东南大學
SOUTHEAST UNIVERSITY

计算机系统结构

实验报告

姓名：王亮

学号：09020328

东南大学计算机科学与工程学院

School of Computer Science & Engineering

Southeast University

20 22 年 6 月

实验一　流水线的冒险处理

一）实验目的

（1）加深对指令流水线基本概念的理解。

（2）理解 MIPS 指令流水线的实现方法，理解各段的功能和基本操作。

（3）加深对数据冒险、结构冒险的理解，理解这两类冒险对 CPU 性能的影响。

（4）理解数据冒险的处理方法，掌握转发（定向）技术的基本原理。

二）实验内容

（1）熟悉 MIPSsim 模拟器的使用方法。

（2）观察程序在流水线中的执行情况。

（3）观察和分析结构冒险对 CPU 性能的影响。

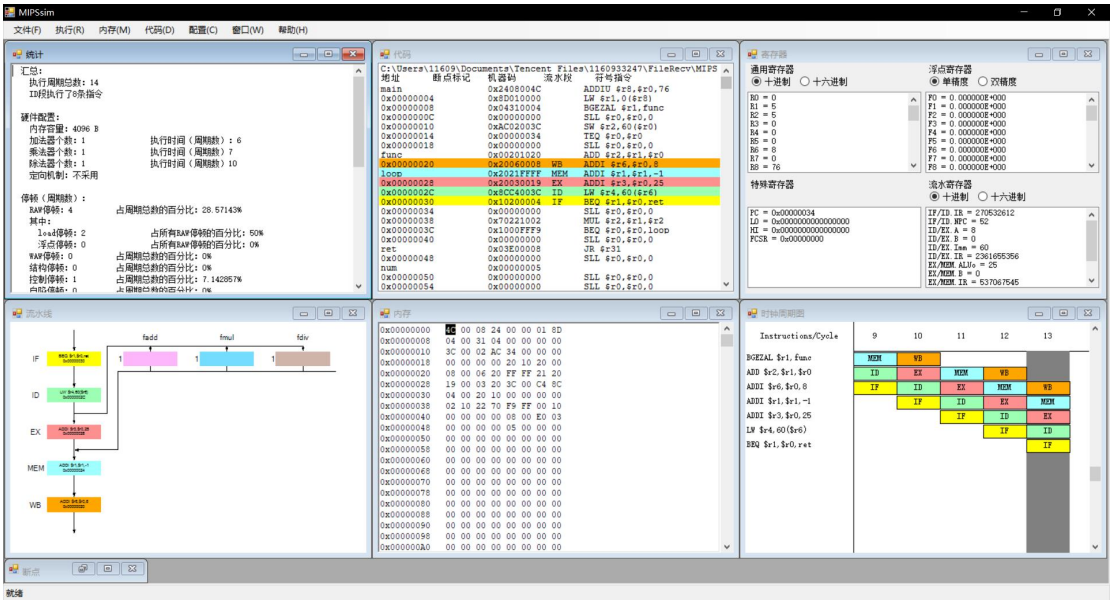
（4）观察数据冒险并用定向技术来减少停顿。

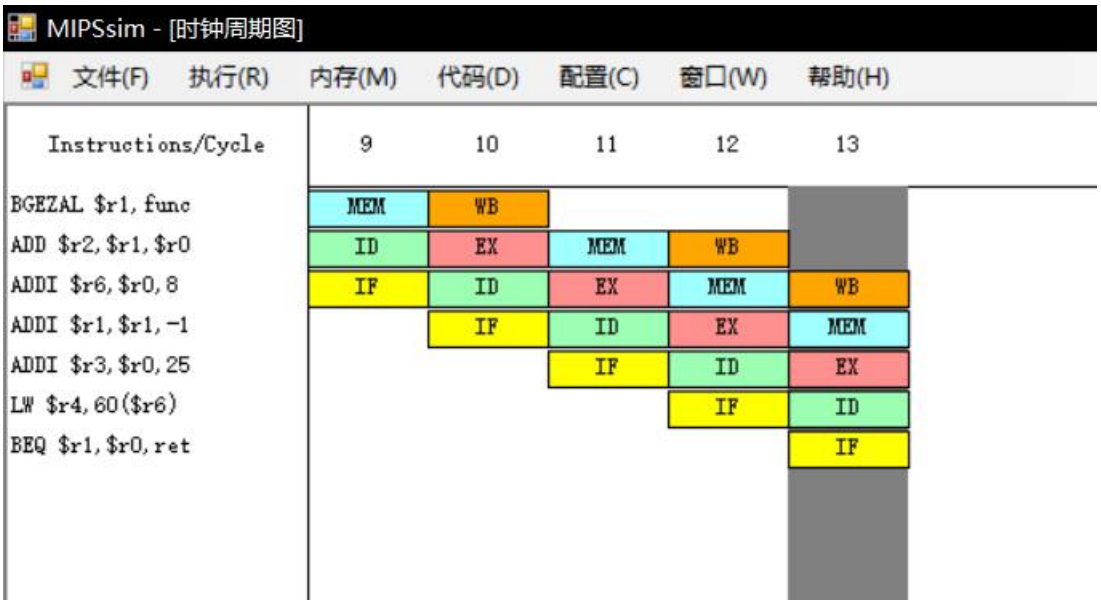
三）实验结果

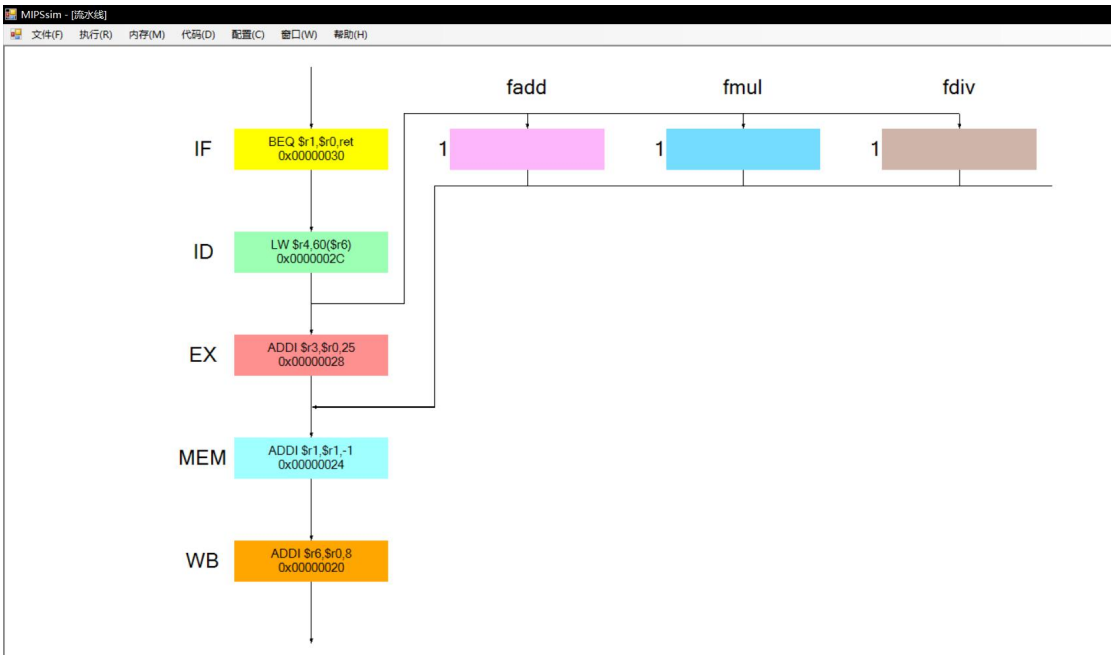
注：所有验证信息正确性操作，若无不符合之处，不单独指出。

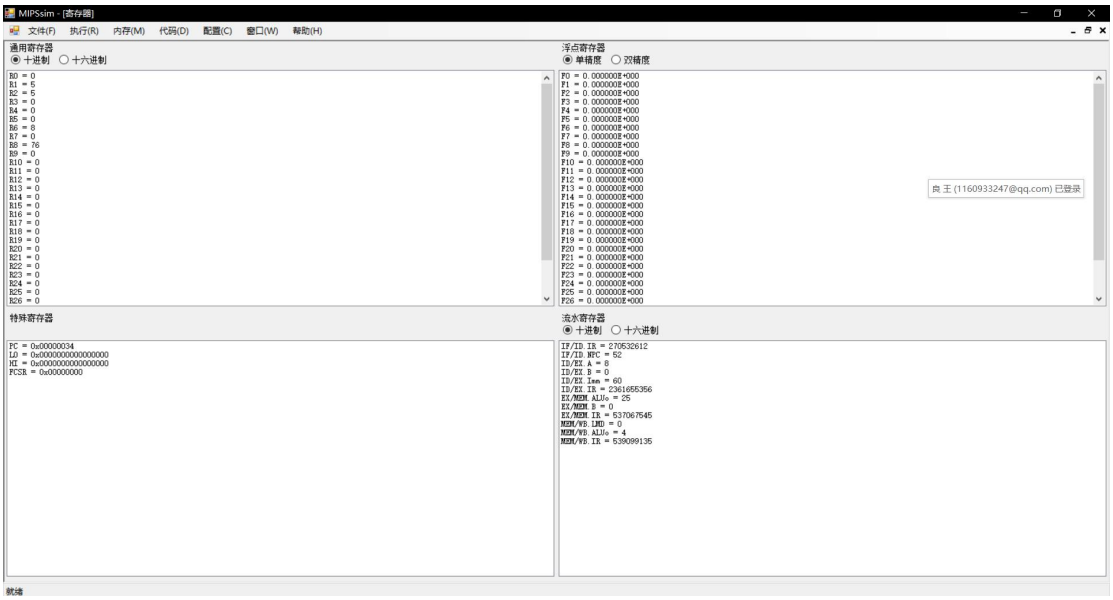
1、程序在流水线中的执行情况

1）实验数据记录




MIPSsim - [时钟周期图]
文件(F)
执行(R)
内存(M)
代码(D)
配置(C)
窗口(W)
帮助(H)
Instructions/Cycle
9
10
11
12
13
BGEZAL $r1, func
ADD $r2, $r1, $r0
ADDI $r6, $r0, 8
ADDI $r1, $r1, -1
ADDI $r3, $r0, 25
LW $r4, 60($r6)
BEQ $r1, $r0, ret
MEM
WB
ID
EX
IF

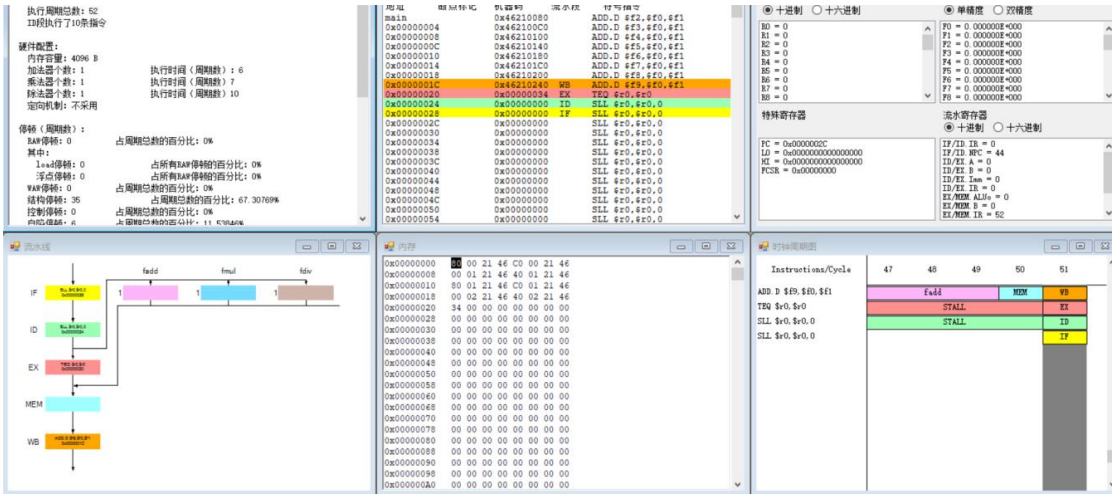

MIPSsim - [流水线]
文件(F)
执行(R)
内存(M)
代码(D)
配置(C)
窗口(W)
帮助(H)
fadd
fmul
fdiv
IF
ID
EX
MEM
WB
BEQ $r1,$r0,ret
0x00000030
LW $r4,60($r6)
0x0000002C
ADDI $r3,$r0,25
0x00000028
ADDI $r1,$r1,-1
0x00000024
ADDI $r6,$r0,8
0x00000020
1
1
1



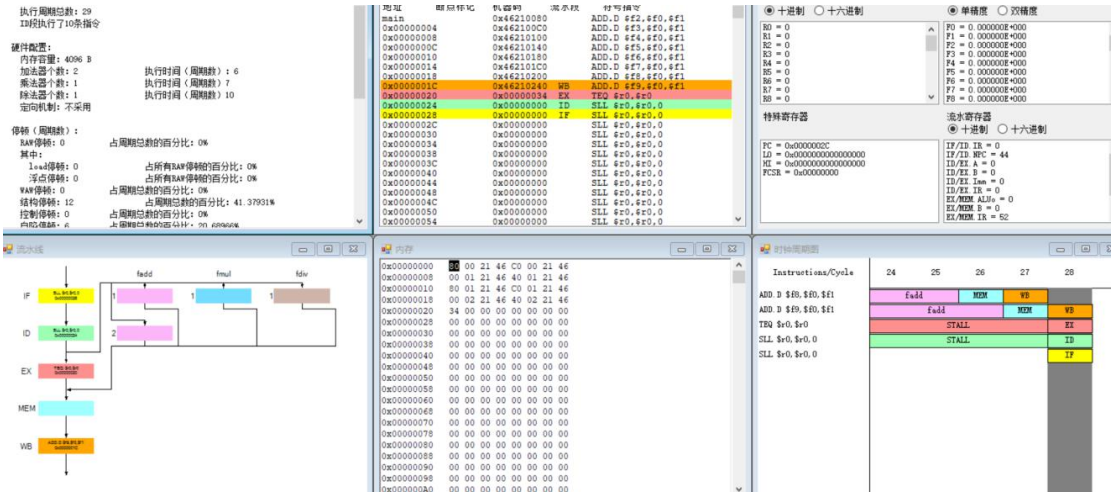
2）实验结果分析

2、结构冲突对 CPU 性能的影响

1）实验数据记录



$$S = \frac{T_s}{T_k} = \frac{82}{52} \approx 1.5769$$



$$S = \frac{T_s}{T_k} = \frac{82}{29} \approx 2.8276$$

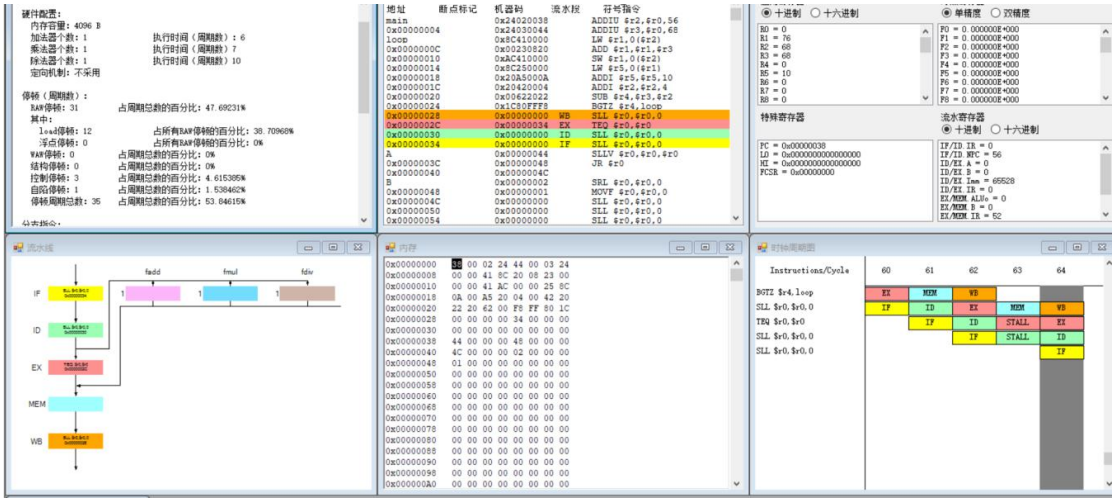
2）实验结果分析

$$\frac{\frac{52}{29} * 0.1 + 0.9 * 1}{1.05} \approx 1.02792$$

分析：将加速比乘以改进比例，与原来不变部分求和除以新的成本，计算性价比。

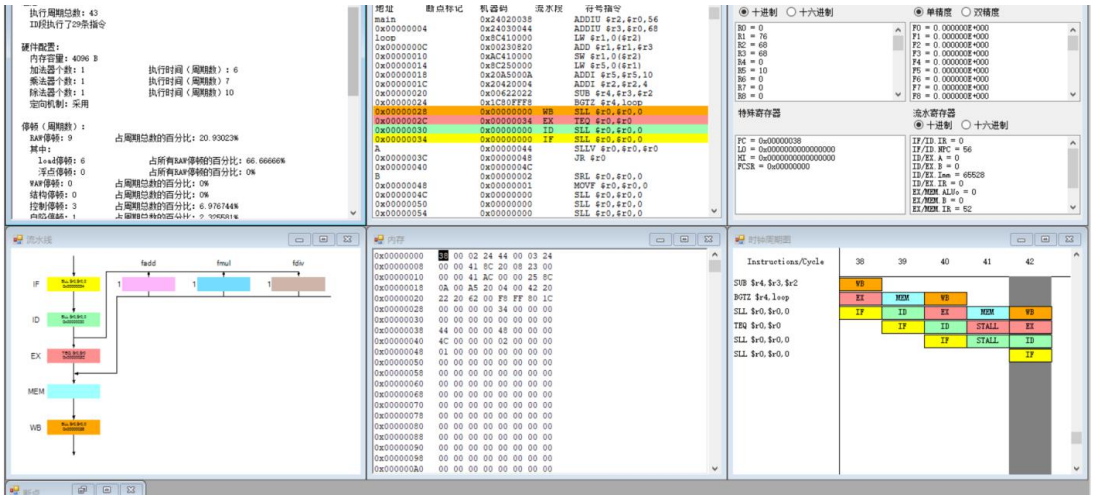
3、数据冲突及用定向技术来减少停顿

1）实验数据记录



RAW指令	发生时刻	停顿时长
LW	4	1
ADD	6	2
SW	9	2
ADDI	13	2
SUB	17	2
BGTZ	20	2
ADD	25	2
SW	28	2
ADDI	32	2
SUB	36	2
BGTZ	39	2
ADD	44	2
SW	47	2
ADDI	51	2
SUB	55	2
BGTZ	58	2

总停顿31拍



RAW指令	发生时刻	停顿时长
ADD	5	1
ADDI	10	1
SUB	13	1
ADD	18	1
ADDI	22	1
BGTZ	25	1
ADD	30	1
ADDI	34	1
BGTZ	37	1

总停顿9拍

2）实验结果分析

因为转发后 RAW 冒险不再出现（无停顿），此时可得：

对于阻塞法：

$$T = \frac{31}{65} \approx 0.4769$$

对于转发法：

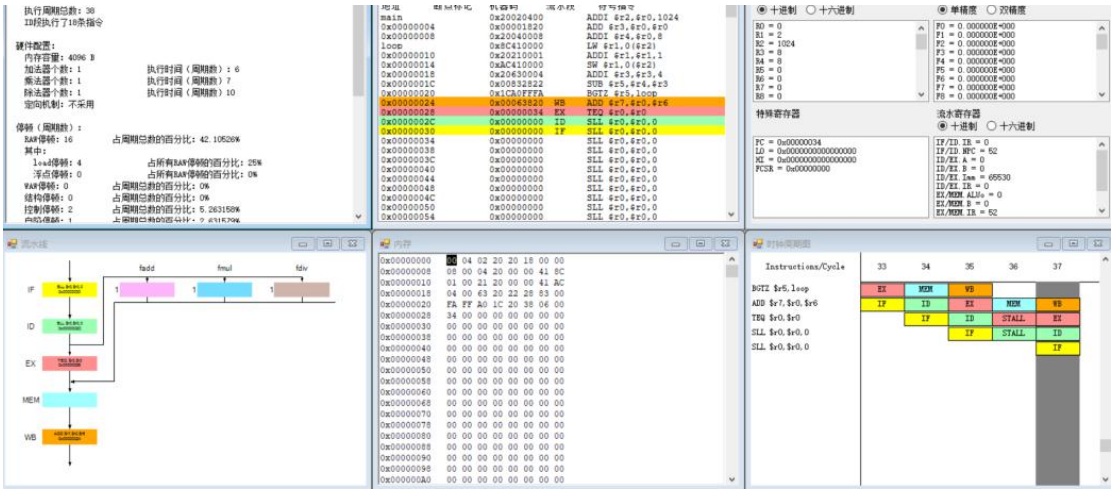
$$T = \frac{9}{43} \approx 0.2093$$

流水加速比（转发对阻塞）：

$$S = \frac{65}{43} \approx 1.5116$$

4、控制冒险的阻塞技术

1） 实验数据记录



2）实验结果分析

RAW指令	发生时刻	停顿时长
ADDI	6	2
SW	9	2
SUB	13	2
BGTZ	16	2
ADDI	21	2
SW	24	2
SUB	28	2
BGTZ	31	2

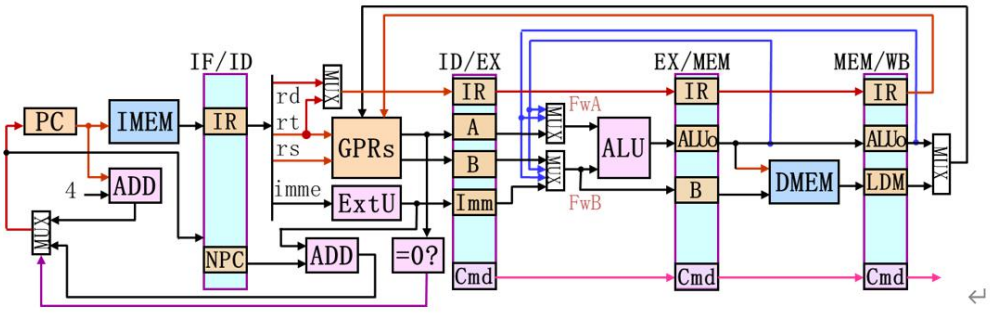
总停顿16拍

控制冒险指令	发生时刻	停顿时长
BGTZ	18	1
BGTZ	33	1

总停顿2拍

控制冒险 BGTZ 停顿一拍，因为其在 ID 段就可以能进行跳转的判断，同时可以借助 ID 段的加法器进行跳转地址的计算。

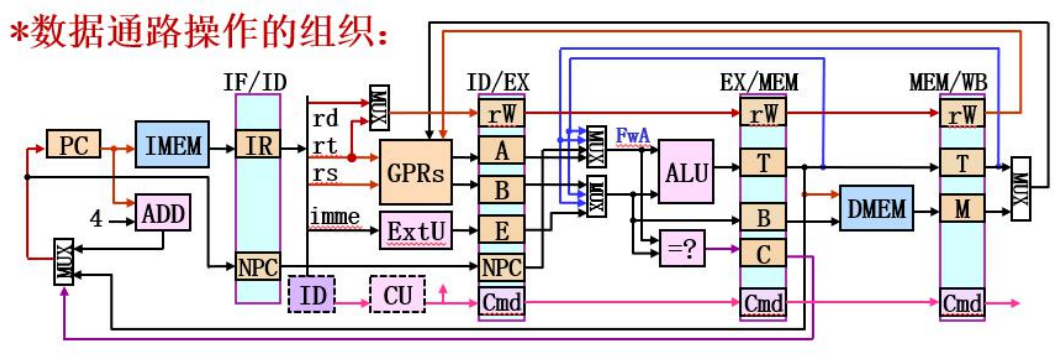
比如下图的 ADD 之后送至 PC 那边的 MUX



数据通路变化参考下图，加入 NPC 于 ID/EX 然后 NPC 连接到上面 MUX，再使得 ALUo（图里的 T）连接另一条输出到 PC 那边的 MUX 去，用于将计算出的地址送至 PC。

原本于 18 时刻发生的控制冒险停顿会变为 2 拍（计算地址 EX 完成，拖了 1 拍）

33 时刻不需要再计算地址，按上图=0？比较完就不需要再计算地址了，因为不会跳转，所以还是 1 拍。



实验二　Tomasulo 算法分析

一）实验目的

（1）加深对 Tomasulo 算法基本思想的理解。

（2）理解采用 Tomasulo 算法的浮点处理部件基本结构、保留站结构。

（3）掌握 Tomasulo 算法在指令流出、执行、写结果阶段的基本操作。

（4）掌握 Tomasulo 算法进行数据冒险处理的基本原理。

二）实验内容

（1）熟悉 Tomasulo 算法模拟器的使用方法。

（2）观察给定代码在 Tomasulo 算法流水线中的执行过程。

三）实验结果

1、加/减法器时延为 2 个时钟周期的实验结果

1）实验数据记录

结构冒险:

指令名称	发生时刻	停顿时长
L.D	CP_4	1
MUL.D	CP_8 -> CP_21 (CP_7取数据，不计入结构冒险)	14
SUB.D	CP_8	1

数据冒险：

指令名称	发生时刻	处理方法
MUL.D (RAW|LOAD-USE) : L.D(2)	CP_4 -> CP_7	转发(Load2)
SUB.D (RAW|LOAD-USE) : L.D(2)	CP_5- > CP_7	转发(Load2)
MUL.D (WAR) : DIV.D	CP_3	转发(Load2)
ADD.D (WAW) : SUB.D	CP_6	将F8中Qi更新为ADD2，覆盖ADD1中SUB写（相当于换名SUB写寄存器）

CP	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
L.D	IF	IS	EX	EX	EX	WB													
L.D		IF	IS	Z	Z	EX	EX	EX	WB										
MUL.D			IF	IS	Z	Z	Z	Z	EX	EX	EX	EX	EX	EX	EX	EX	WB		
SUB.D				IF	IS	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	EX	EX	WB
DIV.D					IF	IS	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	EX
ADD.D						IF	IS	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z

CP	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
DIV.D	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	WB		
ADD.D	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	Z	EX	EX	WB

加速比：37/31 = 1.193548387096774

2）实验结果分析

2、加/减法器时延为 3 个时钟周期的实验结果

1）实验数据记录

加减法时延3：

执行顺序	执行时刻	原因
L.D(1)	CP_2	正常情况
L.D(2)	CP_3	注：第三周期计算完地址，第四周期被卡访存，第五周期继续
DIV.D	CP_6	MUL.D因数据冒险停顿，乘除法器空闲，而该指令操作数已就绪
ADD.D	CP_7	SUB.D因数据冒险停顿，乘除法器空闲，而该指令操作数已就绪
SUB.D	CP_10	由于数据冒险，需要等待到CP_7，才可获取L.D(2)产生的源操作数。由于结构冒险，CP_8 -> CP_9 需要等待ADD.D 使用完加减法部件。
MUL.D	CP_22	由于数据冒险，需要等待到CP_7，才可获取L.D(2)产生的源操作数。由于结构冒险，CP_8 -> CP_21 需要等待DIV.D 使用完乘除法部件。

2）实验结果分析

实验三　Cache 性能分析

一）实验目的

（1）加深对 Cache 的基本概念、基本结构以及工作原理的理解。

（2）掌握 Cache 容量、相联度、块大小对 Cache 性能的影响。

（3）掌握降低 Cache 缺失率的各种方法以及其对提高 Cache 性能的好处。

（4）理解 LRU 与随机算法的基本思想以及它们对 Cache 性能的影响。

二）实验内容

（1）掌握 mycache 模拟器的使用方法。

（2）掌握 Cache 容量、相联度、块大小、替换算法对 Cache 性能的影响。

三）实验结果

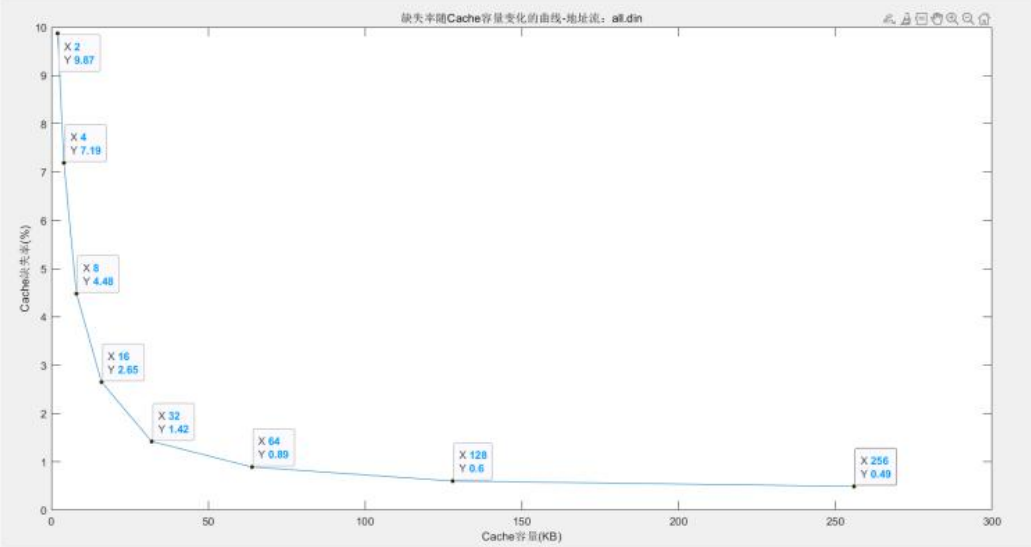
1、Cache 容量对缺失率的影响

1）实验数据记录

表2.1 不同容量时的Cache缺失率

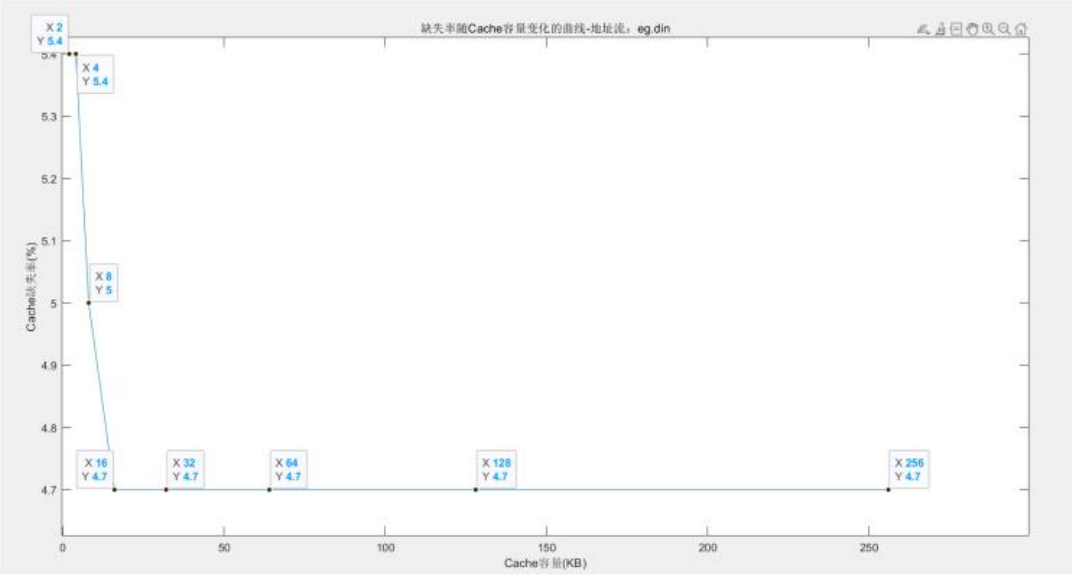
Cache容量(KB)	2	4	8	16	32	64	128	256
缺失率	9.87%	7.19%	4.48%	2.65%	1.42%	0.89%	0.60%	0.49%

地址流文件名： all.din



2）实验结果分析

结论：单独提升 Cache 容量对减小缺失率的影响具有边际效应，每次 Cache 容量翻倍后提升是逐渐 减少的，如图可以看出斜率在逐渐减少，后续已经快趋于 0。可见在只考虑 Cache 容量情况下，Cache 容 量的值应保持一定水平后不必提升，一味提升其容量并不意味着一定带来很好的性能提升，反而可能因 成本因素得不偿失。事实上，如果换成 eg.din 的地址流文件，Cache 容量在 16KB 后就再无提升。



2、相联度对缺失率的影响

1）实验数据记录

表2.2 容量为64KB、不同相联度时的Cache缺失率

相联度	1	2	4	8	16	32
缺失率	1.97%	1.15%	0.99%	0.93%	0.92%	0.91%

地址流文件名：cc1.din

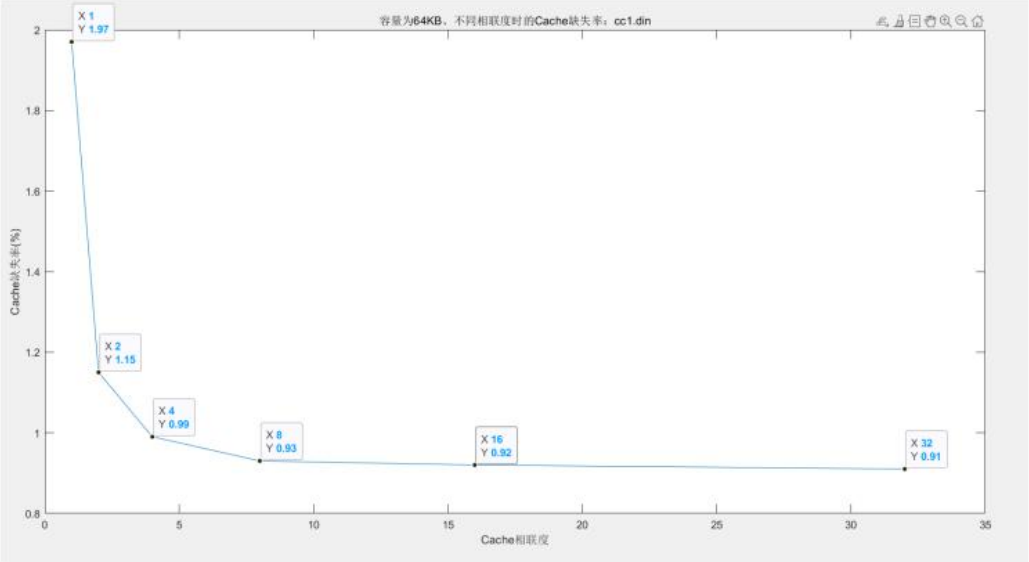
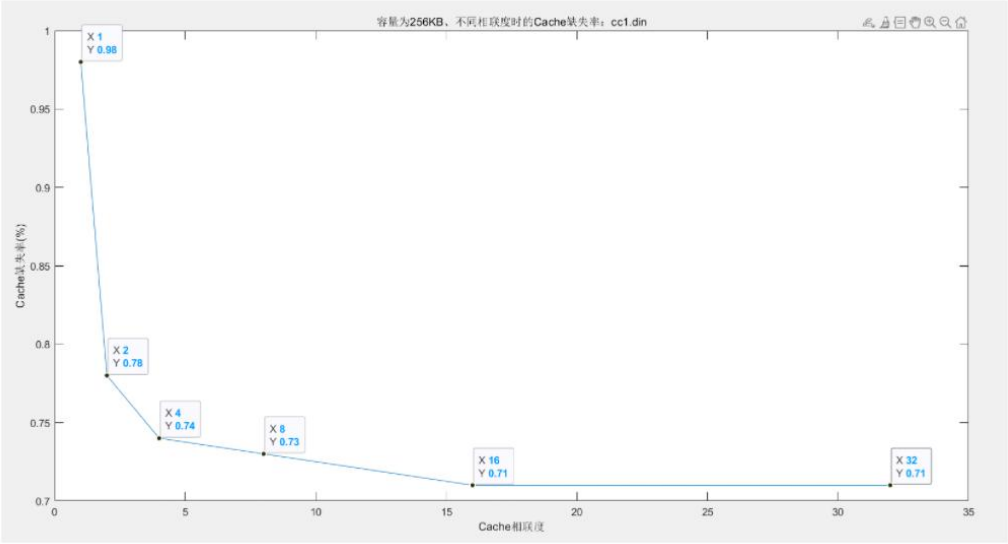


表2.3 容量为256KB、不同相联度时的Cache缺失率

相联度	1	2	4	8	16	32
缺失率	0.98%	0.78%	0.74%	0.73%	0.71%	0.71%

地址流文件名：cc1.din



2）实验结果分析

结论：由**"表2.2 容量为64KB、不同相联度时的Cache缺失率"**，不难理解课件上如①往上的相联度与8相联度的约等关系，进而推论出与全相联的约等关系。

当然在all.din里也有相同结论。

相联度	1	2	4	8	16	32
缺失率	0.89%	0.53%	0.47%	0.45%	0.44%	0.44%

地址流文件名：all.din

由**"表2.3 容量为256KB、不同相联度时的Cache缺失率"**，不清楚是不是巧合，256KB的直接映像与64KB的4路组相连差不多，尽管此时与②中前提$S_{Cache} \leq 128KB$并不相符。

① S_{Cache}不变，$F_{8\text{路}} \approx F_{\text{全相联}}$

② $S_{\text{Cache}} \leqslant 128\text{KB}$，$S_{\text{Cache}}$的$F_{1\text{路}} \approx S_{\text{Cache}}/2$的$F_{2\text{路}}$

可见，相联度的提升同样面临边际效应，应当选取适当的相联度，不过不同于Cache容量的是，可以推出这一具体的值可能不应大于8路。在Cache容量足够大的情况下，可以适当考虑降低相联度以为提升其他方面的性能让步。

3、Cache 块大小对缺失率的影响

1）实验数据记录

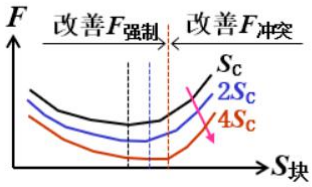
表2.4 各种块大小时的Cache缺失率

----------	Cache容量（KB）				
块大小（B）	**2**	**8**	**32**	**128**	**512**
16	12.02%	5.79%	1.86%	0.95%	0.71%
32	9.87%	4.48%	1.42%	0.60%	0.42%
64	9.36%	4.03%	1.20%	0.43%	0.27%
128	10.49%	4.60%	1.08%	0.35%	0.20%
256	13.45%	5.35%	1.19%	0.34%	0.16%

地址流文件名：all.din

2）实验结果分析

分析：在Cache容量不够时，将块大小提升的过大可能适得其反，如果能保证Cache容量足够大，适当提升块大小还是可以很好地帮助减小Cache缺失率。



例1：表中$T_{传输}$=2CLK/16B，$T_{命中}$=1CLK，选择各容量的$S_{块}$

$S_{块}$及$T_{缺失}$ S_C		4KB	16KB	64KB	256KB
16B	80+2	8.57%	3.94%	2.04%	1.09%
32B	80+4	7.24%	2.87%	1.35%	0.70%
64B	80+8	7.00%	2.64%	1.06%	0.51%
128B	80+16	7.78%	2.77%	1.02%	0.49%
256B	80+32	9.51%	3.29%	1.15%	0.49%

思考②：例1中80→40，结果会变化吗？

解：$T_A=T_{命中}+F\cdot T_{缺失}$，

$T_{4K/16}=1+8.57\%*82=8.027CLK$，

$T_{4K/64}=1+7.00\%*88=7.160CLK$，

得 4KB时$S_{块}=32B$，

16KB/64KB/256KB时$S_{块}=64B$

结果—$S_{块}$尽量大，值取决于下级MEM的延迟与带宽

└→保持$F\cdot T_{缺失}$不变→┴→$T_{调入}$较小时$S_{块}$较大

4、替换算法对缺失率的影响

1）实验数据记录

表2.5 LRU和随机替换算法时的Cache缺失率

Cache 容量	相联度					
----------	2路		4路		8路	
----------	LRU算法	随机算法	LRU算法	随机算法	LRU算法	随机算法
16KB	1.71%	2.24%	1.33%	2.41%	1.21%	2.84%
64KB	0.53%	0.68%	0.47%	0.72%	0.45%	0.83%
256KB	0.38%	0.39%	0.36%	0.37%	0.36%	0.36%
1MB	0.35%	0.35%	0.35%	0.35%	0.35%	0.35%

地址流文件名：all.din

2）实验结果分析

分析：可以看出在Cache容量达到一定程度（当然也与块大小有关系），比如256KB，无论相联度如何，LRU与RAND间差距都显得比较小了，在1MB时更是没有任何区别。

在16KB，64KB时还可以看出LRU算法相对于RAND算法的优势，且LRU算法很明确随相联度提升会提升命中率，而RAND随机算法则是不确定的。

结合课件，由16KB，64KB对应LRU算法，可以看出随着n（相联度）增大，LRU算法的H（命中率）确实在增大（对应缺失率减小），而RAND算法与n并无明确关系，虽然16KB，64KB时由随n增大而缺失率有所增大，但256KB时却在减小。

*常见算法：

	状态的个数	状态更新的时机	牺牲行的选择	对H的影响
RAND	1个随机数/Cache	块替换时，产生随机数	随机数对应的行	H随机
FIFO	1个计数值/行	块调入时，更新n个值	(n个)值最大的行	H随机
LRU	1个计数值/行	块访问时，更新n个值	(n个)值最大的行	H随n增大
注：n—组相联的路数，即候选行的个数；计数值—刚调入/访问的行清零				

实验四　OpenMP 编程

一）实验目的

（1）加深对并行计算及其开发的理解。

（2）掌握基于 OpenMP 进行并行程序设计的方法。

二）实验内容

（1）掌握并行划分和计算的方法。

（2）使用 OpenMP 编写并执行用积分方法求 π 的并行程序。

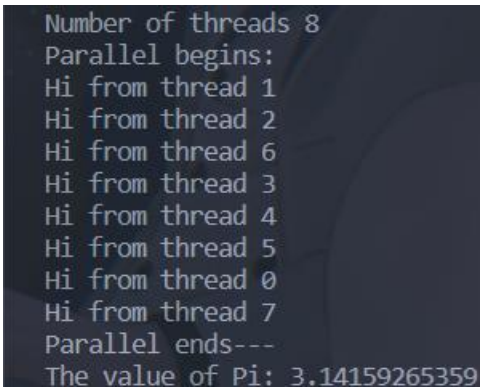
三）实验结果

1、源程序/实验过程记录

```
#include <iostream>
#include <cstdio>
#include "omp.h"
using namespace std;

constexpr long long N = 100000000 * 8;
constexpr int threadNum = 8;
double total = 0;

double calFunc(double x) {
    return 4 / ((x * x) + 1);
}

double task(int tid, long long N) {
    double calPi = 0;
```

```
    for (long long i = tid; i < N; i += threadNum) {
        calPi += calFunc((i + 0.5) / N);
    }

    return calPi;
}

int main() {

    int tid;

    printf("Number of threads %d\n", threadNum);

    cout << "Parallel begins:\n";

    #pragma omp parallel private(tid) num_threads(threadNum)
    {
        tid = omp_get_thread_num();
        printf("Hi from thread %d\n", tid);

        total += task(tid, N);
    }

    cout << "Parallel ends---\n";

    total *= 1.0 / N;

    printf("The value of Pi: %.11lf", total);
    return 0;
}
```

2、运行结果贴图



3、编程与调试心得（遇到的问题和解决的办法，以及获得的收获）

软件层面极高层次的封装使得多线程编程十分的容易，尤其是不需要考虑线程同步的，数据之间毫无相关性的任务，多线程的使用十分简单，openmp 使得这就像编写一般单线程代码一样简单，极大降低了多线程程序的设计难度。