# 2026 集创赛龙芯中科杯
# 初赛发布包说明手册

# 赛题讲义版权声明

# 目 录

第 1 章 实验环境搭建 ..... 1
1.1 实验环境介绍 ..... 1
1.2 安装实验所需软件 ..... 1
1.2.1 安装 vivado 2019.2 ..... 1
1.2.2 使用 WSL2 安装 Ubantu 22.04.5 ..... 1
1.2.3 配置 SDK 中的工具链与 C 库 ..... 3
1.3 远程 FPGA 实验平台 ..... 4
第 2 章 阶段任务一 搭建基础 SoC ..... 5
2.1 实验任务 ..... 5
2.2 实验所用 IP 简介 ..... 5
2.2.1 OpenLA500 处理器核 ..... 5
2.2.2 AXI4 总线 ..... 5
2.2.3 SRAM 控制器 ..... 6
2.2.4 UART ..... 6
2.2.5 时钟与复位 ..... 16
2.3 实验所用嵌入式软件 ..... 17
2.3.1 在启动引导函数中进行串口初始化 ..... 17
2.3.2 串口输出字符 ..... 18
2.4 实验流程 ..... 18
2.4.1 搭建基础 SoC 顶层 ..... 18
2.4.2 进行功能仿真和 FPGA 验证 ..... 24
第 3 章 阶段任务二 外部中断控制器实验之弹球游戏 ..... 30
3.1 实验任务 ..... 30
3.2 实验所用 IP 简介 ..... 30
3.2.1 Confreg ..... 30
3.2.1 DVI 控制器 ..... 31
3.3 实验所用嵌入式软件 ..... 35
3.3.1 中断测试软件 ..... 35
3.3.2 弹球游戏软件 ..... 37
3.4 实验流程 ..... 39
3.4.1 添加外部中断控制器 ..... 39
3.4.2 添加 DVI 控制器 ..... 40

3.4.3 进行功能仿真和 FPGA 验证 ..... 41
附录一 向 Confreg 设计中添加外设寄存器 ..... 46

# 图目录

图 1-1 vscode 添加 WSL 插件 ..... 2
图 1-2 vscode 连接到 WSL ..... 2
图 1-3 vscode 打开文件夹 ..... 3
图 1-4 完成工具链安装 ..... 4
图 2-1 UART 数据传输格式 ..... 7
图 2-2 APB 状态机 ..... 15
图 2-3 APB 写操作 ..... 15
图 2-4 APB 读操作 ..... 16
图 2-5 SoC 架构 ..... 19
图 2-6 SoC 时钟与复位网络 ..... 19
图 2-7 运行 make 进行编译 ..... 24
图 2-8 vivado 创建工程 ..... 25
图 2-9 vivado 进行 RTL 分析 ..... 25
图 2-10 vivado 运行功能仿真 ..... 26
图 2-11 hello_word 功能仿真通过 ..... 26
图 2-12 添加 debug 信号至波形窗口 ..... 27
图 2-13 反汇编文件 ..... 27
图 2-14 远程 FPGA 下载 SRAM ..... 28
图 2-15 设置串口波特率 ..... 28
图 2-16 远程 FPGA 运行 Hello World 例程 ..... 29
图 3-1 DVI-I 端口定义 ..... 32
图 3-2 显示器扫描方式 ..... 33
图 3-3 行扫描周期 ..... 33
图 3-4 场扫描周期 ..... 33
图 3-5 DVI-A 时序图 ..... 34
图 3-6 int_test 串口输出仿真结果 ..... 42

图 3-7 int_test DVI 输出仿真结果 ..... 42
图 3-8 int_test FPGA 验证结果 ..... 43
图 3-9 弹球游戏 ..... 44

## 表目录

表 2-1 UART 寄存器总览 ..... 7
表 2-2 Receiver Buffer Register 位定义 ..... 8
表 2-3 Transmitter Holding Register 位定义 ..... 8
表 2-4 Interrupt Enable Register 位定义 ..... 9
表 2-5 Interrupt Identification Register 位定义 ..... 9
表 2-6 FIFO Control Register 位定义 ..... 10
表 2-7 Line Control Register 位定义 ..... 10
表 2-8 Line Status Register 位定义 ..... 11
表 2-9 Divisor Latche LSB 位定义 ..... 13
表 2-10 Divisor Latche MSB 位定义 ..... 13
表 2-11 APB SLAVE 信号列表 ..... 14
表 3-1 Confreg 寄存器总览 ..... 30
表 3-2 DVI 寄存器总览 ..... 35
表 3-3 外部中断控制器信号列表 ..... 39
表 3-4 外部中断控制器寄存器总览 ..... 40

# 第 1 章 实验环境搭建

## 1.1 实验环境介绍

SoC 课程实验文件夹 la32r_soc 下包含四个文件夹，分别是 rtl、sim、fpga、sdk。各个文件夹的功能简介如下

rtl:硬件设计文件，包含提供给后续实验所用的各类 ip，以及 SoC 的顶层文件。

sim:存放功能仿真用到的 testbench。

fpga:存放设计约束，以及 FPGA 工程目录。

sdk:软件开发工具包，提供功能仿真以及 FPGA 上板时所用测试软件及其编译环境。

## 1.2 安装实验所需软件

SoC 实验所用 IDE 主要是 vivado，可以直接在 windows 下安装，推荐版本 2019.2。实验所用软件需要在 Linux 环境下进行编译，推荐开发环境是 windows+wsl2，将软件编译需要的 gnu 工具链放在 wsl2 运行的 ubantu 操作系统中。

### 1.2.1 安装 vivado 2019.2

直接在 windows 下安装，具体安装教程可参见下述链接：

https://bookdown.org/loongson/_book3/appendix-vivado-install.html#vivado-%E7%AE%80%E4%BB%8B

### 1.2.2 使用 WSL2 安装 Ubantu 22.04.5

打开 Microsoft Store，搜索 WSL，选择 Ubuntu22.04.5 进行安装；具体流程可参见下述链接：

https://zhuanlan.zhihu.com/p/692671957

为方便在 WSL 中进行代码编辑，可以在 windows 下安装 vscode，添加 wsl 插件，在搜索框中搜出 wsl 后直接点击 Install 即完成插件安装。安装完毕后在左下角看到一个按钮（一个>和一个<组成），点击在中间选择"连接到 wsl",然后 Vscode 会弹出一个新窗口，左下角显示 WSL 子系统名称，如：WSL:Ubuntu。最后点击左侧文件夹图标，资源管理器，打开文件夹，开始代码编辑。

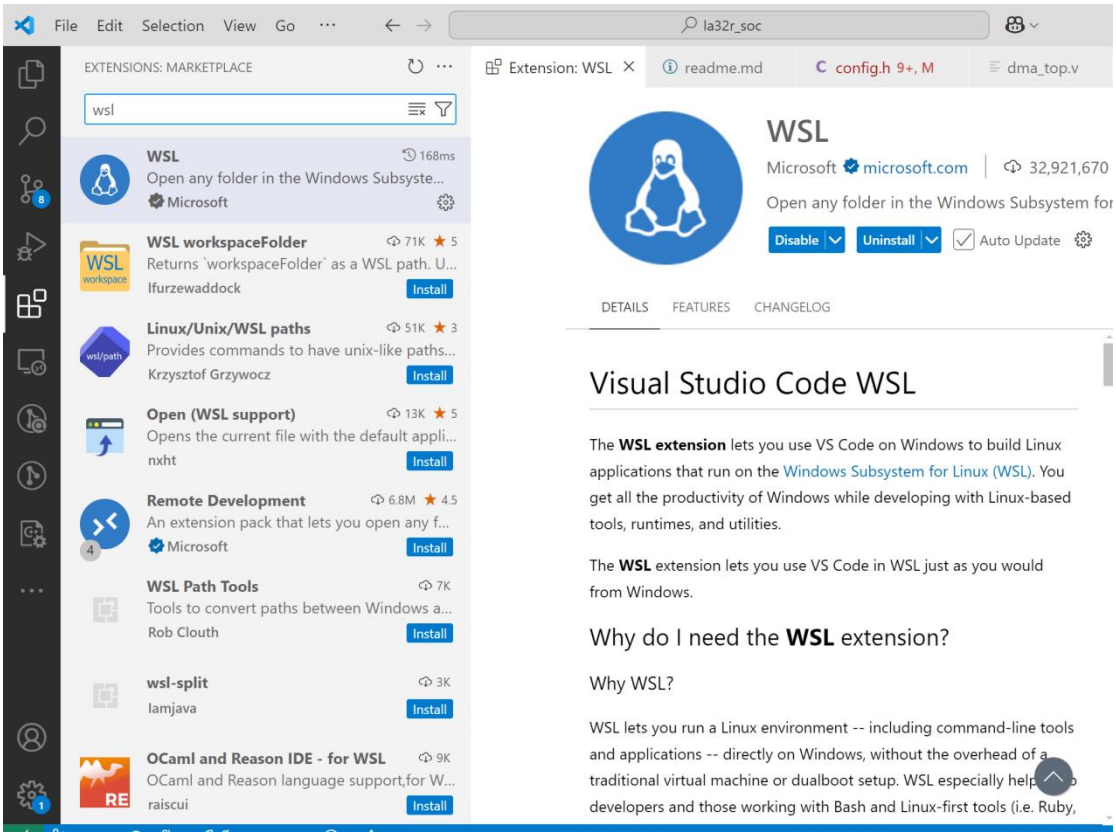



**图 1-1　vscode 添加 WSL 插件**

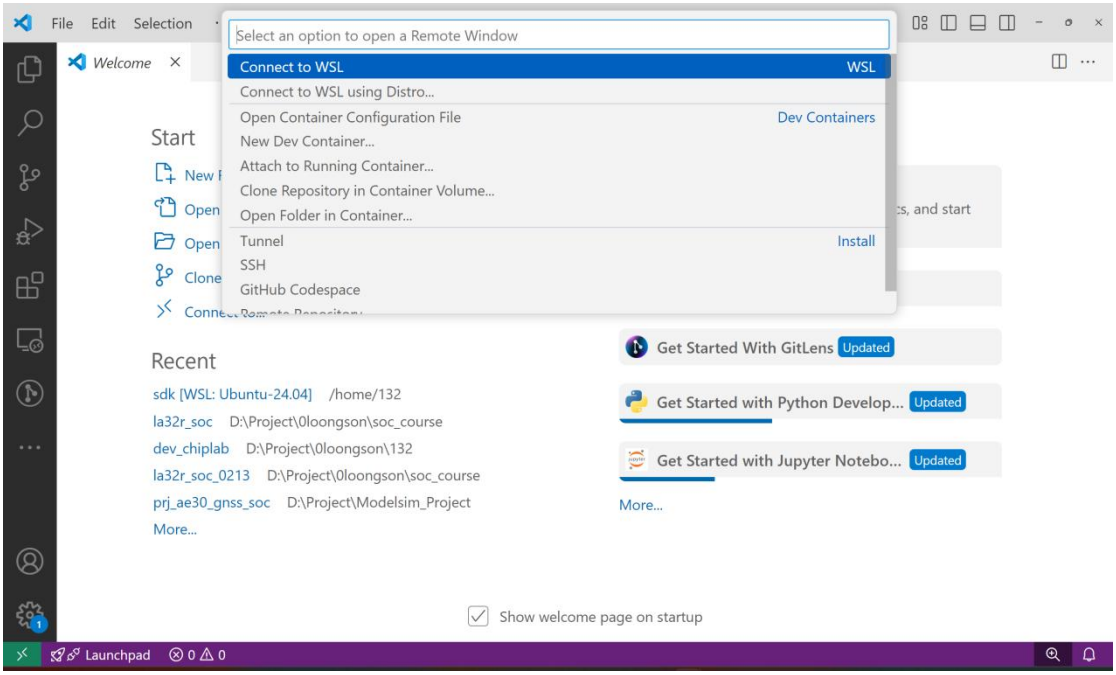



**图 1-2　vscode 连接到 WSL**

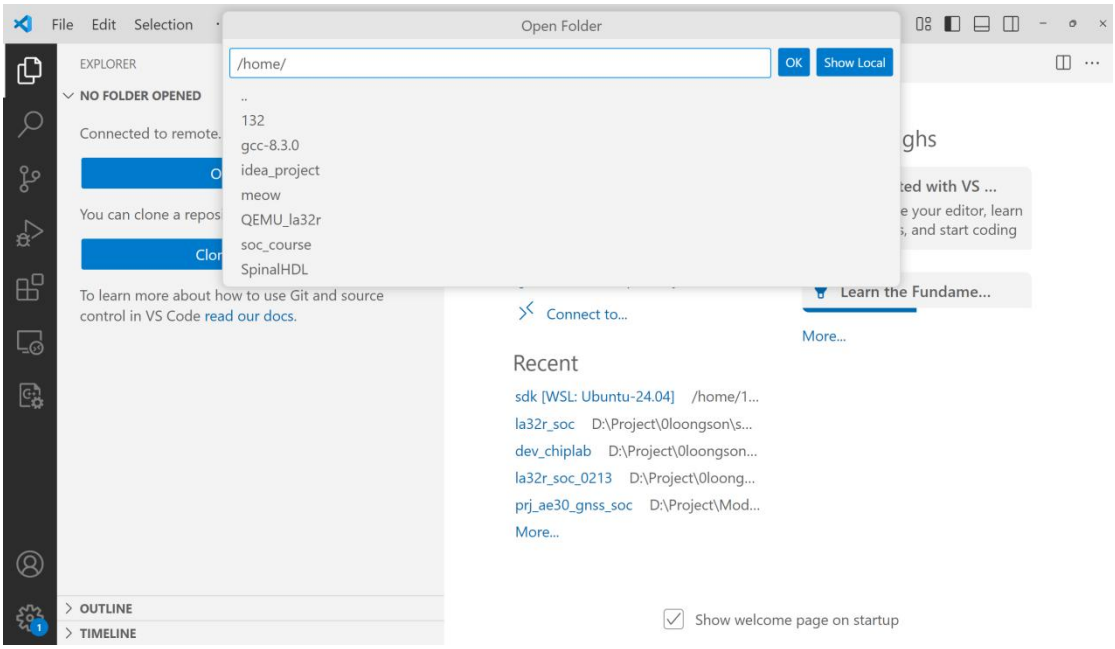



**图 1-3　vscode 打开文件夹**

### 1.2.3　配置 SDK 中的工具链与 C 库

完成 Ubantu 的安装后，将实验文件夹中的 sdk 目录拷贝至 Ubantu 的目录下。在 sdk 目录下有两个文件夹，software 文件夹是包含实验所用软件源码，toolchains 文件夹中包含 gnu 工具链和 c 库用于支持编译。下面具体介绍如何进行 gnu 工具链和 c 库的安装。

首先修改 la32r_soc/sdk/toolchains/init.sh 文件，其中一行代表发布包文件夹在 windows 中的位置，如果发布包直接放在 D 盘下则为下列默认值。否则根据实际情况进行修改。（本文中的代码行第一个字符$代表这是一行可执行的命令行代码,复制粘贴时无需复制该字符）

```
$ sed -i '$a\export LA32RSOC_WINDOWS_HOME="/mnt/d/la32r_soc"' ~/.bashrc
```

该变量控制测试软件编译结束后将 mif（内存初始化）文件放在 windows 下的何处，mif 文件用于 windows 中 vivado 的仿真。

完成路径修改后在 WSL 的终端中进入 la32r_soc/sdk/toolchains 目录，直接运行 init.sh 脚本即可完成工具链和 C 库的安装。执行的命令如下

```
$ ./init.sh
```

完成脚本执行后会在 toolchains 目录下看到多出三个目录：loongson-gnu-toolchain-8.3-x86_64-loongarch32r-linux-gnusf-v2.0、newlib、picolibc。在终端中使用下述命令检查工具链是否可用：

```
$ loongarch32r-linux-gnusf-gcc --version
```

应能看到如下输出，证明安装成功：

```
root@LAPTOP5BEO50RJ:/home/test1# loongarch32r-linux-gnusf-gcc --version
loongarch32r-linux-gnusf-gcc (LoongArch GNU toolchain LA32 v2.0 (20230903)) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

**图 1-4　完成工具链安装**

如果终端没有输出版本信息，一般是 bashrc 在当前终端未生效。在终端中输入下述命令后再进行检查即可。

```
$ source ~/.bashrc
```

## 1.3　远程 FPGA 实验平台

Vivado 综合实现、bit 生成后，将 bit 文件上传至远程 FPGA 平台，并向 BaseRAM 下载软件 bin 文件，即可看到测试结果。

# 第 2 章　阶段任务一 搭建基础 SoC

## 2.1　实验任务

基于实验环境提供的处理器核、总线互联、SRMA 和 UART 的 IP，搭建一个基础 SoC，使用 Vivado 进行仿真、综合实现后通过 FPGA 上板测试。所搭建的基础 SoC 应能完成经典程序 Hello World 的正确执行，在 FPGA 串口中输出相应字符串。

## 2.2　实验所用 IP 简介

该实验中用到的四款 IP 分别是：OpenLA500 处理器核、AXI4 总线、片上 SRAM 以及 UART。下面分别介绍这四款 IP 具体信息。

### 2.2.1　OpenLA500 处理器核

源码位于 rtl/ip/open-la500 目录下。

openLA500 是一款实现了龙芯架构 32 位精简版指令集（loongarch32r）的处理器核。其结构为单发射五级流水，分为取指、译码、执行、访存、写回五个流水级。并且含有两路组相连结构的指令和数据 cache；32 项 tlb；以及简易的分支预测器。此外，处理器核对外为 AXI 接口，容易集成。

OpenLA500 已经过流片验证，.13 工艺下频率为 100M，dhrystone，coremark 分数分别为 0.78 DMIPS/MHz(指令数有点高)，2.75 coremark/Mhz。软件方面，uboot、linux 5.14、ucore、rt-thread 等常用工具及内核已完成对 openLA500 的适配。

### 2.2.2　AXI4 总线

源码位于 rtl/ip/ Bus_interconnects 目录下。

其中 AxiCrossbar_2x8.v 是 AXI4 总线矩阵，可以连接两个 Master 和八个 Slave。八个 Slave 的地址分布为：

axiOut_0 : 1c000000 – 1c7fffff 8Mbytes (SRAM)
axiOut_1 : 00000000 – 007fffff 8Mbytes (reserved)
axiOut_2 : 1f000000 – 1f0fffff 1Mbytes (apb-UART)
axiOut_3 : 1f100000 – 1f0fffff 1Mbytes (DVI)
axiOut_4 : 1f200000 – 1f0fffff 1Mbytes (Confreg)
axiOut_5 : 1f300000 – 1f0fffff 1Mbytes (DMA)
axiOut_6 : 1f400000 – 1f0fffff 1Mbytes (FFT)
axiOut_7 : 1f500000 – 1f0fffff 1Mbytes (reserved)

其中 axiOut_0 用来连接片上 SRAM，axiOut_1 保留，axiOut_2 连接 axi 转 apb 桥后连接 UART，axiOut_3 连接 dvi 显示控制器，axiOut_4 连接 Confreg（包含按键、拨码开关、LED、数码管、中断控制器），axiOut_5 连接 DMA 的 slave 端口，axiOut_6 连接 FFT 加速器，axiOut_7 保留。

OpenLA500 处理器核（顶层模块 core_top 位于 mycpu_top.v）与 AxiCrossbar 连接前还需要先连接 axi_wrap（axi_wrap.v）和 Axi_CDC（Axi_CDC.v）。其中 axi_wrap 确保进入处理器核的总线 r 通道和 b 通道的值在 valid 不为高电平时保持全 0。Axi_CDC 是异步 FIFO 模块，用于将处理器核时钟域的信号同步到总线时钟域。

片上 SRAM 与 AxiCrossbar 连接前需要接 soc_axi_sram_bridge（soc_axi_sram_bridge.v），实现 AXI 协议转 SRAM 读写信号。

UART 与 AxiCrossbar 连接前需要接 axi2apb_bridge（rtl/ip/APB_UART/axi2apb.v），实现 AXI 协议转 APB 协议。

### 2.2.3 SRAM 控制器

源码位于 rtl/ip/ram_wrap/axi_wrap_ram_sp_external.v，用于控制远程 FPGA 平台开发板上的两块 4MB SRAM。外部 SRAM 的最小读写周期是 10ns，为保证读写操作的稳定性，请保证 SRAM 控制器的时钟频率小于等于 50MHz。控制器根据地址的 bit22 判断是 BaseRAM（bit22 为 0）或者 ExtRAM（bit22 为 1），从而可以控制整个 8MB 的存储空间。

在 rtl/ip/ram_wrap 文件夹下还有其他文件，这里给出其用途：

- fpga_sram_dp.v:片上伪双端口 SRAM，使用综合推导式描述。
- fpga_sram_sp.v:片上单端口 SRAM，使用综合推导式描述。
- axi_wrap_ram_dp.v:连接了 axi 转 sram 桥的伪双端口 sram（一个端口仅用于读取，另一端口仅用于写入）。
- axi_wrap_ram_sp.v:连接了 axi 转 sram 桥的单端口 sram。

### 2.2.4 UART

源码位于 rtl/ip/APB_UART，其中串口模块顶层位于 URT/uart_top.v，该串口模块是 APB 协议接口的，在 rtl/ip/APB_UART/axi_uart_controller.v 中提供了已经加入 axi 转 apb 桥的顶层模块 axi_uart_controller。

#### 2.2.4.1 UART 传输协议

UART（Universal Asynchronous Receiver and Transmitter）：通用异步收发器是一种通用串行数据总线，用于异步通信，其中数据格式和传输速度是可配置的，可通过逐个发送和接收数据位传输数据，并用起始位和停止位进行帧定界。

该总线双向通信，可以实现全双工传输。

UART 协议的输入和输出均为 1 比特位宽的数据通路（RX_i、TX_o）用于传输数据，UART 传输格式为：

在 UART 协议空闲时信号线保持高电平。

一帧数据包括起始位、数据位、校验位、停止位。

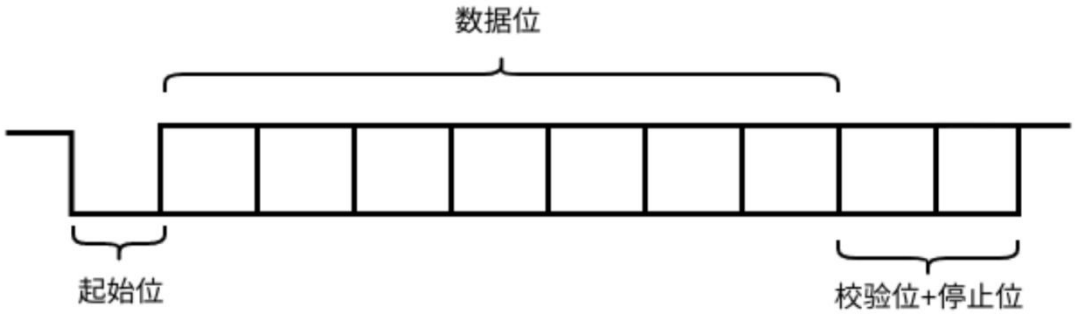



**图 2-1 UART 数据传输格式**

- 起始位：在帧数据起始阶段置为逻辑 0，表示一帧数据传输的开始
- 数据位：在起始位后传输帧的数据位，根据不同的传输配置数据位的比特数可配置为 5、6、7、8 来构成一帧的字符。
- 奇偶校验位：根据不同的配置，在数据位传输后传输后传输数据位中 1 的位数是奇数还是偶数，以校验帧数据传输的正确性。
- 停止位：用于标识一帧数据的传输结束的标志，通过不同的配置，停止位数可配置为 1 位、1.5 位、2 位的高电平。
- 空闲位：处于置逻辑 1 状态，表示当前无数据帧传输。

#### 2.2.4.2 UART 寄存器描述

**表 2-1 UART 寄存器总览**

| 名称 | 地址 | 描述 |
|---|---|---|
| Receiver Buffer Register | 1f000000 | 对其发起读请求，可以读经串并转换后的输入数据帧 |
| Transmitter Holding Register | 1f000000 | 发起写操作可将待并行数据存入缓存，由控制器自动将数据转换并发送 |
| Interrupt Enable Register | 1f000001 | UART 中断的使能/屏蔽寄存器 |

| | | |
|---|---|---|
| Interrupt Identification Register | 1f000002 | 获取中断信息的寄存器 |
| FIFO Control Register | 1f000002 | 控制接收和发送缓存的寄存器 |
| Line Control Register | 1f000003 | 控制帧传输格式的寄存器 |
| Modem Control Register | 1f000004 | 调制解调器控制寄存器 (MCR) 提供启用/禁用自动流的能力功能，并启用/禁用 loopback 功能以进行诊断。 |
| Line Status Register | 1f000005 | 获取传输数据状态的寄存器 |
| Modem Status Register | 1f000006 | 调制解调器状态寄存器(MSR)向 CPU 提供有关调制解调器控制信号的状态，只读。 |
| Divisor Latche LSB | 1f000000 | 分频器的低 8 位 |
| Divisor Latche MSB | 1f000001 | 分频器的高 8 位 |

**表 2-2 Receiver Buffer Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 31-8 | N/A | 保留 |
| 7-0 | 只读 | 最新接受到的字符 |

**表 2-3 Transmitter Holding Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|

| | | |
|---|---|---|
| 31-8 | N/A | 保留 |
| 7-0 | 只读 | 保存下次需要发送的字符 |

**表 2-4 Interrupt Enable Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 2 | 可读可写 | 使能接受状态中断 |
| 1 | 可读可写 | 使能发送数据空中断 |
| 0 | 可读可写 | 使能接受数据有效中断 |
| 其他 | N/A | 保留 |

复位值：0x0

**表 2-5 Interrupt Identification Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 3-1 | 只读 | 中断 ID<br>011:接受状态中断<br>010：接受数据有效中断<br>110：字符超时中断<br>001：发送数据空中断<br>000：调制解调器状态中断 |
| 0 | 只读 | 1：有中断待处理<br>0：无中断 |
| 其他 | N/A | 保留 |

复位值：0x1

**表 2-6 FIFO Control Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 6-7 | 可读可写 | 设置接收缓存容量以触发接收中断的水位线‘00’-1 byte，‘01’-4 bytes，‘10’-8 bytes，‘11’-14 bytes |
| 3-5 | N/A | 保留 |
| 2 | 可读可写 | 发送缓存清零位，对此位写 1 后会将 Transmitter FiFO 中数据清零，但不清除正在接收的数据，这些数据在移位寄存器中正在进行串并转换 |
| 1 | 可读可写 | 接收缓存清零位，对此位写 1 后会将 Receiver FiFO 中数据清零，但不清除正在接收的数据，这些数据在移位寄存器中正在进行串并转换 |
| 0 | 可读可写 | 开关串口控制器是否采用缓存模式的使能位 (在本控制器中此位被忽略) |

复位值：0xc0

**表 2-7 Line Control Register 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 7 | 可读可写 | 波特率分频寄存器访问控制位：‘0’-将共享地址映射为功能寄存器地址，‘1’-将共享地址映射为波特率分频寄存器寄存器地址 |
| 6 | 可读可写 | 传输中断控制位：‘0’-传输中断，表现为串口输出被强制拉低为逻辑 0，‘1’-传输不中断 |

| | | |
|---|---|---|
| 5 | 可读可写 | 固定位校验使能：‘0’-屏蔽固定位校验，‘1’ 若在校验位使能的前提下，奇偶校验选择位为 ‘1’ 则帧传输校验位固定为 0，反之固定为 1 |
| 4 | 可读可写 | 奇偶校验选择位：‘0’-让帧数据位中与校验位共同组合的序列中 1 的个数为奇数，‘1’-帧传输数据位与校验位共同组合的序列中 1 的个数为偶数 |
| 3 | 可读可写 | 校验位使能：‘0’-帧传输无校验位，‘1’帧传输有校验位 |
| 2 | 可读可写 | 设置帧传输停止位比特数：‘0’-1 bit，‘1’-在帧传输数据为 5bits 时停止位为 1.5bit，其余情况停止位为 2 比特 |
| 1-0 | 可读可写 | 设置帧传输数据位比特数：‘00’-5 bits，‘01’-6 bits，‘10’-7 bits，‘11’-8 bits |

复位值：0x3

**表 2-8 Line Status Register 位定义**

| 位 | 读写权限 | 描述 | 清零操作 |
|---|---|---|---|
| 7 | 只读 | 传输错误标识位：‘0’-无传输错误，‘1’-有传输错误（PE、FE、BI） | 读 LSR 寄存器 |
| 6 | 只读 | 发送端为空标识位：‘0’-发送端不为空，‘1’-发送端为空，考虑正处于移位寄存器中是否正在发送的数据和发送缓存都为空 | 向 TxFIFO 写数据 |

| | | | |
|---|---|---|---|
| 5 | 只读 | 发送缓存为空标识位：‘0’-发送缓存不为空，‘1’-发送缓存为空，不考虑正处于移位寄存器中是否正在发送的数据 | 向 TxFIFO 写数据 |
| 4 | 只读 | 发生接收全 0 标识位（Break Interrupt）：‘0’-帧传输无 Break 错误，‘1’-当数据线为 0 超过一帧传输的时间则置起此位 | 读 LSR 寄存器 |
| 3 | 只读 | 缺失停止位标识位（Framing Error）：‘0’-停止位检测无错误，‘1’-若接收缓存中存在缺失停止位错误则置起此位 | 读 LSR 寄存器 |
| 2 | 只读 | 校验错误标识位（Parity Error）：‘0’-无校验错误，‘1’-若接收缓存中存在校验错误则置起此位 | 读 LSR 寄存器 |
| 1 | 只读 | 过载错误标识位（Overrun Error）：‘0’-无过载错误，‘1’-在接收缓存为满时收到了新的输入数据请求，则会出现过载中断 | 读 LSR 寄存器 |
| 0 | 只读 | 数据就绪标识位（Data Ready）：‘0’-接收缓存无字符，‘1’-接收缓存有字符 | 读 LSR 寄存器 |

复位值：0x0

**表 2-9 Divisor Latche LSB 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 0-7 | 可读可写 | 分频寄存器的 LSB, 对此寄存器写入则意味着分频器开始工作 |
| 其他 | N/A | 保留 |

复位值：0x0

**表 2-10 Divisor Latche MSB 位定义**

| 位 | 读写权限 | 描述 |
|---|---|---|
| 0-7 | 可读可写 | 分频寄存器的 MSB，与 LSB 位拼接后组成分频系数 |
| 其他 | N/A | 保留 |

复位值：0x0

Divisor Latches （DL），此寄存器用于产生分频信号，以匹配不同的波特率对应的数据传输频率。控制器时钟频率/(16* 波特率）= {MSB,LSB}。由于对 LSB 写值时同时也会触发 DivisorLatch 开始计数的使能信号，所以在实际配置中要先配置 MSB，再设置 LSB。

#### 2.2.4.3 UART 初始化流程

1）首先通过 APB 总线对数据控制寄存器（LCR）写控制字，描述本次传输所需的帧数据长度和停止位宽度和校验格式，同时开启 LCR 寄存器中分频器（Divisor Latch）的访问权限以向分频器写入此次传输波特率对应的分频计数器的值。

2）其次通过 APB 总线对 Divisor Latch 写波特率对应的分频值来产生传输每个比特的周期时间。Divisor 是由两个 8bit 寄存器构成。其中高 8 位称为 MSB，低 8 位称为 LSB。其计算公式为:

控制器时钟频率/(16* 波特率）= {MSB,LSB}

3）然后通过 APB 总线对数据控制寄存器（LCR）写控制字，关闭对 Divisior Latch 的访问权限，开启对传输/接收缓存和中断使能寄存器的访问权限。

4）接着通过 APB 总线对存储状态控制寄存器（FCR）写控制字，设置接收缓存的触发中断的存储 “水位线”。

注：将接收缓存水位线设高，会降低串口产生过载中断（Overrun Error）的

频次，提高系统整体运行的效率。

5）最后通过 APB 总线对中断使能寄存器（IER）写控制字，开启对应中断的使能位，允许对应不同类型中断的产生。

#### 2.2.4.4 AMBA2 APB 总线协议

UART 模块采用的是 AMBA2 APB 接口，下表给出了 APB SLAVE 的接口信号与定义，可以看出较 AXI 接口简单很多且不存在握手信号（AMBA3 APB 增加了 PREADY 信号，在本示例中无此握手信号）。

**表 2-11 APB SLAVE 信号列表**

| 信号 | IO | 描述 |
|---|---|---|
| PCLK | input | 总线时钟 |
| PRESETn | input | 总线复位信号低有效 |
| PADDR[31:0] | input | 32 位地址总线 |
| PSEL | input | Slave 选择信号 |
| PENABLE | input | 指示 APB 操作的第 2 个周期 |
| PWRITE | input | ‘1’-写，’”0’-读 |
| PRDATA | output | 读操作 Slave 返回给 Master 的数据总线 |
| PWDATA | input | 写操作 Master 传给 Slave 的数据总线 |

APB 总线进行传输时，有空闲（IDLE）、地址阶段（SETUP）、数据阶段(ENABLE)，其状态转换图如下：

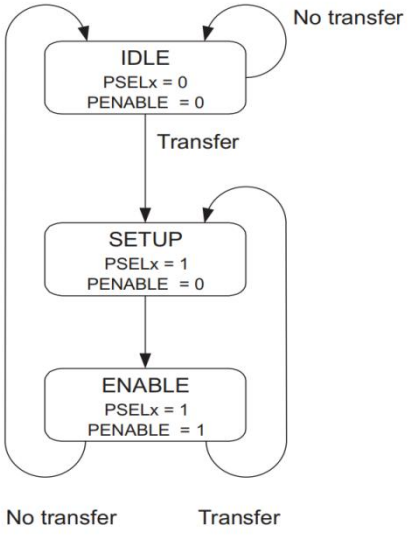



**图 2-2 APB 状态机**

APB 总线的初始状态时 IDLE，在 SETUP 状态进行读/写地址的传输，此时 PSEL=1，PENABLE=0；在 ENABLE 状态进行读/写数据的传输，此时 PSEL=1，PENABLE=1。具体的读写时序图如下所示，在 APB 写操作时，PWDATA 可以在 PSEL 拉高的 T2 时钟周期给出，也可以在 PENABLE 拉高的 T3 时钟周期给出；在 APB 读操作中，PRDATA 必须在 PSEL 和 PENABLE 同时为高的 T3 时钟周期给出。

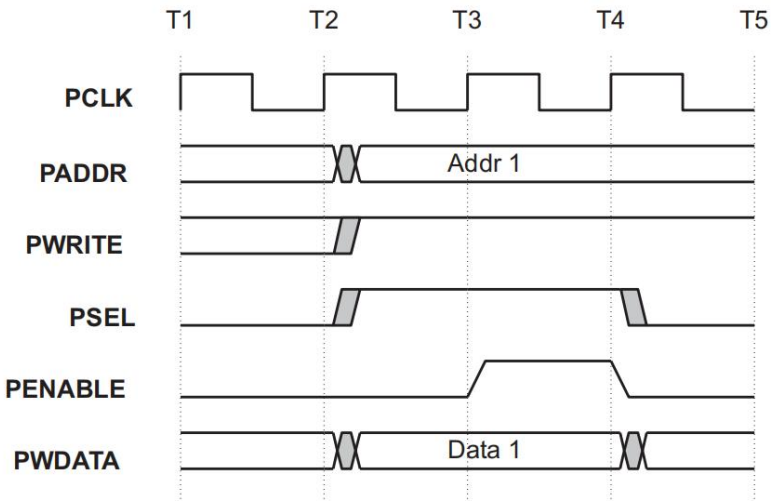



**图 2-3 APB 写操作**

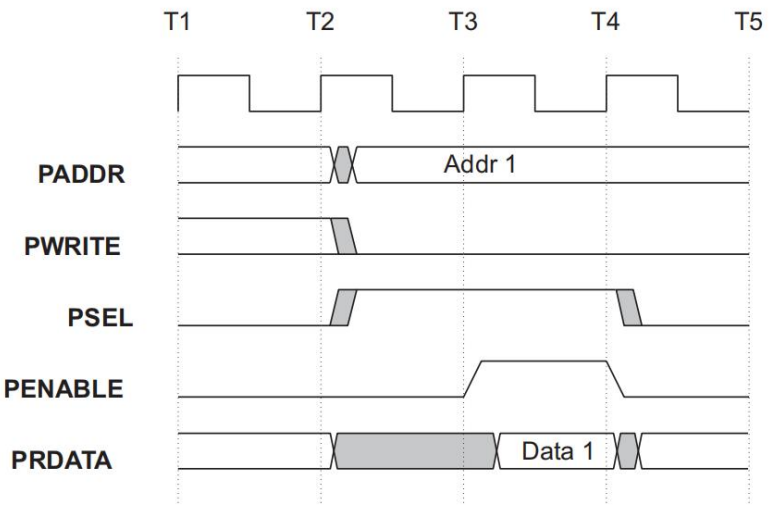



**图 2-4　APB 读操作**

### 2.2.5　时钟与复位

时钟的产生使用 xilinx 的 PLL IP 核，其 xci 文件位于 rtl/ip/PLL_2019_2，其例化模板为：

```
clk_pll u_clk_pll(
        .cpu_clk     (cpu_clk),
        .sys_clk     (sys_clk),
        .resetn      (resetn),
        .locked      (pll_locked),
        .clk_in1     (clk)
    );
```

其中 clk_in1 是输入的 50MHz 的晶振时钟，输出的 cpu_clk 是 33MHz，sys_clk 是 50MHz。PLL 接收低电平的复位信号，等待锁相环稳定后，输出 locked 信号拉高。远程 FPGA 板的复位按键按下时为高电平，需要取反后送入 PLL 的 resetn 端口。

SoC 的复位信号建议使用 PLL 的 locked 信号进行异步复位同步释放后使用，以确保复位信号的时序满足 recovery time（恢复时间）和 removal time（清除时间）。异步复位同步释放模块的源码位于 rtl/ip/rst_sync，例化模版如下：

```
rst_sync u_rst_sys(
        .clk(sys_clk),
        .rst_n_in(pll_locked),
        .rst_n_out(sys_resetn)
    );
rst_sync u_rst_cpu(
        .clk(cpu_clk),
```

```
        .rst_n_in(sys_resetn),
        .rst_n_out(cpu_resetn)
    );
```

经过异步复位同步释放处理后的 sys_resetn 和 cpu_resetn 就可以分别送至 sys_clk 时钟域的模块和 cpu_clk 时钟域的模块使用了。

## 2.3 实验所用嵌入式软件

本实验进行的是串口输出的 Hello World，下面将介绍处理器核如何调用串口模块实现 Printf 功能。

### 2.3.1 在启动引导函数中进行串口初始化

处理器核上电后第一条 PC 是 0x1c000000，对应进入的是 start.S 文件（sdk/software/bsp/env/start.S）中的启动引导函数_start。在_start 函数中，使用下面的代码进行 UART 模块初始化，其中 UART_BASE 在 main.c 中给出，数值是 0xbf000000，对应 UART 模块在 AXI 总线上的基地址 0x1f000000 加窗。CSR.DMW1 在初始化 UART 之前已经配置为 0xa0000009，即虚地址 0xa0000000 到 0xbfffffff 的空间被映射到物理地址 0x00000000 到 0x1fffffff，存储访问类型是强序非缓存；因此处理器核访问物理地址 0x1f000000 时应发出的虚拟地址是 0xbf000000。

串口初始化的具体流程是依次配置 FIFO Control Register、Line Control Register、DIVISOR_MSB、DIVISOR_LSB、Line Control Register、MODEM Control Register。配置 FIFO Control Register 为 0x7，清零接受缓存、清零发送缓存并使能 FIFO。配置 Line Control Register 为 0x80，将地址 1f000000 映射为分频寄存器。配置 DIVISOR_MSB 为 0，配置 DIVISOR_LSB 为 0x1b，即波特率 115200(计算公式是 DIVISOR=50000000/(115200*16))。这里有个 SIMU 宏，SIMU 宏为 1 时分频系数给 1，用于加速仿真。再次配置 Line Control Register 为 0x3，将 1f000000 映射为功能寄存器（Receiver Buffer Register 和 Transmitter Holding Register），并且配置串口通信格式为数据位宽 8bit，无校验位，停止位宽 1bit。最后将 MODEM Control Register 配置为 0，禁用自动流控制，禁用环回模式，至此完成串口初始化。

```
1.    /* init UART */
2.    la.local    t0, UART_BASE
3.    ld.w        t1, t0, 0
4.    #WRITE(li.wne,OFS_FIFO,FIFO_ENABLE|FIFO_RCV_RST|FIFO_XMT_RST|FIFO_TRIGGER_0);

5.    li.w       t2, 0x07
6.    st.b     t2, t1, 2
7.    #WRITE(li.wne,OFS_LINE_CONTROL, 0x80);
8.    li.w       t2, 0x80
9.        st.b     t2, t1, 3
10.   #WRITE(li.wne,OFS_DIVISOR_MSB, (divisor & 0xff00) >> 8);
```

```
    li.w       t2, 0x00
    st.b     t2, t1, 1
    #WRITE(li.wne,OFS_DIVISOR_LSB, divisor & 0xff);
    if SIMU==1
       li.w       t2, 0x1
    else
       li.w       t2, 0x1b
    endif
    st.b     t2, t1, 0
    #WRITE(li.wne,OFS_DATA_FORMAT, data | parity | stop);
    li.w       t2, 0x3
    st.b     t2, t1, 3
    #WRITE(li.wne,OFS_MODEM_CONTROL,0);
    li.w       t2, 0x0
    st.b     t2, t1, 4
```

### 2.3.2 串口输出字符

完成串口初始化后就可以在 main.c 中使用 Printf 函数直接打印字符串了，之所以能实现 Printf 函数控制 UART 模块，最底层的 write 函数同样是在访问 UART 模块的寄存器。Printf 函数调用的底层系统调用函数是 write，在 la32r-picolibc 中，write 函数调用 uart_putchar 函数进行一个字符的输出，其实现源码如下：

```
    void uart_putchar(char c)
    {
        while (!((*( volatile char * ) ( UART_BASE + 0x5 )) & 0x20));
        *( volatile char * ) ( UART_BASE ) = c;
    }

    char uart_getchar(void)
    {
        char data;
        while (!((*( volatile char * ) ( UART_BASE + 0x5 )) & 0x1));
        data = *((volatile unsigned char *)(UART_BASE));
        return data;
    }
```

从 uart_putchar 函数可以看出，输出字符的逻辑就是等待 Line Status Register 的发送缓存为空标识位为 1 后将需要输出的字符发送到 ransmitter Holding Register。而读一个串口输入字符的逻辑与之类似，等待 Line Status Register 的数据就绪标识符为 1 后将接收的字符从 Receiver Buffer Register 读出。

## 2.4 实验流程

### 2.4.1 搭建基础 SoC 顶层

该实验需要大家自行搭建基础 SoC 的顶层文件，在 rtl/soc_top.v 中提供了半空白的顶层模板，仅包含时钟复位网络、互联信号、AXI 互联总线三个部分。挂载在 AXI 总线上的 master 和 slave 模块需要自行添加。完成本章实验后，SoC 硬件架构如下图所示。

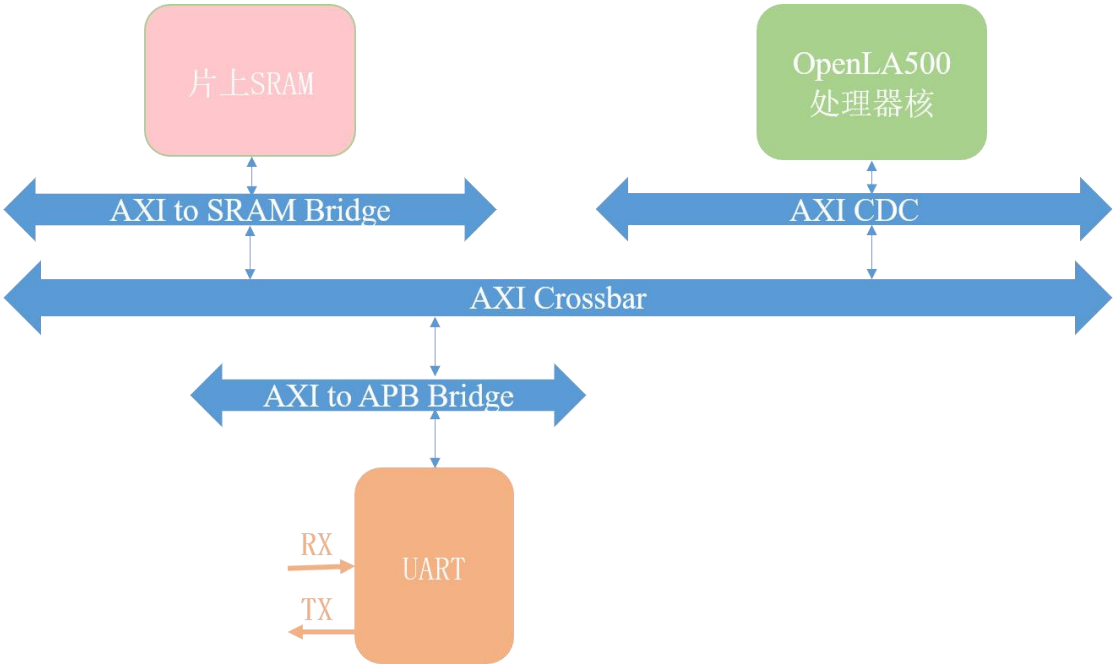



**图 2-5 SoC 架构**

已经实例化好的时钟与复位网络见下图：

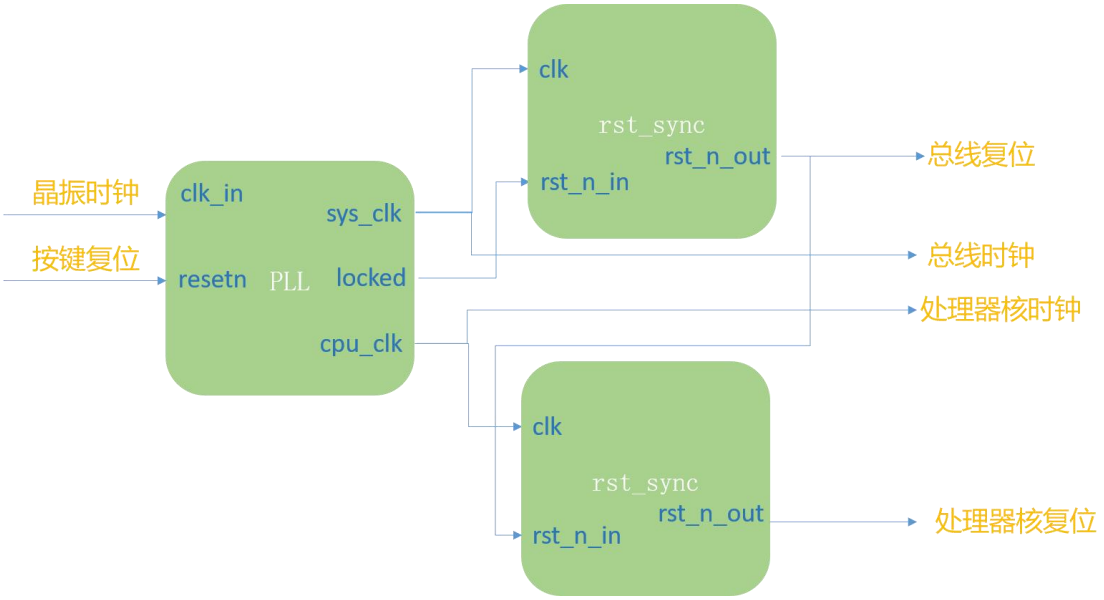



**图 2-6 SoC 时钟与复位网络**

下面具体进行 SoC 顶层搭建。

#### 2.4.1.1 实例化处理器核

首先在 soc_top.v 中实例化 OpenLA500 处理器核，它的时钟域是 cpu_clk（33MHz），总线接口为 AXI 接口，实验一不进行中断测试，中断信号暂时给 8'h0。

```
core_top u_cpu(
    .intrpt     (8'h0           ),    //high active
    .aclk       (cpu_clk        ),
    .aresetn    (cpu_resetn     ),    //low active
```

```
    .arid      (cpu_arid      ),
    .araddr    (cpu_araddr    ),
    .arlen     (cpu_arlen     ),
    .arsize    (cpu_arsize    ),
    .arburst   (cpu_arburst   ),
    .arlock    (cpu_arlock    ),
    .arcache   (cpu_arcache   ),
    .arprot    (cpu_arprot    ),
    .arvalid   (cpu_arvalid   ),
    .arready   (cpu_arready   ),

    .rid       (cpu_rid       ),
    .rdata     (cpu_rdata     ),
    .rresp     (cpu_rresp     ),
    .rlast     (cpu_rlast     ),
    .rvalid    (cpu_rvalid    ),
    .rready    (cpu_rready    ),

    .awid      (cpu_awid      ),
    .awaddr    (cpu_awaddr    ),
    .awlen     (cpu_awlen     ),
    .awsize    (cpu_awsize    ),
    .awburst   (cpu_awburst   ),
    .awlock    (cpu_awlock    ),
    .awcache   (cpu_awcache   ),
    .awprot    (cpu_awprot    ),
    .awvalid   (cpu_awvalid   ),
    .awready   (cpu_awready   ),

    .wid       (cpu_wid       ),
    .wdata     (cpu_wdata     ),
    .wstrb     (cpu_wstrb     ),
    .wlast     (cpu_wlast     ),
    .wvalid    (cpu_wvalid    ),
    .wready    (cpu_wready    ),

    .bid       (cpu_bid       ),
    .bresp     (cpu_bresp     ),
    .bvalid    (cpu_bvalid    ),
    .bready    (cpu_bready    ),

    //debug interface
    .break_point        (1'b0               ),
    .infor_flag         (1'b0               ),
    .reg_num            (5'b0               ),
    .ws_valid           (                   ),
    .rf_rdata           (                   ),

    .debug0_wb_pc       (debug_wb_pc        ),
    .debug0_wb_inst     (debug_wb_inst      ),
    .debug0_wb_rf_wen   (debug_wb_rf_wen    ),
    .debug0_wb_rf_wnum  (debug_wb_rf_wnum   ),
    .debug0_wb_rf_wdata (debug_wb_rf_wdata  )
);
```

#### 2.4.1.2 实例化 AXI 跨时钟域模块

由于处理器核与总线所处的时钟域不同，AXI 信号无法直接连接，中间需要添加 AXI 跨时钟域模块，该模块使用异步 FIFO 实现跨时钟域处理。

```
//clock sync: from CPU to AXI_Crossbar
Axi_CDC  u_Axi_CDC (
    .axiInClk               ( cpu_clk           ),
    .axiInRstn              ( cpu_resetn        ),
```

```
    .axiOutClk              ( sys_clk             ),
    .axiOutRstn             ( sys_resetn          ),

    .axiIn_awvalid          ( cpu_awvalid         ),
    .axiIn_awaddr           ( cpu_awaddr          ),
    .axiIn_awid             ( {1'b0,cpu_awid}     ),
    .axiIn_awlen            ( cpu_awlen           ),
    .axiIn_awsize           ( cpu_awsize          ),
    .axiIn_awburst          ( cpu_awburst         ),
    .axiIn_awlock           ( cpu_awlock[0]       ),
    .axiIn_awcache          ( cpu_awcache         ),
    .axiIn_awprot           ( cpu_awprot          ),
    .axiIn_wvalid           ( cpu_wvalid          ),
    .axiIn_wdata            ( cpu_wdata           ),
    .axiIn_wstrb            ( cpu_wstrb           ),
    .axiIn_wlast            ( cpu_wlast           ),
    .axiIn_bready           ( cpu_bready          ),
    .axiIn_arvalid          ( cpu_arvalid         ),
    .axiIn_araddr           ( cpu_araddr          ),
    .axiIn_arid             ( {1'b0,cpu_arid}     ),
    .axiIn_arlen            ( cpu_arlen           ),
    .axiIn_arsize           ( cpu_arsize          ),
    .axiIn_arburst          ( cpu_arburst         ),
    .axiIn_arlock           ( cpu_arlock[0]       ),
    .axiIn_arcache          ( cpu_arcache         ),
    .axiIn_arprot           ( cpu_arprot          ),
    .axiIn_rready           ( cpu_rready          ),
    .axiOut_awready         ( cpu_sync_awready    ),
    .axiOut_wready          ( cpu_sync_wready     ),
    .axiOut_bvalid          ( cpu_sync_bvalid     ),
    .axiOut_bid             ( {1'b0,cpu_sync_bid}),
    .axiOut_bresp           ( cpu_sync_bresp      ),
    .axiOut_arready         ( cpu_sync_arready    ),
    .axiOut_rvalid          ( cpu_sync_rvalid     ),
    .axiOut_rdata           ( cpu_sync_rdata      ),
    .axiOut_rid             ( {1'b0,cpu_sync_rid}),
    .axiOut_rresp           ( cpu_sync_rresp      ),
    .axiOut_rlast           ( cpu_sync_rlast      ),

    .axiIn_awready          ( cpu_awready         ),
    .axiIn_wready           ( cpu_wready          ),
    .axiIn_bvalid           ( cpu_bvalid          ),
    .axiIn_bid              ( {cpu_bid_4,cpu_bid} ),
    .axiIn_bresp            ( cpu_bresp           ),
    .axiIn_arready          ( cpu_arready         ),
    .axiIn_rvalid           ( cpu_rvalid          ),
    .axiIn_rdata            ( cpu_rdata           ),
    .axiIn_rid              ( {cpu_rid_4,cpu_rid} ),
    .axiIn_rresp            ( cpu_rresp           ),
    .axiIn_rlast            ( cpu_rlast           ),
    .axiOut_awvalid         ( cpu_sync_awvalid    ),
    .axiOut_awaddr          ( cpu_sync_awaddr     ),
    .axiOut_awid            ( {cpu_sync_awid_4,cpu_sync_awid})
,
    .axiOut_awlen           ( cpu_sync_awlen      ),
    .axiOut_awsize          ( cpu_sync_awsize     ),
    .axiOut_awburst         ( cpu_sync_awburst    ),
    .axiOut_awlock          ( cpu_sync_awlock     ),
    .axiOut_awcache         ( cpu_sync_awcache    ),
    .axiOut_awprot          ( cpu_sync_awprot     ),
    .axiOut_wvalid          ( cpu_sync_wvalid     ),
    .axiOut_wdata           ( cpu_sync_wdata      ),
    .axiOut_wstrb           ( cpu_sync_wstrb      ),
    .axiOut_wlast           ( cpu_sync_wlast      ),
    .axiOut_bready          ( cpu_sync_bready     ),
    .axiOut_arvalid         ( cpu_sync_arvalid    ),
```

```
    .axiOut_araddr          ( cpu_sync_araddr     ),
    .axiOut_arid            ( {cpu_sync_arid_4,cpu_sync_arid})
,
    .axiOut_arlen           ( cpu_sync_arlen      ),
    .axiOut_arsize          ( cpu_sync_arsize     ),
    .axiOut_arburst         ( cpu_sync_arburst    ),
    .axiOut_arlock          ( cpu_sync_arlock     ),
    .axiOut_arcache         ( cpu_sync_arcache    ),
    .axiOut_arprot          ( cpu_sync_arprot     ),
    .axiOut_rready          ( cpu_sync_rready     )
);
```

#### 2.4.1.3 实例化 SRAM 控制器

SRMA 控制器用于对板载 SRMA 芯片实现读写操作。

```
//axi ram
axi_wrap_ram_sp_ext u_axi_ram (
    .aclk          ( sys_clk           ),
    .aresetn       ( sys_resetn        ),
    //ar
    .axi_arid      ( ram_arid          ),
    .axi_araddr    ( ram_araddr        ),
    .axi_arlen     ( ram_arlen         ),
    .axi_arsize    ( ram_arsize        ),
    .axi_arburst   ( ram_arburst       ),
    .axi_arlock    ( ram_arlock        ),
    .axi_arcache   ( ram_arcache       ),
    .axi_arprot    ( ram_arprot        ),
    .axi_arvalid   ( ram_arvalid       ),
    .axi_arready   ( ram_arready       ),
    //r
    .axi_rid       ( ram_rid           ),
    .axi_rdata     ( ram_rdata         ),
    .axi_rresp     ( ram_rresp         ),
    .axi_rlast     ( ram_rlast         ),
    .axi_rvalid    ( ram_rvalid        ),
    .axi_rready    ( ram_rready        ),
    //aw
    .axi_awid      ( ram_awid          ),
    .axi_awaddr    ( ram_awaddr        ),
    .axi_awlen     ( ram_awlen         ),
    .axi_awsize    ( ram_awsize        ),
    .axi_awburst   ( ram_awburst       ),
    .axi_awlock    ( ram_awlock        ),
    .axi_awcache   ( ram_awcache       ),
    .axi_awprot    ( ram_awprot        ),
    .axi_awvalid   ( ram_awvalid       ),
    .axi_awready   ( ram_awready       ),
    //w
    .axi_wdata     ( ram_wdata         ),
    .axi_wstrb     ( ram_wstrb         ),
    .axi_wlast     ( ram_wlast         ),
    .axi_wvalid    ( ram_wvalid        ),
    .axi_wready    ( ram_wready        ),
    //b               ram
    .axi_bid       ( ram_bid           ),
    .axi_bresp     ( ram_bresp         ),
    .axi_bvalid    ( ram_bvalid        ),
    .axi_bready    ( ram_bready        ),

    .base_ram_addr ( base_ram_addr     ),
    .base_ram_be_n ( base_ram_be_n     ),
    .base_ram_ce_n ( base_ram_ce_n     ),
```

```
    .base_ram_oe_n ( base_ram_oe_n      ),
    .base_ram_we_n ( base_ram_we_n      ),
    .ext_ram_addr  ( ext_ram_addr       ),
    .ext_ram_be_n  ( ext_ram_be_n       ),
    .ext_ram_ce_n  ( ext_ram_ce_n       ),
    .ext_ram_oe_n  ( ext_ram_oe_n       ),
    .ext_ram_we_n  ( ext_ram_we_n       ),

    .base_ram_data ( base_ram_data      ),
    .ext_ram_data  ( ext_ram_data       )
);
```

#### 2.4.1.4 实例化 UART 控制器

用与 UART 串口输入和输出控制。

```
//AXI2APB
axi_uart_controller u_axi_uart_controller
(
    .clk                (sys_clk                ),
    .rst_n              (sys_resetn             ),

    .axi_s_awid         (uart_awid              ),
    .axi_s_awaddr       (uart_awaddr            ),
    .axi_s_awlen        (uart_awlen             ),
    .axi_s_awsize       (uart_awsize            ),
    .axi_s_awburst      (uart_awburst           ),
    .axi_s_awlock       (uart_awlock            ),
    .axi_s_awcache      (uart_awcache           ),
    .axi_s_awprot       (uart_awprot            ),
    .axi_s_awvalid      (uart_awvalid           ),
    .axi_s_awready      (uart_awready           ),
    .axi_s_wid          (uart_awid              ),
    .axi_s_wdata        (uart_wdata             ),
    .axi_s_wstrb        (uart_wstrb             ),
    .axi_s_wlast        (uart_wlast             ),
    .axi_s_wvalid       (uart_wvalid            ),
    .axi_s_wready       (uart_wready            ),
    .axi_s_bid          (uart_bid               ),
    .axi_s_bresp        (uart_bresp             ),
    .axi_s_bvalid       (uart_bvalid            ),
    .axi_s_bready       (uart_bready            ),
    .axi_s_arid         (uart_arid              ),
    .axi_s_araddr       (uart_araddr            ),
    .axi_s_arlen        (uart_arlen             ),
    .axi_s_arsize       (uart_arsize            ),
    .axi_s_arburst      (uart_arburst           ),
    .axi_s_arlock       (uart_arlock            ),
    .axi_s_arcache      (uart_arcache           ),
    .axi_s_arprot       (uart_arprot            ),
    .axi_s_arvalid      (uart_arvalid           ),
    .axi_s_arready      (uart_arready           ),
    .axi_s_rid          (uart_rid               ),
    .axi_s_rdata        (uart_rdata             ),
    .axi_s_rresp        (uart_rresp             ),
    .axi_s_rlast        (uart_rlast             ),
    .axi_s_rvalid       (uart_rvalid            ),
    .axi_s_rready       (uart_rready            ),

    .apb_rw_dma         (1'b0                   ),
    .apb_psel_dma       (1'b0                   ),
    .apb_enab_dma       (1'b0                   ),
    .apb_addr_dma       (20'b0                  ),
    .apb_valid_dma      (1'b0                   ),
    .apb_wdata_dma      (32'b0                  ),
    .apb_rdata_dma      (                       ),
```

```
51.     .apb_ready_dma      (                       ),
52.     .dma_grant          (                       ),
53.
54.     .dma_req_o          (                       ),
55.     .dma_ack_i          (1'b0                   ),
56.
57.     //UART0
58.     .uart0_txd_i        (uart0_txd_i            ),
59.     .uart0_txd_o        (uart0_txd_o            ),
60.     .uart0_txd_oe       (uart0_txd_oe           ),
61.     .uart0_rxd_i        (uart0_rxd_i            ),
62.     .uart0_rxd_o        (uart0_rxd_o            ),
63.     .uart0_rxd_oe       (uart0_rxd_oe           ),
64.     .uart0_rts_o        (uart0_rts_o            ),
65.     .uart0_dtr_o        (uart0_dtr_o            ),
66.     .uart0_cts_i        (uart0_cts_i            ),
67.     .uart0_dsr_i        (uart0_dsr_i            ),
68.     .uart0_dcd_i        (uart0_dcd_i            ),
69.     .uart0_ri_i         (uart0_ri_i             ),
70.     .uart0_int          (uart0_int              )
71. );
```

至此完成基础 SoC 搭建，下面可以功能仿真和 FPGA 验证。

### 2.4.2 进行功能仿真和 FPGA 验证

下面进行经典 C 程序 Hello_World 的功能仿真和 FPGA 验证。

Step1：准备测试软件 Hello_World

在 WSL 的终端中先进入 sdk/software/examples/hello_word 目录，再使用命令：

```
$ make clean
$ make
```

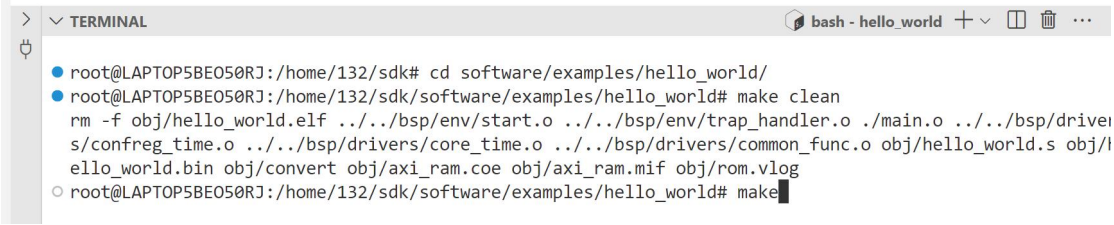



**图 2-7 运行 make 进行编译**

执行 make 完成即可看到 hello_world 目录下产生了 obj 文件夹，里面有生成的内存初始化文件 axi_ram.mif 以及用于下载至板载 BaseRAM 的二进制文件 hello_world.bin。只要工具链安装时正确指定了 LA32RSOC_WINDOWS_HOME 的路径，Makefile 会自动将 axi_ram.mif 和 hello_world.bin 搬移至 windows 目录下 la32r_soc/sdk 目录下。

Step2：创建 vivado 工程

打开 vivado，在 vivado 控制台中首先切换目录至 fpga，再调用脚本 create_project.tcl，完成 vivado 工程创建。控制台中使用的命令如下：

```
$ cd d:/la32r_soc/fpga
```

```
$ source create_project.tcl
```

之后可以看到工程自动被创建并读入了设计文件。

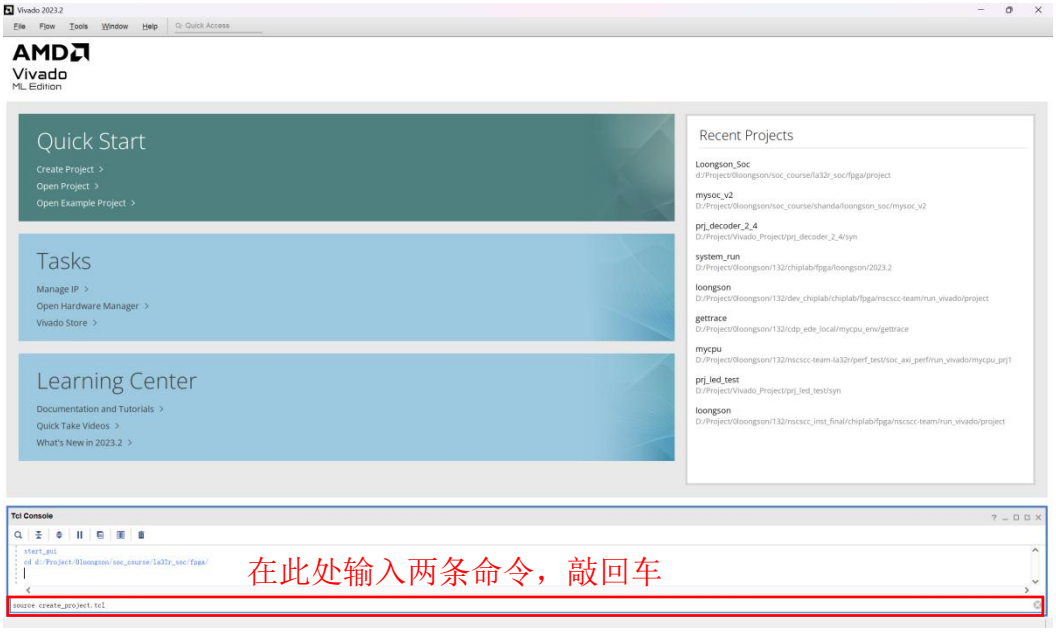



**图 2-8 vivado 创建工程**

Step3：运行功能仿真

进行仿真前先进行 RTL 分析确保没有基础的语法错误，同时这一步也会初始化 PLL IP 核。点击 Vivado 左侧 RTL ANALYSIS -> Schematic，看到正确生成原理图证明 RTL 分析通过。

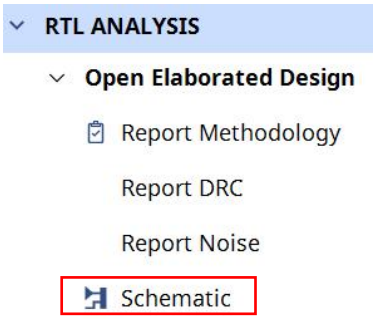



**图 2-9 vivado 进行 RTL 分析**

直接在 vivado 中点击 `run simulation` -> `run behavioral simulation`，在仿真界面点 run all，即可在控制台中看到串口输出`hello world!`

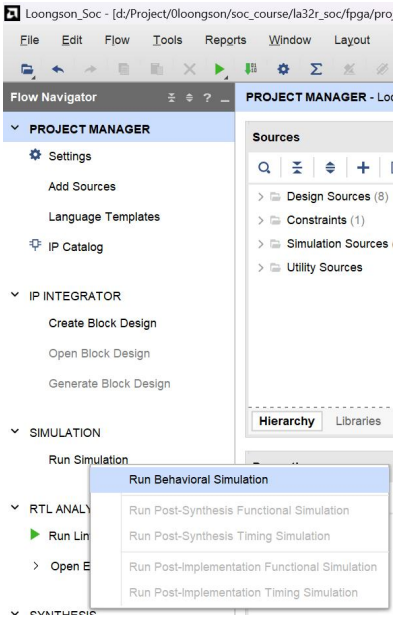



**图 2-10　vivado 运行功能仿真**

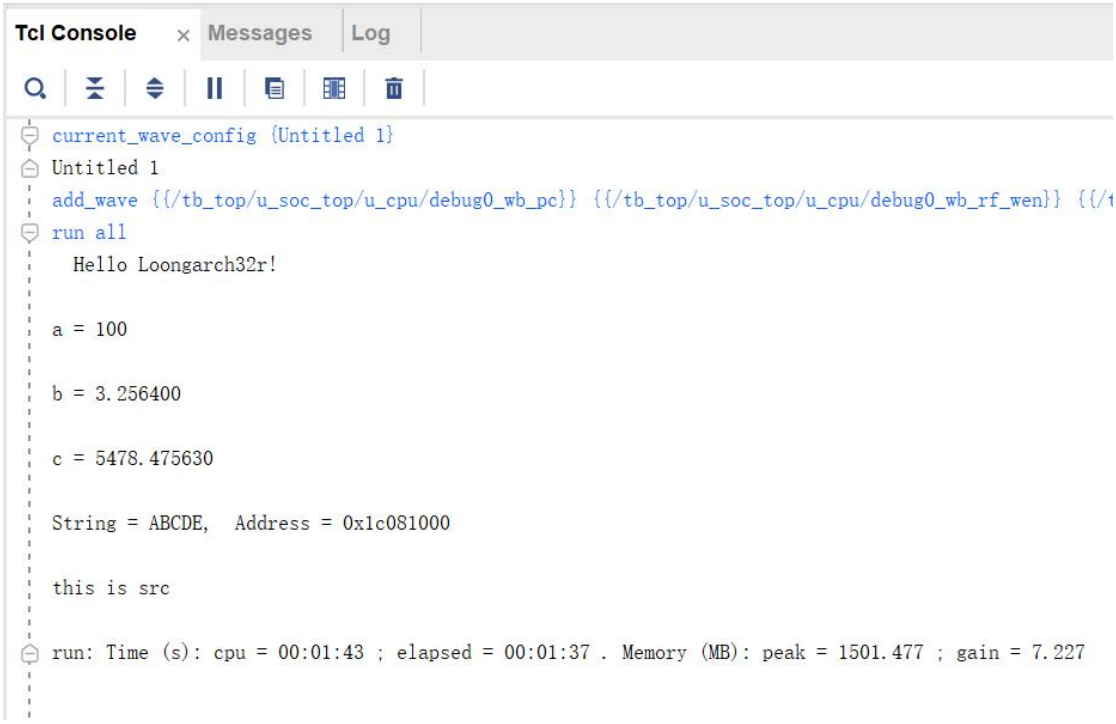



**图 2-11　hello_word 功能仿真通过**

若程序没有正常输出，需要进行 DEBUG，推荐先将处理器核的 debug 信号添加至波形窗口，根据 PC 的值对照反汇编文件（sdk/software/examples/hello_world/obj/hello_world.s）查看出错的位置。

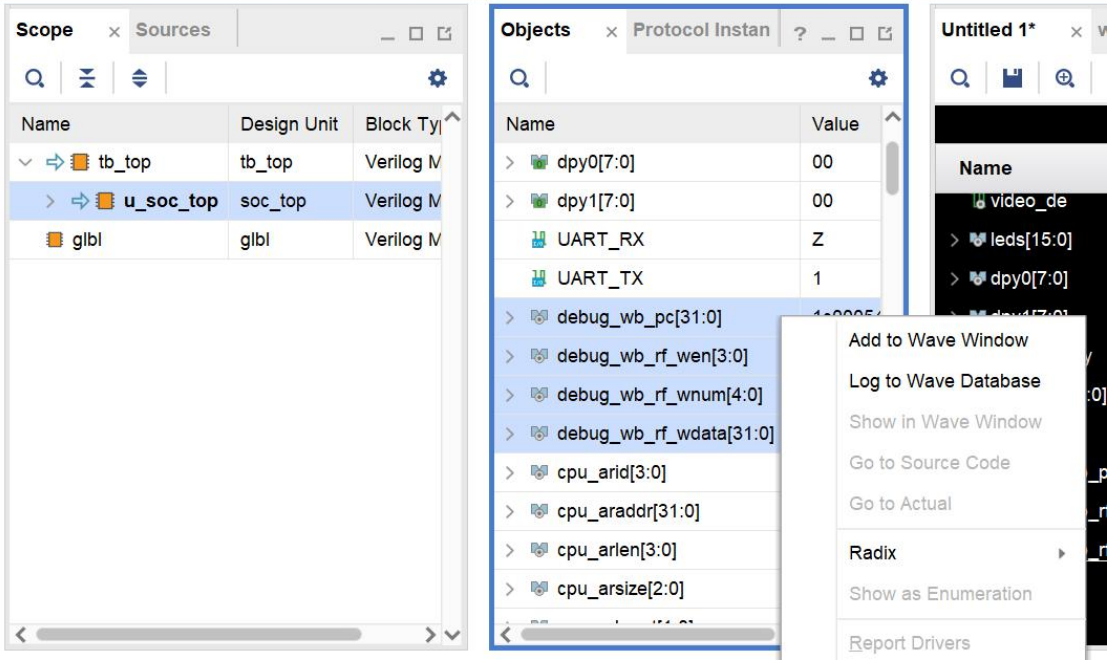



**图 2-12　添加 debug 信号至波形窗口**

```
1c000580 <main>:
#define CPSIZE 12
char src[CPSIZE] = "this is src";
char dst[CPSIZE] = "this is dst";

int main(int argc, char** argv)
{
1c000580: 02bfc063  addi.w  $r3,$r3,-16(0xff0)
  int a = 100;
  float b = 3.2564;
  double c = 5478.47563;
  char *str;

  printf("Hello Loongarch32r!\n");
1c000584: 1c001004  pcaddu12i $r4,128(0x80)
1c000588: 02a9f084  addi.w  $r4,$r4,-1412(0xa7c)
```

PC 值　指令编码　汇编指令

**图 2-13　反汇编文件**

Step4：进行 FPGA 验证

在 vivado 中直接点击 Generate Bitstream，等待完成后会在`fpga\project\Loongson_Soc.runs\impl_1`目录下出现`soc_top.bit`文件。

使用浏览器打开网站，输入账号密码登录后，在上传位流处点击选择文件，选中刚刚生成的 bit 文件，点击上传并开始。看到 FPGA 界面出现后，下拉来到

SRAM 下载功能，在写入数据处选择刚刚生成的` hello_world.bin `文件，点击写入,出现操作成功完成的提示后。设置波特率为 115200 并打开串口，按下 RST 键即可看到串口输出 Hello Loongarch32r!。


**图 2-14　远程 FPGA 下载 SRAM**


**图 2-15　设置串口波特率**

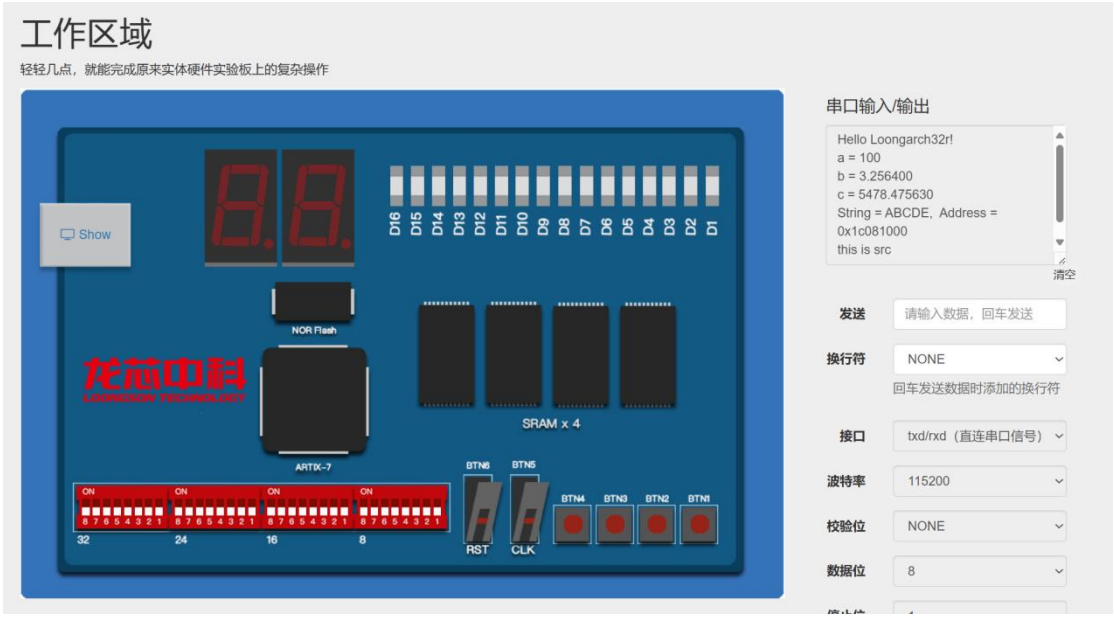



**图 2-16　远程 FPGA 运行 Hello World 例程**

# 第 3 章　阶段任务二 外部中断控制器实验之弹球游戏

## 3.1　实验任务

基于实验环境提供的基础 Confreg，添加外部中断控制器，支持中断使能、电平中断/脉冲中断选择，中断电平设置、中断清除、中断置位、中断状态查询。使用该外部中断控制器处理按键中断、外部定时器中断。另外将实验环境提供的 DVI 控制器添加至 SoC 中，用于支持图像显示功能。最后所搭建的 SoC 在功能仿真中能正常执行中断例程，运行按键中断服务函数和外部定时器中断服务函数，并在 FPGA 上能实现基于按键和 DVI 的弹球游戏。

## 3.2　实验所用 IP 简介

本次实验将在基础 SoC 的基础上新增两个模块，分别是 Confreg（包含按键、拨码开关、LED、数码管、中断控制器）和 DVI 控制器。其中 Confreg 的中断控制器需要自行添加，具体添加方法详见 3.4 实验流程。

### 3.2.1　Confreg

Confreg 用于控制 FPGA 板上的按键、拨码开关、LED、数码管，并包含中断控制器用于统一管理外部中断。Confreg 使用 AXI 接口连接至 AXI_Crossbar 的 axiOut_4，地址空间为 1f200000-1f2fffff。

Confreg 中已经包含基本的拨码开关、LED、数码管、外部定时器寄存器以及未实现的外部中断控制器寄存器，下表给出寄存器清单。

**表 3-1　Confreg 寄存器总览**

| 名称 | 地址 | 描述 |
|---|---|---|
| confreg_int_en | 1f20f000 | 中断使能寄存器，为 1 表示使能比特位对应设备中断 |
| confreg_int_edge | 1f20f004 | 中断边沿选择寄存器，为 1 表示边沿触发中断,为 0 表示电平触发 |
| confreg_int_pol | 1f20f008 | 中断极性选择寄存器，当 int_edge 配置成电平触发时,为 1 表示高电平触发中断,为 0 表示低电平触发；<br>当 int_edge 配置成边沿触发时,为 1 表示上升沿触发中断,为 0 表示下降沿触发 |

| | | |
|---|---|---|
| confreg_int_clr | 1f20f00c | 中断清除寄存器，为 1 表示清除中断（针对边沿触发的中断） |
| confreg_int_set | 1f20f010 | 中断置位寄存器，为 1 表示中断置位（针对边沿触发的中断） |
| confreg_int_state | 1f20f014 | 中断状态寄存器，为 1 表示比特位对应设备产生中断 |
| sys_timer | 1f20f100 | 外部定时器计数值，只读 |
| sys_timer_cmp | 1f20f104 | 外部定时器计数上限，当 sys_timer 计数至 sys_timer_cmp-1 时产生高电平中断 |
| sys_timer_en | 1f20f108 | 外部定时器计数使能，为 0 时清零 sys_timer 值和中断位，为 1 时开始计数 |
| digital_ctrl | 1f20f200 | 仅低 2 位有效，digital_ctrl[0]代表数码管 0 使能，digital_ctrl[1]代表数码管 1 使能 |
| digital_data | 1f20f204 | 仅低 8 位有效，digital_data[3:0]对应数码管 0 显示的数字（十进制），digital_data[7:4]对应数码管 1 显示的数字 |
| led_data | 1f20f300 | 输出 16 位 LED 灯信号 |
| switch_data | 1f20f400 | 只读，输入的 32 位拨码开关值 |

通过对 Confreg 的寄存器访存操作，就可实现对外部按键、拨码开关、LED、数码管、外部定时器的控制。

### 3.2.1 DVI 控制器

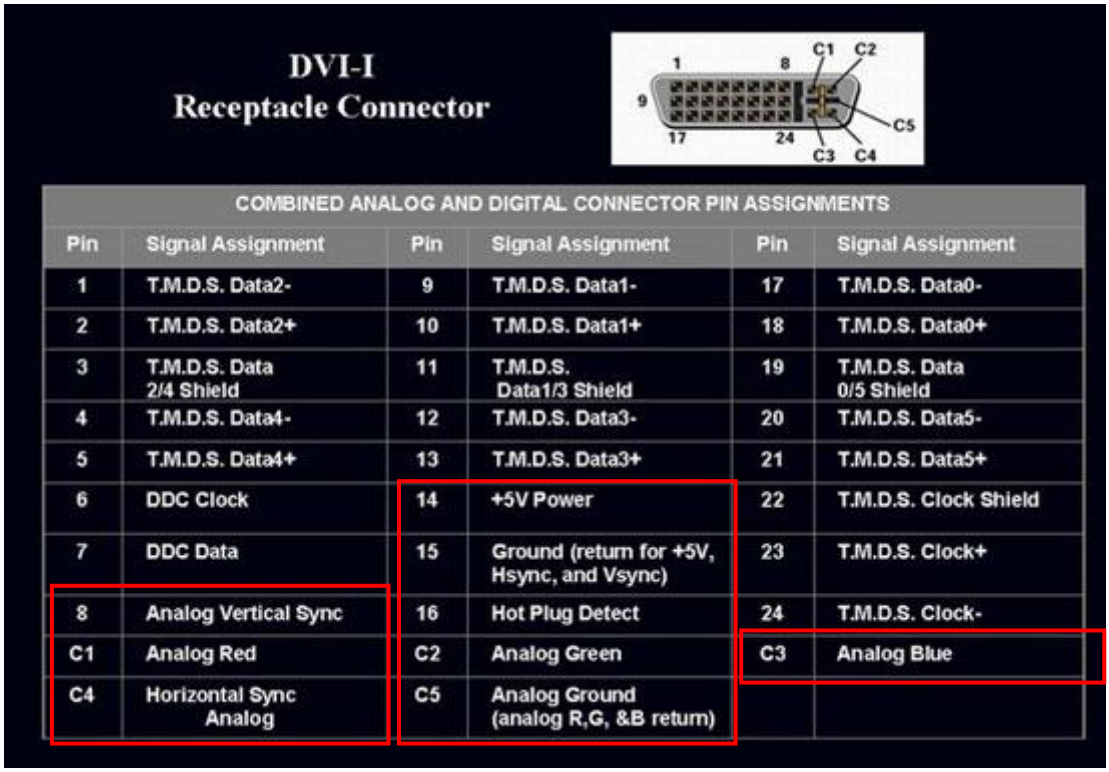
源码位于 rtl/ip/DVI 目录下。

远程 FPGA 提供了 DVI-A 视频接口用于图像显示。DVI 接口分为 DVI-A、DVI-D、DVI-I 三种。

DVI-A ( A= Analog )是模拟信号接口，只能去接 DVI-A 或者 VGA 接口的信号。

DVI-D ( D= Digital )是数字信号接口，只能去接 DVI-D 接口的信号。

DVI-I ( I = A+D = Integrated )含及上述两个接口，在管脚定义上有明显的区分。下面给出 DVI-I 的端口定义，DVI-A 仅使用其中红色框出的模拟信号部分。




COMBINED ANALOG AND DIGITAL CONNECTOR PIN ASSIGNMENTS

| Pin | Signal Assignment | Pin | Signal Assignment | Pin | Signal Assignment |
|---|---|---|---|---|---|
| 1 | T.M.D.S. Data2- | 9 | T.M.D.S. Data1- | 17 | T.M.D.S. Data0- |
| 2 | T.M.D.S. Data2+ | 10 | T.M.D.S. Data1+ | 18 | T.M.D.S. Data0+ |
| 3 | T.M.D.S. Data 2/4 Shield | 11 | T.M.D.S. Data1/3 Shield | 19 | T.M.D.S. Data 0/5 Shield |
| 4 | T.M.D.S. Data4- | 12 | T.M.D.S. Data3- | 20 | T.M.D.S. Data5- |
| 5 | T.M.D.S. Data4+ | 13 | T.M.D.S. Data3+ | 21 | T.M.D.S. Data5+ |
| 6 | DDC Clock | 14 | +5V Power | 22 | T.M.D.S. Clock Shield |
| 7 | DDC Data | 15 | Ground (return for +5V, Hsync, and Vsync) | 23 | T.M.D.S. Clock+ |
| 8 | Analog Vertical Sync | 16 | Hot Plug Detect | 24 | T.M.D.S. Clock- |
| C1 | Analog Red | C2 | Analog Green | C3 | Analog Blue |
| C4 | Horizontal Sync Analog | C5 | Analog Ground (analog R,G, &B return) | | |

**图 3-1　DVI-I 端口定义**

DVI-A 信号可以经过线缆进行转接后连接 VAG。对于数字电路来说，DVI-A 控制器和 VGA 控制器本身是一致的。下面给出 DVI-A 控制器的接口，FPGA 输出的像素信息是数字信号，板上配有数模转换芯片将其转为模拟信号最终输出至 DVI-A 端口。

```
1.    //图像输出信号
2.    output [2:0]    video_red,          //红色像素，3 位
3.    output [2:0]    video_green,        //绿色像素，3 位
4.    output [1:0]    video_blue,         //蓝色像素，2 位
5.    output          video_hsync,        //行同步（水平同步）信号
6.    output          video_vsync,        //场同步（垂直同步）信号
7.    output          video_clk,          //像素时钟输出
8.    output          video_de,           //行数据有效信号，用于区分消隐区
```

下面来介绍这些视频信号是如何控制显示器的。

显示器扫描方式是从屏幕左上角一点开始，从左向右逐点扫描，每扫描完一行,电子束回到屏幕的左边下一行的起始位置，在这期间，CRT 对电子束进行消隐，每行结束时，用行同步信号进行同步；当扫描完所有的行，形成一帧，用场同步信号进行场同步，并使扫描回到屏幕左上方，同时进行场消隐,开始下一帧。完成一行扫描的时间称为水平扫描时间，其倒数称为行频率；完成一帧（整屏）扫描的时间称为垂直扫描时间，其倒数称为场频率，即屏幕的刷新频率，常见的有 60Hz，75Hz 等等。其扫描示意图如下图所示。

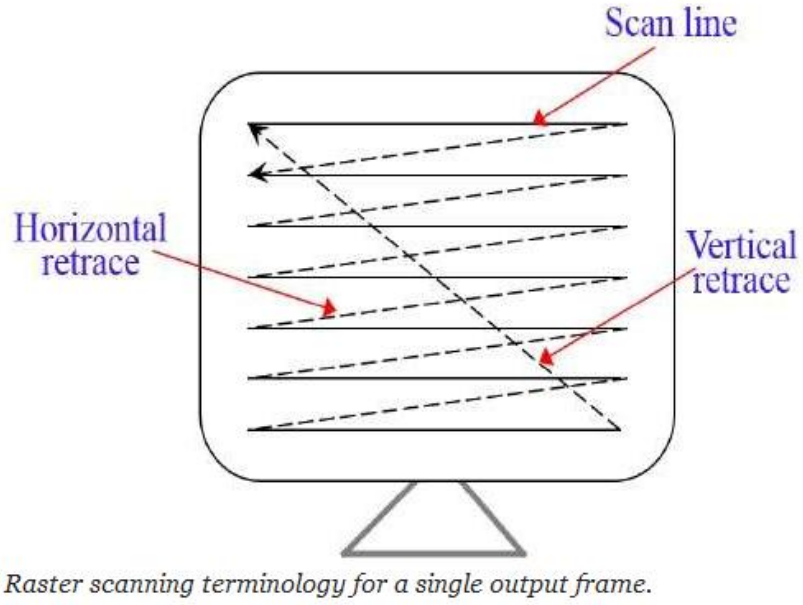



**图 3-2　显示器扫描方式**

一个完整的行扫描周期包含 6 部分：Sync（同步）、Back Porch（后沿）、Left Border（左边框）、“Addressable” Video（有效图像）、Right Border（右边框）、FrontPorch（前沿），这 6 部分的基本单位是 pixel（像素），即一个像素时钟周期。

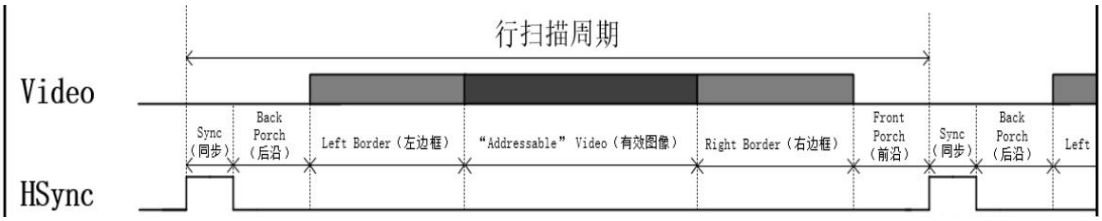



**图 3-3　行扫描周期**

一个完整的场扫描周期，也包含 6 部分：Sync（同步）、Back Porch（后沿）、Top Border（上边框）、“Addressable” Video（有效图像）、Bottom Border（底边框）、FrontPorch（前沿），这 6 部分的基本单位是 line（行），即一个完整的行扫描周期。

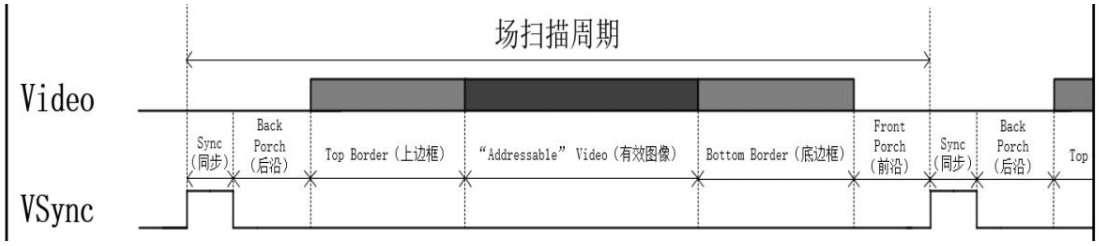



**图 3-4　场扫描周期**

行同步时序图与场同步时序图结合起来就构成了 DVI-A 时序图。橙色区域，是 DVI-A 图像显示区域.

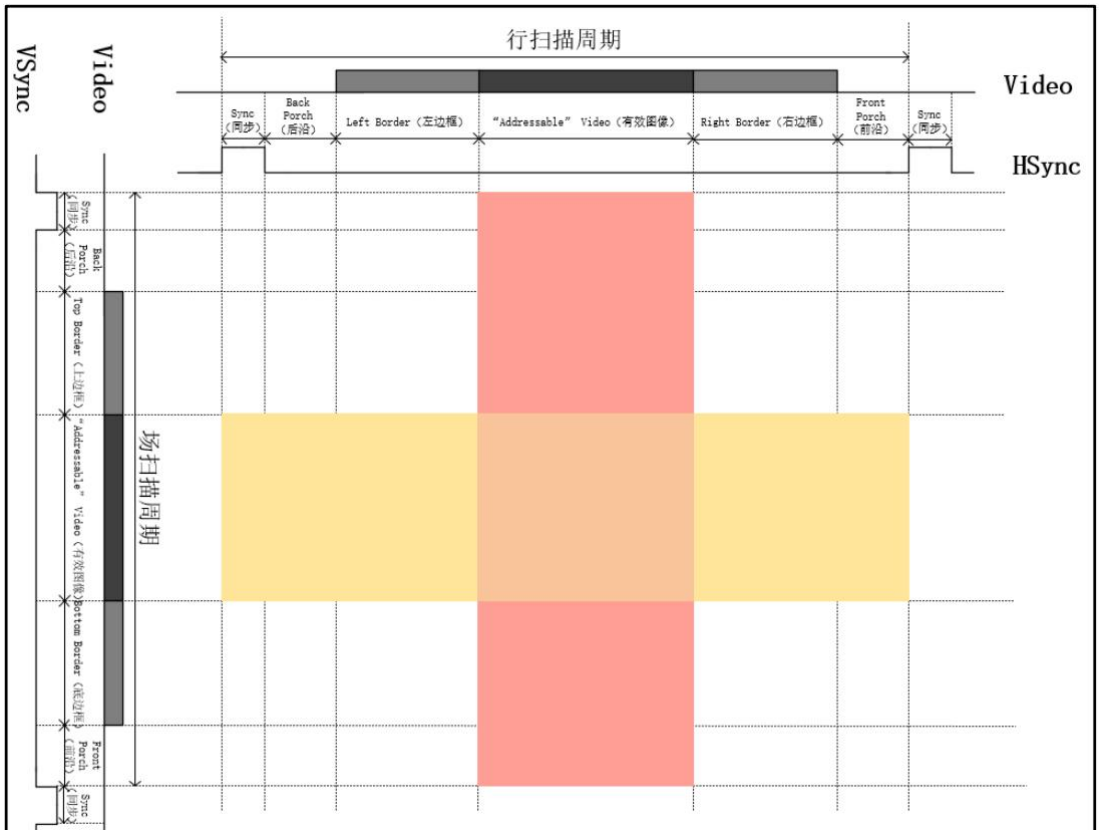



**图 3-5　DVI-A 时序图**

进行 DVI 控制器设计时，对于行扫描周期，可以视为从第 0 个时钟开始先发送有效图像，等到第 HSIZE 个时钟周期时，停止发送有效图像，第 HFP 个时钟周期时（等待右边框＋前沿）拉高行同步信号，第 HSP 个时钟周期时拉低行同步信号，第 HMAX 个时钟周期后开启下一个周期，继续发送有效图像。

对于场扫描周期上述类似，只不过场计数器不是每个时钟周期加一，而是等到行计数到达 HMAX 后才进行加一。场计数值也是从 0 开始发送有效图像，等计数值到达 VSIZE 后停止发送有效图像，计数值到达 VFP 时（等待底边框＋前沿）拉高场同步信号，计数值到达 VSP 时拉低场同步信号，计数值到达 VMAX 时开启下一个周期，继续发送有效图像。

只有行周期和场周期都在发送有效图像时，才输出 video_de 为高电平，即不在消隐区。对于远程 FPGA，图像输出时钟为 50MHZ 时，典型的图像输出参数配置如下：

```
1.    parameter HFP = 856,     // Horizontal front porch
2.    parameter HSP = 976,     // Horizontal sync pulse
3.    parameter HMAX = 1040,   // Horizontal total size
4.    parameter VSIZE = 600,   // Vertical size of visible area
5.    parameter VFP = 637,     // Vertical front porch
6.    parameter VSP = 643,     // Vertical sync pulse
7.    parameter VMAX = 666,    // Vertical total size
```

在 rtl/ip/DVI 目录下已经提供了 DVI 控制器，其基于龙芯杯优秀参赛作品改

写得到，后面使用的弹球小游戏也是基于该参赛作品改写获得，项目源链接为：

https://gitee.com/cui-zhekai/loongson_soc

下面给出该 DVI 控制器的寄存器描述。

**表 3-2 DVI 寄存器总览**

| 名称 | 地址 | 描述 |
|---|---|---|
| DVI_RECT_DIR | 1f100000 | 矩形显示寄存器 |
| DVI_RECT_L_W | 1f100004 | 矩形长宽 |
| DVI_SQU_DIR | 1f100008 | 圆形显示寄存器 |
| DVI_SQU_R | 1f10000c | 圆形半径 |

通过对 DVI 的寄存器访存操作，可以实现在显示屏上用红色画出一条实心矩形和一个实心圆形。

## 3.3 实验所用嵌入式软件

本次实验将在功能仿真和 FPGA 验证中使用两个不同的测试软件。功能仿真使用 sdk/software/examples/int_test 目录下的中断测试软件，FPGA 验证使用 sdk/software/examples/pinball_game 目录下的弹球游戏软件。

### 3.3.1 中断测试软件

中断测试软件主要用于检测外部中断控制器管理的按键中断和外部定时器中断是否能够正常进入中断服务函数并清中断。

中断测试软件将先初始化 CSR 使能处理器核外部中断处理。

```
1.  /* init exception */
2.  la.local t0, trap_handler
3.  csrwr    t0, csr_eentry
4.
5.  #clear int
6.  li.w        t1, 0x1
7.  csrwr       t1, csr_ticlr
8.
9.  #enable int
10. li.w        t1, 0x4
11. csrxchg     t1, t1, csr_crmd
12. li.w        t1, 0x1fff
13. csrwr       zero, csr_ecfg
14. csrwr       t1, csr_ecfg
```

在 start.S 中的处理器核外部中断使能代码，首先初始化了 EENTRY，将函

数 trap_handler 作为例外服务函数入口，之后向 TICLR 写 1 确保处理器核内的定时器中断已被清除，向 CRMD[2]（全局中断使能）写 1 使能中断，最后向 ECFG 写 0x1ffff 使能所有中断源。完成这部分初始化后，当外部中断控制器向处理器核的 HWI0 发中断申请时，PC 便会跳转至 trap_handler 函数（位于 trap_handler.S）。

```
1.  trap_handler:
2.
3.      TRAP_ENTRY_SETJMP
4.
5.      csrrd   a0, csr_estat
6.      csrrd   a1, csr_era
7.
8.      srli.w  t0, a0, 16
9.      bne     zero, t0, is_exceptions
10.     andi    t0, a0, 0x3FC
11.     srli.w  t1, t0, 2
12.
13.     # 根据中断号跳转到相应的处理程序 priority
14.     move t2, t1
15.     andi t2, t2, 0x1
16.     li.w t8, 0x1
17.     beq  t2, t8, handle_hwi0
18.
19.     move t2, t1
20.     andi t2, t2, 0x2
21.     li.w t8, 0x2
22.     beq  t2, t8, handle_hwi1
23.
24.     ......
25.
26.     b trap_jump_exit
27.
28. is_exceptions:
29.     addi.w t8, a1, 4
30.     csrwr t8, csr_era
31. 1:
32.     b 1b
33.
34. trap_jump_exit:
35.     TRAP_EXIT_LONGJMP
36.
37.
38. # HWI0 处理程序
39. handle_hwi0:
40.
41.     bl HWI0_IntrHandler
42.
43.     b trap_jump_exit
```

在 trap_handler 函数中首先进行保存现场，将寄存器堆数据压栈，之后读取 CSR.ESTAT，根据 Ecode 和 EsubCode 的值判断是 exception 还是 interrupt，若是 exception 则跳转至图中第 28 行 is_exceptions，将 CSR.ERA 加 0x4 后陷入死循环。若为 interrupt 则判断硬中断号，跳转至对应的 handle_hwix。本次实验中将外部中断控制器输出的中断请求连接至 HWI0，因此触发外部中断后应掉转至 handle_hwi0 再跳转至 HWI0_IntrHandler。

在 main.c 中需要对外部中断控制器和外部定时器初始化，下面是初始化代码，通过配置寄存器完成，定时器中断设置为每 0.1ms 触发一次。

```
RegWrite(0xbf20f004,0x0f);//edge
RegWrite(0xbf20f008,0x1f);//pol
RegWrite(0xbf20f00c,0x1f);//clr
RegWrite(0xbf20f000,0x1f);//en

RegWrite(0xbf20f104,0x5000);//timercmp 0.1ms
RegWrite(0xbf20f108,0x1);//timeren
```

进入 HWI0_IntrHandler 后的处理也在 main.c 中，首先读取外部中断控制器的 int_state 判断是外部定时器中断/按键中断，外部中断控制器输入中断的 bit3-bit0 是按键，bit4 是外部定时器，int_state 若为 0x10 则为外部定时器中断，若低四位中有高电平则是按键中断。每触发一次定时器中断会串口输出“a=5”，每触发一次按键中断会输出“a=对应按键值”。

```
void HWI0_IntrHandler(void)
{
    unsigned int int_state;
    int_state = RegRead(0xbf20f014);

    if((int_state & 0x10) == 0x10){
        Timer_IntrHandler();
    }
    else if(int_state & 0xf){
        Button_IntrHandler(int_state & 0xf);
    }
}

void Timer_IntrHandler(void)
{
    RegWrite(0xbf20f108,0);
    RegWrite(0xbf20f108,1);
    a = 5;
    printf("a=%d\n",a);
}

void Button_IntrHandler(unsigned char button_state)
{
    if((button_state & 0b1000) == 0b1000){
        a = 4;
        printf("a=%d\n",a);
        RegWrite(0xbf20f00c,0x8);//clr
    }
    else if((button_state & 0b0100) == 0b0100){
        a = 3;
        printf("a=%d\n",a);
        RegWrite(0xbf20f00c,0x4);//clr
    }
    else if((button_state & 0b0010) == 0b0010){
        a = 2;
        printf("a=%d\n",a);
        RegWrite(0xbf20f00c,0x2);//clr
    }
    else if((button_state & 0b0001) == 0b0001){
        a = 1;
        printf("a=%d\n",a);
        RegWrite(0xbf20f00c,0x1);//clr
    }
}
```

### 3.3.2 弹球游戏软件

弹球游戏所使用的中断服务函数名称与中断测试软件一致，只是中断服务函

数的内容用于支持游戏进行。另外弹球游戏软件提供了对 DVI 控制器的驱动文件 dvi.c 和 dvi.h，对数码管的驱动文件 seg7.c 和 seg7.h，对 led 灯的驱动文件 led.c 和 led.h。在 main 函数中，首先进行了外部中断控制器的初始化，之后进入 chooseTime 函数。ChooseTime 用于选择弹球游戏的“弹球”移动快慢，设置的 delay_time 时间值越大则延迟越高，弹球移动速度越慢。设置 delay_time 是依赖的按键中断服务函数，按下最右边的 BTN1 增加 delay_time，按下 BTN4 减少 delay_time，按下 BTN3 确认进入游戏。在游戏过程中 main 函数的 while(1)循环用于更新弹球位置并更新 DVI 显示画面，按键中断服务函数用于根据 BTN1 和 BTN4 的情况调整矩形显示位置，外部定时器中断服务函数用于计算得分并显示在数码管上。另外 delay_ms 函数使用处理器核内部定时器实现。

DVI 显示驱动包含两个函数，分别是 DVI_Draw_Rect 和 DVI_Draw_SQU。DVI_Dram_Rect 传入的参数是矩形左下角起始坐标 x,y，以及矩形的长和宽；DVI_Draw_SQU 传入的参数是圆形的圆心坐标 x,y 和圆形的半径。这些值被配置进 DVI 控制器的寄存器后，就能在 DVI 输出图像中看到红色的矩形和圆形。

```
1.  // 设置坐标和颜色的绘图函数
2.  void DVI_Draw_Rect(uint32_t x, uint32_t y, uint32_t l, uint32_
t w)
3.  {
4.      // 创建坐标值，x 和 y 分别占用 12 位; width 和 height 用于定义
范围
5.      uint32_t coordinates = ((x & 0xFFFF)<<16) | (y & 0xFFFF);

6.
7.      uint32_t size = ((l & 0xFFFF)<<16) | (w & 0xFFFF);
8.
9.      // 写入坐标和颜色寄存器
10.     RegWrite(DVI_RECT_DIR, coordinates);
11.     RegWrite(DVI_RECT_L_W, size);
12. }
13.
14. // 在指定位置绘制一个点的函数
15. void DVI_Draw_SQU(uint32_t x, uint32_t y, uint32_t r)
16. {
17.     // 创建坐标值，x 和 y 分别占用 12 位; width 和 height 用于定义
范围
18.     uint32_t coordinates = ((x & 0xFFFF)<<16) | (y & 0xFFFF);

19.
20.     uint32_t size = ((r & 0xFFFF)<<16) | (r & 0xFFFF);
21.
22.     // 写入坐标和颜色寄存器
23.     RegWrite(DVI_SQU_DIR, coordinates);
24.     RegWrite(DVI_SQU_R, size);
25. }
```

## 3.4 实验流程

### 3.4.1 添加外部中断控制器

外部中断控制器可以接收来自各类外设的中断信号输入，支持高电平/低电平/上升沿/下降沿触发，输出的中断信号均为高电平信号。输出的 int_out 进行或操作后再经过延迟打拍处理（异步信号处理）送至处理器核的 HWI0。输入中断 int_in[3:0]接按键 touch_btn_data，int_in[4]接外部定时器中断 timer_int。外部中断控制器的信号列表和寄存器列表在下面给出。

**表 3-3 外部中断控制器信号列表**

| 信号 | IO | 描述 |
|---|---|---|
| clk | input | 输入时钟 |
| resetn | input | 输入复位信号低有效 |
| int_en[31:0] | input | 输入的 32 位中断使能，为 1 表示使能比特位对应设备中断 |
| int_edga[31:0] | input | 为 1 表示边沿触发中断,为 0 表示电平触发 |
| int_pol[31:0] | input | 当 int_edge 配置成电平触发时,为 1 表示高电平触发中断,为 0 表示低电平触发；当 int_edge 配置成边沿触发时,为 1 表示上升沿触发中断,为 0 表示下降沿触发 |
| int_in[31:0] | input | 输入的 32 位中断信号 |
| int_clr[31:0] | input | 为 1 表示清除中断（针对边沿触发的中断） |
| int_set[31:0] | input | 为 1 表示中断置位（针对边沿触发的中断） |
| int_out[31:0] | output | 输出的 32 位中断信号 |

**表 3-4　外部中断控制器寄存器总览**

| 名称 | 地址 | 描述 |
|---|---|---|
| confreg_int_en | 1f20f000 | 中断使能寄存器，为 1 表示使能比特位对应设备中断 |
| confreg_int_edge | 1f20f004 | 中断边沿选择寄存器，为 1 表示边沿触发中断,为 0 表示电平触发 |
| confreg_int_pol | 1f20f008 | 中断极性选择寄存器，当 int_edge 配置成电平触发时,为 1 表示高电平触发中断,为 0 表示低电平触发；<br>当 int_edge 配置成边沿触发时,为 1 表示上升沿触发中断,为 0 表示下降沿触发 |
| confreg_int_clr | 1f20f00c | 中断清除寄存器，为 1 表示清除中断（针对边沿触发的中断） |
| confreg_int_set | 1f20f010 | 中断置位寄存器，为 1 表示中断置位（针对边沿触发的中断） |
| confreg_int_state | 1f20f014 | 中断状态寄存器，为 1 表示比特位对应设备产生中断 |

### 3.4.2添加 DVI 控制器

在顶层 soc_top.v 中，添加实验环境提供的 DVI 控制器（rtl/ip/DVI/axi_dvi.v），连接至 AxiCrossbar 的 axiOut_3 端口。

首先删除 soc_top.v 中 axiOut_3 端口的空负载代码：

```
assign dvi_arready  = 1'b1;
assign dvi_rid      = 5'b0;
assign dvi_rdata    = 32'b0;
assign dvi_rresp    = 2'b0;
assign dvi_rlast    = 1'b0;
assign dvi_rvalid   = 1'b0;
assign dvi_awready  = 1'b1;
assign dvi_wready   = 1'b1;
assign dvi_bid      = 5'b0;
assign dvi_bresp    = 2'b0;
```

```
assign dvi_bvalid   = 1'b0;
```

之后在 soc_top.v 中添加下述代码进行 DVI 控制器的实例化：

```
axi_dvi  u_axi_dvi (
    .s_awvalid                  ( dvi_awvalid   ),
    .s_awaddr                   ( dvi_awaddr    ),
    .s_awid                     ( dvi_awid      ),
    .s_awlen                    ( dvi_awlen     ),
    .s_awsize                   ( dvi_awsize    ),
    .s_awburst                  ( dvi_awburst   ),
    .s_awlock                   ( dvi_awlock    ),
    .s_awcache                  ( dvi_awcache   ),
    .s_awprot                   ( dvi_awprot    ),
    .s_wvalid                   ( dvi_wvalid    ),
    .s_wdata                    ( dvi_wdata     ),
    .s_wstrb                    ( dvi_wstrb     ),
    .s_wlast                    ( dvi_wlast     ),
    .s_bready                   ( dvi_bready    ),
    .s_arvalid                  ( dvi_arvalid   ),
    .s_araddr                   ( dvi_araddr    ),
    .s_arid                     ( dvi_arid      ),
    .s_arlen                    ( dvi_arlen     ),
    .s_arsize                   ( dvi_arsize    ),
    .s_arburst                  ( dvi_arburst   ),
    .s_arlock                   ( dvi_arlock    ),
    .s_arcache                  ( dvi_arcache   ),
    .s_arprot                   ( dvi_arprot    ),
    .s_rready                   ( dvi_rready    ),
    .aclk                       ( sys_clk       ),
    .aresetn                    ( sys_resetn    ),

    .s_awready                  ( dvi_awready   ),
    .s_wready                   ( dvi_wready    ),
    .s_bvalid                   ( dvi_bvalid    ),
    .s_bid                      ( dvi_bid       ),
    .s_bresp                    ( dvi_bresp     ),
    .s_arready                  ( dvi_arready   ),
    .s_rvalid                   ( dvi_rvalid    ),
    .s_rdata                    ( dvi_rdata     ),
    .s_rid                      ( dvi_rid       ),
    .s_rresp                    ( dvi_rresp     ),
    .s_rlast                    ( dvi_rlast     ),
    .video_clk                  ( video_clk     ),
    .hsync                      ( video_hsync   ),
    .vsync                      ( video_vsync   ),
    .data_enable                ( video_de      ),
    .video_red                  ( video_red     ),
    .video_green                ( video_green   ),
    .video_blue                 ( video_blue    )
);
```

### 3.4.3 进行功能仿真和 FPGA 验证

功能仿真和 FPGA 验证流程与之前的实验基本一致，由于实验一时已经创建了 vivado 工程，这里的步骤简化如下：

①编译软件

软件目录位于 sdk/software/examples/int_test，在 Linux 中使用 cd 命令进入该目录，执行“make clean”和“make”命令完成软件编译。

②添加文件 int_ctrl.v 至工程中

打开实验一创建的 vivado 工程，点击 Sources 窗口的“+”按钮，选择“add or create design sources”,点击 Next，点击 Add Files，找到 int_ctrl.v，点击 OK，点击 Finish 完成添加。

③运行仿真

点击 vivado 的“run behavioral simulation”直接仿真即可看到实验现象。

④生成 Bit 文件

在 vivado 中直接点击 Generate Bitstream,生成`soc_top.bit`文件上传至远程 FPGA，并将 bin 文件烧录至 SRAM 即可看到 FPGA 验证结果。

后续实验中进行功能仿真和 FPGA 验证的流程与上述流程类似，之后不再赘述。

在功能仿真中，首先使用简单的中断和 DVI 测试软件进行验证，软件目录位于 sdk/software/examples/int_test。功能仿真会输出四个按键中断服务函数的打印内容“button1 int” “button2 int” “button3 int” “button4 int”,以及每隔 0.1ms 输出的“timer int”。大概输出八个“timer int”后出现“button1 int”。另外，可以观察 DVI 信号的输出，红色信号和行同步信号都有输出，场同步信号由于仿真时间较短暂时没有拉高，同步信号输出时，数据有效信号 vidoe_de 为 0。

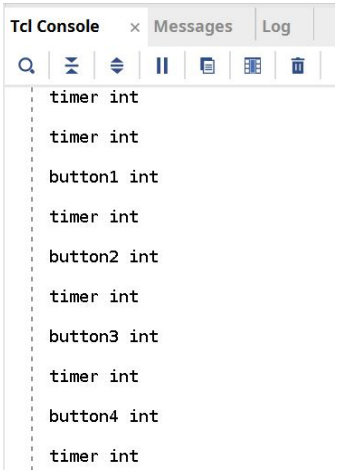



**图 3-6　int_test 串口输出仿真结果**

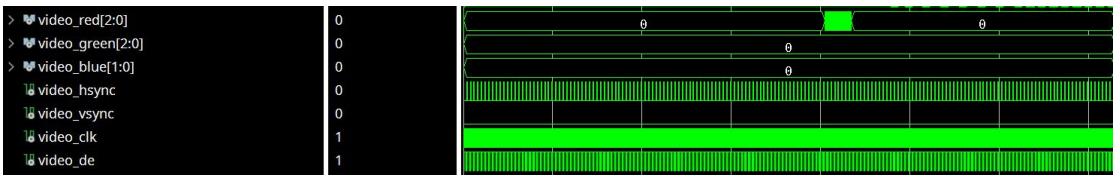



**图 3-7　int_test DVI 输出仿真结果**

完成 int_test 的仿真后可以直接生成 bit 文件进行 FPGA 验证，将 bit 文件和 bin 文件都烧录完成后，打开串口，可以看见一秒输出一次的“timer int”,按下按键会看到“buttonx int”。点击 FPGA 左侧的 Show，可以打开 DVI 输出界面，此时能看到 DVI 存在有效输出，在 800*600 分辨率的屏幕上画出了一个圆和一

个矩形。


**图 3-8　int_test FPGA 验证结果**

若 int_test 程序验证通过，弹球游戏一般也能运行成功，弹球游戏软件位于 sdk/software/examples/pinball_game。进入该目录“make clean” “make”后将 pinball_game.bin 下载至 SRAM，按下复位后串口输出“Please Choosetime !!”，之后按 BTN1 增加 delay_time，最大值为 20，delay_time 的数值会在数码管上显示，完成后按下 BTN3 开始游戏。此时 DVI 图像出现运动的红点和一条红线，按下 BTN1 使红线左移，按下 BTN4 使红线右移，接到一次球会使得得分加一，得分会在数码管中显示。

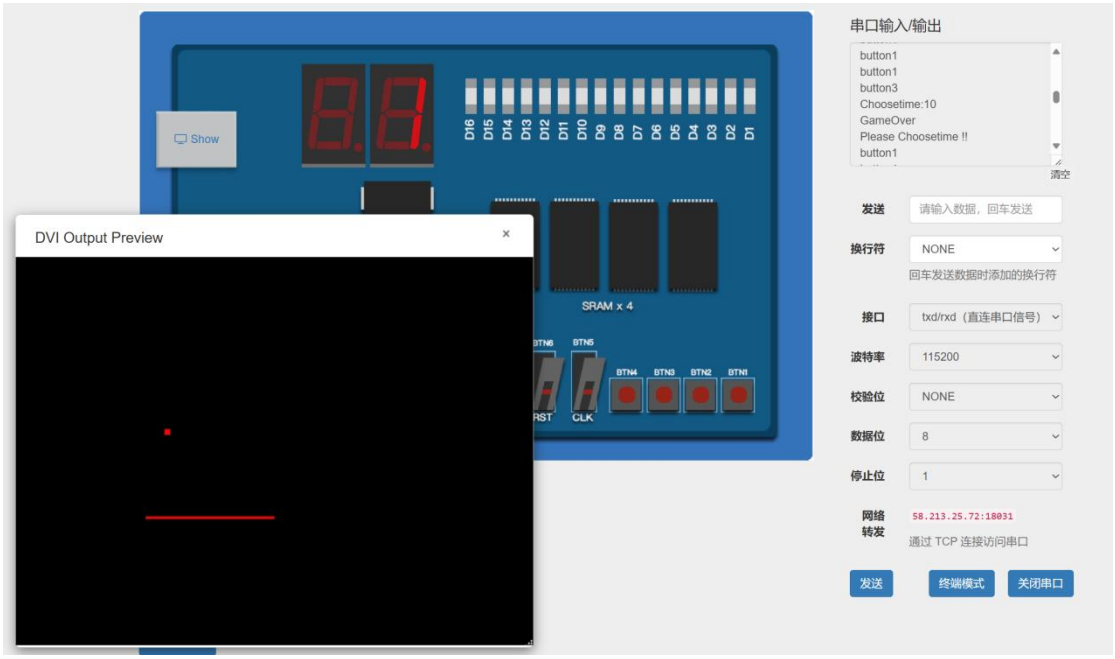


**图 3-9　弹球游戏**

# 附录一 向 Confreg 设计中添加外设寄存器

Confreg 可以作为一个简单的 AXI Slave 模板，下面简单介绍如何向 Confreg 中添加一个自己的外设寄存器。

Step1：定义外设寄存器基地址

```
1.    `define CONFREG_INT_ADDR    16'hf000 //1f20_f000
2.    `define TIMER_ADDR          16'hf100 //1f20_f100
3.    `define DIGITAL_ADDR        16'hf200 //1f20_f200
4.    `define LED_ADDR            16'hf300 //1f20_f300
5.    `define SWITCH_ADDR         16'hf400 //1f20_f400
```

在文件开头位置首先定义该外设寄存器的基地址低位，这里给出了已经定义好的外设寄存器基地址，这里定义的低 16 位地址拼接上 Confreg 在总线上的高位地址 0x1f20 就是外设寄存器完整的 32 位地址。

Step2：添加读信号

```
1.    wire [31:0] rdata_d =   buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'h0)     ?
confreg_int_en        :
2.                            buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'h4)     ?
confreg_int_edge      :
3.                            buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'h8)     ?
confreg_int_pol       :
4.                            buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'hc)     ?
confreg_int_clr       :
5.                            buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'h10)    ?
confreg_int_set       :
6.                            buf_addr[15:0] == (`CONFREG_INT_ADDR + 16'h14)    ?
confreg_int_state     :
7.                            buf_addr[15:0] == (`TIMER_ADDR + 16'h0)           ?
sys_timer             :
8.                            buf_addr[15:0] == (`TIMER_ADDR + 16'h4)           ?
sys_timer_cmp         :
9.                            buf_addr[15:0] == (`TIMER_ADDR + 16'h8)           ?
sys_timer_en          :
10.                           buf_addr[15:0] == (`DIGITAL_ADDR + 16'h0)         ?
digital_ctrl          :
11.                           buf_addr[15:0] == (`DIGITAL_ADDR + 16'h4)         ?
digital_data          :
12.                           buf_addr[15:0] == `LED_ADDR                       ?
led_data              :
13.                           buf_addr[15:0] == `SWITCH_ADDR                    ?
switch_data           :
14.                           32'd0;
```

在 rdata_d 处添加寄存器读信号，寄存器地址使用基地址加偏移量组成。

Step3：添加写使能信号和写逻辑

```
1.    wire write_timer_cmp = w_enter & (buf_addr[15:0]==`TIMER_ADDR+16'h4);
2.    wire write_timer_en  = w_enter & (buf_addr[15:0]==`TIMER_ADDR+16'h8);
3.
4.    always @(posedge aclk) begin
5.        if(!aresetn) begin
6.            sys_timer_cmp <= 32'h0;
7.        end
8.        else if (write_timer_cmp) begin
9.            sys_timer_cmp <= s_wdata;
10.       end
11.   end
```

写使能信号将确认握手有效信号 w_enter 和地址命中，写逻辑可根据具体需求自行设计。