# 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

（详细阐述可以在附件：串行 Y86 实验报告-陈中钰.pdf 中查看）

## 1.1 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）**

2. 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）**

## 1.2 文件结构

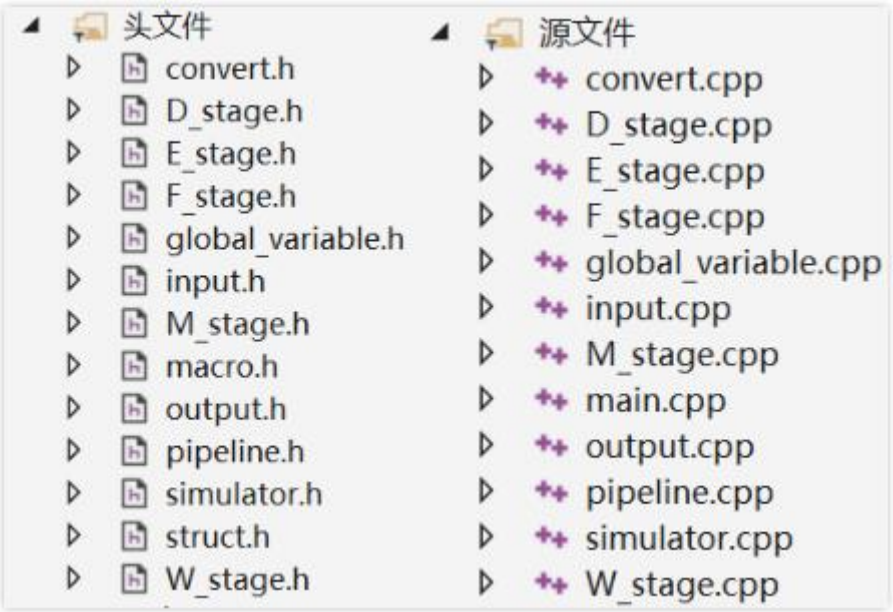


Figure 1 文件结构

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

## 1.3 运行逻辑

1. main()调用 simulator()，运行小黑框的界面。

2. simulator()调用 input()读入文件，调用 run()执行程序。
3. 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
4. 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。

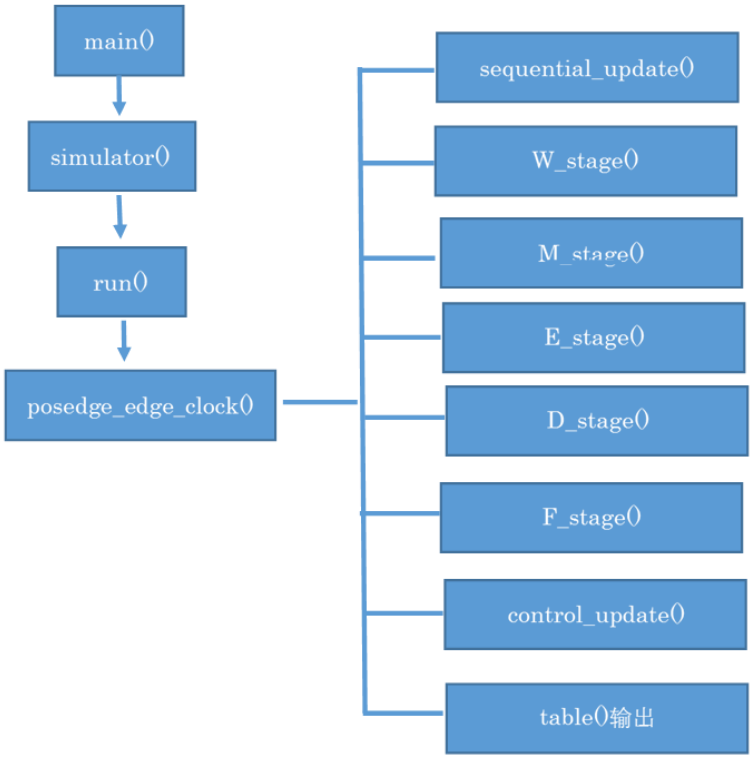


Figure 2 整体运行结构

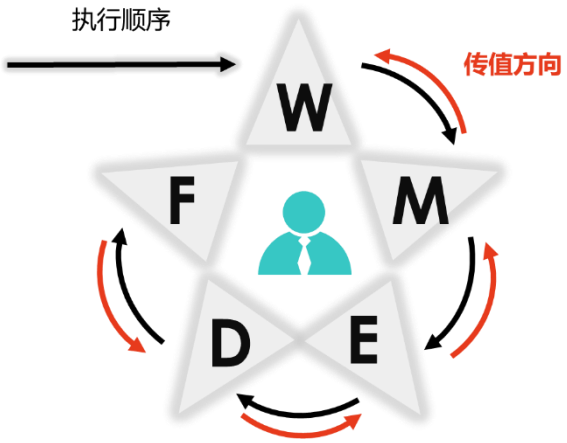


Figure 3 每个周期内的运行逻辑

# 2 并行设计

## 2.1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。**

## 2.2顺序约束

### 2.2.1 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。

### 2.2.2 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**

### 2.2.3 存在的顺序关系

1. 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）
2. 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
3. control_update 必须要在 D_logic、E_logic、W_logic 之后执行
4. 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
5. M stage 中用到 dmem，E stage 中用到 CC，D stage 中用到 regfile
6. 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
7. 其他可以被优化掉的顺序约束...

## 2.3约束优化

1. 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**

2. 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
3. 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

## 2.4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。

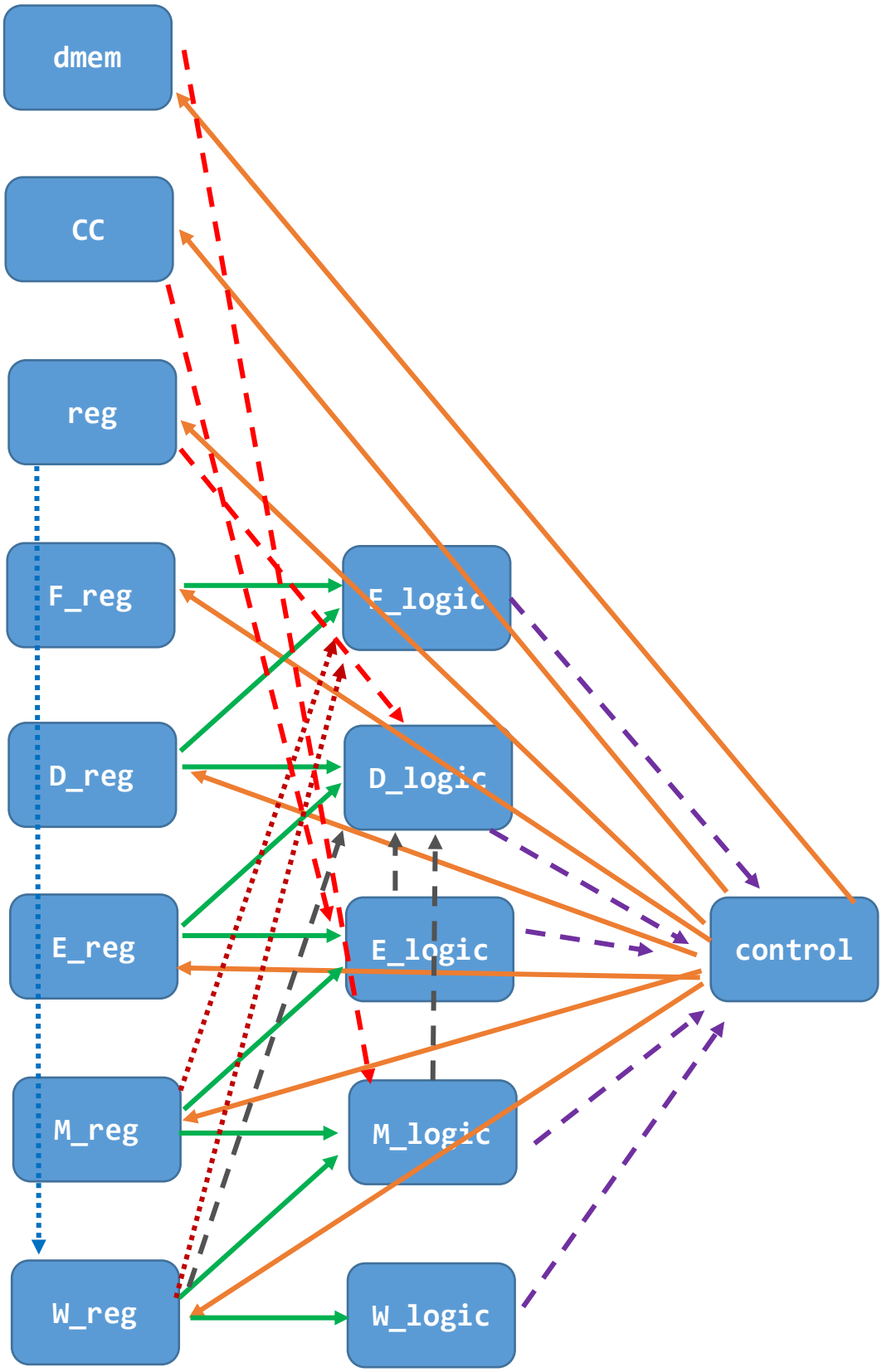


Figure 4 完整顺序约束

# 3 并行实现

## 3.1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）

```
472  void I(sem_t *sem, unsigned int value)
473  {
474      sem_init(sem, 0, value);
475      return;
476  }
477
478  void P(sem_t *sem)
479  {
480      sem_wait(sem);
481      return;
482  }
483
484  void V(sem_t *sem)
485  {
486      sem_post(sem);
487      return;
488  }
```

Figure 5 包装函数

4. 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。

```
1   #include <pthread.h>
2   #include <semaphore.h>
3
4   sem_t A2B;
5
6   void *A(void *vargp)
7   {
8       //...
9
10      V(&A2B);
11  }
12
13
14  void *B(void *vargp)
15  {
16      P(&A2B);
17
18      //...
19  }
```

Figure 6 约束实现方式

5. 全部约束

```
64    sem_t control2dmem, control2CC, control2reg;
65    sem_t control2F_reg, control2D_reg, control2E_reg, control2M_reg, control2W_reg;
66    sem_t F_reg2F_logic, D_reg2D_logic, E_reg2E_logic, M_reg2M_logic, W_reg2W_logic;
67    sem_t F_logic2control, D_logic2control, E_logic2control, M_logic2control, W_logic2control;
68    sem_t M_reg2F_logic, W_reg2F_logic;
69    sem_t E_logic2D_logic, M_logic2D_logic, W_reg2D_logic;
70    sem_t dmem2M_reg, CC2E_logic, reg2D_logic;
71    sem_t D_reg2F_logic, E_reg2D_logic, M_reg2E_logic, W_reg2M_logic;
72    sem_t reg2W_reg;
```

Figure 7 全部顺序约束所用到的 sem_t 变量

## 3.2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

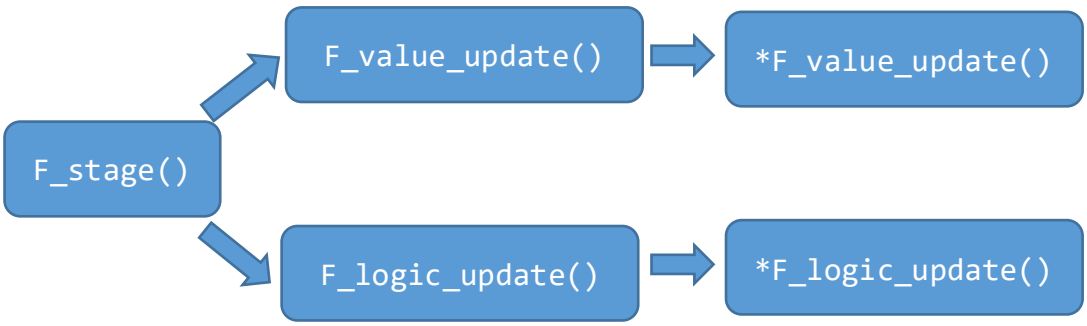


Figure 8 线程函数形成方式

6. 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。
7. **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。**

```c
void concurrent_run()
{
    //create peer threads
    pthread_create(&tid[0], NULL, dmem_update, NULL);
    pthread_create(&tid[1], NULL, CC_update, NULL);
    pthread_create(&tid[2], NULL, reg_update, NULL);
    //...
    pthread_create(&tid[12], NULL, W_logic_update, NULL);

    while (!error)
    {
        cycle++;
        //KICK START the first clock cycle
        V(&control2dmem);
        //...
        V(&control2W_reg);

        P(&F_logic2control);
        //...
        P(&W_logic2control);

        printf("---control\n");

        //the last thread
        control_update();
        //output result
        table();
    }

    for (int i = 0; i < NUM_THREAD; i++)
        pthread_cancel(tid[i]);

    return;
}
```

Figure 9 main thread 部分代码示例

# 4 运行样例

## 4.1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。

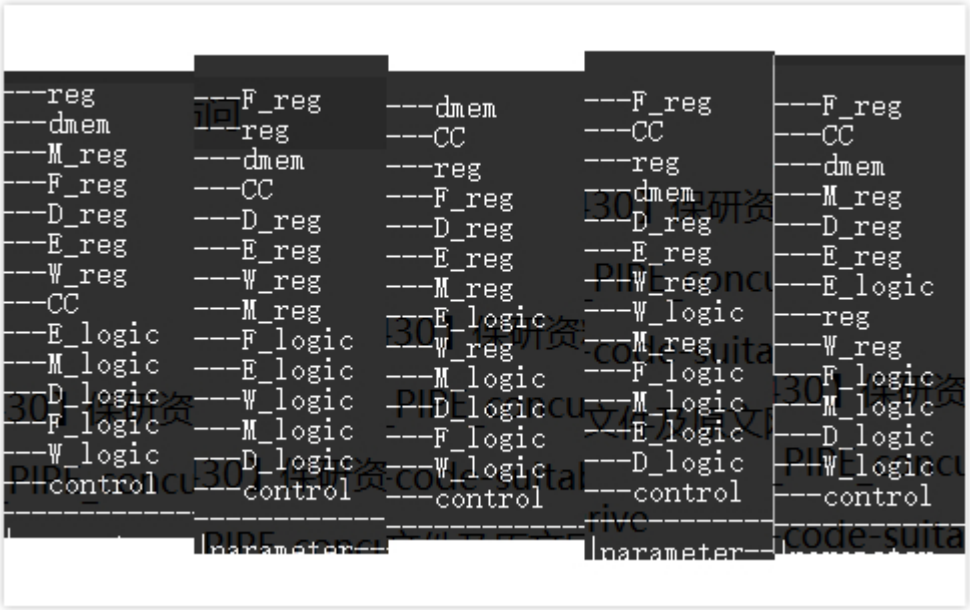


Figure 10 部分线程运行顺序截图

4.2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**。

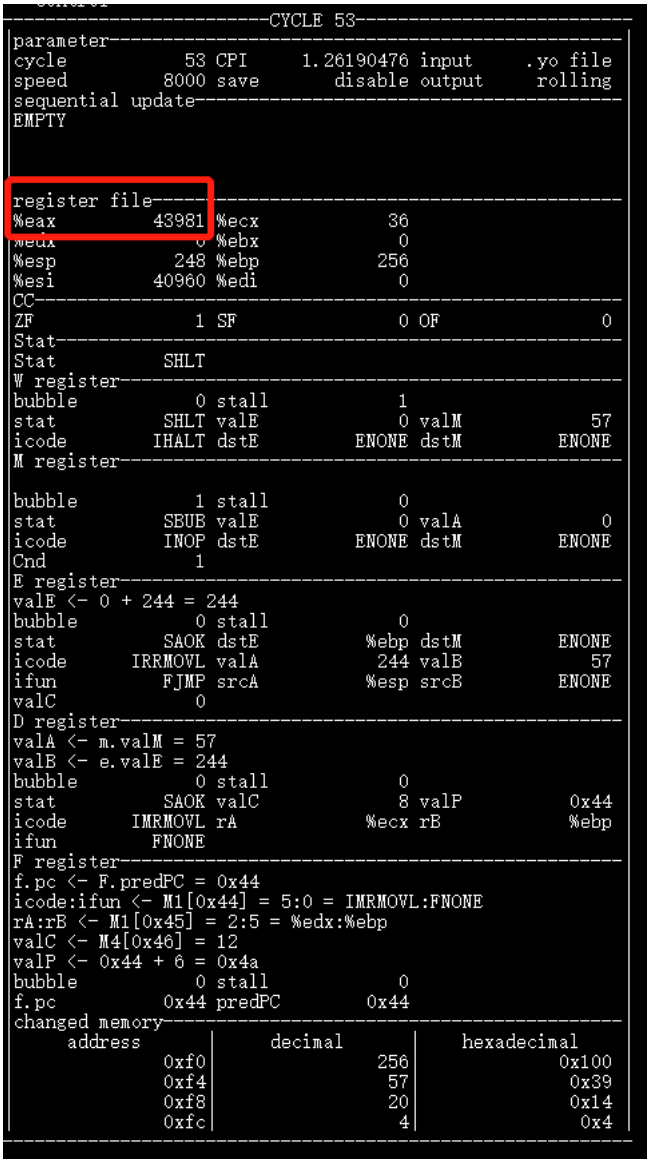


Figure 11 asumi.yo 程序的最后一个 cycle 结果

# 5 实验感想

5.1学习到并发编程，对并发编程技术有了较深的理解；

5.2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；

5.3学会使用信号量、PV 锁来进行基于线程的并发编程；

5.4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；

5.5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。