计算机系统综合实验最终报告
（CPU 部分）

郭昉泽 马强 王倩

1. 设计概述

CPU 内部采用五级流水线，分别是取指（IF）、译码（ID）、仿存（MEM）、执行（EXE）和写回（WB）五个阶段，各阶段之间设置有段间寄存器，当时钟上升沿到来时将上一阶段的结果送到下一阶段。在寄存器组中实现了 32 个通用寄存器、HILO 寄存器以及 11 个 CP0 寄存器。CTRL 单元负责流水线的暂停和清除。在 MMU 单元的内部嵌入了 TLB，遵照 MIPS 标准根据地址所属区段进行映射，并且负责一部分内存管理。

CPU 对外只有数据总线和指令总线。总线设计采用了 Wishbone 总线协议，使用 Wishbone_Conmax 进行总线仲裁，从而决定外设对于总线使用权的要求。这一部分将在外设介绍部分进行详细的说明。

外设方面，总线上挂载了 FLASH、RAM、UART 和 VGA 四种外设，所有外设的控制器模块全部按 wishbone 总线规范进行编写。控制器的功能简要来说就是根据总线给出的信号对相应外设进行数据读写，各个控制器的细节在后面章节进行详细说明。

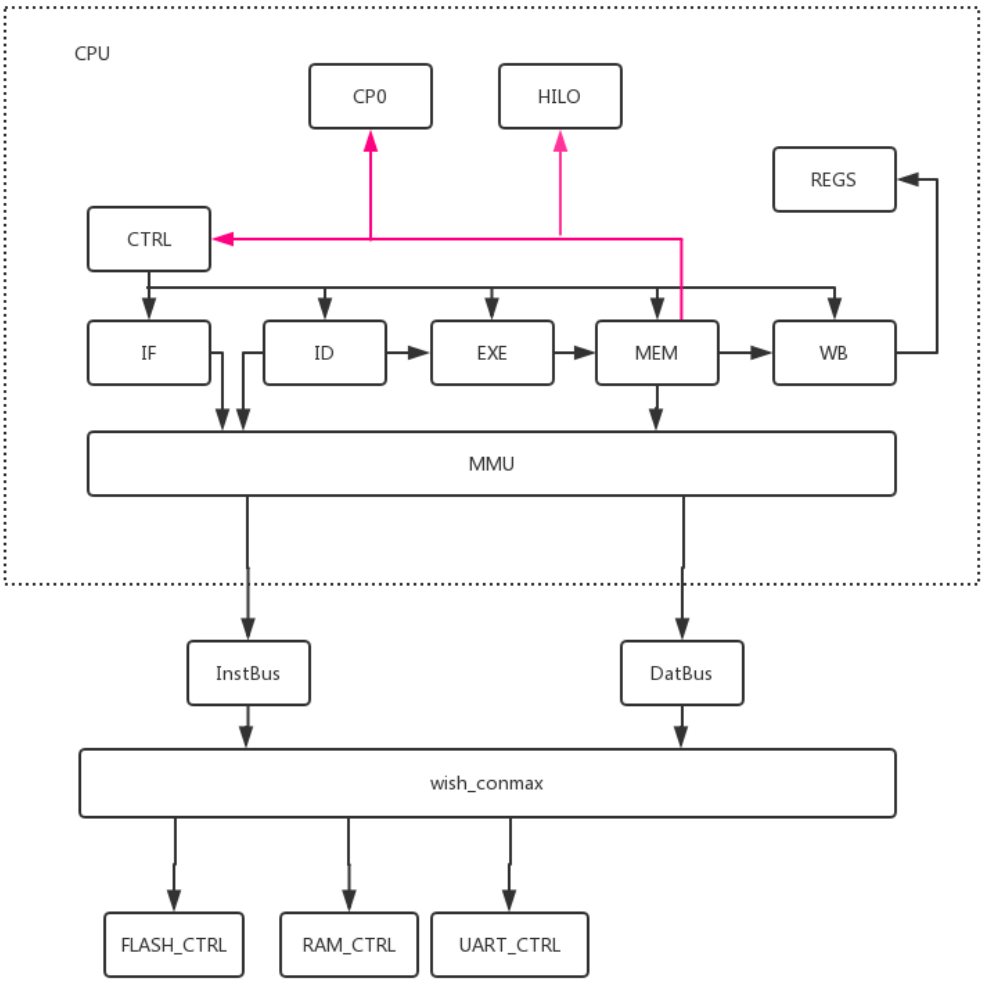


Figure 1 ： 整体设计

2. CPU

2.1 Wishbone 总线协议

2.1.1 概述

在总线的实现上，我们选择了 Wishbone（下面简称 WB）总线协议，这是一个开源的总线协议规范，支持包括点对点、数据流、共享总线、交叉互联等多种链接方式，在 NEHCPU 的设计中，我们使用了交叉互联的方式。WB 总线规范接口如下图所示。

WB 各接口含义简要说明如下：

RST_I	复位信号
CLK_I	时钟信号
ADR_O/ADR_I	地址总线，由主设备传给从设备
DAT_O/DAT_I	数据总线，数据可在主从设备间传输
WE_O/WE_I	写使能
SEL_O/SEL_I	数据总线选择信号
STB_O/STB_I，CYC_O/CYC_I	选通，只有二者均为高电平才能读取数据
ACK_O/ACK_I	确认信号，传输成功时发出
Other	其余为本次实验未涉及信号

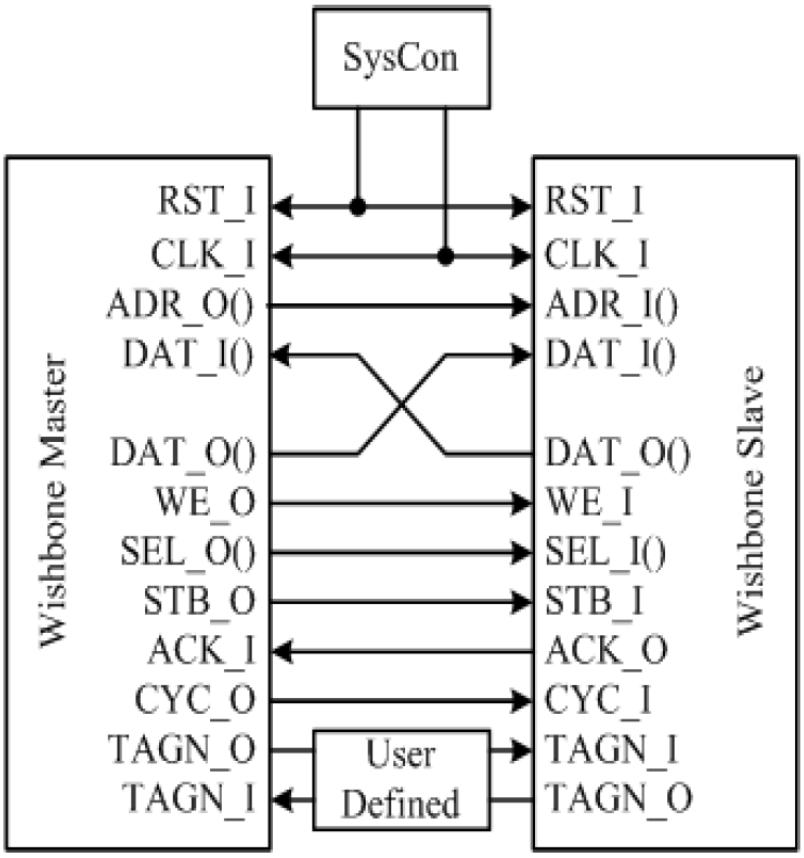


Figure 2：Wishbone interface

2.1.2 WB_Conmax

此处我们采用了开源的 WB_Conmax 模块来管理总线设备请求，该模块支持 8 个主设备和 16 个从设备的互联，允许多对主从设备同时通信。我们的 CPU 采用了默认的设备地址管理方式，即地址高四位作为从设备的索引号，例如，0 号从设备的寻址空间为 0x00000000~0x0FFFFFFF，1 号从设备的寻址空间为 0x10000000~0x1FFFFFFF。NEHCPU 中主从设备地址空间做如下分配：

索引	从设备名称	寻址空间
0	SRAM	0x00000000~0x0FFFFFFF
1	FLASH	0x10000000~0x1FFFFFFF
2	片上 ROM	0x20000000~0x2FFFFFFF
3	UART	0x30000000~0x3FFFFFFF
4	VGA	0x50000000~0x5FFFFFFF

2.1.3 总线状态机

WB 总线通过设置一个状态机解决了一个周期内不能保证仿存结束的问题，状态机及其转换关系如下图所示：

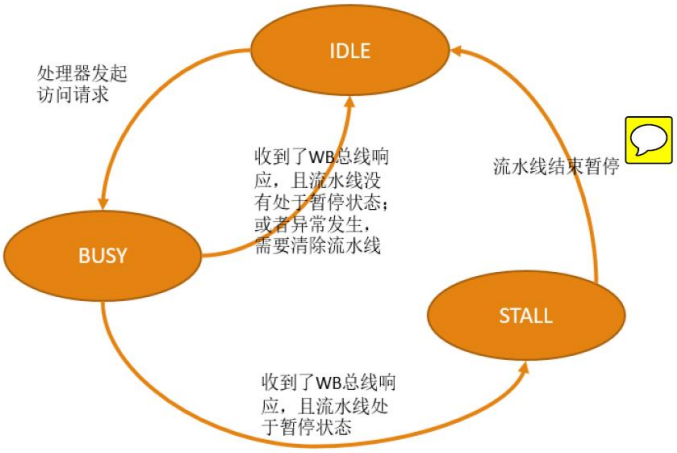


Figure 3：总线状态机转移图

2.2 冲突处理

2.2.1 结构冲突

结构冲突是流水线结构下取指和读取数据时同时访问 RAM 造成的冲突。在 NEHCPU 中，由于我们采用了 WB 互联矩阵来进行总线使用权的分配，指令总线和数据总线的仿存冲突已被涵盖在使用权分配这一大问题下，因而自然得到解决。具体做法是需要同时访问 RAM 时由 WB 互联矩阵来决定 RAM 使用的次序，缺点是可能会对流水线效率造成一定程度的影响。

2.2.2 数据冲突

简单来讲，数据冲突是指一条指令需要用到前面某条指令的执行结果，但在某些时候却得不到所引发的冲突。NEHCPU 采用旁路技术和插气泡来解决以下四种数据冲突的情况。

写后读冲突。例如在执行

$$\text{ori } \$1, \$0, 0x1000$$
$$\text{ori } \$2, \$1, 0x2000$$

第二条指令要用到 1 号寄存器的值，然而在译码阶段第一条指令的结果还未写回寄存器堆，第二条指令的取到的 1 号寄存器的值是旧的，从而导致结果错误。我们采用旁路技术，将第一条指令的执行结果在下一个时钟上升沿到来时直接送到 EXE 模块用以替代第二条指令译码阶段取到的值，从而解决了这种冲突。

由 HI/LO 寄存器引发的冲突。考虑 mfhi 和 mflo 两条指令，他们会在 EXE 阶段读取 HI/LO 寄存器的值，而在此之前正处于 MEM 或 WB 阶段的指令有可能改写 Hi/LO 寄存器的值，由此产生错误，解决办法仍是旁路技术，将 MEM 阶段有关 HI/LO 寄存器操作的信息送至 EXE 段。

由 CP0 寄存器引发的冲突。考虑 mfc0 指令，类似 HI/LO 寄存器引发的冲突，这一指令也需要在 EXE 阶段获取 CP0 寄存器的值，解决办法与前一冲突完全类似。

与 Load 相关的数据冲突。考虑如下代码片段

$$\text{lw } \$1, 0x0(\$0)$$
$$\text{beq } \$1, \$2, \text{label}$$

lw 指令会在 MEM 阶段得到$1 寄存器的真实值，然而 beq 在译码阶段就要得到$1 寄存器的值，至少需要相差两个时钟周期，然而这两条指令在时空上只相差一个时钟周期，单纯的旁路技术无法解决这一冲突。因此，解决方法是在 ID 段检测这种冲突，一旦发生就将流水线暂停，等仿存操作完成后，再利用旁路将信息送至 EXE 段。

2.2.3 控制冲突

控制冲突使用分支指令和跳转指令引发的冲突，NEHCPU 的设计中引入了延时槽机制以减少流水线效率的损失。具体方法是在 ID 阶段对是否跳转进行检测，若发生跳转，则将新的 PC 地址送至 IF 段的 PC 寄存器中，在下一时钟周期进行跳转。这样设置的结果是跳转指令之后的第一条指令必须得执行，之后再从目标地址处开始执行，这一现象也被称作延时槽。延时槽的问题可通过软件层面调度的方式来解决。

2.3 异常处理

实现了 Interrupt、TLBMod、TLBL、TLBS、ADEL、ADES、Syscall、RI 这些异常的支持。这些异常也是成功运行 ucore 所需支持的异常类型。由于异常处理的顺序需要与指令执行的顺序一致，我们采用的策略是发现异常时暂不处理，等到 MEM 阶段统一处理的方法。异常发生后需要填写某些 CP0 寄存器的某些字段，如 status、cause 和 epc 等，此外当 TLB 相关异常发生时，还要填写 entrylo、entryhi 和 badvaddr 等寄存器，且应事先设置好 ebase 寄存器的值，并据此执行跳转操作，以进入异常处理代码段，各异常类型具体实现如下：

- Interrupt。中断异常，在 NEHCPU 中共有两种中断异常，时钟中断和串口中断，时钟中断需要 count 和 compare 寄存器的配合，串口中断则是由外设引起。需要注意的是中断号存储在 status 寄存器相应字段当中，ucore 有其特定的对于中断号的规定，软硬件必须协同才能配合工作。
- TLBMod。TLB 修改异常。当要写入的 TLB 表项的 D 字段为 0 时，表示该表项不可写入，若此时写入该表项将导致 TLBMod 异常。
- TLBL/TLBS。这两种异常发生条件类似，都是在查 TLB 表未命中的情况下或是命中但 V 字段为 0 时触发的，将仿存操作是否是写操作的信息传递到 MMU 模块以确定当发生 TLB Miss 现象时应触发 TLBL 还是 TLBS。
- ADEL/ADES。地址不对齐异常。具体来讲 lw、lhu 和 sb 这三条指令对于要访问的内存地址有对齐要求，当这一要求没有得到满足时就会发生地址不对齐异常。根据当前是读操作还是写操作触发 ADEL 或是 ADES，此外还需要填写 badvaddr 寄存器。
- Syscall。系统调用。当触发该指令时触发 syscall 异常。
- RI。指令无效异常，译码阶段解析到的指令不在所实现的指令集中触发该异常。

2.4 内存管理

2.4.1 TLB

NEHCPU 实现了具有 16 个 entry 的 TLB 表，支持写入、查询以及触发相关异常。详见模块描述中的 MMU 模块。

2.4.2 地址映射

按照 MIPS 规范将整个地址空间划分为 Useg、Kseg0、Kseg1、Kseg2 和 Kseg3 区段。其中 Kseg0 和 Kseg1 不使用 TLB 而是直接将高三位置 0 进行静态映射，其余区段采用查 TLB 表的方式进行。由于采用了 WB 互联矩阵的方式，因此在管理互联矩阵信号的 wishbone_bus 模块中进

行了第二次地址映射，根据要访问的外设的不同，修改高四位的值后再传递给 WB_Conmax，从而将数据发送到正确的外设处。

2.5 模块详述

2.5.1 PC

这一模块的源代码位于 pc_reg.v 中，维护当前执行的指令的地址。当复位信号到来时将初始地址设置为 0xBFC00000，这是片上 bootloader 的起始地址。复位后将执行 bootloader，把 ucore 从 flash 复制到 ram 中，并跳转至操作系统开始执行。一般情况下 pc 会每个周期完成加四操作，而特殊情况的处理如下：

- 异常处理：此时需要清除流水线，并跳转至异常处理的代码段，异常处理可以看做特殊的跳转指令。
- 分支跳转指令。当遇到分支跳转且满足分支条件时，pc 被改为跳转目标地址的值。
- 流水线暂停。当流水线发生某种冲突或者仿存时需要暂停流水线，此时 pc 维持不变而不是自加 4.

2.5.2 IF/ID

源代码位于 if_id.v 中。实现的功能是 IF 模块与 ID 模块的过渡，是两模块间的流水段寄存器。这一模块将从指令总线得到的地址、指令，以及 MMU 阶段传来的是否发生 TLBL 异常等信息存入寄存器堆，并在时钟上升沿到来时传递到 ID 阶段。此外由于要实现异常处理时流水线清除以及冲突发生时流水线暂停，当这两种请求任意一个到来时应将该流水段寄存器堆全部置 0。

2.5.3 ID

该模块位于 id.v 中，具备如下三个功能

- 对当前指令进行解码。
- 处理某些异常信息。这一阶段可能发生的异常类型有指令无效异常、syscall 以及上一阶段传来的有关是否发生 TLBL 的信息，这些异常信息要向下传递。
- 检测 RAW 并做相关处理。根据 EXE、MEM 段传来的信息，检测 RAW 冲突的发生，若冲突，则应用送来的新值代替解码后读取到的值。

2.5.4 ID/EXE

该模块位于 id_exe.v 文件中。实现的功能仍是信息传递，传递的信息包括解码时获得的指令类别、寄存器及立即数的值、延时槽相关信息和异常信息等，还应包括异常发生时或冲突发生时排空流水线或暂停流水线的操作。

2.5.5 EXE

这一模块的源代码位于 ex.v 中，其功能主要是根据指令类别进行相应的运算。具体的有：

- 逻辑运算指令。
- 算术运算指令。其中，乘法使用了 verilog 内置的乘法运算符实现。
- 位移指令。
- 分支跳转指令。ID 已完成跳转工作，EXE 段所要做的只是将延时槽等信息传递至下一阶段供异常处理时使用。
- 特殊寄存器操作指令。指对 HI/LO 寄存器以及 CP0 寄存器的读写操作。
- 仿存指令。主要是计算仿存的地址，可能会出现地址不对齐异常，这些信息直接传递至下一阶段。
- 其他指令。Syscall、eret 指令只需将相应信息传递至 MEM 进行处理即可，cache 指令由于未实现 cache 所以视作空指令，tlbwi 指令只需在 MEM 段传递给 MMU 模块处理即可。

2.5.6 EXE/MEM

这一模块的代码位于 ex_mem.v 中，实现的功能是 EXE 和 MEM 段之间信息的传递，与之前的段间寄存器堆没什么大的不同。不再赘述。

2.5.7 MEM

这一模块的源代码位于 mem.v 中，MEM 段的主要功能如下：

- 进行仿存操作。将 EXE 阶段计算出来的地址传递给 MMU 模块，MMU 模块完成地址映射后传递给响应的外设，进行读取或写入操作，写入时需要提供写入的数据。读取完成后需要将读取得到的数据送至 WB 段。
- 处理异常。从 ID、EXE 阶段会传递来异常，这一阶段本身也可能会出现 TLBMod、TLBL 和 TLBS 等异常，对于这些异常，需要将相应信息送到 CP0 以及 CTRL 模块。值得注意的一点是，异常的处理是有优先级之分的，先发生的异常应先被处理。
- 旁路。需要将发生 RAW 或 load 冲突时需要将信息前推至 ID、EXE 等阶段。
- 写 TLB 表。如果当前指令是 tlbwi 则需要写 TLB 表。这一操作实际在 MMU 模块中完成，为此需要构造 TLB 表项，具体来讲，要从 CP0 中取得某些寄存器的值，可能会产生新的数据冲突。

2.5.8 MEM/WB

段间寄存器堆模块，功能为 MEM 和 WB 两阶段间信息传递，不再赘述。

2.5.9 MMU

这一模块的源代码位于 mmu.v 中，其重要功能有以下三种：

- 维护 TLB 表项。TLB 表项的实现类似于通用寄存器堆，结构如下图所示，执行写操作时需要从 MEM 段拿到 index 寄存器的值以及要写入的数据，对表项进行写操作。
- 实现地址映射。单独实现了一个模块 vir2phy.v 来完成地址的映射。首先判断是否属于 KSeg0 或 Kseg1，若是则直接将高三位置 0 输出；对于其他区段的地址要使用前十九位作为索引与 TLB 表项的高 19 位做比对，若命中则根据下一位判断是奇数页还是偶数页。经此查到的 FPN 字段作为物理地址的高 20 位。与虚拟地址的低 12 位拼接在一起形成物理地址。
- 检测 TLB 异常。主要是区分 TLBL 发生时应区分是指令还是数据引起的，对于指令引起的，应回传至 IF/ID 段，若是数据引起的，则应回传至 MEM 段处理即可。

2.5.10 CP0

该模块位于 cp0_reg.v 文件中。这一模块实现了 11 个 cp0 寄存器，且支持对他们的读写操作。此外这一模块负责在异常发生时，根据传来的异常信息填充相关寄存器的值。延时槽信息也被传递至该模块，其目的是在设置 EPC 时，若发生在延时槽中则应设置 EPC 为前一条指令的地址。

2.5.11 CTRL

该模块位于 ctrl.v 文件中。这一模块起到对流水线的控制作用，具体来说就是控制取指、译码、仿存阶段的暂停流水线请求，以及异常发生时排空流水线的请求。这一模块也要根据异常类型决定要跳转到的异常处理程序的入口地址。值得注意的是，不同阶段流水线暂停请求也有不同的优先级，仿存阶段暂停优先级最高，译码阶段次之，取指阶段暂停请求优先级最低。

3. 外设

3.1 ROM

ROM的作用是存储bootloader。bootloader中不仅能够将操作系统从FLASH加载到SRAM中，

还可以验证 FLASH 和 SRAM 的时序是否正常。如果不正常会卡死在 bootloader 中。ROM 的物理地址为 0xBFC00000~0xBFC00FFF，转化为虚拟地址即为 0x1FC00000~0X1FC00FFF。ROM 的速度是非常快的，所以在返回数据的时候不需要等待若干个周期，直接依照 wishbone 总线的时序规范返回即可。

3.2 SRAM

SRAM 的物理地址为 0x00000000~0x007FFFFF，由于实际上 SRAM 有两片，所以在 NEHCPU 中将有效地址的最高位（23 位）设置为 Select 位，通过这一位来选择正确的 SRAM。SRAM 的读写需要一定的时间，在 NEHCPU 中共使用了 3 个 50M 时钟周期来等待 SRAM 返回数据。NEHCPU 使用一个小型的状态自动机来实现等待的操作。

3.3 FLASH

Flash 的物理地址为 0xBE000000~0xBEFFFFFF，由于 Flash 的字长为 16bit，所以读出来的数据只有低 16 位有效。Flash 读取十分缓慢，所以在 NEHCPU 中使用了 15 个 50M 时钟周期来等待 Flash 返回结果。

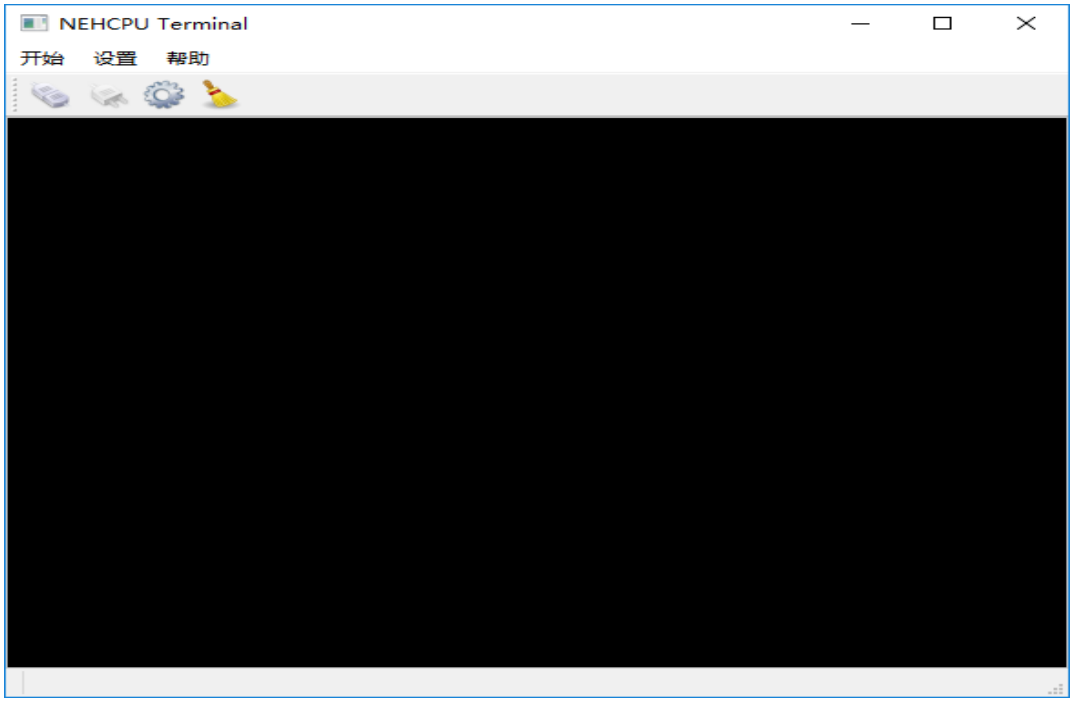
3.4 串口

NEHCPU 使用了 http://www.fpga4fun.com/SerialInterface.html 的开源代码来直接与串口进行交互，并自行实现了一个其与 wishbone 总线交互的接口。为了更方便地与串口进行交互，我们自行开发了一个基于 QT 的串口小工具来代替串口调试助手。这个工具可以很有效地模拟“终端”，且可以跨平台编译使用。

3.5 VGA

对于 VGA 的处理分为两部分。
首先操作系统应该可以通过对某些地址的写操作来向 VGA 输出东西，为此我们在 FPGA 内部实现了一个大小为 0x96000 的“显存”，且实现了一个其与 wishbone 总线交互的接口。
接下来是显示部分。显示的格式为 800*600（75Hz），但是经过我们的测试，如果存储 640000 个像素，会大大拖慢操作系统的运行速度。于是我们这里使用了 400*300 扩展到 800*600 显示的方案。同时为了加速寻址，实际存储的大小为 512*300，且只有 400*300 有效。这样一个异步执行的 VGA 驱动程序就可以不断地扫描显存并输出。
考虑到“终端”会有大量滚屏情况的出现，这里还采用了一个优化，就是在每输出一行后，驱动程序输出 VGA 的首地址会往后移一行。操作系统内部也做一样的处理，就可以实现每输出一行只需要向 VGA 的显存输出一行的数据。这样大幅加快了显示速度。
VGA 显存的物理地址为[0xBA000000,0xBA096000]，而 VGA 首地址的物理地址为 0xBA096000。

附：终端工具简介



由于串口调试工具不能有效地实现“终端”这一功能，我们开发了一个终端工具。使用这个工具可以轻松查看串口的信息，连接指定的串口，接受串口信息并向串口发送信息。

这个工具是基于 QT 开发的，最低的 QT 版本为 QT5.5。使用 QT5.5 以上的版本直接编译运行即可。

源代码已放在压缩包中。