数字逻辑与处理器基础实验
32 位 MIPS 处理器设计

孙伟艺*
白钦博†
王敏虎 ‡

2017 年 7 月 29 日

*无 52 2015011010
†无 52 2015010996
‡无 52 2015011003

1 实验目的

- 熟悉现代处理器的基本工作原理
- 掌握单周期和流水线处理器的设计方法

2 设计方案

2.1 ALU

(孙伟艺)

2.2 单周期数据通路

(孙伟艺)

2.3 流水线数据通路

流水线的数据通路由单周期改进而来，主要的改变是在每个流水级之间增加段间寄存器，另外对于各种冒险加上对应冒险检测电路。如图1。

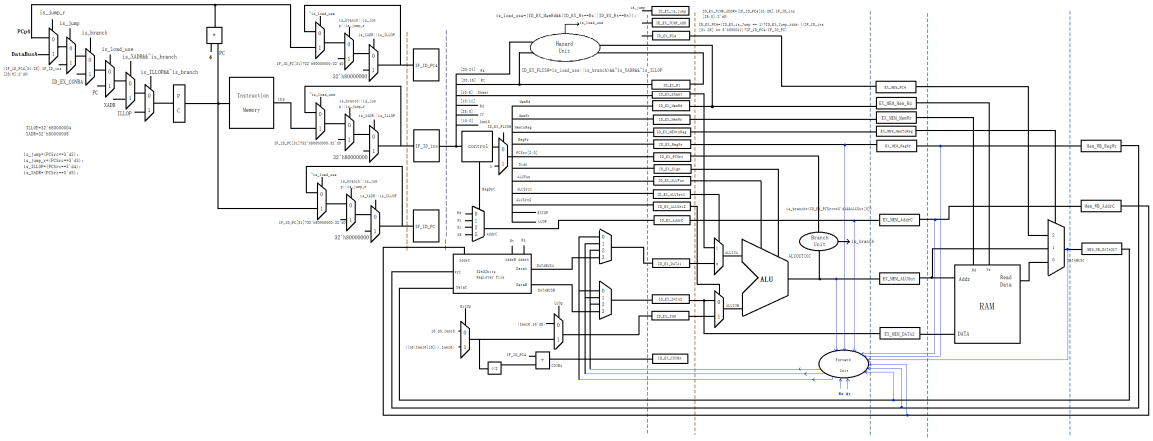

图 1: 流水线数据通路

2.3.1 对单周期电路的直接调整

1. 单周期电路中，负责写单周期的同学将 LUOp 的控制机信号省掉了，将 LUI 的运算结果直接最为最后一个多路复用器到寄存器堆的输入，并且修改了 MemToReg，以来完成此项修改，在流水线中，为了尽量减少段间寄存器的数量，恢复了 LUOp，将 Lui 生成的 32 位立即数在 ID 段就完成计算，与`ID_EX_IMM` 可共用一个寄存器，修改后电路与指导书上参考电路一致，具体可查看流水线数据通路

2. 为了完成中断与 branch 同时出现的问题，修改了 control.v 中 PCSrc，具体参考后面的关键问题，其中有详细解释

2.3.2 段间寄存器

1. IF_ID 段

   该段用来保存指令的 PC 值和具体指令，故设置如下三个寄存器

```
reg [31:0] IF_ID_PC;//保存当前PC，用作中断的返回值
reg [31:0] IF_ID_PC4;//保存当前PC+4，用来作jal的写入值
reg [31:0] IF_ID_ins;//用来保存当前指令，用作ID段译码
```

2. ID_EX 段

   该段用来保存此条指令产生的各种控制信号、送入 ALU 的数据，以及其他部分信号，具体为

```
reg [31:0] ID_EX_CONBA;//保存当前指令的条件分支地址，用来作PC的条件分支跳转值
reg [31:0] ID_EX_DATA_1;//保存当前指令第一个ALU的部分输入，用来EX段计算
reg [31:0] ID_EX_DATA_2; //保存当前指令第二个ALU的部分输入，用来EX段计算
reg [31:0] ID_EX_PC4; //保存当前PC+4，用来作jal的写入值并向后传递
reg [31:0] ID_EX_IMM;//保存当前指令立即数，用来在EX段计算
reg [4:0] ID_EX_Rt;//保存当前指令的Rt值，用来在EX段判断load_use
reg [4:0] ID_EX_shamt;//保存当前指令的位移值，用来在EX段计算
reg [2:0] ID_EX_PCSrc;//保存当前指令的PCSrc，用来向后传递
reg [4:0] ID_EX_AddrC;//保存当前指令AddrC，用来说明当前指令写入哪个寄存器并向后
    传递
reg [1:0] ID_EX_MemToReg;//保存当前指令的MemToReg，用来说明选择哪个信号作为寄存
    器的写入并向后传递
reg [5:0] ID_EX_ALUFun;//保存当前指令的ALUFun，用来在EX段ALU的执行
reg ID_EX_RegWr//保存当前指令的RegWr，用来说明当前指令是否写寄存器并向后传递
reg ID_EX_ALUSrc1//保存当前指令的ALUSrc1，用来说明EX段的ALU选择哪个作为第一个输
    入
reg ID_EX_ALUsrc2//保存当前指令的ALUSrc2，用来说明EX段的ALU选择哪个作为第二个输
    入
reg ID_EX_Sign//保存当前指令的sign，用作EX段ALU的执行
reg ID_EX_MemWr//保存的当前指令的MemWr，用来说明当前指令是否写RAM并向后传递
reg ID_EX_MemRd//保存的当前指令的MemRd，用来说明当前指令是否读RAM并向后传递
```

3. EX_MEM 段

   该段主要保存 ALU 计算的结果及后续需要的控制信号，具体为

```
reg [31:0] EX_MEM_PC4;// 保存当前指令PC+4，用来作jal的写入值
reg [31:0] EX_MEM_ALUOUT;//保存当前指令的ALU计算结果，用来判断branch以及作为当前
    指令的寄存器写入值
reg [31:0] EX_MEM_DATA2;//用来保存当前指令的RAM的writedata
reg [4:0] EX_MEM_AddrC;// 保存当前指令AddrC，用来说明当前指令写入哪个寄存器并向
    后传递
reg [1:0] EX_MEM_MemToReg; //保存当前指令的MemToReg，用来说明选择哪个信号作为寄
    存器的写入并向后传递
reg EX_MEM_RegWr保存当前指令的RegWr，用来说明当前指令是否写寄存器并向后传递
reg EX_MEM_MemWr//保存的当前指令的MemWr，用来说明当前指令是否写RAM
reg EX_MEM_MemRd;//保存的当前指令的MemRd，用来说明当前指令是否读RAM
```

4. MEM_WB 段

   该段保存最后 WB 段需要的少数几个信号，具体为

```
reg [31:0] MEM_WB_DATAOUT;//用来保存最终写入寄存器的值
reg [4:0] MEM_WB_AddrC;//用来保存最终写入寄存器编号
reg MEM_WB_RegWr;//用来保存最终是否写入寄存器
```

2.3.3 冒险检测电路

1. Forward unit

   我们组采用的转发电路与理论课上讲的略有不同，观察我的数据通路可知，转发单元转发数据输入的两个多路选择器在 ID 段，而理论课讲的此多路选择器在 EX 段，这样做的原因是在后面优化主频的时候，发现关键路径在通过 ALU，也就是说关键路径在 EX 段，若能减少 EX 段的 slack 则主频必定能得到优化，故将这两个多路复用器移到 ID 段，另外不将 ALUSrc1，ALUSrc2 控制的两个多路复用器一起移到 ID 段的原因是，经过这样的尝试发现主频不升反降，关键路径发生了不可预测的变化，所以未做此改动。

   转发的具体电路分为三级转发，因为将转发部分移到了 ID 段，所有的转发条件用到的段间寄存器比理论课都要向前移一段，具体来说

   (a) 第一级转发

   转发条件为

```
(EX_MEM_RegWr&&(EX_MEM_AddrC!=0)&&(EX_MEM_AddrC==Rs)&&(ID_EX_AddrC!=Rs||~
    ID_EX_RegWr))
```

   转发数据为 DataBusC

   (b) 第二级转发

```
(ID_EX_RegWr&&(ID_EX_AddrC!=0)&&(ID_EX_AddrC==Rs))
```

转发数据为 ALUOut

(c) 第三级转发

按照理论课所讲的只存在两级转发，因为在理论课中寄存器是先写后读的，故不存在 WB 段需要转发的问题，但上网查阅后发现，如果实现的时候真的分别用上升沿和下降沿读写寄存器，会出现不稳定的问题，故采用三级转发

转发条件为：

```
(MEM_WB_RegWr&&(MEM_WB_AddrC!=0)&&(MEM_WB_AddrC==Rs))
```

转发数据为MEM_WB_DATAOUT

2. Hazard Unit

Load_use 的检测在 EX 段实现，实现方法和理论课完全相同，具体为

```
is_load_use=(ID_EX_MemRd&&(ID_EX_Rt==Rs||ID_EX_Rt==Rt));
```

即 EX 段指令需要读 RAM 并且写入的寄存器号与当前指令读取寄存器号相同时,is_load_uses 为真，当其为真时，IF_ID 段间寄存器不变，PC 值也不发生改变，ID_EX 段的控制信号全部清零，从而实现 stall 一个周期

3. Branch Unit

因为本次 CPU 涉及到的 branch 指令比较多，提前判断需要增加的电路较多，故未采用提前判断，而在 EX 段判断，判断是否为 branch 的方法比较简单，具体为

```
is_branch=(ID_EX_PCSrc==3'd1&&ALUOut[0])
```

ID_EX_PCSrc==1 说明当前 EX 段指令为 branch 类型指令，ALUOUT[0]=1 说明需要跳转, 当此信号为真，IF_ID 段间寄存器全部清零，PC 值加载为跳转地址，ID_EX 段的控制信号全部清零

4. Jump Unit 检测跳转指令的具体电路非常简单，未在数据通路中具体标出，而将其判断语句附在左下角。具体来讲，在 ID 段判断，判断方法为

```
is_jump=(PCSrc==3'd2);
is_jump_r=(PCSrc==3'd3);
```

5. Others 类似跳转指令，还有关于中断和异常的判断，方法与跳转相同，具体为

```
   is_ILLOP=(PCSrc==3'd4);
   is_XADR=(PCSrc==3'd5);
```

2.3.4 细节

(a) 段间寄存器清零

因为 PC 的第 31 位表示内核态，所以在讲段间寄存器的 PC 值清零时，应该根据当前内核态，来确定清零为 32’h80000000 还是 32’h00000000 以保证内核态不变。

(b) branch 与中断的问题

因为 branch 在 EX 阶段判断，中断在 ID 段判断，故有以下四种情况

i. 分支指令此时在 ID 段
因为中断结束后返回地址为IF_ID_PC 而不是IF_ID_PC4，所以该 branch 指令在中断之后会被正确执行，故无需处理

ii. 分支指令此时在 EX 段
此处我们的处理方法是将分支的优先级高于中断，先执行完分支，再进入中断，故在 EX 检测到 branch 时不中断

iii. 分支指令此时在 MEM 段
因为中断指令返回的地址为IF_ID_PC，故此时新取出的地址还没有进入段间寄存器，故修改控制电路的 PCSrc，当 ID 段处理的信号为 nop 时，不进行中断，从而使新地址正确进入段间寄存器

iv. 分支指令此时在 WB 段
如上所说返回地址已正确保存，故不存在问题

(c) 关于 jump 与中断的问题

i. 跳转指令此时在 ID 段
同 branch 指令，jump 指令会在中断结束后正确执行

ii. 跳转指令此时在 EX 段
为了防止在中断结束之后进入错误的地址,故增设寄存器ID_EX_is_jump, ID_EX_JUMP_Addr 前者赋值为is_jump, 来判断 EX 段的指令是否为跳转指令后者赋值为 ID 段计算出来的跳转地址，当这种情况发生时，向ID_EX_PC4 中写入ID_EX_JUMP_Addr，从而保证中断结束之后回到正确的指令地址

iii. 跳转指令此时在 MEM 段
同 branch 指令，此时正确地址已经进入IF_ID_PC4，故不存在问题

(d) 关于 jal 的问题

因为中断返回地址为IF_ID_PC，不进行 +4，故在不发生中断的时候如果遇到 jal 指令写入IF_ID_PC，则会导致死循环，故当指令为 JAL 时ID_EX_PC4 赋值为IF_ID_PC4

此外，现在回想一下当时的设计思路，原本ID_EX_PC4 应该直接赋值为IF_ID_PC4，但是在后来修改的时候，为了解决中断的问题ID_EX_PC4 的含义发生了变化，现在来看应该为中断返回地址另外设计一个寄存器来保存，这样设计思路将更为清晰。

2.4 外设

本次实验需要使用 LED 灯、七段数码管、串口等外设参与 CPU 工作，为 CPU 提供运算所需要的数据并将 CPU 的运算结果表示出来。外设不是 CPU 的硬件组成部分，对于 CPU，外设被看作内存中的一个普通地址，CPU 不需要了解外设工作的具体细节，只需要执行程序员编写好的程序，将数据存入对应的地址即可，后续工作应当由外设电路独立完成。

本次实验中，地址0x40000000 到0x40000020 被用于外设地址。在 CPU 的连接中，这些地址被连接到专门的电路而非内存中。

地址如下表被分配给外设。

地址	功能
0x40000000	定时器 TH
0x40000004	定时器 TL
0x40000008	定时器控制 TCON
0x4000000C	LED
0x40000010	Switch
0x40000014	七段数码管
0x40000018	UART 发送数据
0x4000001C	UART 接收数据
0x40000020	串口状态

以下为实验要求编写的各外设的说明

2.4.1 LED 灯

LED 灯是本次实验中比较简单的外设。在 CPU 访问0x4000000C 地址时将对应位数赋给相应管脚即可。

2.4.2 七段数码管

本次实验使用四个七段数码管，由于要求使用软件译码，硬件部分较为简单，七段数码管使用 12 位进行控制，前 4 位表示当前点亮的数码管位置，后八位表示七段数码管各管脚的电平，与 LED 灯类似的是，当 CPU 访问0x40000014 时，将对应位数赋给对应接线即可。复杂的译码和控制部分将在汇编程序中进行。

2.4.3 switch 开关

switch 开关是输入设备，不能被写入，当 CPU 试图读0x40000010 时，将对应连线上的电平返回。

2.4.4 串口

串口使用轮询方式编写，三位0x40000018,0x4000001C,0x40000020 控制，其中0x40000020 是串口状态位，其最后两位中的第一位用来标示是否收到新的数据，第二位用来表示目前串口的发送状态。当数据被写入0x40000018 时，串口将自动发送其后八位并修改串口发送状态，当0x4000001C 访问后，串口将修改串口状态位，将接收标志位置为低以等待下一个数据。

编写汇编程序时，应当首先访问串口状态位，确定串口已经接收到数据，再访问串口接收数据地址。使用轮询方式的一个问题即是，如果轮询时间过长，串口中的数据可能会丢失，但是在本实验的条件下，9600 波特率的串口接收一次数据的时间足以 CPU 完成上万个时钟周期的运算，可以认为串口数据能够被及时访问。

2.5 汇编程序与汇编器

2.5.1 汇编程序

本次实验中的汇编程序由两部分组成，一部分是从串口读入数据并运行算法计算最大公约数，另一部分是数码管的译码和显示，由于实验中定时器的中断只用来完成数码管的扫描，因此数码管的译码和显示代码就是中断处理代码。

第一部分代码首先启动定时器，而后检查串口标志位，当串口标志位有效时读串口数据。待输入的两个数据均读取完成后，计算结果并访问外设以显示结果。当完成任务后，CPU 将进入无限循环的状态以便观察结果。

第二部分的代码如首先将处理与定时器相关的中断，而后保护现场，由于程序比较简单，主程序与中断处理程序未使用相同的寄存器，因此这一步在代码中未体现。中断处理代码首先检查数码管对应外设数据，并移动扫描位，而后进行软件译码，软件译码的过程即 case 块语法转换成汇编语法，较为繁琐，因此在报告中删去部分。软件译码完成后，修改数码管外设对应地址值，重新启动定时器，回到主程序继续执行。

两部分代码见于附录。

2.5.2 汇编器

汇编器使用 python 语言编写。汇编器依照以下步骤执行工作：

1. 遍历代码，计算 Label 名称对应的地址值
2. 遍历代码，将 Label 名称表示的地址转换成相对地址或绝对地址
3. 遍历代码，将汇编程序转换成机器码

匹配和替换工作主要使用正则表达式完成，主要代码见于附录。

3 关键代码与文件清单

assemble

assemble.py 汇编器程序
data.txt 实验使用测试程序
encode.py 机器码转换为verilog文件结构程序
m_code.txt 汇编器转换机器码文件
verilog.txt 直接贴入verilog rom.v 文件

OneCycle
Adder.v 全加器
ALU.v 单周期ALU
Control.v 单周期控制信号生成文件
CPU.v 单周期CPU结构文件
DataMem.v 单周期Data Memory
digitube_scan.v
divclk.v 分频模块
Peripheral.v 外设模块
regfile.v 寄存器
rom.v 指令存储器
UART.v 串口相关电路实现

output_files
Pipeline
Adder.v 全加器
ALU.v 流水线ALU
Control.v 流水线控制信号生成文件
CPU.v 流水线CPU结构文件
DataMem.v 流水线Data Memory
digitube_scan.v
Peripheral.v 外设模块
regfile.v 寄存器
rom.v 指令存储器
UART.v 串口相关电路实现

output_files

4 仿真结果与分析

4.1 ALU 仿真

顺序执行 ALU 各项功能指令如表1，所得波形如下2。

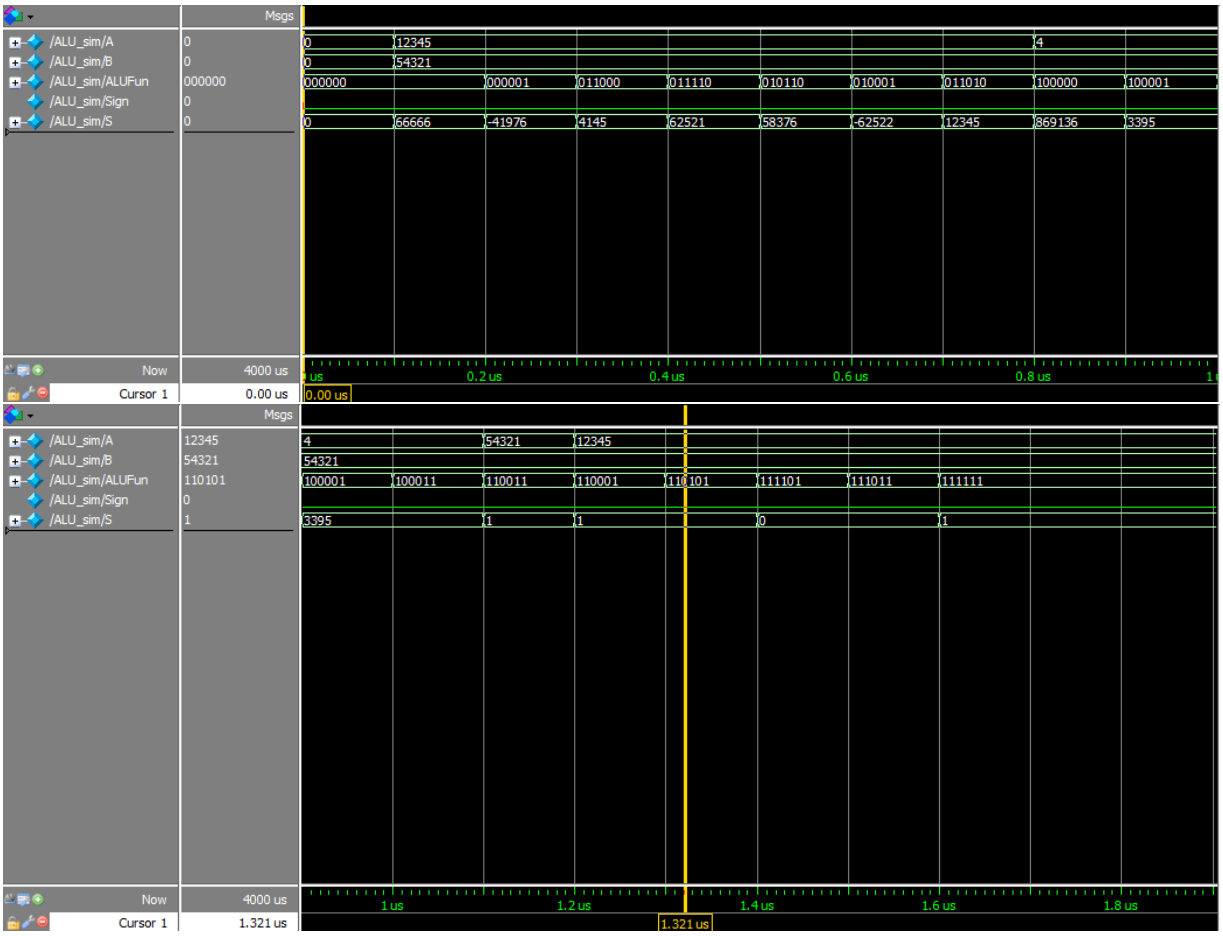
Msgs
/ALU_sim/A
/ALU_sim/B
/ALU_sim/ALUFun
/ALU_sim/Sign
/ALU_sim/S
0
0
000000
0
0
0
12345
54321
000001
011000
011110
010110
010001
011010
100000
100001
66666
-41976
4145
62521
58376
-62522
12345
869136
3395
4
Now
4000 us
Cursor 1
0.00 us
0.2 us
0.4 us
0.6 us
0.8 us
12345
54321
110101
1
100001
100011
110011
110001
111101
111011
111111
1.321 us
1 us
1.2 us
1.4 us
1.6 us
1.8 us

图 2: ALU 仿真波形

序号	指令
1	$12345 + 54321 = 66666$
2	$12345 - 54321 = -41976$
3	$0x3039\&0xd431 = 0x1031$
4	$0x3039\|0xd431 = 0xf439$
5	$0x3039$ XOR $0xd431 = 0xe408$
6	$\sim (0x3039\|0xd431) = 0xffff43a$
7	$54321 << 4 = 869136$
8	$54321 >> 4 = 3395$
9	$54321 >> 4 = 3395$
10	$54321 == 54321, S = 1$
11	$54321! = 12345, S = 1$
12	$12345 < 54321, S = 1$
13	$12345 > 0, S = 0$
14	$12345 > 0, S = 0$
15	$12345 > 0, S = 1$

表 1: ALU 仿真指令顺序表

4.2 单周期程序仿真

4.2.1 基本四则运算

在单周期 CPU 执行如下程序

```
addi $t0, $0, 200
addi $t1, $0, -300
add $t2, $t0, $t1
sub $t3, $t0, $t1
```

得到运行结果如图3。

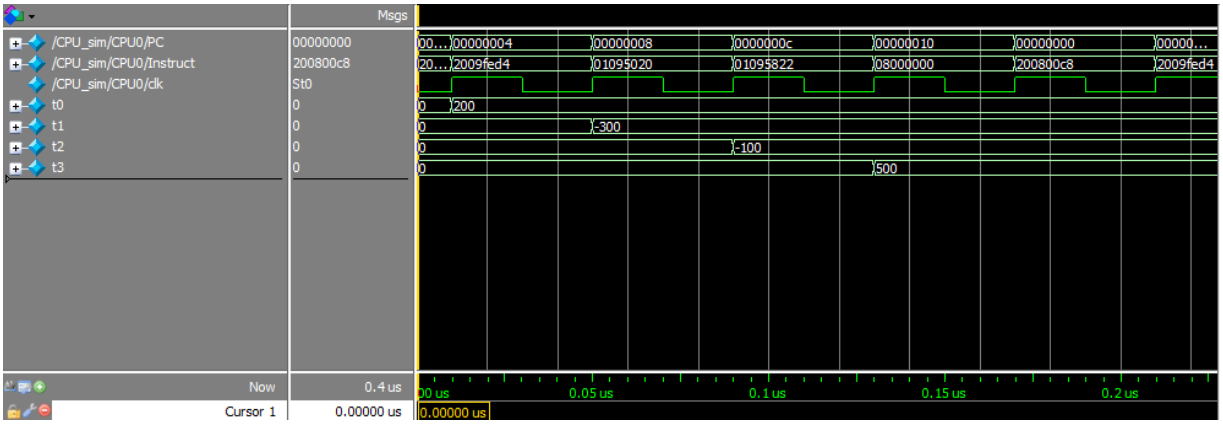

图 3: 单周期 CPU 基本运算波形

指令正常完成，对应寄存器正常写入。

4.2.2 逻辑运算

在单周期 CPU 中执行如下程序：

```
addi $t0, $0, 0x4321
addi $t1, $0, 0x1234
and $t2, $t0, $t1
or $t3, $t0, $t1
xor $t4, $t0, $t1
addi $t5, $0, 0xffff
andi $t5, $t5, 0x1234
```

得到波形如4。

指令正常完成，对应寄存器正常写入。

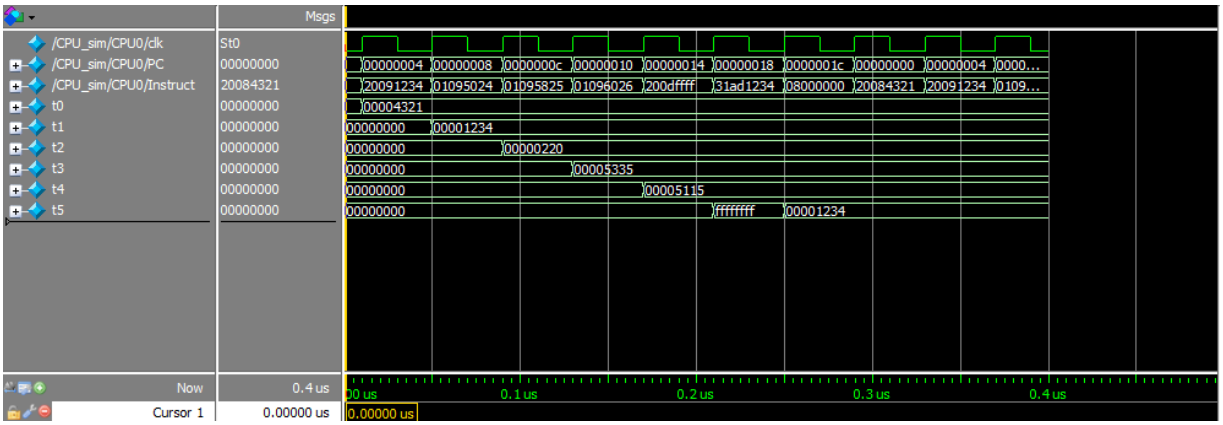

图 4: 单周期 CPU 逻辑运算波形

4.2.3 移位和置位指令

在单周期 CPU 中执行如下程序

```
addi $t0, $0, -1234
sll $t1, $t0, 3
srl $t2, $t0, 3
sra $t3, $t0, 3
slt $t4, $t2, $0
slti $t5, $t3, 0
```

得到波形如5。

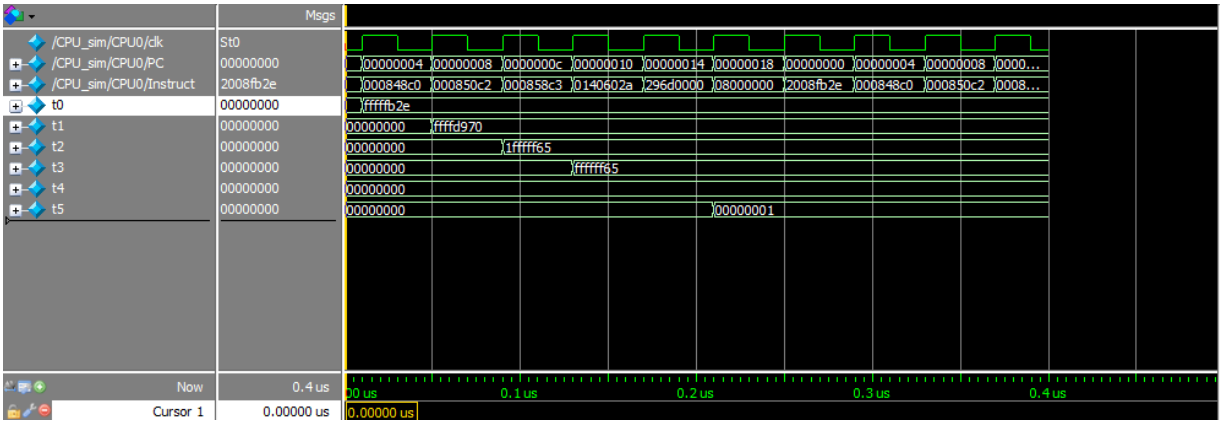

图 5: 单周期 CPU 移位与置位运算

指令正常完成，当使用逻辑右移时，负数将转换为正数，而当使用算数右移时，负数仍然保持符号不变。

4.2.4 跳转指令

在单周期 CPU 中执行如下程序

```
A:
beq $0, $0, D
B:
j B
C:
jal E
bne $0, $0, B
j B
D:
j C
E:
jr $ra
```

得到波形如6。

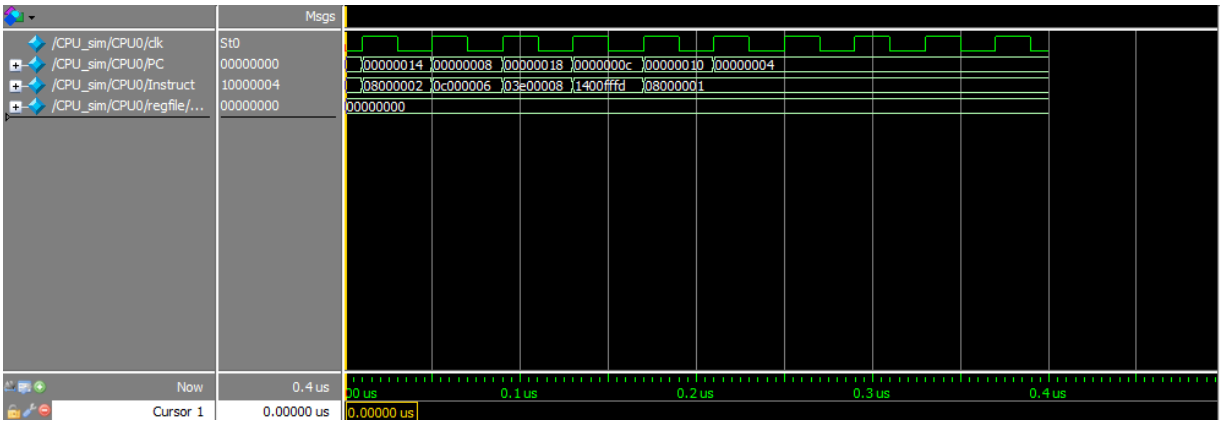

图 6: 单周期 CPU 跳转指令

CPU 在多次跳转中正常工作。

4.2.5 中断与外设

使用实验用最大公约数代码，通过 testbench 模拟串口输入信号，观察中断和串口收发情况。

当计时器计时完成时，CPU 进入中断处理，最高位置为 1，原程序 PC 被存放在 26 号寄存器中。波形如7。

观察外设整体情况，七段数码管在定时中断中轮流扫描，LED 和串口工作正常，波形如 7 系统正常工作。

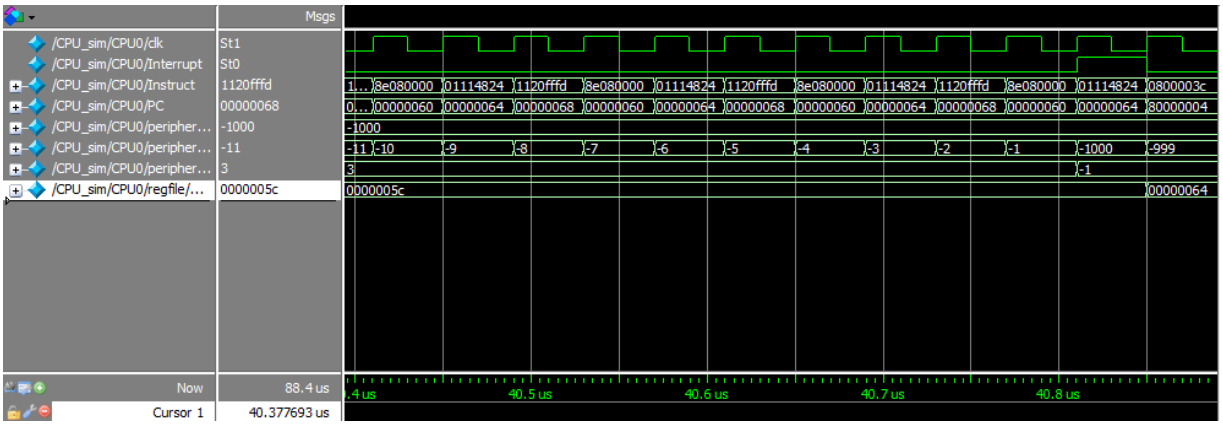

图 7: 单周期 CPU 中断工作波形

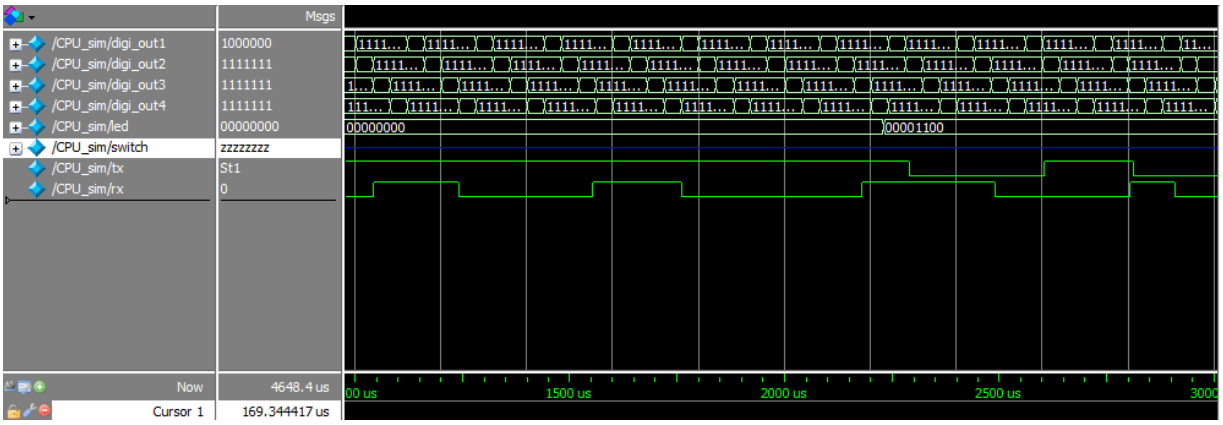

图 8: 单周期外设与串口工作波形

4.3 流水线仿真

4.3.1 冒险与转发

在流水线 CPU 中运行如下指令

```
addi $t0, $0, 1234
add $t1, $t0, $0
sub $t2, $0, $t0
```

仿真波形如9。

可以看到虽然上一条指令的结果还未写入寄存器，但是通过 EX/MEM 和 MEM/WB 两层寄存器的转发，仍然可以获得正确的结果。

4.3.2 load-use 处理

在流水线 CPU 中运行如下指令

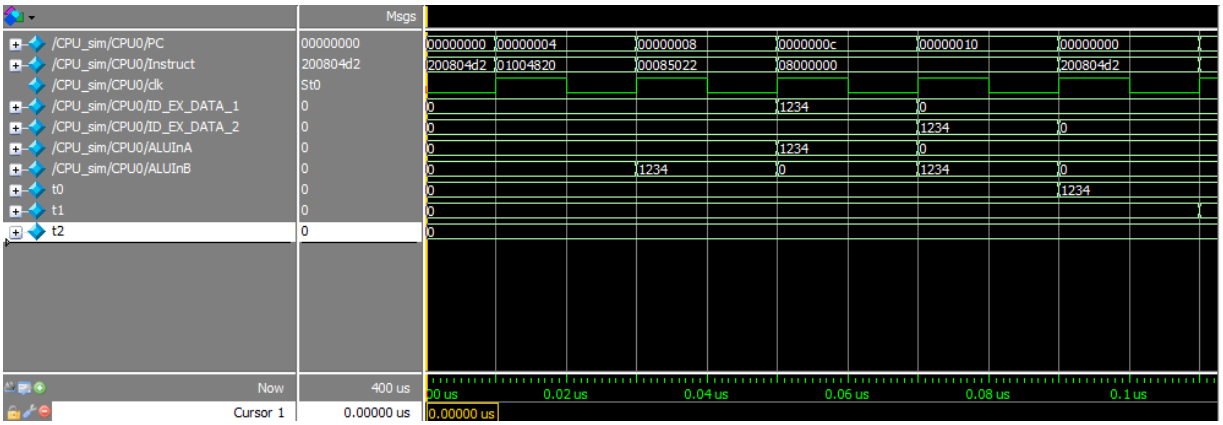

图 9: 流水线冒险与转发

```
addi $s0, $0, 0x4000
sll $s0, $s0, 16
lw $t0, 0($s0)
add $t1, $t0, $t0
add $t2, $t0, $t0
```

运行时波形如10。

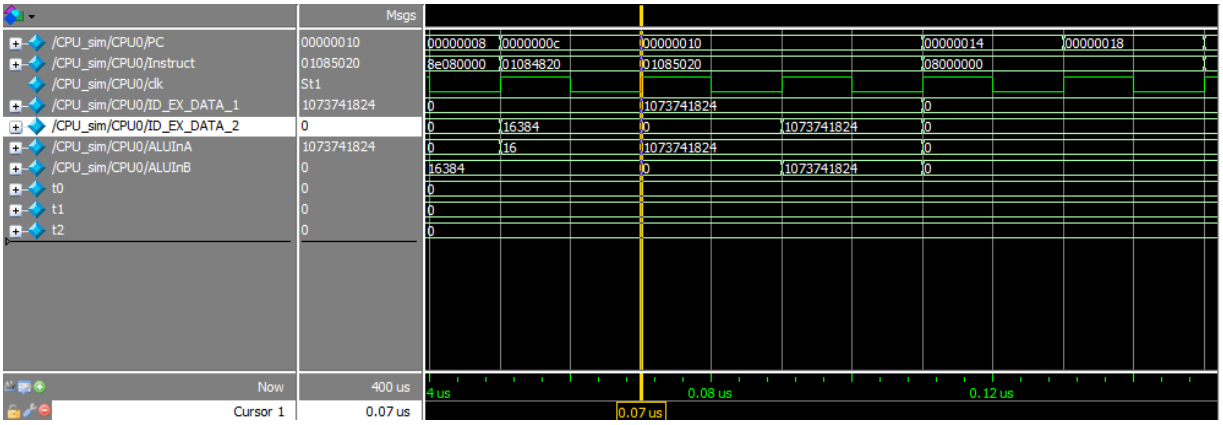

图 10: load-use 处理

当地址为 0x0000000C 的指令，发现其出现 load-use 类竞争时，会阻塞一个周期等待 lw 指令执行完成。通过这一处理，该指令可以获得正确的数据。

4.3.3 branch 指令

在流水线 CPU 中运行如下指令

```
beq $0, $0, A
addi $t0, $0, 1
A:
addi $t0, $0, 2
```

运行时波形如11。

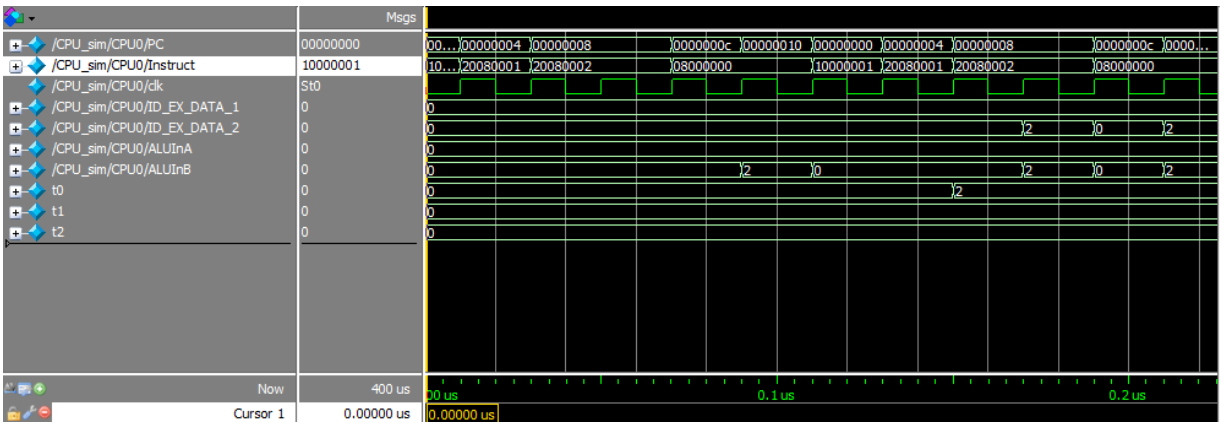

图 11: branch 处理

当 beq 语句执行到 ex 阶段时，判断跳转为真并将当前 IF 和 ID 阶段的两条指令清除，将跳转目标，此处为 0x00000008 放入 PC 中。执行跳转后的目标指令。

4.3.4 j 指令

在流水线中运行如下指令

```
j A
addi $t0, $0, 1
addi $t0, $0, 2
addi $t0, $0, 3
A:
addi $t0, $0, 10
j B
B:
j B
```

得到波形如12。

虽然 j 指令之后的一条指令进入了 IF 阶段，但其从未被执行，j 指令会将其控制信号全部清零后跳转到目标地址。

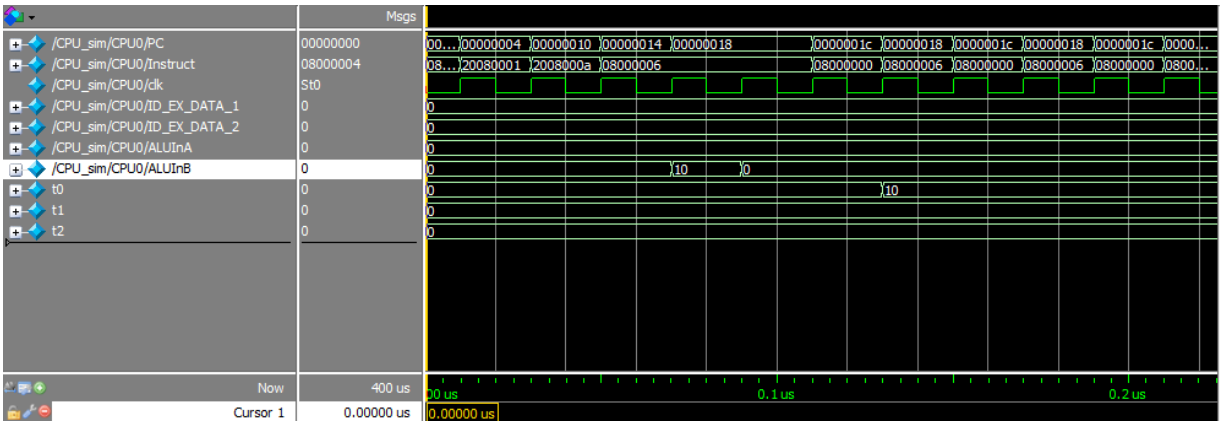

图 12: j 处理

4.3.5 中断和外设

使用实验用最大公约数代码，通过 testbench 模拟串口输入信号，观察中断和串口收发情况。

中断波形如13。在 0x00000068 指令触发中断后，0x00000068 指令并未被实际执行，系统进入中断处理程序，中断处理程序结束后的返回地址将在完成五级流水后存入寄存器中。返回地址与触发中断时执行的指令直接相关。

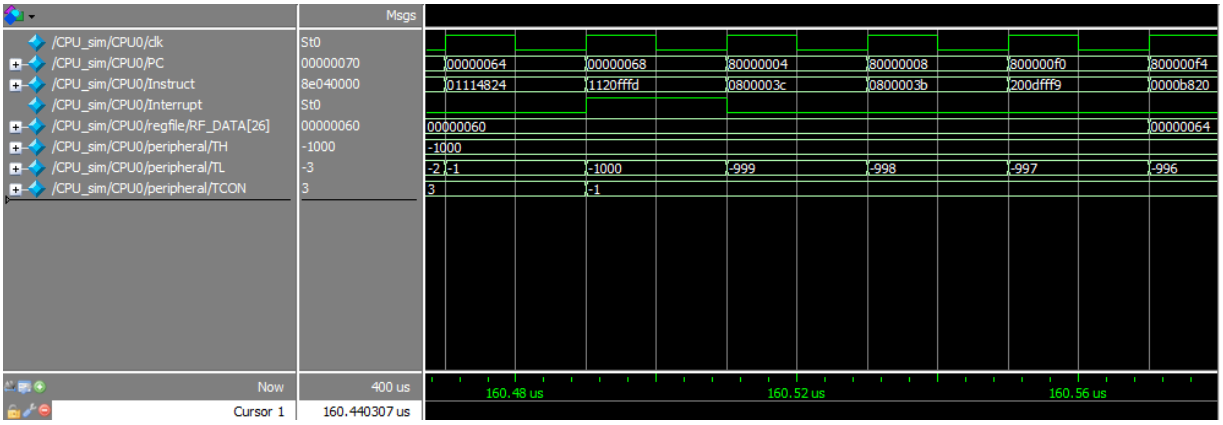

图 13: 中断处理

外设工作情况如图14。

5 综合情况

单周期各项性能指标如下：

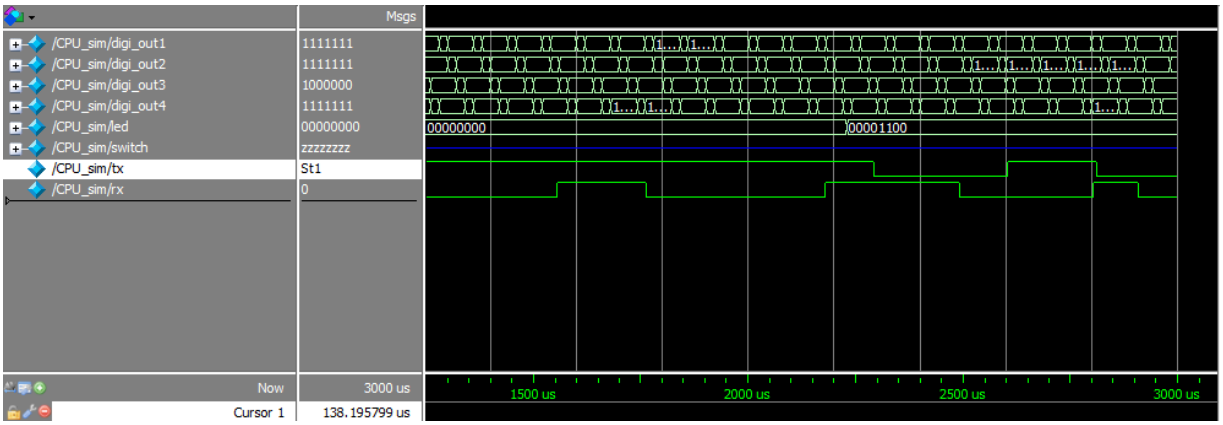

图 14: 流水线 CPU 外设工作情况

性能指标	
Total Logic Elements	4498(14%)
Total Registers	3772(11%)
Total Pins	48(10%)
Restricted Fmax	$41.55MHz$

流水线各项性能指标如下：

性能指标	
Total Logic Elements	3803(11%)
Total Registers	3695(11%)
Total Pins	48(10%)
Restricted Fmax	$88.0MHz$

6 硬件调试情况

- 七段数码管正确扫描显示待计算数据，LED 灯正确显示结果
- 串口收发正确进行
- 运算任务正确完成

7 思想体会

7.1 王敏虎

在团队中，我主要负责汇编器、汇编代码、外设相关代码和 testbench 的编写。

在实验中，最大的收获是理解了中断对于 CPU 的意义，以及 CPU 如何通过中断完成 I/O 操作。对于 CPU 将外设抽象为存储地址这一工程实践有了初步的认识。同时，通过测试 CPU 代码，

了解了 CPU 开发中比较容易出错的地方。例如在流水线 CPU 中断设计中，最初中断始终不成功，在反复比对波形后，最终发现是存入寄存器的地址不对，进而发现此处应当根据执行的指令对存储地址进行判断。

此外，在基于 github 的团队开发方面也获益不少。

7.2 白钦博

总的来说这次流水线 CPU 的工作进展的较为顺利，我已开始负责完成一个基本能跑的流水线，在单周期的基础上，将工作分为添加段间寄存器，增加冒险探测电路及根据流水级数修改单周期的相应信号。在没有中断之前，遇到的 bug 不多，主要是一些信号忘记更改，以及转发电路的第三级转发的问题，更改这些问题后，我用了两个没有中断的简单汇编程序均测试成功，完成之后，将程序交给队友测试最终程序，发现关于中断出了一些问题，因为最终的汇编程序、外设及单周期中断的调试工作均由王敏虎同学负责，故将初步的调试工作交给了他。在完成跑通流水线后，我又根据王敏虎同学修改的基础上做了一些优化，减少了部分信号数量，删除了部分路径，增加了流水线 CPU 的主频。

这次实验是我少数参加合作开发的经历，让我对团队合作有了不少新的认识，在一个合理的分工下，两天跑通单周期，一天基本跑通流水线，我认为团队合作提高了我们不少的效率。除此之外，亲自完成流水线 CPU 的设计，也大大加深了我对理论课程的认识，尤其是画出一张完整的流水线数据通路之后，设计的思路就比较清晰了。在优化 CPU 主频的过程中，我第一次实际应用 quratus 查看关键路径，减小关键路径延迟的方法进行优化。

总体来说，这次 CPU 大实验，对与我个人的实践水平有一个较大的帮助。