Add[i].qj = "";
            Add[i].vj = t_address;
        }
        if (Add[i].qk == t_status) {
            Add[i].qk = "";
            Add[i].vk = t_address;
        }
    }
    for (int i = 0; i < 2; i++) {//更新乘法器的状态
        if (Mult[i].qj == t_status) {
            Mult[i].qj = "";
            Mult[i].vj = t_address;
        }
        if (Mult[i].qk == t_status) {
            Mult[i].qk = "";
            Mult[i].vk = t_address;
        }
    }
    for (int i = 0; i < 3; i++) {
        if (Store[i].qj == t_status) {//更新store的状态
            Store[i].qj = "";
            Store[i].vj = t_address;
        }
    }
}
```

write阶段，这个涉及了两方面的更新，一个部分是关于寄存器状态的更新，涉及指令的区分，因为SD指令是不涉及对寄存器的操作的，对于其他指令，计算好的值会被放入指定的寄存器中，并将此时的寄存器改为空闲状态。另一个部分是关于RS状态的更新，主要是将qj为此时write back后的数据 的RSqj置为0，vj置为计算后的值，即此时的RS等到了需要的数据

```
void tomasulo()
{
    vector<thread> threads(instr_size);
    file_print(filenum);
    int precycle = 0;
    int cycle_flag[100];
    memset(cycle_flag, 0, sizeof(cycle_flag));  //防止重复输出同一个周期
    for (cycle; cycle <= instr_size; cycle++) {
        threads[cycle - 1] = thread(instr_wait, cycle - 1);  //创建线程，并行执行
        Sleep(80);
        for (int i = 0; i < instr_cycle.size(); i++) {   //取巧做法，直接用指令的每个
阶段的周期来表示发生改变的位置，输出那几个周期的内容
            for (int j = 0; j < instr_cycle[i].size(); j++) {
                if (cycle == instr_cycle[i][j]&&cycle_flag[cycle]==0) {
                    cycle_flag[cycle] = 1;
                    if (cycle - precycle == 1) {
                        if (cycle != 1) {       //没有持续的相同输出
                            cout << "no continued same output " << endl;
                            outfile << "no continued same output " << endl;
                            cout << endl;
                            outfile << endl;
                        }
```

```
                        cout << "cycle_" << cycle << ";" << endl;
                        outfile << "cycle_" << cycle << ";" << endl;
                        print();
                        precycle = cycle;
                    }
                    else {  //有持续的相同周期并且输出最后一个相同周期编号至末尾
                        cout << "same output until cycle_" << cycle - 1 << endl;
                        outfile << "same output until cycle_" << cycle - 1 <<
endl;
                        cout << endl;
                        outfile << endl;
                        cout << "cycle_" << cycle << ";" << endl;
                        outfile << "cycle_" << cycle << ";" << endl;
                        print();
                        precycle = cycle;
                    }
                }
            }
        }
    }
    //print();
    Sleep(20);

    while (cycle <= 60) {
        Sleep(80);   //没有指令需要读入，只用考虑输出情况，下面代码和上半部分基本一致
        for (int i = 0; i < instr_cycle.size(); i++) {
            for (int j = 0; j < instr_cycle[i].size(); j++) {
                if (cycle == instr_cycle[i][j] && cycle_flag[cycle] == 0) {
                    cycle_flag[cycle] = 1;
                    if (cycle - precycle == 1) {
                        cout << "no continued same output " << endl;
                        outfile << "no continued same output " << endl;
                        cout << endl;
                        outfile << endl;
                        cout << "cycle_" << cycle << ";" << endl;
                        outfile << "cycle_" << cycle << ";" << endl;
                        print();
                        precycle = cycle;
                    }
                    else {
                        cout << "same output until cycle_" << cycle - 1 << endl;
                        outfile << "same output until cycle_" << cycle - 1 <<
endl;
                        cout << endl;
                        outfile << endl;
                        cout << "cycle_" << cycle << ";" << endl;
                        outfile << "cycle_" << cycle << ";" << endl;
                        print();
                        precycle = cycle;
                    }
                }
            }
        }
        Sleep(20);
        cycle++;
```

```
    }
    for (auto& th : threads) {
        th.join();
    }

    for (int i = 0; i < instr_size; i++) {  //输出最后的各个指令各个阶段的周期编号（指令最终执行情况表）
        for (int j = 0; j < 3; j++) {
            cout << instr[i][j] << " ";
            outfile << instr[i][j] << " ";
        }
        cout << ":";
        outfile << ":";
        for (int j = 0; j < 3; j++) {
            cout << instr_cycle[i][j] << " ";
            outfile << instr_cycle[i][j] << " ";
        }
        cout << endl;
        outfile << endl;
    }
    outfile.close();//关闭输出流
}
```

主函数部分，主函数的主要作用其实可能就是来创建线程了，对于每一个到达的指令，都创建一个线程来表示他们可以并行执行，但是每一个部分又对指令进行了有关约束和等待来保持一致性，之后是关于哪个周期该输出，这里做了个取巧，因为按照pdf上的说法，重复的周期只需要输出头和尾的编号即可，而有变化的阶段一定是所有指令的issue，execute，write阶段才会出现，所以这里采用一个precycle变量来存储上一次发生改变的周期，只输出所有指令的issue，execute，write阶段周期的各个RS，FU等的状态，其余皆为重复，用此时的precycle~cycle-1来代表重复周期数。最后将所有的指令以及对应的issue，execute，write阶段周期输出。

四、实验结果

实验结果采用文件输出和终端输出两个输出方式

input1的部分：

Microsoft Visual Studio 调试控制台

```
issue0 1
cycle_1;
Load1:Yes,M(34+R2);
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:No,,,,,;
Mult2:No,,,,,;
F0:;F2:;F4:;F6:Load1;F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
issue1 2
no continued same output

cycle_2;
Load1:Yes,M(34+R2);
Load2:Yes,M(45+R3);
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:No,,,,,;
Mult2:No,,,,,;
F0:;F2:Load2;F4:;F6:Load1;F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
execute0 3
issue2 3
no continued same output
```

第一个周期和第二个周期，主要涉及的操作是load操作，可以看到此时的RS的输出中load1在第一个周期被使用，load2在第二个周期被使用，并且在下面的FU中，F6也显示在load1，F2显示在load2状态

Microsoft Visual Studio 调试控制台

```
cycle_3;
Load1:Yes,M(34+R2);
Load2:Yes,M(45+R3);
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,MULTD,,,Load2,;
Mult2:No,,,,,;
F0:Mult1;F2:Load2;F4:;F6:Load1;F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
execute1 4
issue3 4
write0 4
no continued same output

cycle_4;
Load1:No,;
Load2:Yes,M(45+R3);
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:Yes,SUBD,M(34+R2),,,Load2;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,MULTD,,,Load2,;
Mult2:No,,,,,;
F0:Mult1;F2:Load2;F4:;F6:M(34+R2);F8:Add1;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
issue4 5
write1 5
no continued same output
```

接下来两个周期，是乘法和加法操作，同时也可以看到此时的RS进行了对应的更新，寄存器状态也进行了改变，并且F6由load1改为了M（34+R2）表明已经开始取数据

```
cycle_5;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:Yes,SUBD,M(34+R2),M(45+R3),,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,MULTD,M(45+R3),,,;
Mult2:Yes,DIVD,,M(34+R2),Mult1,;
F0:Mult1;F2:M(45+R3);F4:;F6:M(34+R2);F8:Add1;F10:Mult2;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
issue5 6
no continued same output

cycle_6;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:Yes,SUBD,M(34+R2),M(45+R3),,;
Add2:Yes,ADDD,,M(45+R3),Add1,;
Add3:No,,,,,;
Mult1:Yes,MULTD,M(45+R3),,,;
Mult2:Yes,DIVD,,M(34+R2),Mult1,;
F0:Mult1;F2:M(45+R3);F4:;F6:Add2;F8:Add1;F10:Mult2;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
execute3 7
no continued same output
```

再接下来两个周期，新的指令为DIVD F10 F0 F6 ， ADDD F6 F8 F2 ， 可以看到此时的ADD已经放入add2中了，divd也放入了mult2，同时寄存器状态也进行了更新。

以上都是每个相邻周期都发生变化的情况的输出，可以看到最后一行都加上了no continued output的字样，接下来来一个有重复输出周期的例子

```
cycle_11;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,MULTD,M(45+R3),,,;
Mult2:Yes,DIVD,,M(34+R2),Mult1,;
F0:Mult1;F2:M(45+R3);F4:;F6:M(34+R2)-M(45+R3)+M(45+R3);F8:M(34+R2)-M(45+R3);F10:Mult2;F12:;F14:;F16:;F18:;F20:;F22:;F24:
;F26:;F28:;F30:;
execute2 15
same output until cycle_14

cycle_15;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,MULTD,M(45+R3),,,;
Mult2:Yes,DIVD,,M(34+R2),Mult1,;
F0:Mult1;F2:M(45+R3);F4:;F6:M(34+R2)-M(45+R3)+M(45+R3);F8:M(34+R2)-M(45+R3);F10:Mult2;F12:;F14:;F16:;F18:;F20:;F22:;F24:
;F26:;F28:;F30:;
write2 16
no continued same output
```

看第11周期的时候有重复输出，重复输出到14周期，最后输出了一行 same output until cycle_14

最后看一下具体的各个指令的三个状态的发生周期

```
cycle_37;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:No,,,,,;
Mult2:No,,,,,;
F0:M(45+R3)*;F2:M(45+R3);F4:;F6:M(34+R2)-M(45+R3)+M(45+R3);F8:M(34+R2)-M(45+R3);F10:M(45+R3)*/M(34+R2);F12:;F14:;F16:;F1
8:;F20:;F22:;F24:;F26:;F28:;F30:;
LD F6 34+ R2 :1 3 4
LD F2 45+ R3 :2 4 5
MULTD F0 F2 F4 :3 15 16
SUBD F8 F6 F2 :4 7 8
DIVD F10 F0 F6 :5 36 37
ADDD F6 F8 F2 :6 10 11
```

可以看到符合pdf上的数据，完成本次实验

input2的部分：

首先对于代码中第一部分，改变文件编号

```
const int filenum = 2;  //输入文件编号
```

然后这个部分就不做详细阐述了，看一下几个周期的截图吧

```
cycle_6;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:Yes,M(0R3);
Store2:No,;
Store3:No,;
Add1:Yes,ADDD,M(R3),M(R2),,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,DIVD,M(R3),M(R2),,;
Mult2:Yes,MULTD,,M(R2),Mult1,;
F0:Add1;F2:M(R2);F4:M(R3);F6:Mult2;F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
execute4 7
no continued same output

cycle_7;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:Yes,M(0R3);
Store2:No,;
Store3:No,;
Add1:Yes,ADDD,M(R3),M(R2),,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:Yes,DIVD,M(R3),M(R2),,;
Mult2:Yes,MULTD,,M(R2),Mult1,;
F0:Add1;F2:M(R2);F4:M(R3);F6:Mult2;F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
write4 8
no continued same output
```

因为有SD操作是新的，所以截取了第六周期的图片，此时store1中存在数据

```
cycle_41;
Load1:No,;
Load2:No,;
Load3:No,;
Store1:No,;
Store2:No,;
Store3:No,;
Add1:No,,,,,;
Add2:No,,,,,;
Add3:No,,,,,;
Mult1:No,,,,,;
Mult2:No,,,,,;
F0:M(R3)+M(R2);F2:M(R2);F4:M(R3);F6:M(R3)+M(R2)*M(R2);F8:;F10:;F12:;F14:;F16:;F18:;F20:;F22:;F24:;F26:;F28:;F30:;
LD F2 0 R2 :1 3 4
LD F4 0 R3 :2 4 5
DIVD F0 F4 F2 :3 25 26
MULTD F6 F0 F2 :4 36 37
ADDD F0 F4 F2 :5 7 8
SD F6 0 R3 :6 39 40
MULTD F6 F0 F2 :27 37 38
SD F6 0 R1 :28 40 41
```

最后的输出（各个指令的三个状态的发生周期）

基本符合tomasulo算法的模拟，本次实验圆满结束，随实验报告附上源码以及output1,2文件