

《数字逻辑与处理器基础》课程大作业报告

李羿璇，2021012644

1. 实验目的

1. 通过使用 Verilog 完成一个五级 MIPS 流水线，巩固理论课所学知识；综合运用前四次小作业中学到的知识，提高 Verilog 的编写能力和 Debug 能力；
2. 连接流水线与外设，体会总线与外设对 CPU 的扩展性起到的好处；
3. 完成流水线后通过软件方式进行译码等操作，与前几次小作业中的硬件方式操作相对比，体会软件方式和硬件方式之间的不同之处；
4. 巩固时序静态分析、资源占用情况分析的能力；

2. 设计方案

本次实验中，我的流水线设计方案综合了理论课上的设计和参考书¹上的数据通路，添加了连接外设的数据通路，并根据实际需要做了一定的修改。此外，我还实现了数码管交替显示输出、通过串口与计算机进行输入输出通信的功能。

2.1. 总体设计

流水线 CPU 共分为 5 个流水阶段，分别为 IF、ID、EX、MEM、WB。流水线在 ID 阶段进行跳转，在 EX 阶段进行分支。此外，为了解决冒险问题，还加入了转发单元 Forward Unit 用于解决数据冒险、冒险单元 Hazard Unit 用于解决 Load-Use 型数据冒险和分支、跳转指令引起的控制冒险。外设部分，LED 用于在数码管上显示、UART 用于与计算机之间通信。

2.2. 控制信号生成

为了最大化复用单周期的代码，我将单周期的控制信号原封不动地搬了过来；同时，为了支持要求中的多种分支指令，以及我自己添加的 bne 指令，我额外加入了 BranchType 控制信号。最终，形成了如附录中图 14 的控制信号真值表。

2.3. Hazard Unit 设计方案

- 对于 Load-Use 冒险，采取在 ID 阶段检出 Load-Use 冒险后 Keep PC、Keep IF/ID、Flush ID/EX 的策略；
- 对于 branch 引发的冒险，采取静态分支预测的方法，在 ID 阶段检出 branch 指令后不做任何操作，在 EX 阶段若检出跳转执行，则 branch PC、Flush IF/ID、Flush ID/EX；
- 对于 J 引发的冒险，在 ID 阶段检出跳转后 jump PC、Flush IF/ID。

2.4. Forward Unit 设计方案

为了尽可能复用代码，我将对 ALU 输入的转发统一封装成了 Forward Unit。判断逻辑有如下伪代码：

¹[美]David A.Patterson, [美]John L. Hennessy:《计算机组成与设计——软件硬件接口》，王党辉等译，机械工业出版社，北京，2015 年。

```

if (EX/MEM.RegWrite && (EX/MEM.RegWrAddr != 0) && (EX/MEM.RegWrAddr == ID/EXRegister))
    forward = 10;
else if ((MEM/WB.RegWrite && (MEM/WB.RegWrAddr != 0) && (MEM/WB.RegWrAddr == ID/
EXRegister))
    forward = 01;
else
    forward = 00;

```

但需要注意，这样的转发逻辑遗漏了 sw 部分的转发，而教材上的数据通路同样遗漏了这一转发，这使得 sw 的内容依赖前一条指令的计算结果时会出错，因此，经过 EX/MEM 中间寄存器写入内存的 writeData 同样需要被转发，转发逻辑同上。

2.5. 连接外设

在 MEM 阶段访存时进行判断：如果地址的高四位为 0x4，则关闭内存的 memWrite 和 memRead，同时将要地址与要写入内存的值传到总线上，再将内存读出的值修改为外设输入的值。然后，将 CPU 输出的各个信号连接到 LED 和 UART 上、将 UART 的输出（LED 无输出）连接到 CPU 的输入即可。

2.6. LED 的设计

由于本次实验要求用软件译码，LED 的实现没有什么难度，只需要在检测到地址为 0x40000010 时，将 CPU 写入的值直接转发到板上对应的 LED 和 ANO 接口上即可。

2.7. UART 的设计

UART 内部使用 RX 和 TX 两个小模块，直接使用小作业中的代码即可。由于串口每次只能收发 8bit 的数据，且串口的频率远低于 CPU，因此在设计汇编时，需要注意添加适当的等待，且需要对读入的数据做移位的转换。

2.8. 总体设计

经过上面各个模块的内部设计及冒险解决的设计，最终形成了总体的设计图如图 15 所示。

3. 关键代码及文件清单

3.1. 文件清单

- ALU.v：控制 ALU 模块，复用自单周期代码，相比之下增加了几种新的分支指令的支持；
- ALUControl.v：给 ALU 模块生成控制信号，完全复用自单周期代码，无改动；
- code.asm：最终使用的汇编代码，使用测试代码为 Dijkstra 算法，支持串口输入、串口输出、LED 输出；
- Control.v：生成控制信号，与单周期相比增加了几种新的分支指令的支持；
- CPU.v：流水线的主体部分，将内部各个模块组合在一起；
- DataMemory.v：内存模块，由于消耗资源太大（需要 4356 Byte 的内存），因此放弃了单周期中使用寄存器的写法，转而使用 BRAM 的 IP 核；
- ForwardUnit.v：转发模块，用于解决数据冒险；
- generate_test_case.cc：用于生成测试样例，分别会生成 test.dat 用于在 C++ 算法中做测试、以及 test2.txt，用于后续测试的处理；
- Hazard.v：冒险检测单元，用于解决控制冒险；

- `instruction_convert.py`: 将 MARS 生成的十六进制串转为适合写入 `InstructionMemory` 的形式;
- `InstructionMemory.v`: 指令存储器, 内部指令翻译自 `code.asm`;
- `LED.v`: 用于将内存写入的地址转发到数码管上, 以便显示;
- `RegBetween.v`: 级间寄存器, 用于在不同的流水阶段之间传递数据;
- `RegisterFile.v`: 寄存器堆, 直接复用自单周期代码;
- `testCPU.v`: 仿真使用的测试代码, 实例化了 `cpu`、`led`、`uart`, 同时现有代码中提供了一组用于串口仿真的代码。为了节省仿真时间, 此处需要将串口改为 10 clks per bit (波特率直接百倍增加了, 只适用于仿真);
- `top.v`: 最终的顶层代码, 实例化了 `cpu`、`led` 和 `uart`, 同时使用了 IP 核的锁相环来分频;
- `top.xdc`: 使用的约束文件, 此处由于使用了锁相环, IP 核为我们自动添加了时钟约束, 因此不需要另写时钟约束;
- `uart_generate.py`: 基于前面生成的 `test2.txt` 生成 `hex.txt` 和 `out.txt`, 前者用于上板后输入到串口中, 后者用于在仿真时模拟串口输入;
- `uart_rx.v`: 串口接收部分, 直接复用自小作业;
- `uart_tx.v`: 串口发送部分, 直接服用自小作业;
- `UART.v`: 串口部分顶层模块, 用于实现串口功能。

3.2. 关键代码

此处列举了部分关键代码, 这些代码或在开发、调试流水线的过程中起到了极大的便利性, 或经过了巧妙的设计, 使我认为有必要在此处一一列举。

3.2.1. 级间寄存器的复用

```
module RegBetween
  #(parameter SIZE=32)
```

使用可变参数的级间寄存器, 从而极大地加强了级间寄存器的复用性

3.2.2. 级间寄存器的刷写

```
always @(posedge clk or posedge rst)
  if (rst)
    dataOut <= 0;
  else if (flush == 2'b00) // no flush
    dataOut <= dataIn;
  else if (flush == 2'b01) // flush
    dataOut <= 0;
  else if (flush == 2'b10) // keep
    dataOut <= dataOut;
  else
    dataOut <= 0;
```

使用信号控制级间寄存器的刷写, 从而解决了 Load-Use 型数据冒险与分支、跳转引起的控制冒险。

3.2.3. 寄存器堆的先写后读

```

assign Read_data1 = (Read_register1 == 5'b00000)? 32'h00000000: (Write_register != 0 &&
RegWrite && (Write_register == Read_register1))? Write_data: RF_data[Read_register1];
assign Read_data2 = (Read_register2 == 5'b00000)? 32'h00000000: (Write_register != 0 &&
RegWrite && (Write_register == Read_register2))? Write_data: RF_data[Read_register2];

```

通过直接转发，实现了寄存器堆的先写后读，无须再解决 WB -> ID 的冒险。

3.2.4. 指令翻译

```

with open("default.txt", "r") as f: # here default.txt includes the machine code
    lines = f.readlines()

with open("convert.txt", "w") as f:
    i = 0
    for line in lines:
        f.write(f"9'd{i}: Instruction <= 32'h" + line.replace("\n", "") + ";\\n")
        i += 1

```

使用 python 对 MARS 生成的机器码做了二次处理，从而大幅提高了仿真 Debug 效率。

3.2.5. 转发单元

```

assign forward = ((EXMEMWrite == 1) && (EXMEMAddr != 0) && (EXMEMAddr == target))?
2'b10:(MEMWBWrite == 1) && (MEMWBAddr != 0) && (MEMWBAddr == target))?2'b01:2'b00;

assign aluSrcA = (ALUSrc1_EX)? {27'h00000, shamt}: readData1_EX;
assign aluInputA = (forwardCtlA==2'b00 || ALUSrc1_EX)?aluSrcA:(forwardCtlA==2'b01)?
writeData:(forwardCtlA == 2'b10)?ALUResult_MEMORY:0;

```

通过转发单元解决了各类数据冒险问题。

3.2.6. 冒险单元

```

assign PCFlush = ((IDEXRt == IFIDRs || IDEXRt == IFIDRt) && memRead)? 2'b10: 2'b00;
assign IFIDFlush = ((IDEXRt == IFIDRs || IDEXRt == IFIDRt) && memRead)? 2'b10: (PCSrcID
== 2'b01 || PCSrcID == 2'b10 || (zero && branch_EX))? 2'b01: 2'b00;
assign IDEXFlush = ((zero && branch_EX) || ((IDEXRt == IFIDRs || IDEXRt == IFIDRt) &&
memRead))? 2'b01: 2'b00;

```

通过生成控制冒险信号与级间寄存器的配合，解决了控制冒险问题。

3.2.7. 连接外设

```

assign readMemData = (ALUResult_MEMORY[31:28] == 4'h4)? deviceData: memData;
assign memAddr = (ALUResult_MEMORY[31:28] == 4'h4)? ALUResult_MEMORY : 0;
assign memDat = (ALUResult_MEMORY[31:28] == 4'h4)? readData2_MEMORY : 0;
// here memAddr and memDat are output to Device

```

通过按照地址检测判断来连接外设。

3.2.8. BCD 译码的汇编实现

```

li $t0, 63
sw $t0, 0($zero)

```

```

li $t0, 6
sw $t0, 4($zero)
li $t0, 91
sw $t0, 8($zero)
li $t0, 79
sw $t0, 12($zero)
# others are duplicated
li $t0, 113
sw $t0, 60($zero)

sll $s2, $s2, 2 # s2 stores a 4-bit number
lw $s2, 0($s2)

```

通过巧妙的内存地址映射，避免了 BCD 译码漫长的 if-else 打表过程。²

3.2.9. UART 的读取

```

readn:
    lui $t1, 0x4000
    addi $t2, $t1, 0x0020
readn_1:
    lw $t3, 0($t2)
    andi $t3, $t3, 0x00000004
    beq $t3, $zero, readn_1
    addi $t4, $t1, 0x001c
    lw $t5, 0($t4)
    sll $t5, $t5, 8 # t5 = n[31:24]
readn_2:
    lw $t3, 0($t2)
    andi $t3, $t3, 0x00000004
    beq $t3, $zero, readn_2
    addi $t4, $t1, 0x001c
    lw $t6, 0($t4)
    or $t5, $t5, $t6 # t5 = n[31:16]
    sll $t5, $t5, 16
readn_3:
    lw $t3, 0($t2)
    andi $t3, $t3, 0x00000004
    beq $t3, $zero, readn_3
    addi $t4, $t1, 0x001c
    lw $t6, 0($t4)
    sll $t6, $t6, 8
    or $t5, $t5, $t6 # t5 = n[31:8]
readn_4:
    lw $t3, 0($t2)
    andi $t3, $t3, 0x00000004
    beq $t3, $zero, readn_4
    addi $t4, $t1, 0x001c
    lw $t6, 0($t4)
    or $t5, $t5, $t6 # t5 = n[31:0]

```

²根据室友的实际体会可知，自己手动用 if-else 打表大概要三三百行才能译码，此处只用了 40 行代码就成功译码，不可谓没有大幅降低重复工作成本。此外，根据使用 <https://godbolt.org> 汇编可知，其实也可以通过地址打表的方式，实现类似 C 中 case 的效果，但是终究不如我的代码来得巧妙、行数来得少，不再赘述。

利用忙等待 + 循环的方法，解决了 CPU 和 UART 串口之间通信速率相差巨大的问题（9600 波特率与 80+MHz 频率，二者相差上百倍），连续读取了四个 Byte，没有在串口端产生过多的额外资源消耗。此外，写串口也采用了类似的忙等待方法。

3.2.10. 生成测试样例

```
std::uniform_int_distribution<int> distrib(1, 200); // 随机数范围，也可以改分布情况
cin >> n; // 输入点数
for (int i = 0; i < n; i++)
    for (int j = 0; j < i; j++)
    {
        if (i == j)
            continue;
        graph[i * 32 + j] = unsigned(distrib(gen));
        if (graph[i * 32 + j] < 60 && graph[i * 32 + j] > 40) // 用于调整删掉多少边
            graph[i * 32 + j] = -1;
        graph[j * 32 + i] = graph[i * 32 + j];
    }
```

生成了不同规模、不同分布的测试样例，从而有利于对流水线工作的正确性做全面测试。

3.2.11. 仿真/上板串口样例处理

```
with open("out.txt", "w") as f:
    for line in lines:
        num = int(line)
        if num == -1:
            num = 10
        f.write(
            f"for (i = 3; i >= 0; i = i - 1) begin\n#100 rx = 0;\nfor (j = 0; j < 8; j
= j + 1) begin\n#100 rx = data{num}[8 * i + j];\nend\n#100 rx = 1;\nend\n"
        )

with open("hex.txt", "w") as f:
    for line in lines:
        if len(line) < 8:
            line = "0" * (8 - len(line)) + line
        f.write(line + " ")
```

利用 Python 脚本，分别基于前面的样例生成了用于仿真的输入和用于输入到串口中的输入，有效避免了重复的机械性劳动。

4. 仿真结果及分析

4.1. 输入数据集准备

我们利用前面提到的 C++ 代码随机生成一组 32 个测试样例点的数据（这也是我的 CPU 支持的上限）。各边长度分布从 1~9 不等，且有超过一半的概率为 -1，这能够比较好地体现出我们 CPU 的性能。具体的测试用图如图 1 所示，这张图上的单源最短路结果如图 2 所示。

```

32
0 6 -1 -1 7 1 -1 4 -1 -1 5 8 7 -1 8 -1 -1 -1 -1 3 -1 -1 -1 -1 1 4 2 -1 -1 -1 -1
6 0 5 3 -1 -1 -1 -1 -1 -1 -1 -1 4 -1 -1 8 -1 -1 5 6 -1 -1 -1 -1 -1 9 -1 -1 2 -1 -1 -1
-1 5 0 7 -1 -1 8 -1 -1 3 -1 -1 9 -1 -1 5 6 -1 -1 -1 -1 -1 9 -1 -1 2 -1 -1 -1 -1
-1 3 7 0 -1 -1 -1 -1 2 -1 2 -1 5 6 5 9 -1 -1 6 9 -1 -1 -1 -1 -1 9 -1 -1 7 -1 3
7 -1 -1 -1 0 6 3 6 8 -1 -1 7 7 -1 8 4 -1 -1 7 -1 3 2 5 3 8 -1 -1 -1 -1 -1 3 7 -1
1 -1 -1 -1 6 0 -1 7 -1 8 1 -1 -1 -1 -1 1 -1 2 -1 -1 9 -1 7 9 -1 -1 -1 -1 3 7 -1
-1 -1 -1 -1 3 -1 0 5 -1 6 1 7 9 -1 2 -1 -1 8 -1 -1 8 5 -1 3 -1 -1 6 5 6 -1 6 9
4 -1 8 -1 6 7 5 0 -1 3 7 -1 3 -1 -1 -1 -1 -1 3 9 3 -1 -1 7 9 -1 -1 3 7 -1 -1
-1 -1 -1 2 8 -1 -1 -1 0 5 4 8 -1 2 -1 6 4 3 -1 9 -1 6 7 9 -1 -1 1 7 -1 -1 1 -1
-1 -1 -1 -1 8 6 3 5 0 -1 -1 2 -1 5 -1 -1 -1 -1 2 -1 -1 5 -1 -1 5 -1 7 -1 5 4 7 -1
5 -1 3 2 -1 1 1 7 4 -1 0 8 -1 5 -1 -1 4 -1 -1 -1 5 6 -1 1 -1 1 -1 8 1 4 6
8 -1 -1 -1 7 -1 7 -1 8 -1 8 0 6 4 -1 8 1 -1 -1 1 6 6 -1 9 -1 -1 2 3 -1 -1 -1 -1
7 -1 -1 5 7 -1 9 3 -1 2 -1 6 0 7 3 -1 7 -1 4 -1 -1 -1 4 5 -1 6 -1 -1 1 6 -1
-1 -1 9 6 -1 -1 -1 2 -1 5 4 7 0 3 1 7 -1 -1 4 -1 1 -1 4 -1 -1 5 9 8 3 -1 -1
8 -1 -1 5 8 -1 2 -1 -1 5 -1 -1 3 3 0 2 -1 -1 1 -1 3 -1 -1 3 -1 -1 8 -1 7 1 -1
-1 4 -1 9 4 -1 -1 -1 6 -1 -1 8 -1 1 2 0 -1 5 -1 7 5 5 3 2 -1 6 -1 3 5 7 -1 4
-1 -1 5 -1 -1 1 -1 4 -1 4 1 7 7 -1 -1 0 -1 -1 4 3 8 9 -1 -1 2 -1 7 -1 -1 3 7
-1 -1 6 -1 -1 -1 8 -1 3 -1 -1 -1 -1 -1 5 -1 0 8 -1 6 1 5 -1 1 9 -1 1 3 -1 -1 6
-1 8 -1 6 7 2 -1 -1 -1 -1 -1 4 -1 1 -1 -1 8 0 -1 6 -1 -1 -1 -1 -1 6 1 3 -1 -1
3 -1 -1 9 -1 -1 -1 3 9 2 -1 -1 4 -1 7 4 -1 -1 0 -1 -1 -1 -1 2 -1 6 -1 6
-1 -1 -1 -1 3 -1 8 9 -1 -1 5 6 -1 -1 3 5 3 6 6 -1 0 -1 -1 -1 5 -1 1 8 1 4 -1 -1
-1 5 -1 -1 2 9 5 3 6 -1 6 6 -1 1 -1 5 8 1 -1 1 -1 0 -1 -1 -1 -1 -1 7 5 -1
-1 -1 -1 -1 5 -1 -1 -1 7 5 -1 -1 -1 -1 -1 3 9 5 -1 -1 -1 -1 0 -1 6 -1 -1 1 9 -1 -1 9
-1 -1 -1 3 7 3 -1 9 -1 1 9 4 4 3 2 -1 -1 -1 -1 -1 0 -1 8 5 -1 -1 -1 6
1 -1 9 -1 8 9 -1 -1 7 -1 5 -1 -1 -1 -1 -1 -1 -1 -1 5 -1 6 -1 0 1 5 -1 9 -1 8 -1
4 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
2 -1 -1 9 -1 -1 6 -1 -1 7 -1 2 6 5 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 4 2 -1 -1 -1 5 -1 7 -1 -1 -1 3 -1 9 8 3 7 1 6 2 8 -1 -1 -1 -1 9 -1 0 1 -1 -1
-1 2 -1 -1 -1 -1 6 3 -1 5 8 -1 -1 8 -1 5 -1 3 1 -1 1 -1 9 -1 9 1 -1 1 0 1 6 7
-1 1 -1 7 -1 3 -1 7 -1 4 1 -1 1 3 7 7 -1 -1 3 6 4 7 -1 -1 -1 8 9 -1 1 0 -1 -1
-1 8 -1 -1 3 7 6 -1 1 7 4 -1 6 -1 1 -1 3 -1 -1 -1 -1 5 -1 -1 8 -1 -1 -1 6 -1 0 6
-1 -1 -1 3 -1 -1 9 -1 -1 -1 6 -1 -1 -1 -1 4 7 6 -1 6 -1 -1 9 6 -1 7 7 -1 7 -1 6 0

```

图 1: 测试样例

```

0 4 5 4 5 1 3 4 3 5 2 3 4 4 4 5 2 2 3 3 3 7 3 1 2 2 3 3 3 4 7
total: 107

```

图 2: 对应答案

4.2. 仿真验证正确性

我们采用前面提到的 Python 脚本将这 1025 个数据用串口仿真的形式发送到 CPU 上。如果计算结果正确，LED 上应该显示为 006b。经过一段时间的仿真，可以看到这个结果是正确的，如图 3 所示。

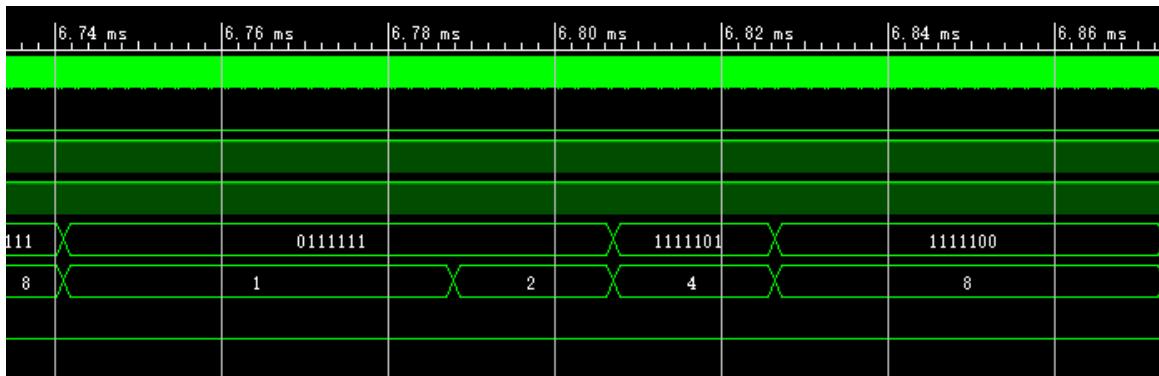


图 3: 仿真波形

此处各个位的持续时间不等长的原因，是因为我的循环显示部分是采取从 0 加上去到某一个数，然后再减下来的做法。由于 `blt` 和 `bge` 编译出来的指令数相差 1，因此持续时间也会发生对应的变化。对于实际效果，则没有明显影响。

4.3. CPI 计算

通过 MARS 运行原始理论课汇编大作业上的代码，发现结果如图 4 所示。可以看到指令数是非常庞大的，因此 `syscall` 被替换等误差均可被忽略。

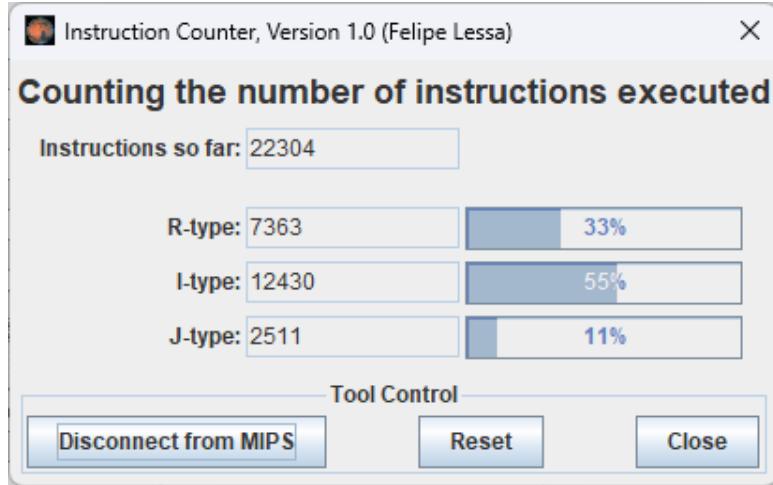


图 4: 指令数

根据汇编代码可知，从结束使用 RX 串口到开始使用 TX 串口这一段时间，可近似为 CPU 上执行 Dijkstra 算法的总运行时间，头尾的几条加减法指令均可以在庞大的总指令数面前被忽略掉。在仿真波形上，我们容易观察到这段时间，如图 5 所示。

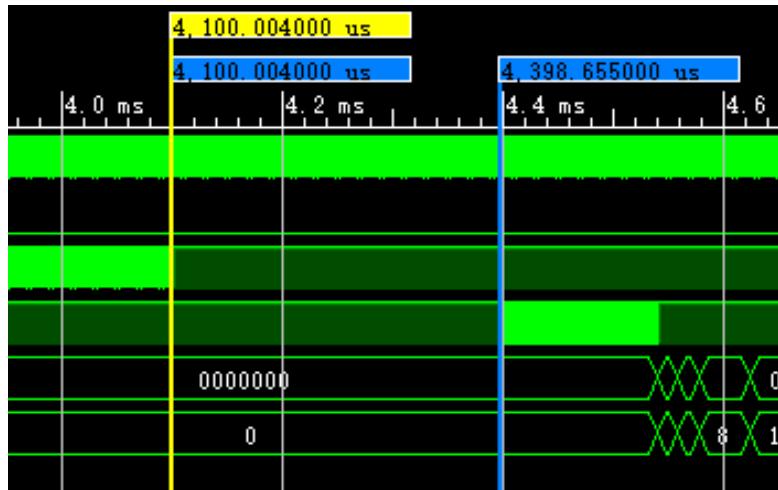


图 5: 运行时间

那么可以计算出，总时钟数（仿真采用周期为 10ns 的时钟）约为

$$\frac{4398655 - 4100004}{10} \approx 29865$$

从而易知

$$CPI = \frac{29865}{22304} = 1.339$$

值得注意的是，此处的 CPI 严格依赖于测试使用的数据。由于我采取静态分支预测不跳转的策略来进行分支处理，因此遇到大规模跳转的情况下，CPI 将会有所上升。例如，我们这里采用了一个 32×32 的图，而我们给定的长度的下限为 1，上限为 9。也就是说，我们遇到分支跳转的频率会非常高，此时性能就会收比较严重的影响。根据不同的测试样例，这个 CPI 在 1.28 到 1.34 之间上下浮动。例如，在实验室线下验收时，我用一个相对比较小的图（10 个点）跑出了 1.29 的 CPI。如果分支进一步减少，那么 CPI 还可以降低。在更小的测试样例上，我也跑出过

1.28 左右的 CPI。因此，此处的 CPI 只能大致确定一个量级，具体 CPI 的上下浮动是意义不大的。

5. 综合情况

5.1. 资源占用

如图 6 所示，由于使用了 IP 核技术，因此 LUT 占用和 FF 占用大规模降低，取而代之的是 BRAM 和锁相环的占用。此处同样贴上单周期的资源占用供对比参考。可以看出，LUT 和寄存器都有所减少。但实际上，只看内存部分以外的 LUT 和寄存器使用是有所增加的：以寄存器为例，单周期只有 $8425 - 8192 = 233$ 个，而流水线却占用了 1467 个，这主要是因为流水线存在级间寄存器，这大量消耗了资源。

Name	^ 1	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Block RAM Tile (50)	Bonded IOB (210)	BUFGCTRL (32)	PLLE2_ADV (5)
top		1866	1562	196	66	1.5	16	2	1
cpu (CPU)		1784	1467	196	66	1.5	0	0	0
freq (FreqDivide)		0	0	0	0	0	0	2	1
led (LED)		0	12	0	0	0	0	0	0
uart (UART)		82	83	0	0	0	0	0	0

图 6: 流水线资源占用

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Bonded IOB (210)	BUFGCTRL (32)
CPU	2970	8425	1024	512	100	1
register_file1 (Register...)	341	224	0	0	0	0
data_memory1 (DataM...)	2214	8192	1024	512	0	0
alu1 (ALU)	404	0	0	0	0	0

图 7: 单周期资源占用

5.2. 静态时序分析

由于转发路径判断逻辑复杂、电路规模庞大、Net Delay 极高等原因，我的电路起初主频很低，只有约 70MHz。经过了一定的优化尝试，我发现除非大规模修改转发路径，否则很难再进一步优化主频。经过一番查阅资料，我发现，由于 Vivado 在实现一级上有更多的优化方案，因此存在综合不满足时序约束、但实现满足时序约束的情况。同时，我和往届学长³进行了一定的交流，了解到可以通过设置更严格的时序约束，去“逼迫”Vivado 采用更激进的优化策略，如打破层级化设计等，来优化主频。经过这样一番尝试，我发现果然可行。最终，我发现在将锁相环分出的时钟调整至 84.5MHz 时，虽然综合部分没有满足时序约束（图 8），但是实现部分却满足了时序约束（图 9）。因此，在 84.5MHz 的主频下，我的设计上板后是可以满足时序约束的。

³ 此处参考了无 06 班刘浩然学长在优化主频上采取的方法，即不断试探着去设置更严格的时序约束

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): -0.916 ns	Worst Hold Slack (WHS): 0.091 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): -95.489 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 327	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2828	Total Number of Endpoints: 2828	Total Number of Endpoints: 1574

Timing constraints are not met.

图 8: 综合时序约束

Design Timing Summary

Setup	Hold	Pulse Width
Worst Negative Slack (WNS): 0.041 ns	Worst Hold Slack (WHS): 0.098 ns	Worst Pulse Width Slack (WPWS): 3.000 ns
Total Negative Slack (TNS): 0.000 ns	Total Hold Slack (THS): 0.000 ns	Total Pulse Width Negative Slack (TPWS): 0.000 ns
Number of Failing Endpoints: 0	Number of Failing Endpoints: 0	Number of Failing Endpoints: 0
Total Number of Endpoints: 2828	Total Number of Endpoints: 2828	Total Number of Endpoints: 1574

图 9: 实现时序约束

6. 硬件调试情况

我们额外生成一组测试数据（之前的数据过小，不够具有代表性）进行测试。选用的测试数据及其结果如图 10、图 11 所示：

```
32
0 -1 425 -1 -1 151 -1 -1 -1 968 -1 -1 -1 -1 370 539 134 -1 -1 -1 201 239 769 -1 460 -1 580
-1 0 -1 -1 -1 946 762 -1 117 -1 -1 -1 327 -1 -1 -1 277 542 959 35 740 526 -1 489 -1 509 -1 710 129 436 -1
425 -1 0 654 686 696 -1 -1 -1 -1 755 742 145 -1 678 -1 -1 406 -1 -1 42 772 -1 -1 -1 622 819 16 -1 -1 615
-1 -1 654 0 -1 -1 230 527 502 665 48 -1 -1 290 917 -1 -1 433 687 -1 386 774 -1 34 -1 -1 844
-1 -1 686 -1 0 153 -1 224 -1 -1 613 581 -1 930 -1 703 -1 -1 -1 250 924 -1 -1 -1 422 376 385 -1 269 -1 -1 516
-1 946 696 -1 153 0 -1 377 -1 229 629 -1 716 -1 -1 1 23 -1 801 -1 727 -1 352 -1 613 635 -1 598 -1 250 2 244
151 762 -1 -1 -1 -1 0 -1 280 73 159 -1 -1 662 -1 696 235 689 -1 -1 13 -1 -1 174 -1 -1 -1 -1 956 58
-1 -1 -1 -1 224 377 -1 0 525 714 -1 526 473 484 918 171 631 921 218 545 166 908 886 89 91 -1 -1 -1 313 -1 -1 658
-1 117 -1 230 -1 -1 280 525 -1 -1 489 -1 -1 565 -1 -1 980 -1 -1 797 79 11 -1 -1 946 752 539 951 656 254 -1
-1 -1 -1 527 -1 229 73 714 -1 0 493 -1 -1 435 688 419 293 197 -1 989 -1 207 683 -1 -1 341 -1 -1 -1 55 -1 503
-1 -1 -1 501 613 629 159 -1 489 493 0 -1 -1 85 930 506 764 547 -1 45 502 -1 -1 771 -1 585 -1 363 923 -1 802 -1
968 -1 755 665 581 -1 -1 520 -1 -1 -1 0 838 385 -1 638 784 -1 235 -1 704 522 -1 356 -1 547 -1 -1 562 595 949 -1
-1 327 742 40 -1 716 -1 473 -1 -1 -1 838 0 -1 -1 -1 778 523 507 -1 650 278 -1 -1 -1 302 310 -1 -1 7
-1 -1 145 -1 930 -1 602 484 565 435 85 385 -1 0 104 641 -1 177 -1 241 -1 591 -1 -1 -1 796 630 179 -1 112 -1 -1
-1 -1 -1 -1 -1 -1 918 -1 688 930 -1 -1 104 0 511 513 -1 -1 238 95 456 -1 -1 436 363 671 656 844 489 -1 -1 896 -1
-1 -1 678 -1 703 -1 696 171 -1 419 506 638 -1 641 511 0 -1 194 -1 -1 575 -1 -1 118 -1 -1 123 -1 -1 -1 809
-1 -1 -1 572 -1 23 235 632 980 293 764 784 -1 -1 513 -1 0 7 251 53 66 -1 -1 252 448 -1 481 136 806 -1 -1 152 570
-1 277 -1 -1 -1 689 921 -1 197 547 -1 778 177 -1 194 7 0 854 148 884 -1 1 977 904 -1 835 -1 949 -1 939 -1
370 542 486 290 -1 801 -1 218 -1 -1 -1 235 523 -1 238 -1 251 854 0 78 803 -1 -1 998 284 -1 -1 918 799 151 -1 -1
539 959 -1 917 250 -1 -1 545 -1 989 45 -1 507 241 95 -1 53 148 78 0 -1 -1 -1 491 -1 -1 522 -1 -1 1 38
134 35 -1 924 727 13 166 797 -1 502 704 -1 -1 456 575 60 884 803 -1 0 -1 869 -1 -1 967 -1 -1 -1 275 -1 -1
-1 740 42 -1 -1 -1 905 79 207 -1 522 650 591 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 734 496 -1
-1 526 772 433 -1 352 -1 886 11 683 -1 -1 278 -1 436 -1 252 -1 -1 -1 869 -1 0 662 363 -1 -1 396 59 -1 256 -1
-1 -1 687 -1 -1 80 -1 -1 771 356 -1 -1 363 118 448 977 998 -1 -1 667 0 -1 741 284 121 64 728 -1 -1
201 489 -1 -1 422 613 174 91 -1 -1 -1 -1 671 -1 -1 904 284 491 -1 -1 363 -1 0 -1 118 101 515 91 967 -1
239 -1 -1 386 376 635 -1 -1 946 341 585 547 -1 796 650 -1 481 -1 -1 967 -1 -1 741 -1 0 -1 -1 -1 419 -1
769 509 622 774 385 -1 -1 -1 752 -1 -1 -1 630 844 123 136 835 -1 -1 -1 284 118 -1 0 -1 353 -1 -1 -1
-1 -1 819 -1 -1 598 -1 -1 539 -1 363 -1 302 179 489 -1 806 -1 918 522 -1 -1 395 121 101 -1 -1 0 165 -1 -1 -1
460 710 16 34 269 -1 -1 312 951 -1 923 563 319 -1 -1 -1 949 799 -1 -1 -1 59 64 515 -1 -1 353 165 0 491 266 291
1 129 -1 -1 -1 250 -1 -1 650 55 -1 505 -1 112 -1 -1 -1 151 -1 275 734 -1 728 91 -1 -1 -1 401 0 121 -1
580 436 -1 -1 -1 256 -1 254 -1 802 949 -1 -1 896 -1 152 939 -1 -1 -1 496 256 -1 967 419 -1 -1 266 121 0 208
759 -1 615 844 516 244 58 658 -1 503 -1 -1 7 -1 809 570 -1 -1 38 -1 -1 -1 -1 291 -1 208 0
```

图 10: 新测试样例

```
0 169 302 252 376 223 147 292 286 220 288 556 212 373 338 401 200 207 321 243 134 344 297 350 201 239 319 307 286 275 225 205
total: 8588
```

图 11: 新对应答案

将其通过串口发送到板子上，发现板子上的 LED 正确显示了结果 (0x218C)，同时串口也正确发回了第 0 个点到各个点的距离，如图 12⁴、图 13 所示。

⁴此处由于拍摄环境较暗，手机自动曝光效果较差，因此使用了较快的快门速度拍摄。



图 12: LED 显示情况

```
00 00 00 00 00 00 00 09 00 00 01 2E 00 00 00 FC 00 00 01 78 00 00 00 DF 00 00 00 93 00 00 01 24 00 00 01 1E 00 00 00 DC 00 00  
01 20 00 00 02 2C 00 00 00 D4 00 00 01 75 00 00 01 52 00 00 01 91 00 00 00 C8 00 00 00 CF 00 00 01 41 00 00 00 F3 00 00 00 86  
00 00 01 58 00 00 01 29 00 00 01 5E 00 00 00 C9 00 00 00 EF 00 00 01 3F 00 00 01 33 00 00 01 1E 00 00 01 13 00 00 00 E1 00 00  
00 CD
```

图 13: 串口发回数据

7. 心得体会

本次大作业约消耗了我 25 小时的时间。其中前 5 小时我对照理论课 PPT 完成了一个简易的流水线（不连接任何外设），接下来 10 小时都是在对简易流水线进行痛苦的仿真 Debug，随后 1 小时将流水线连接到了 LED 上，接下来 5 小时将流水线连接到了串口上，最后 4 小时进行各类测试、小幅优化工作。在这一过程中，我有了不少的体会，此处一一列出。

7.1. 前期设计的重要性

关于顶层设计部分，我最初是以数逻理论课 PPT 的流水线设计为主要参考对象，进行了初版的设计；这带来了一些问题：PPT 上的部分设计和我的设计是不符合的（例如 PPT 上没有出现过在 EX 阶段分支的电路图），我在编写代码时在这部分就直接“脑内设计电路”编写，最终不出所料地出现了错误，这浪费了我大量的 Debug 时间；甚至 PPT 上的电路图还存在错误（PPT 的电路图没有考虑 sw 指令的数据冒险问题），在最初我是没有意识到这一点的，直到最后逐条指令 Debug 时才发现这个问题。也就是说，这种“设计不完善就开始写代码”的问题，给我的 Debug 造成了巨大困扰。

除此之外，有部分细分模块的设计，我其实也是没有想好就开始做了。例如，我们在数逻理论课上总说“寄存器堆的先写后读”，这到底是怎么实现的？在写代码时我其实没有想过这个问题，只是想着“自然地写出来就一定支持吧！”就直接复用了单周期的代码。然而，这个显然是不能自动实现的，最终这个 Bug 依然是在后期 Debug 的时候才被发现修复了。

总的来说，可以看到，一个清晰的前期设计，可以极大程度减少 Debug 过程中浪费的精力。

7.2. 中期 Debug 中的种种困难

通过仿真来调试代码，看起来和正常编程语言的调试没什么本质区别，可实际写起来才发现，这一过程是有本质差异的。例如，正常编程语言调试中可以方便地设置条件断点等一系列方便调试的方法，而 Vivado 仿真就没法做到这些，这给仿真带来了极大的困难。因此，最后我只能把 PC 的波形、各个寄存器的波形、部分关键内存的波形和各种关键控制信号的波形全部拖出来画出来，仿一段时间后去找我们关注的部分，这是极为浪费时间的。不幸中的万幸，是我的代码完全由自己手动编写，因此整个流程中，我对各个信号都应该是极度清楚的，因此还存在 Debug。如果有部分参考“版本”且没有完全理解的话，那出现了 BUG 就完全无法解决了。

7.3. 切忌想当然

在通过串口从计算机向 CPU 发送数据时，我遇到了奇怪的现象：仿真没有任何问题，但是上板怎么也跑不出来正确结果，只有发送的第一个数据能被正常读取。经过反复尝试，我定位到了问题：我发送的数据大概格式为 "num1\n num2\n num3\n..."，而这个回车导致了问题所在（向串口发送了错误的冗余数据）。经过一番 Debug（大概浪费了 3 个小时才发现这个问题），问题得到了解决（改用 "num1 num2 num3..." 这样的格式发送）。这说明，调试过程要多注意细节，切忌想当然。

7.4. 软硬件结合解决问题

在调试 UART 的部分时，我发现，由于我的设计特点，UART 向计算机发送数据是额外需要两个周期来更新状态的。这就导致，如果我前一个时钟写入要写的数据（`0x40000018`），后一到两个周期去读 UART 是否可以发送的状态（`0x40000020`），那读出来的数据就会出现没有及时更新的问题。为此，我选择在写入一个 Byte 之后插入两个 `nop` 再继续操作，这就通过软件硬件相结合来解决了这一问题。

7.5. Vivado 的 BUG

我自己并没有遇到 Vivado 的任何 BUG，但是由于我的大作业完成时间比较早，因此我实际上给几位比较熟悉的同学的开发过程提供了一定的建议。其中，我发现他们有很多人都遇到了 Vivado 的各种 BUG，即仿真波形正确、无 Critical Warning，但是上板结果不对。有的同学使用了一些玄学方法来解决，例如更新 Vivado 版本到 2023.1、在所有涉及冒险的指令后加入 `nop`（但实际上仿真是不需要这样 `nop` 的）等。可见，硬件调试和软件不同，还是要一点玄学和运气的。

附录

控制信号真值表

		PCSrc[1:0]	Branch	RegWrite	RegDst[1:0]	MemRead	MemWrite	MemoReg[1:0]	ALUSrc1	ALUSrc2	ExtOp	LuOp	BranchType[2:0]
lw	0	0 1	0	1	0 1	0 1	0 1	0 1	1 0	1 1	1 0	x	
sw	0	0 0	x	0	1 0	1 x	0 0	0 0	1 0	1 1	0 0	x	
lui	0	0 1	0	0	0 0	0 0	0 0	0 0	1 0	0 1	1 0	x	
add	0	0 1	1	0	0 0	0 0	0 0	0 0	0 0	x x	x x	x	
addu	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
sub	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
subu	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
addi	0	0 1	0	0	0 0	0 0	0 0	0 0	1 1	1 0	x x	x	
addiu	0	0 1	0	0	0 0	0 0	0 0	0 0	1 1	0 0	x x	x	
and	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
or	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
xor	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
nor	0	0 1	1	0	0 0	0 0	0 0	0 0	x x	x x	x x	x	
andi	0	0 1	0	0	0 0	0 0	0 0	0 0	1 0	0 0	x x	x	
sll	0	0 1	1	0	0 0	0 0	1 0	x x	x x	x x	x x	x	
srl	0	0 1	1	0	0 0	0 0	1 0	x x	x x	x x	x x	x	
sra	0	0 1	1	0	0 0	0 0	1 0	x x	x x	x x	x x	x	
slt	0	0 1	1	0	0 0	0 0	0 0	x x	x x	x x	x x	x	
sltu	0	0 1	1	0	0 0	0 0	0 0	x x	x x	x x	x x	x	
slti	0	0 1	0	0	0 0	0 0	0 0	1 1	1 0	0 0	x x	x	
sltiu	0	0 1	0	0	0 0	0 0	0 0	1 1	0 0	x x	x x	x	
beq	0	1 0	x	0	0 x	0 x	0 0	0 0	1 0	1 0	0 4		
bne	0	1 0	x	0	0 x	0 x	0 0	0 0	1 0	1 0	0 3		
blez	0	1 0	x	0	0 x	0 x	0 0	0 0	1 0	1 0	0 1		
bgtz	0	1 0	x	0	0 x	0 x	0 0	0 0	1 0	1 0	0 0		
bltz	0	1 0	x	0	0 x	0 x	0 0	0 0	1 0	1 0	0 2		
j	1	x 0	x	0	0 0	x x	x x	x x	x x	x x	x x	x	
jal	1	x 1	2	0	0 2	x x	x x	x x	x x	x x	x x	x	
jr	2	x 0	x	0	0 x	x x	x x	x x	x x	x x	x x	x	
jalr	2	x 1	1	0	0 2	x x	x x	x x	x x	x x	x x	x	

图 14: 控制信号

流水线-外设示意图

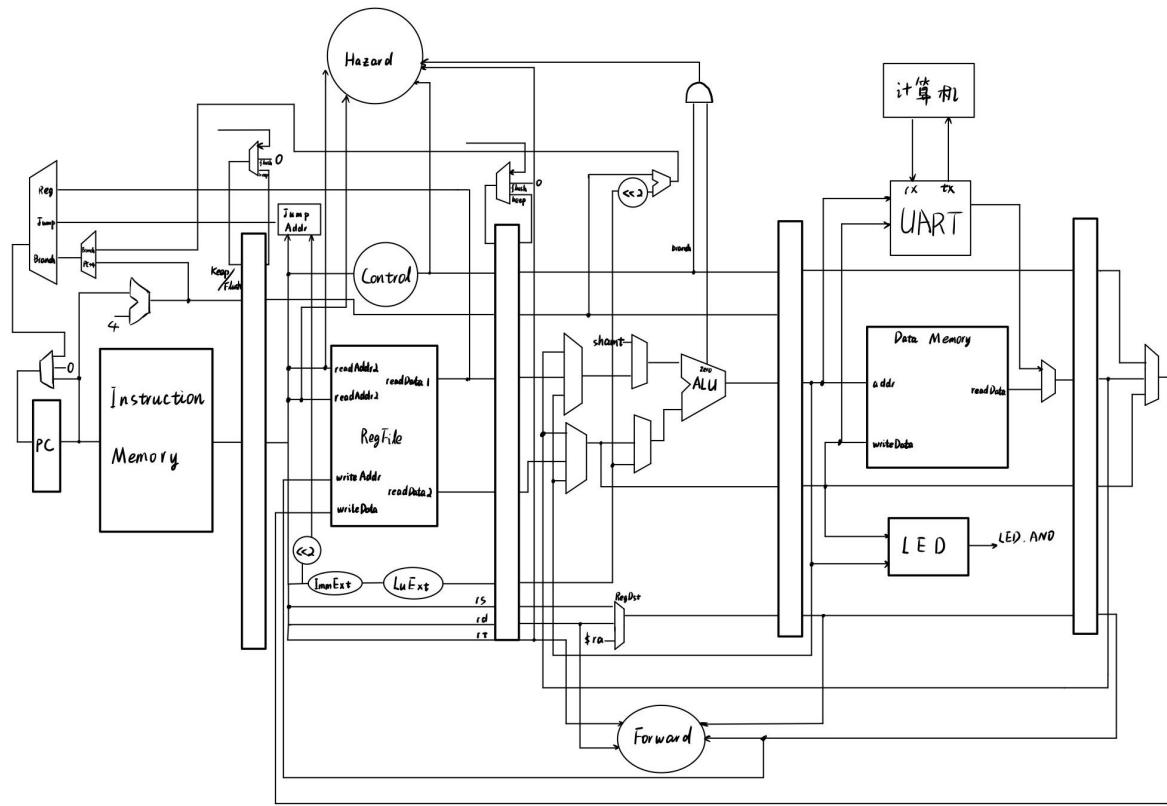


图 15: 数据通路示意图