姓名：朱梓涵 **学号**：24325356

1. 实验目的

本实验旨在深入理解 RISC-V 指令集架构 (ISA) 与单周期 CPU 的工作原理。实验目标包括：1. 设计并实现一个支持部分 RV32I 指令集的单周期 CPU。2. 掌握 CPU 经典五阶段（取指、译码、执行、访存、写回）在单周期模型下的数据通路和控制逻辑设计。3. 学习使用 Chisel 硬件描述语言进行模块化硬件设计。4. 通过编写单元测试和集成测试，利用 `chiseltest` 和 GTKWave 波形图验证 CPU 各模块及整体功能的正确性。

2. 实验环境

- **操作系统**: Windows 11
- **开发工具**: IntelliJ IDEA
- **构建工具**: SBT
- **仿真与测试**: Verilator, GTKWave

3. 阶段功能划分

1. **取指 (IF)**: 程序计数器 (PC) 提供指令地址，指令存储器根据该地址输出 32 位指令。PC 在每个时钟周期默认加 4，若遇跳转或分支成功，则更新为目标地址。
2. **译码 (ID)**: 译码单元解析指令，提取操作数（寄存器地址 rs1, rs2）、目标寄存器地址 (rd) 和立即数。主控制器根据指令操作码生成后续阶段所需的所有控制信号。
3. **执行 (EX)**: 算术逻辑单元 (ALU) 根据译码阶段生成的控制信号，对来自寄存器文件或立即数生成器的操作数进行计算。同时，分支跳转的地址计算和条件判断也在此阶段完成。
4. **访存 (MEM)**: 根据译码阶段生成的访存控制信号，与数据存储器进行交互。Load 指令从此阶段读取数据，Store 指令则向此阶段写入数据。
5. **写回 (WB)**: 将执行结果或从数据存储器中读取的数据写回寄存器文件。写回的数据来源由控制信号决定。

4. 模块实现与分析

4.1 取指

- **功能**: 根据 `PC` 的当前值从指令存储器中取出指令，并计算下一周期的 `PC` 值。
- **实现要点**: `PC` 的更新逻辑是核心。它由跳转标志 `jump_flag_id` 控制：若跳转发生，`PC` 更新为 `jump_address_id`；否则，顺序执行，`PC` 更新为 `PC + 4`。
- **代码实现**:

```
// lab1(InstructionFetch)
when(io.jump_flag_id) {
  pc := io.jump_address_id
}.otherwise {
  pc := pc + 4.U
}
// lab1(InstructionFetch) end
```

- **波形图**:

4.2 译码

- **功能**: 解析指令，生成立即数，并为后续阶段提供控制信号。
- **实现要点**: 控制信号的生成逻辑是关键。例如，`memory_read_enable` 仅在 L-type 指令时有效，`wb_reg_write_source` 根据指令类型选择写回数据是来自 ALU、数据存储器还是 PC+4。
- **代码实现**:

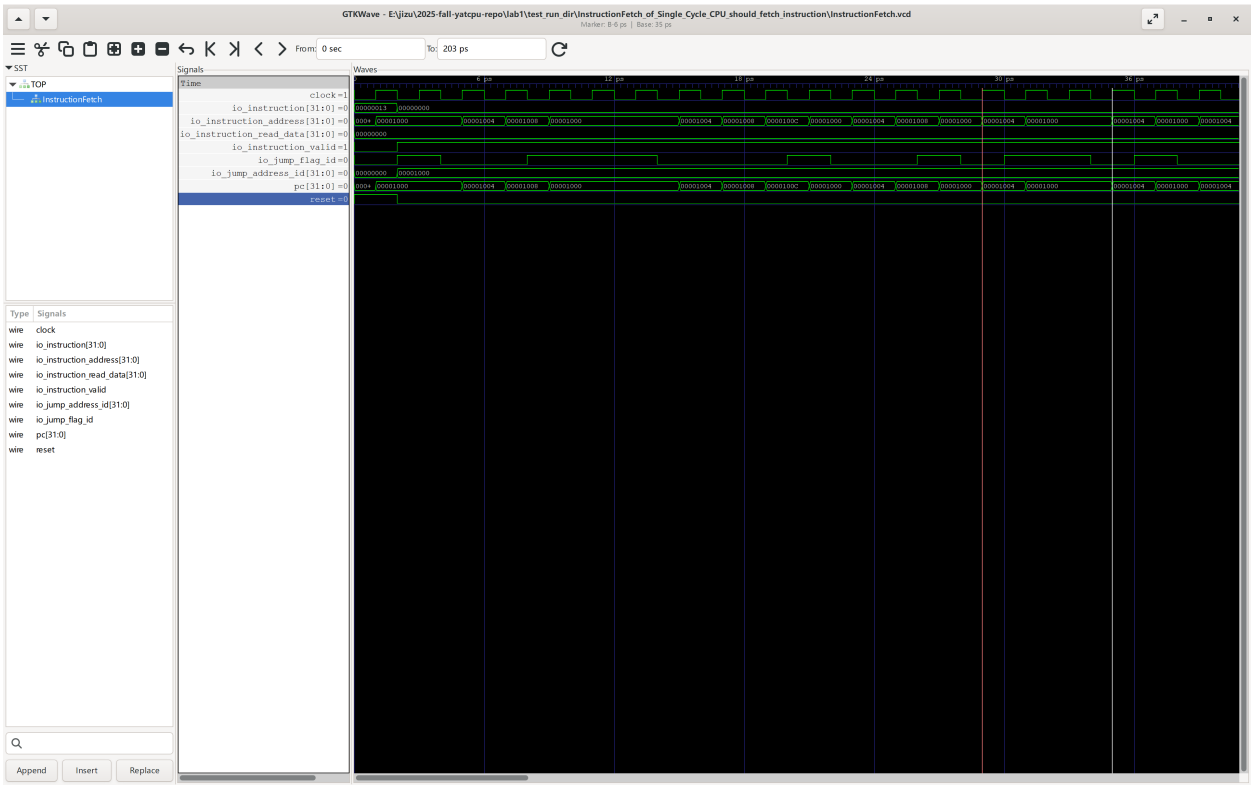

Figure 1: b1

```
// lab1(InstructionDecode)
io.ex_aluop2_source := Mux(opcode === InstructionTypes.RM,
  ALUOp2Source.Register,
  ALUOp2Source.Immediate
)
io.memory_read_enable := opcode === InstructionTypes.L
io.memory_write_enable := opcode === InstructionTypes.S
io.wb_reg_write_source := MuxLookup(
  opcode,
  RegWriteSource.ALUResult
)(
  IndexedSeq(
    InstructionTypes.L -> RegWriteSource.Memory,
    Instructions.jal -> RegWriteSource.NextInstructionAddress,
    Instructions.jalr -> RegWriteSource.NextInstructionAddress
  )
)
// lab1(InstructionDecode) end
```

- **波形图**:

4.3 执行

- **功能**: 执行 ALU 运算，并处理分支和跳转逻辑。
- **实现要点**: ALU 的两个操作数 `op1` 和 `op2` 的来源由译码模块生成的 `aluop*_source` 信号决定，实现了操作数的灵活选择。

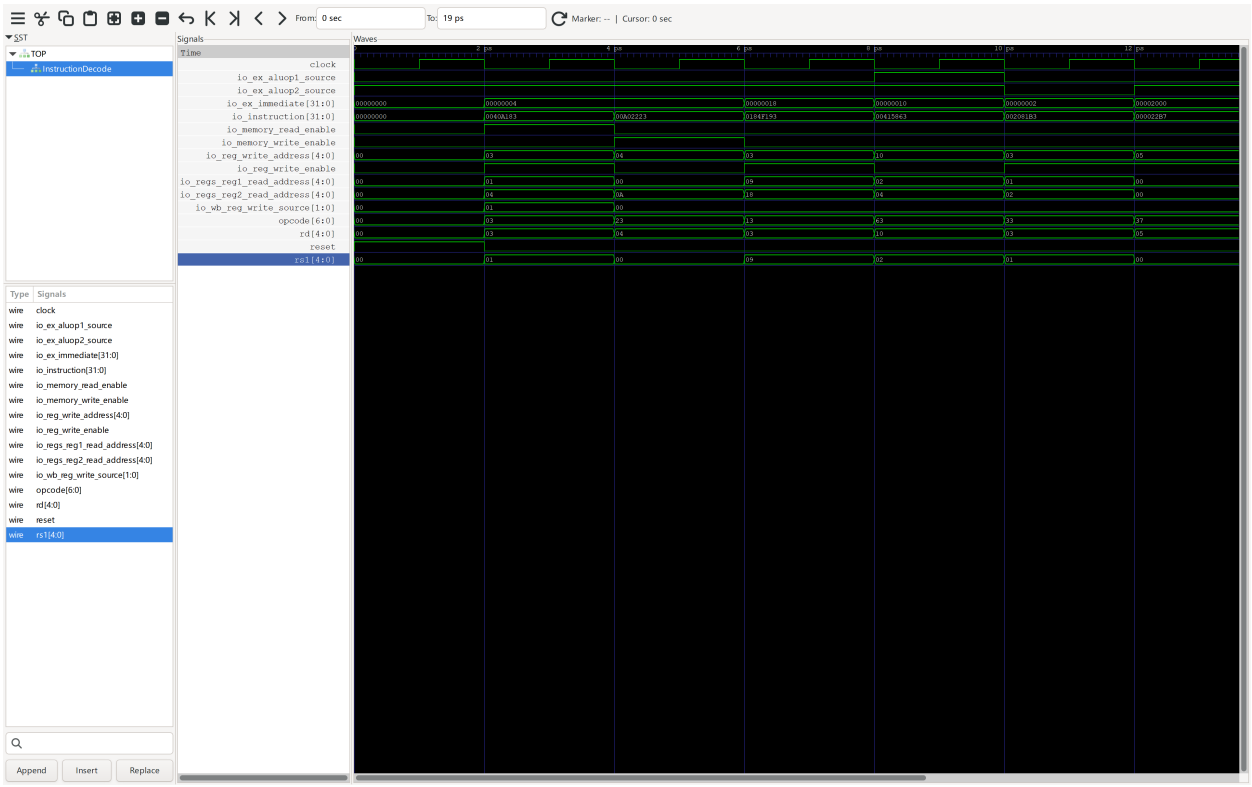

Figure 2: b2

- **代码实现**:

```
// lab1(Execute)
alu.io.func := alu_ctrl.io.alu_funct
alu.io.op1 := Mux(
  io.aluop1_source === ALUOp1Source.Register,
  io.reg1_data,
  io.instruction_address
)
alu.io.op2 := Mux(
  io.aluop2_source === ALUOp2Source.Register,
  io.reg2_data,
  io.immediate
)
// lab1(Execute) end
```

- **波形图**:

4.4 CPU

- **功能**: 实例化取指、译码、执行等所有子模块，并根据数据通路图将它们正确连接起来。
- **实现要点**: 确保数据流和控制流在模块间的正确传递是顶层连接的关键。例如，从取指模块输出的指令和指令地址，需要同时送往译码和执行模块。
- **代码实现**:

```
// lab1(cpu)
ex.io.instruction := inst_fetch.io.instruction
```

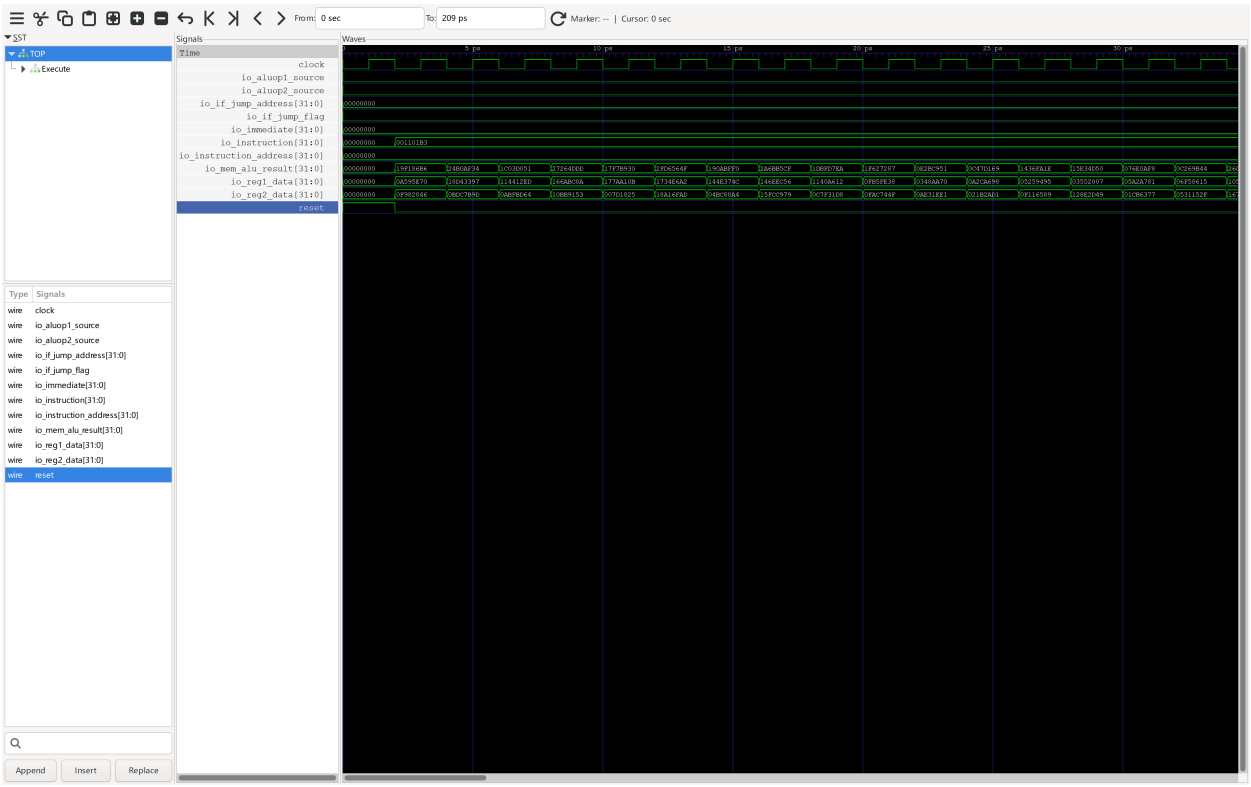

Figure 3: b3

```
ex.io.instruction_address := inst_fetch.io.instruction_address
ex.io.reg1_data := regs.io.read_data1
ex.io.reg2_data := regs.io.read_data2
ex.io.immediate := id.io.ex_immediate
ex.io.aluop1_source := id.io.ex_aluop1_source
ex.io.aluop2_source := id.io.ex_aluop2_source
// lab1(cpu) end
```

5. 测试与结果分析

5.1 单元测试：InstructionDecoderTest 分析

- **L-Type 指令测试分析**:
  - 测试用例首先向译码模块的 `io.instruction` 端口输入一条 L-Type 指令。随后，测试代码使用 `.expect()` 方法断言模块的各个输出端口信号值是否与预设的期望值一致。
  - 对于 `lw` 指令，其核心功能是从内存读取数据并写回寄存器。因此：
    * `memory_read_enable` 必须为 `true`，以启动访存阶段的数据读取。
    * `reg_write_enable` 必须为 `true`，以允许最终结果写入寄存器。
    * `wb_reg_write_source` 必须是 `RegWriteSource.Memory`，因为写回的数据源于数据存储器。
    * `ex_immediate` 必须是指令编码中包含的偏移量（此例中为 32），用于地址计算 `rs1 + imm`。这些 `expect` 值精确地定义了 L-Type 指令在译码阶段应生成的正确行为模式，确保了控制通路的正确性。
- **波形图分析**:

  图片展示了 `lw x10, 32(x5)` 指令（机器码 `0x02028503`）的译码过程。在时钟上升沿之后：

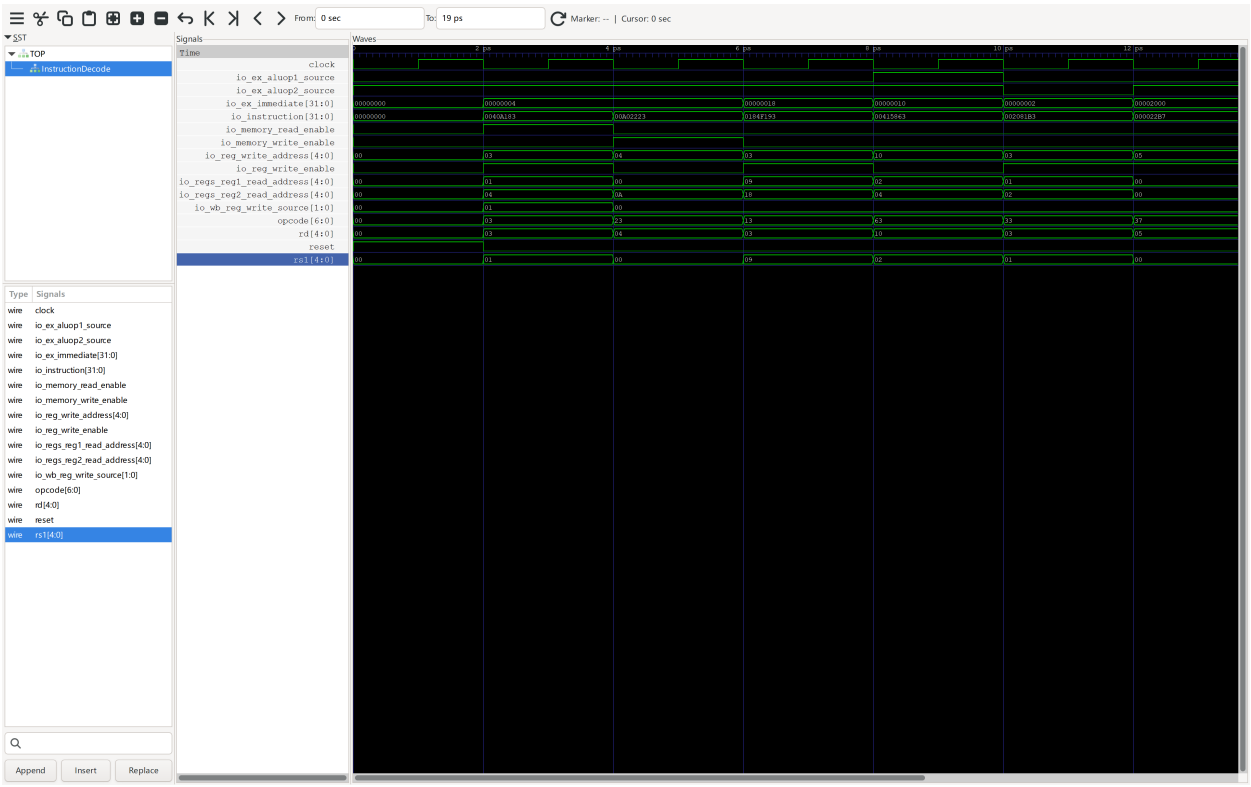

Figure 4: 图 2 lw

1. `id_io_instruction` 端口稳定为 0x02028503。
2. `id_io_memory_read_enable` 和 `id_io_reg_write_enable` 信号被置为高电平 (1)。
3. `id_io_wb_reg_write_source` 信号变为 01 (二进制)，对应 `RegWriteSource.Memory`。
4. `id_io_ex_immediate` 的值为 0x00000020，即十进制的 32。波形图直观地验证了译码模块对于 L-Type 指令的响应与 `expect` 语句的预期完全一致。

5.2 整体测试：CPUTest (以 FibonacciTest 为例)

- **`fibonacci.c` 程序分析**:
  - **程序功能**: 该 C 语言程序定义了一个递归函数 `fib(int a)`，用于计算斐波那契数列的第 `a` 项。其逻辑为：如果 `a` 等于 1 或 2，则返回 1；否则，返回 `fib(a-1)` 与 `fib(a-2)` 之和。`main` 函数是程序的入口点，它调用 `fib(10)` 来计算斐波那契数列的第 10 项，并将最终结果（即 55）存储到内存地址 0x4。

```c
int fib(int a) {
  if (a == 1 || a == 2) return 1;
  return fib(a - 1) + fib(a - 2);
}

int main() {
  *(int *)(4) = fib(10); // 将 fib(10) 的结果写入内存地址 0x4
}
```

  - **Chisel 测试检查方式**: 测试框架首先将 `fibonacci.asmbin` 文件加载到 CPU 的指令存储器和数据存储器中。然后，它驱动 CPU 执行足够多的时钟周期（通过 `c.clock.step(1000)` 大循环），以确保递归计算和内存写入操作全部完成。在仿真结束时，测试代码通过读取数据存储器的调试端口：

```
c.io.mem_debug_read_address.poke(4.U) // 将调试地址设置为 0x4
c.clock.step()                        // 步进一个时钟周期
c.io.mem_debug_read_data.expect(55.U) // 期望从地址 0x4 读出的数据为 55
```

上述语句首先将调试端口的读取地址 `mem_debug_read_address` 设置为 0x4，然后步进一个时钟周期以便数据稳定，最后断言 `mem_debug_read_data` 的值是否等于预期的 55。如果相等，则测试通过，验证了 CPU 能够正确执行递归计算并将结果写入指定内存。

- **波形图分析**:

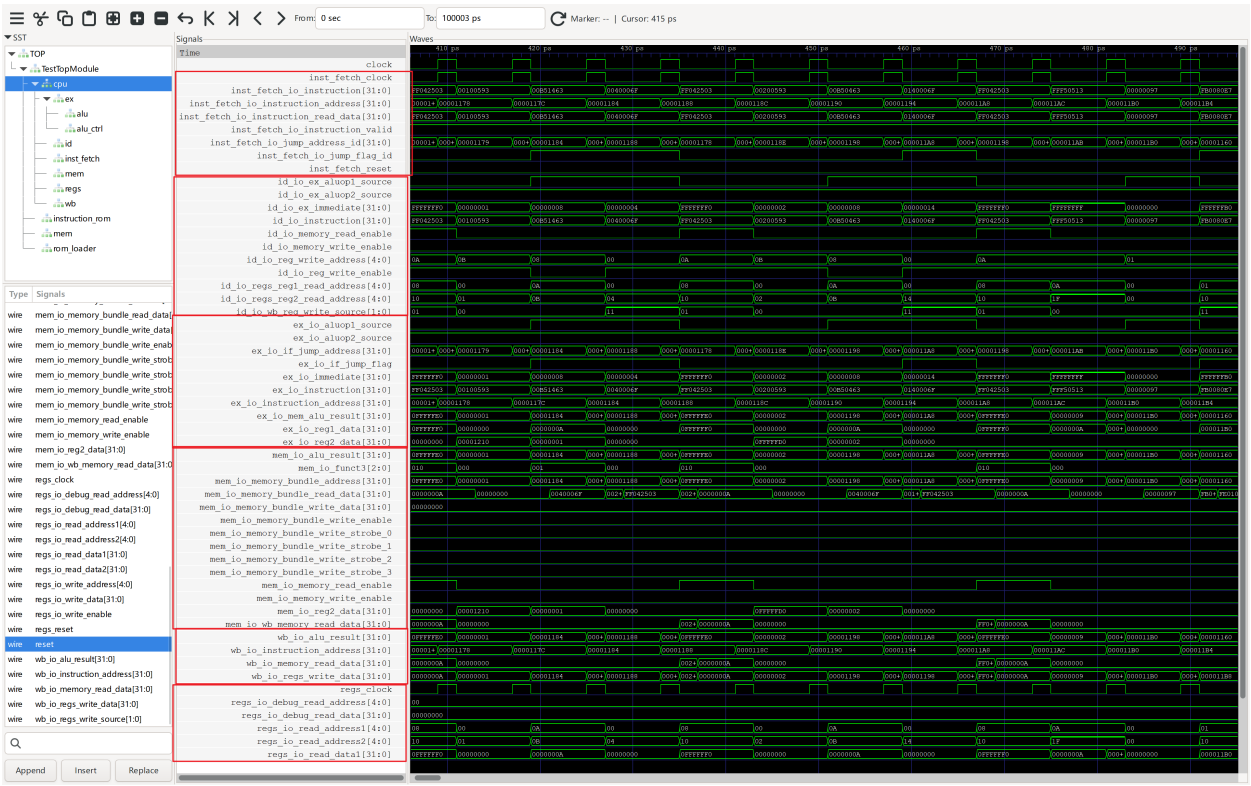

Figure 5: 图 3

图片展示了 `fibonacci` 程序执行的最后几个周期以及结果检查阶段。

1. **执行分析**: 在波形图的前半部分，可以看到 CPU 正在执行 `main` 函数中对 `fib(10)` 的调用，并最终将结果存储到内存地址 0x4。由于斐波那契数列的递归特性，PC (`inst_fetch_io_instruction_address`) 会在函数调用和返回的过程中不断跳转。在程序即将结束时，PC 会最终指向退出或停机指令。
2. **结果检查**: `main` 函数的核心是 `*(int *)(4) = fib(10);` 这条语句，它会在 `fib(10)` 的计算结果 (即 55) 准备好后，将其写入数据存储器的 0x4 地址。在波形图的**最后几个周期**，我们会观察到以下关键信号变化:
   - `mem_io_memory_write_enable` 信号短暂置高 (为 1)，表明数据存储器发生了写入操作。
   - `mem_io_memory_bundle_address` 信号会显示为 0x00000004，确认写入目标是内存地址 0x4。
   - `mem_io_memory_bundle_write_data[31:0]` 信号会显示为 0x00000037 (十六进制的 55)，这是 `fib(10)` 的计算结果。
   - 在这之后，由于测试使用 `c.io.mem_debug_read_address.poke(4.U)` 来读取内存，波形图上 `mem_io_debug_read_address` 会变为 0x4，同时 `io_mem_debug_read_data` (或类似输出调试信号) 会稳定显示出 0x00000037。这些波形特征与 Chisel 测试中 `mem_debug_read_data.expect(55.U)` 的断言完美吻合，证明 CPU 正确地执行了 `fibonacci` 程序并存储了预期结果。

6. 遇到的问题与改进建议

1. **问题：立即数生成规则复杂，指导文档未详细展开。**在实现译码模块时，RISC-V 不同指令格式 (I/S/B/U/J) 的立即数拼接方式各不相同，实验指导对此描述较为简略，导致初期实现困难。
   - **建议**：在实验指导中为每种指令格式提供一个具体的立即数拼接图示或示例代码，能极大帮助理解。
2. **问题：缺少对波形图调试的引导案例。**初次使用 GTKWave 进行硬件调试时，面对海量的信号线，难以快速定位关键信号。
   - **建议**：实验文档中可以附加一个简单的调试案例，例如追踪一条 `add` 指令的数据流，展示如何从取指 PC 开始，逐步添加 `instruction`, `reg_read_data`, `alu_result`, `reg_write_data` 等信号，并解释它们在波形图上的时序关系。
3. **问题：部分模块接口和控制信号的设计意图不够明确。**例如，`ALUOp1Source` 和 `ALUOp2Source` 这类控制信号的设计初衷，需要通过阅读多个模块的代码才能完全理解其作用。
   - **建议**：在代码框架的注释中，对关键的 `object` 或 `Enum` (如 `ALUOp1Source`)，增加注释来说明每个选项的用途和对应的指令场景。

7. 实验结论

通过本次实验，我成功设计并实现了一个功能基本完备的 RISC-V 单周期 CPU。在实现过程中，我深入理解了数据通路与控制信号在指令执行过程中的协同作用，并掌握了模块化的硬件设计思想。通过编写和分析单元测试与集成测试，我学会了如何利用 `chiseltest` 和波形图对硬件设计进行验证和调试。本次实验极大地加深了我对计算机组成原理中理论知识的实践理解，为后续更复杂的处理器设计打下了坚实的基础。

$$e = mc^2$$