SOC 设计与实现 v1.0

版本	日期	人员	备注
0.1	2013.08.05	赵鸿宇	初始版本
0.2	2013.08.11	赵鸿宇	添加 CACHE 指令
0.3	2013.08.18	赵鸿宇	BUG 修正，添加串口
0.4	2013.09.03	赵鸿宇	完善 VGA 接口 添加板级 IO 接口
0.5	2013.10.15	赵鸿宇	修改串口接口
0.6	2013.11.24	赵鸿宇	添加键盘接口
0.7	2014.03.19	赵鸿宇	添加 SPI 接口（接 SD 卡） 修改 IO 地址分配
0.8	2014.09.25	赵鸿宇	CP0 中断优化 MMU/CACHE 控制方式变更
0.9	2014.11.07	赵鸿宇	更改 CP0.ICR 控制方式 指令集新增 BLEZ/BGTZ
1.0	2015.03.15	赵鸿宇	更改板级 IO 地址以支持多种平台

TODO LIST：

实现指令 CLO & CLZ

实现指令 LL & SC

实现硬件断点

将时钟信号生成器当做 0 号外设处理

第1章 综述

本文档介绍 SOC 系统的基本功能，架构和实现方式。其中，CPU 基于 MIPS32 的 5 级流水线设计实现，兼容基本的 MIPS 指令集。总线使用 Wishbone 标准接口，支持 Burst 传输模式。

1.1 物理地址分配

整个 SOC 系统的物理地址空间大致分为 RAM、ROM 和外设 I/O 三部分。外设 I/O 部分虽然较为复杂，但是所需要的地址空间很少，而 ROM 区域则用于存放 Bootloader，也不需要太大的空间。因此，整个物理空间的绝大部分将用于 RAM 空间。

- 0x0000_0000 ~ 0xFEFF_FFFF：RAM 空间，4080M，用作系统内存。
- 0xFF00_0000 ~ 0xFFFE_FFFF：ROM 空间，15.9M，用于存放 bootloader。
- 0xFFFF_0000 ~ 0xFFFF_FFFF：外设 I/O 空间，内部又细分成 256 份（用于设备号），每份 256 字节（用于端口号），提供给每个外设接口。

外设部分，目前只实现了一部分，具体可以参考第 7 章的相关内容。

1.2 五级流水线设计

本 CPU 遵循经典 MIPS 的五级流水线架构，分为 IF、ID、EXE、MEM、WB 五个流水级。各个流水级之间用寄存器控制指令流，并在必要的时候加入 stall 和 forwarding。CPU 使用单条指令的跳转延时槽。

Stall 主要有三种：指令 Cache 失配、数据 Cache 失配、LW-R 型数据竞争。

Forwarding 主要实现的是 R 型指令之间的数据前移，LW-SW 类型由于本身出现几率就很少，因此暂未实现。且由于编译器默认不实现仿存延时槽，本 CPU 也没有进行设计。

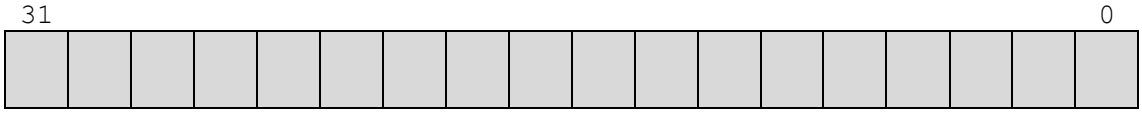
五级流水线部分自带强大的调试功能，实现方式是通过控制各个流水级的使能，将调试功能与原有的 stall 和 exception 的实现方法相整合，支持 CPU 的暂停和单步功能。不过 CPU 的暂停和单步功能不影响外设的功能，因此若在调试时没有关闭外部中断，可能导致中断不断累积来不及处理的情况发生。

第2章 CP0 设计

CP0 全称为协处理器 0（co-processor zero），在经典的五级流水线之外提供完整 CPU 所必须的其他功能如中断控制器，内存控制器，指令数据缓存等。本 CPU 的 CP0 实现与 MIPS 差异较大，仅仅实现了部分功能。

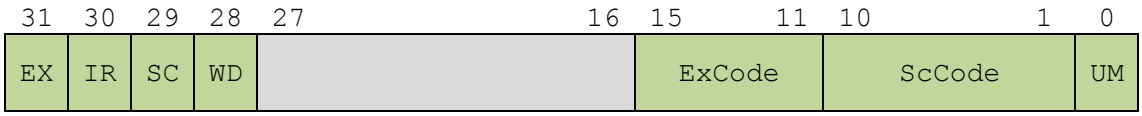
2.1 寄存器设计

在本段中，寄存器各个位及其功能使用如下表格表示：



已经实现的 CP0 寄存器从 0 开始编号，按寄存器号排列如下：

2.1.0 状态寄存器（Status Register，SR）



SR 保存 CPU 的状态信息。该寄存器只读。

EX：内部异常标志，置 1 表示发生了内部异常。

IR：外部中断标志，置 1 表示发生了外部中断。

SC：软件自陷标志，置 1 表示发生了软件自陷。

以上三位只在产生相应中断时会置位，用于向操作系统表明当前的中断类型。当发生中断嵌套或者中断返回时，原有的置位信息都会被清除，因此在任意时刻只可能有一位为 1。若三位全为 0，表示刚刚上电或复位。

WD：看门狗强制复位标志，置 1 表示此次复位由看门狗引发。

ExCode：标识最近一次不可屏蔽中断的中断号。5 位中断号一共可以表示 31 种内部异常（不包括 0）。

ScCode：记录最近一次软件自陷的自陷代码。10 位代码共可以表示 1023 种软件自陷（不包括 0）。

UM：CPU 模式位，置 0 表示内核模式，置 1 表示用户模式。

ScCode 和 ExCode 在 eret 之后并不会重置为 0，可以方便底层的调试。

2.1.1 异常参数寄存器（Exception Argument Register，EAR）

EAR 保存 SR 中所指示的 ExCode 的额外参数信息。该寄存器只读。

这个寄存器的内容在不同的不可屏蔽中断中有不同的含义，详见 3.1.1 节。

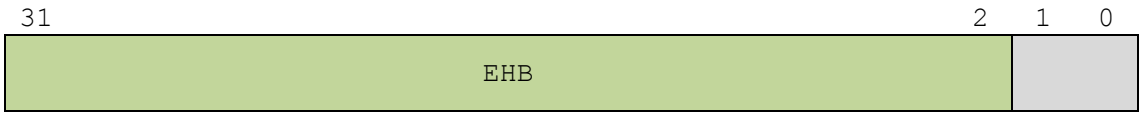
2.1.2 异常返回地址寄存器（Exception Program Counter Register，EPCR）

EPCR 保存当前异常的返回地址。该寄存器可读写。

EPC：异常返回地址的高 30 位（其最低 2 位永远为 0）。

EM：异常返回处用户模式标志。返回后会覆盖状态寄存器的 UM 位。

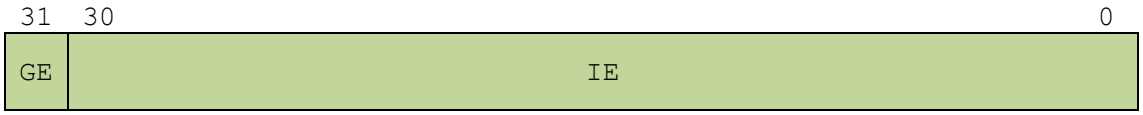
2.1.3 异常处理基址寄存器（Exception Handler Base Register，EHBR）



EHBR 保存当发生异常时 CPU 需要跳转到的目标地址。该寄存器可读写。

EHB：异常处理程序地址的高 30 位（其最低 2 位永远为 0）。

2.1.4 中断使能寄存器（Interrupt Enable Register，IER）

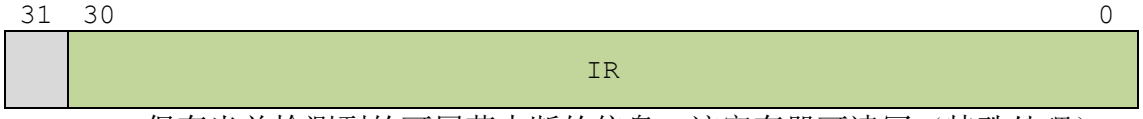


IER 保存对可屏蔽中断的使能控制信息。该寄存器可读写。

GE：全局中断使能位，置 1 表示开启全局中断使能。

IE[30:0]：单个中断使能位，置 1 表示开启相应位的中断使能。共有 31 个中断使能位，分别控制相应中断号对应的可屏蔽中断。

2.1.5 中断来源寄存器（Interrupt Cause Register，ICR）



ICR 保存当前检测到的可屏蔽中断的信息。该寄存器可读写（特殊处理）。

IR[30:0]：中断产生信号，置 1 表示相应中断号的中断已产生，需要进行处理。共有 31 个中断产生信号，与 IER 中的 31 个中断使能位相对应。当全局中断使能为 0 或者对应的中断使能为 0 时，相应中断号的中断产生信号会置 1 但

不会引发中断过程。中断处理程序在完成相关的中断处理后，应手动将相应的中断位清零。

事实上，如果单单依靠 CPU 采用“读-改-写”的方式更新 ICR 来实现中断位清零时，很可能会造成正在发生的其他中断丢失（当中断正好发生在 CPU 读取 ICR 和 CPU 写入 ICR 之间时），因此对 ICR 寄存器的写入操作使用特殊的处理方式。具体为，当 CPU 向 ICR 写入某个值时，ICR 会将写入值中比特值为 1 的那些位清零，而保持其他位不变。因为 CPU 本身没有任何理由将 ICR 的某一位从 0 改为 1 来“伪造”外部中断，因此这种做法是可以接受的。

2.1.6 页目录基址寄存器（Page Directory Base Register，PDBR）

PDBR 保存页目录表的基址供内存控制器使用。该寄存器可读写。

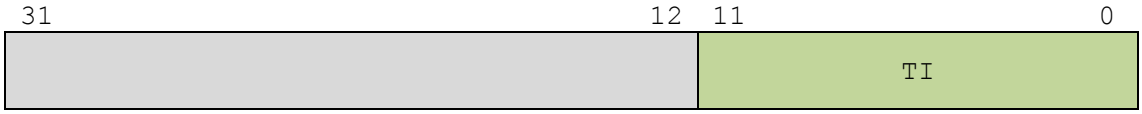
操作系统在实现任务调度时，需要事先设定好页表，并将页目录基址写入该寄存器，才能开启 MMU。对于外设 I/O 地址等必须要使用物理地址访问的情况下，可以考虑关闭 MMU，或者更推荐将相关的物理地址映射到相同的虚拟地址。

PDB：页目录基址的高 20 位。本 CPU 使用 4K 的页大小，页目录表的大小也刚好是 4k，因此在内存中需要以 4K 为边界对齐。

PE：MMU 分页使能，默认关闭。在开关此项时，必须注意让开关此项及其附近的代码具有相同的物理地址和虚拟地址。同时需要注意的是，由于 CACHE 使能位由页表项控制，因此 MMU 关闭时 CACHE 也会自动关闭。

另外，由于未设置 MMU 中 TLB 操作的指令，在每次对该寄存器进行写入操作时都会强制清空整个 TLB 以刷新页表缓存。

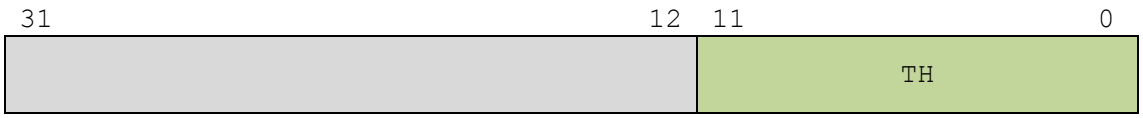
2.1.7 计时间隔寄存器（Timer Interval Register，TIR）



TIR 保存 CPU 内置计时器的相关设定。该寄存器可读写。

TI：计时间隔，单位为毫秒。由于本 CPU 的预期运行频率较低，因此以毫秒来作为单位已经是极限。计时间隔可在 1ms 到 4.096s 之间取值，CPU 会在每个计时间隔到时触发一次计时器中断。为 0 表示关闭计时器。

2.1.8 看门狗寄存器（Watch Dog Register，WDR）



WDR 保存 CPU 内置看门狗的相关设定。该寄存器可读写。

TH：溢出时间，单位为秒，可在 1s 到 4096s 之间取值。当检测到 CPU 的 PC 保持不变超过该时间阈值时，会自动复位 CPU（在调试状态下无效）。为 0 表示关闭看门狗。

第3章 中断处理

中断处理是一个完整的 CPU 非常重要的功能，同时这部分也是 CPU 设计的难点。本 CPU 通过最简单的硬件和软件的配合实现了理论上无限的中断嵌套功能，同时保留了 MIPS 的精确异常特性。

3.1 中断分类

CPU 的中断总共可以分为 4 个大类，按优先级从高到底排列为：内部异常，外部中断，软件自陷，软复位。

3.1.1 内部异常

内部异常是指 CPU 自身在运行过程中可能产生的异常，表示 CPU 在运行过程中发现了一个错误，且必须通过某种方式进行纠正。当产生内部异常时，会同时在异常参数寄存器（EAR）中写入相关信息。内部异常不可屏蔽，其异常号从 1 开始计数，按异常号顺序排列如下：

3.1.1.1 指令未对齐

当 PC 寄存器的最后两位不是 0 时发生，可能由错误的跳转指令产生。在 EAR 中存放异常的 PC 值（很可能是逻辑地址）。

操作系统应决定是否立即终止相应程序的运行，或者挂起整个系统以防止其他不可预知的错误发生。

3.1.1.2 数据未对齐

当 load 或 store 指令的操作地址没有字节对齐时发生（字操作地址没有 2 字节对齐，或者双字操作地址没有 4 字节对齐），可能是寻址时出现错误。在 EAR 中存放异常的目标地址（很可能是逻辑地址）。

操作系统可以简单地报告错误并终止程序运行，但更标准的实现应从 EPC 中获取出错的指令，将其转化为多个对齐的 load 或 store 指令模拟实现该指令功能。若该指令恰好在延迟槽中，则还需要模拟 EPC-4 处指令的跳转情况。之后设置 EPC 到下一条指令，eret 后程序即可正常执行。

3.1.1.3 未定义指令

当 CPU 取指得到的指令无法识别时发生，可能是程序出错或者跳转到了错误的地方。在 EAR 中存放当前的 PC 值（很可能是逻辑地址）。

操作系统可以简单地报告错误并终止程序运行，也可以通过软件方式模拟执行该条指令（如浮点运算指令等，可能 CPU 并不支持而产生该异常，但可以通过一些整数运算来模拟）。若该指令恰好在延迟槽中，则还需要模拟 EPC-4 处指令的跳转情况。若模拟成功，则需要设置 EPC 到下一条指令，eret 后程序即可正常执行。

3.1.1.4 非法指令

当在用户态执行特权指令时发生，一般为应用程序错误。在 EAR 中存放当前的 PC 值（必然是逻辑地址）。

特权指令是只能在内核态执行的一些指令，主要为所有协处理器操作指令和 CACHE 操作指令。

操作系统可以简单地报告错误并终止程序运行，也可以直接忽略该条指令，还可以用软件模拟该指令的执行（比如乘除法），之后设置 EPC 到下一条指令让程序继续执行。若该指令恰好在延迟槽中，则还需要模拟 EPC-4 处指令的跳转情况。

3.1.1.5 页表项不存在

当程序访问的逻辑地址找不到页表项而无法转换为物理地址时发生，可能是操作系统的调页机制将这些页调离了内存。在 EAR 中存放目标地址（必然是逻辑地址）。

操作系统需要判断该页是否实际存在，若存在则需进行调页，之后设置相关页表项，否则说明程序出错，需要强制关闭。

3.1.1.6 非法访问

当用户态程序试图访问（读取、写入或执行）一个在用户态不可访问页内的地址时发生，一般为用户程序错误。在 EAR 中存放目标地址（必然是逻辑地址）。

出于安全性考虑，操作系统应强制关闭该用户程序。

3.1.1.7 非法执行

当程序试图执行一个在不可执行页中的指令时发生，一般为用户程序错误。在 EAR 中存放目标地址（必然是逻辑地址）。

出于安全性考虑，操作系统应强制关闭该用户程序。

3.1.1.8 非法写入

当程序试图写入一个在不可写入页内的地址时发生，一般为用户程序错误。在 EAR 中存放目标地址（必然是逻辑地址）。

出于安全性考虑，操作系统应强制关闭该用户程序。

3.1.1.9 算术溢出

当使用 add、addi、sub 指令出现溢出时发生，一般为程序错误。在 EAR 中存放出错指令的 PC 值（很可能是逻辑地址）。

操作系统应决定是否立即终止相应程序的运行，或者挂起整个系统以防止其他不可预知的错误发生。

3.1.1.10 除 0 错误（暂无）

当整数除法的除数为 0 时发生，一般为程序错误。在 EAR 中存放出错指令的 PC 值（很可能是逻辑地址）。

操作系统应决定是否立即终止相应程序的运行，或者挂起整个系统以防止其他不可预知的错误发生。

3.1.1.11 指令总线错误

在试图从总线获取指令时出错，很可能是访问了一个不存在的地址。在 EAR 中存放出错指令的 PC 值（很可能是逻辑地址）。

这一般是一种硬件错误或者内核错误，因为用户程序受地址转换的保护，不会出现这样的异常。为方便调试，在出现此异常时硬件 CPU 将无条件停止运行。

3.1.1.12 数据总线错误

在试图同总线进行数据交互时出错，很可能是访问了一个不存在的地址。在 EAR 中存放异常的目标地址（很可能是逻辑地址）。

这一般是一种硬件错误或者内核错误，因为用户程序受地址转换的保护，不会出现这样的异常。为方便调试，在出现此异常时硬件 CPU 将无条件停止运

行。

3.1.2 外部中断

外部中断是外设请求 CPU 进行信息处理的主要手段，外设由 CPU 进行控制和设定，在指定的条件满足后再通过中断的方式告知 CPU。与轮询的方式相比，使用中断可以大大提升 CPU 与外设进行数据通信的效率。外部中断受中断使能寄存器（IER）进行使能控制，其中断号从 0 开始，按中断号顺序排列如下：

3.1.2.0 计时器中断

当计时器所计时间达到 TIR 所指定的时间时发生。

3.1.2.1 VGA 中断（暂无）

当 VGA 状态发生变化，如 VGA 线接入时发生。（暂未实现，因为目前的开发板不提供接入检测的功能）

3.1.2.2 板级输入中断

当使用开发板上的 SWITCH 或者 BUTTON 时发生。

3.1.2.3 键盘中断

当使用键盘进行输入时发生

3.1.2.4 鼠标中断（暂无）

当使用鼠标进行输入时发生。

3.1.2.5 SPI 中断

当通过 SPI 串口发送数据完成，或者接收到数据需要 CPU 读取时发生。

3.1.2.6 UART 中断

当通过 UART 串口发送数据完成，或者接收到数据需要 CPU 读取时发生。

3.1.2.7 并口中断（暂无）

当通过并口发送数据完成，或者接收到数据需要 CPU 读取时发生。

3.1.2.8 USB 口中断（暂无）

当通过 USB 口发送数据完成，或者接收到数据需要 CPU 读取时发生。

3.1.2.9 网口中断（暂无）

当通过网口发送数据中断，或者接收到数据需要 CPU 读取时发生。

3.1.3 软件自陷

软件自陷是用户态程序调用内核态的主要方式，所有资源均由操作系统进行管理，用户态程序通过软件自陷的方式“申请”资源，这样也大大提升了整个系统的可用性和安全性。

在 MIPS 中，软件自陷由无条件自陷指令 SYSCALL 或者有条件自陷指令 TEQ、TEQI、TGE、TGEI、TGEIU、TGEU、TLT、TLTI、TLTIU、TLTU、TNE、TNEI 产生。本 CPU 只实现了 SYSCALL 指令，另外，SYSCALL 指令内部可以直接编码一个调用号，但是更多的操作系统会选择使用约定好的寄存器来传递调用号和相关的参数。

3.1.4 复位

MIPS CPU 将复位过程也当做一个异常来处理，这样的好处是可以重用许多异常处理中的流程，比如对 EPC 的写入，内核态的切入等等。需要注意的是，当操作系统完成初始化并开始创建第一个用户线程准备进入用户态时，应当将用户线程的入口点的逻辑地址写入 EPCR.EPC 并将 EPCR.EM 置位，使用 eret 指令进入用户态。

3.2 中断细节

中断可以分为 4 大类，但硬件不对这几类异常进行优先级上的区分，而全部交给软件处理。操作系统的异常处理程序需要首先读取状态寄存器得知当前的异常类别，然后再进一步读取相关寄存器获取异常类型。具体规则如下：

- 若 SR.EX 置位，OS 需要读取 SR.ExCode 判断内部异常类型，之后进行处理。
- 若 SR.IR 置位，OS 需要读取 ICR 寄存器判断外部中断类型（可能有多个等待处理的外部中断），之后自行决定优先级进行处理。处理完成后，需要清空 ICR 寄存器的某一位。
- 若 SR.SC 置位，OS 需要读取 SR.ScCode 判断自陷类型（或读取相应的寄存器获取，自陷参数使用普通寄存器传递），之后进行处理。
- 若三者均没有置位，表示 OS 正在启动或者被重启，需要开始系统初始

化。

此外，如果发现 SR.WD 置位，说明看门狗发生了动作，可以通过 EPCR 查看产生错误的原始 PC 地址（可能是某个用户程序的逻辑地址），由于本系统使用五级流水 CPU，因此该地址前后几条指令都可能是错误源。

在中断处理过程中，为了实现中断嵌套，操作系统需要和硬件完成如下约定：

- 刚进入中断处理过程时，IER.GE 由硬件关闭，软件需尽快保存上下文。
- 软件保存好相关寄存器和 EPC 后，可以打开 IER.GE 允许中断嵌套。
- 软件完成中断相关的处理后，关闭 IER.GE，之后进行寄存器和 EPC 的恢复。
- 调用 eret 指令，硬件打开 IER.GE，同时当前中断返回。

中断处理程序处理完中断之后，需要将 ICR 中的相应位置为 0。同时若软件需要全局禁用中断，请将 IER 所有位全部清零，而不是单单清空 IER.GE。

各种中断和异常使用统一的入口点，具体地址由软件通过 EHBR 寄存器设置，默认为系统启动的地址（即 ROM 空间首地址），中断优先级全权交给软件处理。一般来讲，如果操作系统自行设置了 EHBR，则在中断处理函数中不用考虑系统重启的情况，因为系统重启后将直接从系统启动地址开始运行。

3.3 精确异常

为了实现对各种异常和嵌套情况的正确处理，CPU 统一在 MEM 阶段捕获内部异常和外部中断（在流水线的其他阶段只完成异常状态的记录，但不触发）。同时由于 eret 指令在 EXE 阶段即通过异常处理器件完成跳转，因此 CPU 在硬件层面进行了约束，通过在某几个阶段插入流水线停顿来将所有的特权级指令和其他指令在流水线中进行分隔，保证特权级指令前后不存在其他有效指令。这样可以避免许多由于特权指令（如异常返回，中断屏蔽位修改等）引起的异常错位。

在实现了上述关键技术之后，本 CPU 能够保证当异常触发时，在异常指令之前的指令都能够正常执行完毕，而异常指令之后的指令将全部取消。即支持完整的精确异常。

另外的一些细节问题，如中断发生在延时槽中时，可能会丢失跳转信息。本 CPU 能够自动进行判断，并将 EPC 设置为延时槽上面的跳转指令。

第4章 Cache 设计

为了减缓 CPU 与内存之间速度不对等造成的 CPU 等待问题，CPU 内部集成了一个分离的指令缓存和数据缓存。

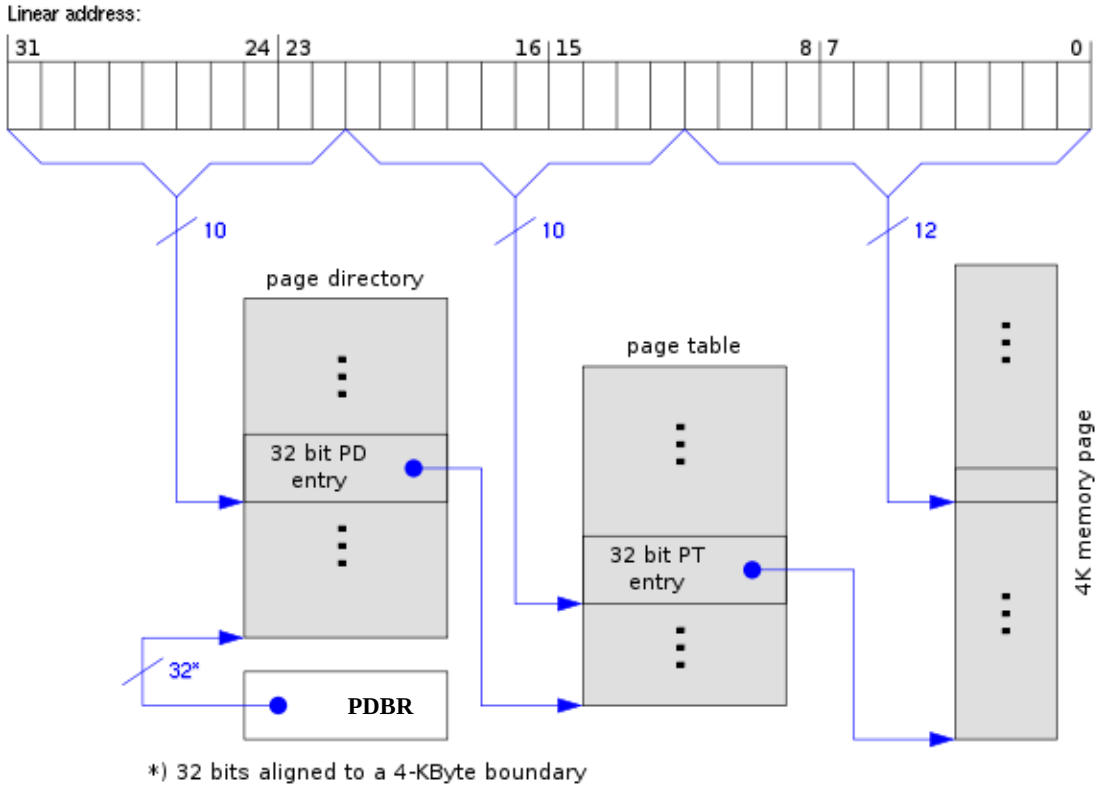
目前 Cache 的设计还比较简单，软件方面仅有一个指令用于 Cache 的强制写回。

第5章 MMU 设计

内存管理单元 MMU（Memory Management Unit）主要用于提供逻辑地址到物理地址的转换。这里借鉴了 Linux 内核的设计思路，仅仅使用完全的分页机制来进行虚拟内存的管理。

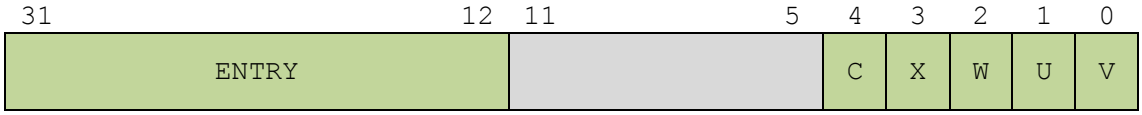
5.1 两级页表

本 CPU 为标准的 32 位架构，因此使用两级页表就足以满足地址转换的需要，简单起见，所有的页面大小均为 4K，整个地址转换的过程全部由硬件完成，如下图所示：



页目录基址位于 CP0 的 PDBR 中。整个页目录表有 1024 项，刚好存放在一个物理页中，通过逻辑地址的[31:22]位当做索引查询页目录表，得到一个页表基址。每个页表都有 1024 项，也恰好存放在一个物理页中，通过逻辑地址的[21:12]位作为索引进行查询，可以得到所需要的页的基址。最后使用逻辑地址的[12:0]位作为物理页内偏移，得到最终的物理地址。

每一个页目录项和页表项的结构如下：



ENTRY：页表基址或者页基址的高 20 位。需要 4K 对齐。

C：Cache 使能位。置 1 表示该页数据可以使用 Cache 以便加速存取。

X：可执行权限位。置 1 表示程序可以将该页内容作为代码执行。

W：可写权限位。置 1 表示程序可以向该页写入数据。

U：用户模式位。置 1 表示用户模式可访问。

V：页表项有效位。置 1 表示该项有效。

页目录项和页表项的属性组合按约束从严处理，即如果页目录项的某个属性位为关闭状态，则无论页表项的相关属性位为何，其最终属性为关闭。

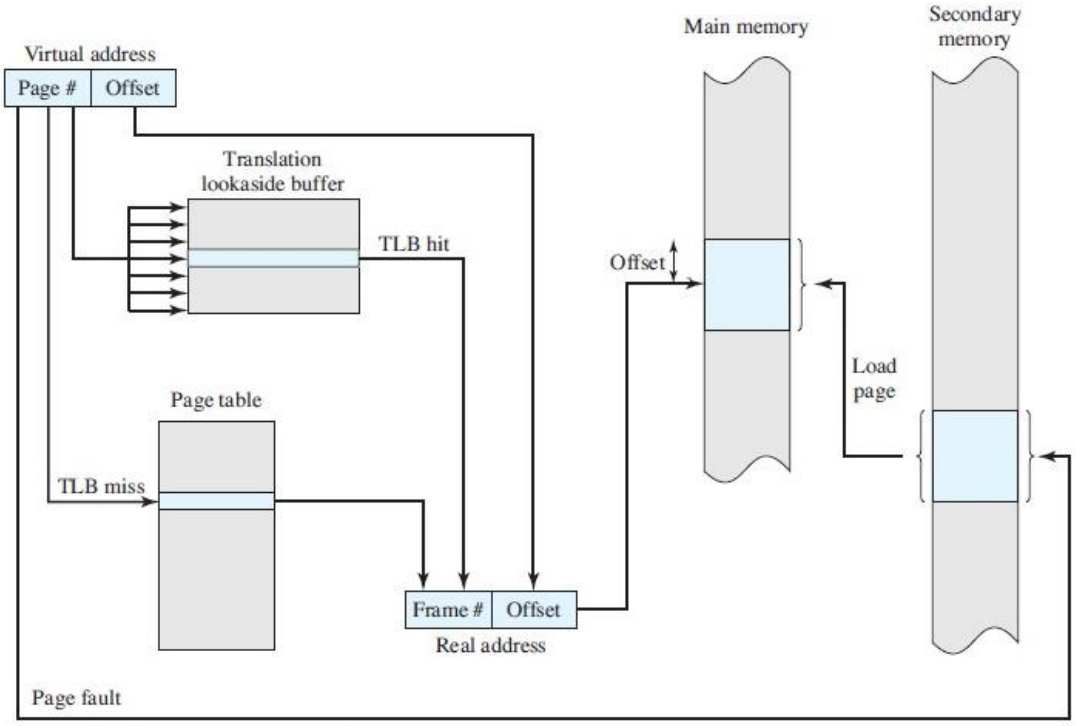
5.2 页表缓冲

TLB（Translation lookaside buffer）是页表项的一个缓冲区。因为使用了两级页表，每次逻辑地址到物理地址的转换都需要读取两次内存，由于内存速度远远低于 CPU，这样的时间开销是非常大的。而在一段时间之内程序一般会使用同一个页表区域里的指令，操作同一个页表区域里的数据，因此可以将查询过的最后一级页表项缓存住，加快相同页表地址的转换。

TLB 本质上是一个全相联的 Cache，每一个 TLB 项的构成如下：

- TLB 标志 - 1 位，表示该 TLB 项是否有效。
- 虚拟地址 - 20 位，表示地址转换之前的虚拟地址的高 20 位。
- 物理地址 - 20 位，表示地址转换之后的物理地址的高 20 位。
- 页表标志 - 5 位，表示该页表项的相关属性，如权限和有效性。

5.3 缺页异常



如上图所示，MMU 在进行地址转换时，会首先查询 TLB，如果 TLB 中已经有相关的页表项，则提取出来直接进行转换得到物理地址。如果 TLB 失配，则通过两级页表查询得到页表项存入 TLB，并完成地址转换。如果前两者都失败，则产生一个缺页异常，交由系统进行调页处理或者强制关闭用户程序。

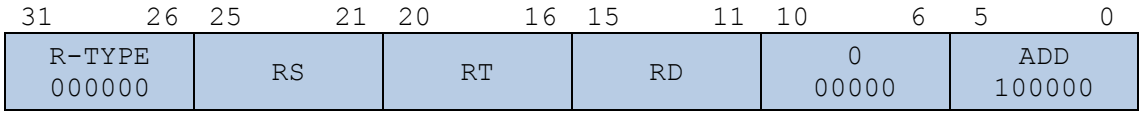
因为 TLB 的填充操作全部由硬件自动完成，大大简化了软件上的相关设计。目前指令集中没有任何关于 MMU 的操作指令，若软件对已被 TLB 缓存的页表项进行了修改，希望清空 TLB 时，只需要发起一个对 PDBR 寄存器的写入操作即可（可以写入与原来相同的值）。

第6章 指令集

本 CPU 的指令集与标准 MIPS32 指令集大致相同，但去除了很多不常见的指令，同时新增了几个自行设计的指令。修改之后的指令会在指令名后添加‘*’字符。

6.1 算术运算

6.1.1 ADD



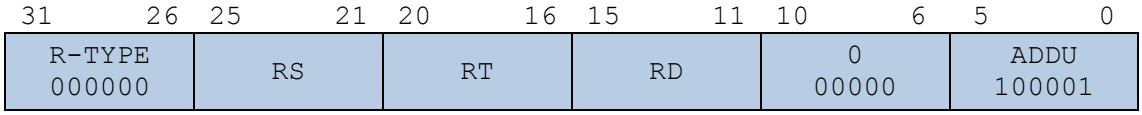
格式：ADD rd, rs, rt

功能：加法

描述：GPR[rd] = GPR[rs] + GPR[rt]

异常：算术溢出

6.1.2 ADDU



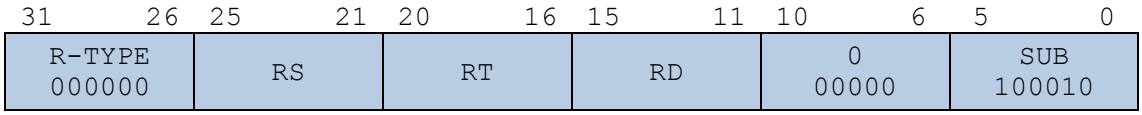
格式：ADDU rd, rs, rt

功能：无符号加法

描述：GPR[rd] = GPR[rs] + GPR[rt]

异常：无

6.1.3 SUB



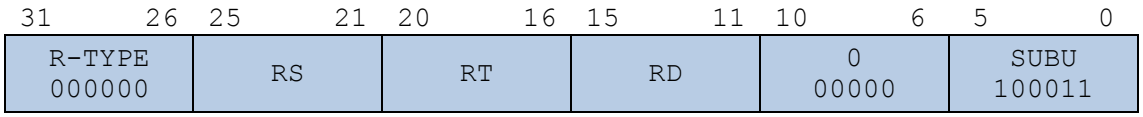
格式：SUB rd, rs, rt

功能：减法

描述：GPR[rd] = GPR[rs] - GPR[rt]

异常：算术溢出

6.1.4 SUBU



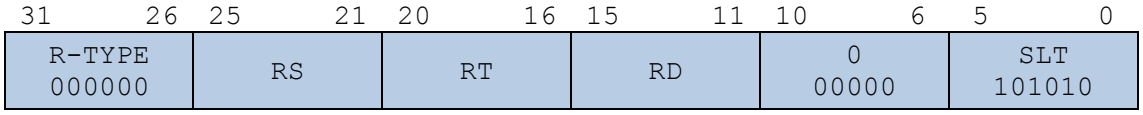
格式：SUBU rd, rs, rt

功能：无符号减法

描述：GPR[rd] = GPR[rs] - GPR[rt]

异常：无

6.1.5 SLT



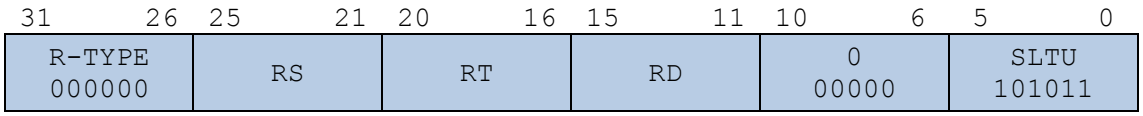
格式：SLT rd, rs, rt

功能：小于时置位

描述：GPR[rd] = (GPR[rs] < GPR[rt])

异常：无

6.1.6 SLTU



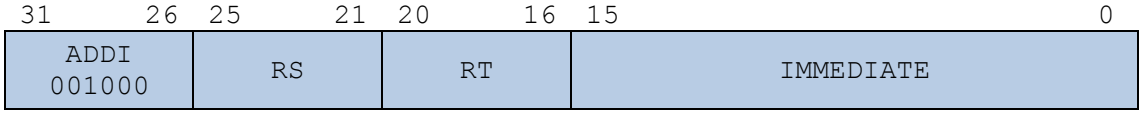
格式：SLTU rd, rs, rt

功能：无符号小于时置位

描述：GPR[rd] = (GPR[rs] < GPR[rt])

异常：无

6.1.7 ADDI



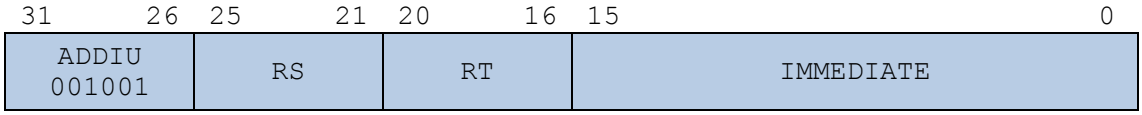
格式：ADDI rt, rs, immediate

功能：立即数加法

描述：GPR[rt] = GPR[rs] + sign_ext(immediate)

异常：算术溢出

6.1.8 ADDIU



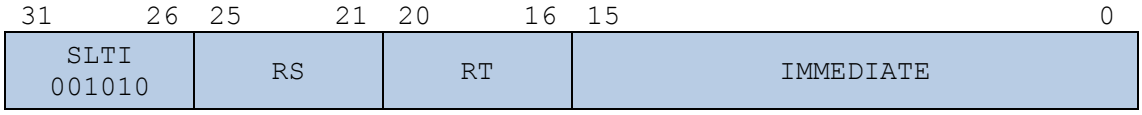
格式：ADDIU rt, rs, immediate

功能：无符号立即数加法

描述：GPR[rt] = GPR[rs] + sign_ext(immediate)

异常：无

6.1.9 SLTI



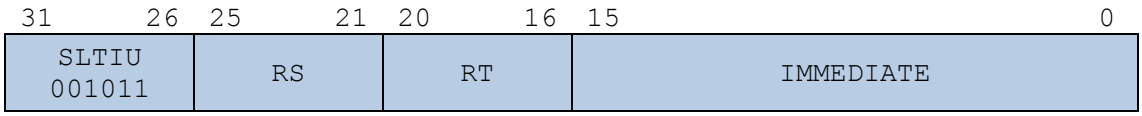
格式：SLTI rt, rs, immediate

功能：小于立即数时置位

描述：GPR[rt] = GPR[rs] < sign_ext(immediate)

异常：无

6.1.10 SLTIU



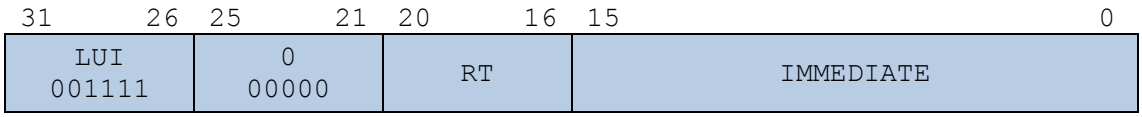
格式：SLTIU rt, rs, immediate

功能：小于无符号立即数时置位

描述：GPR[rt] = GPR[rs] < sign_ext(immediate)

异常：无

6.1.11 LUI



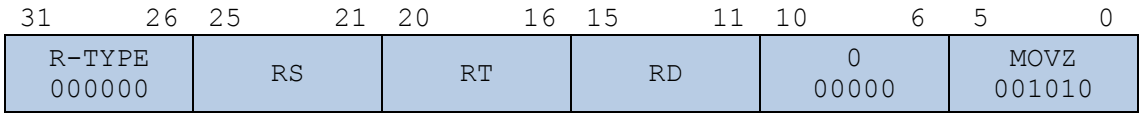
格式：LUI rt, immediate

功能：高位字载入

描述：GPR[rt] = {immediate, 16'b0}

异常：无

6.1.12 MOVZ



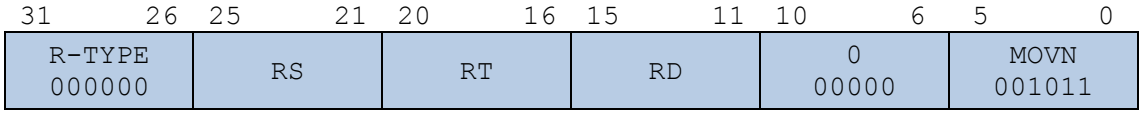
格式：MOVZ rd, rs, rt

功能：为零时移动

描述：if GPR[rt] == 0 then GPR[rd] = GPR[rs]

异常：无

6.1.13 MOVN



格式：MOVN rd, rs, rt

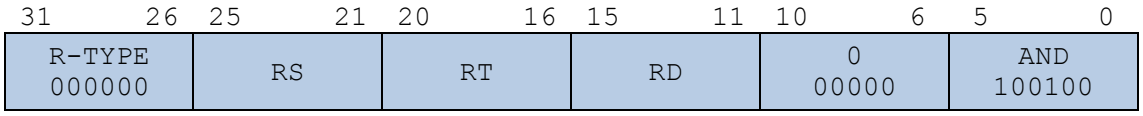
功能：非零时移动

描述：if GPR[rt] != 0 then GPR[rd] = GPR[rs]

异常：无

6.2 逻辑运算

6.2.1 AND



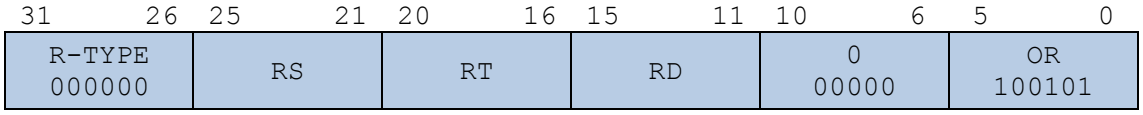
格式：AND rd, rs, rt

功能：逻辑与

描述：GPR[rd] = GPR[rs] & GPR[rt]

异常：无

6.2.2 OR



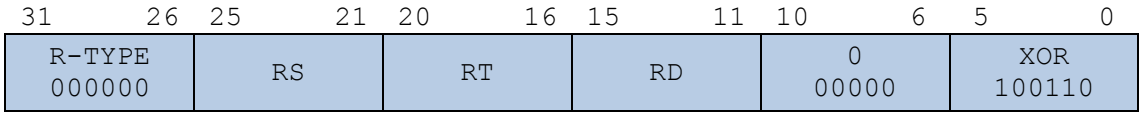
格式：OR rd, rs, rt

功能：逻辑或

描述：GPR[rd] = GPR[rs] | GPR[rt]

异常：无

6.2.3 XOR



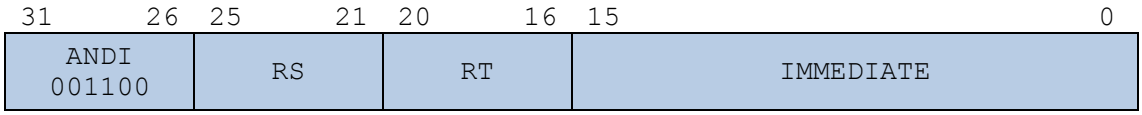
格式：XOR rd, rs, rt

功能：逻辑异或

描述：GPR[rd] = GPR[rs] ^ GPR[rt]

异常：无

6.2.4 ANDI



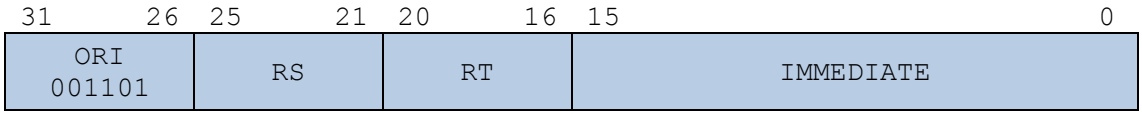
格式：SLTI rt, rs, immediate

功能：立即数逻辑与

描述：GPR[rt] = GPR[rs] & zero_ext(immediate)

异常：无

6.2.5 ORI



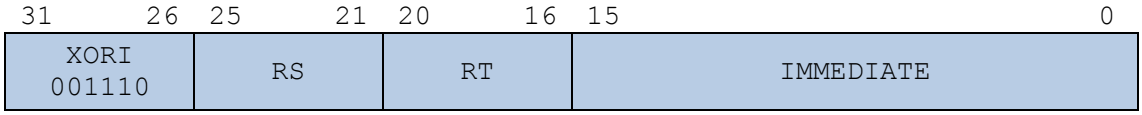
格式：ORI rt, rs, immediate

功能：立即数逻辑或

描述：GPR[rt] = GPR[rs] | zero_ext(immediate)

异常：无

6.2.6 XORI



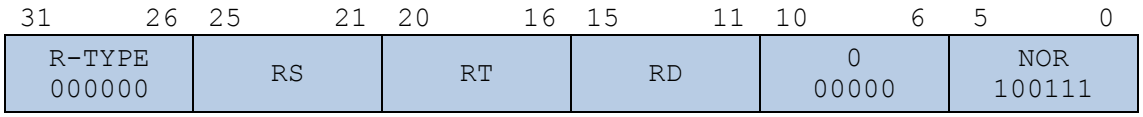
格式：XORI rt, rs, immediate

功能：立即数逻辑与

描述：GPR[rt] = GPR[rs] ^ zero_ext(immediate)

异常：无

6.2.7 NOR



格式：NOR rd, rs, rt

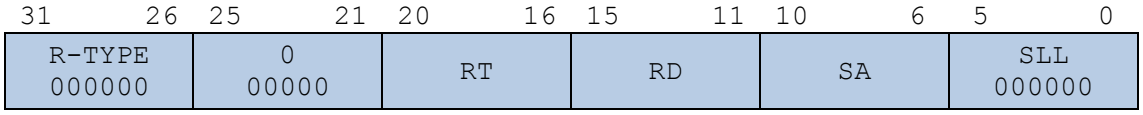
功能：逻辑或非

描述：GPR[rd] = ~(GPR[rs] | GPR[rt])

异常：无

6.3 移位运算

6.3.1 SLL



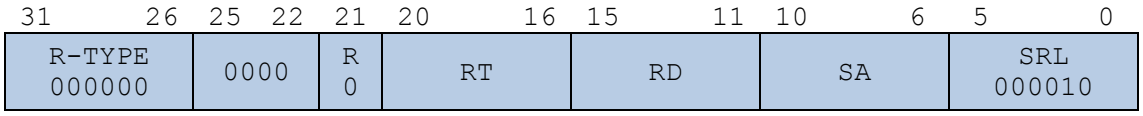
格式：SLL rd, rt, sa

功能：左移

描述：GPR[rd] = GPR[rt] << sa

异常：无

6.3.2 SRL



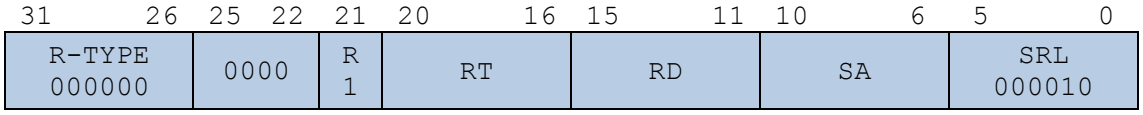
格式：SRL rd, rt, sa

功能：逻辑右移

描述：GPR[rd] = GPR[rt] >> sa (logical)

异常：无

6.3.3 ROTR



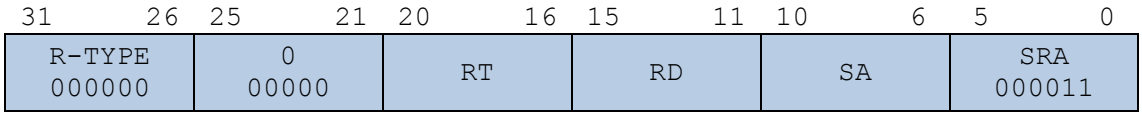
格式：ROTR rd, rt, sa

功能：循环右移

描述：GPR[rd] = GPR[rt] <-> sa

异常：无

6.3.4 SRA



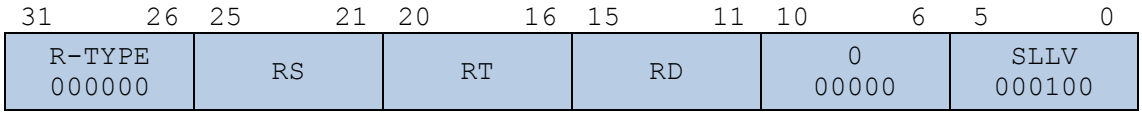
格式：SRA rd, rt, sa

功能：算术右移

描述：GPR[rd] = GPR[rt] >> sa (arithmetic)

异常：无

6.3.5 SLLV



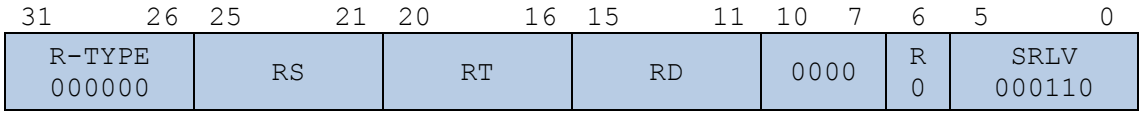
格式：SLLV rd, rt, rs

功能：变量左移

描述：GPR[rd] = GPR[rt] << GPR[rs]

异常：无

6.3.6 SRLV



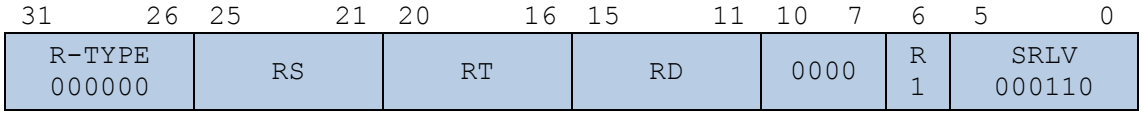
格式：SRLV rd, rt, rs

功能：变量逻辑右移

描述：GPR[rd] = GPR[rt] >> GPR[rs] (logical)

异常：无

6.3.7 ROTRV



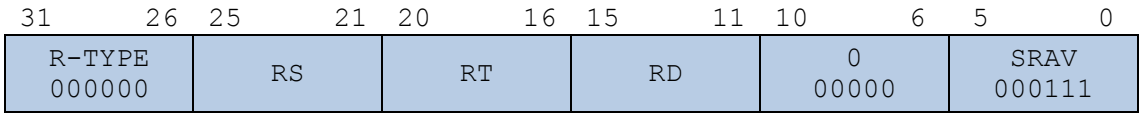
格式：ROTRV rd, rt, rs

功能：变量循环右移

描述：GPR[rd] = GPR[rt] <-> GPR[rs]

异常：无

6.3.8 SRAV



格式：SRAV rd, rt, rs

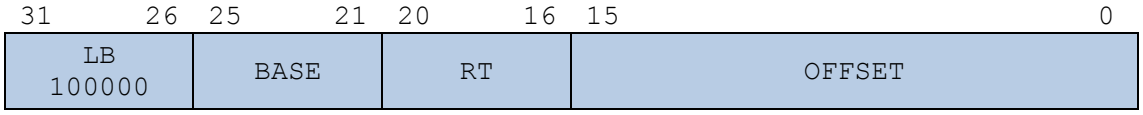
功能：变量算术右移

描述：GPR[rd] = GPR[rt] >> GPR[rs] (arithmetic)

异常：无

6.4 内存读写

6.4.1 LB



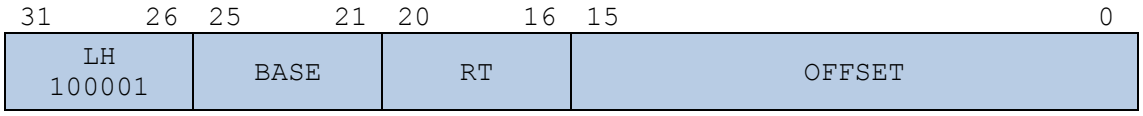
格式：LB rt, offset(base)

功能：字节读取

描述：GPR[rt] = MEM[GPR[base] + sign_ext(offset)]

异常：数据地址无效、非法读取、页表项不存在

6.4.2 LH



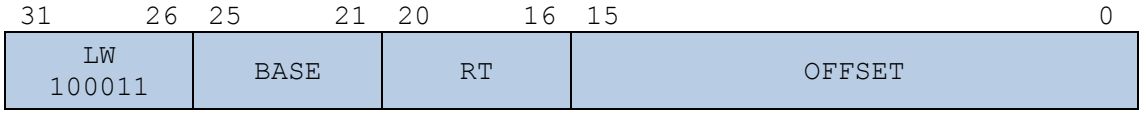
格式：LH rt, offset(base)

功能：半字读取

描述：GPR[rt] = MEM[GPR[base] + sign_ext(offset)]

异常：数据地址无效、数据未对齐、非法读取、页表项不存在

6.4.3 LW



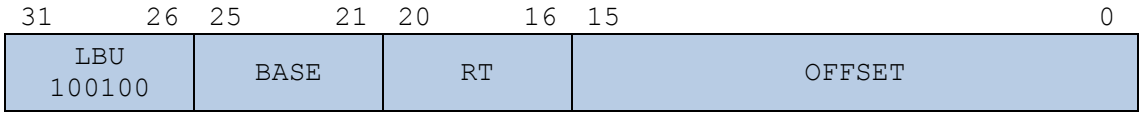
格式：LW rt, offset(base)

功能：字读取

描述：GPR[rt] = MEM[GPR[base] + sign_ext(offset)]

异常：数据地址无效、数据未对齐、非法读取、页表项不存在

6.4.4 LBU



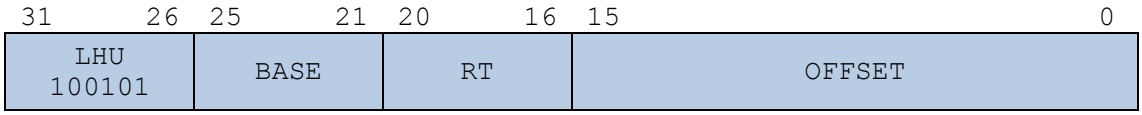
格式：LBU rt, offset(base)

功能：无符号字节读取

描述：GPR[rt] = MEM[GPR[base] + sign_ext(offset)]

异常：数据地址无效、非法读取、页表项不存在

6.4.5 LHU



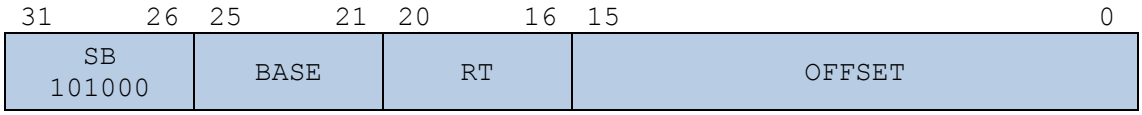
格式：LHU rt, offset(base)

功能：无符号半字读取

描述：GPR[rt] = MEM[GPR[base] + sign_ext(offset)]

异常：数据地址无效、数据未对齐、非法读取、页表项不存在

6.4.6 SB



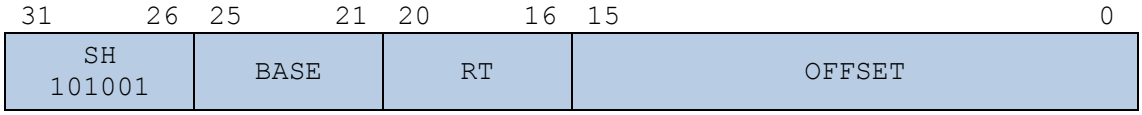
格式：SB rt, offset(base)

功能：字节写入

描述：MEM[GPR[base] + sign_ext(offset)] = GPR[rt]

异常：数据地址无效、非法写入、页表项不存在

6.4.7 SH



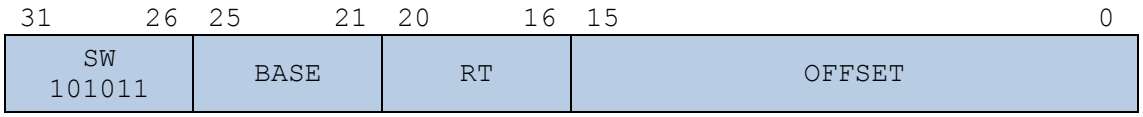
格式：SH rt, offset(base)

功能：半字写入

描述：MEM[GPR[base] + sign_ext(offset)] = GPR[rt]

异常：数据地址无效、数据未对齐、非法写入、页表项不存在

6.4.8 SW



格式：SW rt, offset(base)

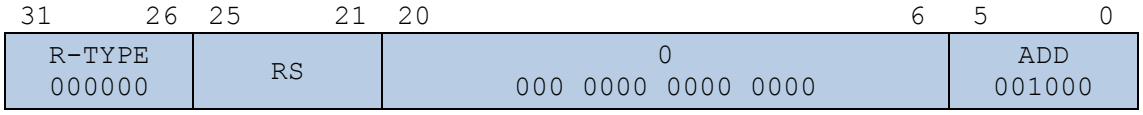
功能：字写入

描述：MEM[GPR[base] + sign_ext(offset)] = GPR[rt]

异常：数据地址无效、数据未对齐、非法写入、页表项不存在

6.5 地址跳转

6.5.1 JR



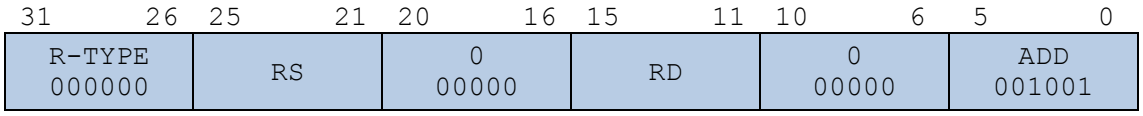
格式：JR rs

功能：寄存器跳转

描述：PC = GPR[rs]

异常：无

6.5.2 JALR



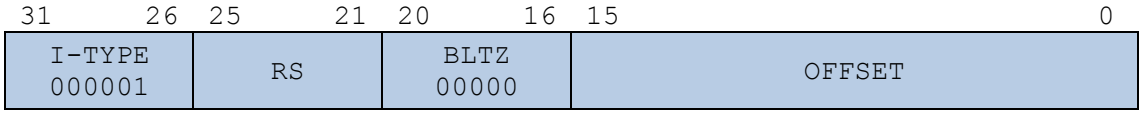
格式：JALR rd, rs （rd 不指定时默认为$31）

功能：寄存器链接跳转

描述：GPR[rd] = PC+8, PC = GPR[rs]

异常：无

6.5.3 BLTZ



格式：BLTZ rs, offset

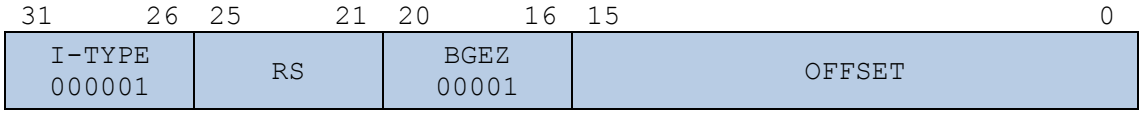
功能：小于零时跳转

描述：if GPR[rs] < 0 then PC = PC+4 + {sign_ext(offset),

2'b00}

异常：无

6.5.4 BGEZ



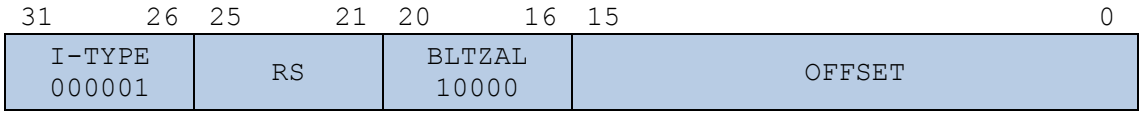
格式：BGEZ rs, offset

功能：大于等于零时跳转

描述：if GPR[rs] >= 0 then PC = PC + 4 + {sign_ext(offset), 2'b00}

异常：无

6.5.5 BLTZAL



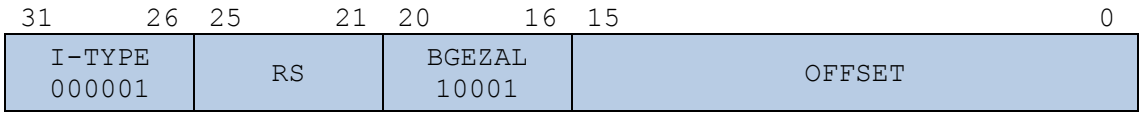
格式：BLTZAL rs, offset

功能：小于零时链接跳转

描述：if GPR[rs] < 0 then GPR[31] = PC+8, PC = PC+4 + {sign_ext(offset), 2'b00}

异常：无

6.5.6 BGEZAL



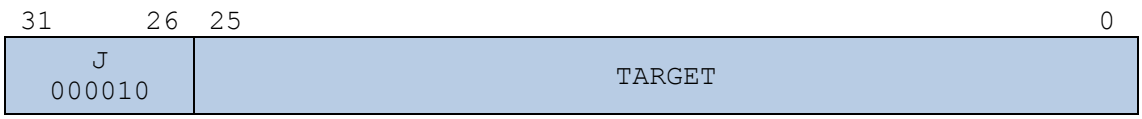
格式：BGEZAL rs, offset

功能：大于等于零时链接跳转

描述：if GPR[rs] >= 0 then GPR[31] = PC+8, PC = PC+4 + {sign_ext(offset), 2'b00}

异常：无

6.5.7 J



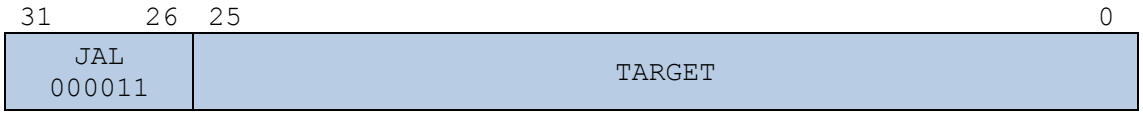
格式：J target

功能：跳转

描述：PC = {PC[31:28], target, 2'b00}

异常：无

6.5.8 JAL



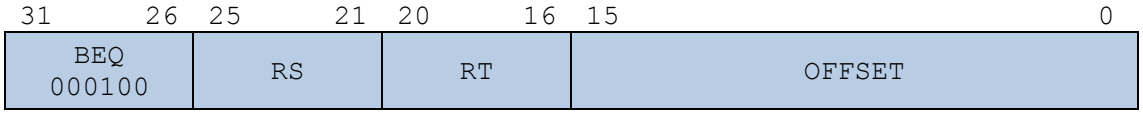
格式：JAL target

功能：链接跳转

描述：GPR[31] = PC+8, PC = {PC[31:28], target, 2'b00}

异常：无

6.5.9 BEQ



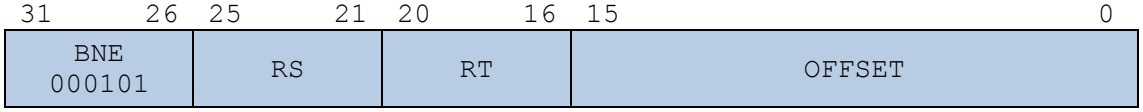
格式：BEQ rs, rt, offset

功能：相等时跳转

描述：if GPR[rs] == GPR[rt] then PC = PC+4 + {sign_ext(offset), 2'b00}

异常：无

6.5.10 BNE



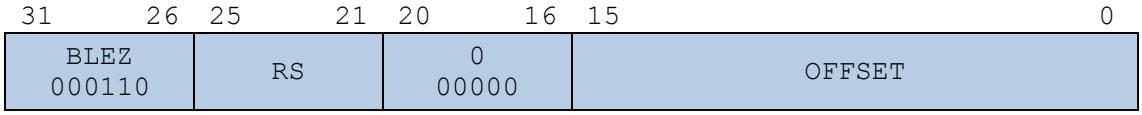
格式：BNE rs, rt, offset

功能：不等时跳转

描述：if GPR[rs] != GPR[rt] then PC = PC+4 + {sign_ext(offset), 2'b00}

异常：无

6.5.11 BLEZ



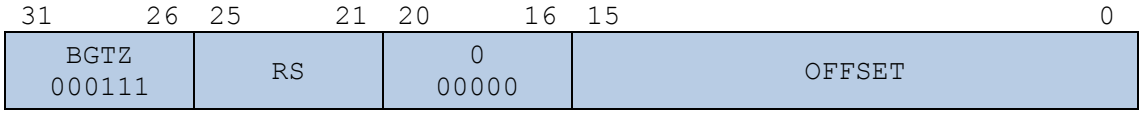
格式：BLEZ rs, offset

功能：小于等于零时跳转

描述：if GPR[rs] <= 0 then PC = PC + 4 + {sign_ext(offset), 2'b00}

异常：无

6.5.12 BGTZ



格式：BGTZ rs, offset

功能：大于零时跳转

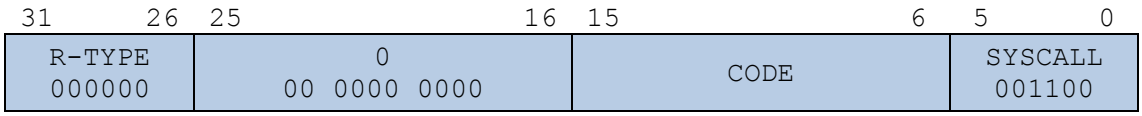
描述：if GPR[rs] > 0 then PC = PC+4 + {sign_ext(offset), 2'b00}

异常：无

6.6 软件自陷

本 CPU 自陷指令的实现方式与标准 MIPS 不完全兼容。主要区别在于会将 CODE 值改为 10 位，并会自动存入 SR 寄存器的 ScCode 中。

6.6.1 SYSCALL



格式：SYSCALL code

功能：自陷，Syscall

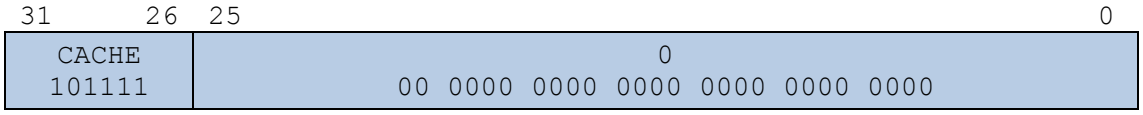
异常：软件自陷

6.7 CACHE 操作*

标准 MIPS 的 CACHE 指令设计十分复杂，且软件对基础硬件有完全的控制

能力。本 CPU 为了减少软硬件之间的耦合度，尽量实现 CAHCE 的透明化，在这里只设置了一条 CACHE 指令，完成 CACHE 的强制写回和作废，用于在写入指令后强制写回到内存，或者其他设备的数据冲刷操作。该指令 OP 码与 MIPS 兼容，换句话说，MIPS 指令中的任何 CACHE 操作指令，均会被本系统判断为 CACHE 写回和作废操作。

6.7.1 CACHE*



格式：CACHE

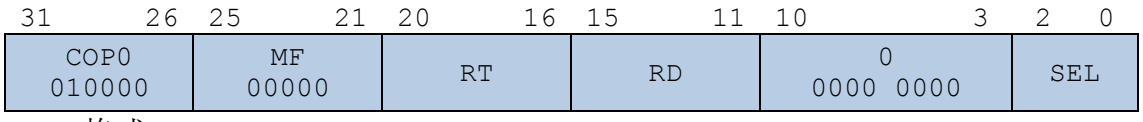
功能：CACHE 写回作废，将同时操作数据缓存和指令缓存。目前暂时不支持部分写回功能，一个 CACHE 指令会将整个 CACHE 写回并作废。

异常：无

6.8 CP 操作

CP 指令包括 CP0、CP1、CP2 等，其中仍然使用 5 位来索引寄存器号，因此每个 CP 中总共可以使用 32 个寄存器。MIPS32 标准还添加了 3 位的 SEL 来增强 CP0 中可索引的寄存器个数，因此扩展到了 256 个。本 CPU 仅使用很少的 CP0 寄存器，因此 SEL 位不使用，需置 0。

6.8.1 MFC0



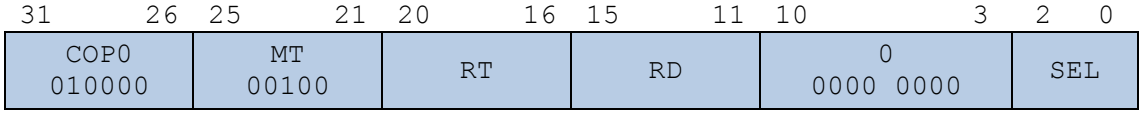
格式：MFC0 rt, rd, sel

功能：从 C0 载入

描述：GPR[rt] = CPR[0, rd, sel]

异常：非法指令

6.8.2 MTC0



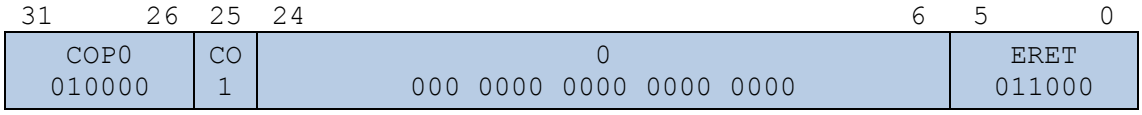
格式：MTC0 rt, rd, sel

功能：存储至 C0

描述：CPR[0, rd, sel] = GPR[rt]

异常：非法指令

6.8.3 ERET



格式：ERET

功能：异常返回

异常：非法指令

第7章 外设接口

外设 I/O 端口统一安排在地址 0xFFFF_0000 往上区域。内部又细分成 256 份，每份 256 字节，提供给每个外设接口，用于相关控制寄存器地址的映射。

特别的，所有对这些地址空间的操作，均不应经过 CPU 的 CACHE，同时有些地址只有最低 1 个字节有效，那么无论使用 LW、LH 还是 LB，本质上都只会进行字节操作，但只要地址对齐，并不会报错。

外设列表按外设编号排列如下：

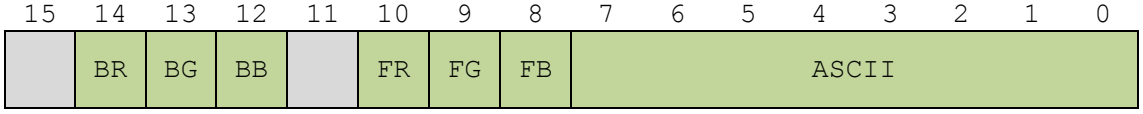
7.0 保留

该外设接口对应的地址空间为 0xFFFF_0000 ~ 0xFFFF_00FF，系统保留。

7.1 VGA

VGA 有文本和图形两种显示模式，支持多种分辨率。文本模式支持 8 色 16*8 点阵 ASCII 码显示，图形模式支持 256 色像素显示。文本模式支持硬件光标。

在文本模式下，每个 ASCII 字符使用两个字节表示，如下所示。



BR：背景红色分量。

BG：背景绿色分量。

BB：背景蓝色分量。

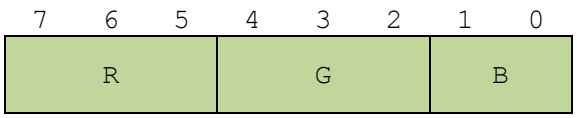
FR：前景红色分量。

FG：前景绿色分量。

FB：前景蓝色分量。

ASCII：待显示的 ASCII 码。

在图形模式下，每个像素使用一个字节表示，如下所示。

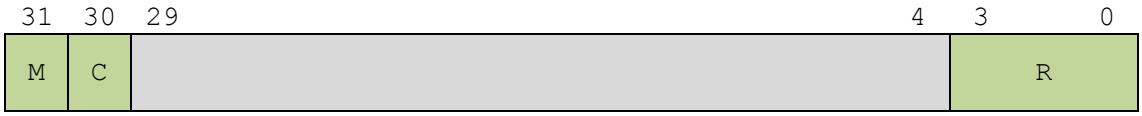


R：像素点红色分量。

G：像素点绿色分量。

B：像素点蓝色分量。

- 0x00 ~ 0x03 用于模式控制。



M：模式位，置 0 为文本模式，置 1 为图形模式。

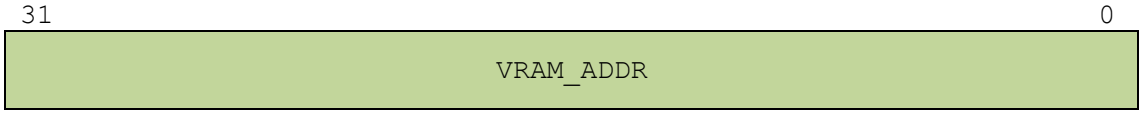
C：硬件光标使能位，置 1 时开启硬件光标。

R：分辨率选择，0 表示关闭 VGA 输出，其余模式如下表所示：

模式	分辨率@刷新率	长宽比	标准类型
1	640 * 480 @ 60Hz	4:3	工业标准
2	640 * 480 @ 72Hz	4:3	VESA 标准
3	640 * 480 @ 75Hz	4:3	VESA 标准
4	800 * 600 @60Hz	4:3	VESA 指导建议
5	800 * 600 @ 72Hz	4:3	VESA 标准
6	800 * 600 @ 75Hz	4:3	VESA 标准
7	1024 * 768 @ 60Hz	4:3	VESA 指导建议

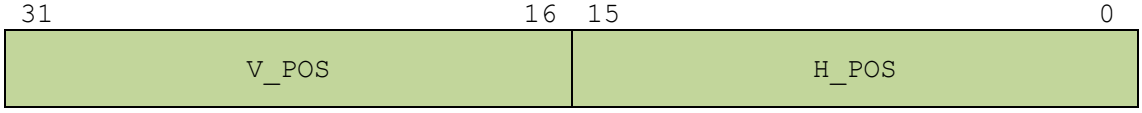
更高的分辨率需要更高的 VGA 时钟频率和总线速率，在 Nexys3 开发板上实现较难且会严重影响总线其余部件的工作效率，故不在此列出。

- 0x04 ~ 0x07 为显存起始地址，文本模式和图形模式共用。



在文本模式下，VRAM_ADDR 的末 16 位需为 0（64KB 对齐），在图形模式下，VRAM_ADDR 的末 20 位需为 0（1MB 对齐）。

- 0x08 ~ 0x0B 为硬件光标位置，文本模式专用。

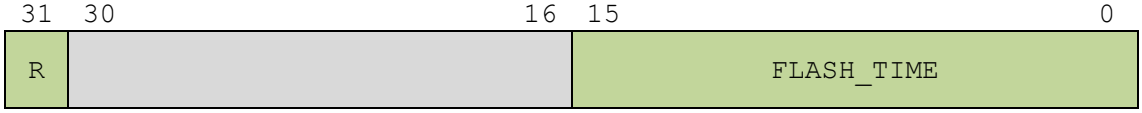


V_POS：硬件光标的行坐标。

H_POS：硬件光标的列坐标。

在文本模式下，V_POS 和 H_POS 为字符计数，从 0 开始。

- 0x0C ~ 0x0F 为硬件光标闪烁时间，文本模式专用。



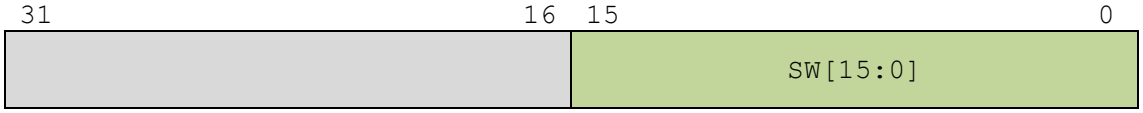
R：置 1 时，当光标位置发生改变时强制点亮光标。

FLASH_TIME：硬件光标的点亮时间（单位为毫秒），为 0 表示长亮。

7.2 板级 IO

板级 IO 包括开发板上的基本输入输出，包括按钮，开关，LED 灯，七段数码管。按钮和开关输入包含基本的防抖功能，七段数码管则可直接显示 16 位的数据。为了兼容不同型号的开发板，这个部分的地址分配有较多留空的部分，请留意。

- 0x00 ~ 0x03 为开关的状态，该寄存器只读。



SW[15:0]：拨位开关的状态。

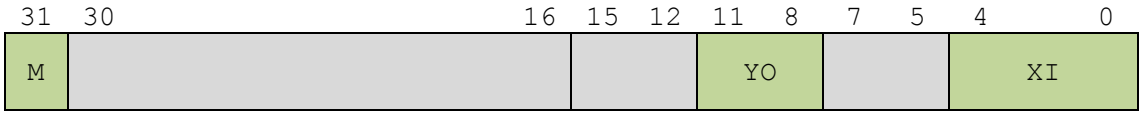
*在 Nexys3 开发板中，SW[7:0]为拨动开关，SW[8]为下按钮，SW[9]为上按钮，SW[10]为右按钮，SW[11]为左按钮，SW[12]为中按钮。

- 0x04 ~ 0x07 为按钮的状态，该寄存器只读。

BTN[19:0]：按钮的状态。若为阵列式键盘，则仅在普通模式有效。

*在 Sword 开发板中，BTN[0]为 X0Y0，BTN[1]为 X0Y1，以此类推，BTN[19]为 X4Y3。

- 0x08 ~ 0x0B 控制阵列式键盘。

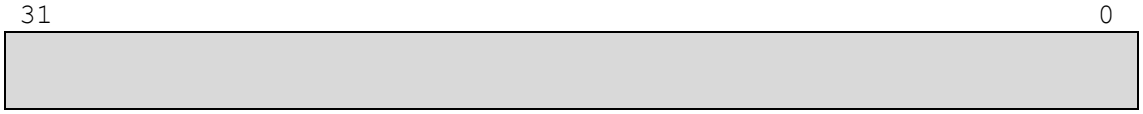


M：模式位，置 0 为普通模式，置 1 为阵列扫描模式。

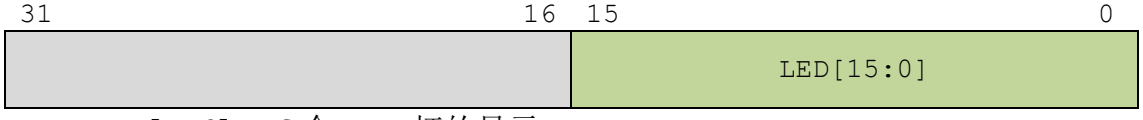
YO[3:0]：阵列式键盘扫描输出。

XI[4:0]：阵列式键盘扫描输入。

- 0x0C ~ 0x0F 为板级输入设备保留。

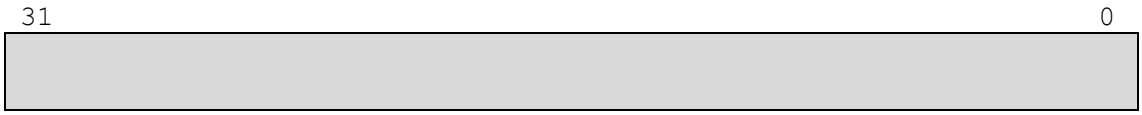


- 0x10 ~ 0x13 控制 LED 灯的状态。



LED[15:0]：16 个 LED 灯的显示。

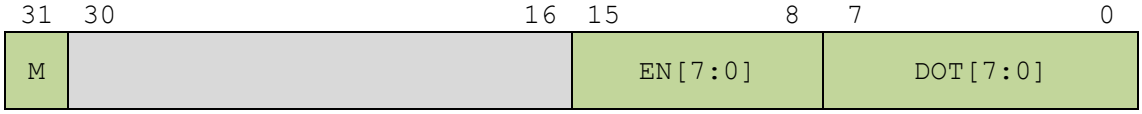
- 0x14 ~ 0x17 为板级输出设备保留。



- 0x18 ~ 0x1B 控制七段数码管的文本显示数据。

DISP：七段数码管显示的文本数据。

- 0x1C ~ 0x1F 控制七段数码管的状态。

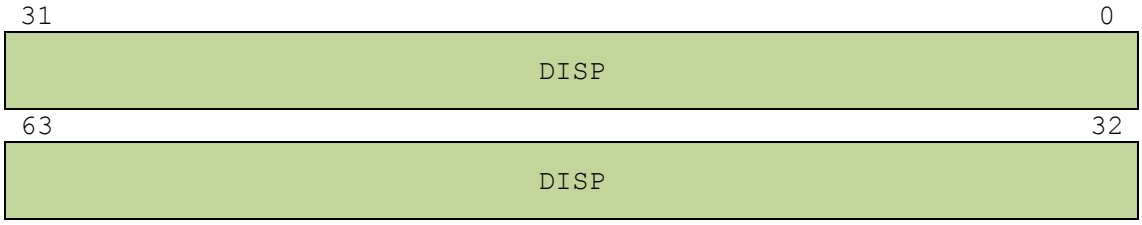


M：模式位，置 0 为文本模式，置 1 为图形模式。

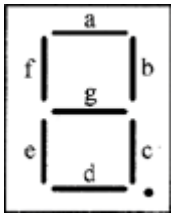
EN[7:0]：七段数码管使能位，供文本模式使用。

DOT[7:0]：七段数码管小数点的显示，供文本模式使用。

- 0x20 ~ 0x27 控制七段数码管的图形显示数据。



DISP：七段数码管显示的图形数据。

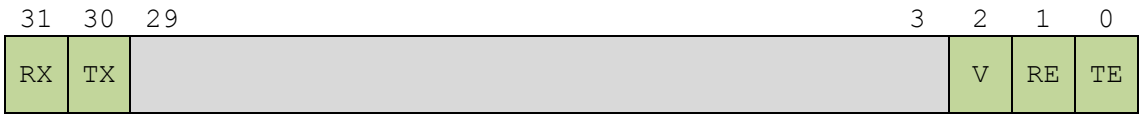


图形数据每 8 个 bit 为一组，高位的组控制高位的数码管。在每个组内，对应上图，bit0 表示 a，bit1 表示 b，以此类推，bit6 表示 g，bit7 表示小数点。

7.3 键盘

这个键盘指 PS/2 键盘，当按下按键时键盘会发送相应按键的扫描码，需要由软件转换成 Ascii 码，同时软件可以向键盘发送控制指令控制键盘的工作方式。由于键盘的数据传输速度很慢，且键盘自身一般有 16 字节的缓冲区，因此键盘控制器不使用数据缓冲区。键盘控制器在接收到数据，或者数据收发出错的时候产生中断（发送控制指令时不产生中断，因为键盘总是会返回一个响应数据）。

- 0x00 ~ 0x03 用于键盘状态查询，只读。



RX：数据接收标志，当正在接收数据时置位。

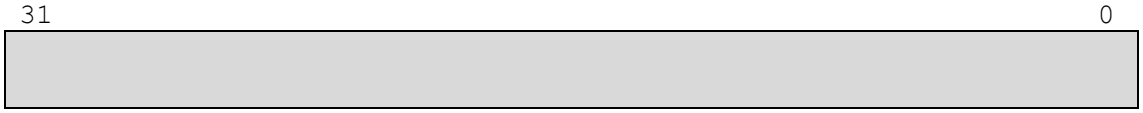
TX：数据发送标志，当正在发送数据时置位。

V：数据有效标志，当已从键盘接收到数据时置位，当 CPU 将数据读走时复位。

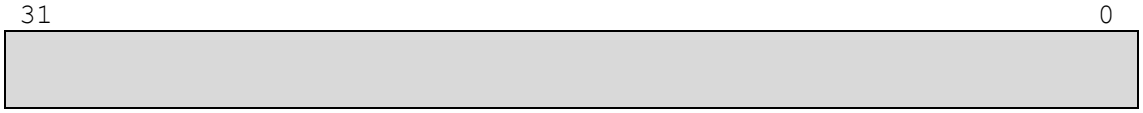
RE：读取出错，当从键盘读取数据发生异常时置位，当 CPU 尝试读取键盘数据时复位。

TE：写入出错，当向键盘写入数据发生异常时置位，当 CPU 尝试向键盘写入数据时复位。

- 0x04 ~ 0x07 保留。



- 0x08 ~ 0x0B 保留。



- 0x0C 用于数据收发。

数据的发送和接收使用同一个地址，只支持字节操作。

键盘的指令控制流程中，每次向键盘写入控制指令时都需要等待键盘的回应，因此没有连续向该地址写入数据的需求，连续写入的多余数据会被丢弃。同时，连续从该地址读取数据并不会触发溢出中断，只会读出相同的数据。

另外需要注意的是，目前绝大多数开发板都通过一个内置的单片机实现 USB 键盘信号到 PS2 键盘信号的转换，但是这个转换并不完美，许多情况下可能并不支持发送数据到键盘。

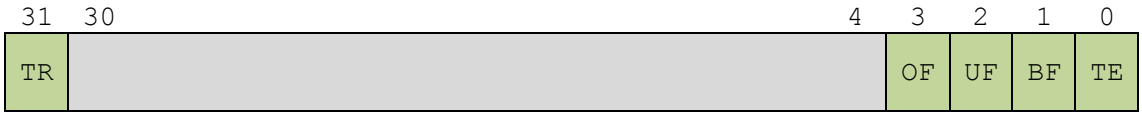
7.4 鼠标（暂无）

7.5 SPI

SPI 控制器自带一个较大的数据缓冲区，由于 SPI 总线的数据收发总是同时

进行，因此发送和接收将共用这个缓冲区。SPI 控制器在数据传输完毕或缓冲区溢出（包括上溢和下溢）时产生中断。

- 0x00 ~ 0x03 用于 SPI 状态查询，只读。



TR：数据传输标志，当正在传输数据时置位。

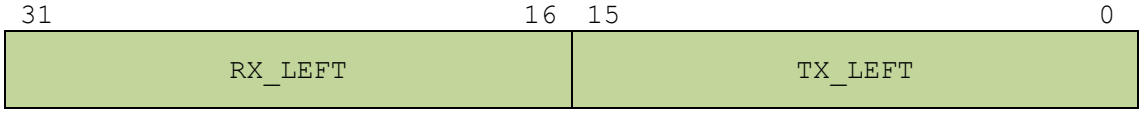
OF：缓冲区上溢中断，当 CPU 向控制器写入过多数据时置位。当控制寄存器重新赋值时复位。

UF：缓冲区下溢中断，当 CPU 向控制器读取过多数据时置位。当控制寄存器重新赋值时复位。

BF：缓冲区为满标志，当缓冲区为满时置位。

TE：数据传输完毕中断，当数据传输完毕时置位。

- 0x04 ~ 0x07 用于缓冲区状态查询，只读。

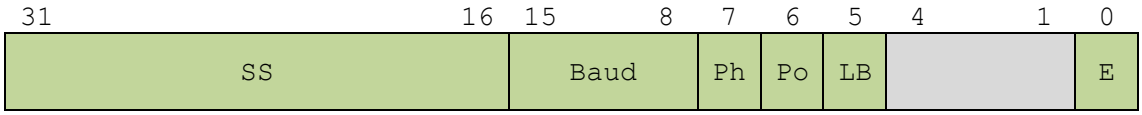


RX_LEFT：缓冲区内已接收但未取走的数据大小，即当前还可以从缓冲区读取的有效字节数。

TX_LEFT：缓冲区内还可以储存的待发送数据大小，即当前还可以向缓冲区写入的有效字节数。

在 SPI 控制器空闲状态下，RX_LEFT 与 TX_LEFT 之和即为缓冲区总大小。在任何状态下，缓冲区总大小减去这两者之和，即为当前已储存待发送的数据大小。

- 0x08 ~ 0x0B 用于 SPI 控制。



SS：SPI 从设备选择。31-17 留空，16 表示 SD 卡。

Baud：波特率设置，这里需要设置的值是 5e6/baudrate-1。如，0 表示 5M 的波特率，249 表示 20K 的波特率。

Ph：时钟相位 CPHA 设置。1 表示传输时在偶数个时钟跳变沿采样，0 为奇数个时钟跳变沿采样。

Po：时钟极性 CPOL 设置。1 表示空闲时时钟为高电平，0 为低电平。

LB：数据大小端设置。1 表示小端数据传输，即低位在前，0 为高位在前。

E：SPI 使能位，该位置 1 时 SPI 控制器开始工作。

一般情况下，使用 Ph=0、Po=0、LB=0 居多。

需要注意的是，修改该寄存器会清空数据缓冲区，并重置 SPI 状态。因此 CPU 在设置该寄存器前应检查数据是否发送完毕，即使错误操作，由于这属于 CPU 的明确指令，将不会产生中断。

- 0x0C 用于数据收发

数据的发送和接收使用同一个地址，只支持字节操作。

7.6 UART

UART 控制器自带一定大小且相互独立的发送缓冲区和接收缓冲区，以尽量减轻 CPU 的负担。UART 控制器在发送缓冲变空，或者接收缓冲超过 3/4，或者数据接收时出现校验错误，或者接收缓冲存在数据且超过 100ms 没有读取操作，或者发送缓冲、接收缓冲溢出（包括上溢和下溢）时产生中断。

- 0x00 ~ 0x03 用于串口状态查询，只读。

31	30	29 … 6	5	4	3	2	1	0
RX	TX		ER	RO	RU	TO	RI	TI

RX：数据接收标志，当正在接收数据时置位。

TX：数据发送标志，当正在发送数据时置位。

ER：校验错误中断，当数据接收过程中校验出错时置位。当控制寄存器重新赋值时复位。

RO：接收缓冲区上溢中断，当 CPU 没有及时取走接收到的数据时置位。当控制寄存器重新赋值时复位。

RU：接收缓冲区下溢中断，当 CPU 向控制器读取过多数据时置位。当控制寄存器重新赋值时复位。

TO：发送缓冲区溢出中断，当 CPU 向控制器写入过多数据时置位。当控制寄存器重新赋值时复位。

RI：数据接收中断，当接收缓冲区非空时置位。

TI：数据发送中断，当发送缓冲区为空时置位。

- 0x04 ~ 0x07 用于缓冲区状态查询，只读。

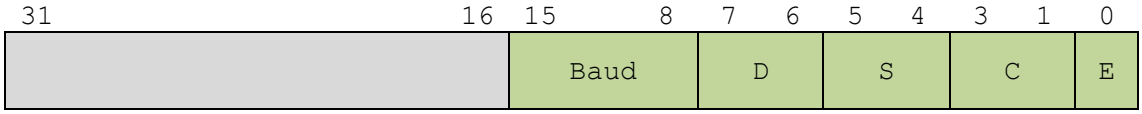
31 … 16	15 … 0
RX_LEFT	TX_LEFT

RX_LEFT：接收缓冲区内未取走的数据大小，即当前还可以从缓冲区读取的有效字节数。

TX_LEFT：发送缓冲区内未使用的空闲空间大小，即当前还可以向缓冲区写入的有效字节数。

在 UART 控制器空闲状态下，TX_LEFT 的值即为发送缓冲区总大小。

- 0x08 ~ 0x0B 用于串口控制。



Baud：波特率设置，这里需要设置的值是 1.25e6/baudrate-1。如，10 表示 115200 的波特率，129 表示 9600 的波特率。

D：数据位长度，2'b00 表示 8 位数据位，2'b01 表示 7 位数据位，2'b10 表示 6 位数据位，2'b11 表示 5 位数据位。

S：停止位长度，2'b00 表示 1 位停止位，2'b01 表示 1.5 位停止位，2'b10 表示 2 位停止位。

C：校验位类型，3'b000 表示不使用校验位，3'b100 表示奇校验，3'b101 表示偶校验，3'b110 表示 MARK 校验（校验位始终为 1），3'b111 表示 SPACE 校验（校验位始终为 0）。

E：串口使能位，该位置 1 时串口开始工作。

需要注意的是，修改该寄存器会清空发送缓冲区和接收缓冲区，并重置串口状态。因此 CPU 在设置该寄存器前应检查数据是否发送完毕，即使错误操作，由于这属于 CPU 的明确指令，将不会产生中断。

- 0x0C 用于数据收发。

数据的发送和接收使用同一个地址，只支持字节操作。

7.7 并口（暂无）

7.8 USB 口（暂无）

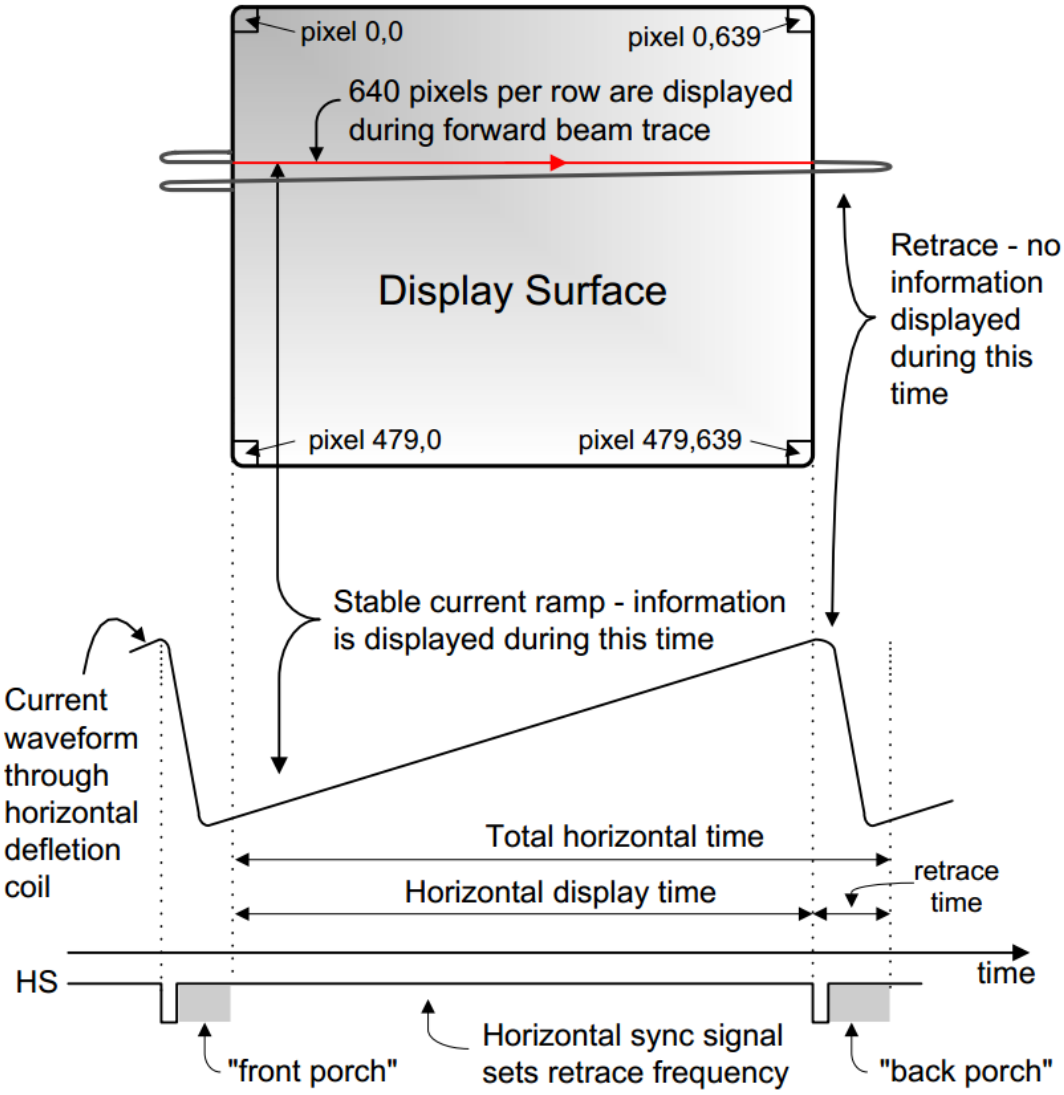
7.9 网口（暂无）

第8章 外设实现

8.1 VGA

VGA（Video Graphics Array）是广泛使用的一种视频传输标准，最早是指640×480 分辨率的显示模式，后来基于此发展出了很多不同的分辨率，由于使用的是相同的接口，也统称为 VGA。

VGA 的时序主要由行同步信号（HS）和场同步信号（VS）控制。最早是为了 CRT 显示器逐行扫描的显示原理而设计的，现在 LCD 显示器的显示原理已经完全不同了，但也都兼容这种 VGA 的信号格式。标准 VGA 信号的时序图如下：



VGA 信号在每个行周期或场周期中，同步脉冲前后各有一段空余时间，分别叫做前沿（front porch）和后沿（back porch）。VGA 在不同分辨率下，其同步时间，显示时间，前沿和后沿时间都是不同的。具体的 VGA 时序标准，可以参考 VESA（Video Electronics Standards Association）制定的 DMT（Display Monitor Timing）文档。

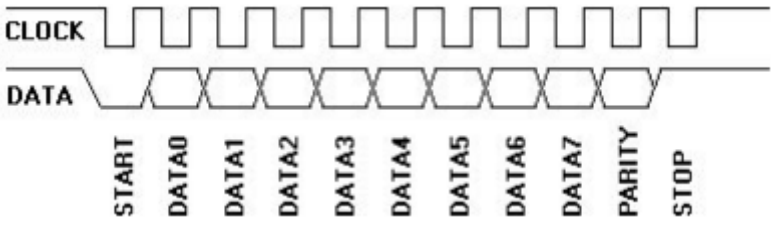
为了实现对 VGA 不同分辨率的支持，我们需要不同的时钟频率，使用 FPGA 中的专有部件 DCM_CLKGEN 可以很方便地实现多种时钟频率的输出。不同于一般的 DCM_SP，DCM_CLKGEN 支持动态调整 M 和 D 的值，从而动态调整输出时钟的频率。

在 Spartan-6 上编写 VGA 模块时发现，DCM_CLKGEN 的 PROGCLK 引脚只能由 FPGA 上半部分的时钟缓冲 BUFG 驱动，而默认的 GCLK0（V10）针脚位于 FPGA 底部的中间，通过全局时钟资源只能到达下半部分的两个 BUFG 中，因此如果使用 V10 时钟直接驱动 DCM_CLKGEN 的 PROGCLK 引脚，会将 V10 的 BUFG 放置到 FPGA 上半部分，从而产生时钟偏移。解决的方式是将 V10 输入的时钟先经过一个 DCM，输出时钟通过上半部分的 BUFG 去驱动目标 DCM_CLKGEN 的 PROGCLK 引脚。（DCM_CLKGEN 的 CLKIN 引脚无此约束。）

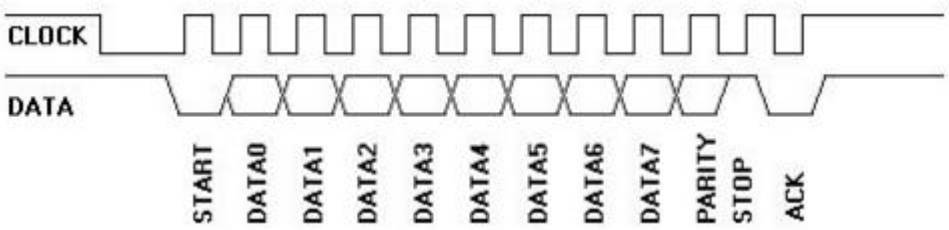
8.2 PS2

PS2 接口主要用于老式的键盘和鼠标，时序比较简单且速度较慢。

PS2 主要由 1 根时钟线和 1 根数据线组成。在没有数据传输时，两者均为高电平。在数据传输过程中，时钟线的频率一般在 15kHz 左右，由设备提供，主机在时钟线为低电平时采样或者输出数据。



上图是设备发送数据到主机的时序图。设备先拉低数据线再拉低时钟线表示一个数据帧的开始，之后发送 8 位数据位，1 位奇校验位和 1 位停止位。主机需在时钟线为低电平时采集数据，在设备发送数据到主机的过程中，主机可随时拉低时钟线来中断数据传输。



上图是主机发送数据到设备的时序图。主机首先拉低时钟线至少 100 微秒表示需要发送数据，然后拉低数据线，之后放开时钟线。之后设备会产生时钟信号，主机在时钟线为低电平时改变数据线，依次发送 8 位数据位，1 位奇校验位和 1 位停止位。之后设备会返回一个应答位，若主机在发送完停止位后 100 微秒内没有接收到应答位，则会产生一个错误。

主机发送数据到键盘或鼠标，发送的数据称为指令，如控制键盘上指示灯的点亮与否，控制按键一直按住时的机打延时和机打频率等。更详细的键盘指令及其格式，可以参考 PS2 技术参考（在线地址：http://www.computer-engineering.org），PS2 键盘发送的扫描码，一般遵循第二套扫描码集（http://www.computer-engineering.org/ps2keyboard/scancodes2.html）。

在 Spartan-6 上编写 PS2 模块的时候，特别注意需要在 UCF 引脚约束文件中给 PS2 的两个引脚加上“PULLUP”约束，表明需要连接上拉电阻。另外经过多次试验发现，Spartan-6 上的 PS2 信号是由其上的 PIC24FJ192 芯片提供的，由于该芯片的时钟和我们的 PS2 模块时钟不同步，在使用 PS2 的两根信号之前必须要使用寄存器进行缓存后输入，而且 PS2 模块必须及时取走键盘或鼠标产生的数据，若拉低时钟线的时间过长导致累计待读取的数据超过 15 个字节，PIC24FJ192 芯片可能会出现内部错误，只能通过重启整个开发板解决。

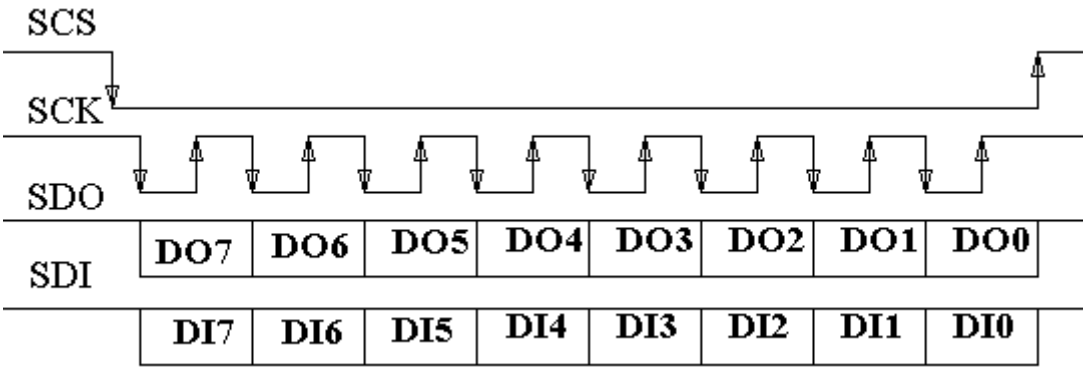
8.3 串口

串行接口（Serial Interface）是最简单的一种数据通信方式。串行通信最少只需要一根传输线即可完成，传输距离长，成本低，但是速度较慢。

串口可以分为同步串口 SPI（Serial Peripheral interface）和异步串口 UART（Universal Asynchronous Receiver/Transmitter）两种。一般情况下，两者均使用两根数据线，一根负责数据发送，另一根负责数据接收，来实现全双工模式。不同的是，同步串口需要一个同步时钟来控制所有的串口设备，异步串口则需要各个串口设备自行产生时钟，并相互约定串口的比特率。

8.3.1 同步串口

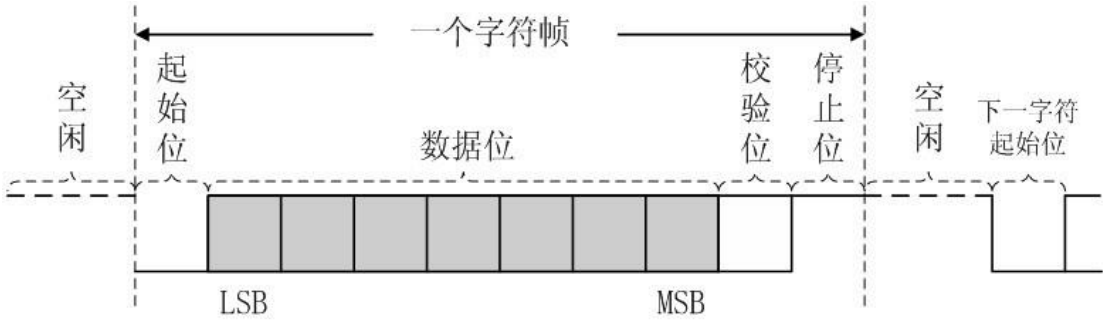
同步串口一般包括片选信号线，时钟信号线，数据发送线，数据接收线这 4 根线。



同步串口严格区分主设备和从设备，时钟信号完全由主设备控制，因此主设备可以随时控制整个数据传输的速率。

8.3.2 异步串口

异步串口一般包括数据发送线，数据接收线这 2 根线。



异步串口不区分主设备和从设备，通信双方处于完全相同的地位，需要事先约定好数据传输的波特率，数据位数，校验位类型，停止位长度等参数才可以开始数据传输。最常用的波特率是 9600 和 115200。

由于异步数据传输过程中没有时钟信号进行相位同步，而只能依赖于起始位进行对齐，串口控制器在处理数据接收时，最好对同一个 bit 进行多次采样，将出现次数较高的电平值作为此 bit 的数据值。

由于历史原因，异步串口的波特率大多是 1200 的整数倍，无法与 FPGA 的内部时钟频率相契合，导致在连续收发数据时可能导致相位差扩大。比较好的解决方式是在检测最后一个停止位的时候“尽快停止”，即只要采样到足够的次数以明确当前是一个停止位时，立即终止本次数据接收流程，并做好下一个起

始位到来的准备。

异步串口的停止位可能被设定为 1.5 位。如果设备无法产生半个位长的数据，可以在发送时使用 2 位停止位而在接收时使用 1 位停止位。

在 Spartan-6 上编写 UART 模块时发现，当串口线未连接时，FPGA 刚刚编程完成后可能会在引脚上收到噪声信号，因此建议在确定不使用串口时关闭其使能，或者在 UCF 中使用 PULLUP 约束，同时进行 RX 信号的去抖动处理。不过相比于板上的按键等设备，串口线的防抖阈值应该设得足够小，以保证不会影响到正常的数据传输。