计算机组成原理 32 位挑战组

实验报告

匡正非
计算机科学与技术系
kzf15@mails.tsinghua.edu.cn

路橙
计算机科学与技术系
c-lu15@mails.tsinghua.edu.cn

李凌新
计算机科学与技术系
lx-li15@mails.tsinghua.edu.cn

September 18, 2018

Contents

1. 实验总体情况

1.1 实验目标

计算机组成原理挑战组实验的主要目标，是利用 FPGA 实现一个基于 32 位 Mips 指令集的计算机。与 16 位实验相比，除了完成运行监控程序这一基本需求之外，还需要进一步加入对 uCore，U-Boot 等更加高级的软件的支持，因此在工作量和难度上都有着很大的提高。为了让我们的工作能够得到有效推进，我们初步把需要完成的目标划分成了以下几个目标，并分析了每一步所需要完成的工作：

1. 实现所有基本指令的功能仿真。需要完成一个支持 32 位 Mips 基本指令集的 CPU。
2. 运行并通过功能测例。需要将 CPU 写入硬件，同时加入访存功能。
3. 运行监控程序，并通过程序中的测试样例。需要完成串口通信、异常处理。
4. 成功运行 uCore。需要完成 TLB 模块，内存转换，Flash、bootloader 以及中断支持。
5. 运行 U-Boot 程序。需要完成网络通信功能。
6. 提供图片显示功能。需要完成对 VGA 接口的支持。

1.2 完成情况

经过长达两个月的工作，我们虽然没能完成上述的所有目标，但基本上实现了其中的大多数。

我们在 25MHz 的 CPU 频率下成功地运行了以此成功运行了测试样例、监控程序以及 uCore 操作系统，完成的指令条数一共 50 余条，其中包括基本数逻运算、访存、CP0 异常中断处理、TLB 读写以及一些高级运算（例如除法）的指令。

此外，我们还设计访存结构的时候，接入了 Wishbone 总线结构进来，并自己设计了一套内存访问控制系统，能够根据地址支持两种访存方式：通过 Wishbone 访问（主要用于外设）和直接访问（主要针对内存），即保证了效率，又提高了系统稳定性。

对于 Mips 当中的异常中断处理部分，我们提供了对于 trap 中断、syscall 异常、eret 异常、tlb 异常以及时钟、硬件中断的支持。其中，监控程序需要使用 syscall 异常和 eret 异常，而 ucore 除此之外，还需要拥有对 tlb 异常和中断触发的处理。trap 中断在 U-Boot 中才有所使用，我们虽然没有运行出 U-Boot 程序，但却依然完成了这一套指令。

下面提供的是我们的实验平台在运行测试样例、监控程序和 uCore 时候的图片：

Figure 1: 监控程序运行图

Figure 2: uCore 运行图

1.3 分工情况

本组的分工情况大致如下：

组员	工作内容
匡正非	CPU 流水线结构、异常处理、TLB、调试
路橙	访存、Wishbone 总线、串口通信、编译软件
李凌新	CPU 流水线结构、串口通信、编译软件、调试

Table 1: 成员分工

2. 结构设计与实现细节

下面的这一张图描述了我们设计的总体结构：

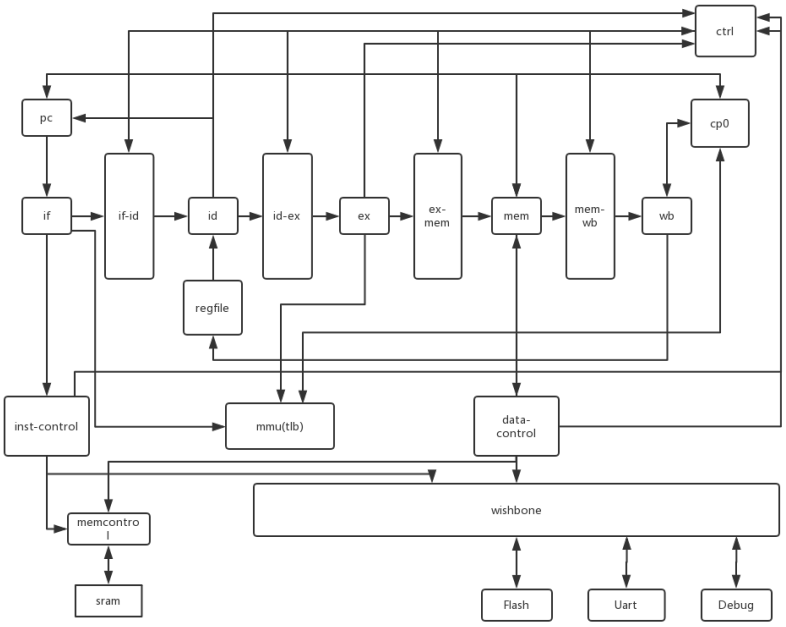

Figure 3: 总体结构设计图

图中的结构可以从上至下分为三个部分：第一部分为 CPU 部分，负责处理指令，主要包含的模块为 if-wb 五级流水线、cp0 协处理器、ctrl 等控制模块；第二部分为总线（访

存）部分，负责 CPU 和各外部硬件之间的数据交互，主要包含的模块有 mem-controller（按照指令和数据分为两个）、mmu、wishbone 以及 sram-controller；第三部分为外设部分，用于处理和各个外部硬件的接口，主要包含 flash、uart 等部件的控制器（controller）。下面就一一介绍这些部分内部的具体实现细节。

2.1 CPU 部分

2.1.1 流水线

在这一阶段中，我们主要参考了《自己动手写 CPU》一书的相关内容，按照 Mips 所提供的标准，一条指令所经过的流水线一共包含五级：取指（if）、译指（id）、运算（ex）、访存（mem）以及写回（wb）。在各级之间通过寄存器来保存中间的处理结果和控制信号。除了这五级结构以外，流水线部分还包含了 ctrl 控制模块（用于暂停流水线，解决冲突问题）、regfile 寄存器堆（保存 Mips 标准中的 32 个寄存器）、以及一些用于特殊指令的子模块（div、hilo 等）。

在流水线中，比较需要关心的是如何解决冲突问题。对于数据冲突，我们主要采用了数据旁路技术，所有的运算结果都回传给了 id 模块（这和课程中所述的略有不同）；对于控制冲突，我们将所有跳转相关的判定都放在了 id 模块完成，程序只需要插入一条延迟槽即可。对于结构冲突，我们把问题主要放在总线部分解决，由总线控制 ctrl 对流水线进行暂停。

2.1.2 CP0 与异常中断处理

虽然在 Mips 标准中关于异常中断的描述十分丰富，但是由于种种限制，我们仅仅完成了其中的一小部分。

在完整的 CP0 中应当包含一共 32 个寄存器，我们最终只使用了其中的 15 个，它们分别是：处理 TLB 异常的 9 个寄存器（Index、Random、EntryLo0/1、EntryHi、Context、PageMask、Wired、BadVAddr、EntryHi），处理时钟中断的 2 个寄存器（Count、Compare）以及处理所有异常都所需的 4 个寄存器（Status、Cause、EPC、PRId）。这些寄存器中有的还会继续按位细分功能，这里就不再详细说了。

为了达到 Mips 所要求的精确异常，所有由指令触发的异常都应当在 mem 模块中进行处理。当 CP0 收到了一条来自 mem 的异常信息时，它会根据异常的类别修改自身状态（包括 Status、Epc 寄存器）。然后，CP0 会异常信息通知给 ctrl 模块，后者则根据这些信息计算出应当跳转到的异常处理程序入口地址，然后产生一个 flush 信号，冲刷掉未处理完的所有指令，以及进行跳转。

除了对异常的处理之外，CP0 还需要支持诸如 mtc0、mfc0 的读写指令，这些操作和正常的寄存器读写相差不大，其中写都是在 wb 模块中才完成的。正因如此，CP0 同样面临着数据冲突的问题，因此同样需要添加一份数据的旁路。

2.2 总线（访存）部分

我们的总线采用了与三总线结构类似的结构，只是 I/O 总线上只有一个从设备。

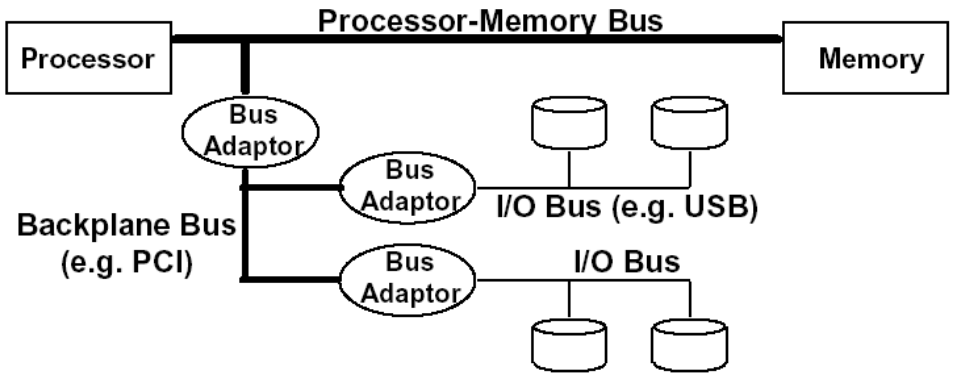

Figure 4: 三总线结构示意图

所有的访存操作均在 CPU 的 mem 阶段进行。当需要访存时，将所需数据的地址、读写信号等一系列数据信息首先传到总线主设备（即 iwishbone 或 dwishbone），总线主设备通过 MMU 将虚拟地址翻译为物理地址，在此过程中查询和维护页表。之后，总线主设备判断物理地址是否为内存，若是，则直接通过 mem_controller 与 sram_controller 进行访存；否则，走 wishbone 总线访问外设。

整体的流程图如下：

2.2.1 直接访存

直接访存通过 mem_controller 和 sram_controller 进行。

当存在结构冲突时，优先进行数据的读写，再进行指令的读写，通过暂停流水线控制整个流水线的稳定。通常情况下，访存只需要占用一个周期。

mem_controller 的状态转移图如下：

在初始时，begin 状态强行暂停流水线一个周期，保证数据的稳定。通常情况下，一直进行指令的读取，每个周期读取一条指令并返回。而当存在数据读写时，第一个周期进行数据读写，在第二个周期进入 data_finish 状态，读取指令。

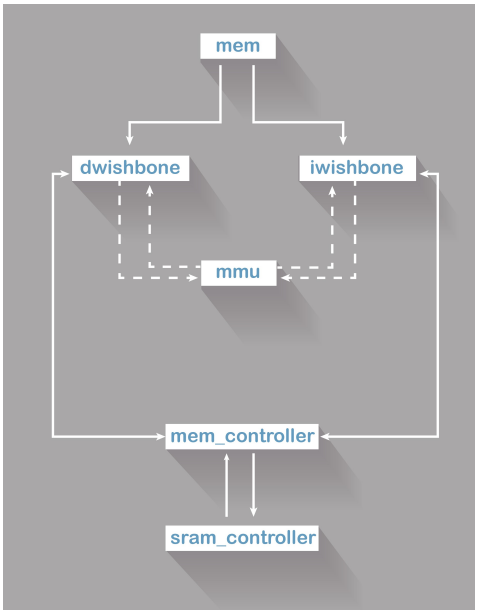

Figure 5: 访存结构示意图

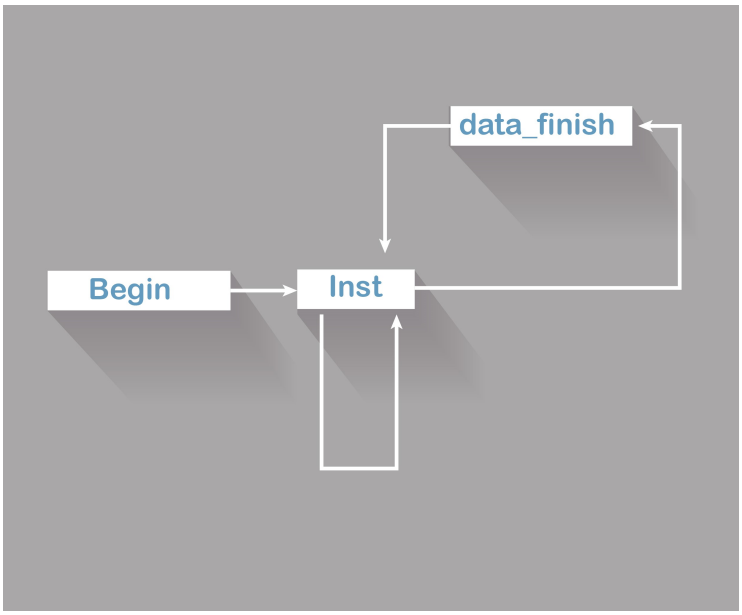

Figure 6: mem_controller 状态图

sram_controller 的状态转移图如下：

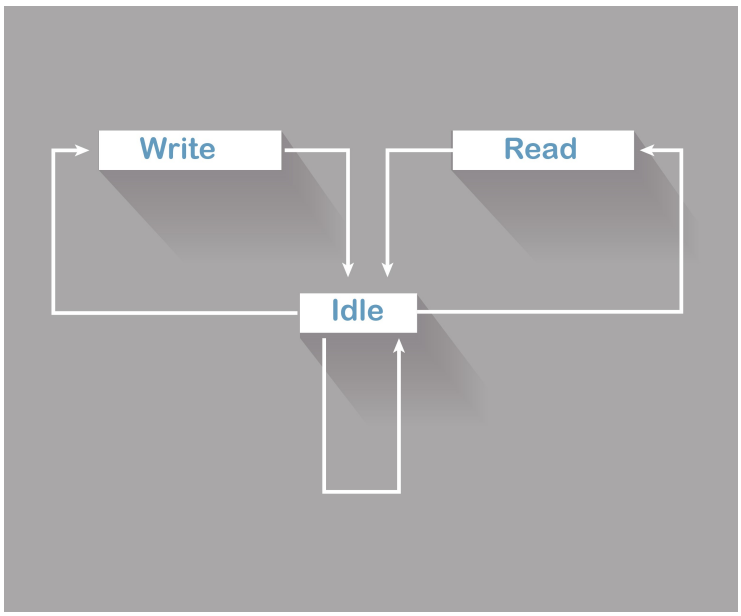

Figure 7: sram_controller 状态图

sram 的时钟周期为 cpu 时钟周期的 2 倍。当需要读取数据时，前半个 cpu 时钟周期为数据准备阶段，后半个 cpu 时钟周期访问 sram。而当需要写数据时，前半个 cpu 周期为数据准备阶段，后半个 cpu 周期写数据，而数据保持时间为下一个 cpu 时钟周期到 cpu 将数据传递到 sram 时。由于数据通路的延迟以及 sram 需要的数据保持时间很少，因此我们的访存可以在一个周期内结束。

2.2.2 Wishbone 总线仲裁

我们采用 wishbone 总线结构实现外设的管理，总线仲裁使用了 opencores 中标准的 ip 核。

总线的主设备为 dwishbone 和 iwishbone，分别管理数据读写与指令读写。而从设备有 flash 和串口。尽管 wishbone 牺牲了一定的效率，但大大提高了我们项目的可扩展性，可以较轻易地增加外设；其次，总线下面的外设可以有不同的时钟周期，利用总线可以方便地进行对不同时钟周期外设的同步。此外，外设访问的结构冲突也因为 stall 与 ack 以及 wishbone_conmax 的仲裁而很好地解决了。

在实现总线结构的过程中，我们开始时将 sram 也放在了总线的从设备中，但这会导致访存需要四到五个 cpu 时钟周期，导致 cpu 增加了很多的气泡。为此，一个简单的

策略是，减小 wishbone 的时钟周期，由于 sram 可以支持 100Hz 的访存，因此若 cpu 为 25Hz，wishbone 最快可以达到 100Hz。为此，需要使 wishbone 支持 4 个周期的访存。具体分析如下：

1. mem 将数据读写信号传递到总线主设备，需要一个周期
2. 总线主设备将信号传递到 wishbone_conmax，需要一个周期
3. wishbone_conmax 进行仲裁，传递到 sram 需要两个周期
4. sram 进行数据的读写并返回 ack
5. 主设备收到 ack，结束流水线的暂停

为了使总线可以做到四周期访存结束，调整如下：

1. mem 将数据读写信号传递到总线主设备，需要一个周期
2. 总线主设备在检测到 mem 来的信号后，在进行状态转移的同时将信号传递到总线，这样在进入 busy 状态的同时总线也可以收到信号
3. wishbone_conmax 进行仲裁，传递到 sram 需要两个周期
4. sram 马上返回 ack，并进行访存
5. 主设备在一个周期后收到 ack。由于 sram 访存较快，主设备进行组合逻辑读取数据时，sram 已将正确的数据读出

经过如上的调整，将 wishbone 时钟频率调整到 cpu 时钟频率的四倍时，理论上可以保证 cpu 在一个周期内访存完毕。但在实际操作的过程中，我们发现由于二者并不是同一个时钟信号，存在跨时钟域的控制信号 stall_req（总线主设备与 cpu 流水线之间交互，判断是否暂停流水线的信号）。由于这个信号在 cpu 和总线主设备中都有可能被更改，同步较为困难，且在进行实验时发现，cpu 会在将数据更改到一半的时候收到 stall_req 的变化，这导致流水线数据传递中某个阶段的数据传递低 16 位没有更改，但高 16 位却被更改。

对于这个 bug，我们尝试更改了 stall_req，保证它不在 cpu 时钟周期的上升沿变化。但由于这个控制需要知晓总线时钟与 cpu 时钟是否同时上升沿，这一检测很难在 verilog 中做到。我们尝试对总线时钟周期进行计数，发现计数信号的初始化或更新，总需要某个跨时钟域的信号。此外，我们还尝试使用 ip 核对时钟进行倍频，控制时钟的相位相同，但依然会有同样的问题。

最终，这一 bug 并没有被解决。我们在这一 bug 上消耗了将近两周的时间，最终回退重来，重新设计了访存的流程。从这一个调 bug 的过程，我们深刻地理解到，硬件的实现对时序的考虑需要十分谨慎，逻辑上可行的通路在考虑时序后很可能有问题。

2.2.3 内存管理 (mmu)

内存管理模块是完成从监控程序到 uCore 至关重要的一步。结合 Mips 所提供的标注内存映射，我们将所有需要使用的内存地址划分成了下表：

内存空间	用途
0x00000000<->0x7fffffff	用户虚拟内存空间（kuseg）
0x80000000<->0x807fffff	直接映射到物理内存
0xbe000000<->0xbeffffff	Flash
0xbfc00000<->0xbfc003ff	BootLoader
0xbfd003f8<->0xbfd003ff	Uart
0xbfd0f000<->0xbfd0f01f	Debug 输出

为了方便处理，我们将所有的地址映射和虚实转换操作都放在了同一个 mmu 模块之中，其内部结构如下图所示：

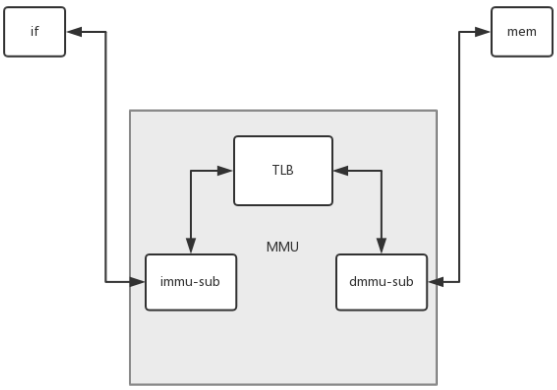

Figure 8: mem_controller 状态图

可以看到，mmu 模块中又一共被被分为了两个部分：mmu-sub 模块和 tlb 模块。前者主要负责地址映射，后者负责虚实地址转换。每次当 mmu 读入一个地址时，如果当

前读入的地址是物理地址，则会将其转换为 wishbone 总线采用的地址；如果当前读入的地址是虚拟地址（此时一定会映射到内存上去），则现在 TLB 中完成虚实地址转换，然后再直接返回。至于在 TLB 中的详细操作，在下一章中会更加详细地说明。

2.2.4 TLB

在一个完整的操作系统中，需要通过页表来完成对程序的虚拟地址映射，页表往往通过操作系统自身实现，而 TLB 则可以看成是页表在硬件中来的一个缓存机制。为了实现 TLB，一方面，需要从 FPGA 中开辟一定的存储空间（通过寄存器或者缓存）来保存相应的表项，另一方面，则需要加入和其相关的异常处理和指令。在我们实现的 TLB 中，一共有 32 个表项（共 2304 位），采用全链接方式进行查表操作。由于对于内存的访问只会存在于 If 和 Mem 阶段，因此当 TLB 接收到一个查询请求时，如果发生丢失，则会直接向两个流水段返回异常信息。当 TLBMiss 异常发生后，具体的重填操作都是通过软件来完成的，其中所需要的大部分硬件操作都属于 Mips 标准，这里就不再详细说明了。另外，按照 Mips 标准，TLB 表项中不应当同时出现两个相同的表项，否则会产生严重的异常，但在 uCore 中却可能会出现这样的现象，因此为了运行，还需要对 TLB 做出相应的一些修改（例如修改表项时进行判定等等）。

2.3 外设部分

2.3.1 串行接口

由于我们采用了 wishibone 总线结构，串口实现可以很容易使用现有的 ip 核。我们使用 UART 16550 core 最为串口控制器。其内置一个 16bytes 的 FIFO，对连续输入的大量数据进行缓存，使得不会因为传输速率瞬间过高而发生丢包。UART 16550 core 内置多个寄存器，可通过修改其中各值来调整串口控制器参数，如波特率，校验位，数据位数。也就是说，可以简单地通过软件上的改变来动态调整串口参数。对于串口信号的处理直接写入 FPGA 中，使得串口数据不用与内存共享数据线，不用对其进行仲裁。

2.3.2 Flash

Flash 只需要支持读操作。读步骤如下：

1. 将 CE 置为 0
2. 将 OE 置为 0
3. 读取到相应的数据

由于 Flash 响应较慢，因此我们在每次读取 Flash 时都留出了 3 个时钟周期的时长。另外，在实际硬件中的 Flash 一次只能读出 16 位数据，因此在 32 位的机器中，需要进行两次读取才能获得一个完整的数据单位。不过，这一的问题是通过 BootLoader 的软件来解决的。

2.3.3 BootLoader(ROM)

Rom 是一块写在 FPGA 中的内存，直接通过寄存器保存固定的值，因此实现起来基本没有难度。其中保存了 BootLoader 的指令，使用的是 Ucore 中自带的 BootLoader。启动时，指令地址设置为 ROM 的起始地址，运行 BootLoader，其作用是将写入 flash 的 elf 文件读入内存中，并将指令地址跳到 elf 文件中指定的起始地址。有了 BootLoader 之后，将监控程序或 Ucore 的编译出来的 elf 直接写入 flash，就可以直接运行。另外，考虑到 BootLoader 的指令长度往往不长，对效率的整体影响不大，因此将其直接接在了 wishbone 下方，使得处理更加方便。

2.3.4 Debug

尽管没有实现 VGA，但我们组实现了一个强大的 debug 工具——二进制优先编码指示灯调试法。我们对每个流水线过程中的数据均采用了二进制编码，既可以看流水线各个阶段的数据，也可以看各个寄存器的值。

具体地，我们通过优先编码的方法确定需要查看的某个流水线阶段，再将该阶段需要输出的数据输出至 debugData 中。通过拨动开关输入编码，再将该编码对应的 debug 信息输出到 LED 灯进行查看。我们的编码方式为：

部件名	编码
1xxxxxxxxx	dwishbone 的 debug 信息
01xxxxxxxx	uart 的 debug 信息
0011xxxxxx	cp0 的 debug 信息
0010xxxxxx	led 的 debug 信息
0010xxxxxx	led 的 debug 信息
0001xxxxxx	模拟 ram 的 debug 信息
0000100000	if 阶段的 debug 信息
0000100001	id 阶段的 debug 信息
0000100010	ex 阶段的 debug 信息
0000100011	mem 阶段的 debug 信息

部件名	编码
0000100100	wb 阶段的 debug 信息
0000100101	ctrl 阶段的 debug 信息
00000xxxxx	寄存器的 debug 信息（对应 32 个寄存器）

由于我们实现了一个庞大的 debug 系统，可以很轻松地将很多重要的输出信息在一次编译后输出，从而大大减少了重复编译的次数，通过观察流水线阶段的信息和寄存器的信息，用 click 的时钟输入可以极快地定位 bug 所在的位置。

3. 仿真与实验

3.1 iverilog 仿真

在 cpu 开发阶段，我们采用 iverilog+gtkwave 进行功能仿真，参考《自己动手写 cpu》中的简单样例，对比波形图，确保 cpu 逻辑上有一定正确性。在开发的过程中，我们对

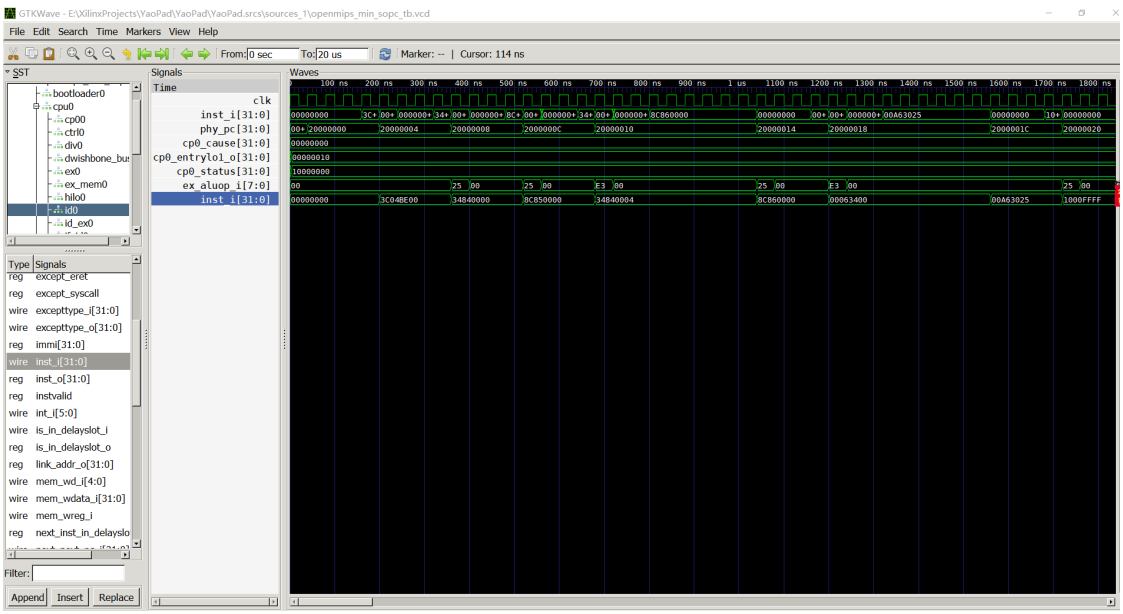

Figure 9: 仿真波形图

每一阶段都进行了功能仿真，以保证在未上板子之前每个模块的逻辑都是正确的，这极大地保障了我们后期的开发进度。而在后期的调试阶段，我们也采取了先仿真后上板子

的方式进行开发，这也减小了我们重复编译的次数。

3.2 硬件信号调试

在写入板上之后，与功能仿真结果不同的情况时有发生。我们在开发进度较早时，就构建了完整的 debug 信号系统，通过开发板上 32 位开关，控制输出各种信号：寄存器值、流水线各阶段关键信号、cpu 与外部交互信号等等。此 debug 信号系统为我们之后的开发带来了很大的方便，加快了 debug 速度。

关于该 debug 系统的具体设计，详见2.3.4。

3.3 功能测例

本次开发过程中，我们有了一套详细的针对每一个指令的功能测例。该测例是测试各个指令是否正常运行以及调试访存是否实现正确的重要测例。我们在开发过程中，通过运行功能测例，依次发现了 syscall、LB、SB 等一系列指令的 bug，这也为我们能跑出 ucore 奠定了基础。

3.4 监控程序

监控程序是我们调试串口的重要测例。在对串口的开发过程中，我们围绕着串口精灵 + 监控程序的一套过程，高效率地对串口程序进行调试。而当我们最终调出串口后，发现监控程序 term.py 里有些地方的实现并不适合我们的串口，因此我们又对 term.py 进行了一些修改（未修改关键指令，只修改了部分阻塞条件），最终可以运行监控程序支持的所有指令。

此外，我们通过对监控程序的测例进行运行，测得我们 cpu 的实际频率为 23.8MHz。

3.5 ucore

ucore 是我们完成的最终成果。由于 ucore 需要的编译工具链较复杂，我们在张宇翔助教的指导下完成了 ucore 的编译，并一次性进入了 debug mode。由于我们事先对 tlb 进行了很多的自己编写的单个功能测例，且除了 tlb 以外的其余部分已在前述阶段调通，因此对 ucore 的调试主要围绕在 tlb 中无法使用单个功能测例测试的部分。

由于 ucore 为 c 语言编写，我们组的同学对 c 语言较为熟悉，调试起来十分顺手，加上我们前期工作奠定的基础较好，我们在一天之内调出了 11 个 bug，跑出了 shell（但只能运行一次）。之后由于马上要进行考试，我们暂停了开发工作。在考试周结束后，我们又用一上午发现了 bug 所在（tlb 的一个 bug），调试后便可以稳定地运行 ucore。

4. 实验中遇到的问题与不足

在整个大作业过程中，我们遇到的最大的问题就是在 2.2.2 中详细说明的有关跨时钟域信号的 bug，最终也没有解决，只是使用其他方法避开这个雷区。我们在这个 bug 上花费了近两周时间，期间常常怀疑人生，多次熬夜而毫无进展，最终和助教讨论后，有了新的思路，巧妙地避开了跨时钟域的信号，也使性能有较大提升。从这个过程，我们深刻了理解了硬件实现中时序的考虑十分重要，以及请教助教能使开发进度大大加快，感谢助教不辞辛苦的帮助！

由于投入精力并不如联合组那么多，我们还有很多可以实现的功能未能实现：VGA，Cache，网线，Uboot... 但我们相信，再多加时日，我们定能完成更加令人激动，骄傲的成果！

最后，再次感谢老师，助教，同学们的指导和帮助！

5. 参考文献

1. 雷思睿，自己动手写 CPU[M]. 电子工业出版社，2014.
2. Dominic Sweetman，See MIPS Run. 2006.
3. 刘卫东，李山山，宋佳兴，计算机硬件系统实验教程. 清华大学出版社，2013.
4. David A.Patterson, John L.Hennessy，计算机组成与设计. 机械工业出版社，2013.