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 17 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。

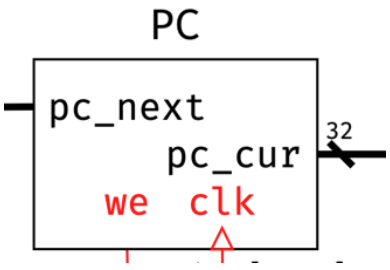
助教为本次实验提供了功能相当强大的 PDU 供大家使用，为了使大家设计的 CPU 能够更好的与提供的 PDU 通信，请务必仔细阅读助教提供的完整的数据通路。与往届实验和课本稍有不同，我们将存储器(指令和数据存储器)在模块层次设计中提高了一层，使其与 CPU 处于同一层级(这相当于将内存作为 CPU 外部设备而非 CPU 的一部分)，来支持 MMIO(Memory-mapped I/O)，即内存映射 I/O。

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

PC 寄存器存储了将要执行的指令的地址，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。



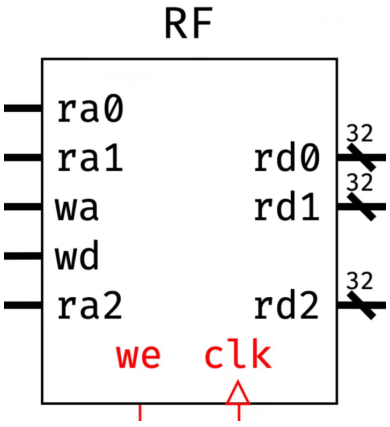
> 你可能注意到，PC 寄存器的行为可以实现为一个单独的寄存器，但模块化设计下我们仍然建议你将 PC 寄存器设计为一个单独的模块，这样可以使得你的设计更加清晰，也避免一个 module 中有过多功能代码。

需要补充的是，我们建议你为 CPU 设计一个异步复位信号接口，并将异步复位值设定为 0x2ffc。这将在接下来的调试过程中有所帮助。

3.2 寄存器堆

寄存器堆我们已经在 lab2 中基本完成了，它有 1 个写端口，2 个读端口。

> 这样的设计是因为我们 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：$addt0, t1, t2$，这条指令就需要读取 $t1$ 和 $t2$ 寄存器的值，经 ALU 运算后将结果写入 $t0$ 寄存器中。

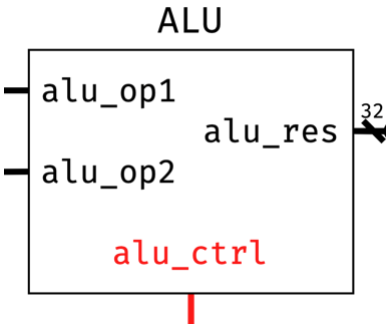


为了方便与 PDU 的通信，我们需要在寄存器堆**额外增加一个读端口**(ra2-rd2)，用于 PDU 从外部读取寄存器的值。请注意：这个端口不应该被汇编指令使用，也就是不能接入 CPU 的数据通路之中。

3.3 ALU

ALU 是算术逻辑单元，它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。现在，我们需要将其放入单周期 CPU 中。

> 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，这可能需要众多的选择器。我们将在下面的数据通路中详细介绍。



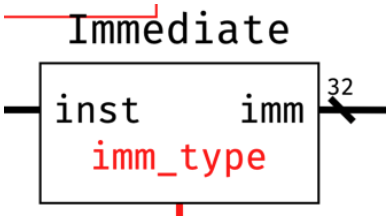
事实上，你几乎可以直接使用 Lab1 中的 ALU，不过溢出位的输出我们暂时不需要，所以可以直接将其悬空（例化时不连接该输出端口即可）。

Lab1 中的 ALU 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32位(有符号还是无符号？)，将结果送给 ALU。IMM 的设计是相当简单的，直接根据不同的指令取出其指令格式中的立即数然后进行相应的扩展即可。

> 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!

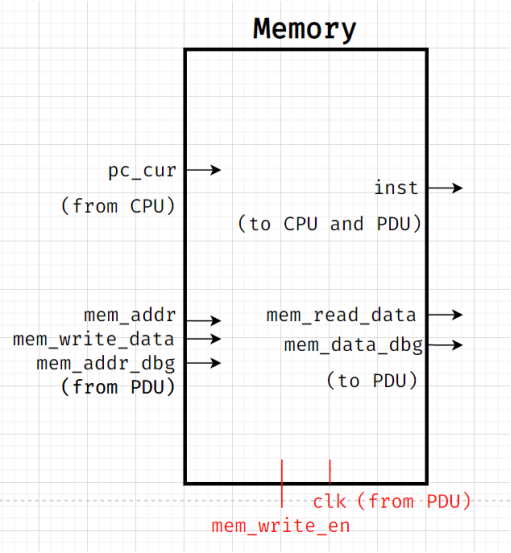


你可能会有疑问：RISC-V 中的立即数有很多种，为什么我们只有一个输出端口呢？这是因为无论是什么格式的立即数，我们都可以通过输入指令以及控制信号来唯一确定其内容。为了简化数据通路，减少不必要的开销，我们将所有种类的立即数从一个端口中输出。为此，你需要根据输入的指令仔细考虑应当输出的立即数格式。

> Control 单元也是 CPU 的一个重要部分，它的功能是根据当前指令的 opcode，控制各个模块行为，它与数据通路紧密相关，我们将在单周期 CPU 数据通路中加以介绍。

4. 内存模块

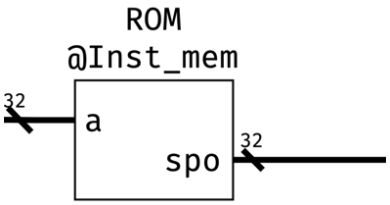
与课本以及往年实验不同，本次实验中我们将内存从 CPU 中分离出来作为统一的模块，它将依据 CPU 与 PDU 的控制信号进行读写操作。内存分为指令存储器和数据存储器两部分。总体可以表示如下：



> 未标注的情况下，含有 dbg 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。

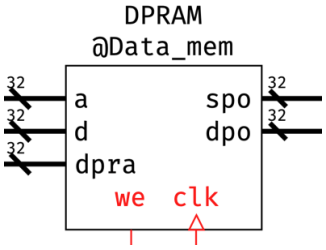
4.1 指令存储器

指令存储器的功能是根据 PC 寄存器的输出地址，将指令从存储器中读出，然后将指令送出。由于汇编程序不会修改指令存储器的值（这是不合法的），我们直接例化一个分布式单端口 ROM 即可。

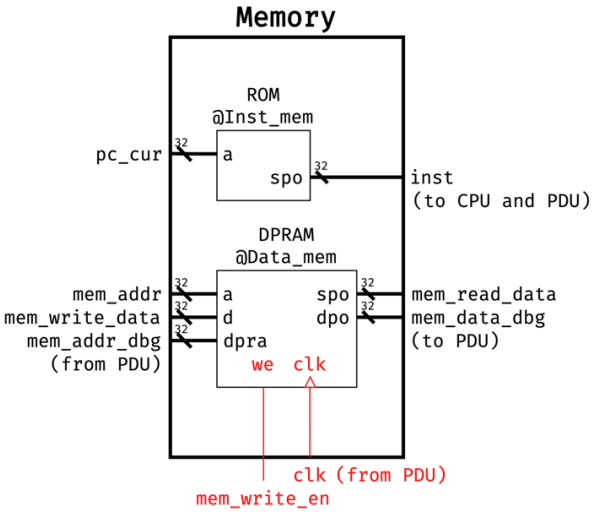


4.2 数据存储器

数据存储器的功能是根据 ALU 的输出地址，将数据从存储器中读出，或者将数据写入存储器。为了我们的 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 从外部读取数据，而不影响 CPU 的正常工作。



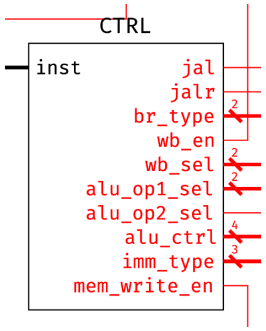
内存单元 MEM 的内部整体结构呈现如下：



5. 单周期 CPU 的数据通路

5.1 控制器

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。



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：

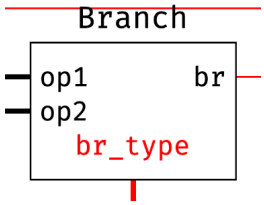
- NextPC 选择信号，来源于 jal、jalr 以及分支指令。Branch 模块也参与了这部分信号的生成；
- 存储器控制信号，在我们的实验中为写使能信号；
- ALU 源操作数选择信号，即 ALU 的两个输入数据的来源；
- ALU 模式信号，用于控制当前进行的运算种类；
- 立即数类别信号，该信号用于立即数模块的类型判断，从而选择产生相应的立即数；
- 寄存器堆写入数据选择信号，该信号用于控制即将写入寄存器堆的数据的来源。

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。

5.2 分支模块

与教材不同，本次实验的通路将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 Ctrl 的控制信号，以

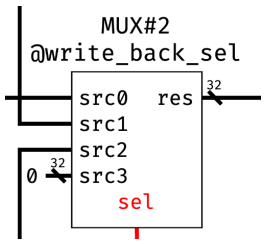
及来自寄存器堆的两个数据。根据这两个数据之间的大小关系，以及控制信号产生相应的跳转信号。下面是分支模块的结构示意图。



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

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。

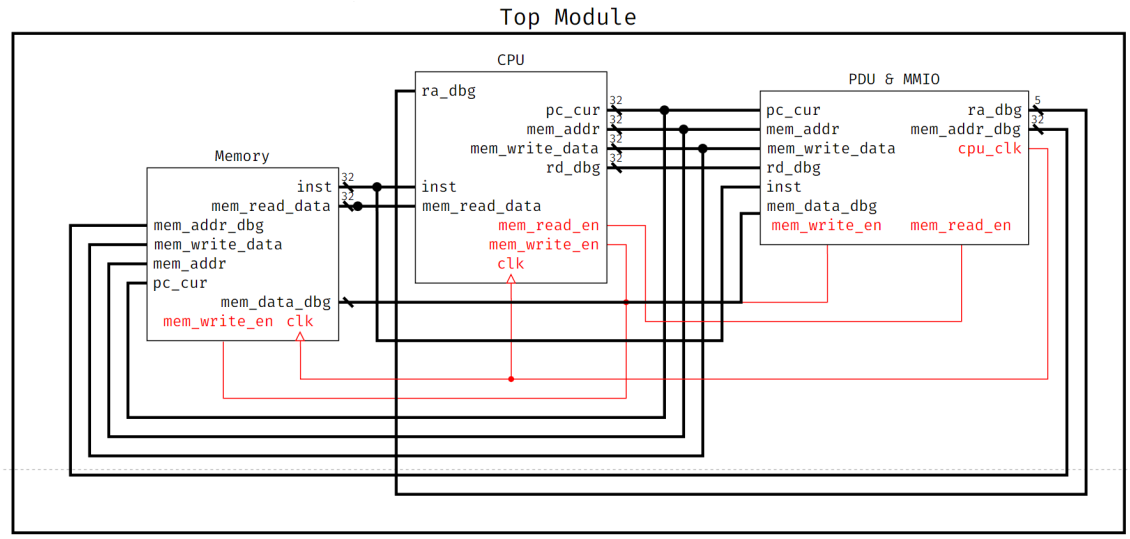


为什么要有如此“多此一举” 的设计呢？一方面，这样可以让通路更为整齐，减少杂乱的门电路的出现（看不见就是没有.jpg）。另一方面，我们可以更为直观地看到各个模块之间数据的传输关系，便于和我们提供的单周期数据通路进行比对。

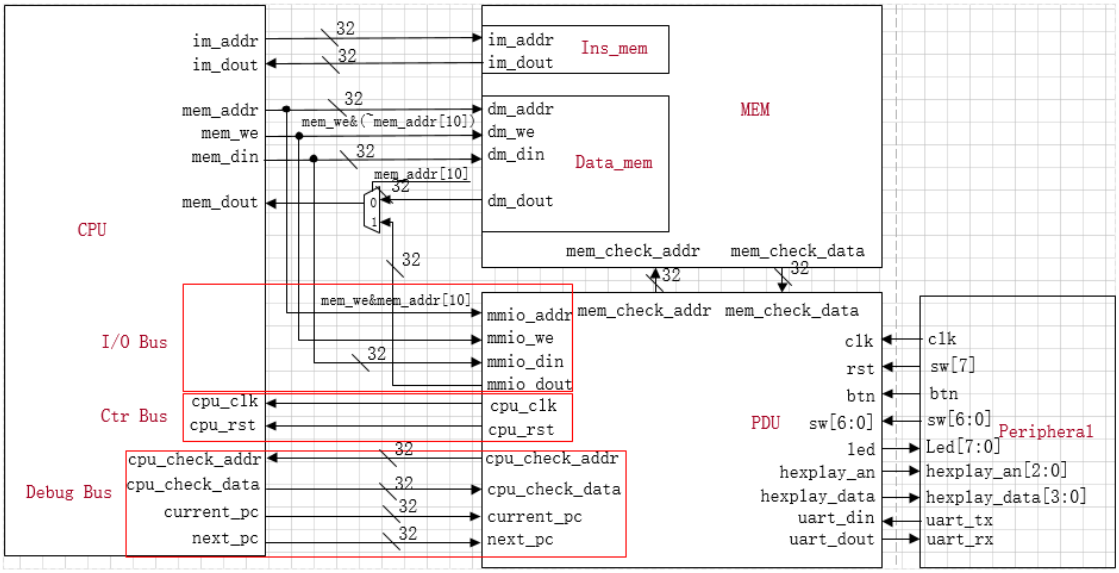
在本次实验中，选择器共有如下的应用：ALU 源操作数的选择、Next PC 的选择，以及写入寄存器的数据选择。你可以在实验中例化四个选择器来实现这些连接。

6. 顶层模块设计

本次实验的顶层模块由 CPU、PDU 以及 MEM 组成。以下是对顶层模块的整体呈现：



你也可以参考老师们画的顶层模块关系图。注意到两幅图中部分端口名称并不相同，但实际上它们是满足对应关系的。



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。 **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。

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，不要求回答）、指令存储器访存接口、数据存储器访存接口，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 Debug 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。

CPU 中央处理器模块

```
1     module CPU(
2         input clk,
3         input rst,
4
5         // MEM And MMIO Data BUS
6         output [31:0] im_addr,
7         input [31:0] im_dout,
8         output [31:0] mem_addr,
9         output  mem_we,
```

```
10          output [31:0] mem_din,
11          input [31:0] mem_dout,
12
13          // Debug BUS with PDU
14          output [31:0] current_pc,
15          output [31:0] next_pc,
16          input [31:0] cpu_check_addr,
17          // Check current datapath state (code)
18          output reg [31:0] cpu_check_data
19          // Current datapath state data
20      );
21
```

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。你需要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。

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，但能理解更好。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）。你需要正确连接调试端口和其他模块之间的线路，以保证 PDU 的正常工作。

此外，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

（而并不需要知道 PDU 内部如何实现）：

1. PDU 如何获取 CPU 的相关信息？

   从 CPU 的输出端口中获取 PC，mem_addr，mem_write_data。此外，通过传给 CPU 的 ra_dbg 信号，从寄存器中额外的读端口或数据通路中获取相应的值 rd_dbg。

2. PDU 如何读取内存中的数据？

   通过给 DataMemory 的 dpra_dbg 端口传入地址 mem_addr_dbg，从 DataMemory 的 dpra_dbg 端口中读取数据，通过 mem_data_debug 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。

4. CPU 如何获取外设提供的数据(如串口，button 等)？

   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 可以通过正常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 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 时就和访问内存一样。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。

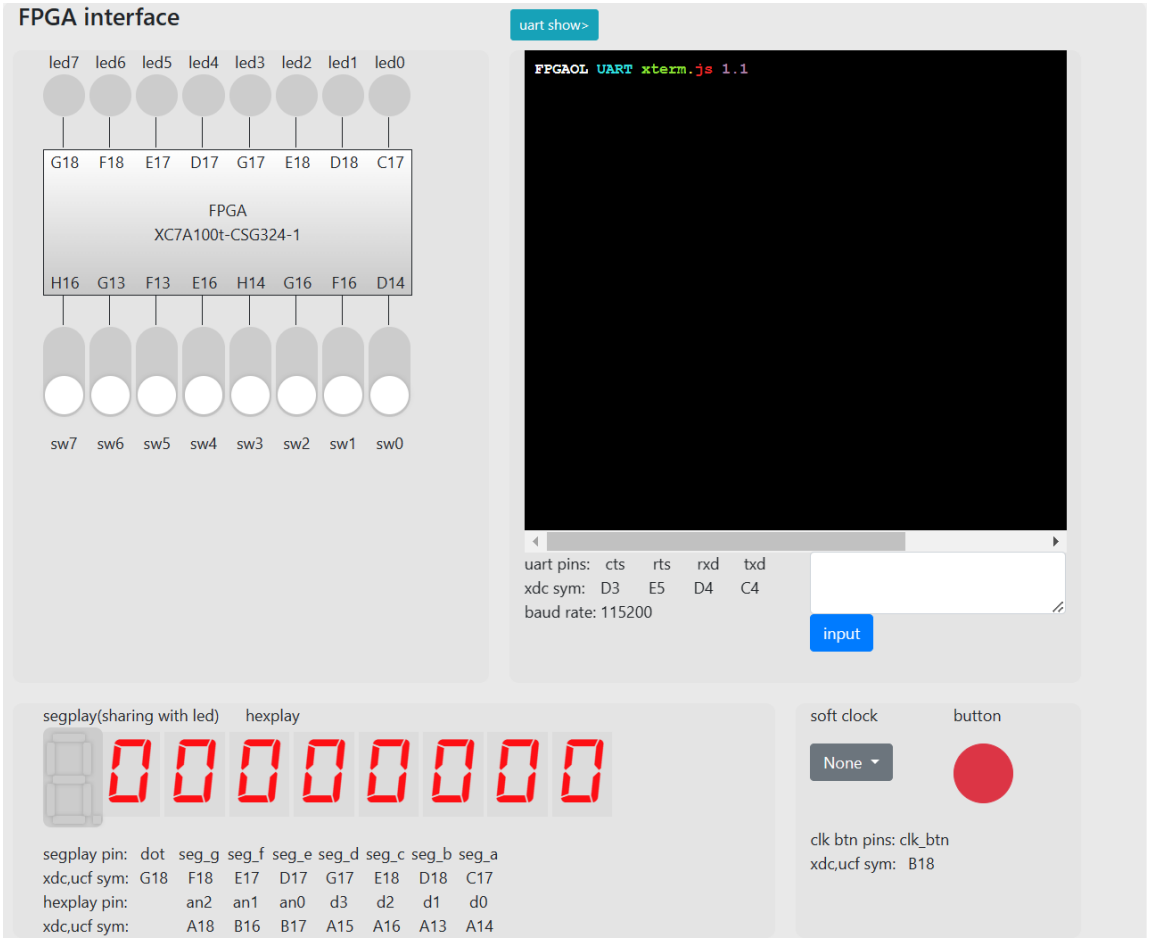
7. 外设与调试单元 PDU

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。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。调试部分负责用户控制 CPU，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。这两部分的详细介绍如下。

7.1 状态界面



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。 **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）。

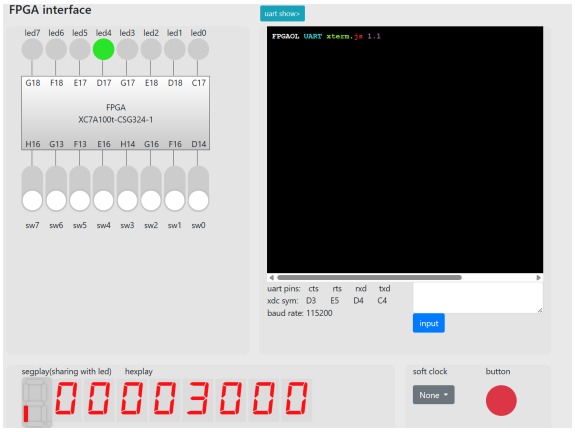
- 数码管——系统输出：数码管只负责输出来自不同位置的 32bits 数据，包括：作为用户输入缓冲区的移位寄存器、PDU 管理的 MMIO 地址单元以及调试信息的输出。不同的数据来源会有不同的 led 指示灯作为提示。

- 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。

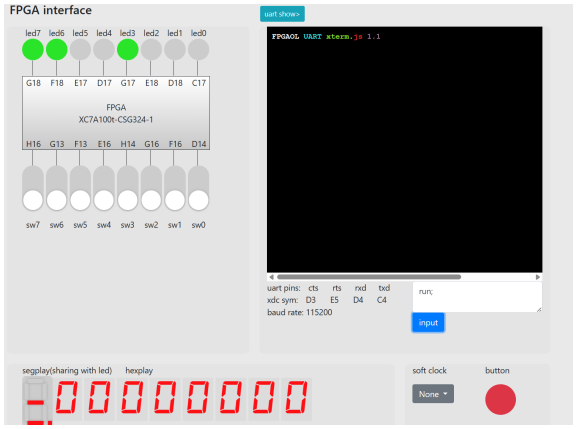
- 按钮——用户的回车键：本次实验中的按钮被用于用户和 PDU 之间交互式 IO 的确认信号。当用户确认传输给 CPU 的数据输入无误，或是看到来自 CPU 的输出数据无误时，都需要按下按钮作为提示。

你可能会注意到，在下面的示意图 1 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。

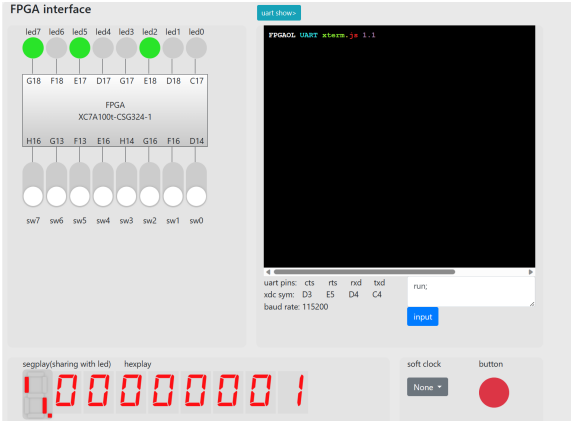
> 注意区分 led[7:4] 与 led[3:0] 代表的不同意义。 led[7:4] 为 PDU 控制，代表系统的工作状态； led[3:0] 为 CPU，即汇编程序控制，代表程序的运行状态。



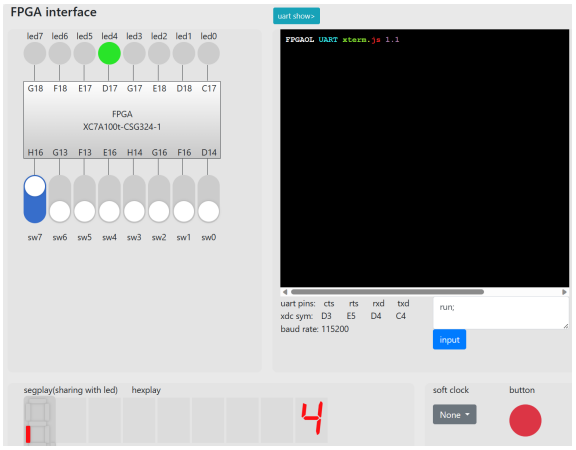
(a) 刚刚上板，此时只有 led4 亮起



(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 1: 用户界面示意图

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，进行相关的调试操作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。

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。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。

- 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，不是地址。例如 ck2 01; 查看的是第 2 个地址单元，对应的地址是 0x0004； ck2 07; 查看的是第 8 个地址单元，对应的地址是 0x001c。
- add：Check_addr 自增 1 。格式为 add + [;]。
- sub：Check_addr 自减 1 。格式为 sub + [;]。

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。还记得我们之前在 PC 寄存器中增加的异步复位信号吗？这里就是它的作用。rst 指令可以将 PC 复位到 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

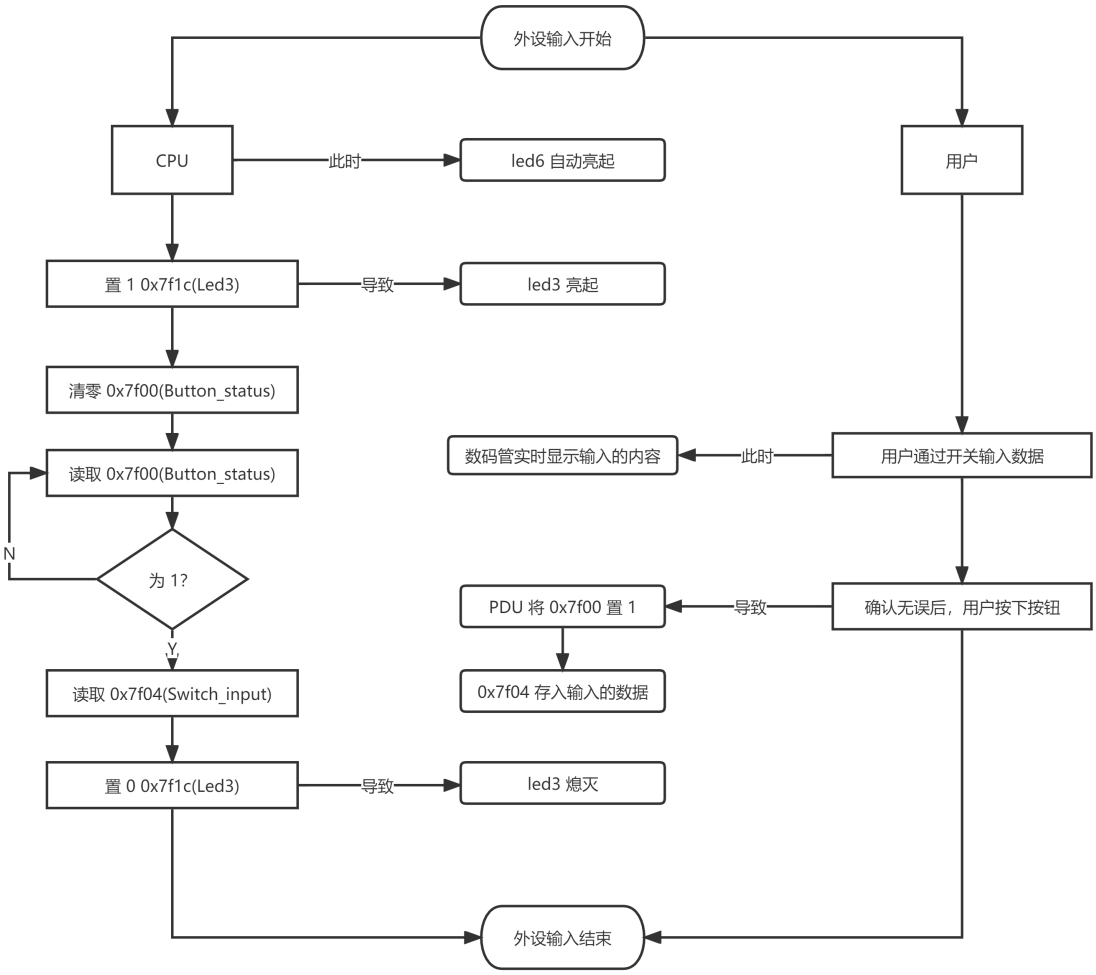
不同的指示灯亮起时，就代表 CPU 正工作到对应的状态。

> 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

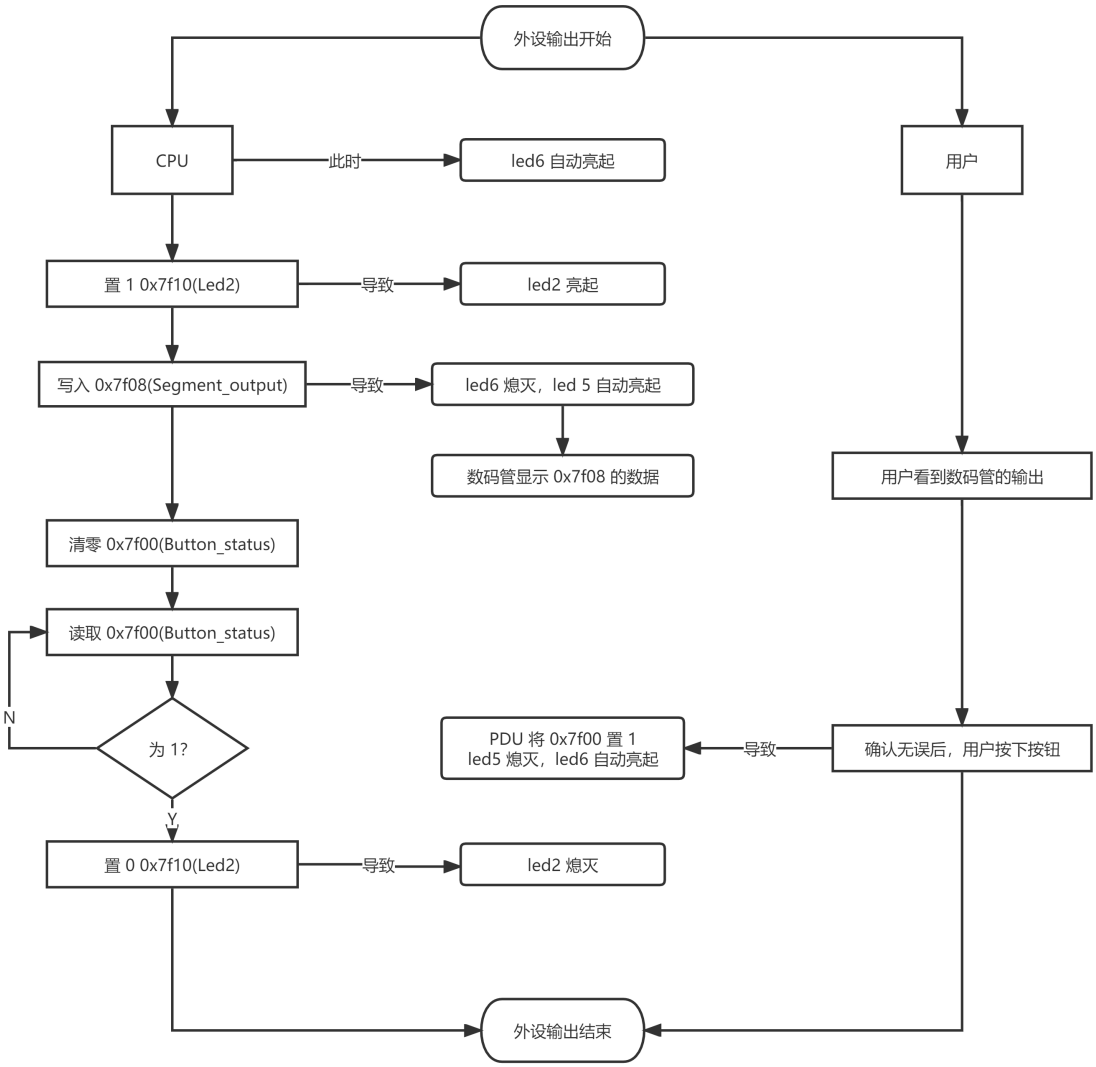
与 RARS 的 IO 过程不同，PDU 的 IO 采用交互式方式进行。这样的设计更接近我们日场生活中的体验。例如：当用户需要向 CPU 输入数据时，需要在输入完成后按下按钮作为确认；当 CPU 需要向用户展示数据时，用户也需要按下按钮，告知 CPU 当前的数据已经被阅读。详细的流程介绍如下：

7.4.1 交互式用户输入



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



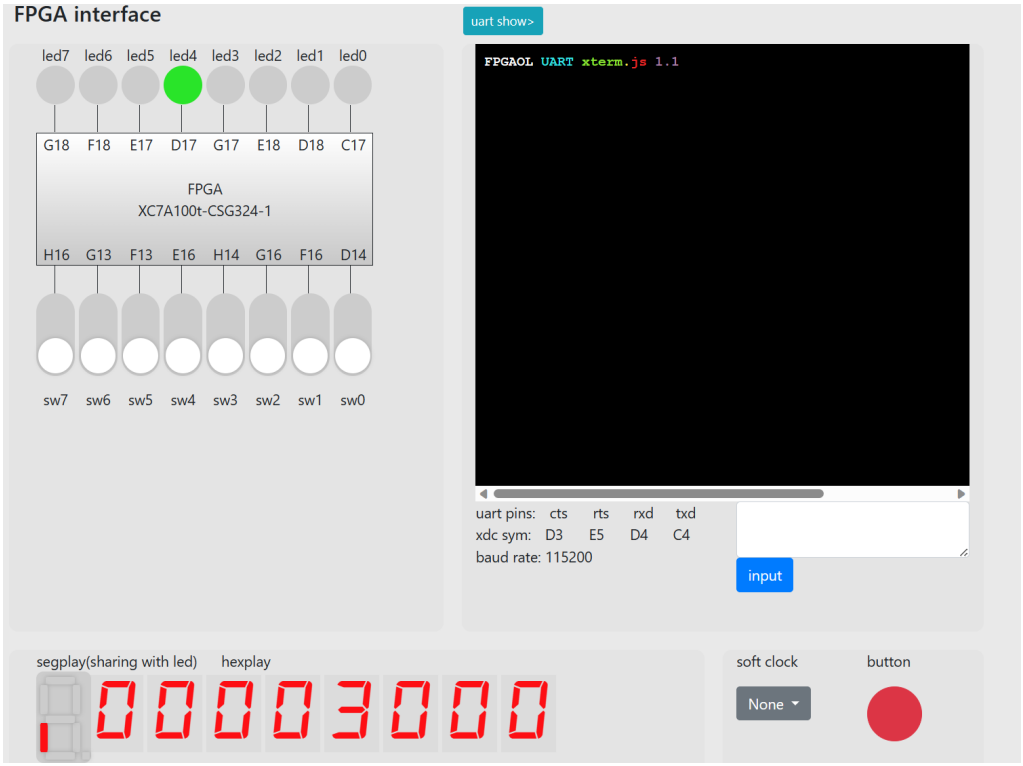
该过程中，CPU 需要通过数码管输出数据，并由用户按下按钮确认已经阅读。

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

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

假定你已经正确连接了 PDU，在上板之后，你看到的界面会是这样的



此时，Debug_addr 为 0x00000000，对应查看的内容为 pc_in。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 0x3000 处存放的指令。

接下来，在串口指令区输入 step;，让程序单步运行。接下来你可以根据我们的 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。你可能会关注的内容包括：Control 的控制信号、寄存器堆的内容、数据存储器的内容等。

如果你希望验证我们提供的 Testcase 程序，你可以在上板后直接输入 run; 并执行。程序会自动跳转到相应的死循环处。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。

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。该 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。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。

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。** 本次实验中，我们为你提供了一个正确性验证程序（Test case），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 CPU 和 PDU 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。** 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 CPU 和 PDU 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。

【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
    ├── 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）。

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。