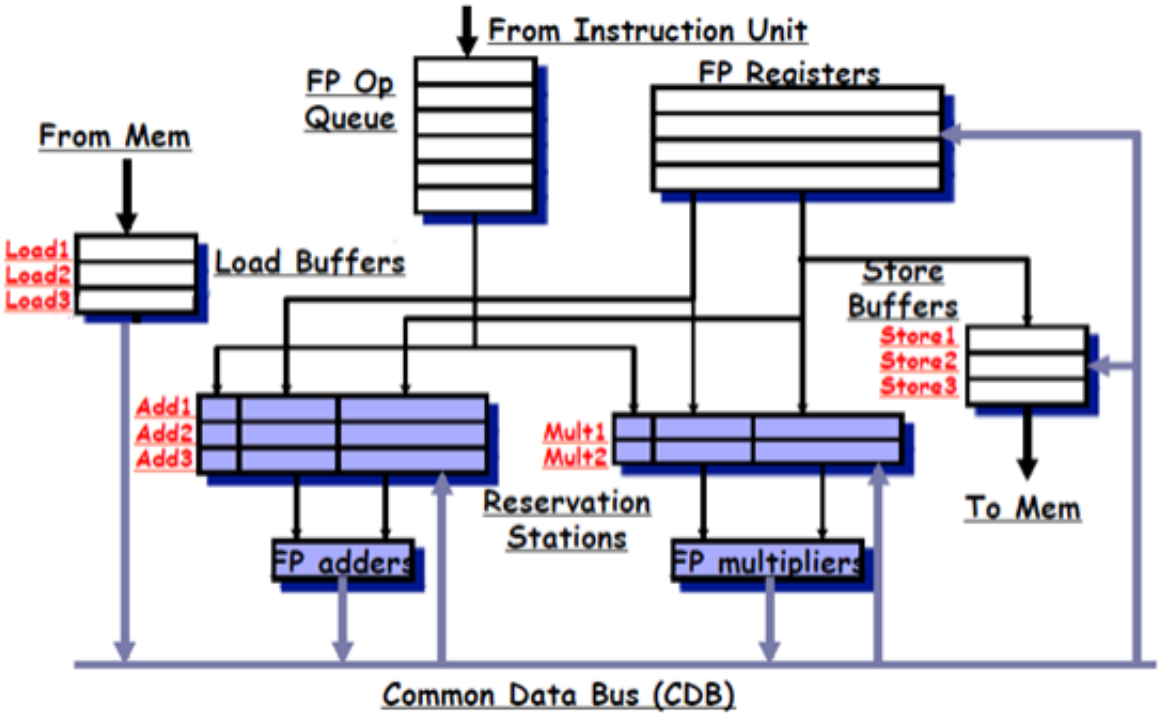
TOMASULO 调度算法

计05 许欣然 2010011358
计05 莫 愁 2010011359
计05 李百恩 2010080054

实验原理

Tomasulo 算法以硬件方式实现了寄存器重命名，允许指令乱序执行，这是提高流水线的吞吐率和效率的一种有效方式。该算法首先出现在 IBM360/91 处理机的浮点处理部件中，后广泛应用于现代处理器设计中。

假设浮点处理部件结构如下图所示。浮点处理部件从取指单元接收指令，存入浮点操作队列。浮点操作队列每拍最多发射1条指令给浮点加法器或浮点乘除法器。 浮点处理部件包含一个浮点加法器和一个浮点乘除法器。浮点加法器为两段流水线，输入端有三个保留站A1、A2、A3，浮点乘除法器为六段流水线，输入端有两个保留站M1，M2。当任意一个保留站中的两个源操作数到齐后，如果对应的操作部件空闲，可以把两个操作数立即送到浮点操作部件中执行。Load Buffer和Store Buffer各缓存三条访存操作。



实验要求

设计实现 Tomasulo 算法模拟器，要求：

- Tomasulo 算法模拟器能够执行浮点加、减、乘、除运算及 LOAD 和 STORE 操作。为简化起见，我们在下表中给出了各种操作的具体描述。

指令格式	指令说明	指令执行周期	保留站/缓冲队列项数
ADDD F1,F2,F3	F1,F2,F3为浮点寄存器，寄存器至少支持F0~F10	2个周期	3
SUBD F1,F2,F3	同上	2个周期	3
MULD F1,F2,F3	同上	10个周期	2
DIVD F1,F2,F3	同上	40个周期	2
LD F1,ADDR	F1为寄存器，ADDR 为地址，0<=ADDR<4096	2个周期	3
ST F1,ADDR	同上	2个周期	3

- 能够单步执行，实时显示算法的运行状况，包括各条指令的运行状态、各寄存器以及内存的值、保留站（Reservation Stations）状态、Load Buffer 和 Store Buffer 缓存的值等。
- 程序执行完毕后，能够显示指令执行周期数等信息。
- 为了简化设计，建议模拟器提供编辑内存值功能，以便实现数据输入；浮点除法可不做除0判断。
- 能够以文本方式输入指令序列。

实现内容

- 实现了 Tomasulo 算法模拟器，能够运行要求的指令操作，且有很大的定制性。
- 能够单步运行，依据要求实时的显示算法的运行情况。
- 显示执行周期数、当前时间等信息
- 提供编辑内存值的功能
- 允许以文本方式输入指令序列

实验环境

- 平台：全平台
- 语言：Javascript
- 软件需求：Chrome浏览器 version 26
- 库需求：Jquery、JqueryUI（基于Javascript，已包含在thridJS文件夹内）

实验设计思路

后端

与计算机硬件的模块化类似，后端采取了类似的模块化设计。 将所要实现的系统的每个功能部件都抽象成一个模块，再将这些功能部件组成一个执行 Tomasulo 调度算法的流水线系统。

由于采用了松耦合模块化设计，基于JS的对象机制，我们允许系统的多样性，通过system变量的配置可以实现 不同结构的 Tomasulo 调度算法的模拟。目前允许配置的内容有（位于 `index.html` 的 `init(program)` 函数中）：

- 寄存器类型（以前缀区分，根据课程只写了浮点寄存器）
- 每类寄存器的个数
- （运算器、保留站）类型与个数
- 指令集的运算内容、对应的保留站种类、运算时间、参数个数及结构
- 系统内存大小及Buffer的写入策略

以下是配置的简略说明：

```
var System = {};
System.memory = new Memory(4096);                //设置内存大小
System.registerFile = new RegisterFile(11, 'F');         //设置11个浮点寄存器
System.commonDataBus = new CommonDataBus();                        //初始化数据通路
System.reservationStations = {                                              //初始化保留站
        ADD_1: new ReservationStation('ADD_1'),
        ...
        STORE_1: new Buffer('STORE_1', System.memory),
};
System.instructionTypes = {              //初始化指令集：名字，运算时间，目标地址，参数种类，运算内
容，对应保留站
        'ADDD': new InstructionType('ADDD', 2, 0,
                        [InstructionType.PARAMETER_TYPE_REGISTER,
                         InstructionType.PARAMETER_TYPE_REGISTER,
                         InstructionType.PARAMETER_TYPE_REGISTER],
                        function(p) { return p[1] + p[2]; },
                        [System.reservationStations['ADD_1'],
                         System.reservationStations['ADD_2'],
                         System.reservationStations['ADD_3']]),

...
        'ST': new InstructionType('ST', 2, 1,
                  [InstructionType.PARAMETER_TYPE_REGISTER,
                   InstructionType.PARAMETER_TYPE_ADDRESS],
                  function(p) { this.memory.store(p[1], p[0]); return p[0]; },
                  [System.reservationStations['STORE_1'],
                   System.reservationStations['STORE_2'],
                   System.reservationStations['STORE_3']])
};

return new Main(program, System);
```

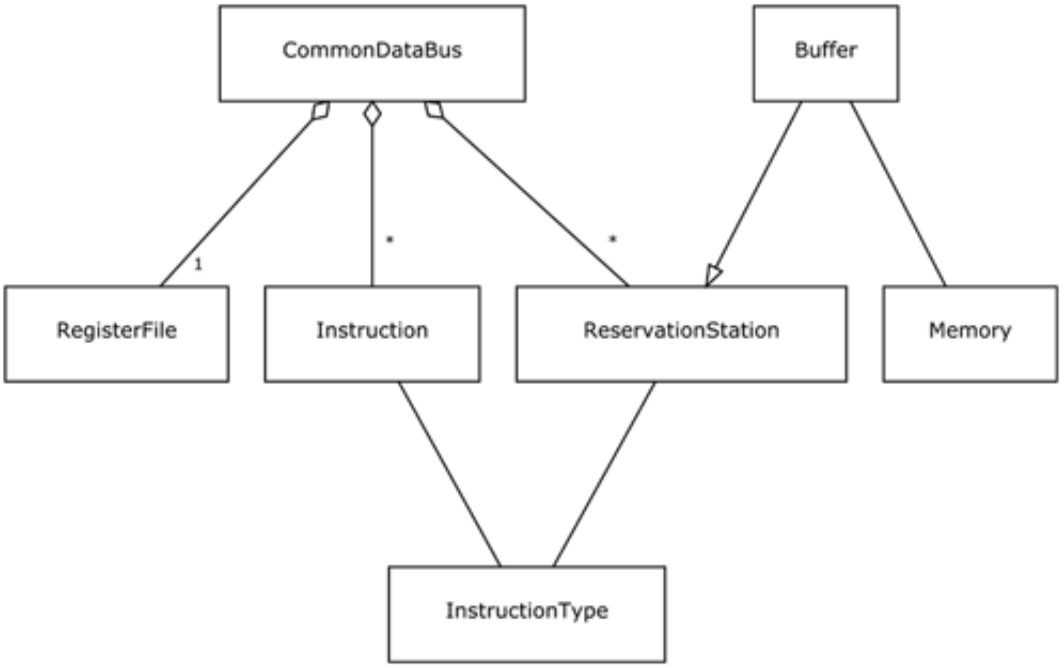
通过main.js中的 `step` 函数完成一部的计算，具体设计见后。

前端

根据前台 `System` 变量的设置，在网页上呈现对应的数据。同时利用 `step` 函数来完成一步、多步等控制。具体实现请参见 `index.html` 内部设置。

具体实现

根据 Tomasulo 算法的基本思想，我们设计了以下7个具有独立功能的模块，数据传递关系图如下：



InstructionType：指令类型

该模块相当于是指令类型的一个抽象类，包含了一条指令本身的性质，如：指令名称，执行时间，参数类型，结果类型，计算方法，以及该指令所使用的保留站类型。使用时，可以根据不同参数构造出不同指令，再在此基础上，解析每一具体指令，得到该指令类型，再根据构造该类型的指令基本信息执行之，从而完成Tomasulo调度算法的实现。

```
/* calculate: (args, memory) */
        function InstructionType(name, cycles, destParameter, parameters, calculate,
stations) {
                this.name = name;
                this.cycles = cycles;
                this.parameters = parameters;
                this.destParameter = destParameter;
                this.calculate = calculate;
                this.stations = stations;
        }
```

Instruction：指令模块

该模块完成了对每条具体的指令的一个抽象。包含有该条指令的基本状态信息，如：指令id（第几条指令），指令类型，指令是否执行完，还需要多少周期完成，指令所带参数等。

```
function Instruction(type, parameters) {
                this.id = ++id;
                this.type = type;
                this.time = type.cycles;
                this.parameters = parameters;
                this.issueTime = -1;
                this.executeTime = -1;
                this.writeBackTime = -1;
        }
```

ReservationStation：保留站模块

该模块是对保留站的一个抽象，记录了保留站的名称，状态，标识，流入该保留站的指令与参数等。需要特别说明的

是：与教材中介绍的不同，我们不是仅仅使用是否为busy来标志保留站状态的，而是按照指令执行的不同情况，为保留站的状态设置了以下4种可能的取
值：`STATE_IDLE`，`STATE_ISSUE`，`STATE_EXECUTE`和`STATE_WRITE_BACK`。除`IDLE`外的所有状态都属于教材中所定义的`busy = yes`的情况，这样做并不会改变算法本质，反而由于这一分类更为细致的状态标识而使得算法在具体实现时更为快捷、高效。

```
function ReservationStation(name) {
                this.name = name;
                this.state = ReservationStation.STATE_IDLE;
                this.parameters = null;
                this.tags = null;
                this.instruction = null;
        }
```

Buffer：load/store缓冲器模块

load缓冲器和store缓冲器中存放的是读/写memory的数据或地址。如教材中所述，load/store缓冲器的行为均与保留站类似，所以此处我们也不做特别区分。之所以特别提出这样一个模块，为的是方便日后若想添加额外功能或者需要与保留站做区分时的扩展。这里的`Buffer`直接套用了`ReservationStation`。

```
function Buffer(_, memory) {
                ReservationStation.apply(this, arguments);
                this.memory = memory;
                return this;
        }
```

CommonDataBus：公共数据总线模块

保留站和寄存器通过CommonDataBus，来注册未完成与已完成的结果，从而实现寄存器的换名与回填。

```
function CommonDataBus() {
                this._busy = {};
                this._result = {};
        }
```

包含有以下5个功能函数：

```
CommonDataBus.prototype.getBusy = function(type, name)
CommonDataBus.prototype.setBusy = function(type, name, instruction)
CommonDataBus.prototype.getResult = function(station)
CommonDataBus.prototype.setResult = function(station, value)
CommonDataBus.prototype.clearResult = function()
```

RegisterFile：寄存器组模块。

通过该模块，我们可以在系统初始化时指定寄存器组中寄存器的个数与名称。默认初始化所有寄存器的值都为1。

```
function RegisterFile(count, prefix) {
                this.registers = new Array(count);
                for (var i = 0; i < this.registers.length; ++i) {
                        this.registers[i] = 1;
                }
                this.count = count;
                this.prefix = prefix.toUpperCase();
        }
```

Memory：内存模块。

可以自定义内存大小。默认内存初始值为每一内存单元的编号。实际执行时可以在UI界面上自行设置。

```
function Memory(size) {
                this.data = new Array(size);
                this.size = size;
                for (var i = 0; i < this.data.length; ++i) {
                        this.data[i] = i;
                }
        }
```

main.js：封装成系统。

利用上述7个模块，将其封装成一个系统执行Tomasulo动态调度算法的流水线系统。可以在index.html中使用new Main(program, System) 实现一个这样的系统的实例，其中，program参数描述的是要执行的顺序指令集，System是包含了上述7个功能模块每种的个数与每个实际部件初始化参数的系统描述变量。

```
function Main(program, system) {
                Instruction.resetID();
                this.system = system;
                this.system.clock = 0;

                /* parse program */
                program = program.toUpperCase()
                                 .replace(/[\s,]+/g, ',')
                                 .replace(/^,|,$/g, '');

                var tokens = program.split(',');
                var instructions = [];

                for (var i = 0; i < tokens.length;) {
                        var instructionType = this.system.instructionTypes[tokens[i++]];
                        var params = [];
                        for (var j = 0; j < instructionType.parameters.length; ++j, ++i) {
                                switch (instructionType.parameters[j]) {
                                case InstructionType.PARAMETER_TYPE_REGISTER:
                                        params.push(tokens[i]);
                                        break;
                                case InstructionType.PARAMETER_TYPE_ADDRESS:
                                        params.push(parseInt(tokens[i], 10));
                                        break;
                                }
                        }
                        instructions.push(new Instruction(instructionType, params));
                }

                this.instructions = instructions;
                this.issuedInstructions = 0;
        }
```

该模块包含有两个函数接口：

step()

若 step() 函数返回true，表示所有指令队列中的指令都执行完成。 这是在Tomasulo调度算法下，每个时钟周期流水线执行时所调用到的函数，共分为ISSUE，EXECUTE，WRITEBACK三个阶段。

ISSUE 阶段

如果指令队列非空，且队列头部指令所需求的保留站有空闲的，就将该条指令取出送入这个空闲的保留站中。 具体判断流程就是遍历检查所有符合指令类型的保留站（或load/store缓冲器），一旦发现有state为IDLE的保留站，就可以在这个时钟周期派送指令队列头部的指令了。

EXECUTE 阶段

遍历所有保留站，根据保留站的state参数区分其在该时钟周期所要进行的动作。如上面ReservationStation模块中的介绍，如果保留站状态是STATE_EXECUTE，说明该保留站中的指令尚未执行完成，那么就根据其所对应的InstructionType，将instruction的executeTime减1，表示模拟这一指令在这一时钟周期是在执行；如果executeTime减至0，说明这一时钟周期该指令刚好执行完成，那么在本周期内就将该保留站的state标记为STATE_WRITE_BACK，以便下一时钟周期即可写回。

WRITEBACK 阶段

如果保留站是STATE_WRITE_BACK状态，那么根据这条instruction的各项参数，利用commonDataBus的getBusy()

函数获得总线上完成计算的数据，将运行结果写入所有指向这一保留站的保留站或寄存器组或缓冲器。利用缓冲站内的行号来从软件的角度解决 WAW 的冲突问题。

结尾阶段

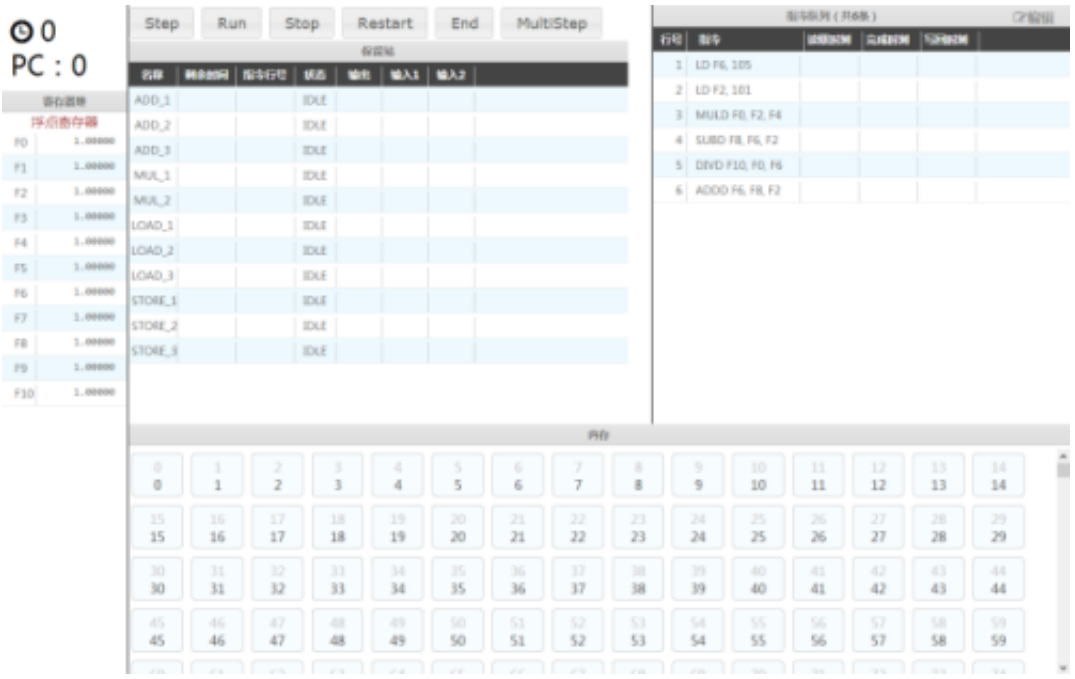
检查所有保留站，看是否有处于STATE_ISSUE的保留站所有参数都准备就绪，如果有的话就将其state置为STATE_EXECUTE，以便在下一时钟周期开始时即可执行之。

run()

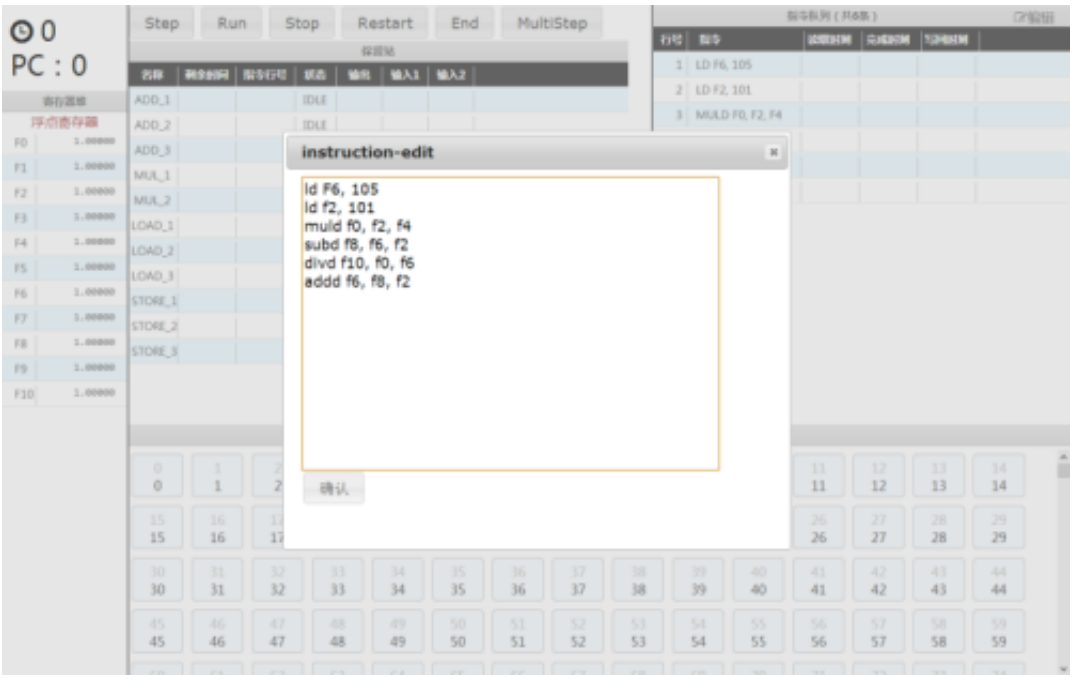
连续执行，直至step() 返回true，也就是执行直至指令队列为空。

实验结果

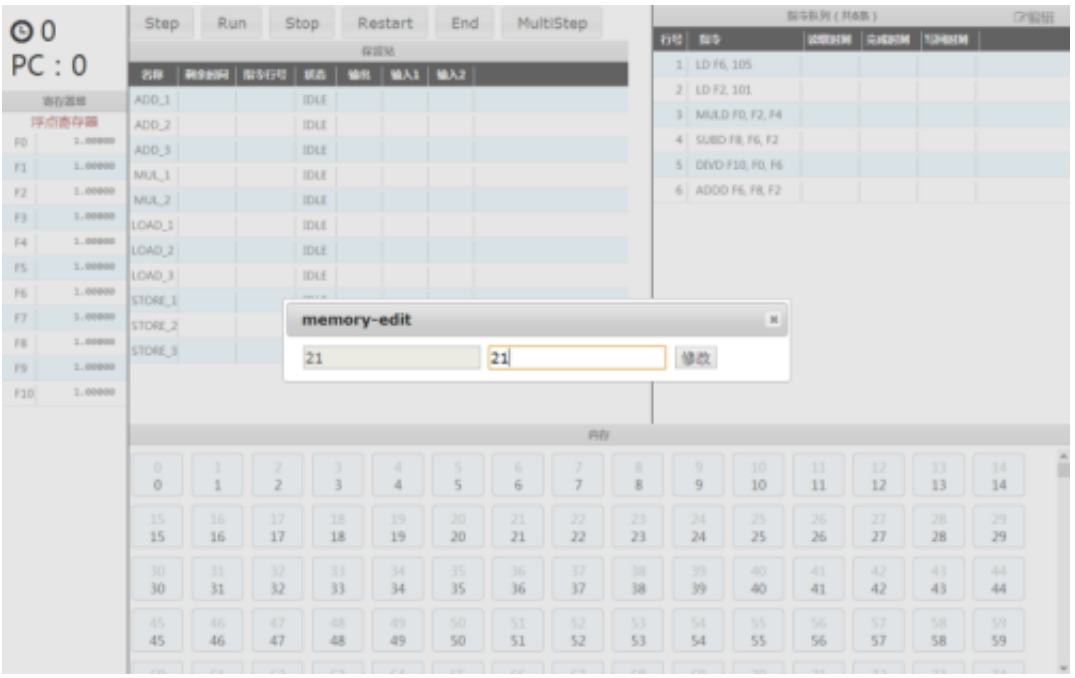
初始界面（屏幕分辨率不同可能有所不同）



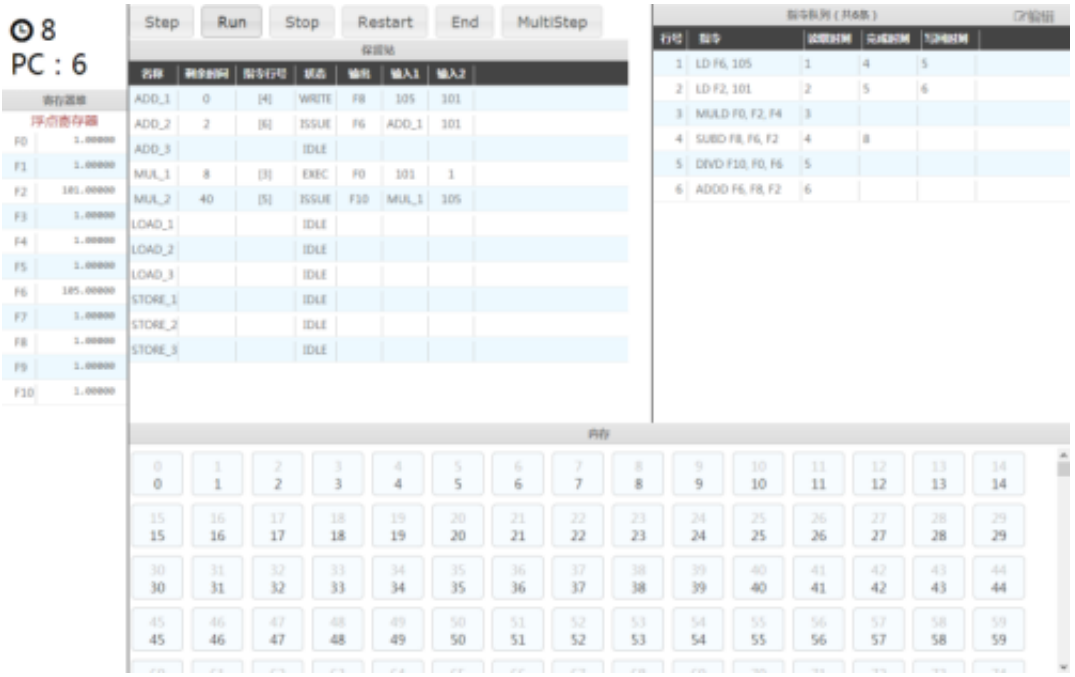
程序编辑界面



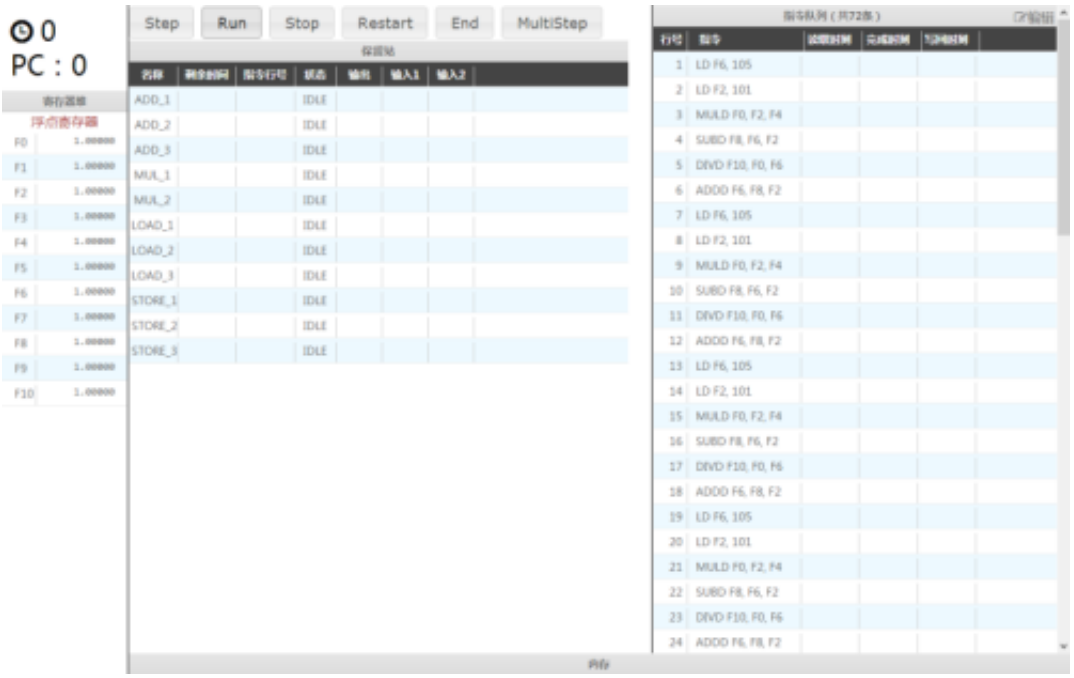
内存编辑界面



运行结果界面



收起内存面板界面



实验感想

当然，因为同时涉及到模拟器的前后端，我们在具体实现时也遇到了一些问题，具体的问题与解决方案如下：

Problem1

> 前端在加入了内存显示之后，每次单步执行界面都会因为更新内存显示值而消耗太多时间（约*2s*）。

弃用jquery库，使用原生js。速度明显变快。通过改用特殊的事件监听方式，大幅度减少内存的使用程度。但在reset时仍需要略长的时间。

Problem2

> 设计交互方式为每个内存单元都可以通过点击弹出对话框的方式更改该内存单元的值，但是这样需要为每个内存单元（*index.html*中设定为*4096*个）都一一注册*click*事件，同样在每次刷新显示时过于耗时。

不对每个内存单元注册事件，而改为对其父节点，即整片内存空间的显示元素注册一个click事件，然后根据点击的位置计算具体是点击的哪个内存单元，这也就是用户想要修改的内存单元咯。

Problem3

> 教材中所述“保留站设置在运算部件入口，浮点加法器的入口处共有*3*个加法保留站”，据此意思应该是一个运算部件对应多个保留站。但这样显然当保留站中参数都准备就绪时仍会存在结构冲突，所以实现时，我们默认是一个计算部件对应一个保留站的。

Problem4

> `WAW冲突`。因为*Tomasulo*算法是乱序执行乱序完成的，所以可能存在这种情况：同一时钟周期内，两条指令都刚好执行完成，那么在下一时钟周期，*hold*它们的保留站就都处于*STATE_WRITE_BACK*状态，而它们刚好要写回的是同一寄存器。这就存在了一个*WAW*冲突。

在Instruction中加入了指令行号（Instruction.id字段），据此，选择后id最大的写入寄存器。但这只是我们利用软件所进行的Tomasulo算法的模拟，查阅资料得知，在实际硬件实现时，是每条instruction在获得保留站时，就在寄存器组里将该指令的目的寄存器做了一个指向该保留站输出结果的标记，这样，因为Tomasulo算法是顺序流出指令的，所以最后流出的指令最后修改了寄存器的引用，WAW冲突得以解决。

Problem5

教材中所述的算法，保留站中的源擦作数标记的是寄存器号，而寄存器组中对应的寄存器又指向了寄存器状态表中该寄存器的位置，这个位置记录的才是该寄存器值的来源保留站。在我们的实现中，为了显示更为直观，去掉了中间过程，将保留站中的源直接指向了来源保留站。

附录

```
|
|  index.html    （页面入口，前台及后台设置）
|
├─css    （样式表）
|  |  font-awesome.min.css                （字体）
|  |  jquery-ui-1.10.3.custom.min.css     （对话框库）
|  |  style.css （页面样式）
|  |
|  └─images      （第三方图片）
|
├─font   （字体）
|
├─js     （后台实现）
|       buffer.js
|       common_data_bus.js
|       instruction.js
|       instruction_type.js
|       main.js
|       memory.js
|       register_file.js
|       reservation_station.js
|
└─thirdJS        （第三方库）
```