
2025 集创赛龙芯中科杯 初赛发布包说明手册

目 录

第 1 章 初赛环境搭建	1
1.1 初赛环境介绍	1
1.2 安装所需软件	1
1.2.1 安装 vivado 2019.2	1
1.2.2 使用 WSL2 安装 Ubuntu 22.04.5	1
1.2.3 配置 SDK 中的工具链与 C 库	1
1.3 远程 FPGA 实验平台	2
第 2 章 初赛 SoC 搭建	3
2.1 初赛 SoC 搭建任务	3
2.2 初赛所用 IP 简介	3
2.2.1 OpenLA500 处理器核	3
2.2.2 AXI4 总线	5
2.2.3 SRAM 控制器	7
2.2.4 UART	9
2.2.5 Confreg	11
2.2.6 时钟与复位	12
2.3 SoC 结构	13
2.4 SoC 验证流程	15
2.4.1 尝试 Hello_World	15
2.4.2 功能测试程序	19
第 3 章 附录一 UART 模块说明	22
3.1 UART 传输协议	22
3.2 UART 寄存器描述	22
3.3 UART 初始化流程	28
3.4 AMBA2 APB 总线协议	29
第 4 章 附录二 向 Confreg 设计中添加外设寄存器	32

图目录

图 1-1	完成工具链安装	2
图 2-1	OpenLA500 处理器核架构	4
图 2-2	基础 SoC 架构	14
图 2-3	基础 SoC 时钟与复位网络	14
图 2-4	运行 make 进行编译	15
图 2-5	vivado 创建工程	16
图 2-6	vivado 进行 RTL 分析	16
图 2-7	vivado 运行功能仿真	17
图 2-8	hello_word 功能仿真通过	17
图 2-9	添加 debug 信号至波形窗口	18
图 2-10	反汇编文件	18
图 2-11	远程 FPGA 下载 SRAM	19
图 2-12	远程 FPGA 运行 Hello World 例程	19
图 2-13	仿真中功能测试程序的串口输出结果	20
图 2-14	仿真中功能测试程序的 LED 和数码管输出	20
图 2-15	功能测试程序的 FPGA 验证结果	21
图 2-16	功能测试程序的 FPGA 串口输出内容	21
图 3-1	UART 数据传输格式	22
图 3-2	APB 状态机	30
图 3-3	APB 写操作	30
图 3-4	APB 读操作	31

表目录

表 2-1	Confreg 寄存器总览	11
表 3-1	UART 寄存器总览	22
表 3-2	Receiver Buffer Register 位定义	23
表 3-3	Transmitter Holding Register 位定义	24
表 3-4	Interrupt Enable Register 位定义	24
表 3-5	Interrupt Identification Register 位定义	24
表 3-6	FIFO Control Register 位定义	25
表 3-7	Line Control Register 位定义	25
表 3-8	Line Status Register 位定义	26
表 3-9	Divisor Latch LSB 位定义	28
表 3-10	Divisor Latch MSB 位定义	28
表 3-11	APB SLAVE 信号列表	29

第 1 章 初赛环境搭建

1.1 初赛环境介绍

初赛发布包 la32r_soc_ciciec 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 安装所需软件

比赛所用 IDE 主要是 vivado 2019.2，可以直接在 windows 下安装。比赛所用测试程序需要在 Linux 环境下进行编译，推荐开发环境是 windows+WSL2，将编译需要的 gnu 工具链放在 WSL2 运行的 Ubuntu 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://loongson-neuq.pages.dev/p/vivado-2023.2%E5%AE%89%E8%A3%85/>

1.2.2 使用 WSL2 安装 Ubuntu 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>

1.2.3 配置 SDK 中的工具链与 C 库

完成 Ubuntu 的安装后，将初赛发布包文件夹中的 sdk 目录拷贝至 Ubuntu 的目录下。在 sdk 目录下有两个文件夹，software 文件夹包含比赛所用软件源码，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_ciciec/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。

```
$ sed -i '$a\export CICIEC_WINDOWS_HOME="/mnt/d/la32r_soc_ciciec"' ~/.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_ciciec/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-gnusr-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-gnusr-gcc --version
```

应能看到如下输出，证明安装成功：

```
root@LAPTOP5BE050RJ:/home/test1# loongarch32r-linux-gnusr-gcc --version
loongarch32r-linux-gnusr-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 1-1 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

网址：

<http://grs.nscscc.com:18000/signin>

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。下一章的“尝试 Hello_World”中将给出使用实例。

第 2 章 初赛 SoC 搭建

2.1 初赛 SoC 搭建任务

基于初赛发布包提供的处理器核、总线互联、SRAM 控制器、UART 和 Confreg 的 IP，可搭建出基础 SoC。基础 SoC 应能完成功能测试程序的正确执行，通过功能仿真和 FPGA 验证。

基于基础 SoC，参赛队可以自行扩展其他模块形成参赛 SoC。扩展模块的验证，需要参赛队自行向 sdk 中添加测试软件，通过串口打印或数码管显示等手段，在功能仿真和 FPGA 中展示出扩展模块的功能正确性。

2.2 初赛所用 IP 简介

初赛中用到的五款 IP 分别是：OpenLA500 处理器核、AXI4 总线、SRAM 控制器、UART 和 Confreg（包含按键、拨码开关、LED、数码管）。下面分别介绍这五款 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接结构的指令和数据 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。

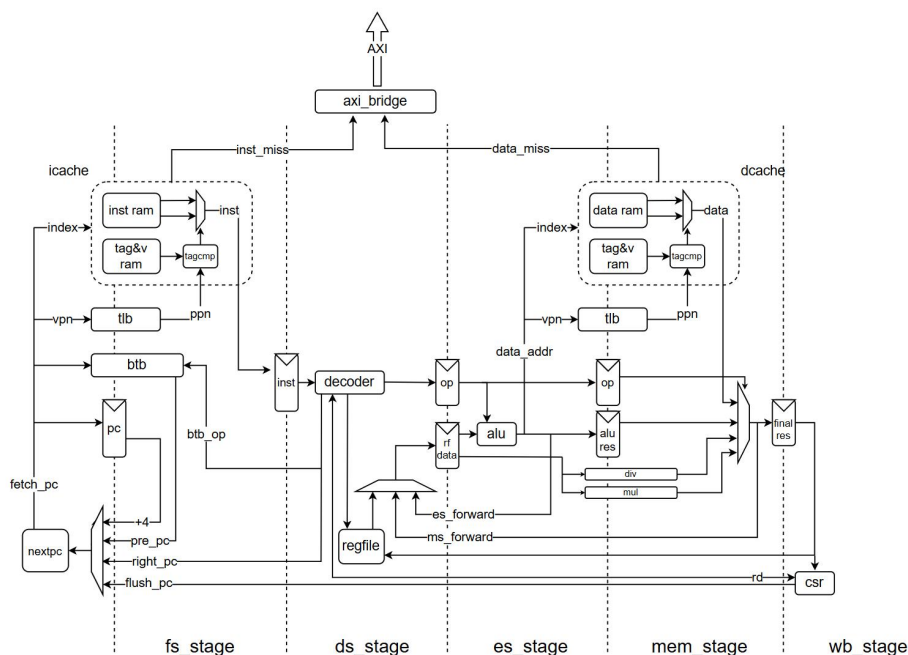


图 2-1 OpenLA500 处理器核架构

OpenLA500 已经过流片验证，130nm 工艺下频率为 100M，dhrystone，core mark 分数分别为 0.78 DMIPS/MHz，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。

Gitee 上的 OpenLA500 为实现 cache 使用了 xilinx ip 的 SRAM。在本次竞赛中不再使用该 xilinx ip，而是由文件 rtl/ip/ram_wrap/cache_sram.v 提供 Cache 所需的片上 SRAM，该文件使用综合推导式写法，可以使 vivado 进行综合时自动调用 Block RAM 实现。另外，为 OpenLA500 进行物理设计时，存在该 SRAM 会使得物理设计的复杂度更高，因此在 config.h 中有 USE_CACHE 宏控制是否存在该 SRAM。若希望删除该 SRAM，首先将 config.h 中的 `define USE_CACHE` 注释掉。之后在 sdk/software/bsp/common.mk 中将 `CFLAGS += -Dhas_cache=1` 改为 `CFLAGS += -Dhas_cache=0`。通过这种方式删除 SRAM 后，OpenLA500 的 Cache 不可用，后续进行 CoreMark 测试时跑分也将大幅降低。

OpenLA500 的实例化示例如下：

```

1. core_top u_cpu(
2.     //外部中断信号
3.     .intrpt    (8'h0),    //high active
4.
5.     .aclk      (cpu_clk    ),
6.     .aresetn   (cpu_resetcn),    //low active
7.
8.     //读地址通道
9.     .arid      (cpu_arid    ),
10.    .araddr    (cpu_araddr  ),
11.    .arlen     (cpu_arlen   ),
12.    .arsize    (cpu_arsize  ),
13.    .arburst   (cpu_arburst ),
14.    .arlock    (cpu_arlock  ),
15.    .arcache   (cpu_arcache ),

```



```

16. .arprot (cpu_arprot ),
17. .arvalid (cpu_arvalid ),
18. .arready (cpu_arready ),
19. //读数据通道
20. .rid (cpu_rid ),
21. .rdata (cpu_rdata ),
22. .rresp (cpu_rresp ),
23. .rlast (cpu_rlast ),
24. .rvalid (cpu_rvalid ),
25. .rready (cpu_rready ),
26. //写地址通道
27. .awid (cpu_awid ),
28. .awaddr (cpu_awaddr ),
29. .awlen (cpu_awlen ),
30. .awsize (cpu_awsiz ),
31. .awburst (cpu_awburst ),
32. .awlock (cpu_awlock ),
33. .awcache (cpu_awcache ),
34. .awprot (cpu_awprot ),
35. .awvalid (cpu_awvalid ),
36. .awready (cpu_awready ),
37. //写数据通道
38. .wid (cpu_wid ),
39. .wdata (cpu_wdata ),
40. .wstrb (cpu_wstrb ),
41. .wlast (cpu_wlast ),
42. .wvalid (cpu_wvalid ),
43. .wready (cpu_wready ),
44. //写响应通道
45. .bid (cpu_bid ),
46. .bresp (cpu_bresp ),
47. .bvalid (cpu_bvalid ),
48. .bready (cpu_bready ),
49.
50. //已弃用
51. .break_point (1'b0 ),
52. .infor_flag (1'b0 ),
53. .reg_num (5'b0 ),
54. .ws_valid ( ),
55. .rf_rdata ( ),
56.
57. //写回阶段提供的调试信息
58. .debug0_wb_pc (debug_wb_pc ),
59. .debug0_wb_inst (debug_wb_inst ),
60. .debug0_wb_rf_wen (debug_wb_rf_wen ),
61. .debug0_wb_rf_wnum (debug_wb_rf_wnum ),
62. .debug0_wb_rf_wdata (debug_wb_rf_wdata )
63. );

```

OpenLA500 的对外接口主要是 AXI master 接口，对应上面代码中的读地址通道、读数据通道、写地址通道、写数据通道、写响应通道；另外还有外部中断接口 intrp[7:0]和输出的调试信息 debug0_wb_xx。基于 OpenLA500 搭建 SoC 的关键就在于构造 AXI 总线互联，使得 OpenLA500 通过 AXI master 接口可以访问到板载 SRAM、UART 模块、Confreg 模块。

2.2.2 AXI4 总线

源码位于 rtl/ip/Bus_interconnects 目录下。包含的文件有：

- AxiCrossbar_1x4.v：总线互联模块

- Axi_CDC.v: 总线跨时钟域模块
- axi2sram_sp_ext.v: AXI 转外部 SRAM 桥，是外部 SRAM 控制器的主要模块
- axi2sram_dp.v: AXI 转片上伪双端口 SRAM 桥
- axi2sram_sp.v: AXI 转片上单端口 SRAM 桥

AxiCrossbar_1x4.v 可以连接一个 Master 和四个 Slave。四个 Slave 的地址分布为:

axiOut_0 : 1c000000 – 1c7fffff 8Mbytes (SRAM)
 axiOut_1 : 1f000000 – 1f0fffff 1Mbytes (apb-UART)
 axiOut_2 : 1f100000 – 1f0fffff 1Mbytes (reserved)
 axiOut_3 : 1f200000 – 1f0fffff 1Mbytes (Confreg)

其中 axiOut_0 用来连接 SRAM 控制器, axiOut_1 连接 axi 转 apb 桥后连接 U
 ART, axiOut_2 保留, axiOut_3 连接 Confreg。

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_CDC（Axi_CDC.v）。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。

下面给出 AxiCrossbar_1x4 的部分实例化示例:

```

1. AxiCrossbar_1x4 u_AxiCrossbar_1x4 (
2.   .clk          ( sys_clk          ),
3.   .resetn       ( sys_resetn      ),
4.
5.   //master 0
6.   //aw
7.   .axiIn_awvalid ( cpu_sync_awvalid ),
8.   .axiIn_awready ( cpu_sync_awready ),
9.   .axiIn_awaddr  ( cpu_sync_awaddr  ),
10.  .axiIn_awid    ( cpu_sync_awid    ),
11.  .axiIn_awlen   ( cpu_sync_awlen   ),
12.  .axiIn_awszise ( cpu_sync_awszise ),
13.  .axiIn_awburst ( cpu_sync_awburst ),
14.  .axiIn_awlock  ( cpu_sync_awlock  ),
15.  .axiIn_awcache ( cpu_sync_awcache ),
16.  .axiIn_awprot  ( cpu_sync_awprot  ),
17.  //w
18.  .axiIn_wvalid  ( cpu_sync_wvalid  ),
19.  .axiIn_wready  ( cpu_sync_wready  ),
20.  .axiIn_wdata   ( cpu_sync_wdata   ),
21.  .axiIn_wstrb   ( cpu_sync_wstrb   ),
22.  .axiIn_wlast   ( cpu_sync_wlast   ),
23.  //b
24.  .axiIn_bready  ( cpu_sync_bready  ),
25.  .axiIn_bvalid  ( cpu_sync_bvalid  ),
26.  .axiIn_bid     ( cpu_sync_bid     ),
27.  .axiIn_bresp   ( cpu_sync_bresp   ),
28.  //ar
29.  .axiIn_arvalid ( cpu_sync_arvalid ),
30.  .axiIn_arready ( cpu_sync_arready ),
31.  .axiIn_araddr  ( cpu_sync_araddr  ),
32.  .axiIn_arid    ( cpu_sync_arid    ),

```

```

33.     .axiIn_arlen      ( cpu_sync_arlen      ),
34.     .axiIn_arsize     ( cpu_sync_arsize     ),
35.     .axiIn_arburst    ( cpu_sync_arburst    ),
36.     .axiIn_arlock     ( cpu_sync_arlock     ),
37.     .axiIn_arcache    ( cpu_sync_arcache    ),
38.     .axiIn_arprot     ( cpu_sync_arprot     ),
39.     //r
40.     .axiIn_rvalid     ( cpu_sync_rvalid     ),
41.     .axiIn_rready     ( cpu_sync_rready     ),
42.     .axiIn_rdata      ( cpu_sync_rdata      ),
43.     .axiIn_rid        ( cpu_sync_rid        ),
44.     .axiIn_rresp      ( cpu_sync_rresp      ),
45.     .axiIn_rlast      ( cpu_sync_rlast      ),
46.
47.
48.     //slave 0
49.     //aw
50.     .axiOut_0_awvalid  ( ram_awvalid      ),
51.     .axiOut_0_awready  ( ram_awready      ),
52.     .axiOut_0_awaddr   ( ram_awaddr       ),
53.     .axiOut_0_awid     ( ram_awid         ),
54.     .axiOut_0_awlen    ( ram_awlen        ),
55.     .axiOut_0_awsz     ( ram_awsz         ),
56.     .axiOut_0_awburst  ( ram_awburst      ),
57.     .axiOut_0_awlock   ( ram_awlock       ),
58.     .axiOut_0_awcache  ( ram_awcache      ),
59.     .axiOut_0_awprot   ( ram_awprot       ),
60.     //w
61.     .axiOut_0_wvalid   ( ram_wvalid       ),
62.     .axiOut_0_wready   ( ram_wready       ),
63.     .axiOut_0_wdata    ( ram_wdata        ),
64.     .axiOut_0_wstrb    ( ram_wstrb        ),
65.     .axiOut_0_wlast    ( ram_wlast        ),
66.     //b
67.     .axiOut_0_bready   ( ram_bready       ),
68.     .axiOut_0_bvalid   ( ram_bvalid       ),
69.     .axiOut_0_bid      ( ram_bid          ),
70.     .axiOut_0_bresp    ( ram_bresp        ),
71.     //ar
72.     .axiOut_0_arvalid  ( ram_arvalid      ),
73.     .axiOut_0_arready  ( ram_arready      ),
74.     .axiOut_0_araddr   ( ram_araddr       ),
75.     .axiOut_0_arid     ( ram_arid         ),
76.     .axiOut_0_arlen    ( ram_arlen        ),
77.     .axiOut_0_arsize   ( ram_arsize       ),
78.     .axiOut_0_arburst  ( ram_arburst      ),
79.     .axiOut_0_arlock   ( ram_arlock       ),
80.     .axiOut_0_arcache  ( ram_arcache      ),
81.     .axiOut_0_arprot   ( ram_arprot       ),
82.     //r
83.     .axiOut_0_rvalid   ( ram_rvalid       ),
84.     .axiOut_0_rready   ( ram_rready       ),
85.     .axiOut_0_rdata    ( ram_rdata        ),
86.     .axiOut_0_rid      ( ram_rid          ),
87.     .axiOut_0_rresp    ( ram_rresp        ),
88.     .axiOut_0_rlast    ( ram_rlast        ),
89.
90.     .....
91. );

```

2.2.3 SRAM 控制器

源码位于 rtl/ip/ram_wrap/axi_wrap_ram_sp_ext.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。

下面给出外部 SRAM 控制器的实例化示例：

```

1. //axi ram
2. axi_wrap_ram_sp_ext u_axi_ram (
3.     .aclk          ( sys_clk          ),
4.     .aresetn       ( sys_resetn       ),
5.     //ar
6.     .axi_arid       ( ram_arid        ),
7.     .axi_araddr     ( ram_araddr      ),
8.     .axi_arlen      ( ram_arlen       ),
9.     .axi_arsize     ( ram_arsize      ),
10.    .axi_arburst     ( ram_arburst     ),
11.    .axi_arlock      ( ram_arlock      ),
12.    .axi_arcache     ( ram_arcache     ),
13.    .axi_arprot      ( ram_arprot      ),
14.    .axi_arvalid     ( ram_arvalid     ),
15.    .axi_arready     ( ram_arready     ),
16.    //r
17.    .axi_rid         ( ram_rid         ),
18.    .axi_rdata       ( ram_rdata       ),
19.    .axi_rresp       ( ram_rresp       ),
20.    .axi_rlast       ( ram_rlast       ),
21.    .axi_rvalid      ( ram_rvalid      ),
22.    .axi_rready      ( ram_rready      ),
23.    //aw
24.    .axi_awid        ( ram_awid        ),
25.    .axi_awaddr      ( ram_awaddr      ),
26.    .axi_awlen       ( ram_awlen       ),
27.    .axi_awsiz       ( ram_awsiz       ),
28.    .axi_awburst     ( ram_awburst     ),
29.    .axi_awlock      ( ram_awlock      ),
30.    .axi_awcache     ( ram_awcache     ),
31.    .axi_awprot      ( ram_awprot      ),
32.    .axi_awvalid     ( ram_awvalid     ),
33.    .axi_awready     ( ram_awready     ),
34.    //w
35.    .axi_wdata       ( ram_wdata       ),
36.    .axi_wstrb       ( ram_wstrb       ),
37.    .axi_wlast       ( ram_wlast       ),
38.    .axi_wvalid      ( ram_wvalid      ),
39.    .axi_wready      ( ram_wready      ),
40.    //b
41.    .axi_bid         ( ram_bid         ),
42.    .axi_bresp       ( ram_bresp       ),
43.    .axi_bvalid      ( ram_bvalid      ),
44.    .axi_bready      ( ram_bready      ),

```

```

45.
46.     .base_ram_addr ( base_ram_addr      ),
47.     .base_ram_be_n ( base_ram_be_n      ),
48.     .base_ram_ce_n ( base_ram_ce_n      ),
49.     .base_ram_oe_n ( base_ram_oe_n      ),
50.     .base_ram_we_n ( base_ram_we_n      ),
51.     .ext_ram_addr  ( ext_ram_addr       ),
52.     .ext_ram_be_n  ( ext_ram_be_n       ),
53.     .ext_ram_ce_n  ( ext_ram_ce_n       ),
54.     .ext_ram_oe_n  ( ext_ram_oe_n       ),
55.     .ext_ram_we_n  ( ext_ram_we_n       ),
56.
57.     .base_ram_data ( base_ram_data      ),
58.     .ext_ram_data  ( ext_ram_data       )
59.
60.);

```

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 apb 转接桥的顶层模块。UART 模块使用 AXI 接口连接至 AXI_Crossbar 的 axiOut_1，地址空间为 1f000000-1f0fffff。

下面给出 UART 模块的实例化示例：

```

1. //uart
2. wire UART_CTS,   UART_RTS;
3. wire UART_DTR,   UART_DSR;
4. wire UART_RI,    UART_DCD;
5. assign UART_CTS = 1'b0;
6. assign UART_DSR = 1'b0;
7. assign UART_DCD = 1'b0;
8. assign UART_RI  = 1'b0;
9. wire uart0_int  ;
10. wire uart0_txd_o ;
11. wire uart0_txd_i ;
12. wire uart0_txd_oe;
13. wire uart0_rxd_o ;
14. wire uart0_rxd_i ;
15. wire uart0_rxd_oe;
16. wire uart0_rts_o ;
17. wire uart0_cts_i ;
18. wire uart0_dsr_i ;
19. wire uart0_dcd_i ;
20. wire uart0_dtr_o ;
21. wire uart0_ri_i  ;
22. assign  UART_RX   = uart0_rxd_oe ? 1'bz : uart0_rxd_o ;
23. assign  UART_TX   = uart0_txd_oe ? 1'bz : uart0_txd_o ;
24. assign  UART_RTS  = uart0_rts_o ;
25. assign  UART_DTR  = uart0_dtr_o ;
26. assign  uart0_txd_i = UART_TX;
27. assign  uart0_rxd_i = UART_RX;
28. assign  uart0_cts_i = UART_CTS;
29. assign  uart0_dcd_i = UART_DCD;
30. assign  uart0_dsr_i = UART_DSR;
31. assign  uart0_ri_i  = UART_RI ;
32.
33. //UART_CONTROLLER
34. axi_uart_controller u_ axi_uart_controller
35. (
36.     .clk                (sys_clk                ),

```

```

37. .rst_n          (sys_resetsn      ),
38.
39. .axi_s_awid      (uart_awid      ),
40. .axi_s_awaddr    (uart_awaddr    ),
41. .axi_s_awlen     (uart_awlen     ),
42. .axi_s_awsz      (uart_awsz      ),
43. .axi_s_awburst   (uart_awburst   ),
44. .axi_s_awlock    (uart_awlock    ),
45. .axi_s_awcache   (uart_awcache   ),
46. .axi_s_awprot    (uart_awprot    ),
47. .axi_s_awvalid   (uart_awvalid   ),
48. .axi_s_awready   (uart_awready   ),
49. .axi_s_wid       (uart_wid       ),
50. .axi_s_wdata     (uart_wdata     ),
51. .axi_s_wstrb     (uart_wstrb     ),
52. .axi_s_wlast     (uart_wlast     ),
53. .axi_s_wvalid    (uart_wvalid    ),
54. .axi_s_wready    (uart_wready    ),
55. .axi_s_bid       (uart_bid       ),
56. .axi_s_bresp     (uart_bresp     ),
57. .axi_s_bvalid    (uart_bvalid    ),
58. .axi_s_bready    (uart_bready    ),
59. .axi_s_arid      (uart_arid      ),
60. .axi_s_araddr    (uart_araddr    ),
61. .axi_s_arlen     (uart_arlen     ),
62. .axi_s_arsize    (uart_arsize    ),
63. .axi_s_arburst   (uart_arburst   ),
64. .axi_s_arlock    (uart_arlock    ),
65. .axi_s_arcache   (uart_arcache   ),
66. .axi_s_arprot    (uart_arprot    ),
67. .axi_s_arvalid   (uart_arvalid   ),
68. .axi_s_arready   (uart_arready   ),
69. .axi_s_rid       (uart_rid       ),
70. .axi_s_rdata     (uart_rdata     ),
71. .axi_s_rresp     (uart_rresp     ),
72. .axi_s_rlast     (uart_rlast     ),
73. .axi_s_rvalid    (uart_rvalid    ),
74. .axi_s_rready    (uart_rready    ),
75.
76. .apb_rw_dma      (1'b0          ),
77. .apb_psel_dma    (1'b0          ),
78. .apb_enab_dma    (1'b0          ),
79. .apb_addr_dma    (20'b0         ),
80. .apb_valid_dma   (1'b0          ),
81. .apb_wdata_dma   (32'b0         ),
82. .apb_rdata_dma   (                ),
83. .apb_ready_dma   (                ),
84. .dma_grant        (                ),
85.
86. .dma_req_o        (                ),
87. .dma_ack_i        (1'b0          ),
88.
89. //UART0
90. .uart0_txd_i      (uart0_txd_i    ),
91. .uart0_txd_o      (uart0_txd_o    ),
92. .uart0_txd_oe     (uart0_txd_oe   ),
93. .uart0_rxd_i      (uart0_rxd_i    ),
94. .uart0_rxd_o      (uart0_rxd_o    ),
95. .uart0_rxd_oe     (uart0_rxd_oe   ),
96. .uart0_rts_o      (uart0_rts_o    ),
97. .uart0_dtr_o      (uart0_dtr_o    ),
98. .uart0_cts_i      (uart0_cts_i    ),
99. .uart0_dsr_i      (uart0_dsr_i    ),
100. .uart0_dcd_i      (uart0_dcd_i    ),
101. .uart0_ri_i       (uart0_ri_i     ),
102. .uart0_int        (uart0_int      )

```

103.);

2.2.5 Confreg

Confreg 用于控制 FPGA 板上的按键、拨码开关、LED、数码管。Confreg 使用 AXI 接口连接至 AXI_Crossbar 的 axiOut_3，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，下表给出寄存器清单。

表 2-1 Confreg 寄存器总览

名称	地址	描述
sys_timer	0x1f20f100	外部定时器计数值，只读
sys_timer_cmp	0x1f20f104	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	0x1f20f108	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	0x1f20f200	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	0x1f20f204	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	0x1f20f300	输出 16 位 LED 灯信号
switch_data	0x1f20f400	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。

下面给出 Confreg 的实例化示例：

```

1. confreg #(.SIMULATION(SIMULATION)) u_confreg (
2.     .aclk          ( sys_clk          ),
3.     .aresetn       ( sys_resetsn     ),
4.     .cpu_clk       ( cpu_clk         ),
5.     .cpu_resetsn   ( cpu_resetsn     ),
6.     .s_awid        ( confreg_awid    ),
7.     .s_awaddr      ( confreg_awaddr   ),
8.     .s_awlen       ( confreg_awlen   ),

```

```

9.      .s_awsz ( confreg_awsz ),
10.     .s_awburst ( confreg_awburst ),
11.     .s_awlock ( confreg_awlock ),
12.     .s_awcache ( confreg_awcache ),
13.     .s_awprot ( confreg_awprot ),
14.     .s_awvalid ( confreg_awvalid ),
15.     .s_wid ( confreg_wid ),
16.     .s_wdata ( confreg_wdata ),
17.     .s_wstrb ( confreg_wstrb ),
18.     .s_wlast ( confreg_wlast ),
19.     .s_wvalid ( confreg_wvalid ),
20.     .s_bready ( confreg_bready ),
21.     .s_arid ( confreg_arid ),
22.     .s_araddr ( confreg_araddr ),
23.     .s_arlen ( confreg_arlen ),
24.     .s_arsize ( confreg_arsize ),
25.     .s_arburst ( confreg_arburst ),
26.     .s_arlock ( confreg_arlock ),
27.     .s_arcache ( confreg_arcache ),
28.     .s_arprot ( confreg_arprot ),
29.     .s_arvalid ( confreg_arvalid ),
30.     .s_rready ( confreg_rready ),
31.
32.     .s_awready ( confreg_awready ),
33.     .s_wready ( confreg_wready ),
34.     .s_bid ( confreg_bid ),
35.     .s_bresp ( confreg_bresp ),
36.     .s_bvalid ( confreg_bvalid ),
37.     .s_arready ( confreg_arready ),
38.     .s_rid ( confreg_rid ),
39.     .s_rdata ( confreg_rdata ),
40.     .s_rresp ( confreg_rresp ),
41.     .s_rlast ( confreg_rlast ),
42.     .s_rvalid ( confreg_rvalid ),
43.
44.     .switch ( dip_sw ),
45.     .touch_btn ( touch_btn ),
46.     .led ( leds ),
47.     .dpy0 ( dpy0 ),
48.     .dpy1 ( dpy1 ),
49.     .confreg_int (
50. );

```

其中 confreg_int 是预留的中断控制器输出信号，暂时不用接。

2.2.6 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。

如果参赛 SoC 的需要更多的时钟域，请自行修改 PLL IP 核的配置。

SoC 的复位信号使用 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` 时钟域模块和 `cpu_clk` 时钟域模块使用了。

2.3 SoC 结构

在 `rtl/soc_top.v` 中提供了空白的顶层模板，请将模块 `core_top`、`Axi_CDC`、`AxiCrossbar_1x4`、`axi_wrap_ram_sp_ext`、`axi_uart_controller`、`confreg` 以及自行扩展的模块正确连接形成 SoC 顶层文件。

基础 SoC 顶层架构如下图所示：

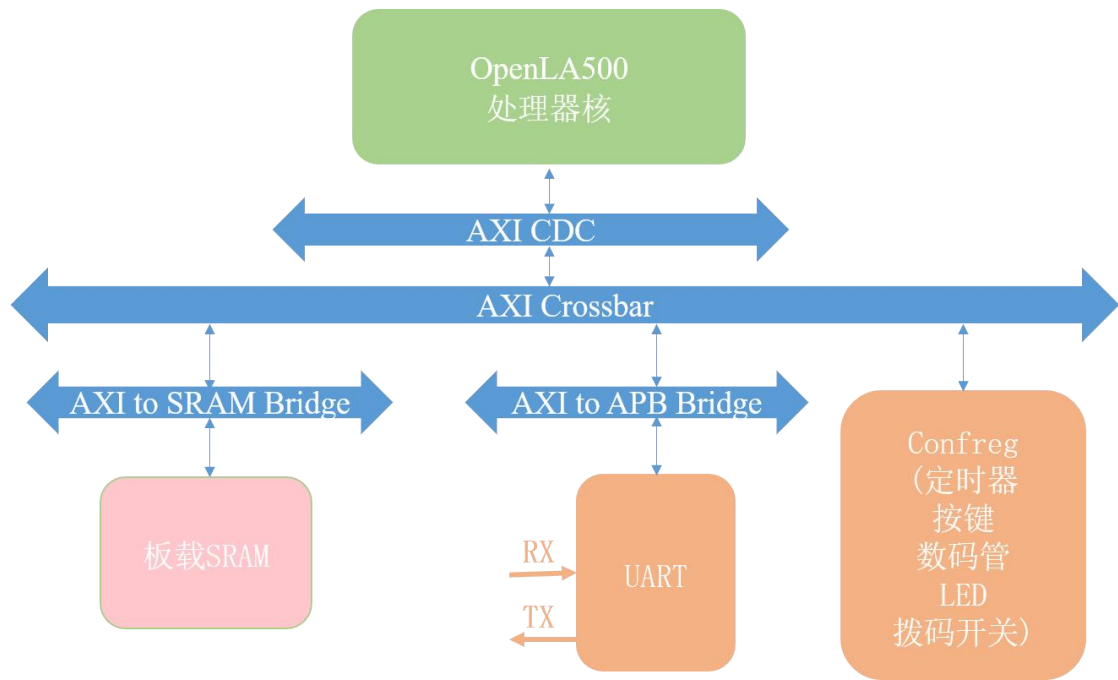


图 2-2 基础 SoC 架构

基础 SoC 的时钟与复位网络见下图：

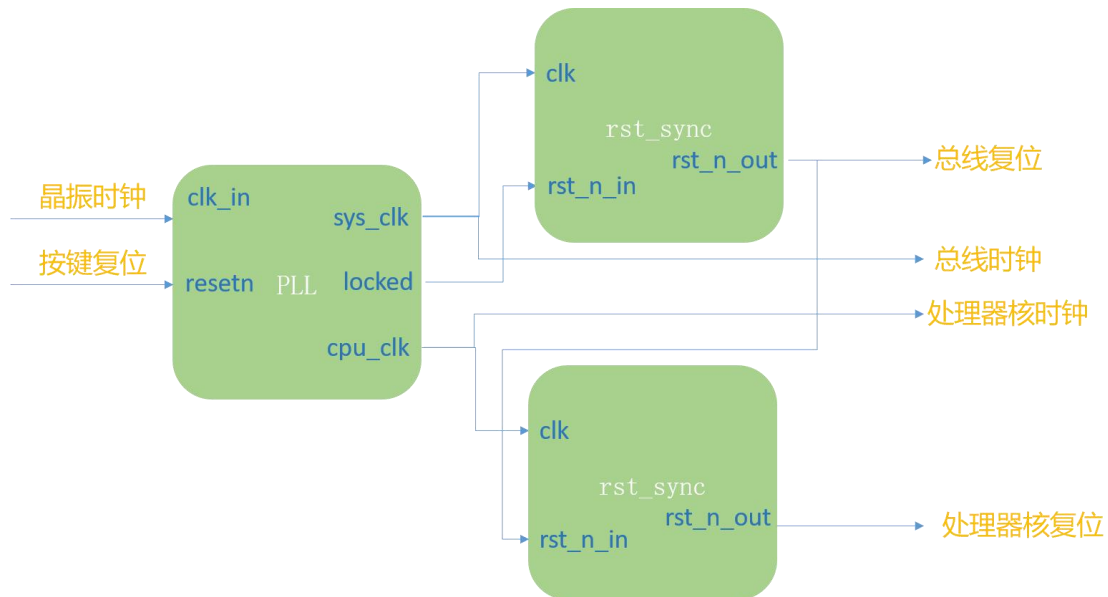


图 2-3 基础 SoC 时钟与复位网络

推荐连接顺序：

依次例化 core_top、Axi_CDC、AxiCrossbar_1x4、axi_wrap_ram、axi_uart_controller、confreg。

对于 AxiCrossbar_1x4，axiIn_0 连接处理器核（core_top）经过 Axi_CDC 处理后的 axi 信号，axiOut_0 连接 SRAM 控制器（axi_wrap_ram_sp_ext），axiOut_1 连接 axi 转 apb 桥后接 UART 模块（axi2apb_misc），axiOut_3 连接 confreg。

对于总线互联模块上未使用的接口，注意连接 dummy Slave，避免总线信号

出现 X 态。

Dummy Slave 信号如下：

```
assign axiOut_2_arready = 1'b1;
assign axiOut_2_rid     = 5'b0;
assign axiOut_2_rdata   = 32'b0;
assign axiOut_2_rresp   = 2'b0;
assign axiOut_2_rlast   = 1'b0;
assign axiOut_2_rvalid  = 1'b0;
assign axiOut_2_awready = 1'b1;
assign axiOut_2_wready  = 1'b1;
assign axiOut_2_bid     = 5'b0;
assign axiOut_2_bresp   = 2'b0;
assign axiOut_2_bvalid  = 1'b0;
```

2.4 SoC 验证流程

2.4.1 尝试 Hello_World

运行功能测试程序之前，可以先使用一个简单的 Hello_World 进行简单验证并熟悉验证环境。

首先进行 Hello_World 的功能仿真

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
```

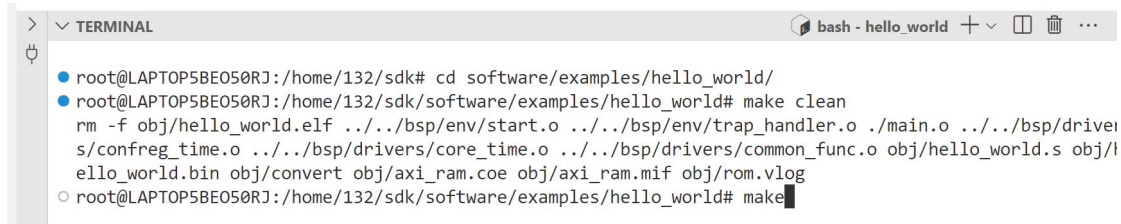


图 2-4 运行 make 进行编译

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 CICIEC_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_ciciec/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_ciciec/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。

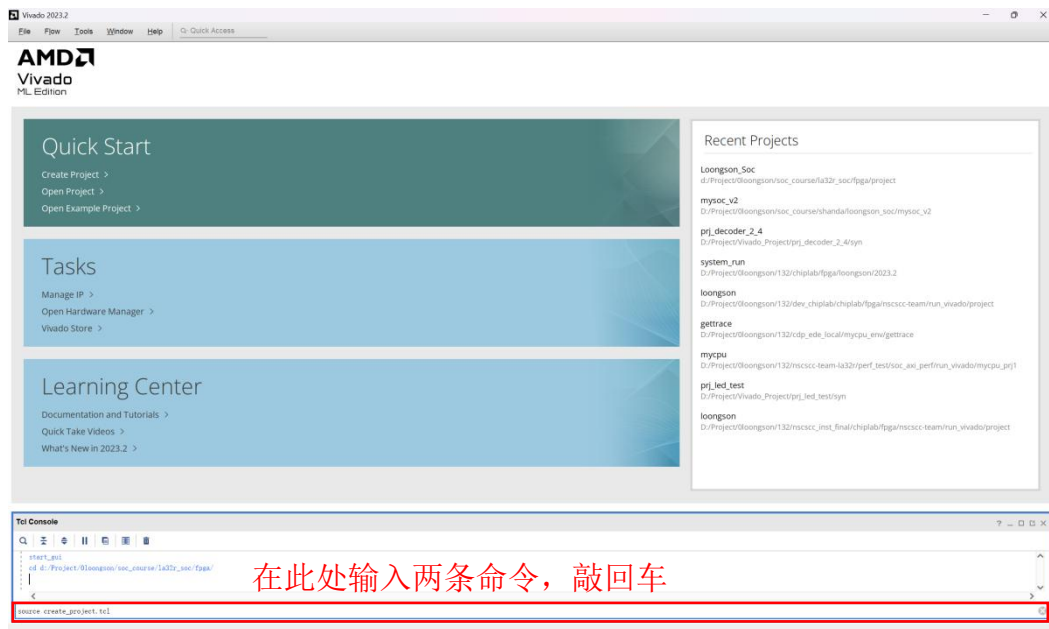


图 2-5 vivado 创建工程

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。

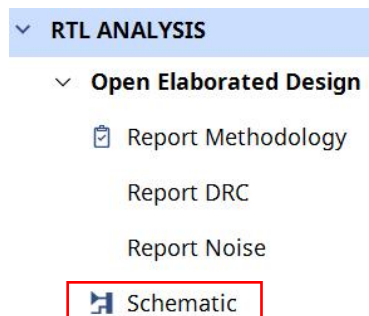


图 2-6 vivado 进行 RTL 分析

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!`

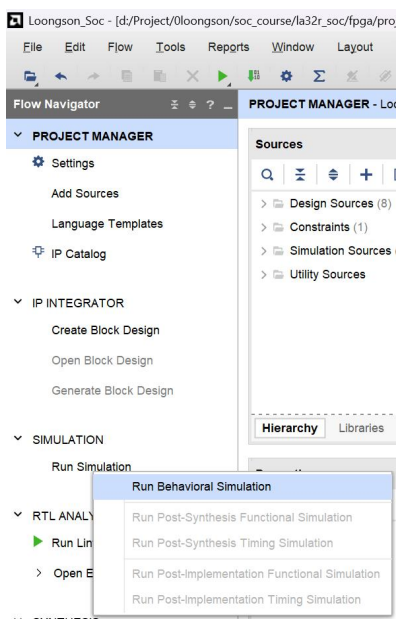


图 2-7 vivado 运行功能仿真

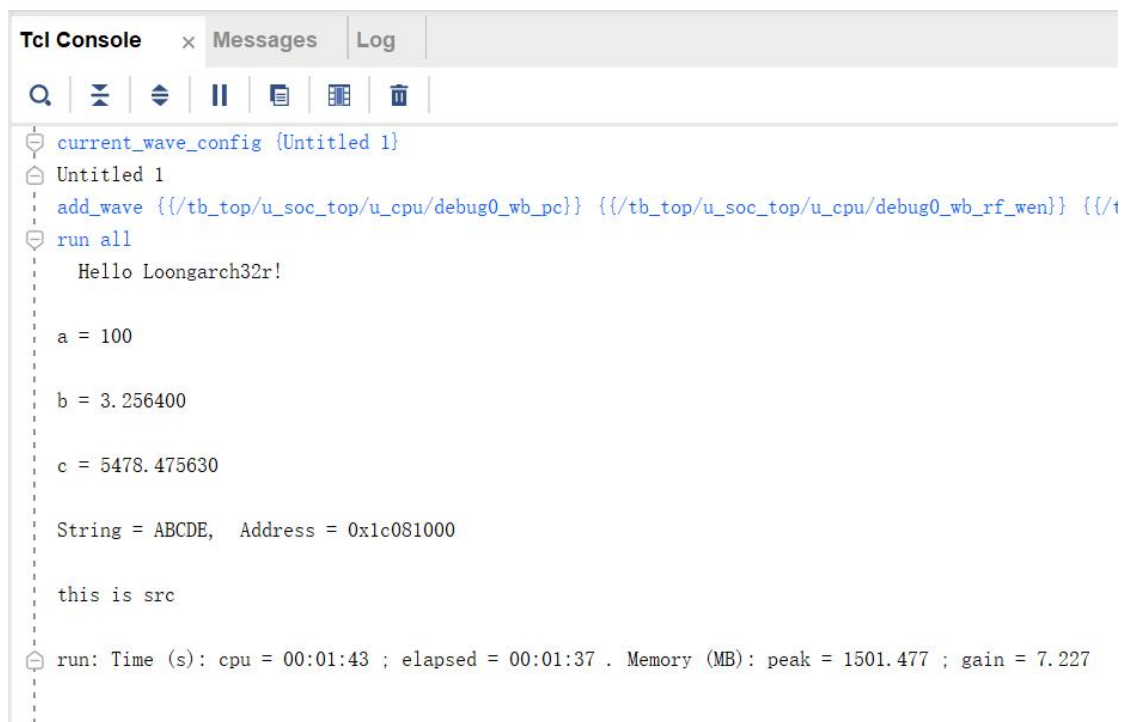


图 2-8 hello_world 功能仿真通过

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_worldad.s）查看出错的位置。

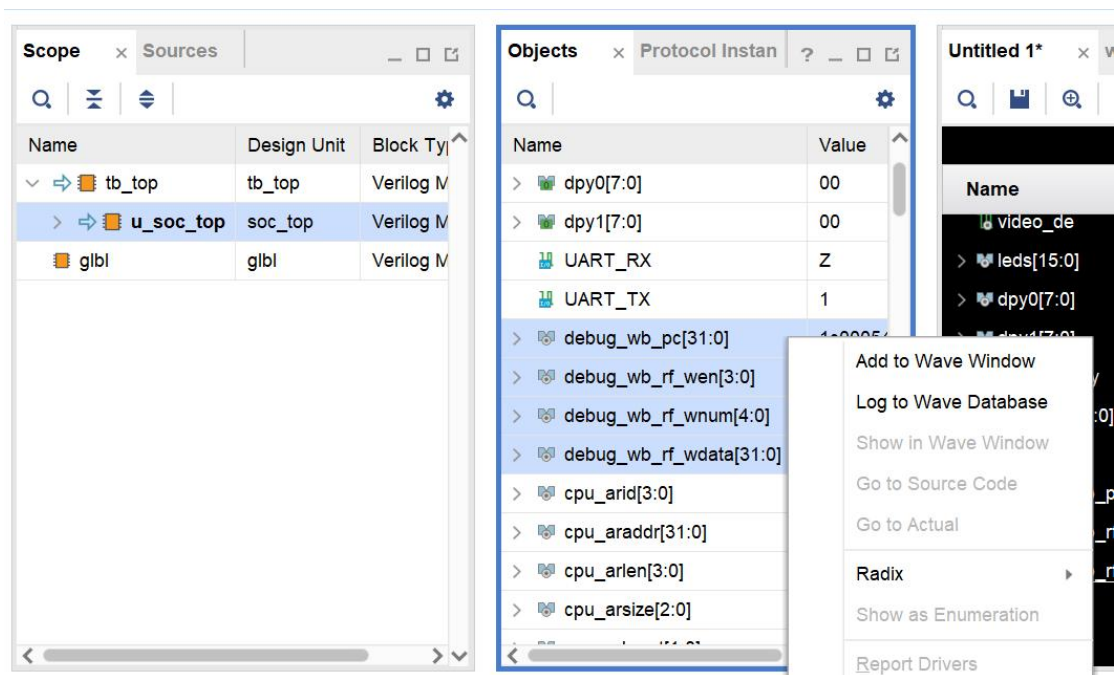


图 2-9 添加 debug 信号至波形窗口

```

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

int main(int argc, char** argv)
{
    PC 值 1c000580 指令编码 02bfc063 汇编指令 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)

```

图 2-10 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`<http://grs.nscsc.com:18000/signin>`，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，即可看到串口输出 Hello Loongarch32r!。



图 2-11 远程 FPGA 下载 SRAM

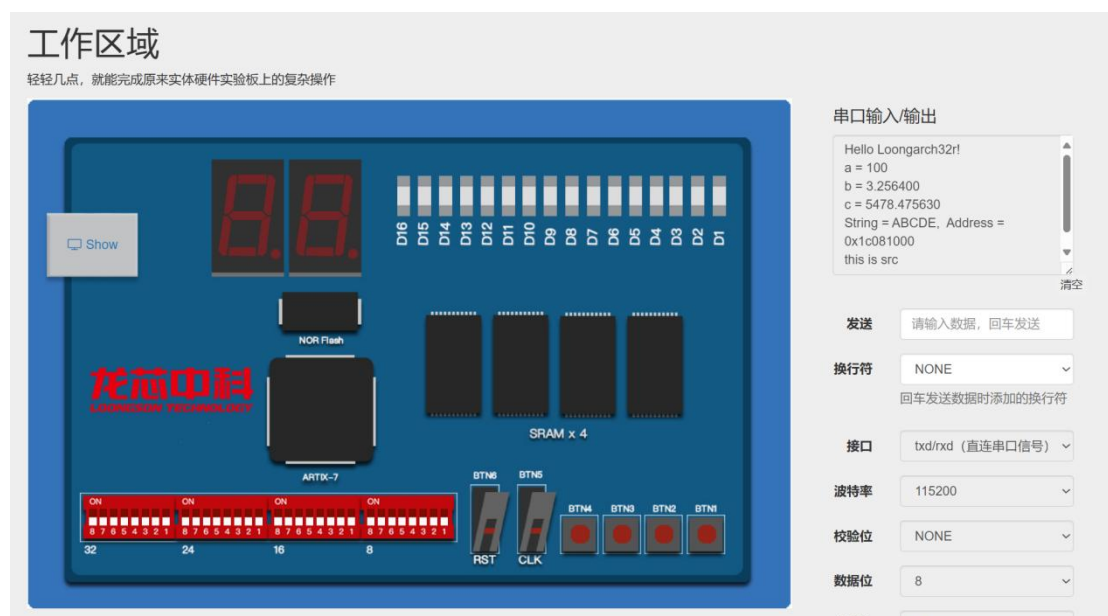
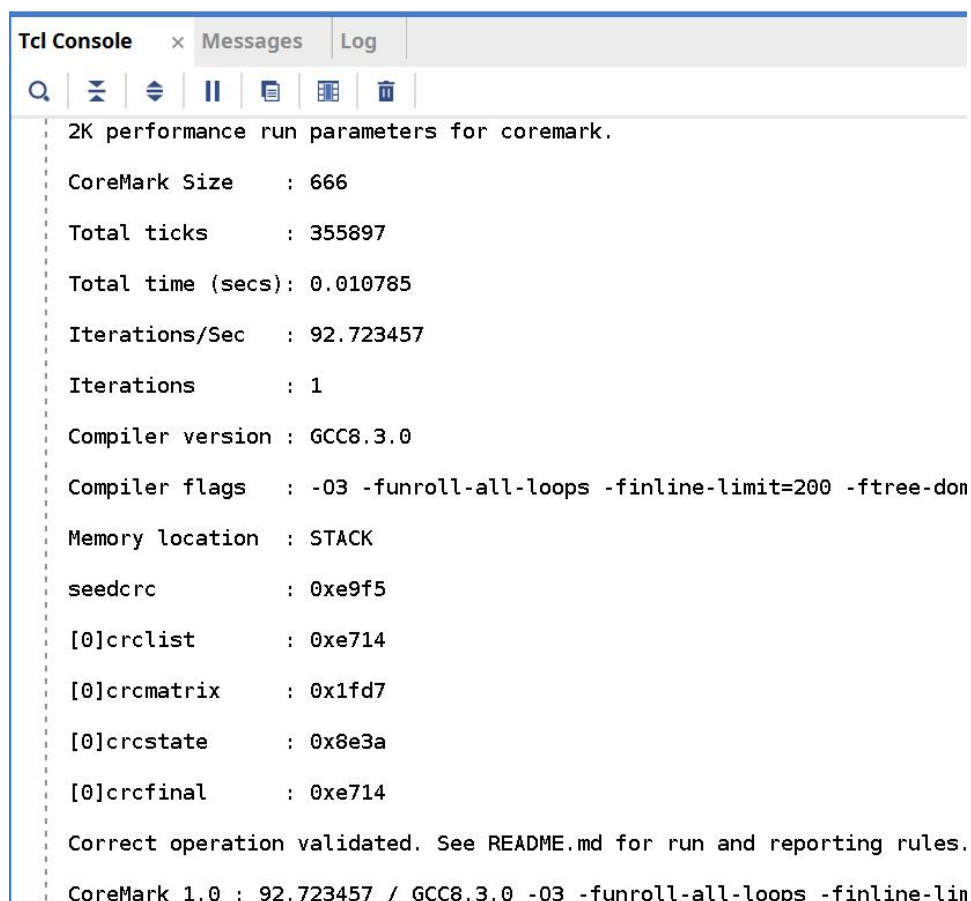


图 2-12 远程 FPGA 运行 Hello World 例程

2.4.2 功能测试程序

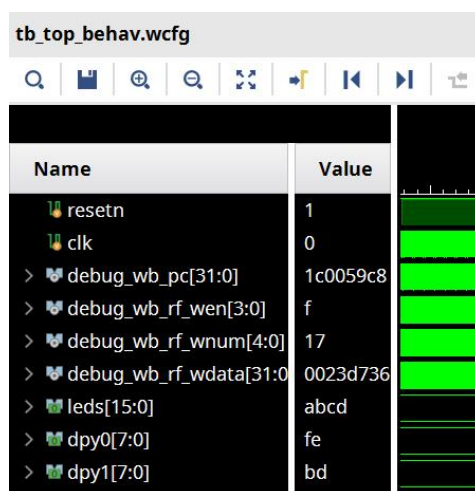
功能测试程序位于 `sdk/software/examples/ciciec_func` 目录，运行方法与 Hello_World 一致。功能测试程序将进行 CoreMark 测试和基础外设测试。CoreMark 测试结束后，会通过数码管将跑分展示出来，之后程序轮询拨码开关值，并将拨码开关的值输出在 LED 灯上，实现的效果就是拨码开关控制小灯亮灭。

CoreMark 程序在仿真时需要运行的时间比较长，下图是最终通过测试时的 Vivado 控制台输出的信息和跑完 CoreMark 后的 LED 输出与数码管输出。



```
Tcl Console x Messages Log
2K performance run parameters for coremark.
CoreMark Size : 666
Total ticks : 355897
Total time (secs): 0.010785
Iterations/Sec : 92.723457
Iterations : 1
Compiler version : GCC8.3.0
Compiler flags : -O3 -funroll-all-loops -finline-limit=200 -ftree-don
Memory location : STACK
seedcrc : 0xe9f5
[0]crc1st : 0xe714
[0]crcmatrix : 0x1fd7
[0]crcstate : 0x8e3a
[0]crcfinal : 0xe714
Correct operation validated. See README.md for run and reporting rules.
CoreMark 1.0 : 92.723457 / GCC8.3.0 -O3 -funroll-all-loops -finline-lin
```

图 2-13 仿真中功能测试程序的串口输出结果



Name	Value
resetn	1
clk	0
> debug_wb_pc[31:0]	1c0059c8
> debug_wb_rf_wen[3:0]	f
> debug_wb_rf_wnum[4:0]	17
> debug_wb_rf_wdata[31:0]	0023d736
> leds[15:0]	abcd
> dpy0[7:0]	fe
> dpy1[7:0]	bd

图 2-14 仿真中功能测试程序的 LED 和数码管输出

进行 FPGA 验证时，CoreMark 跑分信息将通过串口输出，并能在数码管上看到 2.8 的数字显示，拨动拨码开关可以看到小灯点亮。下图是进行 FPGA 验证时通过功能测试程序的效果图。

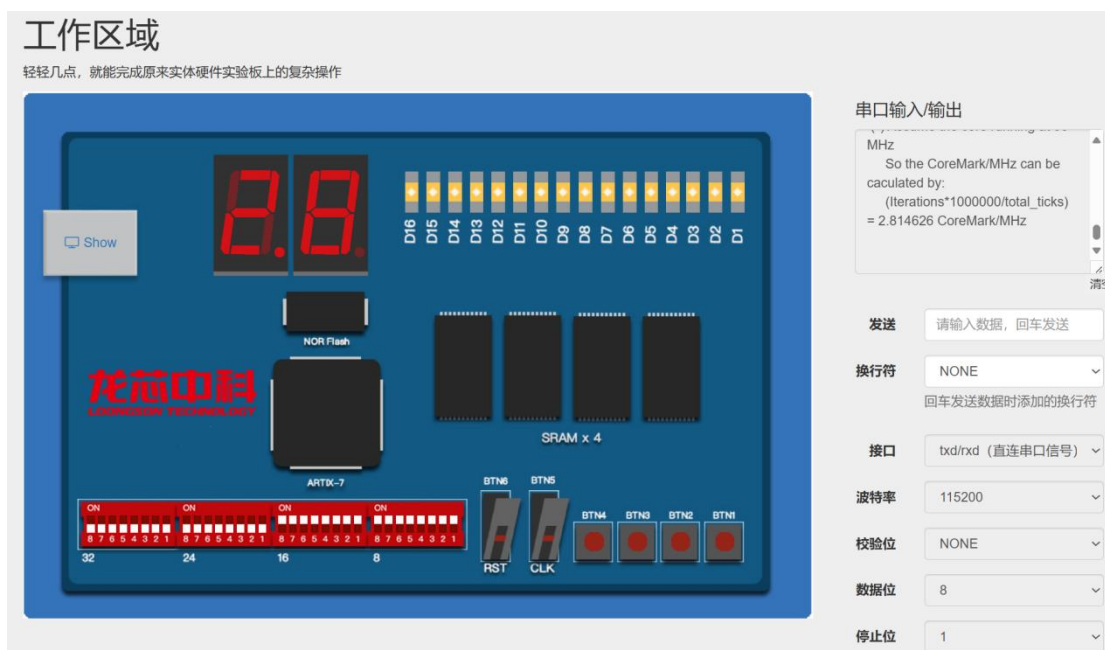


图 2-15 功能测试程序的 FPGA 验证结果

其中 FPGA 串口的完整输出内容如下图。

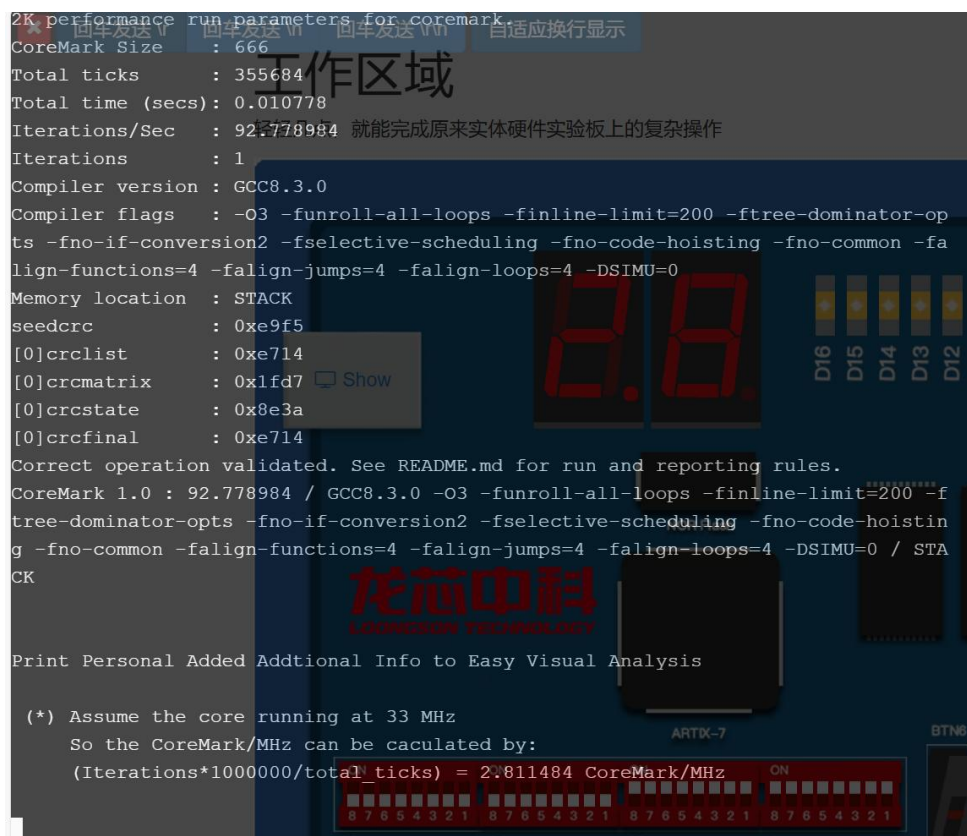


图 2-16 功能测试程序的 FPGA 串口输出内容

第 3 章 附录一 UART 模块说明

3.1 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。

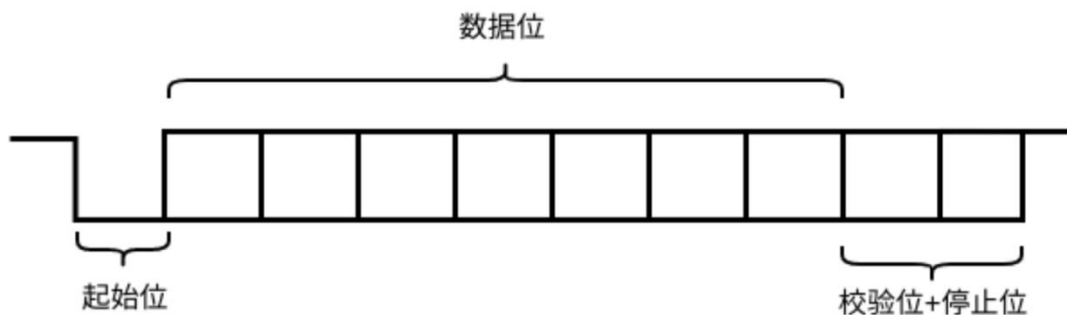


图 3-1 UART 数据传输格式

- 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。

3.2 UART 寄存器描述

表 3-1 UART 寄存器总览

名称	地址	描述
----	----	----

Receiver Buffer Register	0x00	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	0x00	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	0x01	UART 中断的使能/屏蔽寄存器
Interrupt Identification Register	0x02	获取中断信息的寄存器
FIFO Control Register	0x02	控制接收和发送缓存的寄存器
Line Control Register	0x03	控制帧传输格式的寄存器
Modem Control Register	0x04	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	0x05	获取传输数据状态的寄存器
Modem Status Register	0x06	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 Latch LSB	0x00	分频器的低 8 位
Divisor Latch MSB	0x01	分频器的高 8 位

表 3-2 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 3-3 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 3-4 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 3-5 Interrupt Identification Register 位定义

位	读写权限	描述
3-1	只读	中断 ID 011:接受状态中断 010: 接受数据有效中断 110: 字符超时中断 001: 发送数据空中断 000: 调制解调器状态中断

0	只读	1: 有中断待处理 0: 无中断
其他	N/A	保留

复位值: 0x1

表 3-6 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 3-7 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

		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 3-8 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	向 TxFIFO 写数据

		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	
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	读 LSR 寄存器

		符, '1' -接收缓存有字符	
--	--	-----------------	--

复位值: 0x0

表 3-9 Divisor Latch 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 3-10 Divisor Latch 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。

3.3 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:

$$\text{控制器时钟频率}/(16 * \text{波特率}) = \{\text{MSB}, \text{LSB}\}$$

3) 然后通过 APB 总线对数据控制寄存器 (LCR) 写控制字, 关闭对 Divisor 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。

3.4 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 3-11 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：

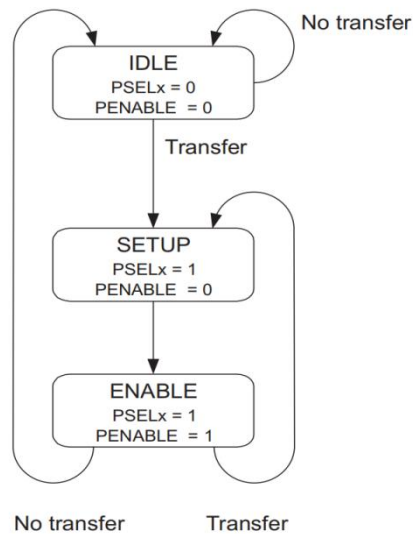


图 3-2 APB 状态机

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。

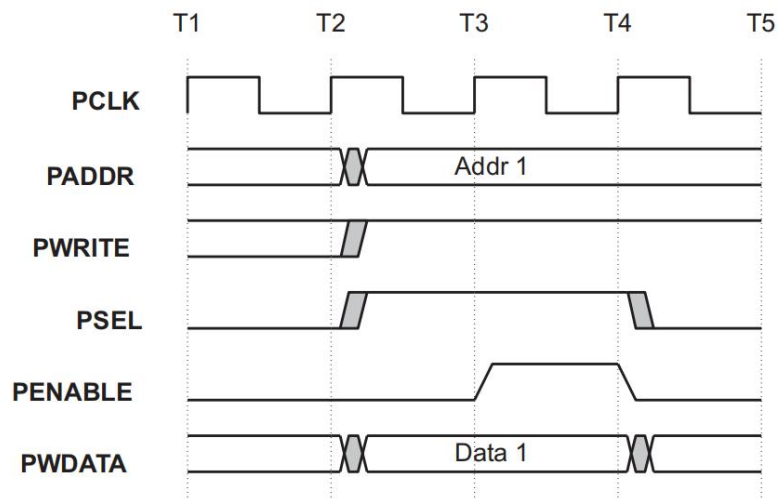


图 3-3 APB 写操作

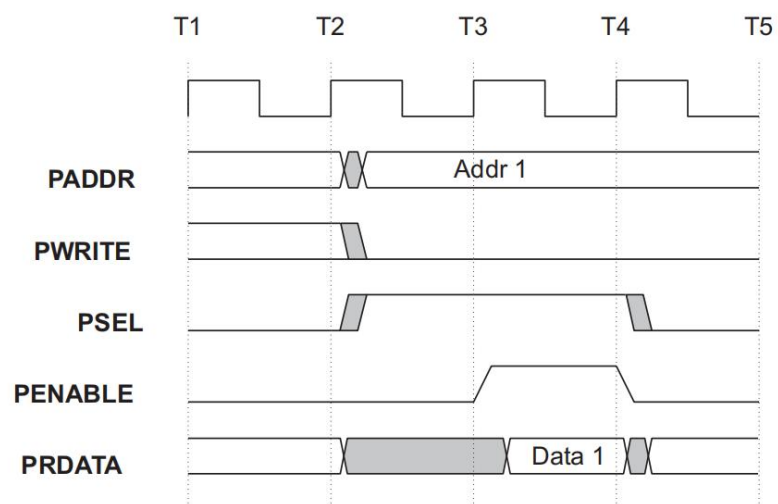


图 3-4 APB 读操作

第 4 章 附录二 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。