

CSAPP Y86 PJ Report

Qingfu Wan

Department of Computer Science, Fudan University

May 31, 2015

1 Introduction

建议用250%模式查看

主界面：两个按钮。

单击“编辑器”按钮或选择菜单栏“编辑器”进入文本编辑器，单击“处理器”按钮或选择菜单栏“处理器”进入处理器，单击关于进入版本信息界面。

本程序主要实现了以下几个功能：

- (1) 将Y86汇编代码处理成指令编码的形式，即用十六进制指令表示。
- (2) 模拟Y86指令在CPU中的执行，实时显示如书上图4.52的PIPE Hardware Structure，实时显示每条指令流经FDEMW的二维表格(行为指令，列为周期)，实时显示历史周期中FDEMW各状态变量的值，实时显示内存数据及栈状态。
- (3) 支持修改CPU频率，控制指令执行速度。
- (4) 支持运行到某一特定周期，支持多断点（指令执行到断电集合中任意一个程序暂停），支持自动运行，支持前进一定指令和后退一定指令。
- (5) 显示代码的编辑器关键词高亮，背景色高亮，在行头显示该指令所处的FDEMW阶段，随着程序执行高亮代码不断变化。

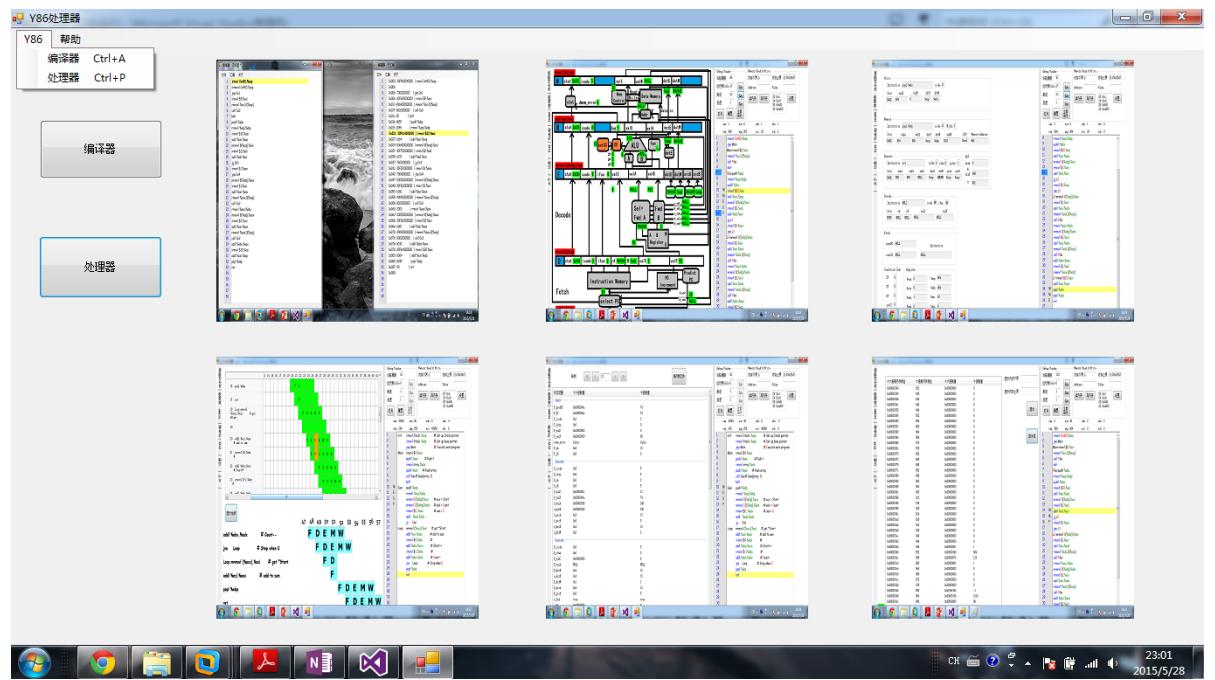


Figure 1: Overall Form

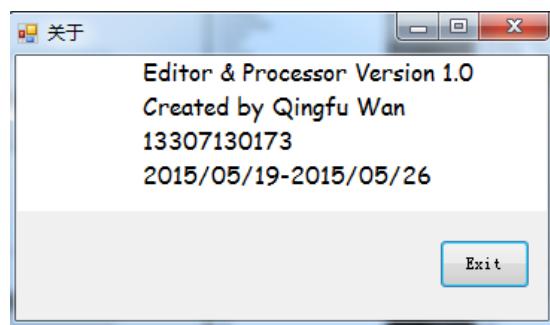


Figure 2: About All

2 About The Program

关于整个软件

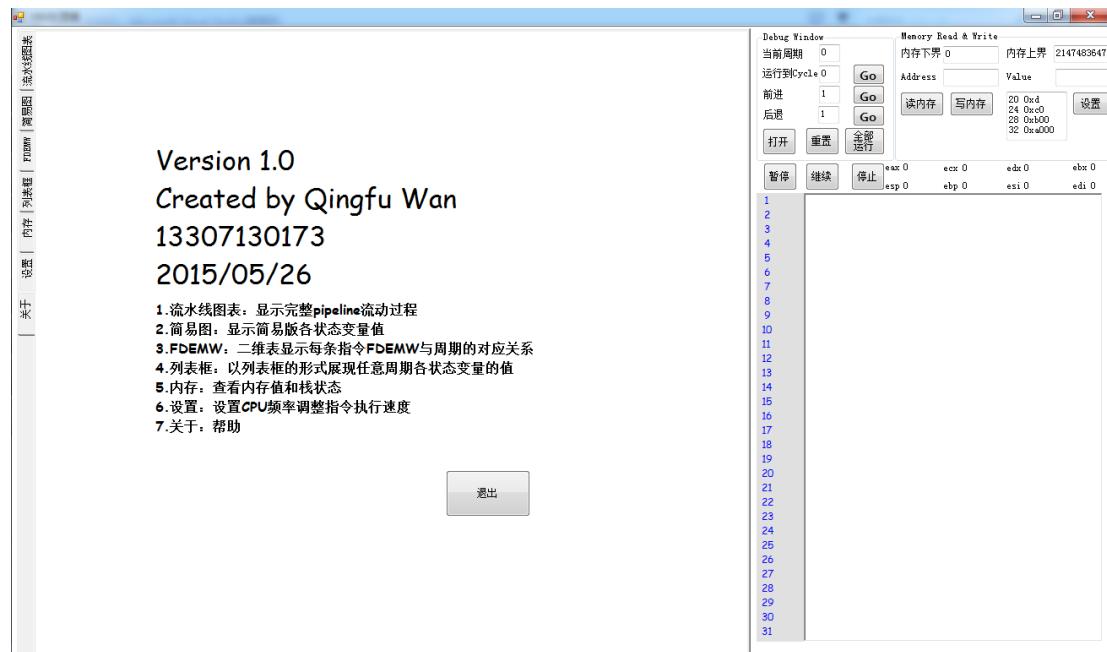


Figure 3: Introduction To This Program

3 Form1—Editor

3.1 使用说明

单击“文件”菜单，选择“新建”或“打开”，如果当前编辑的代码没有保存，标题栏会显示“无标题- *”，且会弹出窗口提示未保存。



Figure 4: Save Before Open Or New File

输入一段Y86代码，注意仅在指令如irmovl和寄存器（立即数）之间有空格，irmovl,rmmovl,mrmovl的立即数用诸如\$0x5F或\$1234表示，call和jXX后面直接跟地址如0xAB0或2345这样的数字表示。

单击汇编,左图为输入的Y86代码，右边为转换后的指令码，可单击“菜单”——“保存”按钮保存。

A screenshot of the application showing two windows. On the left, a window titled '编辑器 - 无标题' displays Y86 assembly code. On the right, a window titled '编辑器 - 已汇编' displays the corresponding machine code. The assembly code includes instructions like irmovl, jmp, addl, and call. The machine code shows the binary representation of these instructions.

Figure 5: Before & After Assembling——Convert To Instruction Code

3.2 实现细节

- 编辑器用的RichTextBox,由于RichTextBox的高亮只对一行的选中文本有效,要实现整行高亮比较困难,于是设计了一个填充颜色为黄色的Label控件悬浮在RichTextBox上方,根据光标所在位置移动Label。由于整个窗口可能不能显示全部代码, RichTextBox滚动条滑动VScroll事件触发行号发生变化,在ListBox中只显示当前窗口内所有行的行号。
- 将Y86处理成指令码的过程:
把指令符号如rrmovl, 立即数或寄存器, 逗号, 立即数或寄存器用字符串处理分割开, 然后编码, 没什么好说的。

4 Form2—CPU Processor

4.1 流水线图表

4.1.1 使用说明

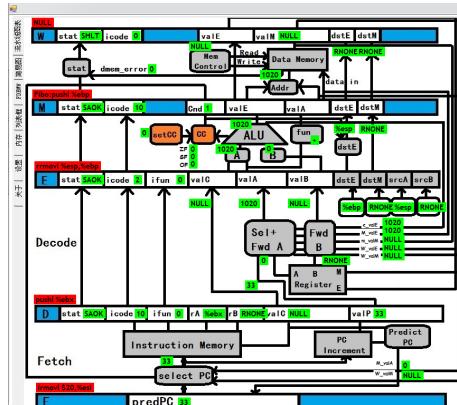


Figure 6: Overall Pipeline Diagram

- 单击“全部运行”按钮, 程序自动执行, 可以看到各个阶段各个状态变量的值自动变化, 绿色框中是变量的值, 红色框中是当前某个状

态寄存器中的指令。运行完毕后会弹出窗口显示运行完毕时的状态(*SHTL*, *SADR*),三个Go按钮、暂停、继续、停止按钮失效。



Figure 7: Run All

- 单击“重置”按钮后，按钮恢复有效，回到最开始的周期0,按全部运行可以从头开始运行指令。
- 单击左侧行号可以设置断点，支持多选设置多个断点，再次单击可以取消设置断点。当断点集合中任一指令进入Fetch阶段，程序自动暂停，如下图。断点会随滚动条滚动而自动滚动。左侧FDEMW显示每条指令进行到哪一阶段。

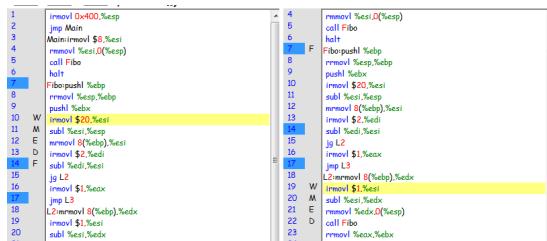


Figure 8: Breakpoint Diagram

- 单击左侧选项卡“设置”可以设置CPU频率，1Hz大概是1秒一条指令。单击“暂停”按钮暂停指令执行，单击“继续”按钮继续执行，单击“停止”按钮停止当前整个Y86指令的执行，如下图。比如按“全部运行”程序停在断点处，再按“继续”或“全部运行”程序继续运行直到下一个断点或程序终止处。
- 在Address中输入想要读/写的内存地址，单击“读内存”在Value中显示内存存在这一地址的值，单击“写内存”将Value赋值给内存Address地址，或者按照文本框中的格式：地址 值，然后单击“设置”按钮将文本框中的信息写进内存。



Figure 9: Timer Setting



Figure 10: Memory Read And Write

4.1.2 实现细节

当内存地址 ≤ 20000 时，直接存在hash表中，否则存在一个二维数组mapmem中，第一维是地址，第二维是值。本来是想存在类似C++的STL map里面的，考虑到只作演示用，没有太多考虑效率的问题。具体流水线各寄存器、状态变量值的更新见下文。有了单步执行的代码，运行到某一cycle，前进/后退若干cycle，断点暂停等都是小事。具体使用了RichTextBox控件和左侧两个ListBox控件。

4.2 简易图

简易图里在文本框中显示各状态变量各寄存器的值，文本框只读。

4.3 FDEMW

4.3.1 使用说明

左侧为指令，右侧为指令所经过的阶段，有可能不是全部的FDEMW，上方数字表示对应周期，可以看到每条指令在什么周期进入到了什么阶段。暂停(stall)的F或者D用另一种颜色填充。

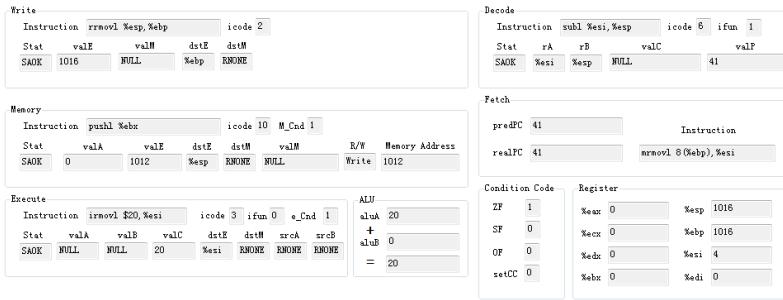


Figure 11: Simple Diagram

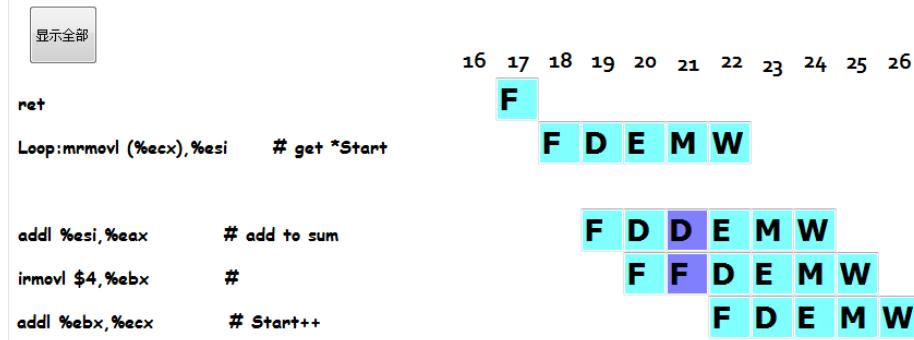


Figure 12: FDEMW

单击“显示全部”按钮可以看到每条指令执行FDEMW阶段所对应的周期。

4.3.2 实现细节

为了方便展示，需要记录每个周期FDEMW五个寄存器中的指令分别是什么，需要记录每条指令是否经过了F、D、E、M、W五个阶段以及分别在什么周期处于这些阶段。值得注意的是当遇到加载/使用异常时，F和D分别为stall，上一条指令FDDEMW，下一条指令FFDEMW，到再下一条指令就恢复正常了，即stall只会在前后两条指令(一条load一条use)执行时发生。具体使用了DataGridView控件。

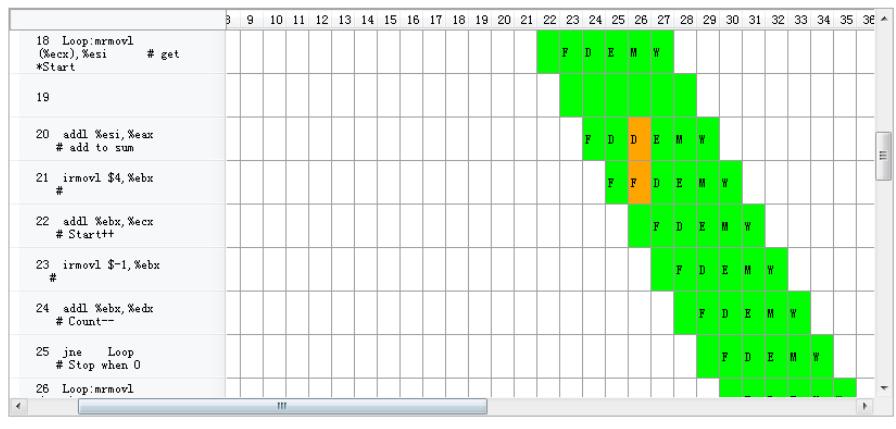


Figure 13: showall

4.4 Show On ListBox

4.4.1 使用说明

在文本框中输入想要查看的周期数,按回车即可看到输入周期的Fetch、Decode、Execute、Memory、Write Back阶段所有状态变量的值(包括异常状态,所在执行的指令具体是哪条以及一些不太重要的中间变量)、寄存器的值等等。按“⟨”回退,“⟩”前进,“⟨⟨”设置第一个周期,“⟩⟩”设置最后一个周期,“保存到文件”将每个周期各个变量的值保存到文件,就像pj的ppt要求上写的那样。

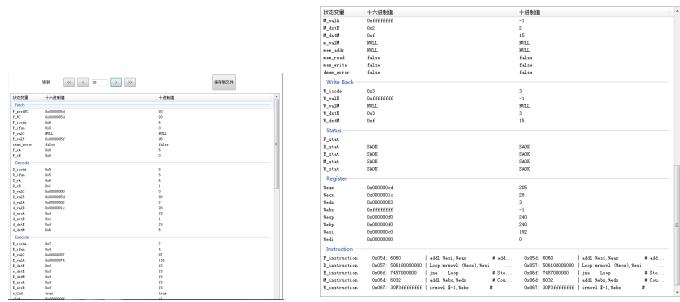


Figure 14: ListBox

4.4.2 实现细节

只需要用数组保存每个周期各个状态变量的值即可。

4.5 Memory

4.5.1 使用说明

在文本框中输入想要显示的内存上界和内存下界，单击“显示”按钮，在列表框中呈现地址和地址对应的值。

十六进制内存地址	十进制内存地址	十六进制值	十进制值	显示内存下界	显示内存上界	显示	显示栈
0x00000290	940	0x00000000	0				
0x0000034	948	0x00000001	1				
0x0000036	952	0x00000000	0				
0x0000038	956	0x00000000	0				
0x000003d	960	0x00000001	1				
0x000003e	964	0x00000000	0				
0x000003f	968	0x00000000	0				
0x0000040	972	0x00000000	0				
0x000004d	976	0x00000000	0				
0x000004e	980	0x00000000	0				
0x000004f	984	0x00000000	0				
0x0000050	988	0x00000000	0				
0x0000054	992	0x00000000	0				

Figure 15: Show Memory

单击“显示栈”按钮显示从栈底esp往上若干地址元素的值，一个绿色的箭头指向栈底。

十六进制内存地址	十进制内存地址	十六进制值	十进制值	显示内存下界	显示内存上界	显示	显示栈
0x00000360	954	0x00000000	0				
0x00000364	958	0x00000000	0				
0x00000368	972	0x00000000	0				
0x00000370	980	0x00000000	0				
0x00000374	984	0x00000000	0				
0x00000378	988	0x00000000	0				
0x0000037c	992	0x00000000	0				
0x00000380	996	0x00000000	0				
0x00000384	990	0x00000000	0				
0x00000388	984	0x00000000	0				
0x00000390	980	0x00000000	0				
0x00000394	972	0x00000000	0				
0x00000398	960	0x00000000	0				
0x0000039c	956	0x00000000	0				
0x000003a0	950	0x00000001	1				
0x000003a4	944	0x00000000	0				
0x000003a8	948	0x00000001	1				
0x000003b2	952	0x00000000	0				
0x000003b6	956	0x00000000	0				
0x000003c0	960	0x00000001	1				
0x000003c4	964	0x00000000	0				
0x000003c8	968	0x00000000	0				
0x000003cc	972	0x00000000	0				
0x000003d0	976	0x00000000	0				
0x000003d4	980	0x00000002	2				
0x000003d8	984	0x00000000	0				
0x000003dc	988	0x00000000	0				
0x000003e0	992	0x00000000	0				
0x000003e4	996	0x00000000	0				
0x000003e8	1000	0x00000000	0				
0x000003ec	1004	0x00000000	0				
0x000003f0	1008	0x00000000	0				
0x000003f4	1012	0xffffffff	-1				
0x000003f8	1016	0x00000000	0				
0x000003fc	1020	0x00000000	0				
0x00000400	1024	0x00000004	4				

Figure 16: Show Stack

4.5.2 实现细节

简单的ListBox操作。

4.6 Setting

4.6.1 使用说明

选择“不使用计时器”时，程序自动执行，当选择“使用计时器”时，可设置CPU每秒执行的指令数，即Hz，范围是1-1000Hz，滑动滑动条时，文本框中的频率自动变化，当然也可以手动输入设置频率。

4.6.2 实现细节

使用一个timer，不使用计时器时，将其Enabled属性设置为false，否则设置为true，每次触发计时器Tick事件，都判断当前是否在运行程序（只对“全部运行”按钮有效），当单击过“暂停”后，每次计时器计时，Tick事件中什么也不执行，当再单击“继续”后，下次计时器计时触发Tick事件就又会正常执行程序。

5 Y86 Implementation

整个Y86执行部分是按照以下机制进行的：

- 基本执行单位为周期，对于每个周期按照Write Back,Memory,Execute,Decode,Fetch部分计算。
- 当前周期的W由上一周期的M计算得来，如果上一周期M的状态是SBUB（气泡未执行）或SHLT（halt）或SADR（访问内存地址错误或当前指令地址错误）则W阶段没有做任何事。
- 类似的，当前周期的M由上一周期的E计算得来，若E非SBUB，SHLT或SADR

- 当前周期的E由上一周期的D计算得来，需要注意的是如果D_stall=1即D阶段需要暂停一周期，则向当前E插入一个气泡。
- 当前周期的D由上一周期的F计算得来，如果F_stall=1，则向当前D插入一个气泡。
- 当前周期的F阶段获取的PC由PC预测值F_predPC和上一周期M_valA, W_valM综合决定。
- 按照从W到M到E到D到F计算的一大好处是算W时可以直接用M阶段各个状态变量值，因为此时这些状态变量并未更新，还是上一周期时候的值，这样执行保证了有序性。相当于是从F推进到D，从D推进到E，从E推进到M，从M推进到W，从W推进到F，各状态变量是通过流水线一点一点向上传递的，本质是一个迭代过程。
- 计算F、D、E、M、W五大阶段各变量的代码主要参考书上HCL代码，做了一些小改动，总计约400行。
- 一些重要的细节：
 - (1)当D阶段暂停时不是什么都不做，还是需要Decode valA和valB，根据条件选择forward哪个值到d_valA和d_valB。
 - (2)如果某一阶段（如Decode）在当前周期是Bubble状态，即未执行，Decode部分的所有相关变量都要强行设为Null(我用的是INTMIN=-2147483648)，如果这个错的话可能会引起其它地方出现异常。
 - (3)如何决定程序如何终止？如果当W寄存器状态为SHLT（程序本身自带的halt）或者为SADR（两种情况，一种是访问内存Memory阶段产生的内存地址超过范围出错，一种是Fetch阶段Fetch到一个wrong的PC，即PC地址超过范围），则程序应当终止。
 - (4)有些变量在某些周期是没有用的，比如有些指令 JMP不用ALU，要设为空。
- 伪代码：

1. $W \leftarrow M$
 stat: i-code, valE = valM (m-valM) / dstE destM
 stat: m-start { M-start
 | mem-error (mem-addr 超过允许范围)

2. $M \leftarrow E$ (E # bubble)
 stat: i-code, M-addr = E-addr, M-valE, M-valA, M-dstE, M-dstA
 Memory Stage
 stat: MR, pop, RET M-valM \leftarrow Memory (mem-addr)

3. RAN, PUSH, CALL Memory (mem-addr) \leftarrow M-valA
 mem-addr: { M-valE : RAN, PUSH, CALL, MR
 M-valA : POP, RET

4. $E \leftarrow D$ (D # stall, E # bubble, D # bubble)
 stat: i-code, ifun, valC, valA (from d-valA), valB (from d-valB),
 dstE (d-dstE), dstM (d-dstM), srcA (d-srcA), srcB (d-srcB)
 aluA { Eval C LR RU MR
 | E-valA ret op aluB { O RR LR CMov
 | ret pop | valB, RU MR OP CAN PUSH RET, pop
 | push call

set-CC = Et-iwave (t cycle) = l(OPI) & ! M_i-stat in { SADR, SARS, SHCR }
 SARS, SHCR & ! W-stat in { SADR, SARS, SHCR }.

alu fun: OPI
 1: ADDL (J ~~MR~~ ^{MR} ALU)

.E-valE = ~~val~~ aluA op alu B,
 set-CC \rightarrow CC-, R-addr \rightarrow E-dstE (may be RegE)

4. $R \leftarrow F$ (F # stall, D # bubble, D # stall)
 stat: i-code, ifun, rA, rB, valL, valP
 { A-str, valid
 mem-error

Figure 17: 1

$d\text{-srcA}$	$\begin{cases} rA \quad D_t\text{-icode} (\$R) \text{ in RR, RM op. PUSHL, CMovxx} \\ RSP \quad D_t\text{-icode in POPL, RET} \\ \text{RvNGz: } 1 \end{cases}$
$d\text{-srcB}$	$\begin{cases} rB \quad D_t\text{-icode in }\{OPL, RM, MR\} \\ RSP \quad D_t\text{-icode in }\{\text{PUSHL, POPL CALL, RET}\} \\ \text{RvNG: } 1 \end{cases}$
$d\text{-dstG}$ $D_t\text{-icode}$	$\begin{cases} rB \quad D_t\text{-icode in RR, LR, OPL, } \\ RSP, \text{ PUSH, POP, CALL, RET} \\ \text{RvNG: } 1 \end{cases}$
$d\text{-dstM}$	$\begin{cases} rA \quad D_t\text{-icode in MZ, POPL, } \\ \text{RvNG: } 1 \end{cases}$
$d\text{-valA: }$ $\text{popl } rA$ $R2\text{-esp, ral, T}$	$\begin{cases} D_t\text{-icode CALL JXX} \rightarrow D_t\text{-valP} \\ d\text{-srcA} = E_t\text{-dstE} \quad \# : R_t\text{-valT} \\ d\text{-srcA} = M_t\text{-dstM} \quad M_t\text{-valG} \\ d\text{-srcA} = M_t\text{-dstE} \quad M_t\text{-valE} \\ d\text{-srcA} = W_t\text{-dstM} \quad W_t\text{-valM} \\ d\text{-srcB} = W_t\text{-dstE} \quad W_t\text{-valE} \\ \text{RvalA} \in \text{valM} \quad (\vdash d\text{-rvclA} \quad \text{RvalA} - rA) \end{cases}$
$d\text{-valB: }$	$\begin{cases} d\text{-srcB} = E_t\text{-dstE} \quad \cancel{W_t\text{-valP}} \text{ RvalT} \\ d\text{-srcB} = M_t\text{-dstM} \quad M_t\text{-valM} \\ d\text{-srcB} = M_t\text{-dstE} \quad M_t\text{-valE} \\ d\text{-srcB} = W_t\text{-dstM} \quad M_t\text{-valM} \\ d\text{-srcB} = W_t\text{-dstE} \quad W_t\text{-valE} \\ \vdash d\text{-rvclB} \quad \text{RvalB} \end{cases}$

Figure 18: 2

5. F-pred PC = E_{t1} -val C F_{t1} -Node in {WXX CAM},
 F_{t1} -valp 且 \in $\frac{1}{2}$
 E_{t1} PC $\left\{ \begin{array}{l} M_{t1} \text{-icode} = JXX \& !M \text{-cond } (\#E_{t1} \#S) \\ W_{t1} \text{-icode} = URET. W_{t1} \text{-valm} \\ |: F \text{-pred PC} \end{array} \right.$
 fetch $\frac{1}{2}$ 3 由 $\frac{1}{2}$ F_t - icode, $\frac{1}{2}$ - if $\frac{1}{2}$ F_t - rA $\frac{1}{2}$ F_t - rB,
 F-val C, F_t - valp
 6. D-stall: E_{t1} -icode MR pop
 E_{t1} - dstm in {dt1-srcA, dt1-srcB}
 D-bubble: E_{t1} -icode: JXX & !e-Cnd ||
 ! (Gt1-icode { MR pop } & &
 Gt1-dstm { dt1-srcA, dt1-srcB })
 & URET in { Dt1-icode, Gt1-icode, M_{t1}-icode }
 G-bubble: (E_{t1} -icode = JXX & !e-Cnd)
 || (E_{t1} -icode in MR, pop) & &
 Gt1-dstm in { dt1-srcA, dt1-srcB }
 F-stall: (E_{t1} -icode { MR pop })
 E_{t1} -dstm in { dt1-srcA, dt1-srcB }
 || URET & { Dt1-icode, Gt1-icode, M_{t1}-icode }

Figure 19: 3

6 Test

6.1 Test 1: Fibonacci(n=7)

计算Fibonacci第7项，执行情况如下（结果保存在eax中，eax=13）

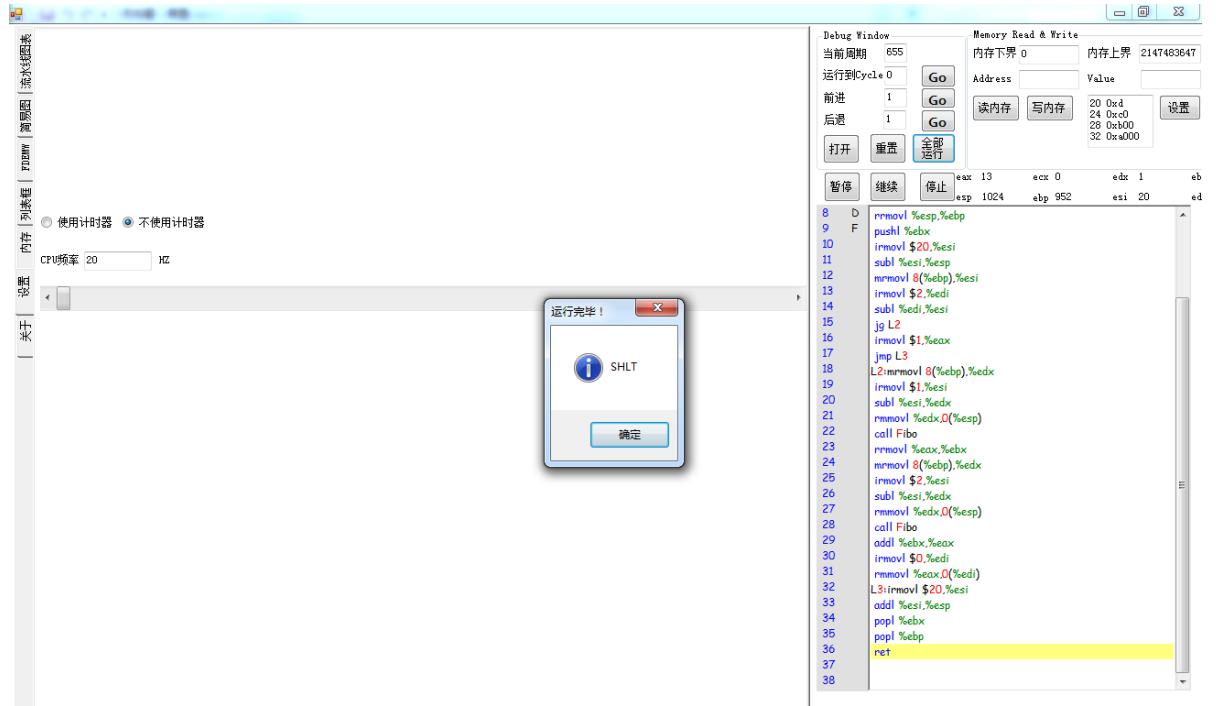


Figure 20: Fibonacci N=7

6.2 Test 2: Fibonacci(n=10)

计算Fibonacci第10项，执行情况如下（结果保存在eax中，eax=55）

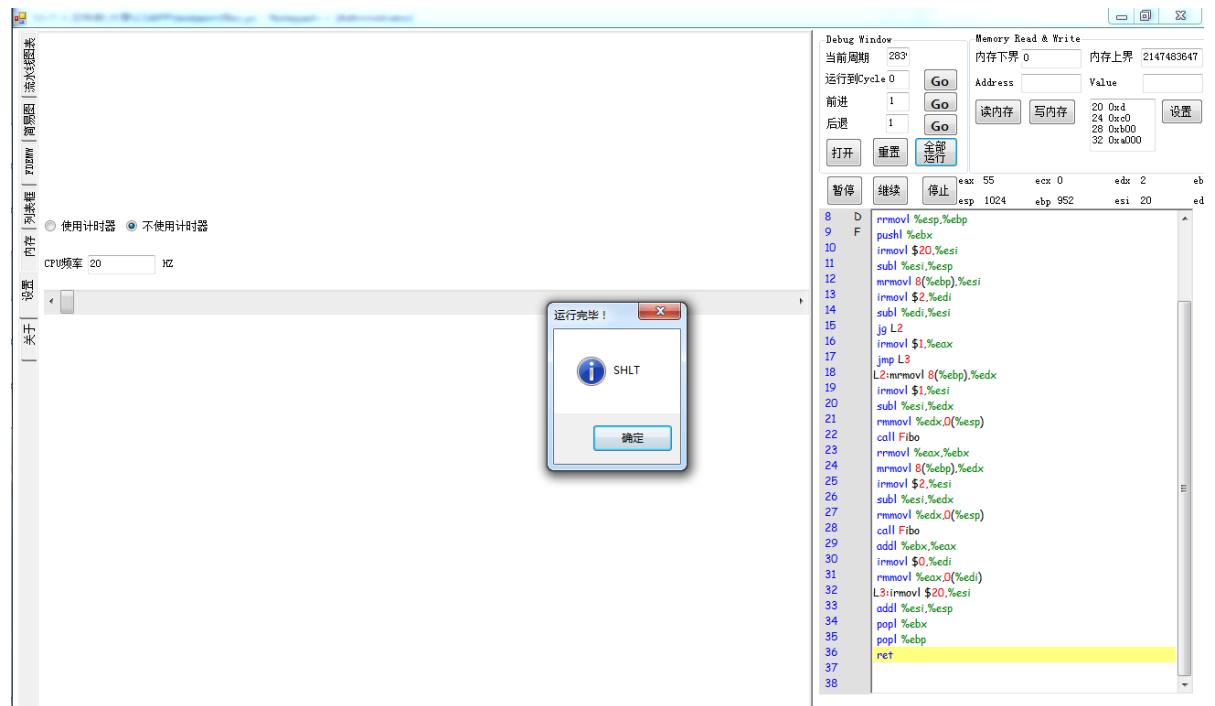


Figure 21: Fibonacci N=10

6.3 Test 3: asum

书上4.1.2小节的例子

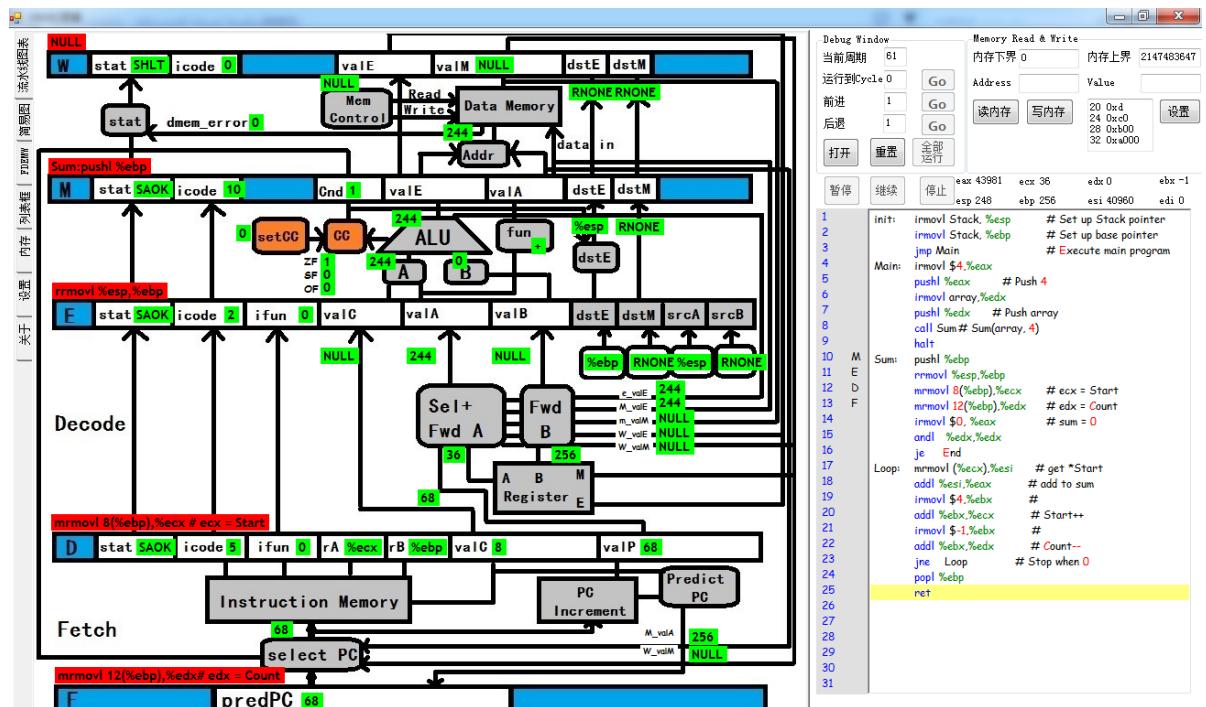


Figure 22: asum

7 Key Point

本程序有以下几个亮点：

1. 自动将汇编语句转换为指令码。
2. 画出了书上完整的PIPE图，能动态显示指令语句流经流水线各阶段的过程，动态显示各变量向上传递的过程。
3. 支持设置多个断点，精确控制指令执行速度，随时暂停继续终止，前进或后退任意周期，在程序执行过程中，一行的最前面显示这条指令

处于F、D、E、M、W中的哪个阶段，调试功能强大。

4.代码编辑器背景高亮，关键字高亮，显示行号。

5.以多种方式展现寄存器、状态变量、内存和栈中的值，可以方便查看历史的任意时刻的这些值。

6.画出了类似书上的梯形图，可以方便的查看当前周期有哪些语句处在什么状态，有哪些状态并没有任何语句在执行，可以完整地画出这样的梯形图。

7.在所有程序段中都加入了try...catch(System.Exception err)这样的捕捉异常语句，保证程序在执行过程中若出现运行时错误会自动跳出所在程序块，而不会退出整个程序。

8.布局还可以，虽然没有动画，但是表现形式多样，生动形象，便于理解整个Pipeline的流程。

8 Fundamentation

本人有一定的UI和c#基础，上学期的数据结构pj用c#调用c++程序实现了一个轨迹分析系统。

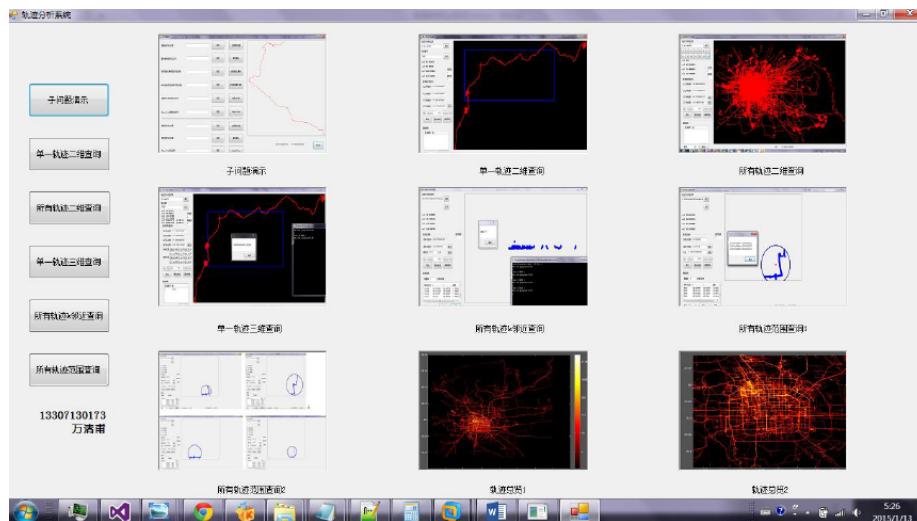


Figure 23: ds1

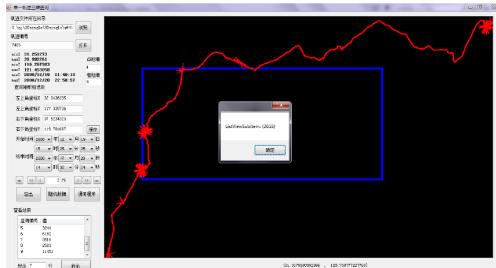


Figure 24: ds2

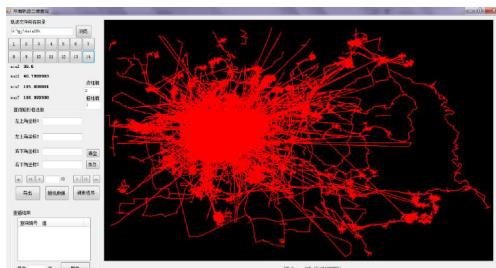


Figure 25: ds3

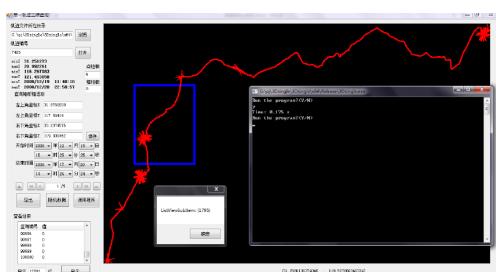


Figure 26: ds4

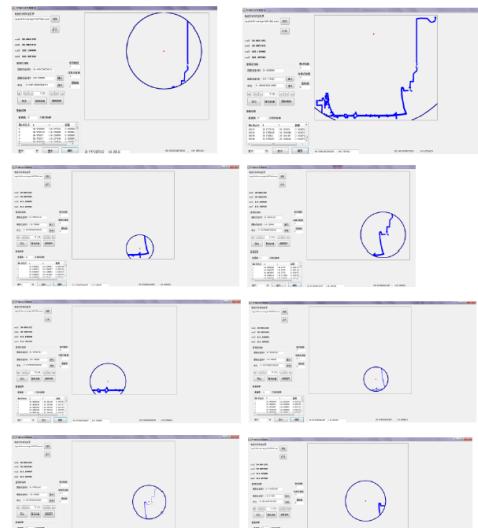


Figure 27: ds5

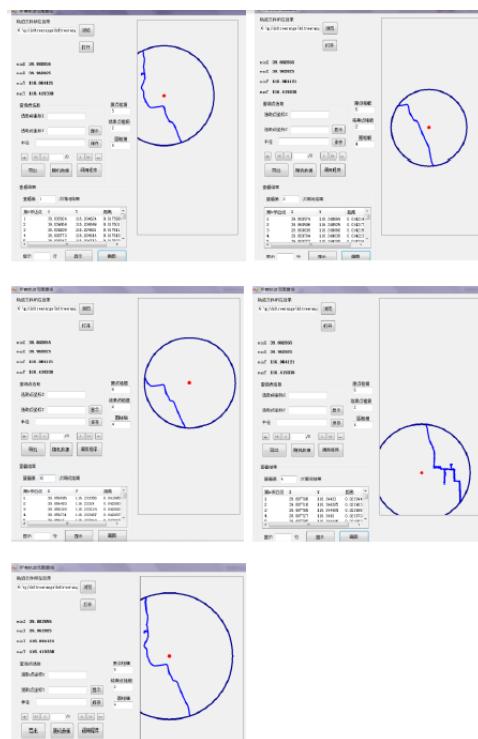


Figure 28: ds6

9 Shortcomings

存在的缺点和不足有以下几点：

- 1.没有完整的动画，不能动态展现值传递过程。
- 2.流水线图表中的背景图画的线有些乱，覆盖在其上的Label控件布局不够美观。
- 3.汇编器暂不支持jXX+函数名比如jmp .L2这种形式转换为指令码（觉得没有什么必要），如果要保存常量数组到内存中，只能手动输入进行设置（觉得没有什么必要）。
- 4.没有用STL中的map模拟内存存储，觉得耗时费力。
- 5.查看历史状态值不支持查看历史内存中的值。（这样的话要保存二维数组，空间开销会非常大）
- 6.简易图没有仔细布局。
- 7.最多只支持11111条指令，FDEMW选项卡中那个DataGridView最多之能显示1111行*1111列。
- 8.模拟CPU运行Y86指令速度比较慢。
- 9.没有增加新指令，或者模拟Cache机制？乱序执行？(什么鬼)

10 Preview Of All Files

文件名	功能	行数
frmaboutall.cs	主界面的帮助——关于	26
frmabouthitor.cs	编辑器的帮助——	26
frmeditor.cs	编辑器（编译Y86码）	629
frmmain.cs	主界面	49
frmtaby86.cs	处理器（核心部分）	2669
	总计	3350

Table 1: Preview

11 Experiment Environment

需要安装.NET Framework 4.5

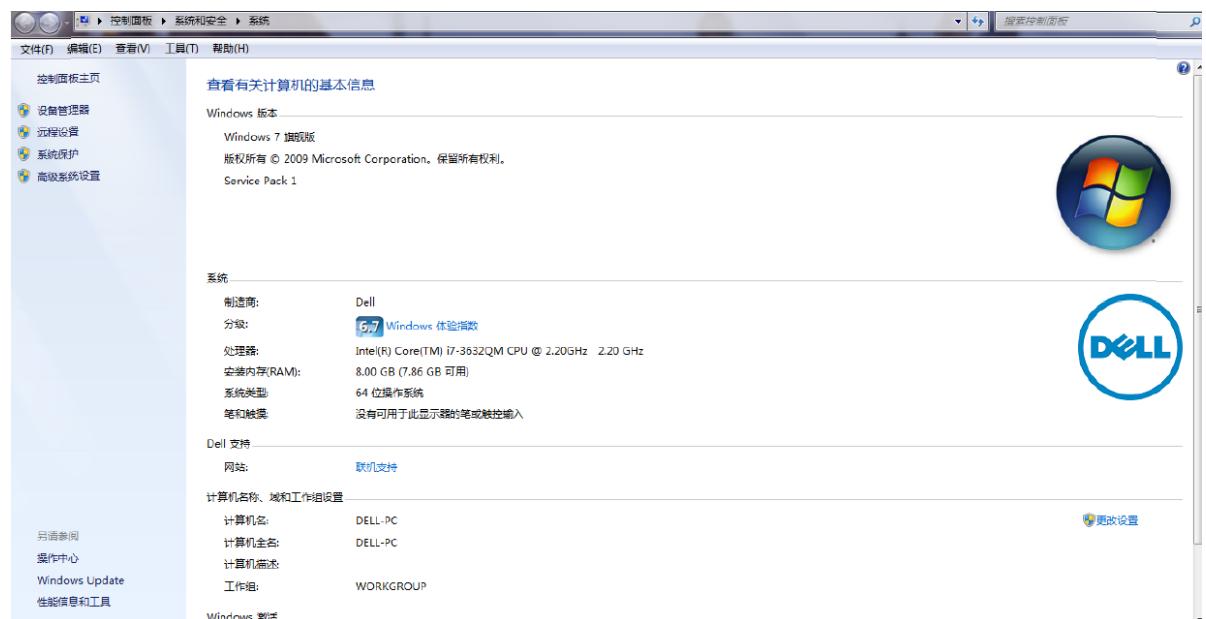


Figure 29: Computer

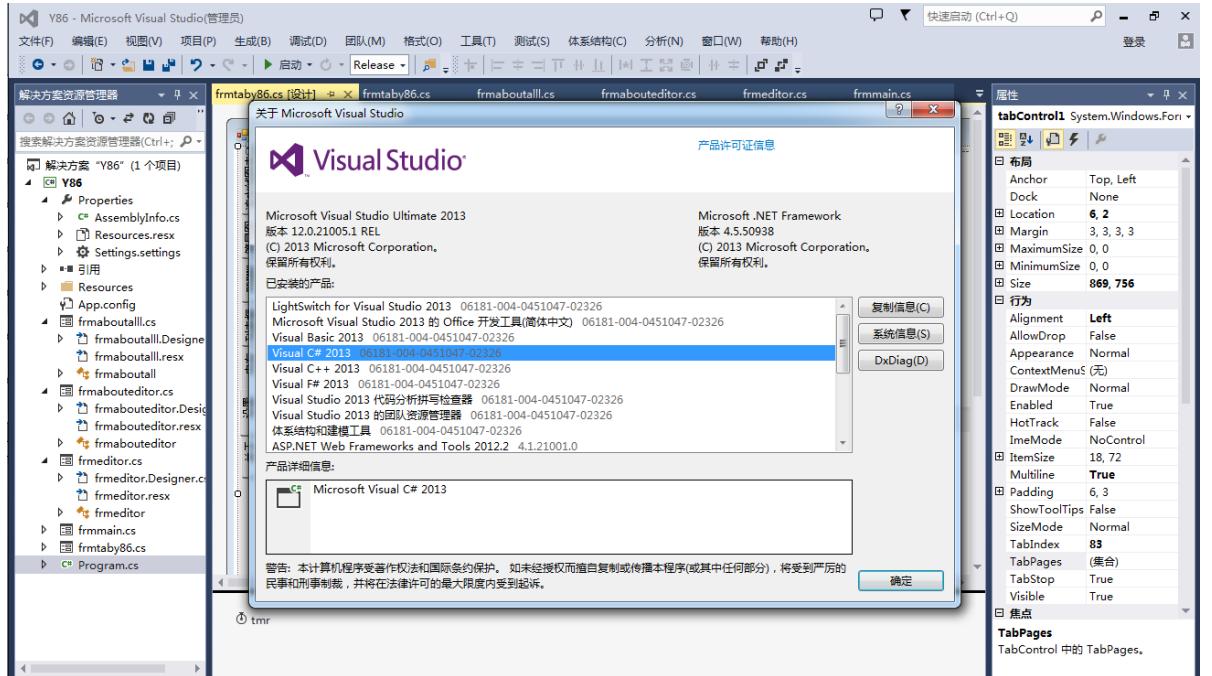
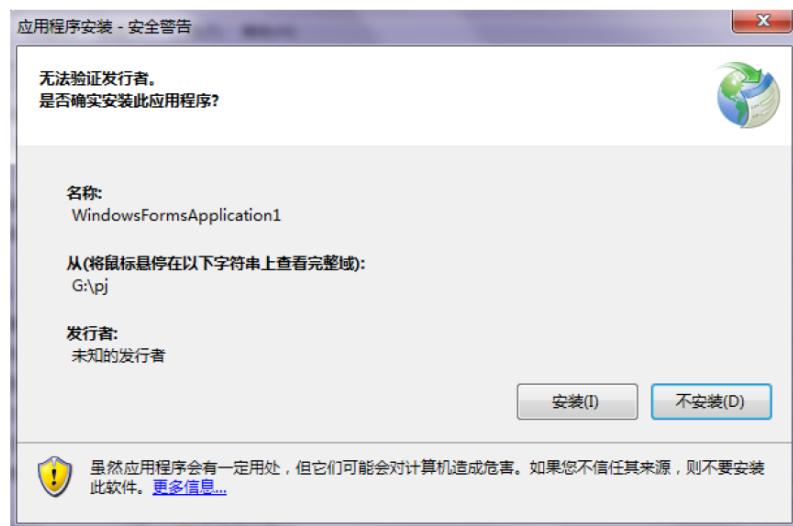


Figure 30: Visual Studio

setup.exe即为安装程序



安装后即可运行

Figure 31: Setup.exe

12 Feelings

感受就是感觉流水线部分一开始看起来挺复杂，后来对着书上HCL代码把每种情况都推导模拟测试了一遍，想通了流水线运作的机制，只要确定好顺序，比如W < -M,M < -E,E < -D,D < -F,就可以比较轻松地写出代码了。流水线核心部分400行一共写了一晚上，调试了一下午+一晚上，中间有很多非常细微的bug只能通过不断测试才能找到，一开始我通过了助教给的asum，但是测试我自己在ubuntu32位系统下生成的求fibonacci第n项的Y86码时，eax返回值一直是错的，最后通过单步调试和静态查错找到了漏洞。另外需要说的是即使单步调试找到了错误，最好再对着代码把整个框架在脑子里模拟一遍，模拟每一种情况下程序会给变量赋什么样的值，执行怎样的if语句，这一步是非常有用的，因为很多调试需要很久的错误可以通过仔细静态查错找出来。另外就是UI部分虽然没有什么技术含量，但是耗时非常多，细节繁琐，需要考虑的问题非常多。整个Y86程序流水线部分只写了一天，调试了一天，UI部分写了6天左右。从代码长度上看，Y86只占411行，UI占了大部分。总的来说，写完以后对流水线的认识谈不上突破天际，也有了很大的提升，比单纯看书本学到了更多实践知识。