基于 logisim 的单周期硬布线 24 指令 MIPS-CPU 设计

学　　号	E12214052	专　　业	计算机科学与技术	姓　　名	赵宸宇
实验日期	2024 年 10 月 10 日	教师签字		成　　绩	

摘要

本实验设计了一个 24 指令单周期硬布线的 MIPsCPU，该 CPU 设计的特点有：

syscall 指令停机方式的实现采用**截断 clk** 的方式实现，相比 PPT 的 PC 控制方案，clk 方式实现了**全机停机**，提升了电路整体的停机的鲁棒性。停机后重新启动的 go 中断的实现采用**强制执行 syscall 的下一条命令**，刷掉 halt 状态的方式实现。

在硬件电路设计上采用了**冗余设计**，可以很方便地添加 24 指令之外的 RIJ 指令。

使用三态门、T 触发、移位器等组件，优化了电路设计。使用三态门和或门的配合**替换了多路选择器**。

对 PC 的功能进行了加强，实现了**地址自减运行** (图 4)。

使用原本的带显示窗口的 reg file，以方便后续实验观察运行情况。

大量使用隧道，增强电路的逻辑性和可调试性，减少横穿电路的信号线的数量。

实验的主要目的：

在 Logisim 中实现能运行 **24 条基础指令**，能运行**标准测试程序**的 Mips 单周期硬布线 CPU。

附加项目：优化 systemcall 实现机制

实验的主要任务：

1. 构建 MIPSCPU **数据通路**
2. **单周期硬布线控制器**实现
3. 软硬件**测试联调**
4. 附加项目：优化 system call 实现机制

实验产出：

1. 自主设计电路图
2. 头哥网通关
3. 实验报告（LaTeX）
4. 支持材料
5. **github | gitee** 更新请见`https://gitee.com/cslearnerer/AHU-CSHT`或`https://github.com/YUCHENYUXI/CPUDesign`

目录

一、【实验内容】

1.1 构建 MIPSCPU 数据通路

MIPsCPU 通常由数据通路和硬件控制电路两个部分组成。这两者相辅相成，共同协作，达成了 ISA 的正常运转。

这里本实验根据 HUST 课程给定的数据通路原理图图 1，结合先前 8 指令单周期硬布线 CPU 的数据通路设计方案、多周期的数据通路的简化方案，对 24 指令 CPU 的数据通路电路图进行重构设计。

对于数据通路的构件，这里选用原版 reg file，以方便后续实验观察运行情况；其余部件则灵活搭配，按需增减。

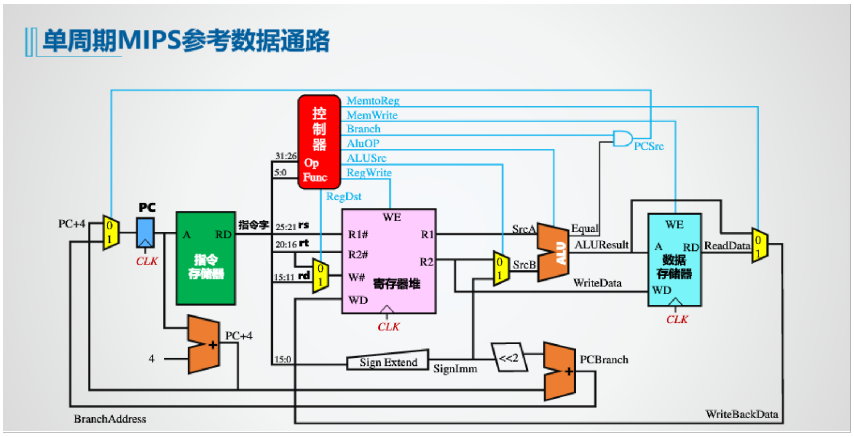

图 1　经典数据通路

通过基础设计、大量调试、动态指令扩展（数据通路部分）、联调调通这几步工作，本实验的最终数据通路得到了实现（如下图）。

本实验设计的数据通路（图 2）采取冗余设计，具有非常好的鲁棒性和可扩展性，以下是几个特色部分的展示。

1.1.1 取指令和指令分解电路设计

这个部分 (图 4) 的主要功能是根据 PC 提供的字节地址，自动计算字地址。然后将字地址打入指令 RAM，取得当前指令。同时将指令分解为 RIJ 型指令的各个组成成分，送往其他电路使用。

为了方便调试，这里还提供了反汇编、动态显示等功能。

其下的电路是清零电路（清 PC、LED 等的数据）、和地址自减模式开关。

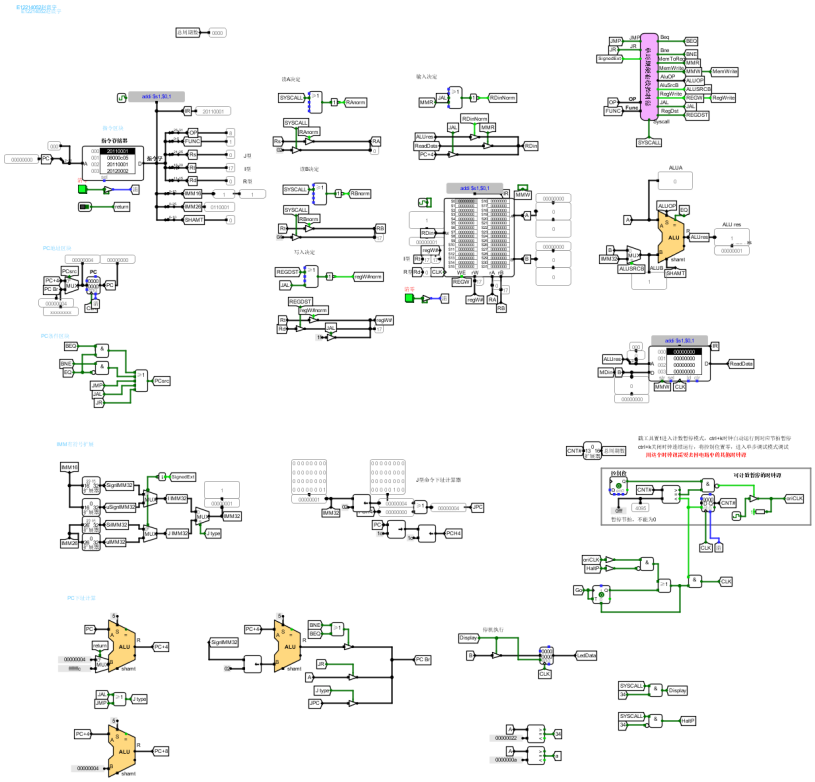

图 2　数据通路

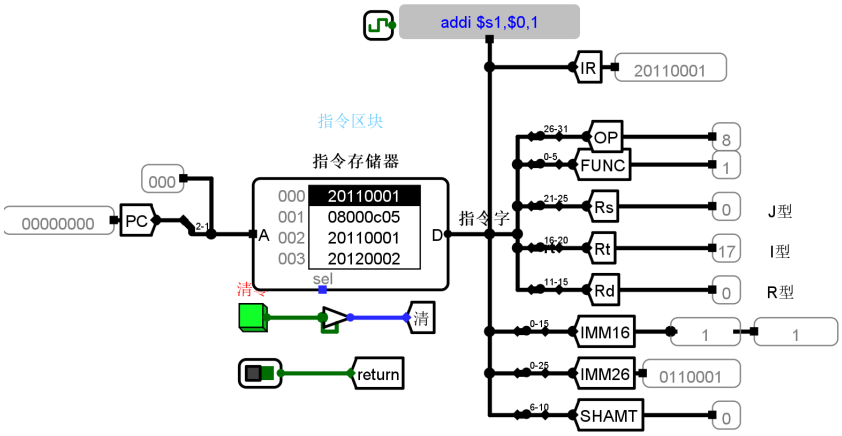

图 3　指令处理

1.1.2 Reg File 电路设计

这是一个集成调试窗口是本次数据通路设计的核心区块之一，显示了当前 reg file 的值、当前执行指令、关键性控制信号的值、当前的写入目的 reg、当前的应写入目的 reg（分 I 或 R 型指令）、当前的两个读 reg 地址和两个读 reg 的读出数值、时钟状态、写入内存状态等信息。同时提供了快捷清零按钮（需要配合 section 3.1的代码）。

在调试 24 条指令时，大部分对指令执行情况的判断均可在这一集成调试区域完成。

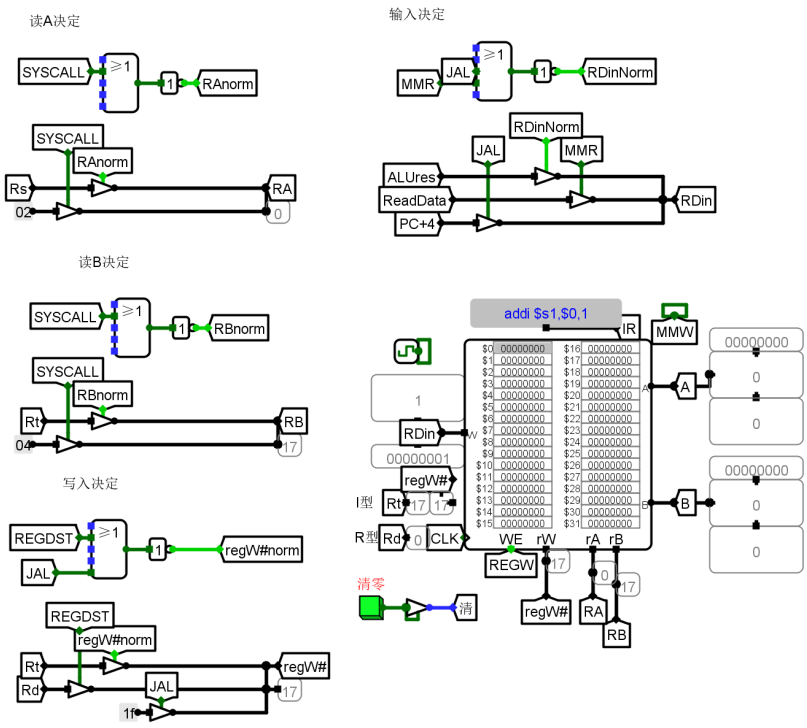

图 4　**REG** 综合调试区域

1.1.3 SYSCALL-CLK 截止方案的电路设计

当执行 syscall 指令时，如果 rs 寄存器读出的值不是 34（0x22）那么就需要触发 halt 即停机状态。

本实验采用的设计方案（图 5）是：通过阻断 clk 信号，自锁 halt 状态实现停机。这种方案的好处是能够实现全机的真停机，确保任何电路不会在停机期间按时序继续运行。

在电路设计方面，如果 $2reg 的值不是 22H 那么就会在 syscall 指令周期触发 haltP 信号，该信号会阻断 clk 信号，使得 clk 保持低电平，将机器停在 syscall-haltP 状态。

当需要继续运行时，触发 GO 信号，该信号通过 T 触发器发出一个尖峰脉冲，该脉冲将充当 clk 信号让 PC 强制指向下一条指令。此时 syscall 指令已经过去，haltP 信号无法维持，造成自锁状态消失，释放了对 clk 的锁定。具体原理如下图所示（图 5）。

1.2 单周期硬布线控制器实现

基于 HUST 课程和 PPT，不难得到 24 指令硬布线控制器的真值表（l3 8 到 24 指令/实验/单周期硬布线控制器表达式自动生成 2020-3-12.xlsx）和各项的表达式。然后手动将表达式导入 logisim 生成硬布线控制电路（见图 6和图 7）。

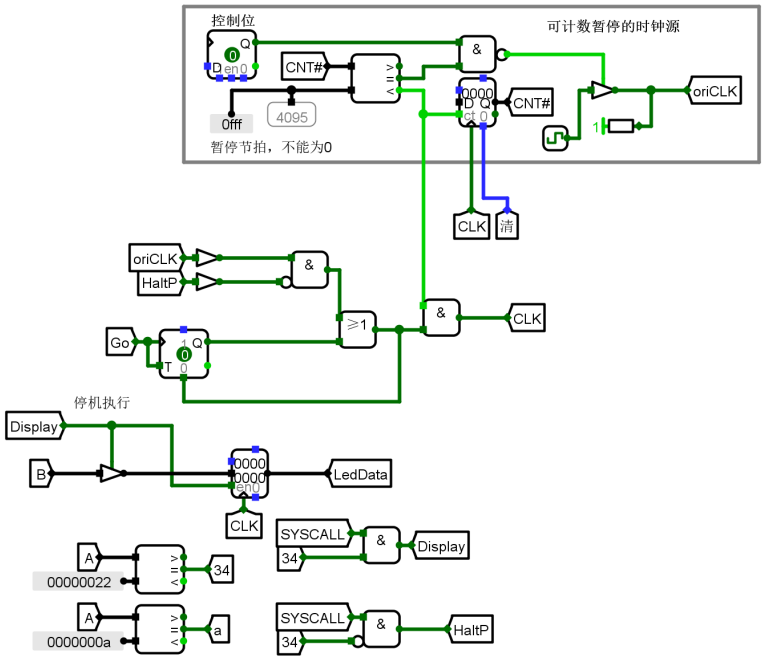

图 5　**syscall** 停机控制电路

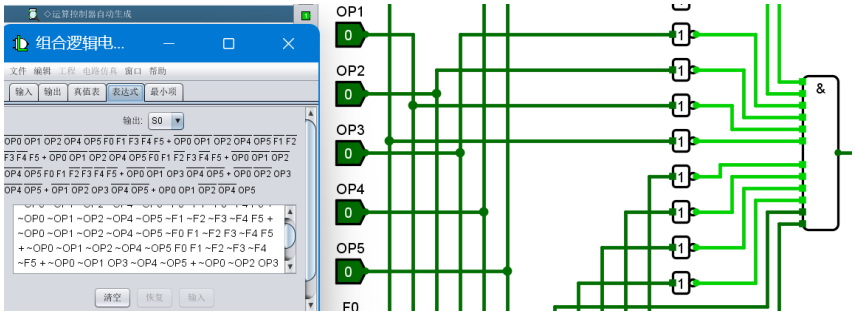

图 6　运算器控制

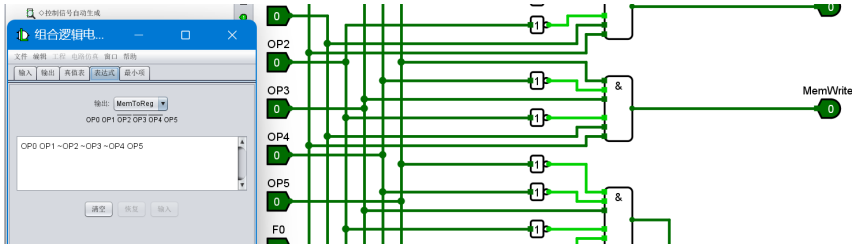

图 7　控制信号

1.3 软硬件测试联调

经过了基础数据通路设计、进阶数据通路优化、硬布线控制器设计、联调控制电路控制信号优化四个设计阶段，并查阅了大量文档、使用了大量辅助工具，本实验才通过了头哥网测试（图 8），告示实验终于完成，同时得到了如上文所示的特色电路设计。

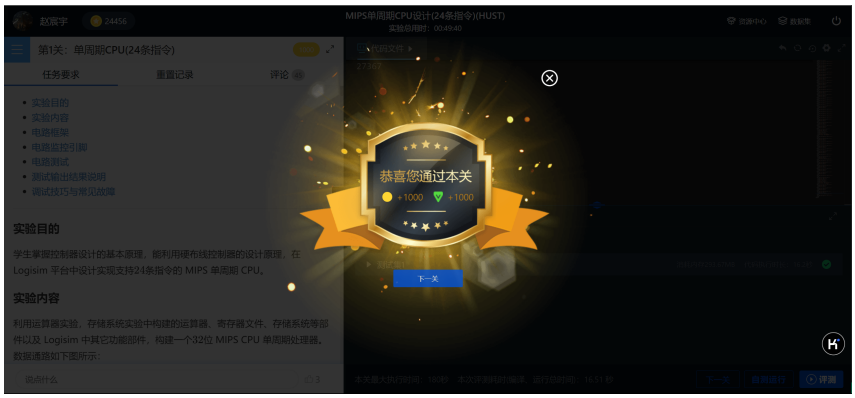

图 **8**　头哥网通关

二、 【小结讨论】

2.1 小结

该实验极大的丰富了我对 CPU 设计、逻辑部件的使用方法、CPU 工作原理的理解。

通过本次实验我完成了 24+ 指令的 Mips 单周期硬布线 CPU 数据通路和控制逻辑的设计工作，并完成了调试。

最终的实验电路具有较普通方案更强的可拓展性、鲁棒性、兼容性。

同时在本次实验的过程中我还习得了用 T 触发器触发正脉冲的方法，从而实现了 clk halt 后电路能够通过长短不定的 go 信号重新运行。

2.2 讨论

该电路仍然存在需要改进的地方。

第一是电路逻辑图不够规整，隧道的命名需要优化；

第二是探究各个电路设计中是否存在一种更好的方法来替代当前的设计，从而达到更好的性能、安全性；

第三是继续寻找该 CPU 设计中的其他不足之处，并在接下来的实践中不断优化这些不足。

三、 附件

3.1 自建 24+ 指令（含一些扩展指令）调试用代码清单

```
https://www.eg.bucknell.edu/~csci320/mips_web/

## SLL
sll s1 s2 0x4
Binary: 00000000000100101000100100000000
Hex: 0x  00128900
```

```
ok

## SRA
sra s1 s2 0x4
Binary: 00000000000100101000100100000011
Hex: 0x  00128903
ok

## SRL
srl s1 s2 0x4
Binary: 00000000000100101000100100000010
Hex: 0x  00128902


## ADD
add s1 s2 s3
Binary: 00000010010100111000100000100000
Hex: 0x  02538820


## ADDU
addu s1 s2 s3
Binary: 00000010010100111000100000100001
Hex: 0x  02538821


## SUB
sub s1 s2 s3
Binary: 00000010010100111000100000100010
Hex: 0x  02538822


## AND
and s1 s2 s3
Binary: 00000010010100111000100000100100
Hex: 0x  02538824


## OR
or s1 s2 s3
Binary: 00000010010100111000100000100101
Hex: 0x  02538825


## NOR
nor s1 s2 s3
Binary: 00000010010100111000100000100111
Hex: 0x  02538827

xor s1 s2 s3
Binary: 00000010010100111000100000100110
Hex: 0x  02538826

## SLT
slt s1 s2 s3
Binary: 00000010010100111000100000101010
```

```
Hex: 0x  0253882A


## SLTU
sltu s1 s2 s3
Binary: 00000010010100111000100000101011
Hex: 0x  0253882B


## JR
jr s1
Binary: 00000010001000000000000000001000
Hex: 0x  02200008


## SYSCALL
syscall
Binary: 000000XXXXXXXXXXXXXXXXXXX001100
Hex: 0x  0XXXXXXC


## J
j 0x123
Binary: 00001000000000000000000100100011
Hex: 0x  08000123


## JAL
jal 0x123
Binary: 00001100000000000000000100100011
Hex: 0x  0C000123


## BEQ

Result
beq s1 s2 0x123
Binary: 00010010001100100000000100100011
Hex: 0x  12320123
31  26  25  21  20  16  15  0
BEQ s1  s2  offset
000100  10001 10010 0000000100100011
6 5 5 16



## BNE
bne s1 s2 0x123
Binary: 00010110001100100000000100100011
Hex: 0x  16320123
31  26  25  21  20  16  15  0
BNE s1  s2  offset
000101  10001 10010 0000000100100011
6 5 5 16


```

```
119
120
121 ## ADDI
122 addi s1 s2 0x123
123 Binary: 00100010010100010000000100100011
124 Hex: 0x  22510123
125 31  26  25  21  20  16  15  0
126 ADDI  s2  s1  immediate
127 001000  10010 10001 0000000100100011
128 6 5 5 16
129
130
131
132
133 ## ANDI
134 andi s1 s2 0x123
135 Binary: 00110010010100010000000100100011
136 Hex: 0x  32510123
137 31  26  25  21  20  16  15  0
138 ANDI  s2  s1  immediate
139 001100  10010 10001 0000000100100011
140 6 5 5 16
141
142
143
144
145 ## ADDIU
146 addiu s1 s2 0x123
147 Binary: 00100110010100010000000100100011
148 Hex: 0x  26510123
149
150
151 ## SLTI
152 slti s1 s2 0x23
153 Binary: 00101010010100010000000000100011
154 Hex: 0x  2A510023
155
156
157 ## ORI
158 ori s1 s2 0x23
159 Binary: 00110110010100010000000000100011
160 Hex: 0x  36510023
161
162
163 ## LW
164 lw s1 0x32 s2
165 Binary: 10001110010100010000000000110010
166 Hex: 0x  8E510032
167
168
169 ## SW
170
171 sw s1 0x32 s2
172 Binary: 10101110010100010000000000110010
173 Hex: 0x  AE510032
174
```

```
175 ---
```