计算机组成原理实验

Lab 4 实验手册

单周期 CPU 设计

Made by TA

2023 年 4 月 21 日

目录

在阅读实验手册之前，你需要了解的内容包括：

1. **本次实验包括实验 PPT、实验手册、演示视频以及附件文件（包括 PDU）等内容。**其中，实验手册（也就是你正在看的这个文档）是实验 PPT 讲解的额外补充，用于明确实验细节。
2. 实验手册的每一部分内容都有着对应的作用。当你遇到困难无法继续时，请确保你已经认真查阅了实验手册中的全部内容！如果你依然对实验内容有所疑问，欢迎你在群聊或私聊中提出你的问题，我们会在许可的范围内进行解答。
3. 请保证实验内容为自己独立完成。我们将对重复率过高的实验结果进行严肃处理。
4. 为了保证区分度，实验的部分内容难度较大。请量力而行，不要在超出自身能力范围外的部分投入过多的精力。

祝大家实验顺利！

本次实验已开通 FAQ 文档！

FAQ 是 Frequently Asked Questions 的缩写，中文释义为常见问题解答，或者是帮助中心。你也可以将其理解为一份针对大家提出问题的统一解答。本次实验的公开 FAQ 文档地址为 https://cscourse.ustc.edu.cn/vdir/Gitlab/PB20020586/lab-of-cod-faq/-/blob/master/Lab4FAQ/lab4.md。当你遇到疑惑的地方时，可以先看看这里，如果还是没有解决你的问题，可以在群聊或私聊中提问。我们会根据大家的问题不定期更新 FAQ 文档，也请大家随时保持关注。

本次实验已开通 PDU bug 反馈渠道！

PDU 为助教针对本学期课程需求重新编写的板上调试工具，由于时间紧张，难免会出现 bug。**PDU 源文件内容请参考文档末尾的附件链接。**为了保障大家实验的顺利进行，本次实验为大家开通了 bug 反馈通道。如果你在实验中遇到了难以解决的问题，或影响正常使用的恶性 bug，可以在下面的链接中反馈 https://www.wenjuan.com/s/UZBZJv96IMN/。我们会及时据此更新 PDU 的相关内容，并在群聊以及 FAQ 中及时告知大家。感谢大家对我们的理解与支持！

1. 前言

在经过了三个实验的摸索之后，我们终于可以开始搭建自己的第一个 CPU 了。当然，这只是一个十分基础的单周期 CPU，但这将是“图灵完备” 在 COD 课程中的直接体现。

单周期 CPU（Single Cycle Processor）是指一条指令在一个时钟周期内完成，并在下一个时钟周期开始下一条指令的执行的 CPU。单周期 CPU 由时钟的上升沿或下降沿（请思考：为什么可能会用到下降沿呢）控制相关操作，两个相邻的上升沿或下降沿之间的时间间隔就是 CPU 的时钟周期。

需要注意的是，由于没有额外的暂存寄存器，单周期通路中的关键路径对应的延迟很高（参考书上的习题 4.7），以致于上板时的单条指令运行时长大于 10 纳秒。所以，我们不能直接使用开发板上的 100MHz 时钟作为 CPU 的运行时钟。除此之外，上板时我们也无法了解当前 CPU 运行到哪条指令，以及相应的结果如何。为此，我们不得不请出 PDU 来帮助我们完成这些工作。

Lab4 里，我们将深刻领悟到 CPU、PDU 以及 MEM 之间是如何相互合作的。你将根据我们提供的框架，实现属于自己的单周期 CPU。那么，祝大家实验愉快!

2. 主要内容

本文档主要介绍的内容如下：

1. 对单周期 CPU 所需要的基本组件的介绍，包括寄存器、ALU、控制器等；
2. 对单周期 CPU 数据通路的介绍。
3. 对于外设与调试单元 PDU 的详细介绍（推荐仔细阅读）

你需要完成的内容概括如下：

> 详细的实验内容见文档结尾，此处给出大致内容以方便同学们带着目的去阅读文档

- 根据我们提供的数据通路完成单周期 CPU 的硬件设计，并完成仿真。
- 将 PDU 接入到 CPU 上，完成上板测试。

- 运行给定的汇编程序，并检查运行结果。

本文档的描述顺序提供了一种 CPU 的设计思路及顺序(先完成各个模块，最后将它们连接起来)，你可以按照本文档的描述顺序进行设计，或者你也可以按照自己的思路进行设计。但是，你需要保证 CPU 模块的接口与我们给出的一致。

助教为本次实验提供了功能相当强大的 PDU 供大家使用，为了使大家设计的 CPU 能够更好的与提供的 PDU 通信，请务必仔细阅读助教提供的完整的数据通路。与往届实验和课本稍有不同，我们将存储器(指令和数据存储器)在模块层次设计中提高了一层，使其与 CPU 处于同一层级(这相当于将内存作为 CPU 外部设备而非 CPU 的一部分)，来支持 MMIO(Memory-mapped I/O)，即内存映射 I/O。

3. 单周期 CPU 的各个模块

CPU 是高度模块化的，它的各个部件功能明确，边界清晰，是模块化设计的典范。下面，我们将首先介绍单周期 CPU 的各个模块，然后介绍单周期 CPU 的数据通路。

3.1 PC 寄存器

寄存器是我们最先接触过的时序元件。一般而言，我们可以用如下的 Verilog 程序描述寄存器的行为：

```
1    always @(posedge clk) begin
2        if (reset)
3            q <= 0;
4        else if (en)
5            q <= d;      // q stores the value inside the register
6    end
7
```

上面的代码给出了带有同步复位、使能信号的寄存器单元。

在 CPU 中，PC 寄存器时刻存储了正在执行的指令的地址。它的功能是将当前指令的地址传递给指令存储器。同时，它也需要能够接受下一条指令地址的输入(+4 还是跳转)，并在时钟上升沿到来时更新自己的值，从而实现了指令的连续运行。

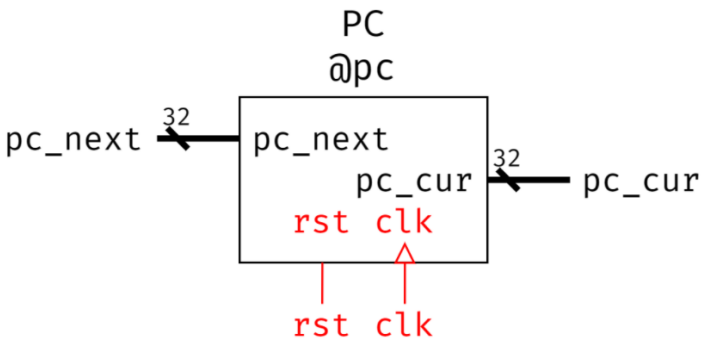


图 1: PC 寄存器示意图

考虑到单周期 CPU 每个时钟周期都会更新 PC 寄存器中的值，我们便不再需要写使能 en 信号，将其置一即可。图 1 中直接省略了 en 端口，你也可以在代码中将其移除。

你可能注意到，PC 寄存器的行为可以实现为一个单独的寄存器，但模块化设计下我们仍然建议你将 PC 寄存器设计为**一个单独的模块**。为什么要有如此“多此一举” 的设计呢？一方面，这样可以使得你的设计更加清晰，减少杂乱的门电路的出现（看不见就是没有.jpg）。另一方面，你可以更为直观地看到各个模块之间数据的传输关系，便于和我们提供的单周期数据通路进行比对。此外，为了代码的可读性，我们一般不会在一个 module 中保留过多的代码。因此，对于部分程序的封装是很有必要的。

需要补充的是，我们建议你将 CPU 的复位信号改为异步复位，并将复位值设定为 0x2ffc。这样可以保证在上板时，程序即将执行的下一条指令为 0x3000，从而无缝衔接我们填写在指令存储器中的指令内容。

3.2 寄存器堆 RF

我们已经在 Lab2 中基本完成了寄存器堆的设计。它有 1 个写端口，2 个读端口。本次实验中，寄存器堆的大小需要设置为 32×32 bits，也就是包含 32 个大小为 32bits 寄存器的阵列。

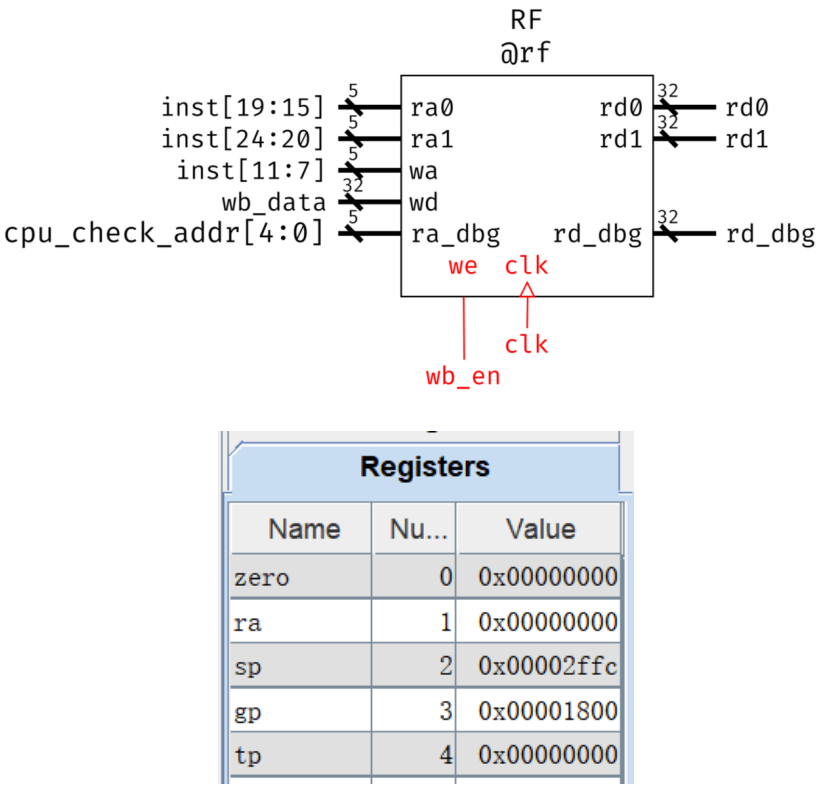
> 这样的设计是因为 RV32I 指令集中的指令最多只有两个操作数，最多只需要写入一个寄存器，例如：$addt_0, t_1, t_2$，这条指令就需要读取 t_1 和 t_2 寄存器的值，经 ALU 运算后将结果写入 t_0 寄存器中。因此，两读一写的寄存器堆就足以支持 RV32I 指令集的全部需求了。

为了与 Rars 保持一致，你需要将 Regfile[2] 初始化为 0x2ffc，Regfile[3] 初始化为 0x1800。其他寄存器均初始化为 0。你可以使用下面的语句对寄存器堆进行初始化：

```
integer i;
initial begin
    i = 0;
    while (i < 32) begin
        Regfile[i] = 32'b0;
        i = i + 1;
    end
    Regfile[2] = 32'h2ffc;
    Regfile[3] = 32'h1800;
end

```

这里的 initial 语句可以夹在寄存器堆模块的设计文件中，在仿真和上板时都可以生效。



Registers

Name	Nu...	Value
zero	0	0x00000000
ra	1	0x00000000
sp	2	0x00002ffc
gp	3	0x00001800
tp	4	0x00000000

图 2: 寄存器堆设计思路示意图

为了方便 CPU 与 PDU 的通信，我们需要在寄存器堆**额外增加一个读端口**(ra_dbg、rd_dbg)，用于 PDU 从外部读取寄存器的值。如图 2 所示，**这个端口不应该被汇编指令**

使用，也就是不能接入 CPU 的数据通路之中。

3.3 算术逻辑单元 ALU

ALU 的功能是对两个操作数进行算术或逻辑运算，然后将结果输出。在 Lab1 中，我们已经实现了对其的设计，因此直接将其复用即可。本次实验中，你需要将例化参数 WIDTH 设置为 32，代表我们使用的是 32bits 位宽的 ALU。

> ALU 的两个操作数可能是寄存器堆的输出，也可能是立即数，还可能是 IMM ，它的输出可能用于寄存器堆的写入，还可能用于数据存储器的地址输入等。为此，通路中需要有众多的选择器，用于控制 ALU 的数据来源与去向。我们将在数据通路中对其详细介绍。

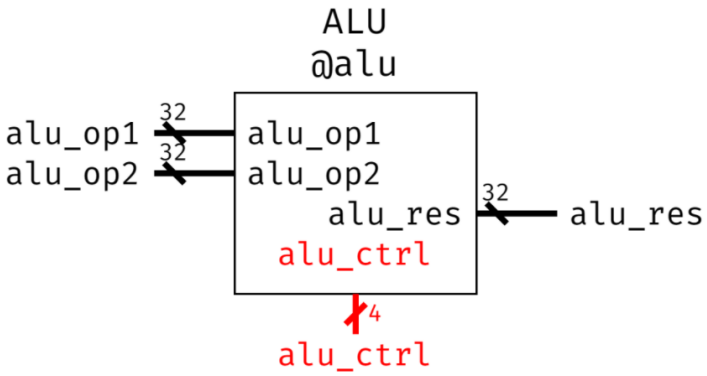


图 3: ALU 示意图

事实上，你几乎可以直接使用 Lab1 中的 ALU，不过溢出位的输出我们暂时不需要，所以可以直接将其悬空（例化时不连接该输出端口即可）。

此外，Lab1 中的 ALU 还有许多的运算功能，你可以在本次实验的选做部分中用到它们。

3.4 IMM 生成器

IMM 生成器的功能是将指令中的立即数提取出来，然后扩展为 32bits（思考：是有符号还是无符号呢？）。IMM 模块的设计是相当简单的：根据来自 Control 模块的 imm_type 信号

决定立即数的格式，再从输入的指令中按照对应格式生成所需要的立即数。当然，imm_type 信号不是必须的，因为该信息在输入的 Instruction 中已经被完全包含了。

> 一些指令并不需要立即数，怎么办？我们可以输出任意的默认值，后面的选择器会将正确的输入交给 ALU！一种推荐的思路是，为立即数模块的输出默认赋值 0。

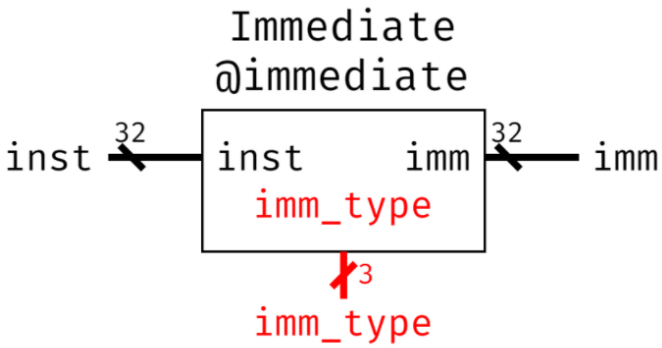


图 4: IMM 示意图

你可能会有疑问：既然 RISC-V 中的立即数有很多种，为什么我们只有一个输出端口呢？这是因为无论是什么格式的立即数，我们都可以通过输入指令以及控制信号来唯一确定其内容。为了简化数据通路，减少不必要的开销，我们将所有种类的立即数从一个端口中输出。为此，你需要仔细考虑应当输出的立即数格式。

4. 内存模块

与课本以及往年实验不同，本次实验中，我们将内存从 CPU 中分离出来作为统一的模块，它将依据 CPU 与 PDU 的控制信号进行读写操作。内存分为指令存储器和数据存储器两部分。总体可以表示如下：

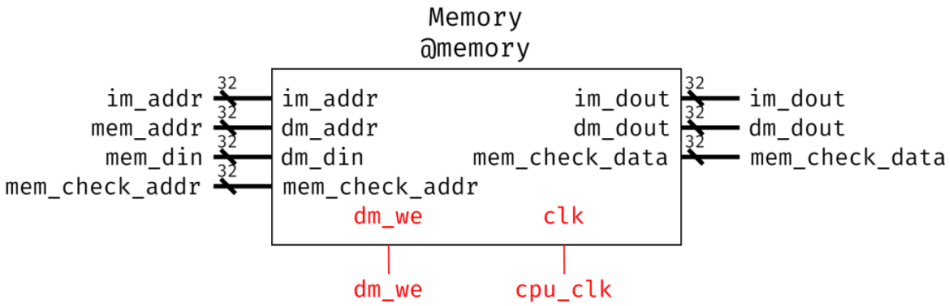


图 5: 内存模块示意图

> 未标注的情况下，含有 check 的端口为调试端口，一般与 PDU 相连接，其余端口均与 CPU 相连接。

在单周期 CPU 中，我们将指令与数据存放在两个不同的存储器里。这是由于我们必须在一个时钟周期内同时读写两个存储器。如果仅使用一个存储器，则无法同时进行指令读取与数据的写入。

4.1 指令存储器 IM

指令存储器根据 PC 寄存器的输出地址，将指令从存储器中读出，然后将指令交付给 CPU。由于汇编程序不会修改指令存储器的值（因为这是不合法的），我们直接例化一个分布式单端口 ROM 作为指令存储器即可。

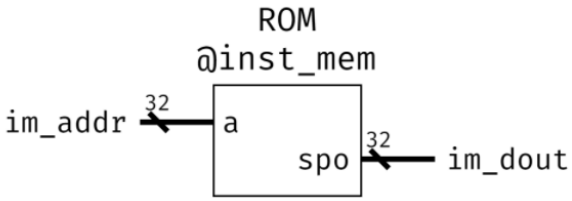


图 6: 指令存储器 Inst_mem

4.2 数据存储器 DM

数据存储器根据 CPU 传来的读写地址，将数据从存储器中读出，或者将数据写入存储器。区分当前为读取还是写入的标识是 dm_we 信号，高电平代表写入数据，低电平代表读取数据。

为了让我们的 PDU 能够便捷访问数据存储器，我们选择具有双端口的 DRAM（一个为读写公用端口 a，另一个为只读端口 dpra），这样我们就可以用 PDU 读取数据，同时不影响 CPU 的正常工作。

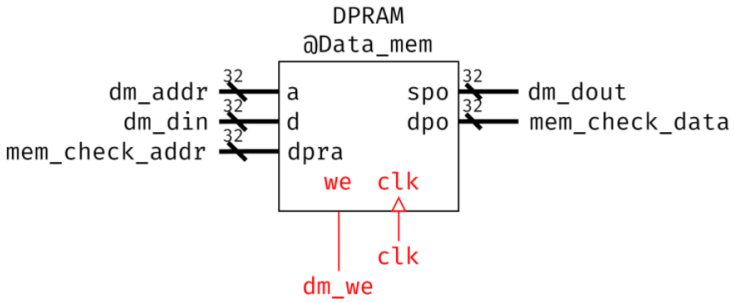


图 7: 数据存储器 Data_mem

总的来说，内存单元 MEM 的内部整体结构呈现如下：

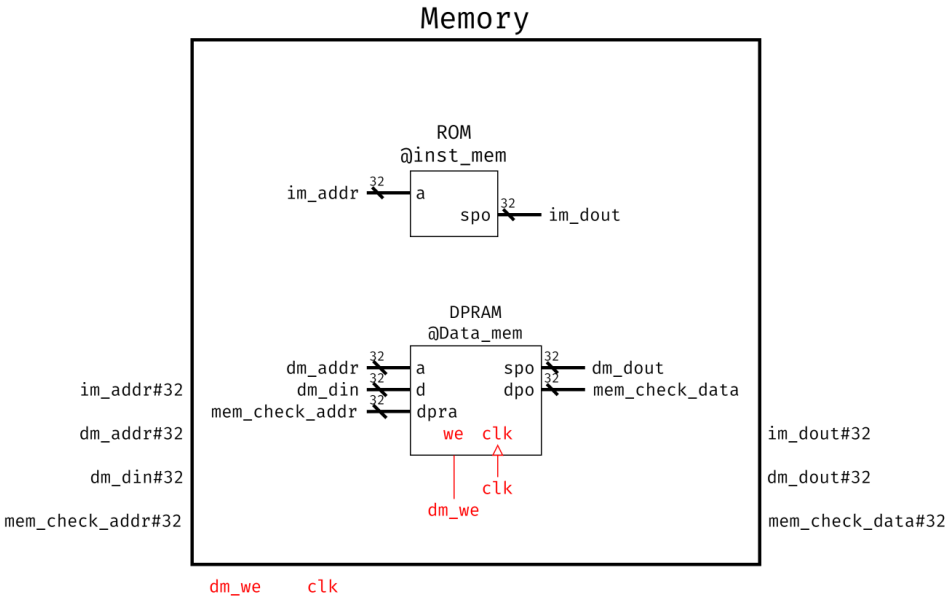


图 8: 包含 IM 和 DM 的内存模块示意图

需要强调的一点是，我们传入的 mem_addr 是 32bits 宽度、字节对齐的地址，而存储器的存储单元宽度即为 32bits。为此，实际传入存储器的地址应当为 mem_addr[9:2]。

> 为什么是 [9:2] 呢？这是因为教材上约定的地址单元宽度为 8bits，CPU 需要从某一地址开始，连续读出四个单元的内容才能得到完整的 32bits 数据（因此每次递增时是 PC + 4 而不是 PC + 1）。但是在本次实验中，为了简化设计，我们将地址单元的宽度设置为 32bits，此时读取一个地址单元即可得到完整的数据。所以 MEM 需要对传入的地址除以 4，也就是右移两位之后再接入存储器。

除此以外，我们在使用 PDU 调试时，使用的 mem_check_addr 对应的单元大小是 32bits，因此可以直接与 DataMemory 相连，而无需进行移位处理。你可以参考 PDU 指令手册中关于 ck2 指令的相关介绍来进一步理解这个问题。

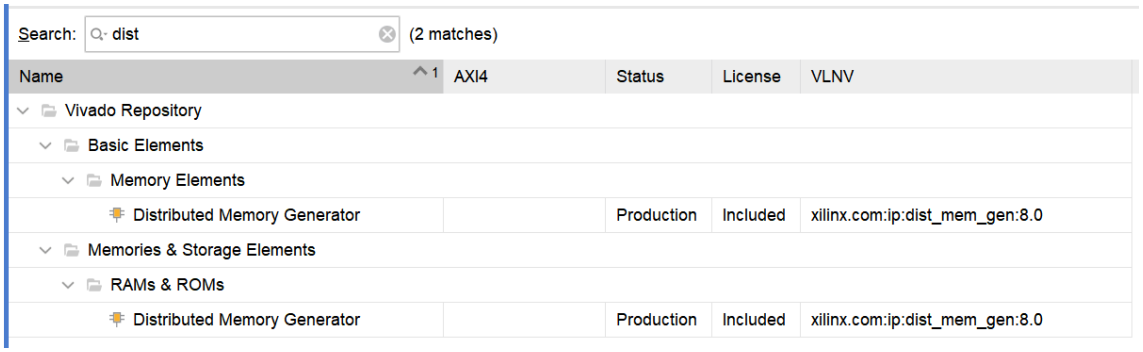
4.3 生成 IP 核并例化

在 Lab2 中，我们已经初步了解了分布式存储器与块式存储器在读写时序上的差异。在单周期 CPU 中，由于所有的操作都需要在一个 CPU 时钟周期内完成，我们选择分布式存储器作为本次实验的指令存储器与数据存储器。

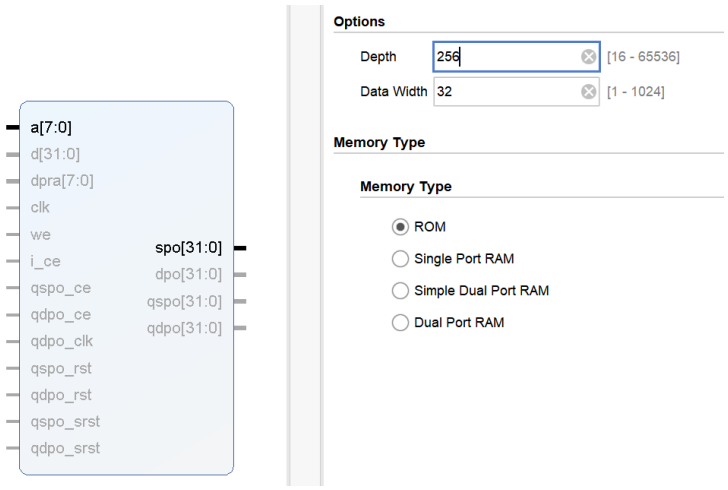
与寄存器堆一样，我们的存储器每个存储单元大小设置为 32bits。那地址单元的数目呢？我们的指令段范围为 0x3000∼0x3ffc，对应着 0x400 （1024）个 32bits 存储单元，数据段同理可得。然而，1024 个存储单元实在是太奢侈了，目前我们暂时用不到如此多的存储单元。因此本实验中，我们的 IP 核大小均选择 256×32bits。

正如前文所述，指令存储器是一个分布式 ROM，数据存储器是分布式 RAM。在配置 IP 核选项时，请确保**勾选的是 Non-Registered 选项**，否则存储器的读写时序会出现问题。

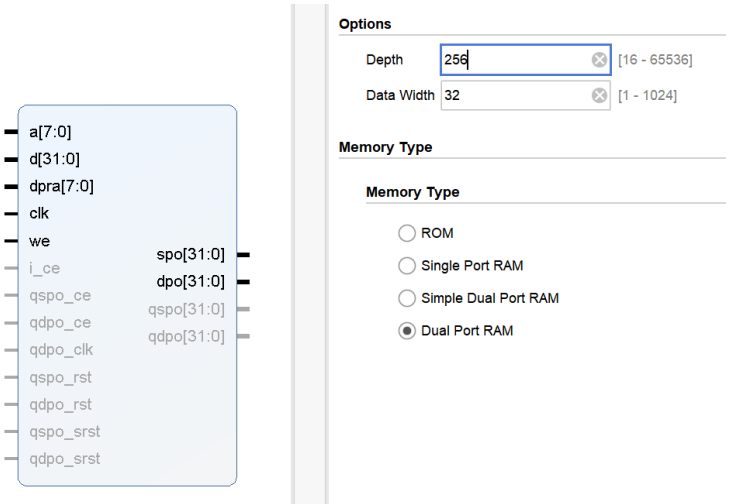
下面的图 9 展示了使用 Vivado 生成 IP 核时的相关信息。生成完成后，你可以在 Project_name\Project_name.srcs\sources_1\ip\IP_name\IP_name.veo 文件中找到该 IP 核的例化端口信息。你需要据此在 MEM 模块中完成对 IP 核的例化。



(a) 分布式存储器的位置



(b) 指令存储器 IP 核设置



(c) 数据存储器 IP 核设置

图 9: 分布式存储器例化示意图

5. 单周期 CPU 的数据通路

介绍完 CPU 内部基本的功能模块后，我们回到 CPU 模块本身。如何将我们设计的功能模块连接起来，并保证汇编指令的正确执行呢？试想，如果把指令看作从同一起点出发，但终点不同的车辆，那么任何车辆都会有不同的行驶路径（可能会有公共部分）。现在，我们可以通过控制各个路径上的信号灯，引导车辆沿着正确的路径前往目标点。

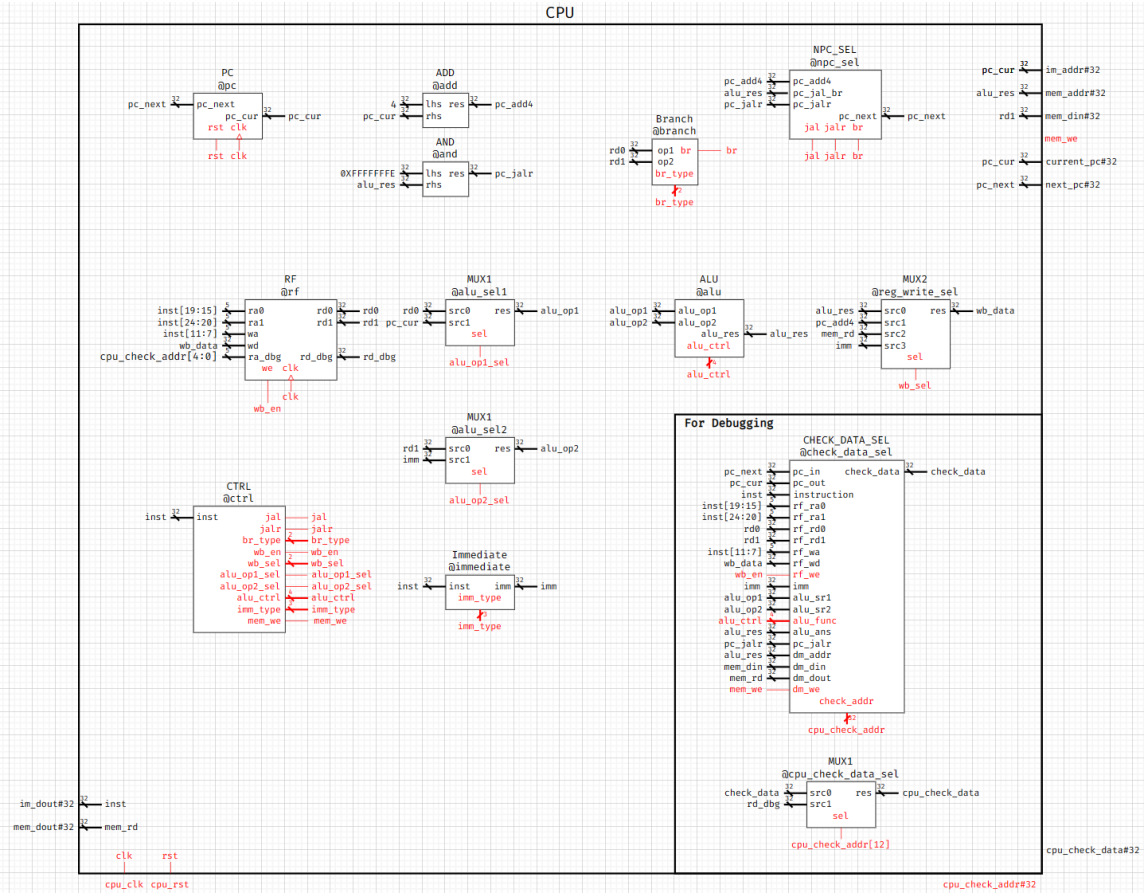


图 10: CPU 模块通路示意图（附件中有原图）

图 10 是我们为大家绘制的单周期 CPU 数据通路简图。其中，所有的模块之间并没有显式用线连接，而是标注了相应的线名。这与大家使用 Verilog 例化模块时的思路是一致的，只需要确定相应接口的连线即可，而不在意这些线路具体是如何连接的。

让我们把之前的例子对应到单周期 CPU 上，车辆的路径对应指令经过的不同线路，信号灯对应着通路中的控制信号。在正确的控制信号的引导下，我们就可以确保指令的正确执行（硬件比起人类还是可靠的）。

那么，如何根据指令产生控制信号呢？这就是 Control 模块与 Branch 模块的工作了。

5.1 控制器 Control

控制器是流水线 CPU 的“心脏”，负责根据输入的指令生成相应的控制信号。下面是本次实验中控制单元的结构示意图。

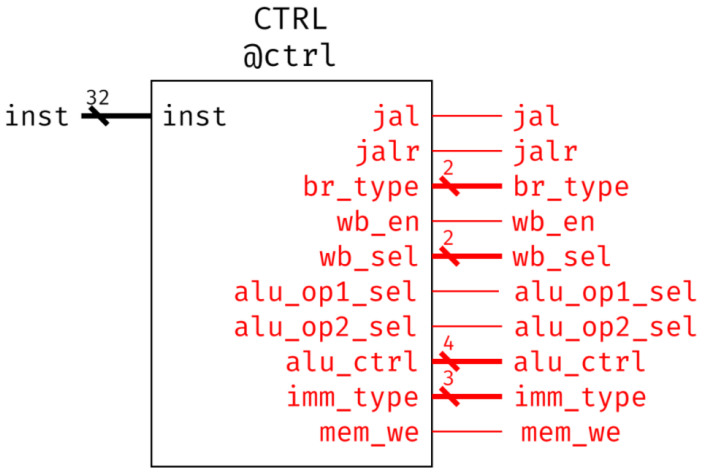


图 11: Control 控制器示意图

类似于有限状态机的第三部分，控制器单元是一个组合逻辑模块，根据当前输入的指令产生不同的控制信号。单周期 CPU 的控制信号包括：

- NextPC 选择信号，来源于 jal、jalr 以及分支指令。Branch 模块也参与了这部分信号的生成。该信号决定了下一条执行的指令的地址；
- 存储器控制信号，在我们的实验中为写使能信号；
- ALU 源操作数选择信号，即 ALU 的两个输入数据的来源；
- ALU 模式信号，用于控制当前进行的运算种类；
- 立即数类别信号，该信号用于立即数模块的类型判断，从而选择产生相应的立即数；
- 寄存器堆写入数据选择信号，该信号用于控制即将写入寄存器堆的数据的来源。

我们建议你在设计 Control 模块时，针对每一条指令，认真分析其在数据通路中经过的路径，从而确定相关信号的具体内容。当然，不要忘记为每一个控制信号添加默认赋值，否则会在电路中产生锁存器。

5.2 分支模块 Branch

与教材不同，我们把条件跳转指令的逻辑判断从 ALU 中移出，封装成了独立的模块。Branch 模块是专门用来处理分支指令的模块。它接收来自 Control 的控制信号，以及来自寄存器堆的两个数据。根据这两个数据之间的大小关系以及控制信号， Branch 模块最终产生相应的跳转信号。下面是分支模块的结构示意图。

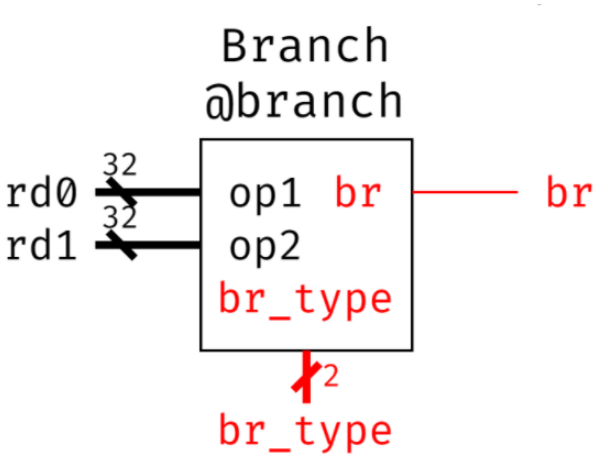


图 12: Branch 模块示意图

实验中必做部分的分支指令包括：beq、blt，选做部分的分支指令包括 bne、bge、bltu、bgeu。你可以参考 Lab1 ALU 模块中的部分内容实现 Branch 模块的相应逻辑。

5.3 选择器模块

与控制单元不同，选择器并不扮演着信号灯的角色，而是作为车辆合流的引导者。在电路中，我们不能允许一根数据线有多个输出源，毕竟高电平 + 低电平 $\neq$ 0.5 电平。为此，我们需要让多个来源经过统一的合流器，也就是选择器，最终汇合到同一根线上。

Mux 作为一个选择器，自然可以用 always@(*) 搭配 case 语句实现。然而，我们建议你将该语句封装在一个独立的模块之中，并在 CPU 的数据通路里例化。这与将 PC 寄存器封装成一个模块的思想是一致的。

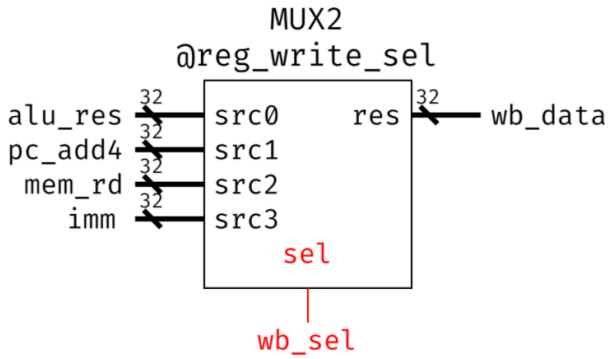


图 13: 寄存器堆写回时 Mux 选择器模块示意图

图 13 是寄存器堆写回数据的选择器。注意观察该选择器的四个数据来源： ALU 运算结果、PC + 4 的结果、内存读取出的数据结果，以及立即数单元的输出结果。这些数据来源分别对应了什么指令呢？提示：我们的通路中，lui 指令应当经过哪些部分？

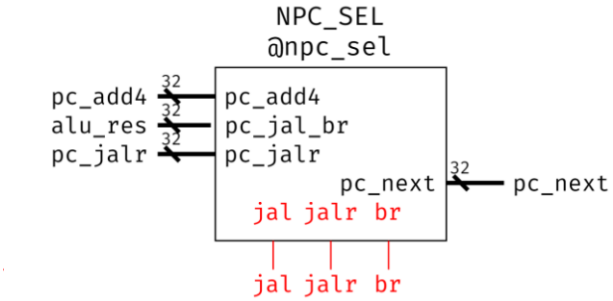


图 14: PC 寄存器写入时 Mux 选择器模块示意图

一个关键的选择器是图 14 所展示的 pc_next 选择器。该选择器用于控制即将写入 PC 寄存器的数据来源。上板时 PC 的初值为 0x2ffc，对应 IM 的第 0xff，也就是第 255 个存储单元。但我们的程序往往不会存入该单元，因此这个时候读出的 Instruction 内容为 0。为了保证下一条指令能够正常跳转到 0x3000，你需要保证自己的 Control 模块在 Instruction 为 0 时能够给出正确的 NextPC 跳转结果。

在本次实验中，选择器会有如下的应用：ALU 源操作数的选择、Next PC 的选择，以及写入寄存器的数据选择。你可以在实验中例化一些选择器来实现这些连接。当然，我们建议你采用参数化的方式设计模块。

6. 顶层模块设计

接下来，让我们再上一层，从单周期 CPU 的数据通路来到顶层模块的数据通路。我们将以最上层的视角带大家分析本次实验的相关内容。

本次实验中，顶层模块我们已经为你设计好，包括各个模块之间的端口连接。你需要在 CPU 模块中实现自己的相关设计，并在 MEM 模块中例化自己的存储器。除此之外，其他部分的连接已经由我们为你写好。**请不要私自修改 PDU、CPU 和 MEM 的端口信息。**如果有相关变动，我们会统一向大家反馈。

6.1 概述

本次实验的顶层模块由 CPU、PDU 以及 MEM 三部分组成。图 15 是对顶层模块的整体呈现：

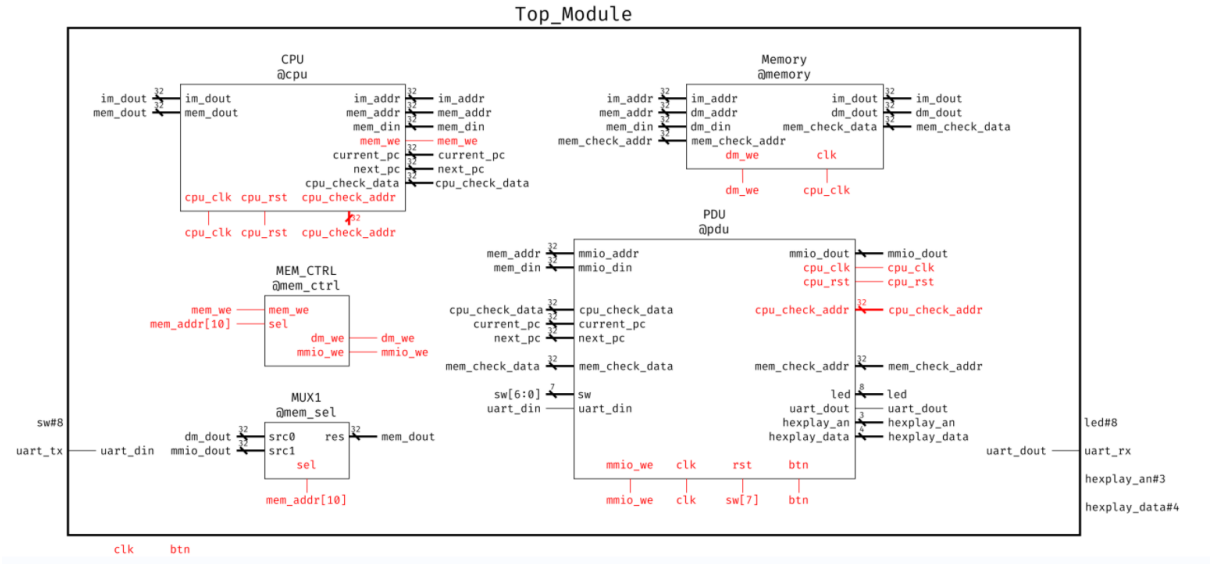


图 15: 顶层模块示意图（By 助教，原图见附件）

三个模块两两之间都有相应的数据交互，彼此合作实现了我们本次实验的全部功能。我们提供的框架代码中，Top、CPU、MEM、PDU 之间的连线也是与该图相对应的。

你也可以参考老师们画的顶层模块关系图。

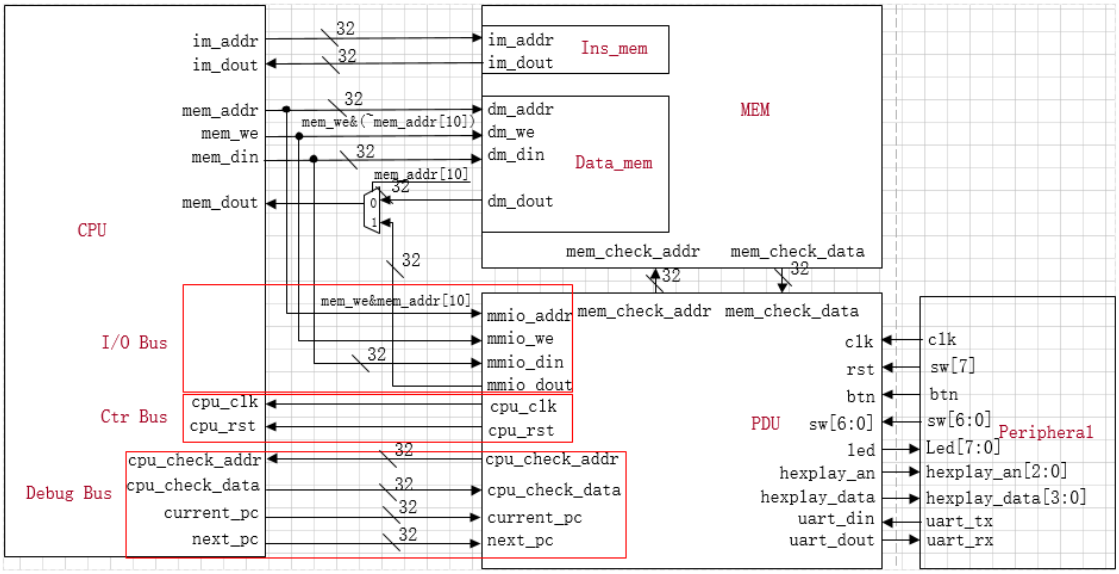


图 16: 顶层模块示意图（By 老师）

你可能已经发现，顶层模块中有一个选择器，用于选择 CPU mem_dout 数据的来源。这是因为数据存储器和 MMIO 分别位于 MEM 和 PDU 中，但 CPU 只能通过地址访问相应的存储单元，并不知道存储单元真实的位置。因此，我们需要根据 mem_addr 判断存储单元的位置，进而将正确的数据交付给 CPU。

此外，图 15 中也标出了 dm_we 和 mmio_we 的生成逻辑。事实上数据存储器和 MMIO 是不会被同时写入的，因为二者的地址范围没有交叉。为了保证这一约束，我们在 Top 模块中加入了下面的语句：

```
1    assign mmio_we = (mem_addr[15:8] == 8'h7f) ? mem_we : 0;
2    assign dm_we = (mem_addr[15:8] == 8'h7f) ? 0 : mem_we;
3
```

由此我们可以确保两个写使能信号不会在同一时间有效。

> 数据段理论上的地址范围是 0x0000~0x1000，但由于我们只有 256 个地址单元，所以实际上的地址范围是 0x0000~0x03fc；MMIO 的地址范围是 0x7f00~0x7ffc。二者彼此之间是无交的。那么，程序段的地址范围是多少呢？

6.2 端口说明

下面是对于各个模块的接口介绍。

MEM 内存模块

```
1   module MEM(
2       input clk,
3
4       // MEM Data BUS with CPU
5       // Instruction memory ports
6       input [31:0] im_addr,
7       output [31:0] im_dout,
8
9       // Data memory ports
10      input  [31:0] dm_addr,
11      input dm_we,
12      input  [31:0] dm_din,
13      output [31:0] dm_dout,
14
15      // MEM Debug BUS
16      input [31:0] mem_check_addr,
17      output [31:0] mem_check_data
18  );
19
```

内存模块实际上是对于指令存储器、数据存储器的封装。注意到课本上将这两部分放在了 CPU 内部，而我们选择将其与 CPU 并列。我们希望通过这种方式，让大家意识到 CPU 访问内存是需要很高的代价的（至少在距离上更远）。

MEM 模块接口由四部分组成：时钟（请思考：这个时钟是 CPU 时钟还是系统时钟呢）、指令存储器访存接口（im）、数据存储器访存接口（dm），以及内存 Debug 端口。在内存单元中，我们只需要查看数据存储器的内容，因此需要接入 Debug 信号的只有数据存储器。你可以将 Debug 端口与数据存储器的只读端口相连。

为了避免可能的误会，我们在此统一约定：**_addr 代表存储器的地址端口/线， _din 代表存储器的数据输入端口/线， _dout 代表存储器的数据输出端口/线。所有的输入、输出都是相对存储器而言的。**

4.2 小节中介绍了 MEM 模块中存储器例化时需要注意的内容。

CPU 中央处理器模块

```
module CPU(
    input clk,
    input rst,

    // MEM And MMIO Data BUS
    output [31:0] im_addr,
    input [31:0] im_dout,
    output [31:0] mem_addr,
    output  mem_we,
    output [31:0] mem_din,
    input [31:0] mem_dout,

    // Debug BUS with PDU
    output [31:0] current_pc,
    output [31:0] next_pc,
    input [31:0] cpu_check_addr,
    // Check current datapath state (code)
    output reg [31:0] cpu_check_data
    // Current datapath state data
);

```

在我们看来，CPU 是一个负责执行指令的“黑盒”。我们负责为其提供时钟与复位信号， CPU 可以自动从内存中获取指令与数据，并执行相应的运算，向内存中写入得到的结果。因此，我们希望可以查看 CPU 内部各个时刻程序的运行状态，这就是 CPU Debug 端口的作用。

CPU 的接口由三部分组成：时钟与复位信号、与 MEM 的交互端口、与 PDU 的交互端口。其中，指令存储器的输入地址即为当前的 PC 值，输出结果即为当前的指令。数据存储器的相关接口需要你根据我们提供的单周期 CPU 数据通路图确定如何连接。此外，为了保证 PDU 相关功能的使用，你需要根据我们在实验手册中列出的 Check_addr 和 Check_data 的对应关系，在 CPU 内部正确实现 Debug 线路的连接。

注：CPU 输入的时钟和复位信号均不需要额外处理。所有对外部的读取操作都是时钟

异步的，所有对外部的写入操作都是时钟同步的。

PDU 外设与调试单元模块

```
1   module PDU(
2       input clk,  // system clock
3       input rst,  // system rst
4
5       // ======== Peripherals Part ========
6       // Input: buttons and switches
7       input btn,      // button
8       input [7:0] sw, // sw7-0
9
10      // Output: leds and segments
11      output [7:0] led,     // led7-0
12      output [2:0] hexplay_an,    // hexplay_an
13      output [3:0] hexplay_data,    // hexplay_data
14
15      // Uart: data transmission
16      input uart_din,         // uart_tx
17      output uart_dout,        // uart_rx
18
19
20      // ======== MMIO Part ========
21      // MMIO BUS
22      input [31:0] mmio_addr,
23      input mmio_we,
24      input [31:0] mmio_din,
25      output [31:0] mmio_dout,
26
27
28      // ======== Debug Part ========
29      // CPU control signals
30      output cpu_rst,
31      output cpu_clk,
32
33      // CPU debug bus
34      input [31:0] cpu_check_data,
```

```
35          output [31:0] cpu_check_addr,
36          input [31:0] current_pc,
37          input [31:0] next_pc,
38
39          // MEM debug bus
40          output [31:0] mem_check_addr,
41          input [31:0] mem_check_data
42      );
43
```

PDU 是本次实验的鼎力支柱，负责协调用户与各个模块之间的交流。你并不需要理解全部的 PDU 源代码，只要根据手册上的内容熟练使用即可。 **PDU 部分我们已经为你写好，你不需要修改内部的任何内容。**

PDU 的端口由四部分组成：系统时钟与复位信号、外设交互端口（对应 Top 模块与外界的接口）、 MMIO 端口（负责与 CPU 的内存映射交互，本次实验可以不用）以及调试端口（很重要）。我们已经为你在顶层模块中正确连接了调试端口和其他模块之间的线路，以保证 PDU 的正常工作。

6.3 一些讨论

最后，我们还为大家准备了一些问答，希望能够帮助你了解 PDU 获取调试信息的方式。

1. PDU 如何获取 CPU 的相关信息？

   PDU 从 CPU 的输出端口中获取 PC 的值，以及此时 CPU 与 MEM 单元之间的相关信号。此外，通过传给 CPU 的 cpu_check_addr 信号，PDU 可从寄存器中额外的读端口或数据通路中获取相应的值 cpu_check_data。

2. PDU 如何读取内存中的数据？

   与 CPU 类似，PDU 通过给 DataMemory 的 dpra_dbg 端口传入地址 mem_check_addr，从 DataMemory 的 dpra_dbg 端口中读取数据，通过 mem_check_data 传回。

3. PDU 是如何控制 CPU 运行的？

CPU 和内存的时钟信号都是由 PDU 控制的。PDU 通过向 CPU 和内存的时钟端口传入处理后的时钟信号 cpu_clk 来控制 CPU 和内存的运行。也就是说，只有 PDU 的 clk 来源于外部时钟，CPU 和内存的时钟都是由 PDU 控制的。

本次实验中 CPU 与 MEM 单元的时钟频率均为 1MHz。你可以在 PDU_ctrl.v 文件的最末端看到时钟信号生成的相关代码。

4. CPU 如何获取外设提供的数据（如串口，button 等）（**非实验必做 or 选做的内容**）？

MMIO(Memory Mapped I/O) 是一种通过内存地址访问外设的方式，通过 MMIO，CPU 可以通过以读写内存的方式来访问外设（正如其名，将外设映射到内存地址），我们通过事先约定好的一些地址与外设的对应关系（比如0x10000000对应串口，0x10000001对应 button），让 CPU 读写这些地址来获取外设的数据。

一种可能的实现方式是，当 PDU 检测到 CPU 正在访问外设地址时，通过控制选择器将外设的数据而非内存中的数据传给 CPU。本次实验中助教已经为大家实现了这一功能，大家只需要按照我们提供的约定地址访问相应外设即可。

MMIO(Memory mapping I/O)，即内存映射I/O，将 I/O 设备被放置在内存空间而不是 I/O 空间。从处理器的角度看，内存映射 I/O 后，系统设备访问 I/O 的过程就和访问内存一样。因此，CPU 就可以使用读写内存一样的汇编指令完成对于 I/O 的访问，从而简化了程序设计的难度和接口的复杂性。

什么意思呢？你可以将其理解为有特殊的设备负责在外设与内存单元之间建立联系。这一设备会把外设的结果放入内存单元，也会将内存单元中的数据交给外设。因此，CPU 在运行时就可以通过访问这些内存单元，进而实现与外设的间接交互。本次实验中，这一设备即为我们的 PDU。

外设映射到的内存地址与数据存储器、指令存储器的地址本质上没有任何区别。也就是说，CPU 和汇编程序并不知道该地址单元是否存在与外设之间的映射。这些地址单元与内存单元的区别在于：外设地址单元是可以被外设修改的，而内存地址单元只能被 CPU 的访存指令修改。

7. 外设与调试单元 PDU

在 Lab3 中，我们为大家提供了一个简化版的 PDU，用于帮助大家体验 PDU 的工作流程，熟悉相关代码与操作。从 Lab4 开始，我们将提供具有完整功能的 PDU 供大家使用。

> 本章节是一份对于 PDU 的详细说明手册。实验 PPT 受限于篇幅，很多地方无法展开介绍。为了保证大家后续实验的顺利进行，**强烈建议大家认真阅读本章节的内容！**

顾名思义，PDU 分为调试部分和外设部分。调试部分由用户控制 PDU，在板上环境下检验 CPU 的工作情况；外设部分负责协调平台、用户、CPU 三方之间的数据交互，扩展 CPU 的工作范围。下面是对 PDU 的详细介绍。

7.1 状态界面

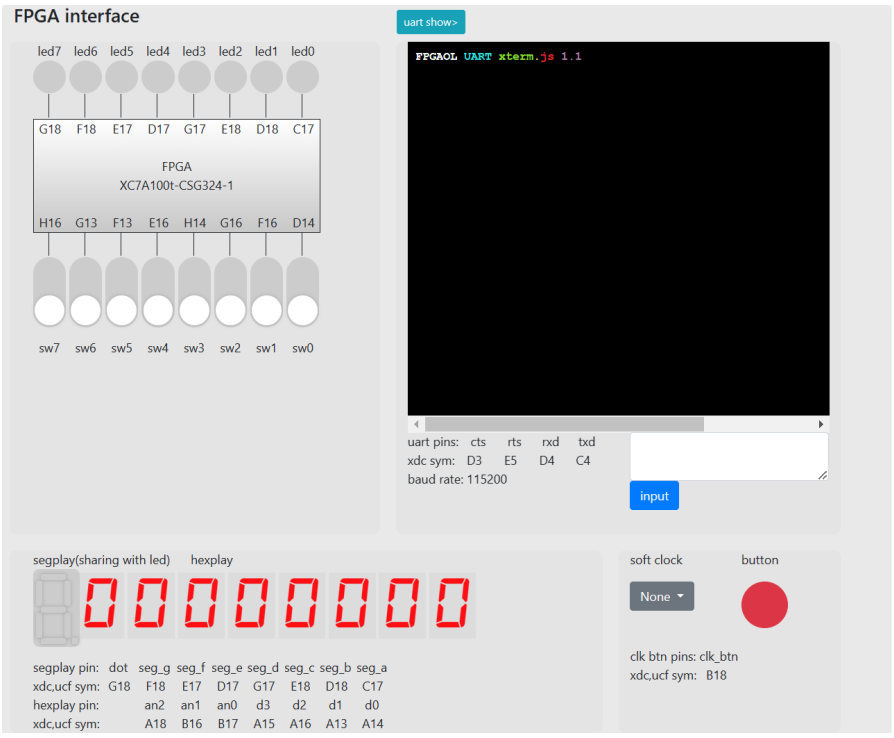


图 17: FPGAOL 界面示意图

回顾 FPGAOL 的操作界面（如上图所示），我们将不同的外设划分成了不同的功能组件。根据功能的不同，我们可以将其概括如下：

- LED——状态显示：led 指示灯表明了当前系统的工作状态。
  - led[7] 为系统状态指示灯。当其亮起时，表明 CPU 正在运行，即为工作状态；熄灭时表明 CPU 停止运行，即为调试状态。初始上板时，系统处于调试状态，此时 led[4] 亮起。
  - led[6:4] 为数码管输出指示灯。当 led[6] 亮起时，数码管显示的是开关输入缓冲区（也就是我们的移位寄存器）中的数值。此时拨动开关可以看到数据的实时变化；当 led[5] 亮起时，数码管显示的是 CPU 向 Segment_output 地址单元写入的数据；当 led[4] 亮起时，数码管显示的是 Check_addr 所对应的 Check_data 的内容。需要注意的是，led[6:4] 在任何时刻只会有一个亮起。
  - led[3:0] 为 CPU 向 MMIO 地址单元写入的数据。该数据不会随着 led[7] 的变化而变化。一般而言，CPU 通过向地址单元中写入不同的数据来展现汇编程序的不同运行状态。

- 开关——用户输入：与 Lab3 的规则基本一致，我们通过拨动 sw[6:0] 进行用户的数据输入。开关输入对应的映射表可以参考 Lab3 的实验 PPT。 **小心！sw[7] 此时不是 del 按键，而是 reset 按键！不要误触了！**如果输入的数据有误，你可以通过不断写入 0 来刷新移位寄存器，进而写入新的数据。

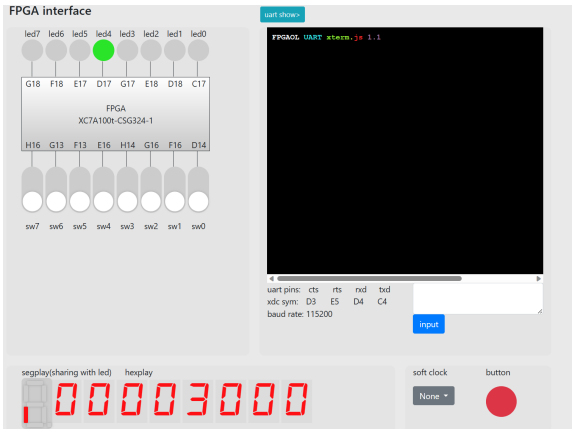
- 开关——系统 Reset：sw[7] 与 PDU 的 rst 信号相连。因此，当 CPU 意外卡死或出现死循环（led[7] 常亮）时，可以通过拨动 sw[7] 回到调试状态（led[4] 亮起）。 **在本次实验中，你将会经常使用这一功能。**

- 数码管——系统输出：数码管只负责输出来自不同位置的 32bits 数据，包括：作为用户输入缓冲区的移位寄存器、PDU 管理的 MMIO 地址单元以及调试信息的输出。不同的数据来源会有不同的 led 指示灯作为提示。

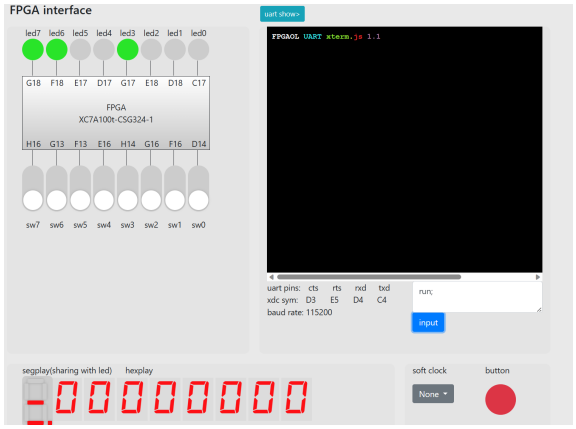
- 串口——命令控制：与去年的 PDU 相比，今年 PDU 最大的改动就是引入了串口通信作为控制方式，从而将其余外设解放出来。用户在串口输入框中输入相应的 PDU 指令， PDU 进行相应的状态跳转，并完成一系列设定的功能。

- 按钮——用户的回车键：本次实验中的按钮被用于用户和 PDU 之间交互式 IO 的确认信号。当用户确认传输给 CPU 的数据输入无误，或是看到来自 CPU 的输出数据无误时，都需要按下按钮作为提示。

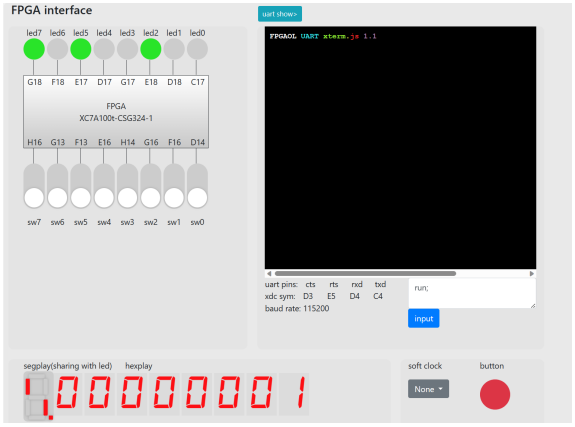
你可能会注意到，在下面的示意图 18 中，用户输入时 led3 也亮起，CPU 输出时 led2 也亮起。这是由于 CPU 在进行输入输出时，向它们对应的地址单元写入了数据而导致的。



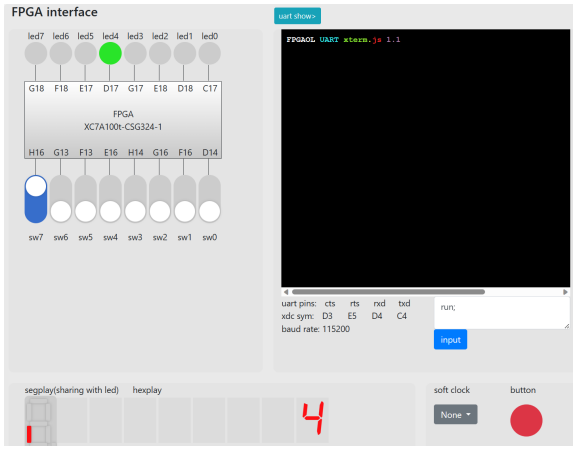
(a) 刚刚上板，此时只有 led4 亮起



(b) 等待用户输入，此时 led6 自动亮起



(c) CPU 进行数码管显示，此时 led5 自动亮起



(d) 拨动 sw7，此时回到调试状态，只有 led4 亮起

图 18: 用户界面示意图

> 注意区分 led[7:4] 与 led[3:0] 代表的不同意义。 led[7:4] 为 PDU 控制，代表系统的工作状态； led[3:0] 为 CPU，即汇编程序控制，代表程序的运行状态。

7.2 串口指令

串口指令是本次实验 PDU 的一大亮点。在去年的实验中，我们约定了复杂的外设使用规定，导致进行相关的调试操作时需要花费较多的精力。为此，今年的 PDU 经过了全新的升级，为大家带来了功能强大的串口指令操控模式。

> 这一部分内容你也可以参考附件中的 PDU 指令手册。我们建议你熟练掌握串口指令的使用，这将有效帮助你检验 CPU 的上板结果。

我们可以大致将指令分为两类：调试指令与 CPU 控制指令。

调试指令

> Check_addr 是我们针对通路中不同的数据，所赋予的唯一编码。也就是说，一个 Check_addr 对应了一种特定的数据。这个数据的来源可以是多样的，例如通路中一个元件的输出结果、寄存器堆中某一寄存器存储的值、数据存储器中某一单元的内容等等。我们通过设置 Check_addr，即可查询相应的内容，并通过 Check_data 传给 PDU，进而显示在数码管上。上板时，Check_addr 初始为 0。
> 本次实验中，Check_addr 为 32bits 整数，一般其高 16bits 均为 0，低 16bits 的对应关系如下：
>
> - Check_addr[15:8] = 8'h00：查看的是数据通路中的相关信息；
> - Check_addr[15:8] = 8'h10：查看的是寄存器堆中的相关信息；
> - Check_addr[15:8] = 8'h20：查看的是数据存储器中的相关信息。

- ck0：查看通路信息。格式为 ck0 + [空格] + [2 个 16 进制数码] + [;]。通路信息的对照表可以参考 ppt，或是我们提供的指令手册。
- ck1：查看寄存器堆信息。格式为 ck1 + [空格] + [2 个 16 进制数码] + [;]。例如 ck1 03; 代表查看 RegFile[3] 的内容。
- ck2：查看数据存储器信息。格式为 ck2 + [空格] + [2 个 16 进制数码] + [;]。需要注意的是，这里的十六进制数码是存储器单元的编号，而不是地址。例如 ck2 01; 查看的是

第 2 个地址单元，对应的数据段地址是 0x0004； ck2 07; 查看的是第 8 个地址单元，对应的数据段地址是 0x001c。

- add：Check_addr 自增 1 。格式为 add + [;]。
- sub：Check_addr 自减 1 。格式为 sub + [;]。

ck 系列指令在输入时，不可以省略后面的数字，否则会导致地址设置错误。例如：ck0 02; ≠ ck0 2;

CPU 控制指令

- bp：设置断点。格式为 bp + [空格] + [4 个 16 进制数码] + [;]。例如：bp 3004; 。请注意，断点地址应当为 4 的整数倍（字节对齐），因此形如 3001 的地址是不合法的。目前，一次只能设置一个断点。
- step：单步运行。格式为 step + [;]。step 指令会让 CPU 运行一个时钟周期（以上升沿开始）。需要注意的是，step 指令会覆盖先前设下的断点，将其更改为当前 PC 的值。因此在 step 之后，上一次 bp 命令的结果就会失效。
- run：连续运行。格式为 run + [;]。一般来说，run 指令会让 CPU 一直运行，直到达到断点或超出了 0x3ffc 的范围。因此建议大家在使用 run 指令之前，先通过 bp 指令设置断点。
- rst：CPU 复位。格式为 rst + [;]。一条没有写在指令手册上的隐藏指令！该指令会控制 cpu_rst 信号，发出一个系统时钟周期的高电平脉冲。还记得我们之前 PC 寄存器中的异步复位信号吗？这里就是它的作用。rst 指令可以将 PC 复位到 0x2ffc（或 0x3000）。需要注意的是，数据存储器和寄存器堆无法复位。因此尽管程序可以从头重新开始执行，但执行的结果有可能有所不同。

> 7.3 节与 7.4 节不是本次实验的必做与选做内容，因此你可以跳过这两小节的阅读。

7.3 MMIO 约定

PDU 提供的 MMIO 地址单元介绍如下：

- Button_status-0x7f00：初始值(复位值)为 0，CPU 可读可写。在按钮按下时，PDU 会自动将该地址单元置一。一般而言，CPU 通过反复读取该地址单元的值，判断用户是否按下了按钮(轮询)。为了消除之前按钮信息的影响，在读取该地址单元之前，CPU 需要先通过 sw 指令将其置零。一个简单的轮询查询例子如下

```
1    # Suppose t0 stores the value 0x7f00
2    sw x0 0(t0)     # This line is VERY important!
3    Read:
4        lw s0 0(t0)
5        beq s0 x0 Read
6    # Do something next
7
```

- Switch_input-0x7f04：初始值(复位值)为 0，CPU 只读。该地址单元会在按钮按下时保存用户通过开关输入的值，也就是移位寄存器的值。 CPU 通过读取该地址单元获得来自用户的输入。

- Segment_output-0x7f08：初始值(复位值)为 0，CPU 可读可写。 CPU 可以向该地址单元写入数据，从而显示在数码管上。PDU 会自动检测来自 CPU 的写入，并将数据交付给数码管输出单元。

- Led0-0x7f0c：初始值(复位值)为 0，CPU 可读可写。若为 1，则对应的 LED 灯亮起。

- Led1-0x7f10：初始值(复位值)为 0，CPU 可读可写。若为 1，则对应的 LED 灯亮起。

- Led2-0x7f14：初始值(复位值)为 0，CPU 可读可写。若为 1，则对应的 LED 灯亮起。

- Led3-0x7f18：初始值(复位值)为 0，CPU 可读可写。若为 1，则对应的 LED 灯亮起。

四个 LED 地址单元可以用来表示 CPU 的运行状态。本次实验中，我们约定如下：

- led0：程序运行正常指示灯
- led1：程序运行异常指示灯
- led2：程序正在输出指示灯
- led3：程序等待输入指示灯

不同的指示灯亮起时，就代表 CPU 正工作到对应的状态。

> 需要注意的是，所有的地址单元存放的数据都是 32bits 宽度的。对于 led 的控制，我们实际上读取的是 32bits 中最低位的数据，也就是 led[0] = LED[0]。

7.4 交互式 IO

与 RARS 的 IO 过程不同，PDU 的 IO 采用交互式方式进行。这样的设计更接近我们日场生活中的体验。例如：当用户需要向 CPU 输入数据时，需要在输入完成后按下按钮作为确认；当 CPU 需要向用户展示数据时，用户也需要按下按钮，告知 CPU 当前的数据已经被阅读。详细的流程介绍如下：

7.4.1 交互式用户输入

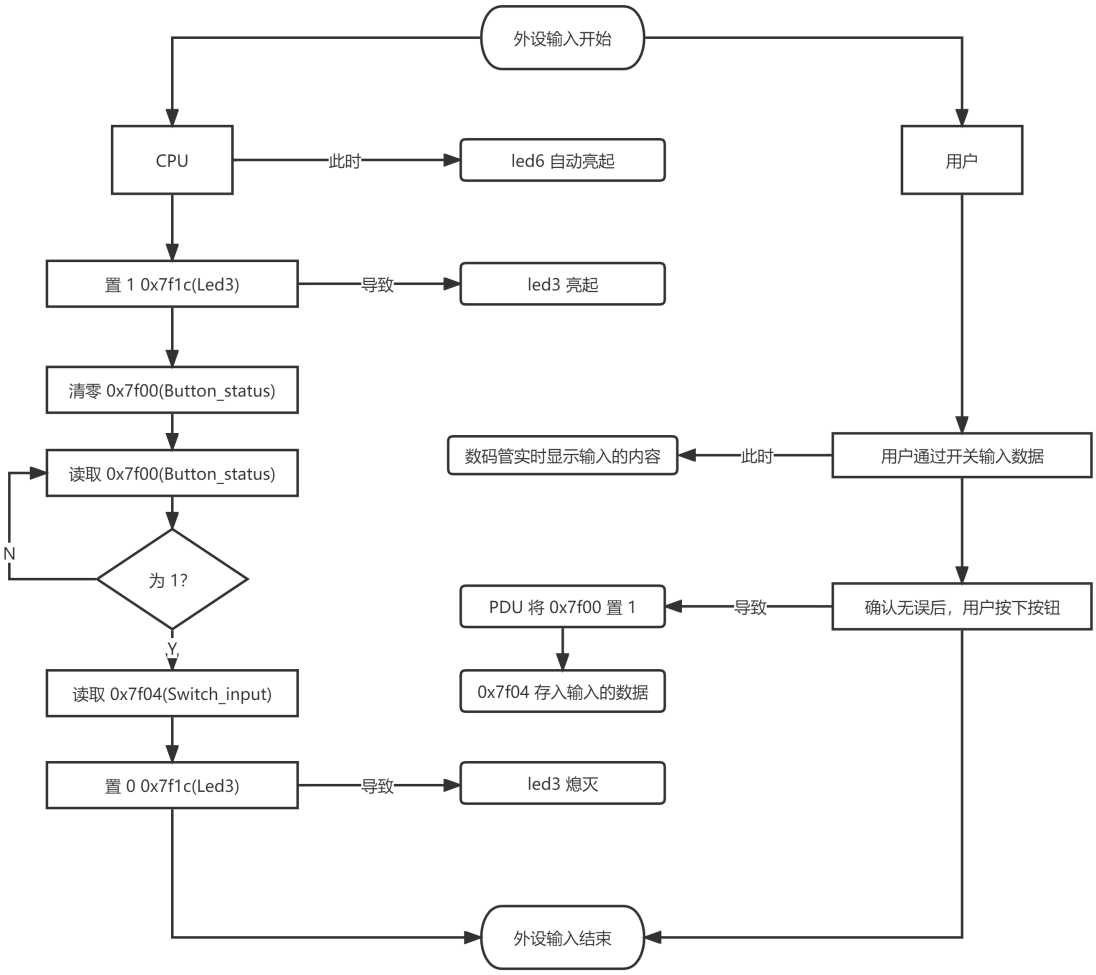


图 19: 交互式输入流程图

该过程中，CPU 需要用户通过 sw[6:0] 输入数据，并按下按钮确认输入完成。

7.4.2 交互式用户输出

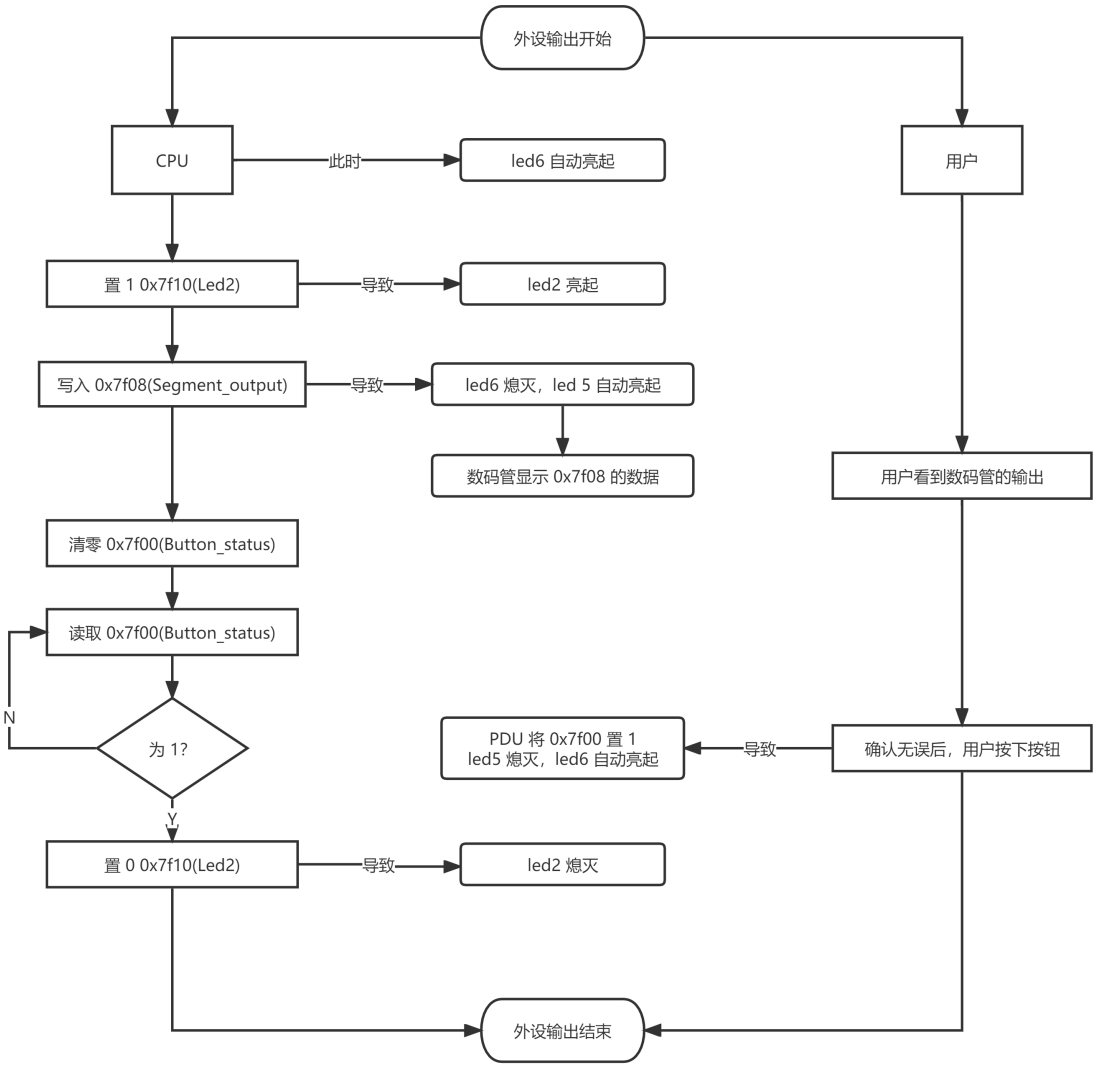


图 20: 交互式输出流程图

该过程中，CPU 需要通过数码管输出数据，并由用户按下按钮确认已经阅读。

交互式 IO 过程中，你应当根据我们指定的 MMIO 规则，读取或写入指定地址的内容。**本次实验中，我们不要求你实现交互式 IO 的汇编程序设计。**

7.5 调试流程（这一部分很关键）

最后，我们来介绍一下使用 PDU 进行调试的一般流程。我们建议你在仿真结果正确后再上板测试，这将为你节省很多调试的时间。PDU 提供的调试功能更多的是一种正确性验

证。

注意：在使用 PDU 进行调试之前，你需要确保：

- CPU 内部的所有模块都已经实现（正确性可以不用验证）， PC 寄存器可以被正确写入与读出；
- CPU 内部的 cpu_check_addr 和 cpu_check_data 已经被正确连接。这一部分中你需要参考 PDU 指令手册上关于 ck0、ck1 两条指令的相关内容，根据 cpu_check_addr 选择 cpu_check_data 的来源。你可以参考图 21 来实现，也可以根据我们提供的单周期 CPU 数据通路图来实现。

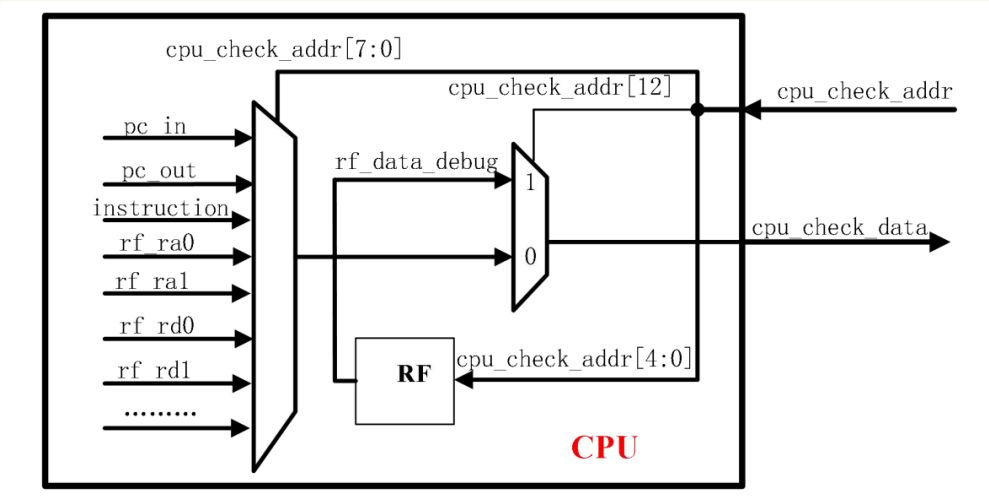


图 21: CPU 内部 debug 接口信号处理示意图

如果 cpu_check_addr 为 0x0XXX，则代表查询的是 CPU 内部通路上的信息，如果 cpu_check_addr 为 0x1XXX，则代表查询的是寄存器堆的信息。因此，可以通过检查 cpu_check_addr[12] 来判断 Debug 数据的来源。

- MEM 内部的指令与数据存储器已被正确例化并初始化。上板后，我们需要 PDU 控制 CPU 执行相应的汇编程序，因此需要保证此时 MEM 模块中已经保存了待检测的汇编程序。

假定你已经正确连接了 PDU，在上板之后，你看到的界面会如下图所示。

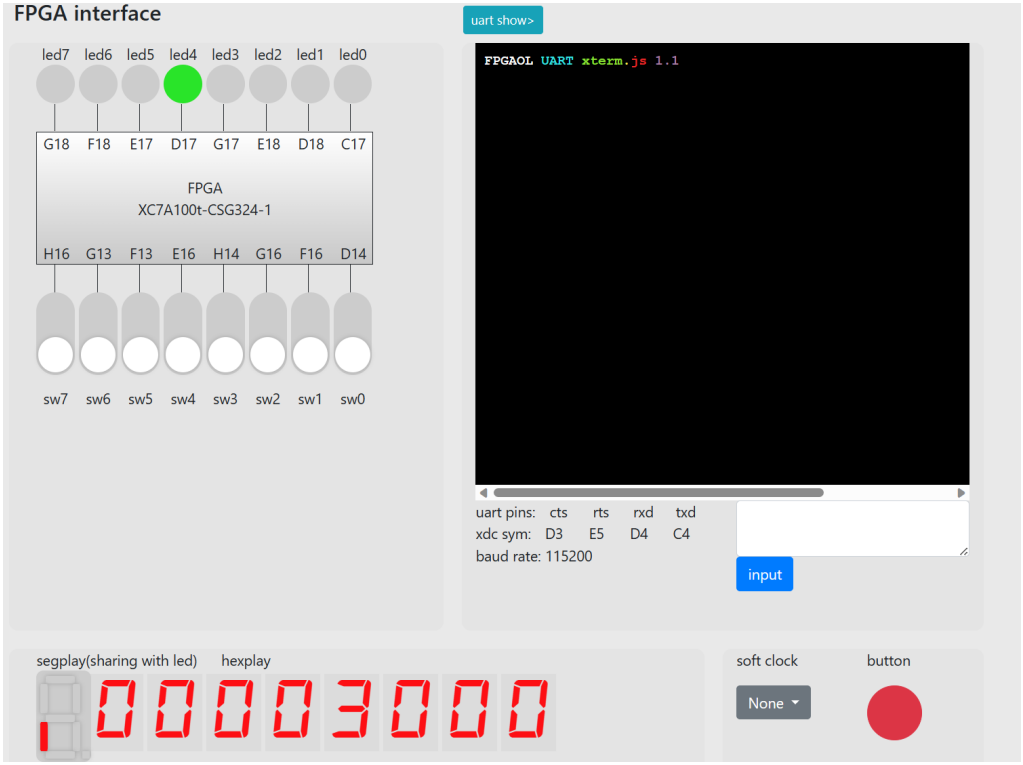


图 22: 上班初始界面示意图

此时，Debug_addr 为 0x00000000。我们首先输入 add; 指令，将 Debug_addr 设置为 0x00000001，对应查看的内容为 pc_in。这个时候，数码管显示的内容为 0x3000（有可能是 0x3004，根据个人的实现），代表即将执行的指令是地址 0x3000 处存放的指令。

> 如果你在执行 add; 后数码管显示依然为 0x00000000，则表明 CPU 内部 PC 寄存器的写入数据（也就是 next_pc）没有被正确设置。请检查这一部分的电路逻辑是否正确。

接下来，在串口指令区输入 step; ，让程序单步运行。每一次单步运行后，你都可以根据我们的 PDU 指令手册查询通路中的相关信息，并与 Rars 或 Ripes 上的结果进行对比。

如果你希望验证我们提供的 Testcase 程序，可以在上板后先通过命令 bp 3400; 设置断点为 0x3400，随后输入命令 run; 运行 CPU，程序会自动运行到相应的死循环处。你可以通过观察 led0 以及 led1 的亮灭情况判断此时 CPU 的运行结果。假定此时你发现 led7、led6 和 led1 亮起，这说明测试程序运行至结果错误对应的死循环处。你可以按照如下的顺序进行检查：

(1) 拨动 sw7，该开关对应着系统 reset 信号。开关拨下后，PDU 会中断 CPU 时钟信号，将状态由 CPU 运行状态（led7 亮起）转变为调试状态（led7 熄灭，led4 亮起）。

(2) 使用调试指令 ck1 03; 查看寄存器 x03 中存储的数值。该数值对应的是最后出现异常的 Test 编号。你需要根据该编号，从我们提供的 Test 程序中找到相应的测试部分，查看该部分的起始指令地址（不妨假定为 0x3XXX）。

(3) 重新烧写 bit 流，设置断点 bp 3XXX; 随后输入 run; ，让 CPU 恰好运行到该测试样例的开始处。

(4) 接下来，通过 step; 指令单步运行 CPU，并使用 ck 系列指令设置相应的 check_addr 查看 CPU 内部的信息，并于自己在 Rars 或 Ripes 上的结果进行对比，直至找到执行结果不一致的指令，即可定位 CPU 的 bug。我们建议你关注以下内容： ALU 的运算结果、PC_in 的选择、立即数 imm 的结果、相关控制信号等。

如果有其他的调试需求，请查阅 PDU 指令手册或询问助教。

8. 实验任务

本次实验所需完成的各项工作介绍如下：

【必做部分】

1. **根据单周期 CPU 数据通路，结构化描述单周期 CPU，并进行功能仿真。** 请结合我们提供的数据通路图与所学知识，搭建单周期 CPU。我们建议你在 CPU 内部按照我们给出的通路实现相关的模块设计。

   本次实验的单周期 CPU 需要支持以下 10 条指令的功能： add、addi、lui、auipc、beq、blt、jal、jalr、lw、sw。在内存单元方面，你需要例化 ROM 作为指令存储器，例化 DRAM 作为数据存储器。两个存储器的容量均为 256x32bits。

   关于功能仿真，你可以自行设计一些简单的汇编程序，导出 COE 文件进行初步验证。我们为你准备了包含 CPU 和 MEM 的仿真文件 CPU_tb.v，你可以通过该文件检查自己的 CPU 运行情况是否正常。（仿真时记得用 COE 文件初始化存储器）当仿真部分通过后，再使用我们提供的汇编程序进行上板测试。

> 为了与 PDU 相连，数据存储器所使用的 DRAM 的选项应设置为真正双端口。此外，你还需要为 Lab2 中设计的寄存器堆增加 1 个用于调试的读端口。此时的寄存器堆有三个读端口，一个写端口。注意：RegFile[0] 依然需要保持恒零。
>
> 此外，你需要根据 PDU 指令手册，实现 ck0、ck1 两条指令的 CPU 模块内部连接。你可以参考 7.5 小节的内容进行实现。

2. **使用本次实验提供的测试程序生成的 COE 文件作为指令存储器的初始化文件，验证 CPU 的功能正确性。** 本次实验中，我们为你提供了一个正确性验证程序（Testcase），你可以在附件中找到它。该测试程序将会检测你的 CPU 设计是否存在漏洞，并给出相应的检测结果。将整个项目下载至 FPGA 中测试，采用串口调试功能查看测试程序的运行结果。

3. **使用 Lab3 必做实验要求 1 生成的 COE 文件作为指令存储器与数据存储器的初始化的文件，验证 CPU 的功能正确性。** 该 COE 文件为斐波那契数列的计算程序，且不涉及外设输入输出。将整个项目下载至 FPGA 中测试，采用串口调试功能查看数据存储器以及寄存器中的数据。你可以据此观察程序的运行结果。

【选做部分】

1. **扩展单周期 CPU 指令集。** 本项选做为独立内容，你需要在必做部分 1 的基础上进行修改。在原有的 10 条指令的基础上，增加对于下面指令功能的支持：

   - 移位指令 sll、slli、srl、srli、sra、srai
   - 算数与逻辑指令 sub、xor、xori、or、ori、and、andi
   - 条件置数指令 slt、slti、slti、sltiu
   - 分支指令 bne、bge、bltu、bgeu
   - 访存指令 lb、lh、lbu、lhu、sb、sh

   请从上面列出的指令中，任选至少三类，每一类任意实现至少三条指令。你也可以实现更多的指令，但不会有额外的分数。你需要根据需要修改单周期 CPU 通路或 MEM 单元中的部分内容，以增加对于这些指令的支持。同时，你也需要参考我们提供的 Testcase 中的样例程序，自行编写汇编程序以验证你所实现的指令的功能。

本次实验需要大家在实验平台上在线提交相关内容。你提交的文件结构应当满足下面的文件树格式：

```
/
└── lab4_姓名_学号_ver尝试编号
    ├── figs ..... 图片文件夹，如果没有可以无此文件夹
    ├── Lab4_姓名_学号.pdf ..... 实验报告文件
    ├── src ..... 需要提交的相关程序文件夹
    │   ├── Module_name.v ..... 非仿真 .v 文件
    │   ├── Program_name.asm ..... 汇编源程序文件
    │   └── ...
    └── others ..... 其他你打算提交的文件，如果没有可以无此文件夹
```

请将全部文件按照上面的格式压缩成一个文件，提交到实验平台上。

请确保你的实验报告至少包含以下的内容：

- 实验原理。请根据自己的理解描述本次实验的实验内容以及设计流程，包括部分模块的设计思路，如 Control、Imm、Branch 等；
- 分析本次实验提供的单周期 CPU 数据通路和教材上的单周期 CPU 数据通路的差异。这些差异会在哪些指令上体现出来？
- 实验过程中遇到的一些问题，或者难以解决的内容。你可以记录自己试错的过程，也可以展开自己的心路历程（本项内容不作为评分依据）。

实验手册中有一些我们为大家列出的思考点。这部分内容无需在实验报告上列出。此外，我们也欢迎大家在实验报告中给出对于本次实验的反馈。

> 实验检查与报告提交的 DDL 按照各班各组的约定设置。超出 DDL 的检查与提交将按照规定扣除部分分数。请保证个人实验的独立完成！

9. 附件

本次实验所提供的相关文件如下：

```
/
└── /lab4_files
    ├── figs ..... 图片文件夹
    │   ├── Datapath.png ..... 我们为你绘制的单周期 CPU 数据通路图
    │   └── Top_module.png ..... 老师们绘制的顶层模块示意图
    ├── PDU_src ..... PDU 源代码文件夹
    │   └── ...
    ├── Testcase ..... 测试程序文件夹
    │   └── test.asm ..... 测试汇编程序
    ├── PDU指令手册.xlsx ..... 详细的 PDU 指令手册
    └── lab4.pdf ..... Lab4 实验文档
```

睿客网盘链接：https://rec.ustc.edu.cn/share/7b2aeb60-dc16-11ed-bc95-f534fe1d5c94

> 考虑到我们可能会更新附件内容，请大家定期关注群聊中的消息。我们会在此链接中进行文件更新。