Project: 单周期 CPU 设计

1. 实验目的

- 了解 CPU 设计的基本原理
- 设计 CPU 数据通路
- 设计 CPU 控制单元模块
- 搭建单周期 CPU
- sys1心得体会写了不少，嘻嘻

2. 实验环境

- 操作系统：Ubuntu 24.04
- VHDL：Verilog，SystemVerilog

3. 总体设计

3.1 单周期 CPU 数据通路

本设计实现了一个 RV64I 单周期 CPU，在一个时钟周期内完成取指（IF）、译码（ID）、执行（EX）、访存（MEM）、写回（WB）五个阶段。

3.2 模块层次

```
Core.sv                    ← 顶层数据通路
├── Controller.sv          ← 译码、控制信号生成
├── RegFile.sv             ← 寄存器文件 (x0~x31)
├── ImmGen                 ← 立即数生成 (写在 Core.sv 中)
├── ALU.sv                 ← 算术逻辑单元
├── Cmp.sv                 ← 分支比较器
├── DataPkg.sv             ← 写内存数据打包
├── MaskGen.sv             ← 写掩码生成
└── DataTrunc.sv           ← 读内存数据截断/扩展
```

4. 各模块设计与代码分析

4.1 Core.sv —— 顶层数据通路

`Core.sv` 是 CPU 的核心，负责连接所有子模块、定义数据流，实现完整的五阶段数据通路。

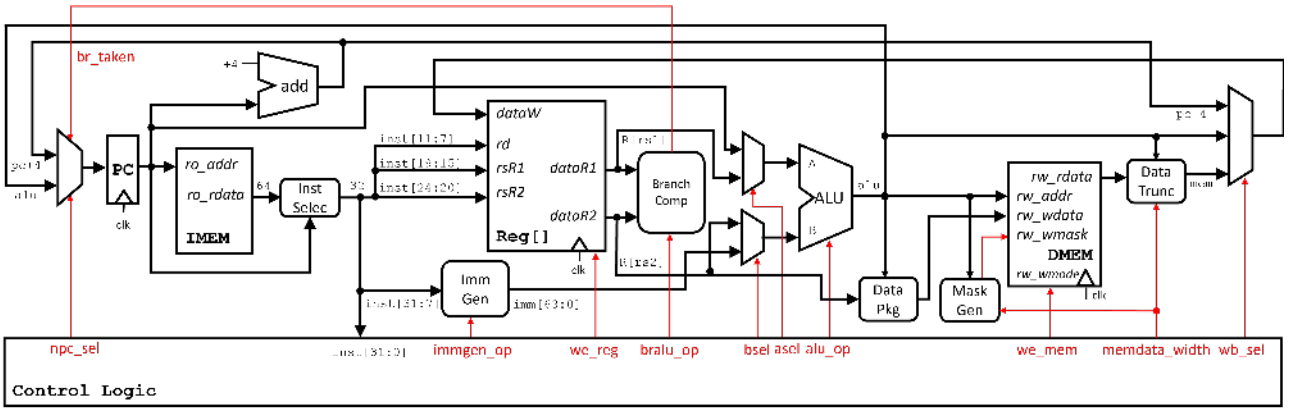

Image 1

- 我的实现逻辑是按照五个阶段的顺序分别写出对应的数据通路：

4.1.1 取指阶段（IF）

```
// 指令内存请求—组合逻辑，始终请求读取
always_comb begin
    imem_ift.r_request_valid = 1'b1;
    imem_ift.r_request_bits.raddr = pc;
    // DRAM 是 64 位宽，一个地址存两条 32 位指令
    // 用 pc[2] 区分取高/低 32 位
    inst = pc[2] ? imem_ift.r_reply_bits.rdata[63:32]
                 : imem_ift.r_reply_bits.rdata[31:0];
end

// PC 更新—时序逻辑
always @(posedge clk or posedge rst) begin
    if (rst)  pc <= 0;
    else      pc <= next_pc;
end
```

设计要点：

- 取指是组合逻辑（`always_comb`），PC 更新是时序逻辑（`always @(posedge clk)`），保证每个周期开始时 PC 已稳定
- `pc[2]` 选择：PC=0/8/16→`pc[2]=0` 取低 32 位；PC=4/12/20→`pc[2]=1` 取高 32 位

4.1.2 下一条 PC 选择

```
wire is_jal  = (opcode == JAL_OPCODE);
wire is_jalr = (opcode == JALR_OPCODE);
assign next_pc = (npc_sel && (br_taken || is_jal || is_jalr))
                 ? alu_res : (pc + 4);
```

设计要点：

此处的关键是区分三种情况：

- **普通指令**（`npc_sel=0`）：`next_pc = pc + 4`
- **条件分支命中**（`npc_sel=1, br_taken=1`）：`next_pc = alu_res`
- **无条件跳转** JAL/JALR（`npc_sel=1, br_taken=0`）：需要额外用 `is_jal || is_jalr` 判断，因为 Controller 对 JAL/JALR 不设置比较操作（`cmp_op = CMP_NO`，故 `br_taken` 恒为 0）

4.1.3 译码阶段（ID）

按照 RISC-V 指令格式（R/I/S/B/U/J）提取对应的寄存器索引、opcode 和 funct 字段。立即数生成由 `ImmGen` 模块完成 (写在 `core.sv` 的末尾了)，Controller 通过 `immgen_op` 控制生成哪种类型的立即数。

4.1.4 执行阶段（EX）

```
// ALU 输入选择
assign A = (alu_asel == ASEL_PC)  ? pc :
           (alu_asel == ASEL_REG) ? read_data_1 : 64'b0;
assign B = (alu_bsel == BSEL_IMM) ? immgen_out :
           (alu_bsel == BSEL_REG) ? read_data_2 : 64'b0;
// 实例化 ALU 和比较器
```

设计要点：

- `alu_asel` 控制 A 端口来源：`ASEL_PC`（PC，用于 AUIPC/JAL/分支）、`ASEL_REG`（rs1）、`ASEL0`（0，用于 LUI）
- `alu_bsel` 控制 B 端口来源：`BSEL_IMM`（立即数）、`BSEL_REG`（rs2）

4.1.5 访存阶段（MEM）

```
// 数据内存请求
always_comb begin
    dmem_ift.r_request_valid       = re_mem;
    dmem_ift.r_request_bits.raddr  = alu_res;
    dmem_ift.w_request_valid       = we_mem;
    dmem_ift.w_request_bits.waddr  = alu_res;
    dmem_ift.w_request_bits.wdata  = dmem_wdata;  // ← 使用 DataPkg 打包后的数据
    dmem_ift.w_request_bits.wmask  = dmem_wmask;  // ← 使用 MaskGen 生成的掩码
end
```

设计要点：

- Store 指令需要通过 `DataPkg` 将寄存器值按字节偏移打包，再用 `MaskGen` 生成字节掩码
- Load 指令读回的数据需要经过 `DataTrunc` 截断/扩展后再写回寄存器

4.1.6 写回阶段（WB）

```
always_comb begin
    case (wb_sel)
        WB_SEL_ALU: wb_val = alu_res;     // 算术结果
        WB_SEL_MEM: wb_val = read_data;   // 内存加载（经 DataTrunc 处理）
        WB_SEL_PC:  wb_val = pc + 4;      // JAL/JALR 的返回地址
        default:    wb_val = '0;
    endcase
end
```

`wb_sel` 由 Controller 根据指令类型决定。JAL/JALR 需要写回 `pc+4`（返回地址）。

4.2 Controller.sv —— 控制单元

Controller 采用二**段译码**方法设计。第一段根据 opcode 识别指令大类，第二段结合 funct3/funct7 细化控制信号。

4.2.1 第一段译码：指令类型识别

```
wire is_load   = opcode == LOAD_OPCODE;      // 0000011
...
wire is_regw   = opcode == REGW_OPCODE;      // 0111011
```

二段译码的优势：复杂度从 $M \times N$ 降到 $M + K$（M = 指令种类数，N = 控制信号数，K = 译码表中输出为 1 的条目数）。

4.2.2 第二段译码：控制信号生成

寄存器/内存使能信号：

```
assign we_reg = is_load || is_imm || is_reg || is_jal
              || is_jalr || is_lui || is_auipc || is_immw || is_regw;
assign we_mem = is_store;
assign re_mem = is_load;
assign npc_sel = is_branch || is_jal || is_jalr;
```

- `we_reg`：所有需要写回寄存器的指令
- `we_mem`：仅 store 指令
- `re_mem`：仅 load 指令
- `npc_sel`：分支和跳转指令需要选择非顺序的 PC

立即数类型选择：

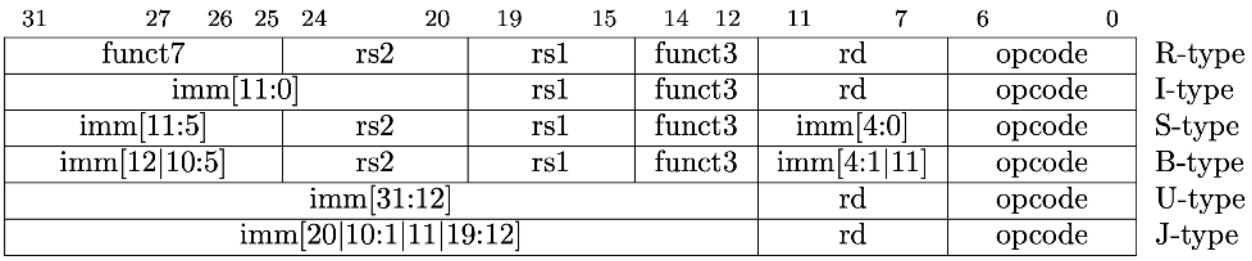

Image 2

```
assign immgen_op =
    (is_load || is_imm || is_immw || is_jalr) ? I_IMM  :
    is_store  ? S_IMM  :
    is_branch ? B_IMM  :
    is_jal    ? UJ_IMM :
    (is_lui || is_auipc) ? U_IMM : IMM0;

```

ALU 输入和写回选择：

```
// ALU 输入 A 选择
assign alu_asel = (is_jal || is_auipc || is_branch) ? ASEL_PC :
                  is_lui ? ASEL0 : ASEL_REG;

// ALU 输入 B 选择
assign alu_bsel = (is_reg || is_regw) ? BSEL_REG : BSEL_IMM;

// 写回数据选择
assign wb_sel = is_load ? WB_SEL_MEM :
                (is_jal || is_jalr) ? WB_SEL_PC : WB_SEL_ALU;
```

访存操作类型：

```
always_comb begin
    mem_op = MEM_NO;
    if (is_load) begin
        case (funct3)
            LB_FUNCT3:  mem_op = MEM_B;
            LH_FUNCT3:  mem_op = MEM_H;
            ...
        endcase
    end else if (is_store) begin
        ...
```

4.3 ALU.sv —— 算术逻辑单元

ALU 支持 RV64I 的全部运算操作：

设计要点：

- 移位量：RV64I 的 64 位移位取 `b[5:0]`（6 位），32 位字移位取 `b[4:0]`（5 位）

- `ADDW/SUBW/SLLW/SRAW` 需要**符号扩展**到 64 位（`{{32{w32[31]}}, w32}`）
- `SRLW` 需要**零扩展**到 64 位（`{32'b0, w32}`）
- `$signed()` 和 `>>>` 用于有符号比较和算术右移

4.4 RegFile.sv —— 寄存器文件

32 个 64 位寄存器，x0 恒为 0：

```
data_t register [1:31];  // x1 - x31, x0 硬连线为 0
// 写端口（时序逻辑）
always @(posedge clk or posedge rst) begin
...
// 读端口（组合逻辑）
always_comb begin
    read_data_1 = (read_addr_1 == 0) ? '0 : register[read_addr_1];
    read_data_2 = (read_addr_2 == 0) ? '0 : register[read_addr_2];
end
```

设计要点：

- 写使能 `we` 来自 Controller 的 `we_reg` 信号
- `x0` 硬连线为 0：读时直接返回 0，写时 `write_addr != 0` 条件阻止写入
- 读为组合逻辑，支持同一周期内写入后再读取（写优先）

4.5 Cmp.sv —— 分支比较器

比较对象是 `read_data_1`（rs1）和 `read_data_2`（rs2）。LT/GE 使用 `$signed()` 做有符号比较，LTU/GEU 直接做无符号比较。

4.6 DataPkg.sv —— 写内存数据打包

将寄存器中的值按 mem_op 类型和地址偏移打包为写内存格式：

```
always_comb begin
    offset = dmem_waddr[2:0];
    shift  = offset * 8;
    case (mem_op)
        MEM_B, MEM_UB: dmem_wdata = {56'b0, reg_data[7:0]}    << shift;
        MEM_H, MEM_UH: dmem_wdata = {48'b0, reg_data[15:0]}   << shift;
        MEM_W:         dmem_wdata = {32'b0, reg_data[31:0]}   << shift;
        MEM_D:         dmem_wdata = reg_data                  << shift;
        default:       dmem_wdata = '0;
    endcase
end
```

例如 `sb x1, 3(x2)`（地址 = x2+3，偏移量 = 3）：`dmem_wdata = x1[7:0] << 24`，将字节放到 64 位数据的第 3 个字节位置。

4.7 MaskGen.sv —— 写掩码生成

```
always_comb begin
    offset = dmem_waddr[2:0];
    case (mem_op)
        MEM_B, MEM_UB: dmem_wmask = 8'b00000001 << offset;  // 1 字节
        MEM_H, MEM_UH: dmem_wmask = 8'b00000011 << offset;  // 2 字节
        MEM_W:         dmem_wmask = 8'b00001111 << offset;  // 4 字节
        MEM_D:         dmem_wmask = 8'b11111111;            // 8 字节（全写）
        default:       dmem_wmask = 8'b00000000;
    endcase
end
```

掩码的每一位对应一个字节。例如 `sw` 在地址偏移 4 时：`dmem_wmask = 8'b11110000`，表示只写高 4 个字节。

4.8 DataTrunc.sv —— 读内存数据截断与扩展

从 64 位内存数据中按地址偏移截取，并做符号扩展或零扩展：

```
always_comb begin
    offset = dmem_raddr[2:0];
    case (mem_op)
        MEM_B:  read_data = {{56{dmem_rdata[8*offset+7]}}, dmem_rdata[8*offset +: 8]};
        ...
        default: read_data = '0;
    endcase
end
```

设计要点：

- `MEM_B/MEM_H/MEM_W`（有符号）：用符号位做扩展（算术扩展）
- `MEM_UB/MEM_UH/MEM_UW`（无符号）：高位补零（零扩展）
- `MEM_D`：64 位直接使用
- `[8*offset +: 8]`：从第 `offset` 个字节开始取 8 位，相当于 `[8*offset+7 : 8*offset]`（Verilator 会报错）

> **特别注意：** `MEM_UW`（Load Word Unsigned, LWU 指令）很容易被遗漏。Controller 正确设置了 `mem_op = MEM_UW`，但 DataTrunc 的 case 语句必须同步包含该分支。

---

5. 仿真测试

5.1 测试体系

课程提供了差分测试框架，在每条指令执行后将 CPU 的状态（PC、指令、寄存器写回值、内存访问）与 Spike 参考模型进行比对。

测试用例位于 `repo/sys-project/testcode/testcase/`，覆盖 RV64I 全部指令：

测试用例	覆盖指令	说明
sample	addi, lw, sw, bne	简单功能验证
rtype	add, sub, sll, slt, sltu, xor, srl, sra, or, and	R 型指令
itype	addi, slti, sltiu, xori, ori, andi, slli, srli, srai, ld	I 型指令
stype	sd	S 型指令
btype	beq, bne, blt, bge, bltu, bgeu	B 型指令
utype	lui, auipc	U 型指令
jtype	jal	J 型指令
remain	jalr, lb/lh/lw/lbu/lhu/lwu, sb/sh/sw, ADDIW/...W, ADDW/...W	剩余指令
full	全部 RV64I 指令	综合测试

Table 1

运行命令：

```
make verilate TESTCASE=<用例名>
```

5.2 仿真结果

所有 9 个测试用例全部通过。以 `full` 测试为例，Spike 输出的部分执行日志：

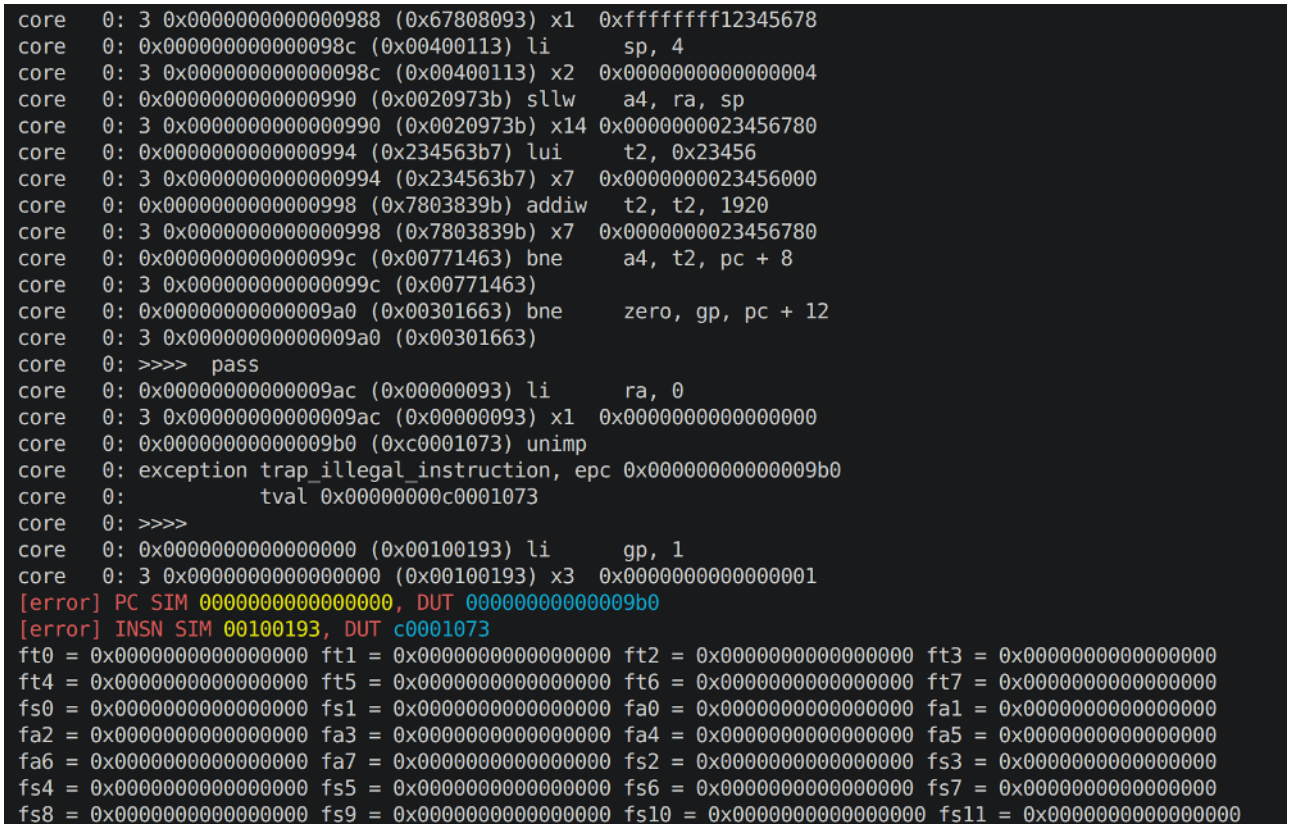

Image 3

测试末尾的 `error` 是**预期行为**：

- 测试程序末尾是一条非法指令 `0x00000000`
- 本设计的单周期 CPU 不支持异常处理，直接执行该指令

判定标准： 最终跳转 `pass` 分支。

---

6. 调试过程与问题总结

在实现过程中遇到的典型问题：

6.1 `next_pc` 逻辑反转

错误代码：

```
1  assign next_pc = (npc_sel && !br_taken) ? (pc + 4) : alu_res;
```

`!br_taken` 写反了。对于普通指令（`npc_sel=0`），条件 `(0 && !0) = 0` 走 ELSE 分支，`next_pc = alu_res`。第一条指令 `addi x1,x0,123` 的 PC 直接跳到了 `alu_res = 123`，后续取指全部错误。另外 JAL/JALR 的 `br_taken` 恒为 0（Controller 未设 `cmp_op`），也需要额外处理。

修复：

```
wire is_jal  = (opcode == JAL_OPCODE);
wire is_jalr = (opcode == JALR_OPCODE);
assign next_pc = (npc_sel && (br_taken || is_jal || is_jalr)) ? alu_res
: (pc + 4);
```

6.2 64 位指令存储器指令选取逻辑

`inst = imem_ift.r_reply_bits.rdata[31:0]` 永远取低 32 位，DRAM 是 64 位宽的（一个地址存两条指令），PC=4 时读到还是第一条指令。改为 `inst = pc[2] ? rdata[63:32] : rdata[31:0]`。

6.3 例化模块的输出未正确使用

- Store 指令写内存时用了 `read_data_2` 而非 DataPkg 打包后的 `dmem_wdata`。DataPkg 已按偏移量做了字节移位（如 `sb` 需将字节移到 64 位的正确位置），但最终写请求却绕过了它
- Load 指令写回时用了 `dmem_ift.r_reply_bits.rdata` 而非 DataTrunc 截断后的 `read_data`。DataTrunc 已按 `mem_op` 做了符号/零扩展，但写回却直接用了原始的 64 位内存数据

6.4 语法错误

6.4.1 切片位宽不能用表达式

初版 DataTrunc 中写 `dmem_rdata[8*offset+7 : 8*offset]` 时 Verilator 报错：切片的**上下界不能同时用表达式**，因为综合工具无法在编译时确定位宽。SystemVerilog 要求 `[base +: width]` 或 `[base -: width]` 写法，其中 `base` 可以是表达式但 `width` 必须是编译期常量：

```
// 错误：上下界都是表达式
read_data = { {56{dmem_rdata[8*offset+7]}}, dmem_rdata[8*offset+7 :
8*offset] };

// 正确：使用 +: 固定位宽语法
read_data = { {56{dmem_rdata[8*offset+7]}}, dmem_rdata[8*offset +: 8] };
```

6.5 环境：libcosim.a 补丁未生效

环境是 arm 架构，并且采用了 docker，疑似构建脚本依赖 sys 仓库的一个文件，需要在 docker 里重新安装，这里云里雾里地构建成功了，没法回忆。

---

7. 上板验证

7.1 生成比特流

修改 `include/initial_mem.vh` 指向测试 hex 文件的**绝对路径**，然后执行：

```
make board_sim TESTCASE=full    # 生成下板用 testcase.hex
make bitstream                  # 综合 + 实现 + 生成 bit 流
```

7.2 硬件调试

开发板上的调试接口：

- `switch[3:0]`：选择要查看的调试信息（PC/inst/rs1/rs2/alu/mem_addr/...）
- `switch[4]`：切换 64 位数据的高/低位
- `switch[15]`：调试模式，拨上后按中央按钮单步执行
- `switch[14]`：时钟频率选择，可以选择较慢的频率看到执行过程
- `button[0]`：中央按钮，单步执行一条指令

7.3 上板结果

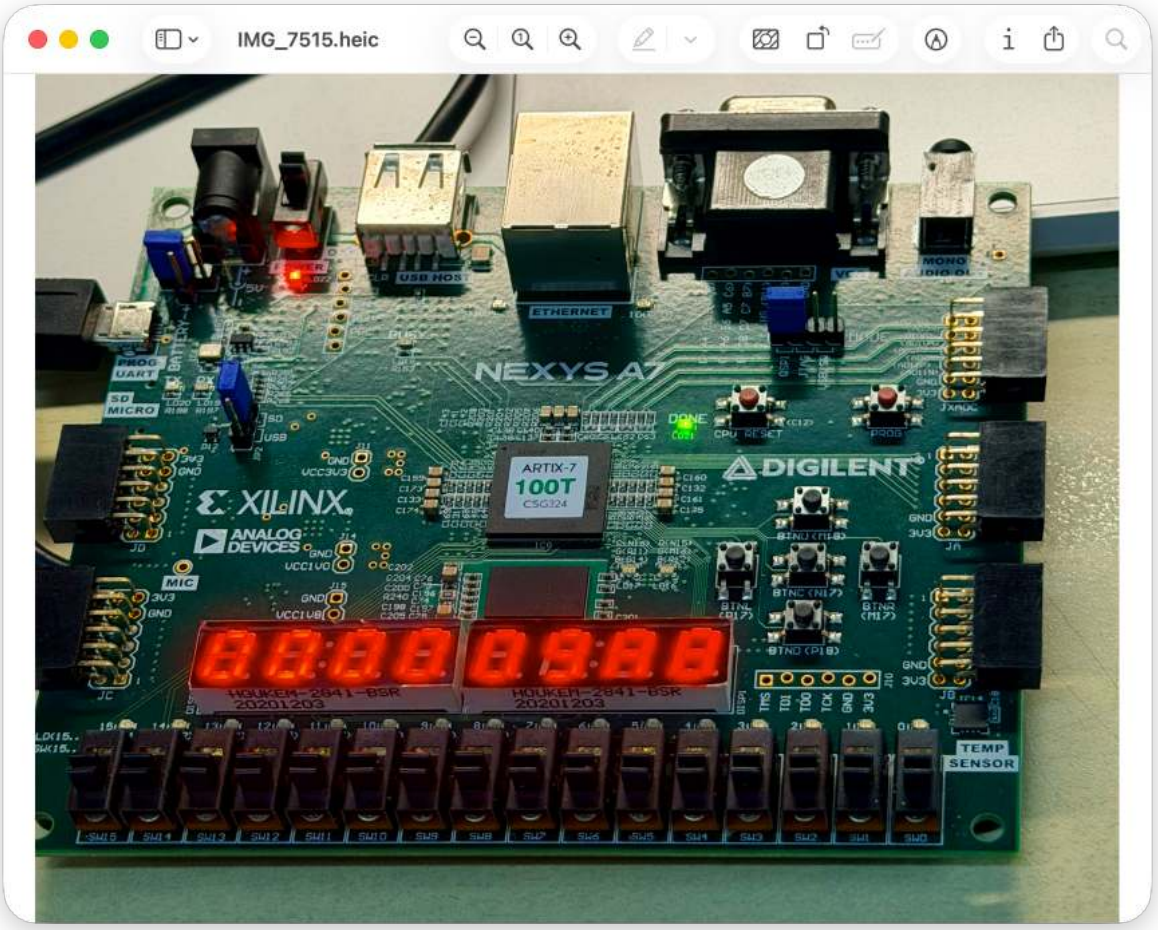

Image 4

上板测试通过 `full` 测试程序，程序执行正确时数码管最后显示地址 `0x9a8`（pass 死循环），执行错误时显示 `0x9a4`（fail 死循环，在reset之后会停留在这里）。

---

8. 总结

本实验完成了一个支持 RV64I 全部指令的单周期 CPU 设计：

- 覆盖 IF/ID/EX/MEM/WB 五阶段
- Controller 采用二段译码方法，降低编码复杂度
- 通过差分测试框架和 9 个测试用例逐级验证
- 理解了 RISC-V 指令编码格式和各指令的数据流

心得体会和建议

可以为系统 I 的实验安排、实验内容、实验指导留下任何宝贵的心得体会和建议吗？（不记录分数，纯属用于吐槽，感谢大家为我们的课改贡献一份力量）

sys1的实验部分告一段落，总体而言我认为这是非常**先进**的一门课。

从工作量上，我们的难度相较于前几轮应该已经降低了。加上sys1没有特别令人头疼的架构和时序等等问题，所以总体上感觉尚可。实验比较依赖自学和探索，这和这个专业的学习方法匹配。课程内容很有时效性和实用性，从真实存在且实用的架构入手。我相信连续三个学期的sys绝对算是zju的特色，这需要大量的精力来维护文档、确保课程连贯、确保环境干净整洁…… 我猜不是所有学校都可以推出这种高质量课程。受益良多。

btw，我在整个实验过程中只用了M芯片Macbook Air，相对顺利，其中有一些独特的配置环境的方法（比如如何借助mac自带的rosetta将vivado装在arm Ubuntu上从而可以流畅地使用batch下板，且不需要配置硬件端口等等），如果以后有机会我会整理出文档。

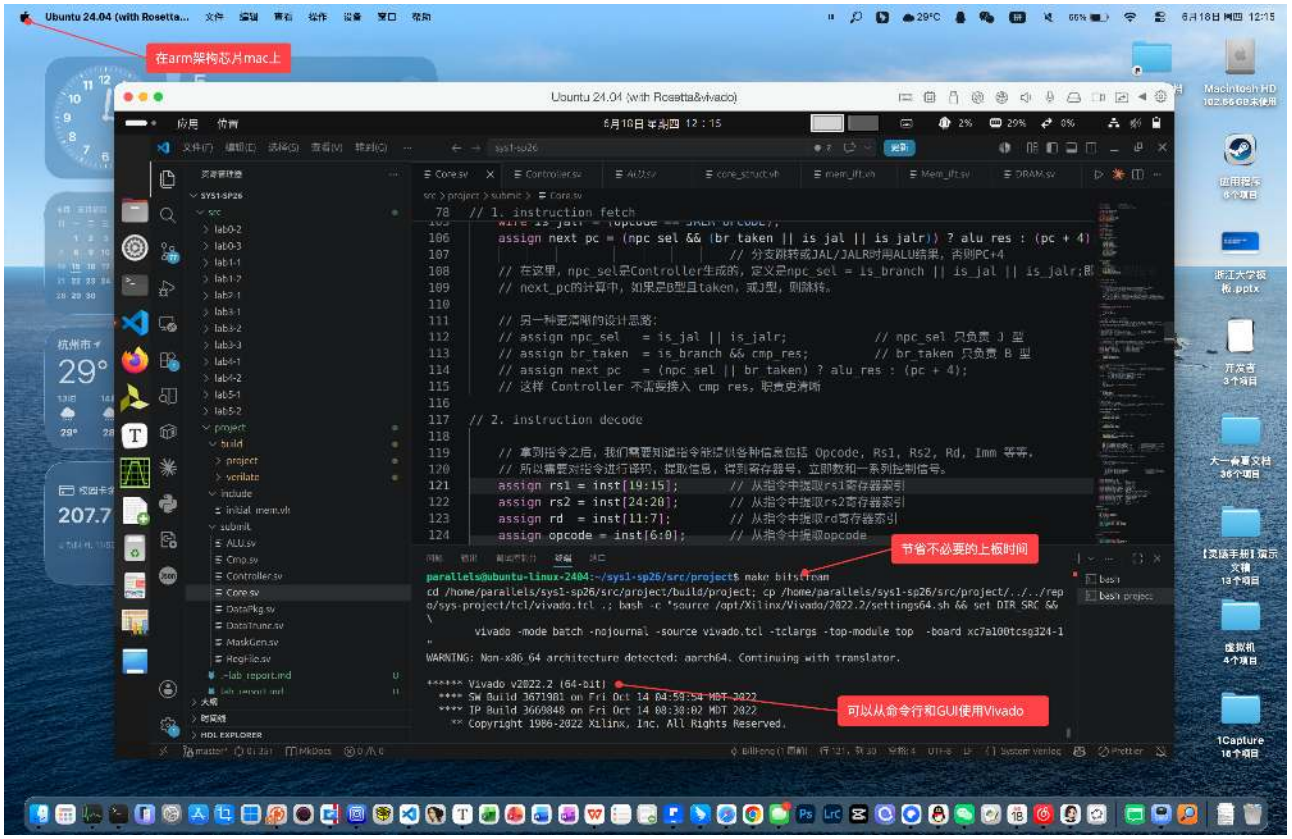

Image 5

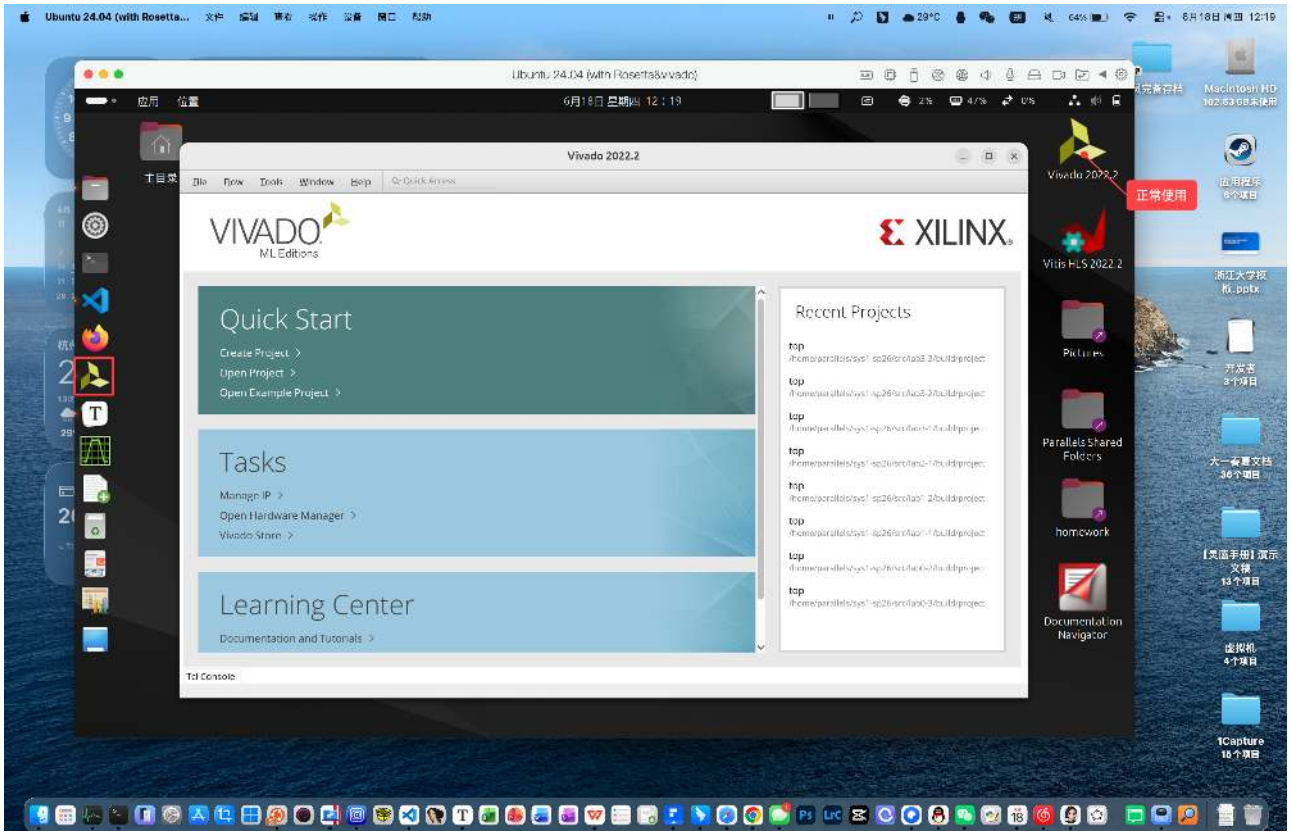

Image 6

以下是我期中总结中写的:

在这门课我体会到了完善的实验文档带来的便捷，听到了优秀的老师和TAgg们的课程和指导。或许信安融洽的氛围正是老带新的传承带来的。

高二时曾玩过一款名为《图灵完备》的游戏，内容是计算机系统I和II的简化版(现在疑似把系统III的内容也更新了)。没想到这成为了我的专业课。或许这就是缘分吧。

用硬件语言写电路是一种不错的体验，尽管没有图形操作来的直观，但胜在简洁、方便移植和维护。