Y86 流水线模拟器项目报告

姓名：刘阳 学号：13307130167

一、项目题目

Y86 流水线模拟器

二、开发环境

本次项目使用的操作系统是 Windows 7，编程语言使用 C#，项目开发环境是 Visual Studio 2013。

三、需求分析

1.实现 Y86 指令集中的所有指令；

2.实现流水线控制逻辑(stall, bubble ,forwarding)，避免冒险；

3.输入 Y86 指令文件(*.yo)，将每个时钟周期的流水线寄存器内的十六进制数值输出到文本文件(*.txt)中；

4.实现 Y86 流水线模拟器的图形化用户界面，展示流水线运行时状态；

5.Y86 流水线模拟器用户界面提供自动运行速度选择，自动运行按钮，暂停按钮，单步运行按钮，重置按钮，前进/回退控制按钮，查看存储器按钮，运行到断点按钮，打开 Y86 汇编器按钮；

6.提供 Y86 编程环境，能够执行汇编，反汇编，并能读入文件，将汇编/反汇编结果输出到文件；

7.实现汇编器图形化用户界面；

四、设计项目

1.流水线内核实现：

流水线内核的实现主要工作是将提供的 Pipeline HCL 描述文件(waside-hcl.pdf)从 HCL 语言等价的翻译成 C#语言。

关于在 UI 中显示流水线运行时状态，主要的思想是将流水线每个周期的状态输出到.txt 文档中，然后 UI 每个周期从文件中读取一行，并显示。

在 C#中创建一个 Pipe 类。

成员变量包括：

a.分配的存储器空间；

b.各个流水线寄存器中的状态量;

c.各个阶段的过程量;

d.各个组合逻辑控制块的端口变量；

这些变量用来记录一个周期内流水线的状态。

成员函数包括：

a.初始化流水线；

b.准备内存;

c.各个阶段的执行;

d.在时钟上升沿到来时向各个流水线寄存器传值;

e.按顺序打印各个流水线寄存器的值，符合样例输出格式要求；

f.输出流水线寄存器的值，每个周期一行；

g.输出组合逻辑块的值，每个周期一行；
h.输出寄存器文件的值，每个周期一行；
i.输出条件码的值，每个周期一行；
j.输出流水线控制逻辑值，每个周期一行；
k.输出运行时栈的信息，每个周期一行；
l.输出 Y86 指令；
m.开始一个新的阶段，执行流水线的各个阶段；
n.一次执行完流水线的所有指令，将每个周期的各类值打印到文件中；

另外，实现了 HCL 描述中没有的流水线模块，如 ALU 的实现，Set_CC 的实现，Read/Write Data Memory 的实现。

2.汇编器内核的实现：

Y86 汇编器的实现主要是实现汇编和反汇编。

汇编主要是将输入的 Y86 指令分割，识别，并对应映射到相应的编码；

反汇编主要是将输入的编码依据第一位来转换到相应的 Y86 指令；

在 C#中创建一个 Assembler 类。

成员变量包括：

a.容纳编码的数组，最长 12 位；
b.分割指令和数据的数组；
c.分割两个数据的数组；

成员函数包括：

a.分割 Y86 的指令和数据；
b.从 Y86 指令中得到 ifun；
c.从 Y86 数据中得到寄存器文件；
d.从 Y86 数据中得到常数值；
e.从 Y86 数据中得到存储器地址；
f.汇编第 0,1 位；
g.汇编第 2,3 位；
h.汇编余下的位；
i.汇编；
j.从编码中得到常数值；
k.反汇编；

3.流水线 UI 的实现：

C#中通过提供的控件能够很方便的实现 UI 的绘制。

UI 从记录文件(*.txt 文档)中读取某个周期内的各类值，用于显示。

UI 具体细节：

a.“流水线视图”框的底图是一张静态的流水线结构示意图，按照书上的图 4-52 绘制；
b.通过在底图上覆盖 Label 控件，实时显示流水线运行时状态。
c.“选择文件”框提供输入文件，输出文件路径；
d.“控制面板”框提供”单步执行”，”自动执行”，”暂停”，”重置”按钮，其中：
    i.单步执行：每次从记录文件读取下一行，然后将该行中的数值显示；
    ii.自动执行：利用定时器，通过多线程，每隔一个时间间隔就调用一次”单步

执行”，直到程序执行完毕，或者”暂停“；

iii.暂停：简单地将定时器停止，继续时则开始定时器；

iv.重置：简单地将要读取的行设置为记录文件中的第一行；

e.”前进/回退”框提供移动周期数设置和”前进”，”回退”按钮，其中：

i.前进：简单地将要读取的行往下移动；

ii.回退：简单地将要读取的行往上移动；

f.”进程”框显示当前周期（当前所读的行号）和执行进度；

g.”查看存储器“框提供输入”起始地址”和“取值长度”，通过“确定”查看存储器中的值。由于 Pipe 类中的 mem 是 public 的，故可直接从中读取想要查看的值；

h.“运行时栈”框显示某周期时的栈信息。实现方法是从记录文件中读取栈信息，并且每个周期先清空显示内容，后重写要显示的内容；

i.“Y86 指令”框从文档中读取简化的指令（去除编码）,并显示。提供“运行到断点”按钮。实现方法是从记录 f_pc 的记录文件中逐行读取 f_pc，并与当前断点处指令的地址进行比较，若相等，则设置读取的行等于 f_pc 记录文件的行号，在 UI 中显示该行的记录信息；

j.”寄存器堆”，“流水线控制机制”，“条件码”框简单地将读取到的记录信息显示，其中，每次读取会读取两行，当前行和当前上一行，将两行对应数据比较，不相等的数据则要设置背景色高亮；

4.汇编器 UI 的实现：

UI 具体细节：

a.“选择源文件”，”输出结果到”框分别用来选择输入文件和输出文件；

b.”输入框”用来接收输入；

c.”输出框“用来展示输出，格式与样例输入相同；

d.”控制面板“框提供”汇编“，”反汇编“，”清空“，”退出“按钮，其中”汇编“简单地调用 Assembler 的 assemble 函数，并将结果显示到输出框，”反汇编“简单地调用 Assembler 的 disassemble 函数，并将结果显示到输出框；

五、测试项目

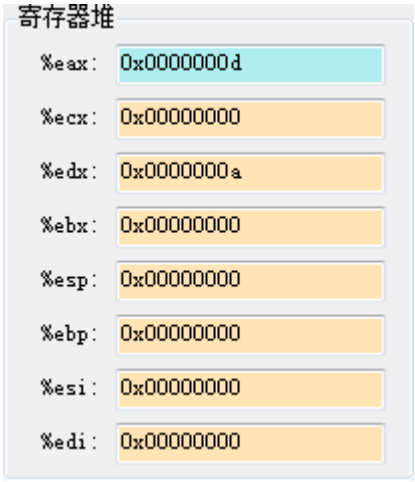
主要为了测试流水线是否能够正确执行所有指令，是否能够有效的实现流水线控制逻辑。

使用的测试文件分别是书本中的 prog1 ~ prog10。

1. prog1：

```
                        | # prog1: Pad with 3 nop's
0x000:                  | .pos 0
0x000: 30820A000000     |   irmovl $10,%edx
0x006: 308003000000     |   irmovl  $3,%eax
0x00c: 00               |   nop
0x00d: 00               |   nop
0x00e: 00               |   nop
0x00f: 6020             |   addl %edx,%eax
0x011: 10               |   halt
```

输出结果：



测试结果正确。

2. prog2：

```
                        | # prog2: Pad with 2 nop's
0x000:                  | .pos 0
0x000: 30820A000000     |   irmovl $10,%edx
0x006: 308003000000     |   irmovl  $3,%eax
0x00c: 00               |   nop
0x00d: 00               |   nop
0x00e: 6020             |   addl %edx,%eax
0x010: 10               |   halt
```

输出结果：

第 6 周期：

测试结果正确。

3. prog3：

```
                        | # prog3: Pad with 1 nop's
0x000:                  | .pos 0
0x000: 30820A000000     |   irmovl $10,%edx
0x006: 308003000000     |   irmovl  $3,%eax
0x00c: 00               |   nop
0x00d: 6020             |   addl %edx,%eax
0x00f: 10               |   halt
```

输出结果：

第 5 周期：

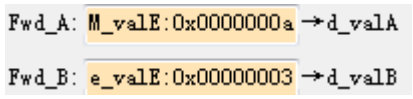
测试结果正确。

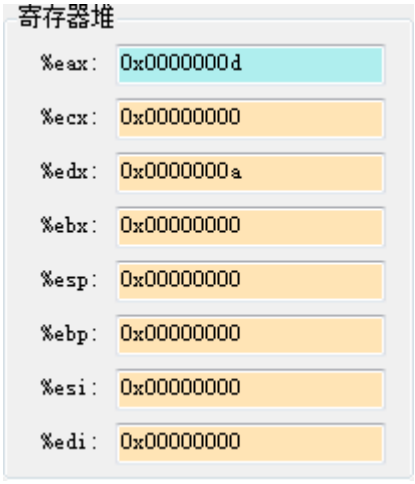
4. prog4：

```
                        | # prog4: No padding
0x000:                  | .pos 0
0x000: 30820A000000     |   irmovl $10,%edx
0x006: 308003000000     |   irmovl  $3,%eax
0x00C: 6020             |   addl %edx,%eax
0x00E: 10               |   halt
```

输出结果：

第 4 周期：





测试结果正确。

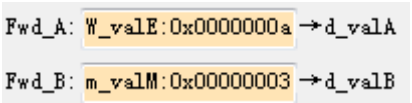
5. prog5:

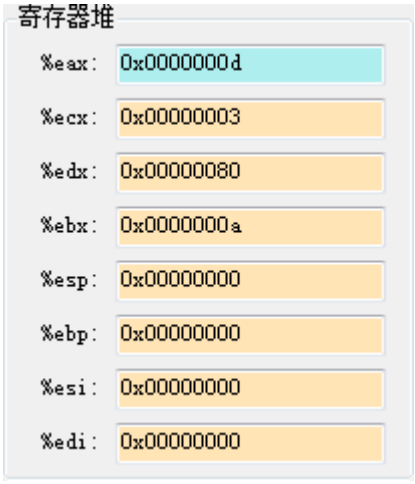
```
                        | # prog5: Load/use hazard
0x000:                  | .pos 0
0x000: 308280000000     |   irmovl $128,%edx
0x006: 308103000000     |   irmovl  $3,%ecx
0x00C: 401200000000     |   rmmovl %ecx, 0(%edx)
0x012: 30830A000000     |   irmovl  $10,%ebx
0x018: 500200000000     |   mrmovl 0(%edx), %eax  # Load %eax
0x01E: 6030             |   addl %ebx,%eax        # Use %eax
0x020: 10               |   halt
```

输出结果：

第 7 周期：

第 8 周期：





测试结果正确。

6. prog6:

```
                        | # prog6: Forwarding Priority
0x000:                  | .pos0
0x000: 30820A000000     |   irmovl $10,%edx
0x006: 308203000000     |   irmovl  $3,%edx
0x00C: 2020             |   rrmovl %edx,%eax
0x00E: 10               |   halt
```

输出结果：

第 4 周期：

测试结果正确。

7. prog7:

```
                        | # prog7:Demonstration of return
0x000: 308430000000     |  irmovl Stack,%esp  # Initialize stack pointer
0x006: 8020000000       |    call Proc        # procedure call
0x00b: 30820a000000     |    irmovl $10,%edx    # return point
0x011: 10               |    halt
0x020:                  | .pos 0x20
0x020:                  | Proc: # Proc:
0x020: 90               |    ret  # return immediately
0x021: 2023             |    rrmovl %edx,%ebx   # not executed
0x030:                  | .pos 0x30
0x030:                  | Stack:  # Stack: Stack pointer
```

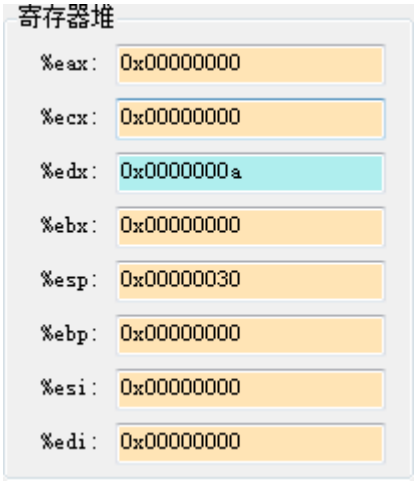
输出结果：

第 4 周期：

第 5 周期：

第 6 周期：

第 7 周期：



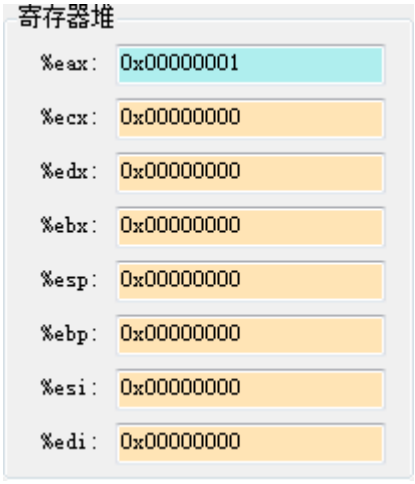
测试结果正确。

8. prog8:

```
                        | # prog8: Demonstrate branch cancellation
 0x000:                 | .pos 0
 0x000: 6300            |    xorl %eax,%eax
 0x002: 740E000000      |    jne  target        # Not taken
 0x007: 308001000000    |    irmovl $1, %eax    # Fall through
 0x00D: 10              |    halt
 0x00E:                 | target:
 0x00E: 308202000000    |    irmovl $2, %edx    # Target
 0x014: 308303000000    |    irmovl $3, %ebx    # Target+1
                        | # /* $end prog8-ys */
 0x01A: 10              |    halt
```

输出结果：

第 4 周期：



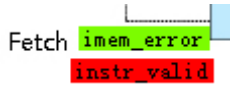
测试结果正确。

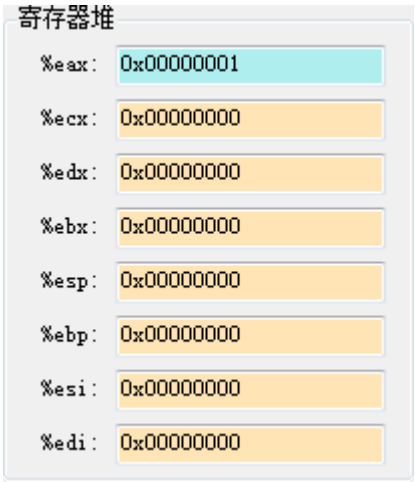
9. prog9:

```
0x000:                      | # prog9: Exception handling
0x000:                      | .pos 0
0x000: 6300                 |    xorl %eax,%eax
0x002: 740e000000           |    jne  Target      # Not taken
0x007: 308001000000         |    irmovl $1, %eax  # Fall through
0x00d: 10                   |    halt
0x00e:                      | Target:
0x00e: ff                   |    .byte 0xFF       # Invalid instruction code
```

输出结果：

第 3 周期：（红色表示 instr_valid 为 false）



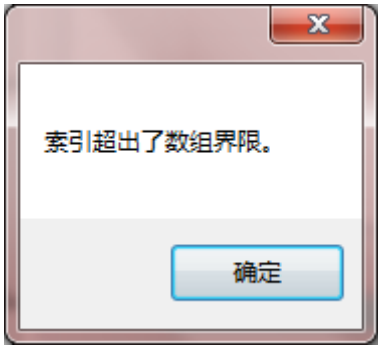


测试结果正确。

10. prog10：

```
                        | # prog10:illegal reference to memory
0x000: 308001000000     |     irmovl $1,%eax
0x006: 6344             |     xorl %esp,%esp        # Set stack pointer to 0 and CC to 100
0x008: a008             |     pushl %eax            # Attempt to write to 0xfffffffc
0x00a: 6000             |     addl  %eax,%eax       # (Should not be executed) Would set CC to 000
0x00c: 308002000000     |     irmovl $2, %eax       # Not executed
0x012: 308003000000     |     irmovl $3, %eax       # Not executed
```

输出结果：（读取的存储器地址超过了默认分配的最大地址）

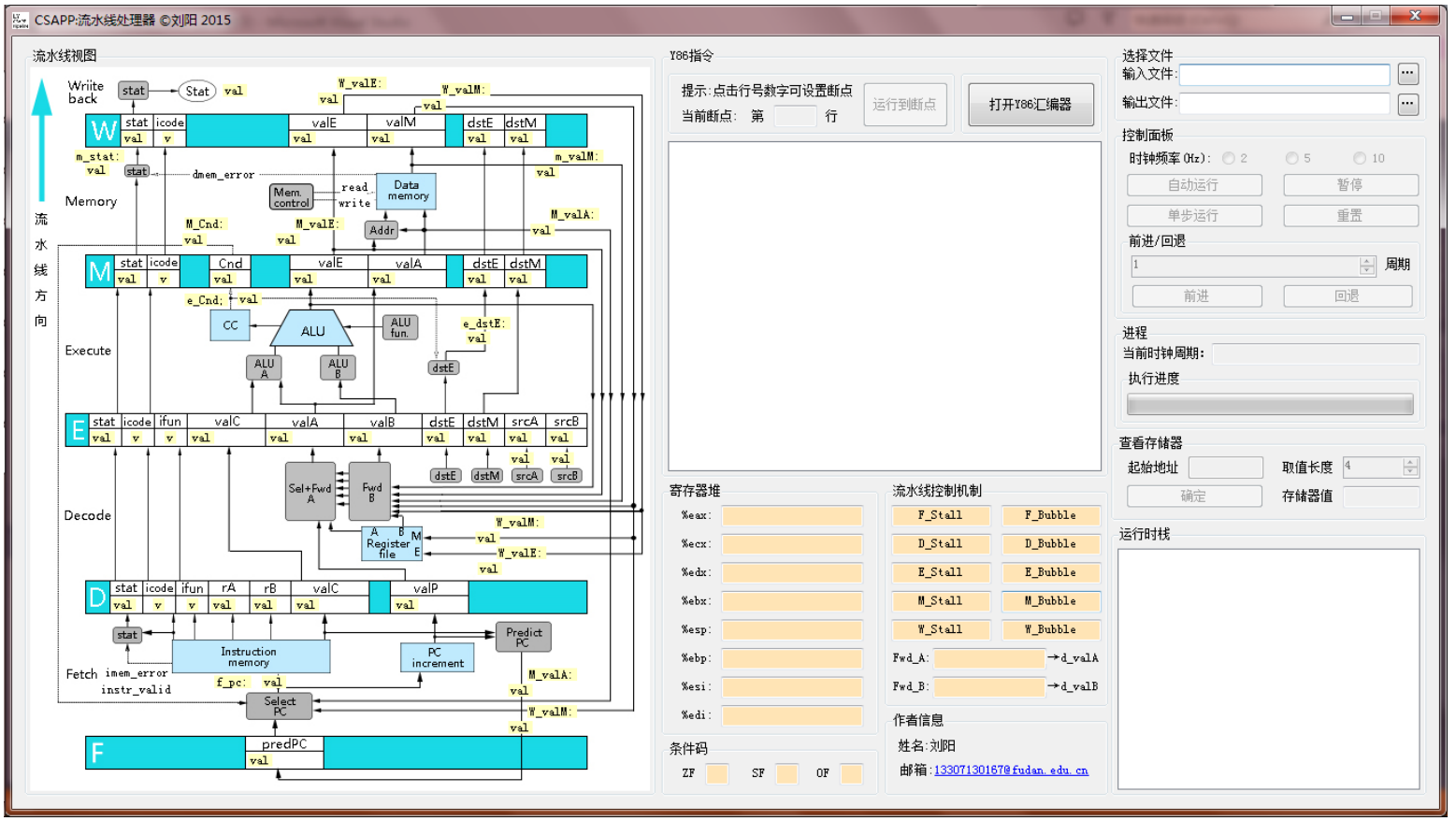


测试结果正确。

测试全部通过。

六、使用手册

1. 打开程序进入主界面：



2. 通过”选择文件”框选择要输入的*.yo 文件，要输出到的*.txt 文件；

3. 通过”控制面板”框选择要执行的操作：

a. 设置时钟频率之后可以开始自动运行；

b. 自动运行开始后可以暂停；

c. 暂停之后可以继续；

d. 一开始或者暂停之后，都可以单步执行；

e. 可以随时重置；

f. 设置前进/回退周期之后可以选择前进或者回退若干个周期；

4. 通过”进程”框可以查看当前时钟周期和执行进度；

5. 通过“查看存储器”框可以查看存储器中的值（如果没有越界）：

a. 设置起始地址和取值长度之后，点击确定即可查看存储器值；

6. 通过“运行时内存栈”框可以查看当前周期的栈信息：

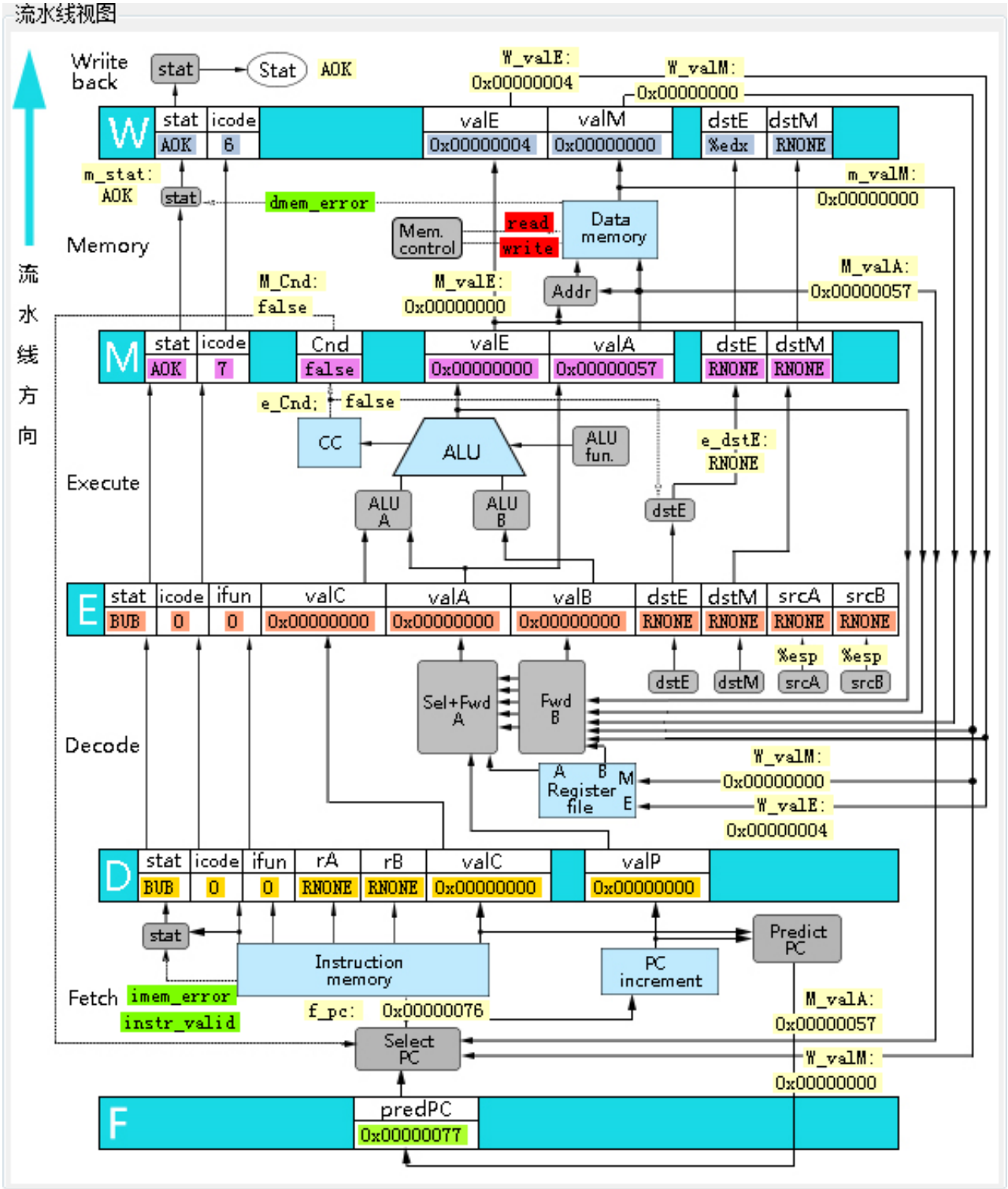
7. 通过“Y86 指令”框可以查看所运行程度的 Y86 指令：

a. 点击 Y86 指令框相应的行，可以在此行设置断点；

b. 点击运行到断点，可以使程序运行到设置的断点处；

8. 通过“寄存器堆”，“流水线控制机制”，“条件码”框可以查看当前周期的流水线状态。通过“作者信息”可以看到作者的信息，以及通过邮件联系作者。

9. 通过“流水线视图”框可以查看流水线寄存器的值和组合逻辑块的路线值：



10. 点击“打开 Y86 汇编器”可以打开 Y86 汇编器：

11. 在 Y86 汇编器中可以手动输入 Y86 指令（或编码），或者从文件中读入 Y86 指令：
（注意：实际使用中，Y86 指令第一行务必初始化内存栈）

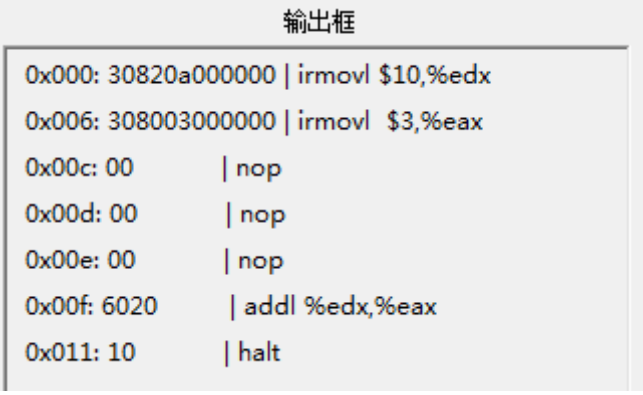
Y86 指令：

输入框
irmovl $10,%edx
irmovl $3,%eax
nop
nop
nop
addl %edx,%eax
halt

Y86 编码：

输入框
30820a000000
308003000000
00
00
00
6020
10

12. 通过点击“汇编”可以汇编输入框中的 Y86 指令，并显示在输出框中。
或者点击“反汇编”可以反汇编输入框中的 Y86 编码，并显示在输出框中：



13.可以通过文件选择按钮选择要输入的指令文件（*.txt），选择要输出到的结果文件（*.yo）；

14.点击“清空”可以清空输入框和输出框的内容。点击“退出”，可以退出 Y86 汇编器；

七、项目结果：

1.asum.yo 的输出文件 asum.txt 位于打包文件中；

2.prog1.yo~prog10.yo 的输出文件 aprog1.txt~aprog10.txt 位于打包文件中；

八、项目总结：

1.Pipe 类与 UI 的信息传送：

一开始考虑单步执行时，是想后台 Pipe 类更新一个时钟周期，然后将更新后的值传给 UI 用来显示。但是这样一来的话，回退功能就很难实现了，毕竟流水线周期是线性执行下去的，如果要得到前面周期的信息，就不得不重新执行流水线到所要的周期处。

于是想到，一旦我确定了输入的 Y86 指令文件，那么我整个 Pipeline 的所有周期的执行信息就都能确定下来了。那么我 UI 只需要从这些信息中找到要显示的信息就好了。

所以我在 Pipe 的每周期执行完之后都将 Pipeline 的各类信息都打印到文件中，每个周期一行，为了方便之后的每周期读取。

然后我 UI 更新的时候，就只要从这些记录文件中读出一行信息，分别用来显示 UI 中的信息。然后下一个周期就读取下一行文件信息。这样一来，就很方便的实现了单步执行，重置，前进和回退。

2.自动执行的多线程：

之前也用 C#写过一些项目，但是都只是用到了一些简单的特性。

这次的项目要考虑自动执行，而我的自动执行只是简单的重复单步执行，直到程序结束。

去网上找了一些 C#设置自动运行的资料（在自动运行的同时还需要更新 UI），推荐使用 C#的多线程特点。所以我就依葫芦画瓢，将更新 UI 写成了多线程，这样我在自动执行的实行也就能实时地更新 UI 了。

3.Pipe 类的内核实现：

本来对内核实现很头疼，后来在同学的提醒下，参照 HCL 描述文件尝试着翻译，并添加了诸如 Set_CC，ALU 模块的代码，调整了一些小 bug 之后，很顺利的实现了内核，完成了大头的任务。

4.实现“断点调试”功能：
一开始棘手的问题是怎样将设置断点所在的行号和指令执行到该断点的周期数联系在一起，后来想要在*.yo 文件中，断点所在行一开始是一个存储器地址，那么我就可以将这个存储器地址与记录文件中的 f_pc 值比较。如果相等，那我我就找到了该断点指令的执行周期数（记录文件中匹配的 f_pc 所在行号）。这样就可以设置 UI 中所要读的行为该行号所指向的行就好了。

5.实现“运行时内存栈”功能：
一开始想要用同样的 stack 数据结构来实现这个功能，但是这样当遇到 ret 指令时就会比较麻烦。与此同时，stack 栈空间是我分配出来的 memory 存储器空间的一部分，且栈顶向下增长。所以就想在 Pipe 中记录初始时 Stack 的值，从%esp 中读取栈顶的地址，然后将栈底和栈顶之间的 stack 内容按照从栈底到栈顶的顺序打印到文件中。后续的 UI 更新就只要每周期从该记录文件中读取栈信息就好了。