



实验六

191098328 计算机科学与技术系 张世茂

191098328@smail.nju.edu.cn



2022-1-1

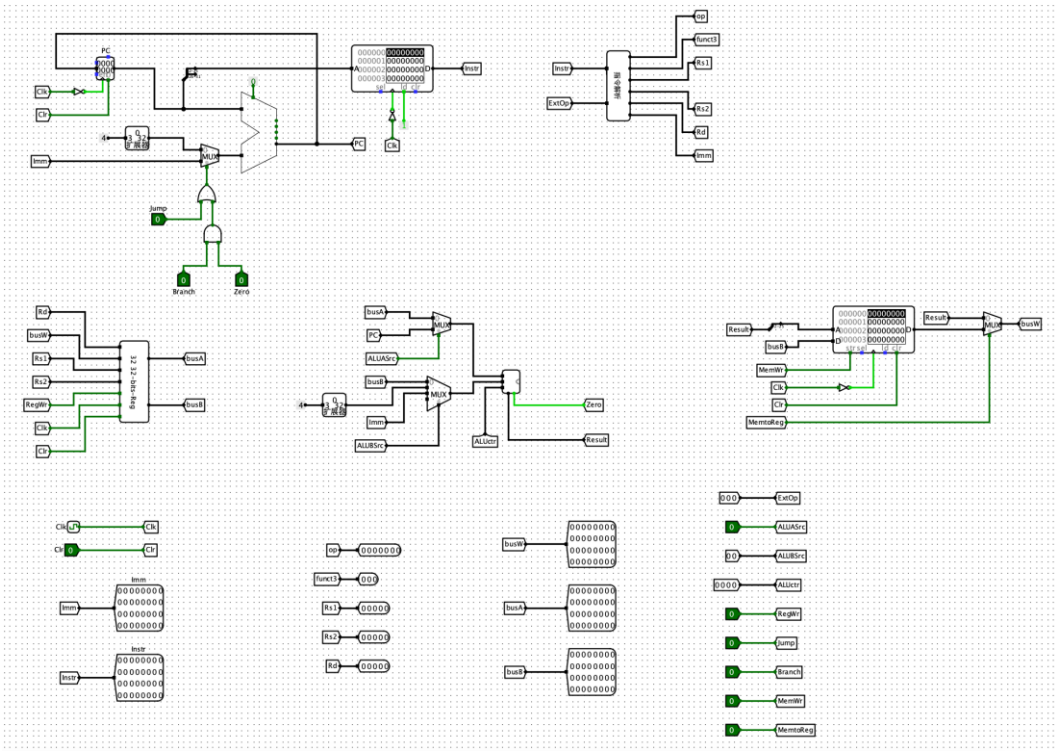
目录

1. 数据通路	2
1.1 实验操作流程	2
1.2 实验结果	2
1.3 实验错误与原因分析	3
2. 单周期 RISC-V CPU	3
2.1 实验操作流程	3
2.2 实验结果	4
2.3 实验错误与原因分析	4
3. 累加和程序	4
3.1 实验操作流程	4
3.2 实验结果	6
3.3 实验错误与原因分析	8
4. 思考题	9
5. 实验总结	12

1. 数据通路

1.1 实验操作流程

按照实验手册上标示的 RISC-V 指令的数据通路原理图并结合之前实现所实现的功能电路对数据通路进行实现，其实现的电路原理图如下图所示：



在实现上述要求的电路后，就可以对电路进行进一步的仿真检验。

1.2 实验结果

首先分析电路的输入输出对应表，该阶段实验所需实现的数据通路是针对已知的 9 条 RISC-V 指令，其指令的具体内容如下：

指令	funct7 (31-25)	rs2 (24-20)	rs1 (19-15)	funct3 (14-12)	rd (11-7)	op (6-0)
add rd, rs1, rs2	0000 000	rs2	rs1	000	rd	011 0011
slt rd, rs1, rs2	0000 000	rs2	rs1	010	rd	011 0011
sltu rd, rs1, rs2	0000 000	rs2	rs1	011	rd	011 0011
ori rt, rs1, imm12	imm[11:0]		rs1	110	rd	001 0011
lui rd, imm20	imm[31:12]				rd	011 0111
lw rd, rs1, imm12	imm[11:0]		rs1	010	rd	000 0011
sw rs1, rs2, imm12	imm[11:5]	rs2	rs1	010	imm[4:0]	010 0011
beq rs1, rs2, imm12	imm[12][10:5]	rs2	rs1	000	imm[4:1][11]	110 0011
jal rd, imm20	imm[20][10:1][11][19:12]				rd	110 1111

由于在下一部分中即将在此数据通路的基础上添加控制器电路并最终实现单周期 CPU，且单独的数据通路手动进行仿真检验未免有些过于繁琐，因此在之后部分的实验中将一并对其进行检验。

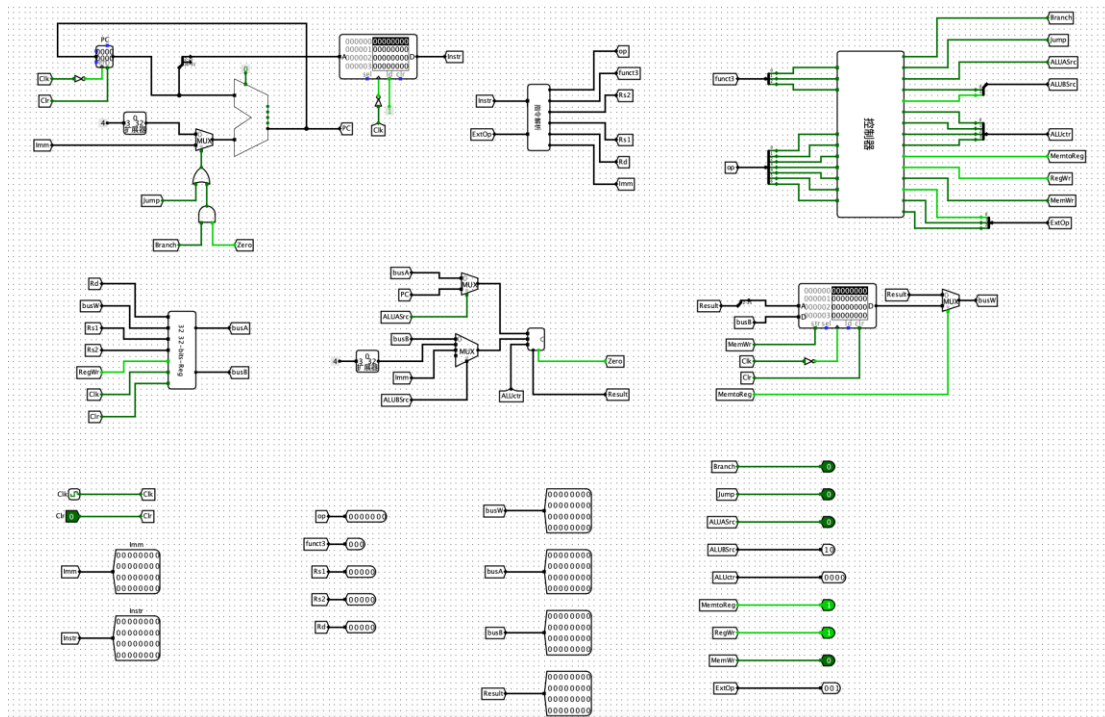
1.3 实验错误与原因分析

这部分实验得到电路原理图后依据电路原理图进行实现即可，但实现过程中要尤其注意 PC 变化与存储器地址编码的关系等细节，若直接取 PC 的低位，则存储器的取数地址会越过中间的指令而到达 4 条指令之后，因此为模拟真实情况应当从 PC 的第 2 位开始取，这一点也是在实验 5 中应当着重注意的。在这部分实验过程中没有出现什么特别的错误。

2. 单周期 RISC-V CPU

2.1 实验操作流程

要实现单周期 RISC-V CPU，则要在上面实现的数据通路的基础上添加控制器电路并将对应的输入和控制信号进行连接，其电路原理图如下所示：



可以看到，右上角是添加的控制器电路，由于之前是采用自动组合电路分析生成的电路，因此这里都是单位输入输出，采用分线器将多位输入输出进行处理。其余剩下的部分是之前实现的数据通路。为了简化电路图，将组件之间值的传递利用隧道元

件进行传输。为了方便进行观察调试，在电路下方添加了一系列探针对电路运行过程中产生的值进行外显。

实现电路后，对电路进行进一步的观察和检验。

2.2 实验结果

该电路实现了一个可以运行九条指令的单周期 RISC-V CPU，具有 CPU 正常运行所应具有的所有基本功能，可以运行包含一系列指令的程序。

在下一部分中，将利用 RARS 实现将累加和计算的汇编程序转换为机器码，并将其加载入已实现的 CPU 电路的指令存储器中进行运行，检验电路的实现是否正确。

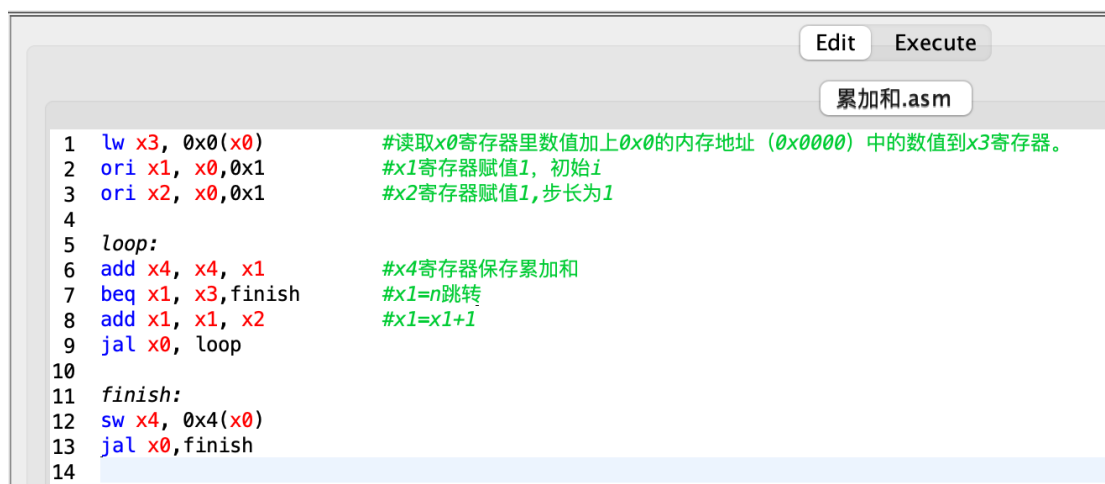
2.3 实验错误与原因分析

这部分实验在上面实验的基础上添加控制器电路并对相关的信号传输进行修改实现即可，实现过程中应注意不同控制信号与原件功能之间的对应关系等细节，其余的在实验过程中没有出现什么特别的错误。

3. 累加和程序

3.1 实验操作流程

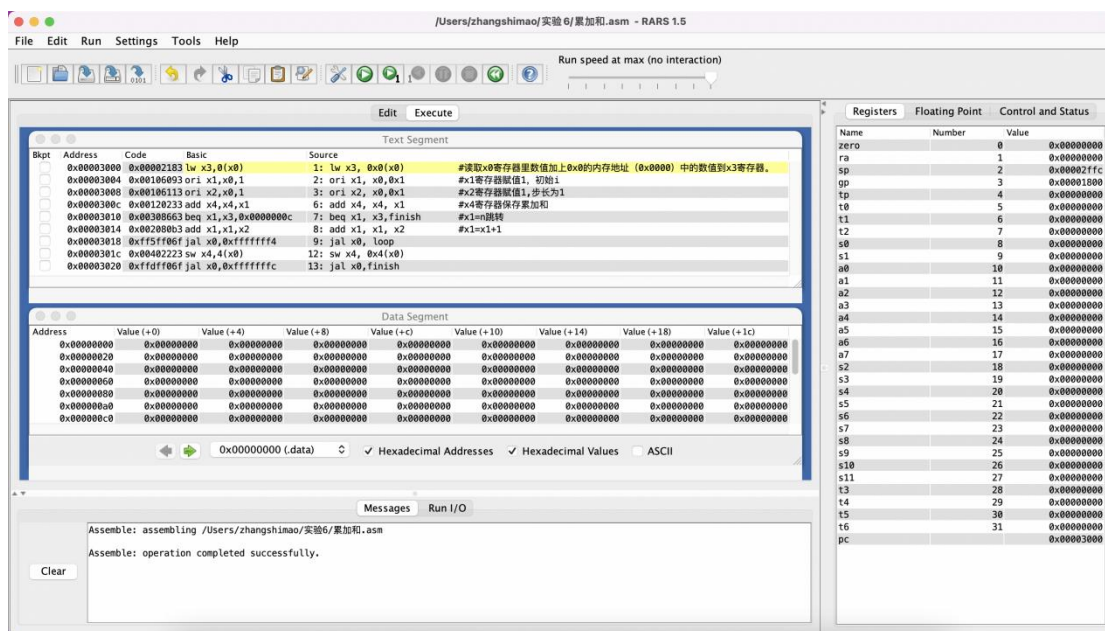
首先下载 RARS 并运行，创建一个新的 asm 文件后将累加和运算的汇编代码写入，最终效果如下：



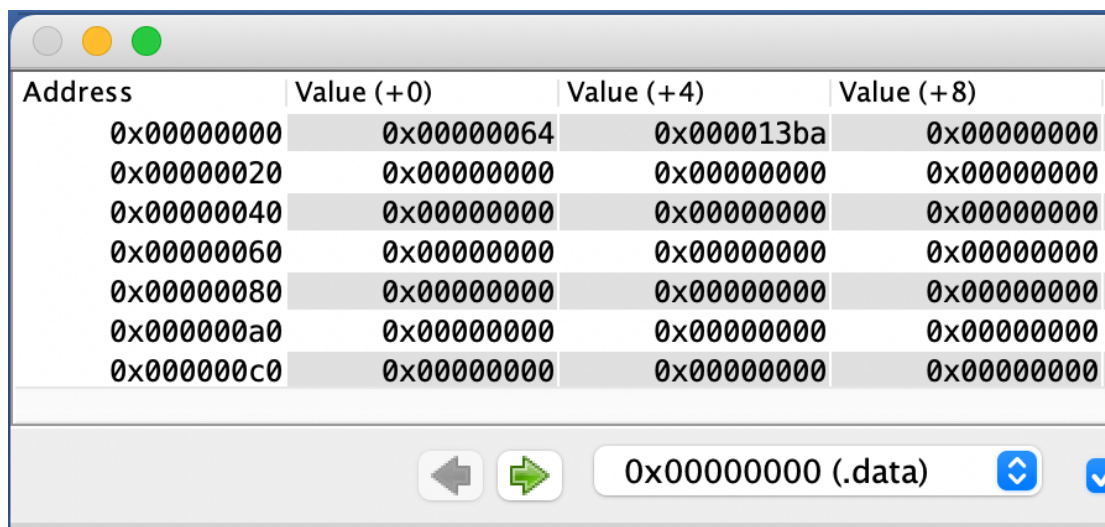
```
1  lw x3, 0x0(x0)           #读取x0寄存器里数值加上0x0的内存地址 (0x0000) 中的数值到x3寄存器。
2  ori x1, x0, 0x1          #x1寄存器赋值1, 初始i
3  ori x2, x0, 0x1          #x2寄存器赋值1, 步长为1
4
5  loop:
6  add x4, x4, x1            #x4寄存器保存累加和
7  beq x1, x3, finish        #x1=n跳转
8  add x1, x1, x2            #x1=x1+1
9  jal x0, loop
10
11 finish:
12 sw x4, 0x4(x0)
13 jal x0, finish
14
```

由于本实验中在单周期 CPU 的设计中没有考虑内存管理单元，而是将指令存储器和数据存储器分离设计，运行时采用物理地址，并且起始地址都是 0，所以在 RARS 中进行设置的相应修改，将 Memory Configuration 选项中设置为 Compact, Data at

Address 0，这样数据段的起始地址就从 0 开始。然后对上面已编写好的代码进行编译，得到如下图所示的机器代码：

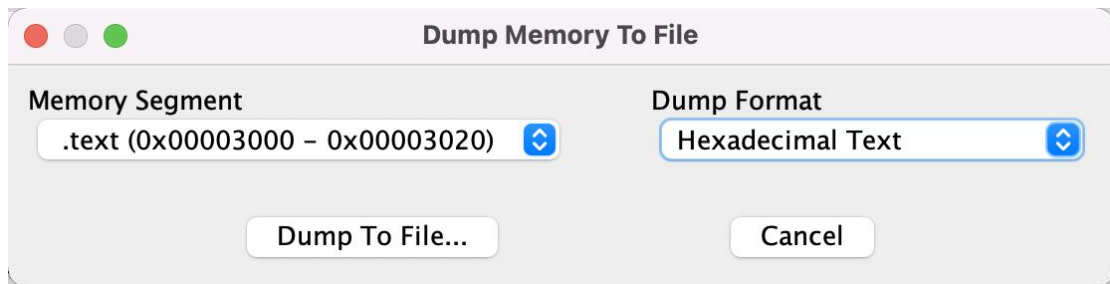


在下方地址 0x00000000 处开始的地址块内写入数据 0x64，然后开始仿真运行，最终可得到计算结果如下：



可以看到程序得到了正确的计算结果 0x13ba，这也就意味着我们上面的汇编程序的机器代码是正确的，因而我们可以利用其对我们之前所实现的 CPU 电路进行仿真检验，验证实现的功能是否可以正确运行。

然后将上面所得的机器代码进行导出，进行如下设置后导出到目标文件：



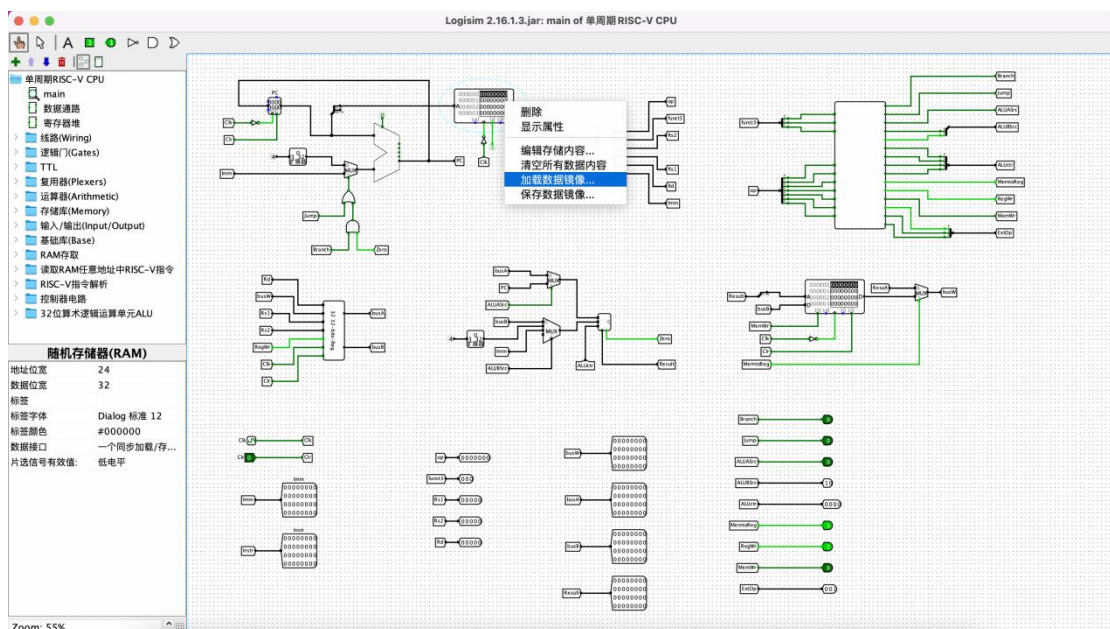
导出到外部文件后为了之后可以进一步将其加载到电路的指令存储器组件中，在文件开头的第一行再添加内容“v2.0 raw”，如下图所示：

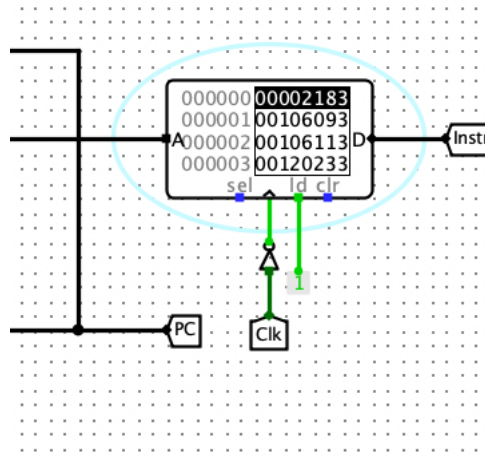


然后下面就可以将该程序加载入指令存储器进行运行验证了。

3.2 实验结果

上面已经得到了累加和程序的机器代码，利用它验证所实现 CPU 电路的功能是否正确，首先将机器代码装载入电路中的指令寄存器，如下图所示：





然后可将数据存储器从 0x0 开始的 32 位数据赋值为 0xa，并在电路仿真中选择合适的时钟频率并开启时钟连续进行自动仿真运行，待一定时间后停止时钟，可以看到数据存储器中存储的内容变化如下：

```

000000 0000000a 00000037 00000000
000010 00000000 00000000 00000000
000020 00000000 00000000 00000000
000030 00000000 00000000 00000000
000040 00000000 00000000 00000000

```

可以看到，此时 0xa 的累加和运算结果 0x37 已经计算完毕并且存入了数据存储器 的相应位置中，CPU 正确执行了我们给出的累加和代码并得到了符合预期的结果。

将电路中的值重置并再次向数据存储器中 0x0 开始的 32 位数据赋初值 0x64，并 开启时钟连续进行自动运行，一段时间后得到结果：

```

000000 00000064 000013ba 00000000
000010 00000000 00000000 00000000
000020 00000000 00000000 00000000
000030 00000000 00000000 00000000
000040 00000000 00000000 00000000

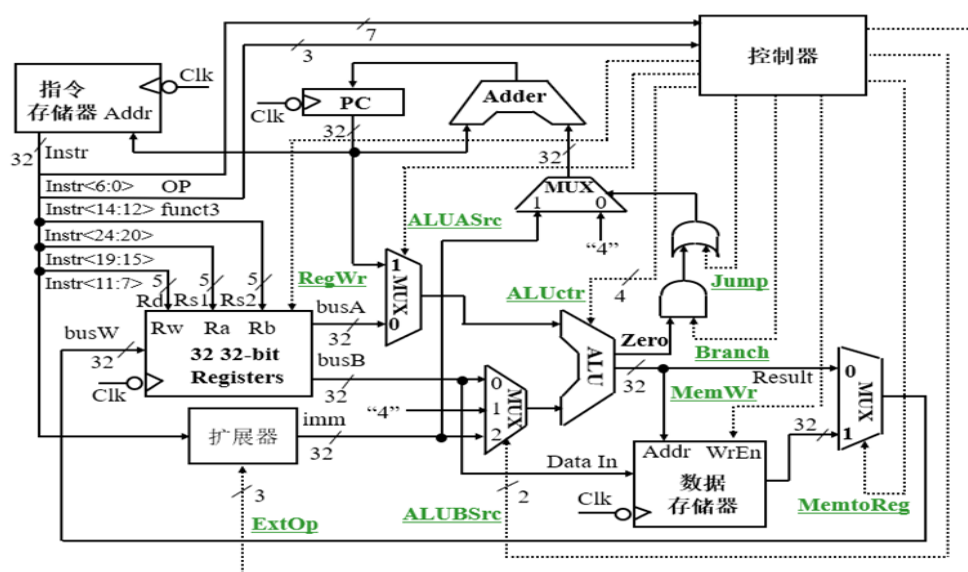
```

可以看到，此时 0x64 的累加和运算结果 0x13ba 已经计算完毕并且存入了数据存 储器的相应位置中。

经过上面的检验，任意挑选的样例进行仿真检验所计算得出的结果都符合预期， 可以在一定程度上验证所实现的 RISC-V CPU 的正确性。

3.3 实验错误与原因分析

1. 该部分电路在实验手册中给出了电路原理图如下：



但如果完全按照该电路原理图实现最终的功能电路的话可能会在运行时出现错误。经过探究和反复测试，我得到一个猜想的可能原因：我们在本课程实验中所实现的功能电路为简化实现和贴近课程耳而与实际的电路存在部分差别，在实际电路中时钟信号是寄存器堆能否进行写入的唯一标准，当时钟信号的上升沿没有来临时寄存器堆无论如何不可能进行写入，但在本课程的实验中实现寄存器堆时，采用将 WE、Clk、Rd 三者通过一个三输入与门后作为时钟信号，这也就意味着此时三者是等价的，Clk 在某些时候并不具有决定性的作用。放到电路图中来看，在时钟信号到达一个下降沿时，寄存器堆的时钟信号最先变化，此时 Rd、RegWr 和 busW 仍是旧值，经过指令解析后 Rd 的值得到更新，但由于 busW 需要经过的电路延迟更多，因此此时 busW 的值仍然是旧值，但可以发现按照我们本课程中寄存器堆的实现，若此时 RegWr 为 1，则 Rd 号寄存器来到了一个触发沿，那么该寄存器终究会被写入 busW 的旧值而非想要存入的新值，那么在新值还未被写入之前的这段时间里，这个寄存器里存储的非预期值都是危险的，若取用该寄存器中的值参与运算则可能会发生意想不到的错误。因此，若想解决此问题，则需要人为地将寄存器堆的时钟触发沿的到来延迟，要求在时钟信号的触发沿到来之前 Rd 和 busW 的值都必须完成更新并到达稳定，这样才能避免在错误的位置写入错误的值。在本实验中的解决办法是在维持其他元件下降沿触发的基础上将寄存器堆改为上升沿触发，这样在触发沿到来时其他的输入以及控制信号早已完成更新。

2. 通过上网查阅资料，我得知在 RISC-V 的 CPU 中存在一个特殊的点，即寄存器堆中 0 号寄存器的值永远为 0，这也就意味着应当在之前实验实现的寄存器堆的基础上进行修改以适应 RISC-V 指令系统的要求。为了使 0 号寄存器堆的值始终为 0，具体的方法是将寄存器堆中 0 号寄存器的 Clr 位始终输入为常量 1，这样其清零位就一直保持有效。实际上，这一点在实验 3 和本次实验的思考题中也有所提及，即如何使 0 号寄存器内存储的数值始终为 0。

4. 思考题

1.RISC-V 中 0 号寄存器的内容始终为 0，则在寄存器堆的设计电路中需要做哪些修改？

将原先连接至 0 号寄存器的 Clr 信号改为常量 1，这样 0 号寄存器的清零端就一直被置为有效，则内容就始终为 0。

2.如果需要增加与运算和逻辑右移运算，分别需要修改电路中的哪些部分？

为了进行与运算和逻辑右移运算，需要对 ALU 进行修改添加相关计算功能，必要时还需扩展 ALUctr 编码，然后为了相关指令的扩展，还需要对控制器电路进行修改，在对指令控制信号进行重新设计后重新实现对应的控制器电路。

增加与运算：因为增加了一种运算操作，因此首先要在 ALU 电路中添加 32 位的 2 输入与门来具体执行该运算，而由于电路整体是多功能算术逻辑单元，因此涉及到输出结果的选择，将输出结果处的多路选择器增加一个输入端并将与门输出结果接入。最后由于需要对新增操作确定对应的操作指令码，因此还要将 ALUctr 和 OPctr 进行位扩展，并根据为与操作确定的新的操作码对 ALU 操作控制信号生成部件和控制器电路中的逻辑进行更改。

逻辑右移操作：因为增加了逻辑右移运算功能，因此首先要在 ALU 电路中增加逻辑右移运算组件，还要增加一个输入端来对逻辑右移的位数进行确定，然后在输出结果处的多路选择器上增加一个输入端并将逻辑移位器的输出结果接入，最后还需要将 ALUctr 和 OPctr 进行位扩展，并根据为逻辑右移操作确定的新的操作码对 ALU 控制信号生成部件和控制器电路中的逻辑进行修改。

3.编写一种排序算法汇编程序，并进行验证结果。

在 RARS 中对冒泡排序算法的汇编代码实现如下图所示，保存在冒泡排序.asm 中：

EditExecute

冒泡排序.asm

```

1      lw x1,0(x0)
2      add x2,x1,x0
3      ori x4,x0,1
4      ori x5,x0,4
5      ori x6,x0,0xffffffff
6  L1:
7      beq x2,x4,finish
8      ori x3,x0,1
9      ori x7,x0,4
10     ori x8,x0,8
11  L2:
12     sltu x11, x3,x2
13     beq x11,x0,L3
14     lw x9,0(x7)
15     lw x10,0(x8)
16     sltu x11,x9,x10
17     beq x11,x4,L4
18     sw x10,0(x7)
19     sw x9,0(x8)
20     jal x0, L4
21  L3:
22     add x2,x2,x6
23     jal x0, L1
24  L4:
25     add x3,x3,x4

```

Line: 30 Column: 20 ☒ Show Line Numbers

完整代码如下图所示：

冒泡排序.asm

```

lw x1,0(x0)
add x2,x1,x0
ori x4,x0,1
ori x5,x0,4
ori x6,x0,0xffffffff
L1:
    beq x2,x4,finish
    ori x3,x0,1
    ori x7,x0,4
    ori x8,x0,8
L2:
    sltu x11, x3,x2
    beq x11,x0,L3
    lw x9,0(x7)
    lw x10,0(x8)
    sltu x11,x9,x10
    beq x11,x4,L4
    sw x10,0(x7)
    sw x9,0(x8)
    jal x0, L4
L3:
    add x2,x2,x6
    jal x0, L1
L4:
    add x3,x3,x4
    add x7,x7,x5
    add x8,x8,x5
    jal x0, L2
finish:
    jal x0, finish

```

然后在 RARS 中对上面的代码进行编译，得到机器代码，仿照之前的累加和程序，在将该程序加载进 CPU 执行之前先对该程序进行仿真检验，验证该程序的正确性。

初始输入如下图所示，其中从地址 0x0 开始的四个字节存放的是等待排序的数的个数，然后向后紧接着每四个字节存放一个等待排序的数：

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000007	0x0000000a	0x0000000f	0x00000004	0x00000023	0x00000064	0x0000002d	0x00000006
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

开始运行，一段时间后中止程序观察相应地址存储的内容，结果如下：

Data Segment								
Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x00000000	0x00000007	0x00000004	0x00000006	0x0000000a	0x0000000f	0x00000023	0x0000002d	0x00000006
0x00000020	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000040	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000060	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x00000080	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000
0x000000a0	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

可以看到，7个数现在已经是有序排列。

然后将程序加载到所实现的 CPU 中验证是否能正确运行，同时验证两者的正确性。

完整的机器代码如下图所示：

冒泡排序

```

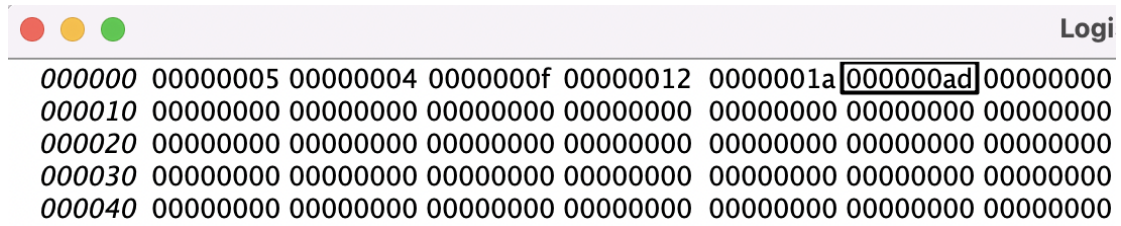
v2.0 raw
00002083
00008133
00106213
00406293
fff06313
04410663
00106193
00406393
00806413
0021b5b3
02058063
0003a483
00042503
00a4b5b3
00458c63
00a3a023
00942023
00c0006f
00610133
fc9ff06f
004181b3
005383b3
00540433
fc9ff06f
0000006f

```

数据存储器中的初始数据如下图所示：

Logis									
000000	00000005	00000012	0000000f	000000ad	00000004	0000001a	00000000	(
000010	00000000	00000000	00000000	00000000	00000000	00000000	00000000	(
000020	00000000	00000000	00000000	00000000	00000000	00000000	00000000	(
000030	00000000	00000000	00000000	00000000	00000000	00000000	00000000	(
000040	00000000	00000000	00000000	00000000	00000000	00000000	00000000	(

然后选择合适的时钟频率并设置时钟连续后开始执行，一段足够长时间后停止，观察数据存储器中的数据变化：



```
000000 00000005 00000004 0000000f 00000012 0000001a 000000ad 00000000
000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000020 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000030 00000000 00000000 00000000 00000000 00000000 00000000 00000000
000040 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

原先的数据已经按从小到大的顺序有序排列了，这说明实现的冒泡排序算法在本次实验所实现的 CPU 电路上可以正确执行，同时验证了所实现的程序以及电路的正确性。

5. 实验总结

本次实验是本课程中的最后一次实验，是对一个单周期 RISC-V CPU 的综合实现，综合了这学期本课程中的所有知识模块和前面的实验，也是对本课程内容的一个总结和回顾。在这个实验的过程中不仅遇到了实现中的错误，也发现了之前实验实现的元件中一些需要适应和修改的地方，同时由于实现了可以运行完整简单程序的 CPU 电路，这次实验总体上也是最有趣的。一学期的实验到这里就要结束了，由于前期疫情的影响后期的实验安排较为紧凑，因此这也成为了我这学期日常课后学习时间中的重要一环，感谢一学期以来老师和助教学长学姐的辛苦和陪伴，也希望在接下来的期末考试中我可以考出理想的成绩。