Concurrent Y86 PIPE Processor 实验报告

**16307130194 陈中钰**

16级 计算机科学技术学院

**Contents**

[1 串行版本 2](#_Toc512891072)

[2 并行设计 4](#_Toc512891073)

[3 并行实现 7](#_Toc512891074)

[4 运行样例 9](#_Toc512891075)

[5 实验感想 11](#_Toc512891076)

# 串行版本

（详细阐述可以在附件：串行Y86实验报告-陈中钰.pdf 中查看）

* 1. 规格

1. 指令集

halt, nop, rrmovl, irmovl, rmmovl, mrmovl, OPl(addl, subl, andl, xorl), jXX(jmp, jle, jl, je, jne, jge, jg), **cmovXX(rrmovl, cmovle, cmovl, cmove, cmovne, cmovge, cmovg),** call, ret, pushl, popl, **iaddl, leave**

**（所有指令结构与CS:APP书中格式一致）**

1. reg: 32bit\*16

memory: 134,217,728B

Language: VS Community 2017 + C++

**库: pthread.h + semaphore.h**

**测试代码：23份.yo文件（其中22份为CMU官网提供）**

* 1. 文件结构

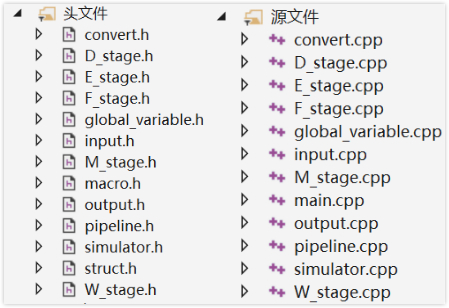


Figure 1 文件结构

1. macro.h定义宏，global\_variable定义全局变量，struct.h定义结构体
2. x\_stage定义每个stage的运行函数
3. convert定义转换函数
4. input定义读入文件方式，output制表
5. pipeline定义流水线操作
6. simulator定义小黑框UI
   1. 运行逻辑
7. main()调用simulator()，运行小黑框的界面。
8. simulator()调用input()读入文件，调用run()执行程序。
9. run()在while(1)中调用posedge\_clock()，产生时钟上升沿，运行程序，直至HALT或其他error
10. 运行一次posedge\_clock()即为一个cycle。在一个cycle中，首先运行sequential\_update更新时序部件CC、memory、reg，接着为了能于并行达到相同的结果，按照W、M、E、D、F阶段的顺序去串行执行（每个stage中先是更新寄存器，再更新逻辑电路），然后再调用control\_update()更新控制信号stall和bubble。最后调用table()输出表格。

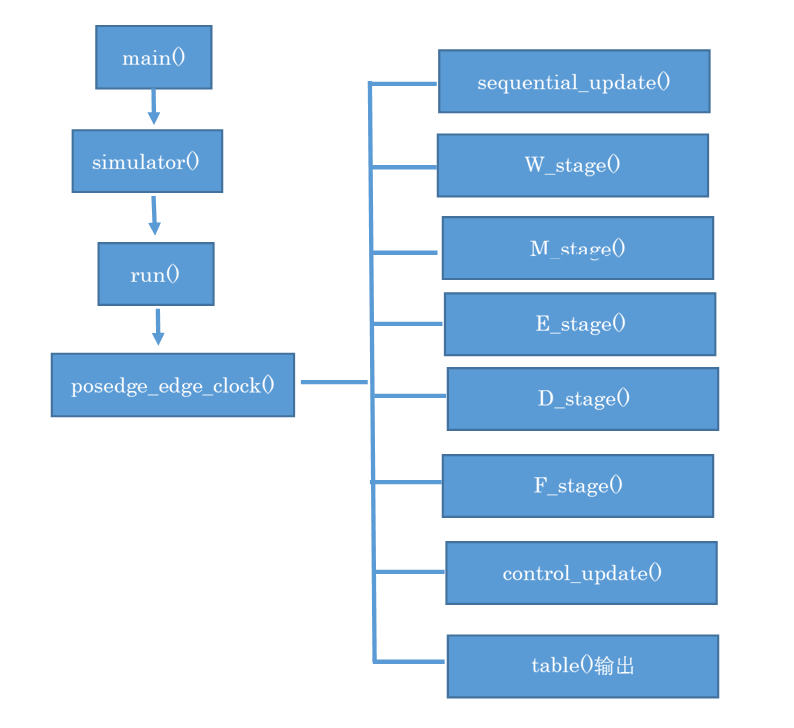


Figure 2 整体运行结构

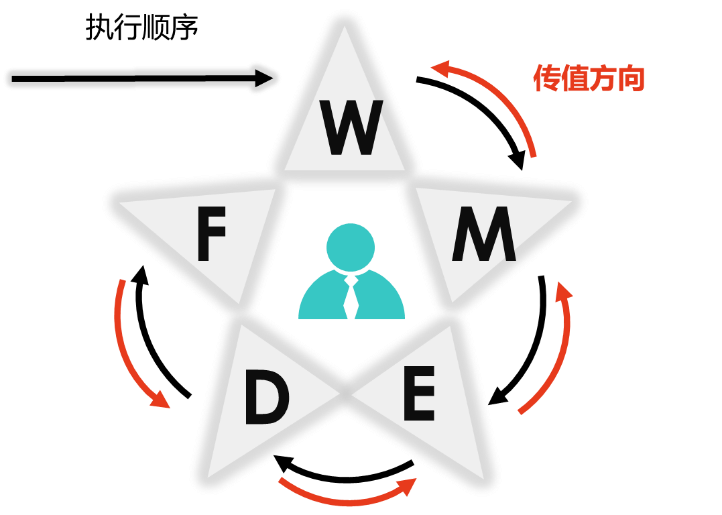


Figure 3 每个周期内的运行逻辑

# 并行设计

* 1. 线程划分

为了实现并行程序，首先要更改原来的程序结构，拆分为多个可以并行运行的板块，而每个板块即可作为一个线程。**为了能更大程度地实现并行，需要把板块分的尽量小，但同时也需要考虑实现复杂性（约束条件太多难以实现），以及线程切换损耗（当线程数超过核数时，运行效率不会增加，甚至还可能下降），因此线程也不可以太多。**

由于时序电路和逻辑电路是明显可以分开的，于是把每个stage中的时序更新和逻辑更新部分分开；另外sequential\_update()中的dmem、CC、reg更新是明显可以并行的，所以也分开。最终考虑到实现的复杂性以及线程损耗，就不再分割下去。**形成dmem、CC、reg、F\_reg（F stage的寄存器更新）、D\_reg、E\_reg、M\_reg、W\_reg、F\_logic（F stage的逻辑电路更新）、D\_logic、E\_logic、M\_logic、W\_logic、control\_update（stall和bubble的更新），共14个线程。**

* 1. 顺序约束
     1. 原因

在真正的硬件中，每个时序部件无时无刻都在根据输入进行计算，并输出对应的结果，因此无论谁先执行，最终只需要时钟周期足够长，每个部件的输入输出最终都会稳定下来，并且是正确值。但是，我们要实现并行的时候，显然是不能通过while(1)来不断进行赋值来更新值的，**故仍然只能采取1次赋值的方式去更新值**，所以**为了保证正确性，某些值的更新之间是存在顺序关系的**，所以这14个线程必须要添加顺序约束。

* + 1. **顺序约束规则：**

1. **如果线程A使用到某个当前周期更新后的变量b，而b是在线程B中更新的，那么在这个周期内，B必须在A之前执行完毕。只有这样，线程A执行时才能取到正确的b值。这样可以记为：B->A，表示B必须在A之前执行完毕**
2. **如果上述用到的b值是前一个周期更新的值，那么A必须在B之前执行。可以记为A->B**
   * 1. 存在的顺序关系
3. 显然，每个stage的寄存器更新必须在逻辑更新之前（如F\_reg->F\_logic）、下一个stage的寄存器更新必须在当前周期前（如D\_reg->F\_logic）
4. 由于数据转发，存在M\_reg->F\_logic, W\_reg->F\_logic, E\_logic->D\_logic, M\_logic->D\_logic, W\_reg->D\_logic等顺序
5. control\_update必须要在D\_logic、E\_logic、W\_logic之后执行
6. 由于control的更新是cycle的末尾，因此要control执行结束后，5个阶段的寄存器以及CC、dmem、reg寄存器才可以进行下一个cycle的更新
7. M stage中用到dmem，E stage中用到CC，D stage中用到regfile
8. reg更新中用到上一个周期的W寄存器值，所以reg必须在W\_reg之前执行
9. 其他可以被优化掉的顺序约束...
   1. 约束优化
10. 在硬件中当时钟上升沿到达时，最先更新的是时序部件，因此时序更新的线程可以认为是周期的开始。此外，除了control的更新以外，可以发现F\_logic、W\_logic的更新也是末尾，因此需要添加这两个结束之后时序更新才能开始的约束，才能保证当前周期能在下一次上升沿前完整执行。但是这样子**会出现F\_reg->F\_logic、F\_logic->F\_reg的死锁情况**。因此**人为添加F\_logic->control、W\_logic->control的约束，人为使得control是唯一的末尾，并且避免了添加会造成死锁的F\_logic->F\_reg等约束**
11. 由于control是周期末尾，需要添加control->8个时序更新线程的顺序约束
12. 存在A->……->C、A->C情况时，由于顺序约束的传递性，A->C可以省略
    1. 最终实现

以下是最终实现的完整顺序约束，每个A->B代表A必须在B之前完成，而相同颜色、虚实的箭头是同一类的顺序约束。

**dmem**

**CC**

**reg**

**F\_reg**

**D\_reg**

**E\_reg**

**M\_reg**

**W\_reg**

**F\_logic**

**D\_logic**

**E\_logic**

**M\_logic**

**W\_logic**

**control**

Figure 4 完整顺序约束

# 并行实现

* 1. **约束实现方式**

1. #include <semaphore.h>
2. 假设有void \*A(void \*vargp)和void \*B(void \*vargp)两个线程，而且存在A->B的顺序关系，即A必须在B之前运行，可以通过一个原子变量sem\_t A2B（2代表的是->的意思，也就是表达A->B，A必须在B前运行，在我的代码实现中遵守这一规则）来实现。
3. 把int sem\_wait(sem\_t \*sem);包装为void P(sem\_t \*sem), 另外sem\_post()包装为V()，sem\_init包装为I()。（在pipeline.cpp中实现）

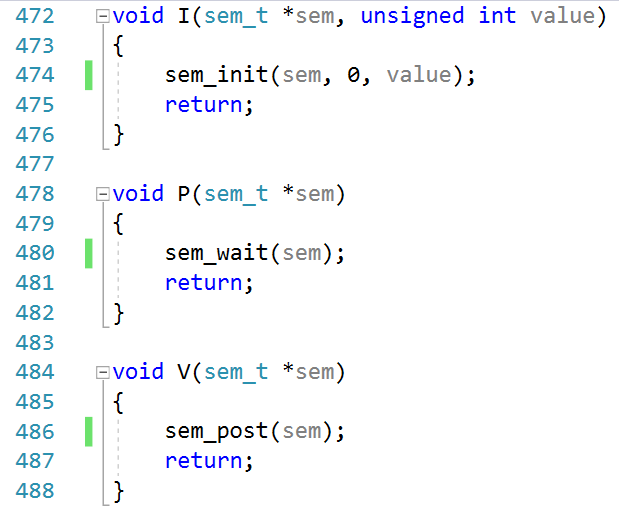


Figure 5 包装函数

1. 如下图，利用PV锁对sem\_t A2B进行操作，使得只有当A线程执行结束后，才对A2B进行加1操作，才会激活B线程运行，否则B线程被P()挂起，等待A线程结束。这样就能实现A->B的顺序约束。

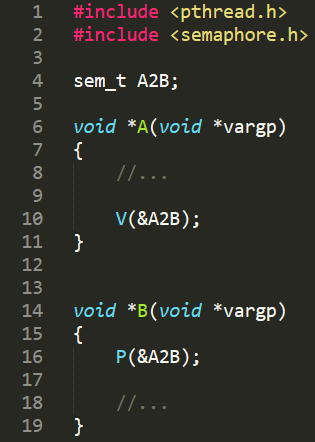


Figure 6 约束实现方式

1. 全部约束

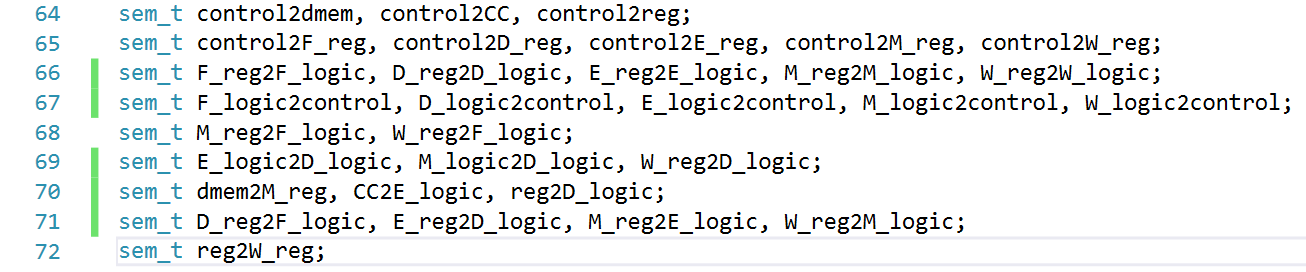


Figure 7 全部顺序约束所用到的sem\_t变量

* 1. **线程实现方式**

1. #include <pthread.h>
2. 把原来运行程序的run()函数改为concurrent\_run()函数
3. 添加全局数组变量pthread\_t tid[13]，用于储存各个线程的tid
4. 在concurrent\_run()函数中，调用13次pthread\_create()函数，创建13个peer thread，而concurrent\_run()本身作为main thread，运行末尾的control\_update()线程，13个peer thread运行其他的线程，一共有14个线程
5. 把另外13个板块分别打包成函数，再更改为线程函数

F\_stage()

F\_value\_update()

F\_logic\_update()

\*F\_value\_update()

\*F\_logic\_update()

Figure 8 线程函数形成方式

1. 每个线程都进入while(1)的循环中，并按照上文的顺序约束添加好全部约束。其中main thread中也进入循环，控制每个周期的开始，**并以control update作为每个周期的结尾**，最后调用table()输出表格，然后进入下一个周期。
2. **最后当Stat出现error时，程序结束了，接着main thread会跳出while循环，并调用pthread\_cancel()来结束另外13个线程，最后自己也return结束。**

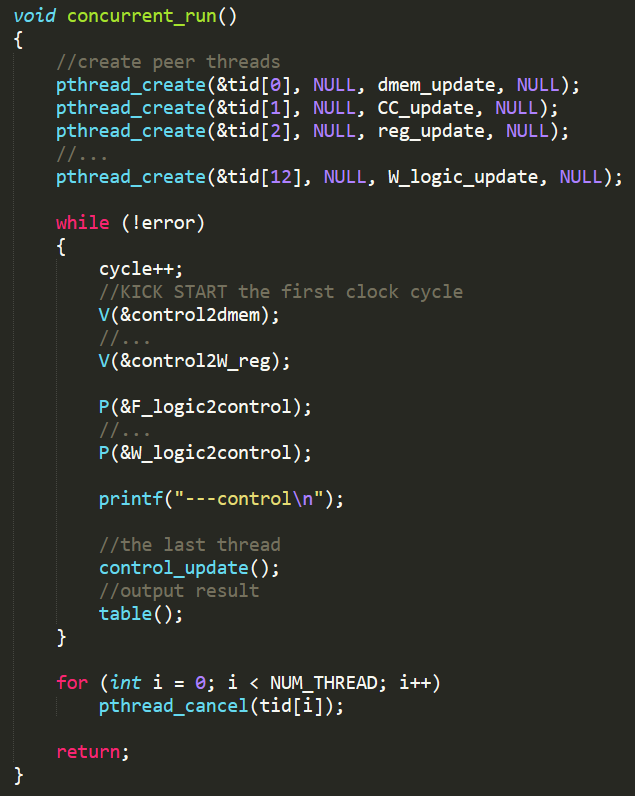


Figure 9 main thread部分代码示例

# 运行样例

* 1. 并行验证

在每个线程内运行的时候，会打印出自己的名字。比如\*CC\_update()在运行中会打印printf(“---CC\n”);并以此大致地显示出每个周期的线程执行顺序。运行asumi.yow文件，可以发现有多种运行顺序出现，且每个周期不可以预测顺序，因此基本达到了并行的效果。

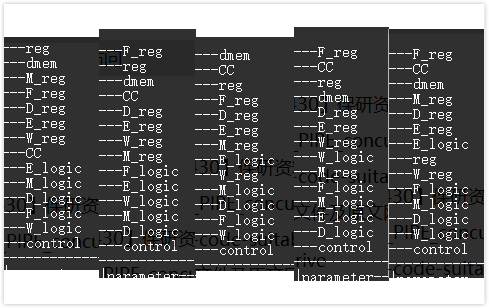


Figure 10 部分线程运行顺序截图

* 1. 正确性验证

运行asumi.yo文件，可见%eax结果为0xABCD，因此正确。此外还对**一共23份.yo文件**进行了正确性验证，**不仅保证了并行的效果，而且最终结果均正确**。

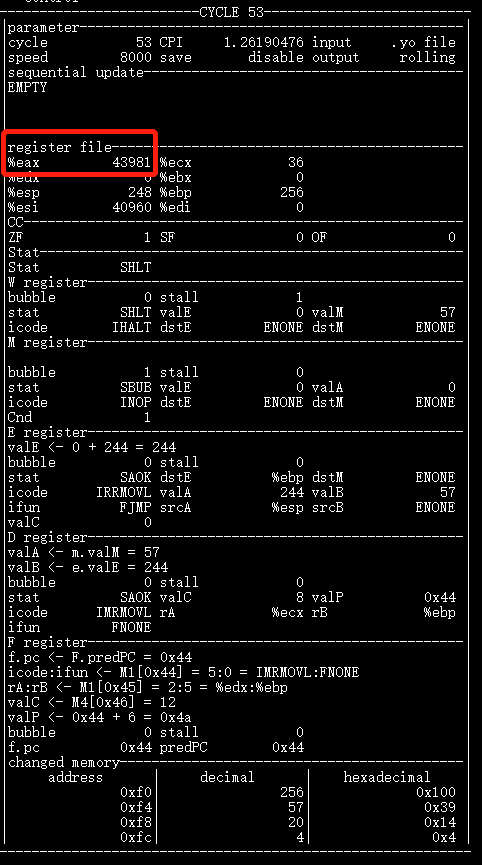


Figure 11 asumi.yo程序的最后一个cycle结果

# 实验感想

* 1. 学习到并发编程，对并发编程技术有了较深的理解；
  2. 学习到三种并发编程方式：基于进程、基于事件、基于线程，了解到各自的优劣，深刻体会到基于线程的并发编程的极大优势；
  3. 学会使用信号量、PV锁来进行基于线程的并发编程；
  4. 学会了对竞争、死锁等线程安全问题进行分析，能在并发编程中避免这些问题；
  5. 学会分析并行程序的性能，了解到核数对并发编程性能的限制。