

# Cortex-M0 Based SoC Building

## Quick tutorial

**Version 5.0.0**

**2019 年 12 月**

# 目录

<b>第一章 简介 .....</b>	<b>1</b>
1.1. CORTEX-M0 与 AMBA3 AHBLITE .....	1
1.2. 文档使用说明.....	1
1.3. 相关软硬件介绍 .....	2
<b>第二章 背景知识简介 .....</b>	<b>5</b>
2.1. 处理器核端口介绍与配置.....	5
2.1.1. 处理器核关键信号说明.....	11
2.2. AMBA 3 AHB-LITE 总线协议 .....	14
2.2.1. 部分信号说明.....	14
2.2.2. Cortex-M0 支持的总线传输 .....	15
2.2.3. NONSEQ 传输时序.....	16
2.2.4. Memory Map 与总线扩展.....	19
2.3. XILINX BLOCK RAM 与 DISTRIBUTED RAM .....	22
<b>第三章 搭建 SOC.....</b>	<b>24</b>
3.1. 实验一：总线、RAM、汇编与调试.....	24
3.1.1. 硬件部分 .....	24
3.1.2. 汇编、启动与 Keil 工具 .....	28
3.1.3. Modelsim 仿真.....	36
3.1.4. Vivado 与调试.....	42

---

3.2. 实验二：数据存储器与流水灯 .....	50
3.2.1. 硬件部分 .....	51
3.2.2. 汇编 .....	59
3.2.3. Modelsim 仿真 .....	60
3.2.4. 调试 .....	60
3.2.5. 附加题一：蜂鸣器 .....	63
3.3. 实验三：中断与 C 语言编程 .....	64
3.3.1. 硬件部分 .....	66
3.3.2. 启动代码与 C 语言编程 .....	70
3.3.3. 调试与运行 .....	74
3.3.4. 附加题二：UART 控制流水灯 .....	76
3.4. 实验四：初探虚拟仪器及 DAC .....	77
3.5. 实验五：点亮 LCD 显示屏 .....	81
3.5.1. 显示屏简介 .....	81
3.5.2. 硬件部分 .....	82
3.5.3. 软件部分 .....	90
3.6. 本章小结 .....	95
<b>第四章 终极测试 .....</b>	<b>96</b>
<b>第五章 常见问题汇总 .....</b>	<b>97</b>
5.1. 软件包安装问题 .....	97
5.2. VERILOG 或汇编编译错误 (SYNTAX ERROR) .....	97

---

5.3. MODELSIM 仿真错误 .....	98
--------------------------	----

# 第一章 简介

本手册将介绍如何快速搭建基于 ARM Cortex M0 CPU 的 SoC，将会参考 AMBA AHB 总线相关知识、cortex m0 用户手册、cortex m0 技术参考手册以及 ARMv6-M 架构参考手册，以及将会使用到 Keil、Modelsim、串口调试助手以及 Vivado 等工程软件。

本手册的配套代码可以在 [https://github.com/liufengrui/CortexM0\\_SoC\\_Task](https://github.com/liufengrui/CortexM0_SoC_Task) 获取。

## 1.1. Cortex-M0 与 AMBA3 AHBLite

感谢 ARM 在其 DesignStart 项目中开放 Cortex M0 CPU 能让我们有机会学习研究。CM0 CPU 总体结构如图 1-1 所示。

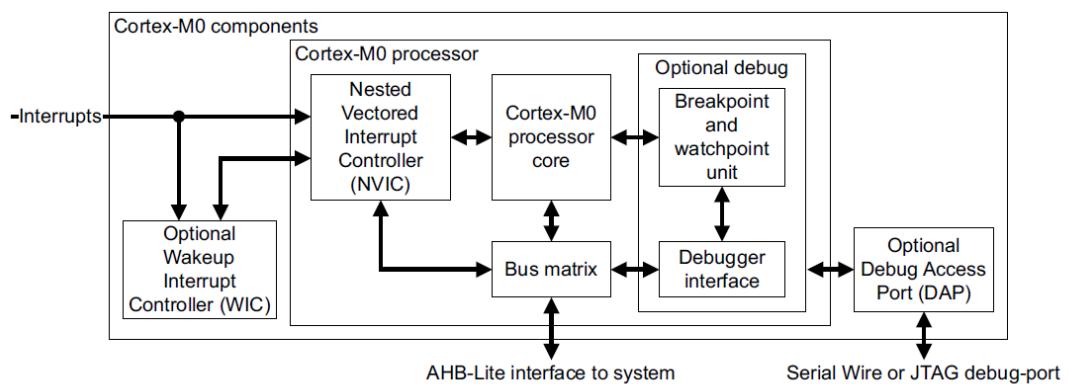


图 1-1 Cortex-M0 架构图

CPU 提供了中断向量端口、AHB-Lite 端口以及 DAP 端口。  
关于 AMBA3 AHBLite 需要读者根据相关文档自行学习，由于 Cortex-M0 的特性，将不会需要 AHBLite 所有功能，在第二章将会有详细的说明。

## 1.2. 文档使用说明

"/CortexM0\_SoC/docs/" 提供了本手册所需要的所有参考文档，还需读者反复仔细阅读。

"/CortexM0\_SoC/Task\*/rtl" 提供了本手册每次实验对应代码。

### 1.3. 相关软硬件介绍

本手册将会用到三个软件：

- Vivado 2018.3
- Modelsim
- Keil

本手册将会用到以下硬件平台：

- FPGA (Xilinx Artix-7)，如图 1-1 所示

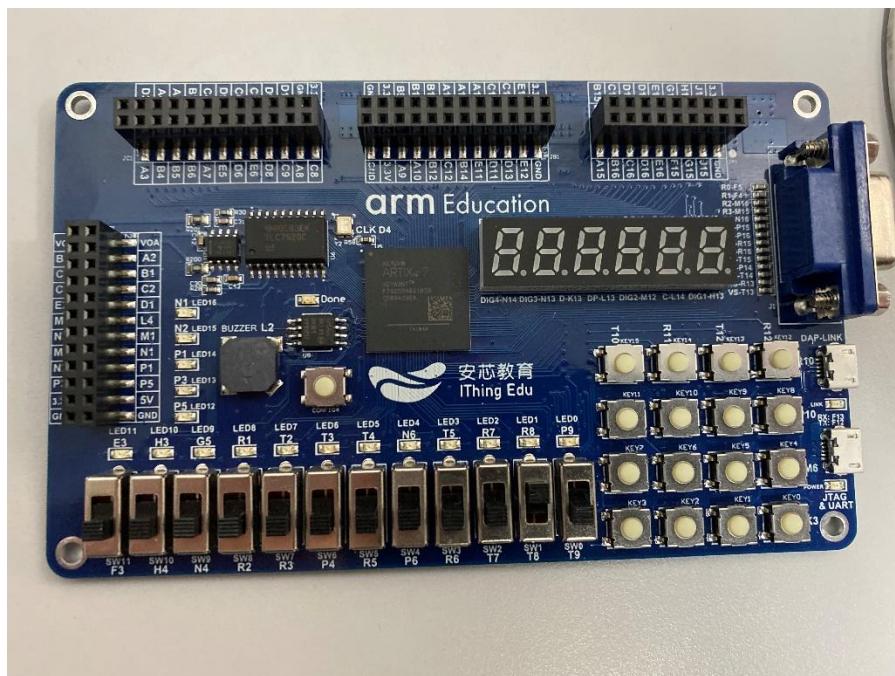


图 1-1 FPGA 开发板

那么，FPGA、SoC 以及这些软硬件工具的区别与联系在哪呢？请看下图 1-2。

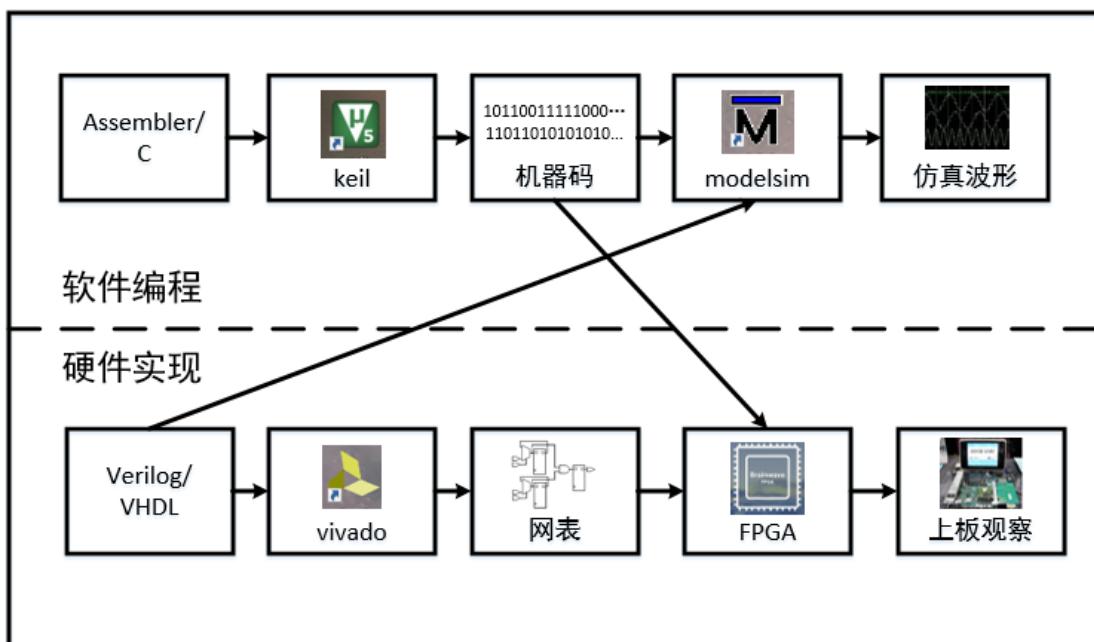


图 1-2 软硬件关系

可以看到，vivado 负责将硬件描述语言所描述的 SoC (Verilog/VHDL) 编译、综合、实现，将 FPGA 内部本身无序的各种逻辑资源（例如：查找表、触发器、RAM 等）配置成为有序的电路，实现 SoC 功能。而 keil 负责将编写的软件编程语言 (C/Assembler) 编译成为机器码十六进制文件。在 modelsim 中将机器码作为 Verilog 描述的 RAM 的初始化内容，即可进行仿真，看到 SoC 工作时各个信号的波形。若将机器码通过工具下载进由 FPGA 实现的 SoC 中，那么就可以让 SoC 执行编写的程序，通过开发板看到执行结果。

软硬件开发的层次结构如图 1-3 所示。

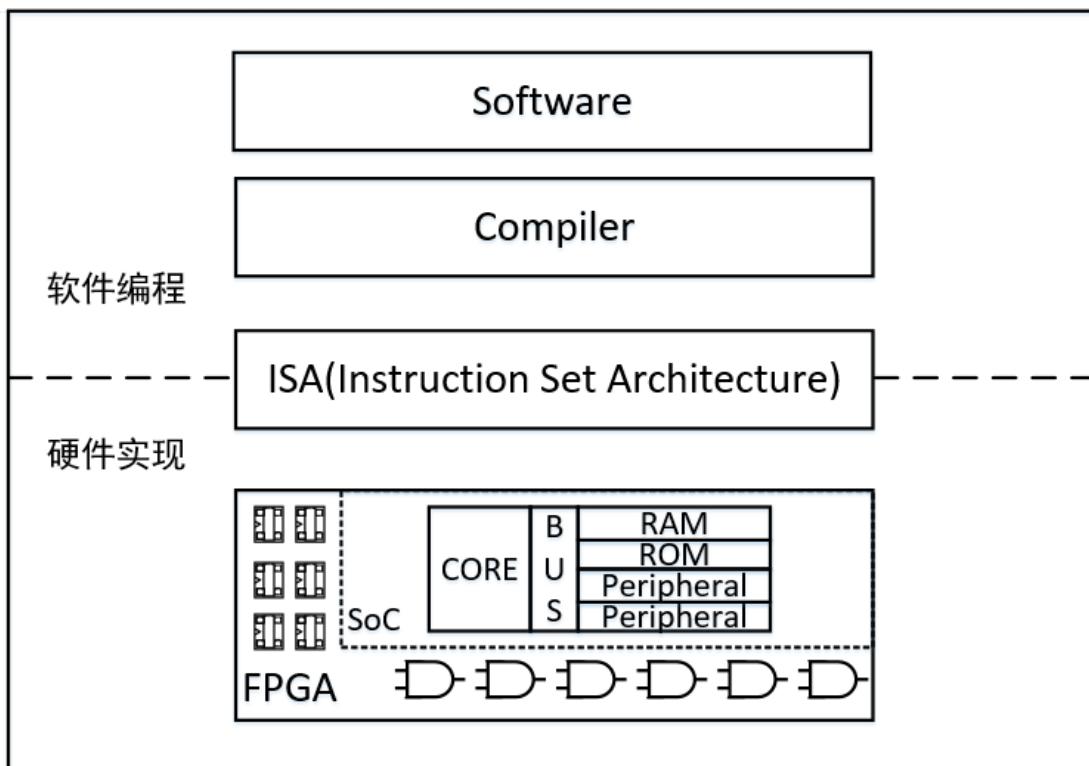


图 1-3 软硬件开发层次

从下往上看（从硬件到软件），FPGA 被配置成为 SoC，SoC 的工作依赖于一条一条的指令，而指令则是由对应的汇编代码生成，最后汇编代码又由编译器将高层次编程语言编译而来。

从上往下看（从软件到硬件），如果想要运行编写好的软件，首先需要利用编译器将代码编译为汇编代码，然后将汇编代码与指令集对应生成机器码，接着将机器码存入 SoC 的存储器中，这时候 SoC 就能根据指令开始执行，最后 SoC 则需要利用 FPGA 构建。

## 第二章 背景知识简介

本章将会简要介绍搭建 SoC 需要的两方面背景知识：

- a) Cortex-M0 相关介绍与端口配置（参考 “/docs/Cortex\_M0\_SPEC/” 文件夹相关资料），必须先清楚处理器核的各个特征，才能正确将其运行起来。
- b) AMBA3 AHB lite 总线协议（参考 “/docs/AMBA3\_AHBLite\_SPEC /” 文件夹相关资料），处理器核依赖总线与外设进行通信，所以总线系统基本上作为 SoC 硬件设计的核心模块，在本手册后面涉及到的硬件实现内容都是围绕总线设计开展的，因此在此之前必须先要清楚总线协议以及相关的时序内容。

### 2.1. 处理器核端口介绍与配置

在 ARM DesignStart 网址下载的 Cortex-M0 DesignStart Eval 文件资源中找到名为 “cortexm0ds\_logic.v” 的文件，这便是处理器核的网表形式的 Verilog 代码。在实验开始前，我们需要对处理器核的时钟、复位、无用端口以及 DAP 的 iobuf 进行配置。

```
//-----
// Instantiate Cortex-M0 processor logic level
//-----
cortexm0ds_logic u_logic (
    // System inputs
    .FCLK          (clk),           //FREE running clock
```

	.SCLK	(clk),	//system clock
	.HCLK	(clk),	//AHB clock
	.DCLK	(clk),	//Debug clock
	.PORESETn	(RSTn),	//Power on reset
	.HRESETn	(cpuresetn),	//AHB and System reset
	.DBGRESETn	(RSTn),	//Debug Reset
	.RSTBYPASS	(1'b0),	//Reset bypass
	.SE	(1'b0),	// dummy scan enable port for synthesis
			// Power management inputs
	.SLEEPHOLDREQn	(1'b1),	// Sleep extension
PMU	.WICENREQ	(1'b0),	// WIC enable request from PMU
	.CDBGPWRUPACK	(CDBGPWRUPACK),	// Debug Power Up ACK from PMU
			// Power management outputs
	.CDBGPWRUPREQ	(CDBGPWRUPREQ),	
	.SYSRESETREQ	(SYSRESETREQ),	
			// System bus

.HADDR	(HADDR[31:0]),
.HTRANS	(HTRANS[1:0]),
.HSIZE	(HSIZE[2:0]),
.HBURST	(HBURST[2:0]),
.HPROT	(HPROT[3:0]),
.HMASTER	(HMASTER),
.HMASTLOCK	(HMASTLOCK),
.HWRITE	(HWRITE),
.HWDATA	(HWDATA[31:0]),
.HRDATA	(HRDATA[31:0]),
.HREADY	(HREADYOUT),
.HRESP	(HRESP),

// Interrupts

.IRQ	(32 'b0),	//Interrupt
.NMI	(1'b0),	//Watch dog interrupt
.IRQLATENCY	(8'h0),	
.ECOREVNUM	(28'h0),	

// Systick

.STCLKEN	(1'b0),
.STCALIB	(26'h0),

```
// Debug - JTAG or Serial wire

// Inputs

.Ntrst          (1'b1),
.SWDITMS        (SWDI),
.SWCLKTCK       (SWCLK),
.TDI             (1'b0),

// Outputs

.SWDO            (SWDO),
.SWDOEN          (SWDOEN),
.DBGRESTART     (1'b0),

// Event communication

.RXEV            (1'b0),           // Generate event
.EDBGRQ          (1'b0)           // multi-core halt request
);
```

由于 ARM DesignStart Eval 中提供的处理器核代码为不可读的网表结构（即直接基于基本门电路的描述，想要读懂或了解此类电路逻辑几乎不可能），相关说明文档也极少对处理器核端口有详细的描述，因此这里对端口的配置基本上依赖于 DesignStart 资料包中提供的参考设计，根据参考资料中的代码尽可能的理解各个端口的意义。

- A. 首先是 System input 部分端口，此部分端口为处理器核所需要的各种时钟与复位，时钟与复位作为触发器的基本输入，设置错误将会直接导致整个系统的崩溃

与失效，因此这部分端口至关重要。

B. 接下来是 Power management input/output 部分端口，此部分与处理器核功耗管理单元有关，控制处理器核出入睡眠状态（由 WFI 与 WFE 指令控制处理器核流水线停止，停止读取与处理指令，整个电路进入功耗极低的状态，最后通过中断与事件唤醒，在本手册内容中不涉及）、调试启动（要将输出端口 CDBGWRUPREQ 信号经过同步后接在输入端口 CDBGWRUPACK 处）与软复位（CPU 的 SYSRESETREQ 输出信号需要经过同步后接入 HRESETn，这是因为在处理器核进入调试模式后，调试器能够向处理器核发送特定代码实现软复位，当其内部复位寄存器被置位后，则通过 SYSRESETREQ 信号发出复位请求）。

```
//-----  
  
// RESET AND DEBUG  
  
//-----  
  
wire SYSRESETREQ;  
  
reg cpuresetn;  
  
  
  
always @(posedge clk or negedge RSTn)begin  
  
    if (~RSTn) cpuresetn <= 1'b0;  
  
    else if (SYSRESETREQ) cpuresetn <= 1'b0;  
  
    else cpuresetn <= 1'b1;  
  
end
```

```
wire CDBGPWRUPREQ;  
  
reg CDBGPWRUPACK;  
  
always @(posedge clk or negedge RSTn)begin  
  
    if (~RSTn) CDBGPWRUPACK <= 1'b0;  
  
    else CDBGPWRUPACK <= CDBGPWRUPREQ;  
  
end
```

C. 然后是 System bus 部分，总线作为 SoC 的大动脉，负责几乎所有数据的传输，

在手册接下来的内容将会利用此总线接口构建整个 SoC。

D. 在 interrupt 相关端口可以看到，处理器核在这里提供了两种中断，IRQ 与 NMI

中断，处理器核在处理这两种中断的优先级不同，具体将会在后面实验三中具体

介绍 M0 的中断处理机制以及使用 IRQ 中断。

E. Systick 是一个独立计数器，由于处理器核在工作时，指令与时间的相关性并不

大，因此在处理一些依赖于时间的工作时利用指令实现计数往往是不可靠的，因

此 Systick 作为独立计数模块，能够在完成预设计数后向处理器核提供中断，通

过调用中断处理函数来及时处理相关指令。

F. 接下来是 Debug 端口，由于本手册使用 CMSIS-DAP 作为调试器，需要用到

SW (Serial Wire) 调试工具（不用 JTAG 调试端口），其中，调试器上 SWD 端

口中的数据通道 (SWDIO) 为 inout 类型，与通常使用到 的单向信号类型

(INPUT、OUTPUT) 不同，inout 类型的引脚既能作为输入也能作为输出，能

够有效地节约管脚数量，降低芯片面积。但是在处理器核中需要对此复杂的信号

类型进行扩展，将其扩展为多个单向信号类型，对应处理器核端口中的 SWDI、SWDO、SWDOEN 三个信号，处理器核端的输入信号 SWDI 由三态信号线直接连接，而处理器核端的输出信号线 SWDO 则与 SWDOEN 控制信号一起组成三态输出门输出到三态信号线 SWDIO 上。

```
//-----  
  
// DEBUG IOBUF  
  
//-----  
  
wire SWDO;  
  
wire SWDOEN;  
  
wire SWDI;  
  
  
  
assign SWDI = SWDIO;  
  
assign SWDIO = (SWDOEN) ? SWDO : 1'bz;
```

### 2.1.1. 处理器核关键信号说明

为了后续实验能够顺利进行，这里将会对本书所需要的处理器核关键信号做一个简要的说明。

信号名	描述
HXXXX(以 H 开头的信号)	总线相关，下一节将会详细介绍

vis_rX_o	通用寄存器，通过仿真观察这些寄存器能够知道相关汇编指令是否正常执行 (ADD、MOV 等)
vis_msp_o	栈指针
vis_r14_o	连接寄存器，程序返回出错时需要检查此寄存器
vis_pc_o	程序计数器，可以用来判断处理器是否在正常运行
vis_ipsr_o	中断状态寄存器，当发生错误的时候，PSR 的值会改变，用于判断发生何种类型错误
LOCKUP	处理器处理 NMI 或者 HardFault 时又发生错误后，处理器将会被死锁

处理器核寄存器如下图 2-1 所示：

The processor core registers are:

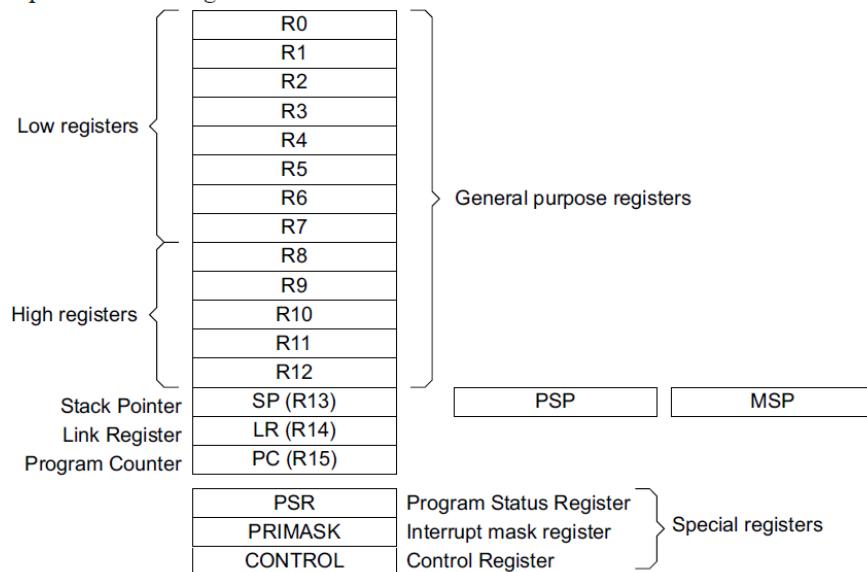


图 2-1 处理器核寄存器

IPSR 寄存器如下图 2-2 所示，当仿真时出现了错误后可以观察此寄存器判断发生了何种错误或者异常，以便对症下药。

Bits	Name	Function
[31:6]	-	Reserved
[5:0]	Exception number	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4-10 = Reserved 11 = SVCall 12, 13 = Reserved 14 = PendSV 15 = SysTick, if implemented <sup>a</sup> 16 = IRQ0
		.
		.
		n+15 = IRQ(n-1) <sup>b</sup>
		(n+16) to 63 = Reserved.
		see <i>Exception types</i> on page 2-19 for more information.

图 2-2 IPSR 寄存器描述

在书《Cortex™-M0 Devices Generic User Guide》中 2.4.1 章详细描述了 LOCKUP 出现的条件以及退出 LOCKUP 的方法，在本实验中，若仿真出现 LOCKUP 通常是由连环的 HardFault 导致的，而 HardFault 出现的原因如下图 2-3 所示。

Faults are a subset of exceptions, see *Exception model* on page 2-19. All faults result in the HardFault exception being taken or cause lockup if they occur in the NMI or HardFault handler. The faults are:

- execution of an SVC instruction at a priority equal or higher than SVCall
- execution of a BKPT instruction without a debugger attached
- a system-generated bus error on a load or store
- execution of an instruction from an XN memory address
- execution of an instruction from a location for which the system generates a bus fault
- a system-generated bus error on a vector fetch
- execution of an Undefined instruction
- execution of an instruction when not in Thumb-State as a result of the T-bit being previously cleared to 0
- an attempted load or store to an unaligned address.

图 2-3 导致 HardFault 的原因

在完成本手册后续讲解的实验时，如果出现了 HardFault 通常需要检查：

- 总线是否发生时序错误，检查总线接口时序以及 Verilog 代码
- 处理器是否读取到错误的指令，检查 Keil 编译情况，相关地址是否正确设置，再逐步分析错误原因

## 2.2. AMBA 3 AHB-lite 总线协议

在《AMBA® 3 AHB-Lite Protocol》文档中详细介绍了总线的信号描述、传输类型与时序，由于 Cortex-M0 处理器核的特性，本文档将会在此文档的基础上精简设计，简化相关模块的设计。

### 2.2.1. 部分信号说明

名称	来源	描述
HADDR[31:0]	Master	传输地址
HBURST[2:0]	Master	Burst 类型

HSIZE[2:0]	Master	数据宽度 00: 8bit Byte 01: 16bit Halfword 10: 32bit Word
HTRANS[1:0]	Master	传输类型 00: IDLE, 无操作 01: BUSY 10: NONSEQ, 主要的传输方式 11: SEQ
HWDATA[31:0]	Master	核发出的写数据
HWRITE	Master	读写选择 (1: 写, 0: 读)
HRDATA[31:0]	Slave	外设返回的读数据
HREADOUT	Slave	何时传输完成 (通常为 1)
HRESP	Slave	传输是否成功 (通常为 0)

### 2. 2. 2. Cortex-M0 支持的总线传输

虽然处理器核的代码不具可读性，但也能从中找到一些蛛丝马迹，有助于简化设计。

```

assign HTRANS[0] = 1'b0;
assign HBURST[2] = 1'b0;
assign HBURST[1] = 1'b0;
assign HBURST[0] = 1'b0;

```

可以看到处理器核端的 HTRANS 最低为恒为 0, 因此说明 M0 仅支持 NONSEQ 传输, 并且 HBURST 也恒为 0, 因此 M0 不支持 BURST 传输。在后面的外设总线接口设计中, 只需要满足 NONSEQ 类型传输即可, 为简化设计提供的条件。

### 2. 2. 3. NONSEQ 传输时序

NONSEQ 作为传输类型中最简单的一种, 只需要查阅参数文档中 Basic transfers 一节。

基本读操作: 如图 2-4.

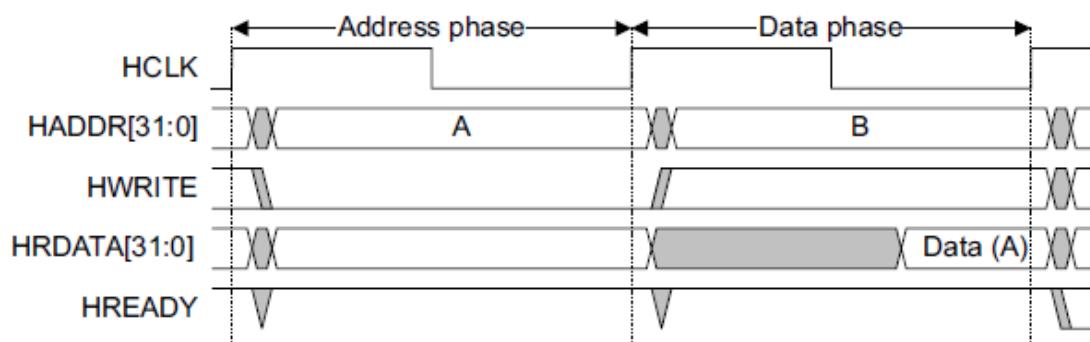


图 2-4 基本读操作

当 Master 需要从外设读取数据时, 总共需要经历两个阶段: Address phase & Data phase, 因此一次读传输至少需要 2cycle。在 Address phase 时, Master 会把读取地址输出在地址总线上, 直到 HREADY 为 '1', 在图 2-1 中, 由于 HREADY 一直为 '1', 那么 Master 在 Address phase 放出地址后直接进入 Data phase; 在 Data phase 时, Master 会在 HREADY 为 '1' 时读取数据总线 HRDATA 上的数据, 至此传输完成。

基本写操作：如图 2-5 所示。

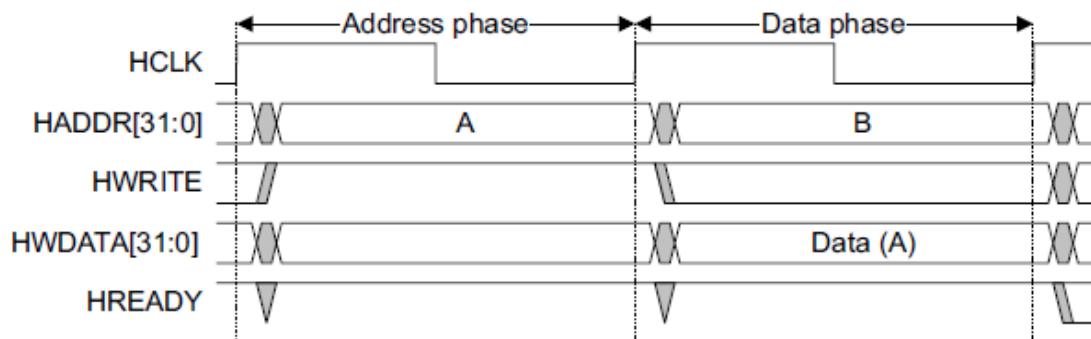


图 2-5 基本写操作

类似基本读操作，写操作也会经历两个阶段：在 Address phase 时，Master 会把写地址输出在地址总线上，直到 HREADY 为 ‘1’，在图 2-2 中，由于 HREADY 一直为 ‘1’，那么 Master 在 Address phase 放出地址后直接进入 Data phase；在 Data phase 时，Master 会将写数据放在数据总线 HWDATA 上，直到 HREADY 为 ‘1’，传输完成。

具有等待的读写操作：如图 2-6、图 2-7 所示。

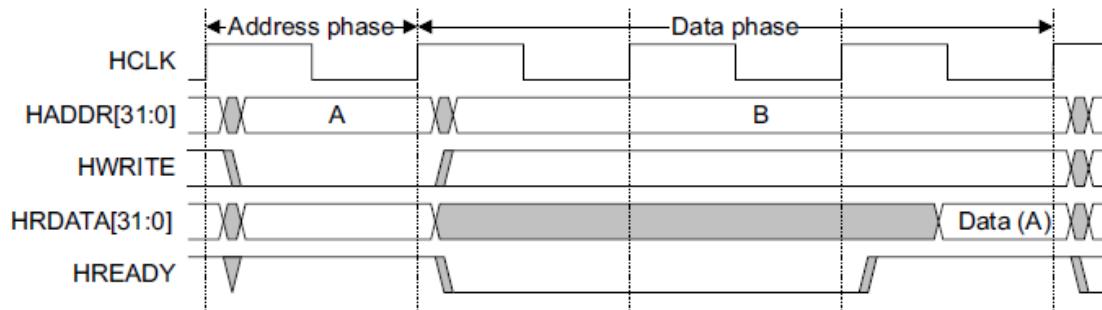


图 2-6 具有等待的读操作

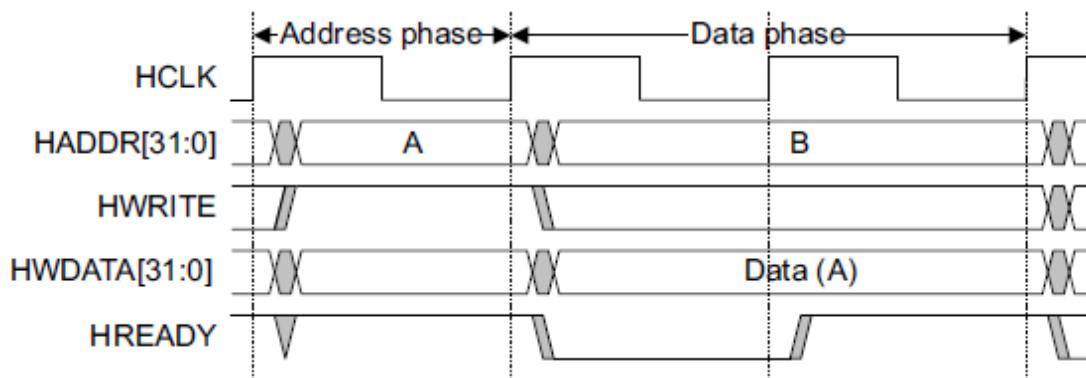


图 2-7 具有等待的写操作

正如前面所说, HREADY 为当前正在进行传输的 Slave 返回的 HREADYOUT, Master 端会把 HREADY 既作为进入传输的判断条件(在 HREADY 为 '0' 时不会开始下一个传输), 也会作为传输完成的条件 (在 HREADY 为 '0' 时不会退出当前传输)。但是, 总线是流水线结构, 虽然对于一次传输至少需要两个 cycle, 但是对于两次传输, 例如把图 2-8 中的 A 与 B 看作两次传输, 图中的 Address phase 为写传输 A 的地址阶段, 图中的 Data phase 为 A 的数据阶段, 但也同时作为读传输 B 的地址阶段。B 地址对应的外设在 Data phase 的第一个周期时, 由于 HREADY 为 0, 并不能进入传输, 而在第二个周期时, 对于 B 而言与图中 A 的 Address phase 无异, 最终实现如图 2-5 所示的流水线传输。

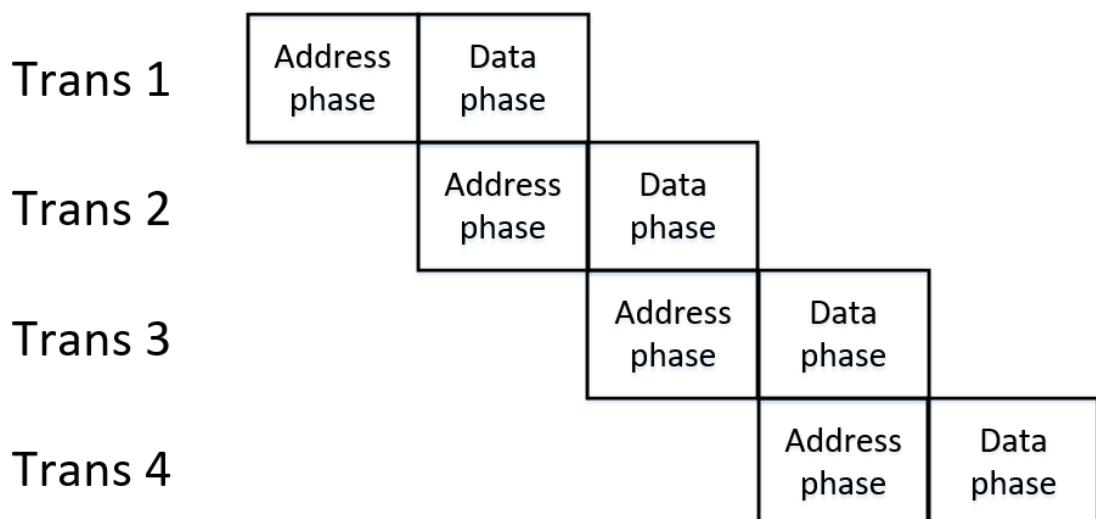


图 2-8 总线流水

对与 Slave 而言, HREADY 只需要作为进入传输的判断条件, 因为进入传输后, HREADY 就会被切换到自己的输出 HREADYOUT 上, 因此当 Slave 根据 HREADY 等信号进入传输状态后, 自行控制传输结束的时间, 并依此控制 HREADYOUT 输出。

在后面 SoC 具体设计中, 所有的 Slave 接口都能在两个周期内完成读写 (即判断出 Address phase 后能在下一个周期完成数据读写), 因此所有外设的 HREADYOUT 信号都被置为常量 '1', 这也是简化设计的一个体现。

## 2.2.4. Memory Map 与总线扩展

本手册所搭建的 SoC 的 memory map 如图 2-9 所示。

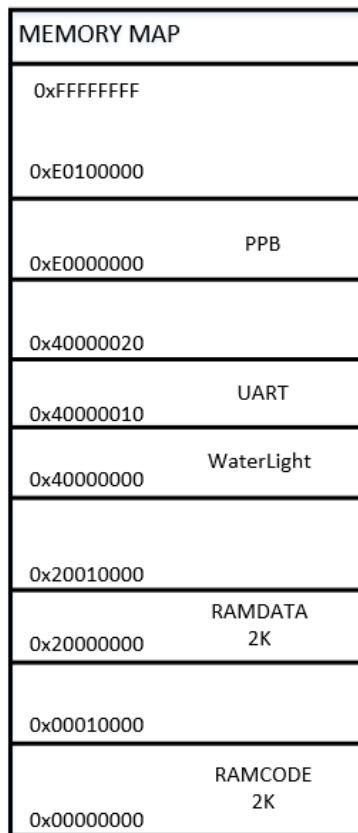


图 2-9 memory map

处理器核通过地址编码访问外设，所有的外设在处理器核看来都是 memory map 上的一块连续区域，访问这块区域就是访问对应的外设。例如图 2-6 中的 RAMCODE，其地址编码为 0x00000000-0x0000ffff，那么处理器核通过 AHB 总线发出的任何一次总线操作，只要地址总线上的值在 0x00000000-0x0000ffff 之间，都认为是处理器核在向 RAMCODE 提出总线操作。

那么在 SoC 具有多个外设，但是处理器核只有一个 Master 总线接口的情况下，就需要用到总线扩展模块，使处理器核能够访问多个外设，在《AMBA® 3 AHB-Lite Protocol》文档中提供了 Single Master AHB Interconnect 结构，如图 2-10。

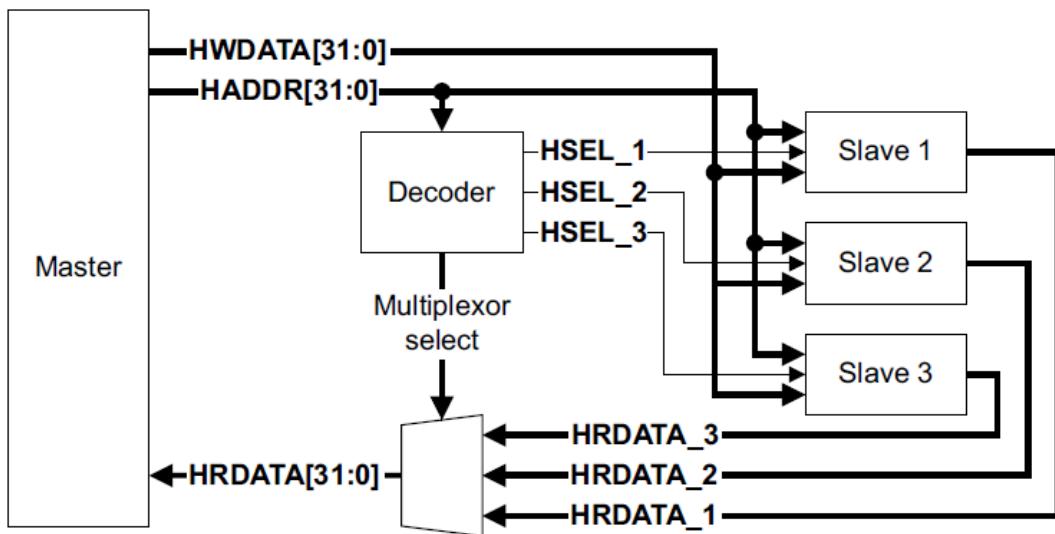


图 2-10 总线扩展

总线扩展模块主要由两部分组成：Decoder 与 Slave MUX。

Decoder 的作用是对地址总线进行译码，生成对应的外设的选择信号，同样以 RAMCODE 外设为例，由于其地址编码为 0x00000000-0x0000ffff，那么只要地址总线 HADDR 的高 16bit 为 0x0000 时，地址总线的值必定处于 RAMCODE 的地址编码区域中，则判定为处理器核对 RAMCODE 提起的一次总线操作，因此 RAMCODE 对应的选择信号 HSEL 将会被置位有效；若 HADDR 的高 16bit 不为 0x0000 时（例如 0x4000），地址总线的值不处于 RAMCODE 的地址编码区域，则判定为不是对 RAMCODE 的总线操作，因此对应的选择信号被置位无效。如图 2-11 所示，每一个外设在 Decoder 中都需要一个比较器用于产生相应的选择信号。

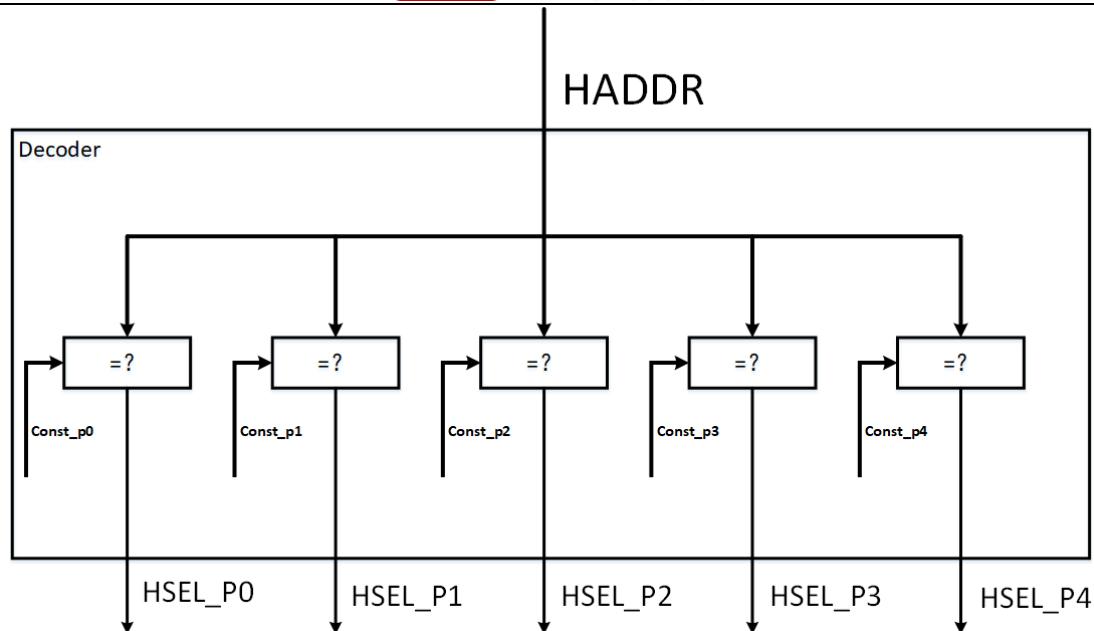


图 2-11 Decoder 内部结构

需要注意的是，对于每个外设，Decoder 利用地址总线生成选择信号 HSEL 所需要的宽度是不同的。例如 RAMCODE 的地址编码为 0x00000000-0x0000ffff，其有效长度为 0x0000-0xffff，那么 RAMCODE 对应的选择信号 (HSEL\_RAMCODE) 需要对 HADDR 的高 16bit 进行译码；而 WaterLight 的地址编码为 0x40000000-0x4000000f，其有效长度为 0x0-0xf，因此 WaterLight 对应的选择信号 (HSEL\_WaterLight) 需要对 HADDR 的高 28bit 进行译码。

Slave MUX 的作用则是通过每个外设的选择信号，对所有外设返回的读取数据 (HRDATA)、响应信号 (HRESP) 以及反馈信号 (HREADYOUT) 进行选择，保证返回给 Master 端口的数据来自于当前总线操作的目标外设。例如当前总线操作是读取 RAMCODE 的数据，那么所有外设的选择信号中只有 RAMCODE 对应的选择信号为 '1'，其他所有选择信号为 '0'，那么 Slave MUX 则根据这些选择信号选中 RAMCODE 返回的数据，保证处理器核能够正确读取。

根据以上分析，本手册已经编写好如图 2-12 所示的总线扩展模块。

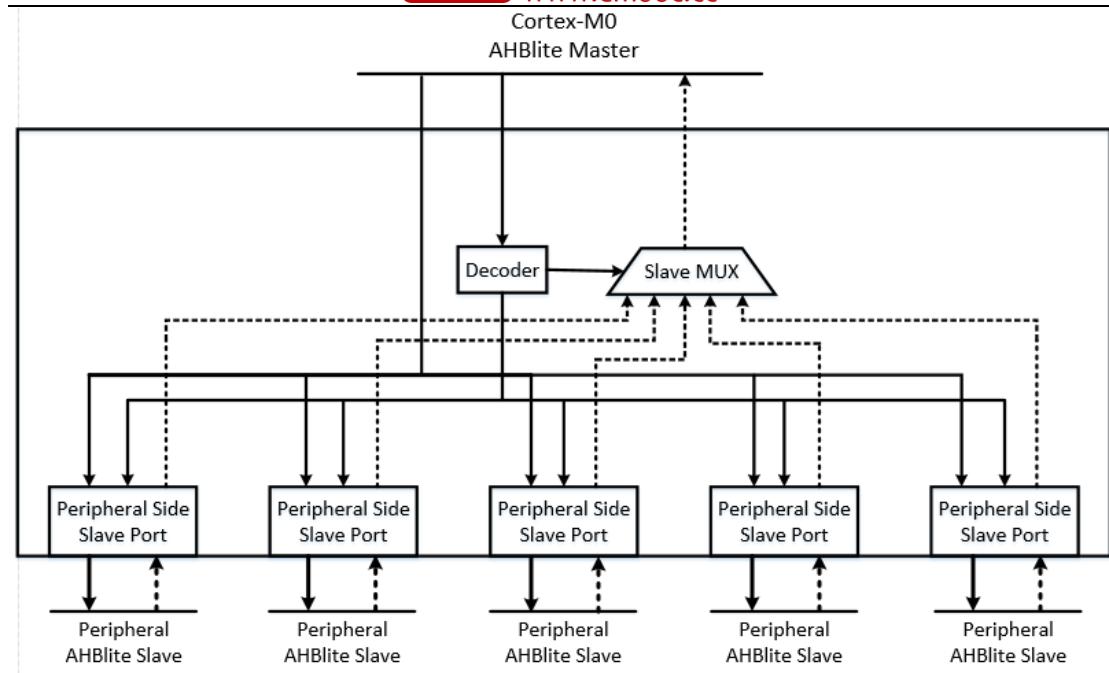


图 2-12 总线扩展模块结构图

在接下来的讲解中，每增加一个外设，只需要将外设 Slave 接口与图 2-8 中的 Peripheral Side AHBlite Master 接口相连接，再在 Decoder 模块中添加对应的译码模块生成选择信号即可。

## 2.3. Xilinx Block RAM 与 Distributed RAM

在 Xilinx ARTIX 7 系列 FPGA 中，总共有两类 Memory 资源，分别是 Block Memory 与 Distributed Memory。BRAM 由 FPGA 内专用 SRAM 构成，其输出为寄存器输出，及给出地址后的下一个时钟上升沿输出读取数据；而 DRAM 由 FPGA 内的 LUT 逻辑资源构成，其输出为逻辑输出，与时钟无关。由此可见，DRAM 的优势在于读取速度更快，但是其缺点则是在需要的 Memory 较大时，将会占用 FPGA 内部极大的逻辑资源，因此，本手册选用 BRAM 作为存储介质。

需要注意的是，此 Memory 存储的数据宽度为 32bit，然而总线操作则是以 8bit 为基

---

础，因此如果需要实现指令存储器的地址空间在 0x00000000-0x0000ffff（有效长度为 0x0000-0xffff，共 16bit），则只需要 Memory 地址则为有效地址的高 14bit。

## 第三章 搭建 SoC

### 3.1. 实验一：总线、RAM、汇编与调试

本节以一个简单的实验，在裸核+总线扩展模块的基础上，添加 Block RAM 与 BRAM 总线接口模块，并将其接入总线扩展模块预留的 Master 接口下，然后完成 Decoder 模块中对 RAM 的译码，如图 3-1 所示，最终经过 vivado 编译、综合、实现，下载至 FPGA 中。完成硬件设计后，本节实验还需要学习 keil 的使用，以及编写简单的汇编代码，并通过调试器将代码下载至 SoC 中，利用 keil 进行调试运行。

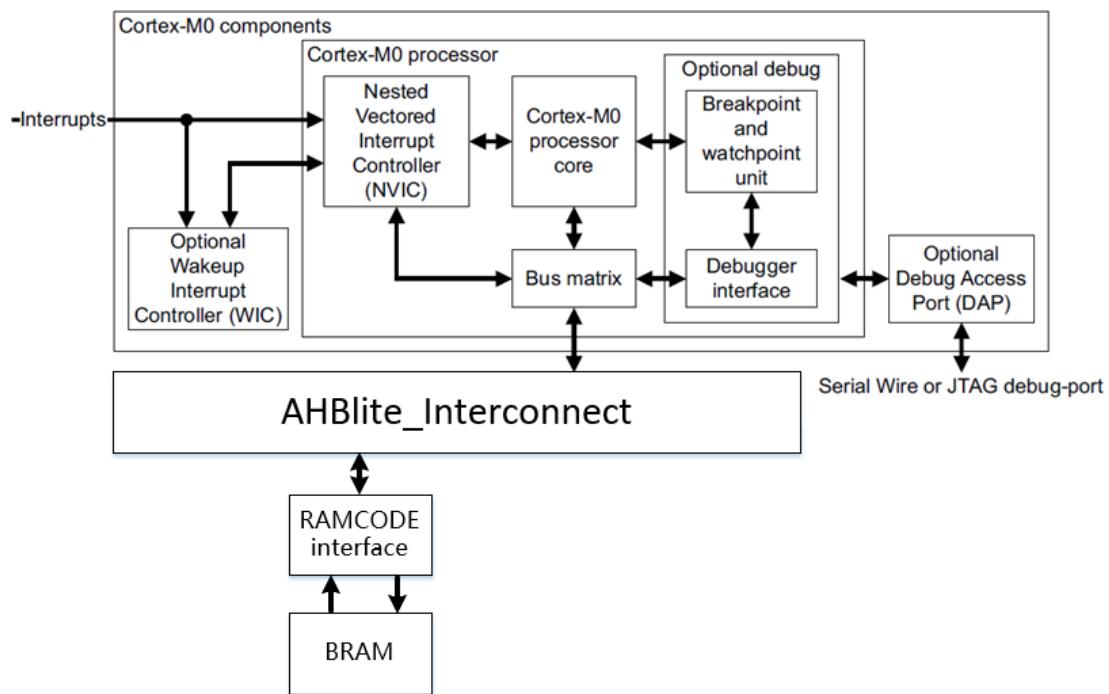


图 3-1 实验一搭建的 SoC

#### 3.1.1. 硬件部分

我们要搭建如图 3-1 所示的简单 SoC，总共需要完成修改两个部分的硬件设计：

- 在顶层文件中将 RAMCODE 总线接口与总线扩展模块连接

- 在总线扩展模块中的 Decoder 内添加对应的译码电路

第一步，在 “CortexM0\_SoC/Task1/rtl/AHBlite\_Decoder.v” 文件中修改 Decoder 模块代码。

A. 在端口参数部分，令 RAMCODE 使能有效。

```
/*RAMCODE enable parameter*/  
  
parameter Port0_en = 0,  
  
*****
```

改为：

```
/*RAMCODE enable parameter*/  
  
parameter Port0_en = 1,  
  
*****
```

B. 根据第二章所述的 memory map，RAMCODE 的总线编码为 0x00000000-0x0000ffff，因为对于一次总线操作，只要地址总线的高 16 位为 0，则 Decoder 认为这是一次对指令存储器的操作，进而生成指令存储器总线选择信号。在译码部分插入 RAMCODE 的译码器代码。

```
//0x00000000-0x0000ffff  
  
/*Insert RAMCODE decoder code there*/  
  
assign P0_HSEL = 1' b0;
```

```
*****
//0x00000000-0x0000ffff
/*Insert RAMCODE decoder code there*/
assign P0_HSEL = (HADDR[31:16] == 16'h0000) ? Port0_en : 1'b0;
*****

```

第二步，在顶层文件中将 RAMCODE 总线接口与总线扩展模块连接。

在 “CortexM0\_SoC/Task1/rtl/ CortexM0\_SoC.v” 中，已经完成了处理器核、总线扩展模块、RAMCODE 总线接口模块以及 Block RAM 模块的例化（调用这些模块，将这些模块添加至设计里面），但未在总线扩展模块接口部分连接 RAMCODE 总线接口。

```
/* Connect to Interconnect Port 0 */
.HCLK          (clk),
.HRESETn       (cpuresetn),
.HSEL          /*Port 0*/,
.HADDR         /*Port 0*/,
.HPROT         /*Port 0*/,
.HSIZE         /*Port 0*/,
.HTRANS        /*Port 0*/,
.HWDATA        /*Port 0*/,

```

```
.HWRITE          /*Port 0*/,
.HRDATA          /*Port 0*/,
.HREADY          /*Port 0*/,
.HREADYOUT      /*Port 0*/,
.HRESP           /*Port 0*/,
.BRAM_ADDR      (RAMCODE_ADDR),
.BRAM_RDATA    (RAMCODE_RDATA),
.BRAM_WDATA    (RAMCODE_WDATA),
.BRAM_WRITE     (RAMCODE_WRITE)

/*******************/
```

**改为：**

```
/* Connect to Interconnect Port 0 */

.HCLK            (clk),
.HRESETn        (cpuresetn),
.HSEL            (HSEL_P0),
.HADDR          (HADDR_P0),
.HPROT          (HPROT_P0),
.HSIZE          (HSIZE_P0),
.HTRANS         (HTRANS_P0),
.HWDATA         (HWDATA_P0),
```

```
.HWRITE      (HWRITE_P0),  
.HRDATA      (HRDATA_P0),  
.HREADY      (HREADY_P0),  
.HREADYOUT   (HREADYOUT_P0),  
.HRESP       (HRESP_P0),  
.BRAM_ADDR   (RAMCODE_ADDR),  
.BRAM_RDATA  (RAMCODE_RDATA),  
.BRAM_WDATA  (RAMCODE_WDATA),  
.BRAM_WRITE   (RAMCODE_WRITE)  
  
/***************************/
```

硬件部分已经完成了。

### 3.1.2. 汇编、启动与 Keil 工具

从 Keil 安装目录找到 Cortex-M0 的启动汇编代码 “startup\_CMSDK\_CM0.s ”，并根据目前 SoC 的进度做适当修改。

根据 ARMv6-M 架构参考手册，Cortex-M0 启动过程如下：

- 在复位使能时，CPU 处于 Reset 异常状态；
- 复位释放后，从地址 0x00000000 出加载栈顶地址，及汇编代码中 \_Vector 的第一行则为栈顶地址，由于我们目前的 SoC 没有数据存储器，自然也就没有堆栈一说，因此随便设置一个地址即可（此地址必须符合 Memory Map 定义的可读可写地址段，详情见 M0 的用户手册）；

- 从地址 0x00000004 初加载复位处理函数的地址；
- PC 改变为 0x00000004 中的值，开始执行复位处理，同时 CPU 的工作状态从异常模式切换为线程模式，开始正常工作。

由于没有数据存储器，因此我们不能进行相应的 load/store 指令，仅仅对 R0,R1 两个寄存器进行操作。

在 “CortexM0\_SoC/Task1/keil/startup\_CMSDK\_CM0.s” 文件中，实现 ARM 汇编编写计数，使得 R0 从 0 计数到 4 后重新开始计数，循环往复。

```
;Inset a loop algorithm there;
```

```
;*****
```

**改为：**

```
;Inset a loop algorithm there;
```

```
    MOVS R1, #4
```

```
Clear      MOVS R0, #0
```

```
Adder      ADDS R0, R0, #1
```

```
        CMP R0, R1
```

```
        BEQ Clear
```

```
        BNE Adder
```

```
;*****
```

汇编代码编写完成后，我们需要开始配置 Keil 工程，此过程参考《ARM Cortex-M0 权威指南》中，“在 SRAM 中调试程序”一章，具体过程如下。

第一步，打开 Keil，点击左上角 Project 菜单，选择 new uVision project，在“/CortexM0\_SoC/Task1/keil/”文件夹下新建名为 code 的工程，在第一个弹框初并选择 CMSDK\_CM0，如图 3-2。

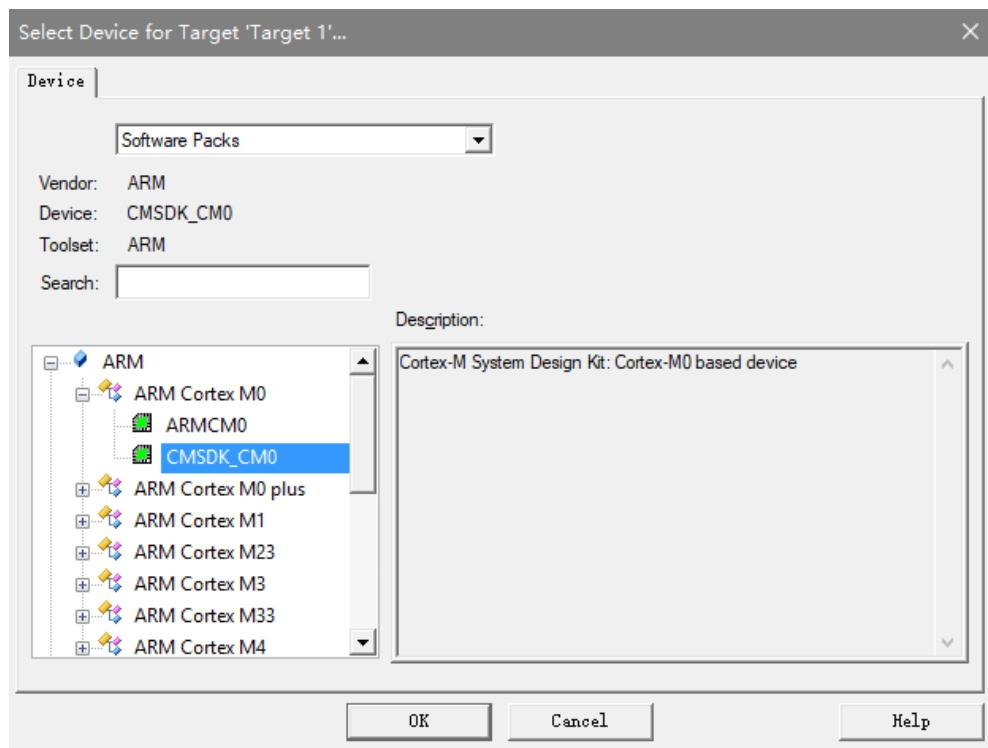


图 3-2 新建 Keil 工程，选择对应 CPU

第二步，在左侧 Project 导航栏中，展开 Target1，右键点击 Source Group 1，在工程中添加之前编写的汇编文件，如图 3-3。

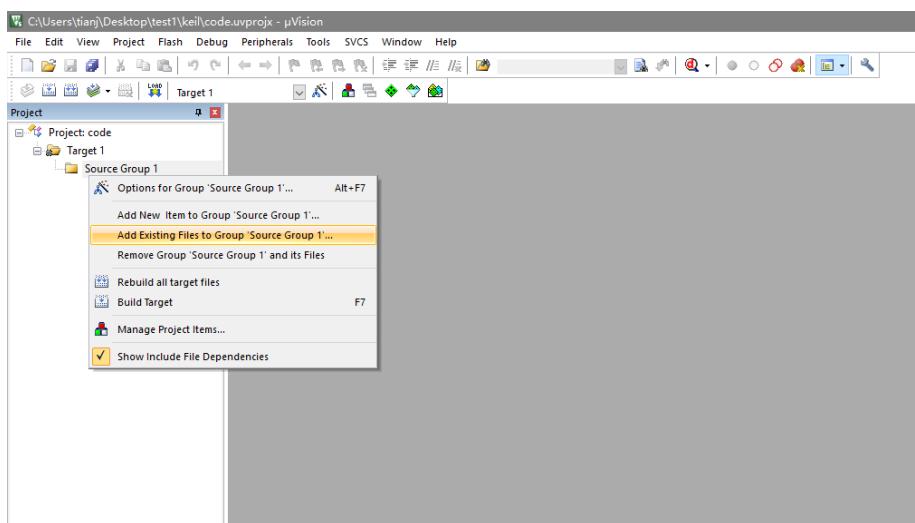


图 3-3 向工程中添加汇编文件

第三步，在左边导航栏处右键点击 Target 1，选择 Option，对 Target 栏进行配置，如图 3-4。此步骤为将片上起始地址为 0x00000000，大小为 0x10000 的 Memory 作为 ROM，Keil 将会通过调试器把程序下载到这一段存储器中。

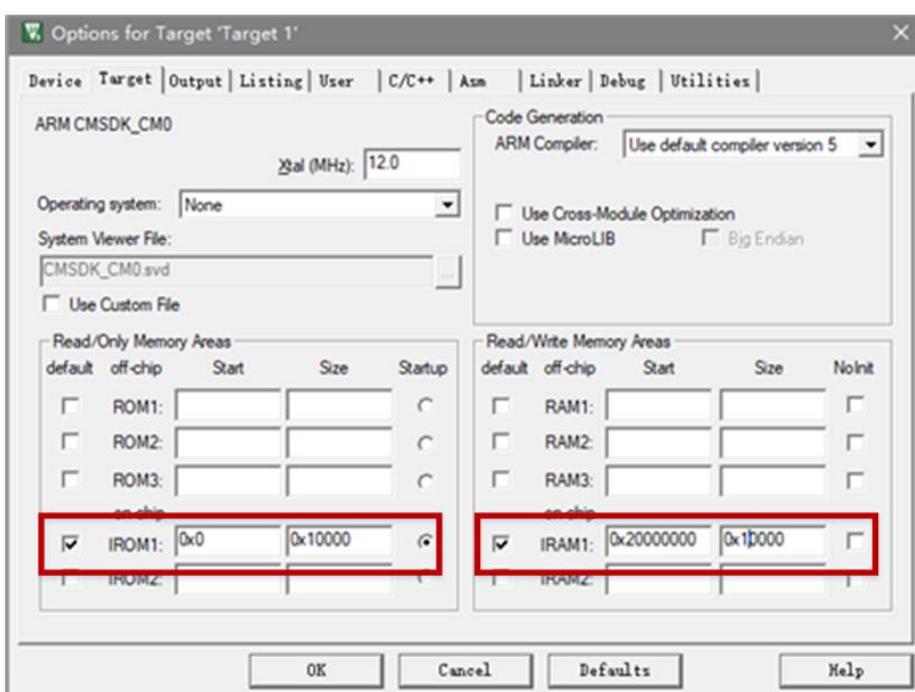


图 3-4 Target 配置

第四步，在 Output 栏处修改输出文件夹，点击 Select Folder for Objects，将输出文件地址从 Objects 改为上一级文件夹地址，及工程所在文件夹地址，如图 3-5.

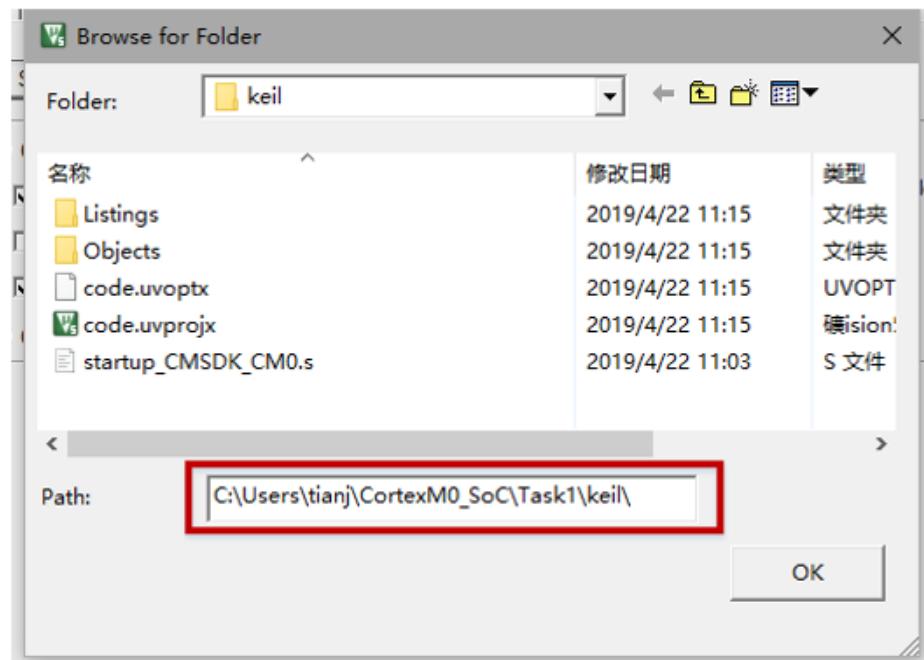


图 3-5 修改输出文件地址

第五步，在 User 栏添加两行指令，用于将 axf 文件转换为 modelsim 仿真所需要的 hex 文件，作为存储器的初始化文件，如图 3-6。

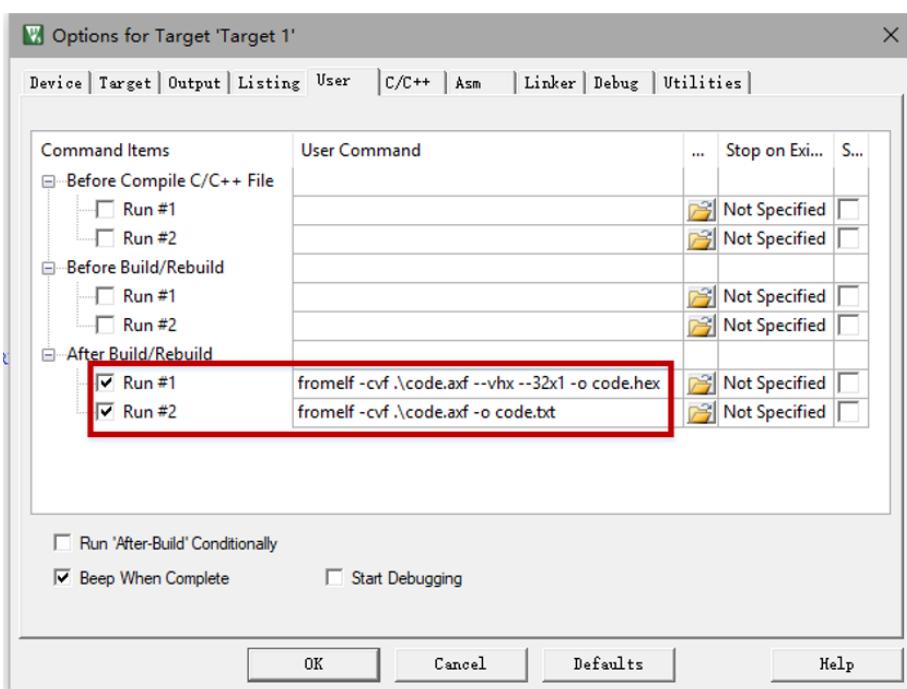


图 3-6 添加指令

勾选 Run #1，并在后面添加如下的代码：

```
fromelf -cvf .\code.axf --vhx --32x1 -o code.hex
```

勾选 Run #2，并在后面添加如下代码：

```
fromelf -cvf .\code.axf -o code.txt
```

第六步，在 Linker 处勾选 Use Memory Layout from Target Dialog 以及 Don't Search Standard Libraries，如图 3-7。

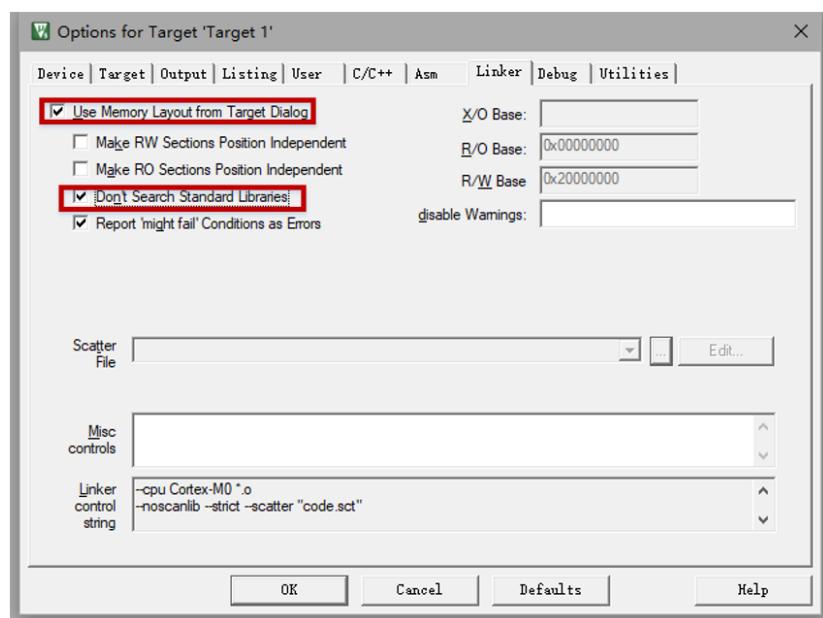


图 3-7 Linker 设置

第七步，在 Debug 处取消勾选 Load Application at Startup，选择 CMSIS-DAP Debugger 并在工程目录下新建名为 code.ini 的启动脚本文件，如图 3-8 所示，代码如下。

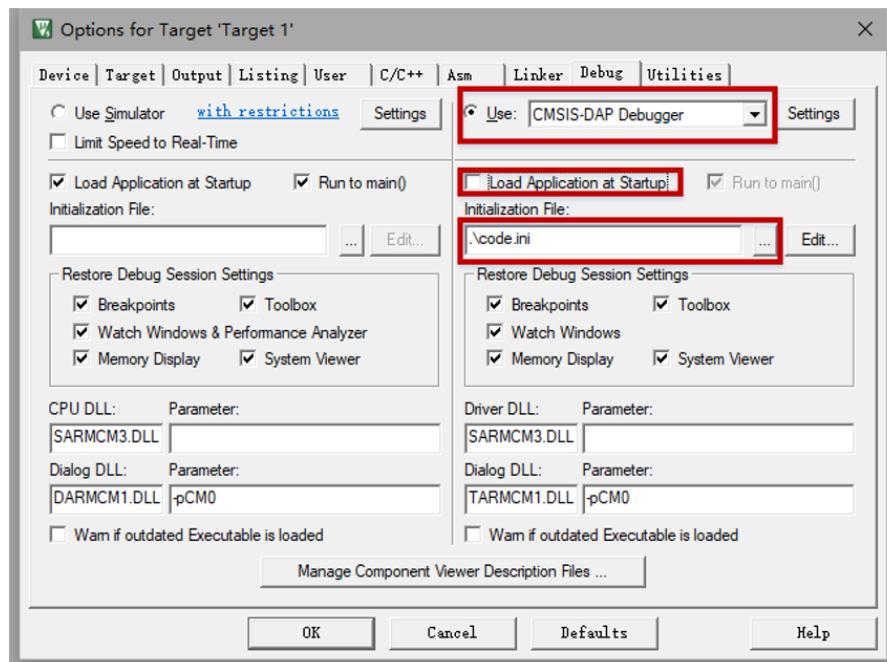


图 3-8 Debug 设置

```
reset

_WDWORD(0xE000ED08,0x00000000);

LOAD code.axf

SP = _RDWORD(0x00000000);

PC = _RDWORD(0x00000004);
```

第八步，进入 Debugger setting，选择 Flash Download 栏目，由于我们没有 Flash，所以选择 Do not Erase，并且取消勾选 Program 以及 Verify，如图 3-9。

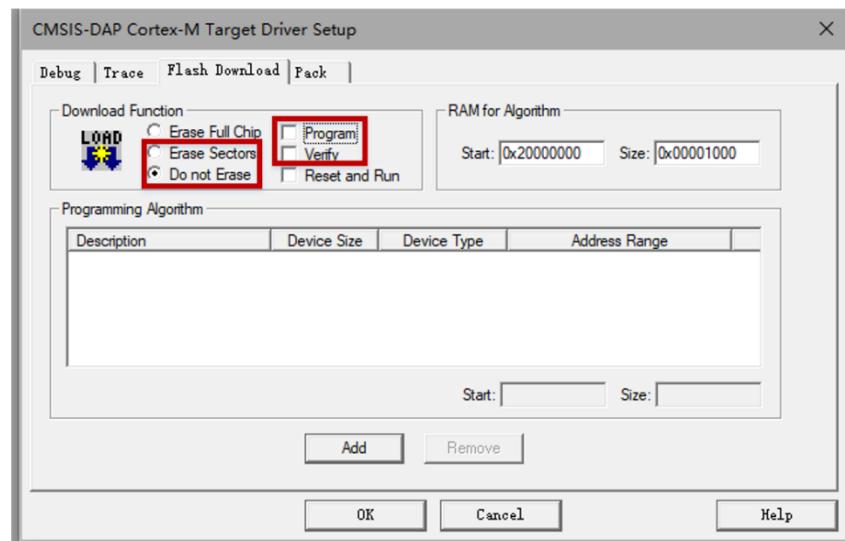


图 3-9 Flash Download 设置

第九步，在 Utilities 出取消勾选 Update Target before Debugging，如图 3-10 所示。

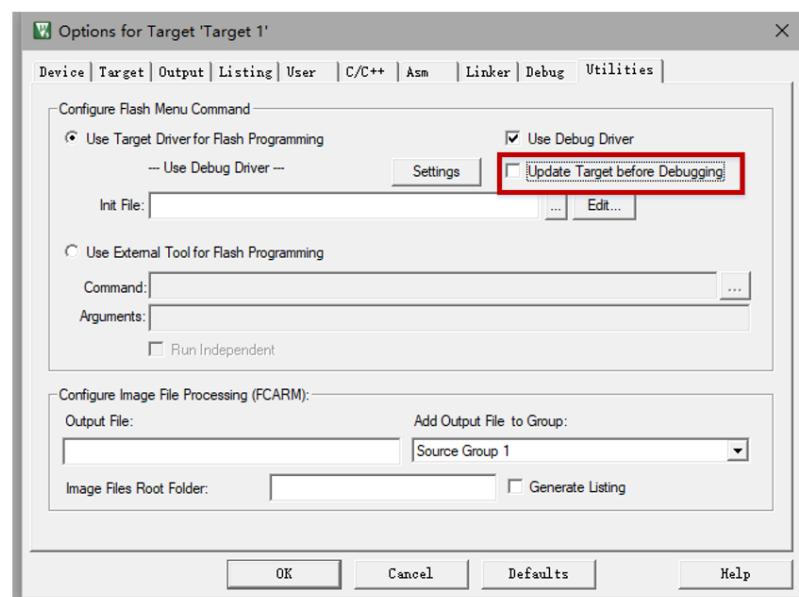


图 3-10 Utilities 设置

Keil 就设置完成，点击编译 Target，如图 3-11。

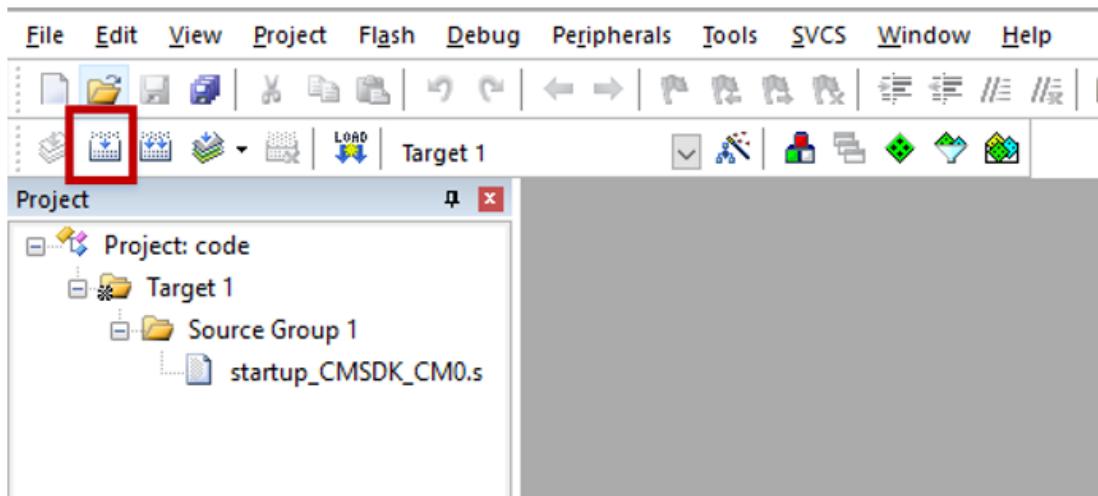


图 3-11 编译工程

### 3.1.3. Modelsim 仿真

接下来介绍如何用 modelsim 进行仿真。编写 testbench 文件 "CortexM0\_SoC\_vlg\_tst.v" 并保存在 "CortexM0\_SoC/Task1/modelsim/" 文件夹下。

```
`timescale 1 ps/ 1 ps

module CortexM0_SoC_vlg_tst();

reg clk;
reg RSTn;
reg TxD;

CortexM0_SoC i1 (
    .clk(clk),
    .RSTn(RSTn)
```

```
);  
  
initial begin  
    clk = 0;  
    RSTn=0;  
    #100  
    RSTn=1;  
  
end  
  
always begin  
    #10 clk = ~clk;  
end  
  
endmodule
```

注意，我们需要回头在“Block\_RAM.v”文件中修改 initial 里面 readmemh 函数文件地址，此地址在 Keil 工程文件夹下名为 code.hex，并且此地址必须为文件的绝对路径，其分隔符为‘ / ’ 而不是‘ \ ’。

然后我们打开 Modelsim，点击左上角 File 菜单，选择 new->project 创建一个 modelsim 工程，名为 “code”，工程地址应为“Task1/modelsim/”文件夹下，如图 3-12。

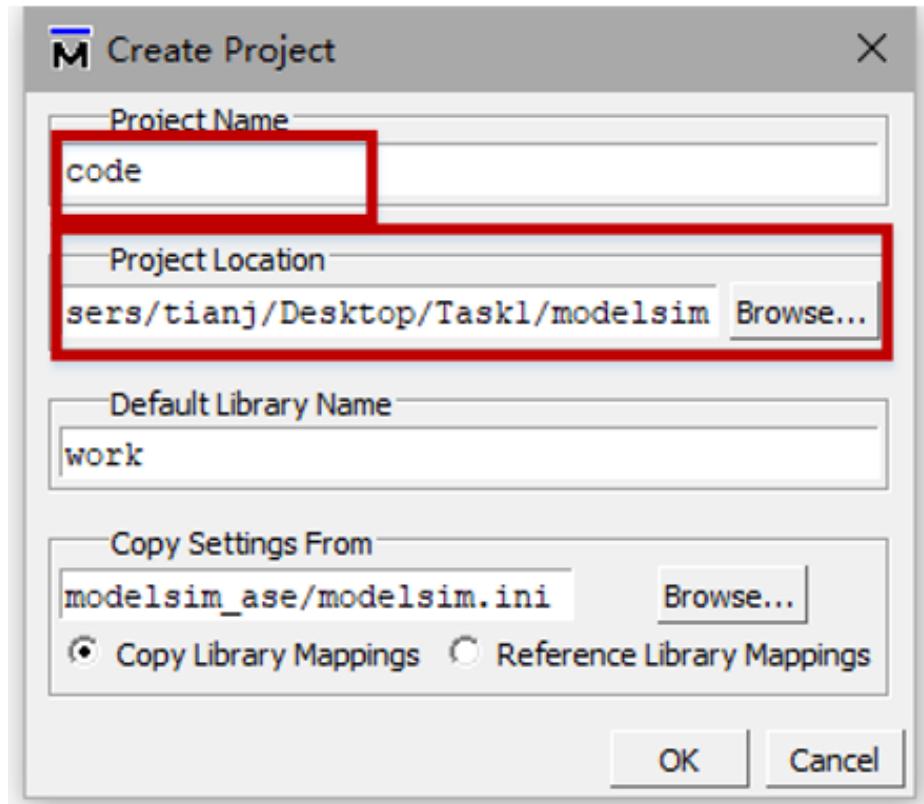


图 3-12 新建 modelsim 工程

点击 OK 后, 选择 Add Existing File, 如图 3-13。

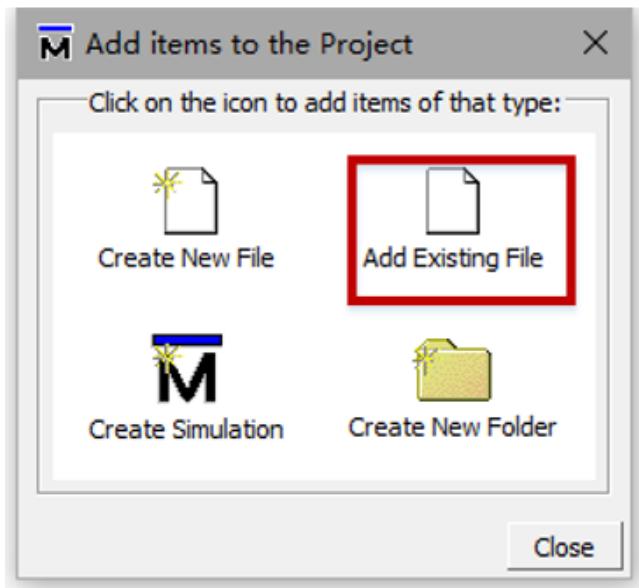


图 3-13 Add items to the Project

将“/rtl/”文件夹下所有 Verilog 文件以及“/modelsim/”文件夹下刚才编写的 testbench 文件添加至工程中, 并点击编译, 如图 3-14。

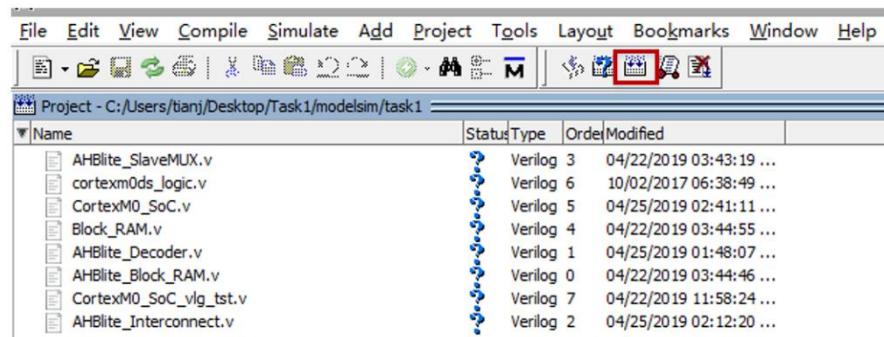


图 3-14 编译文件

所有文件前的问号? 变为对勾√后, 编译成功, 如图 3-15 所示。

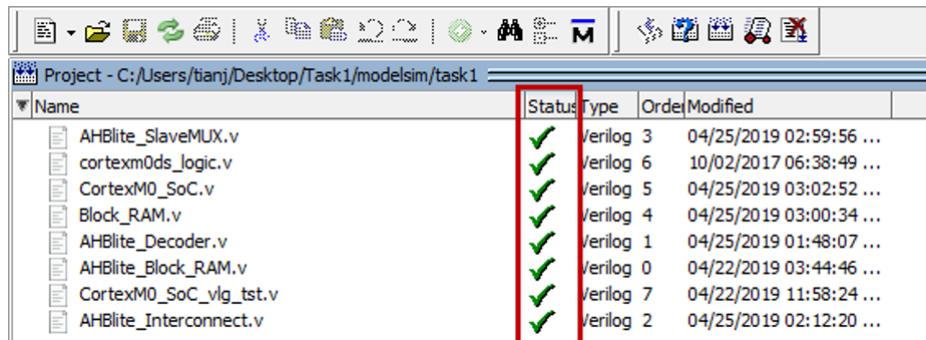


图 3-15 编译成功

选择左侧导航栏下方的 Library，展开 work，双击之前编写的 testbench 文件，如图

3-16。

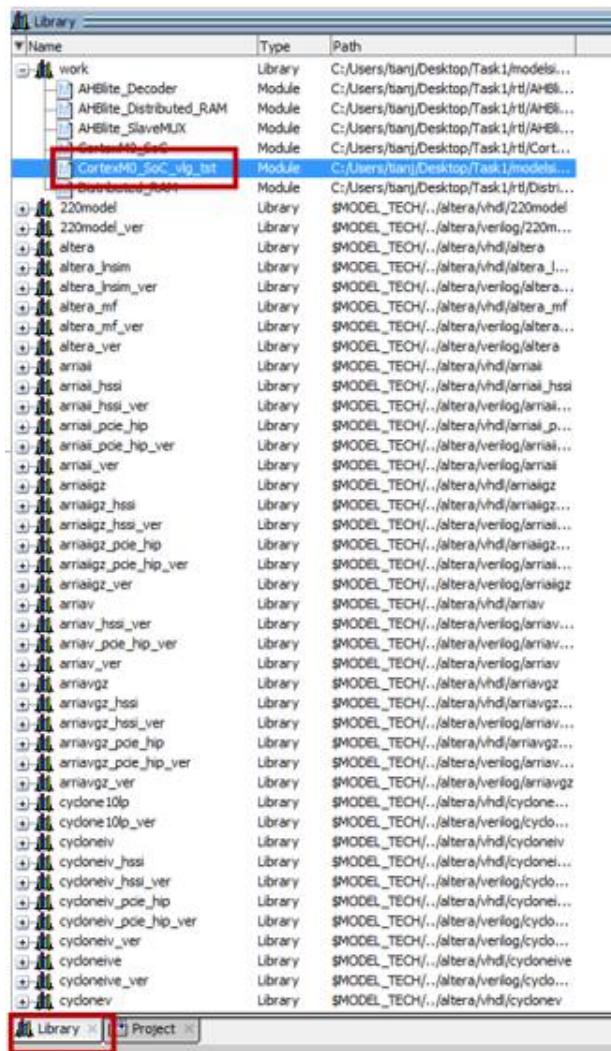


图 3-16 进入仿真

进入仿真界面后，在左侧仿真导航栏中，展开 i1，选中 u\_logic，如图 3-17 所示。

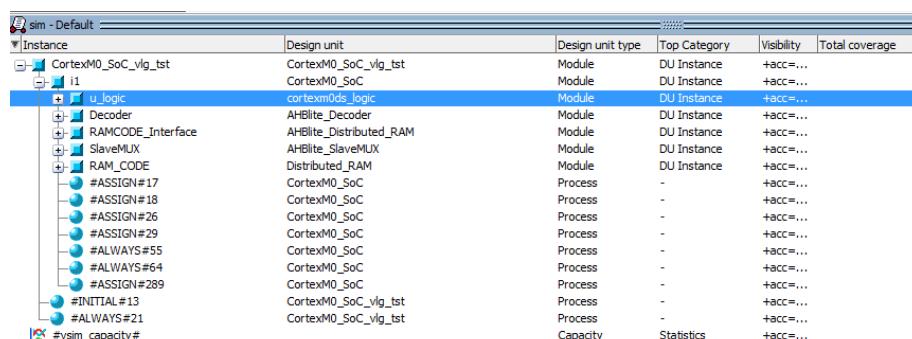


图 3-17 选择模块

在右侧信号列表中, 按住 Ctrl 选择 HADDR、HSIZE、HTRANS、HRDATA、vis\_r0\_o、

vis\_r1\_o、vis\_pc\_o 共 6 个信号, 点击右键选择 Add wave, 如图 3-18。

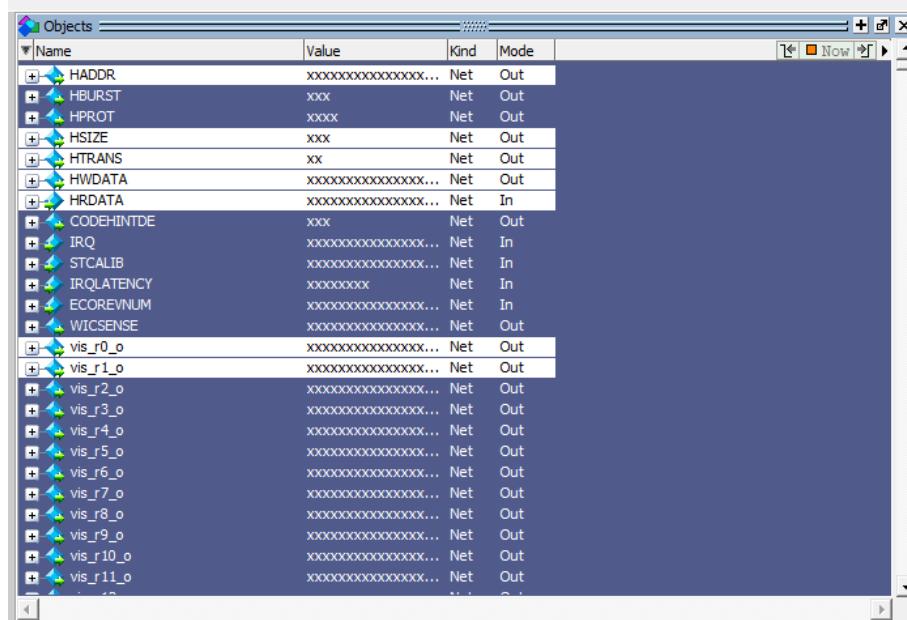


图 3-18 选择需要观察的信号

在示波器中选择所有信号, 右键点击 Radix->Hexadecimal, 将其改为 16 进制显示,

如图 3-19。

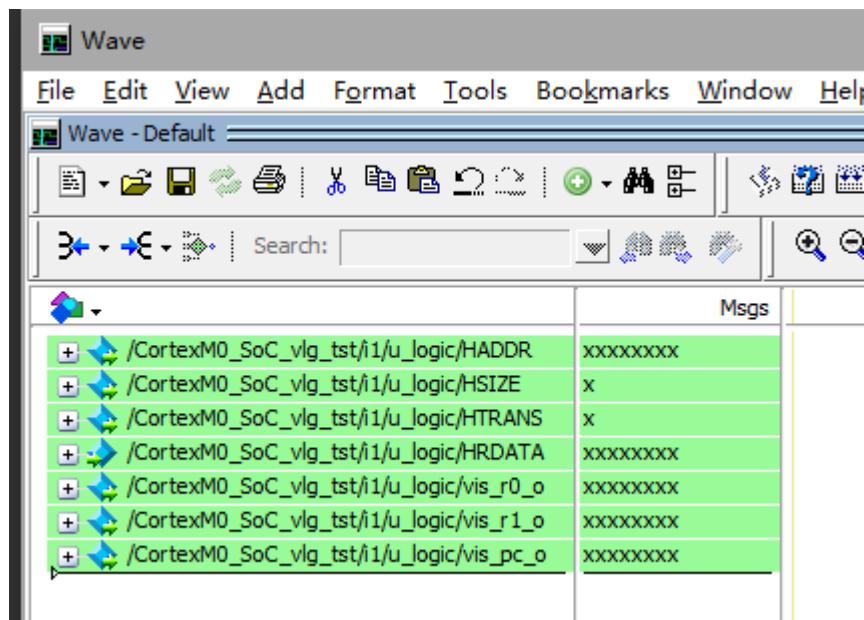


图 3-19 修改波形显示

最后点击左上角 File 菜单, 选择 Save Format, 将波形格式保存, 在下一次仿真时就

可以直接 load 波形格式文件，不用再每次都点击添加信号再改进制，如图 3-20 所示。

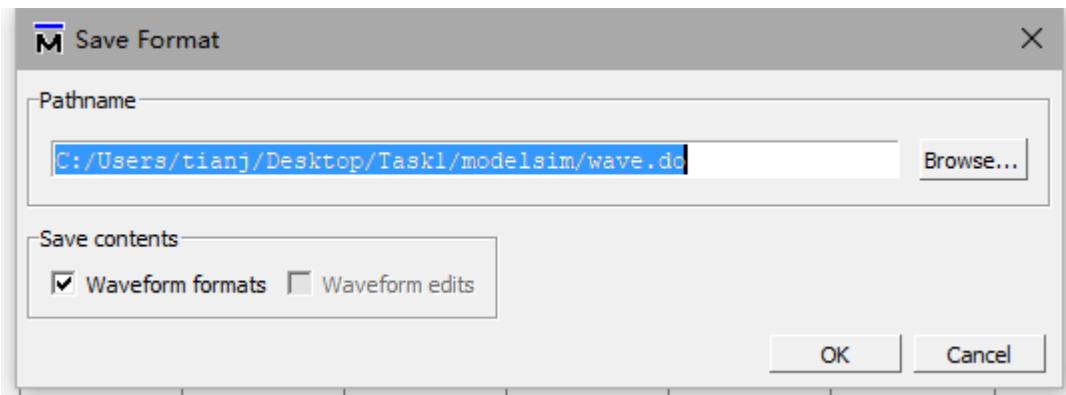


图 3-20 保存信号格式

设置仿真时间为 100ns，并点击开始仿真，如图 3-21。

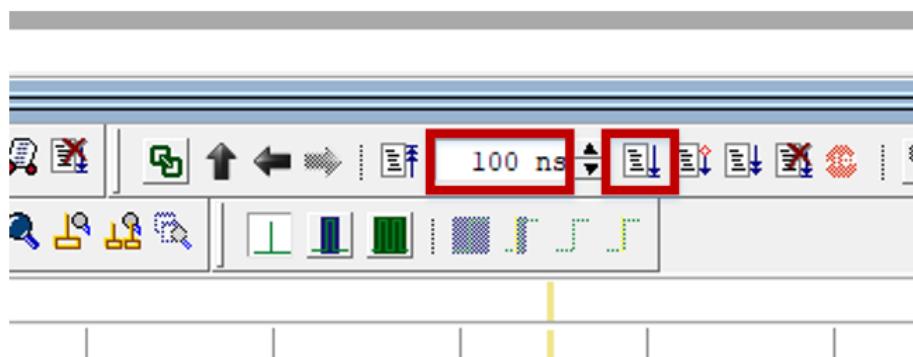


图 3-21 设置仿真时间并开始仿真

可以看到总线正常工作，CPU 正常执行，对应的 R0,R1 两个寄存器符合汇编代码所实现的功能，如图 3-22 所示。

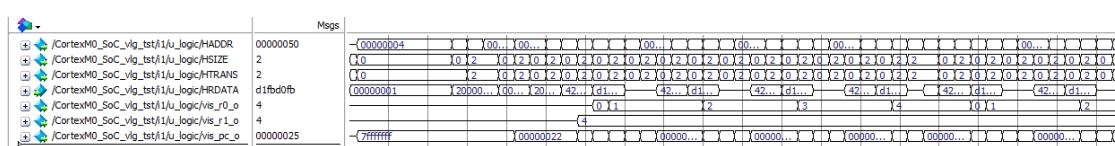


图 3-22 仿真波形

### 3.1.4. Vivado 与调试

仿真验证完成后，接下来将会介绍上板与使用 Keil 调试。

打开 vivado，在 “/Task1/vivado/” 文件夹下新建工程，直接点击 Create project，

然后点击右下角 Next, 如图 3-23。

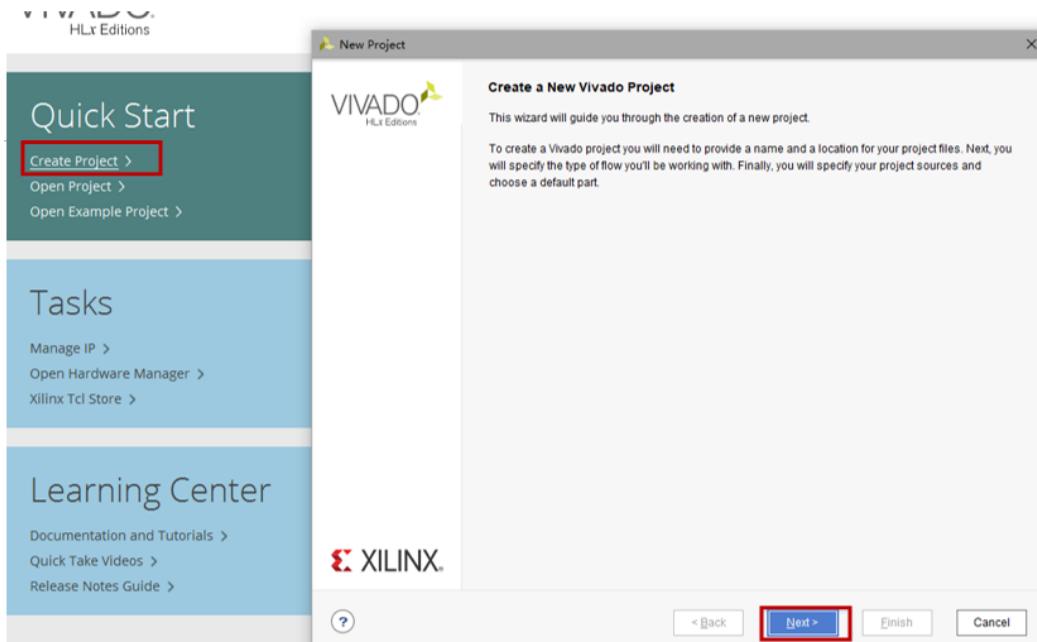


图 3-23 新建 vivado 工程

命名工程名称，选择工程地址，最后点击 Next，如图 3-24 所示。

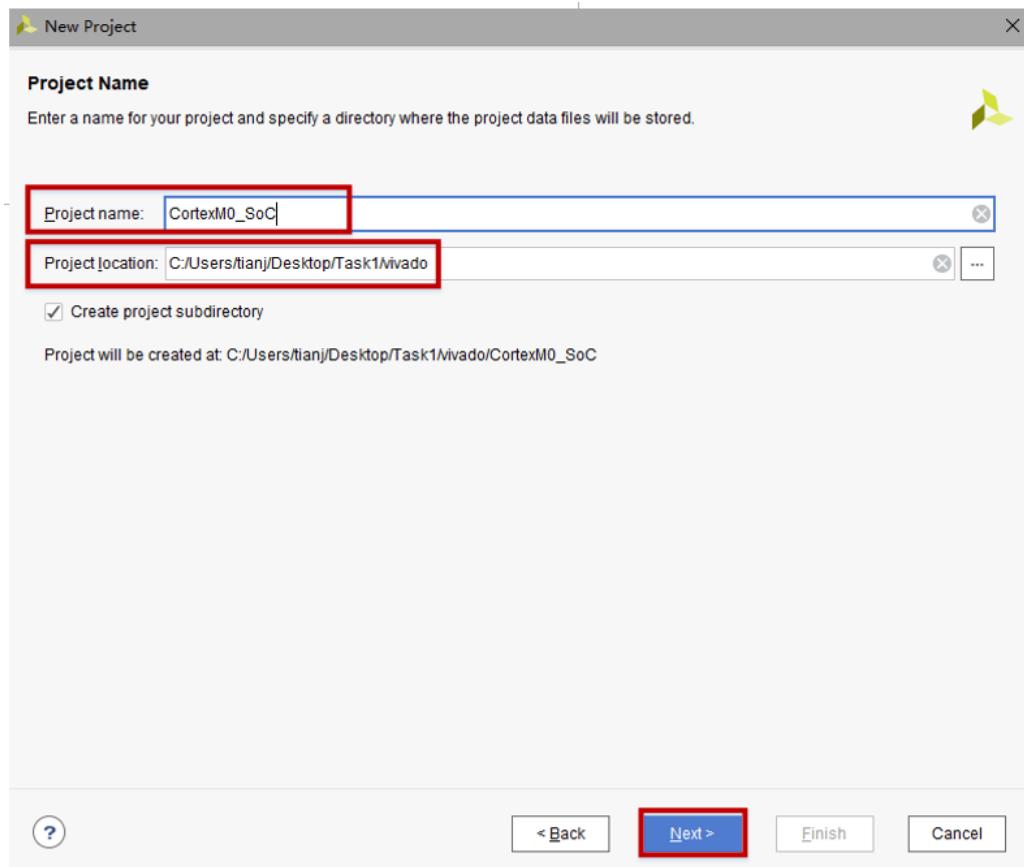


图 3-24 输入名称并选择位置

选择 RTL Project 并点击 Next, 如图 3-25 所示。

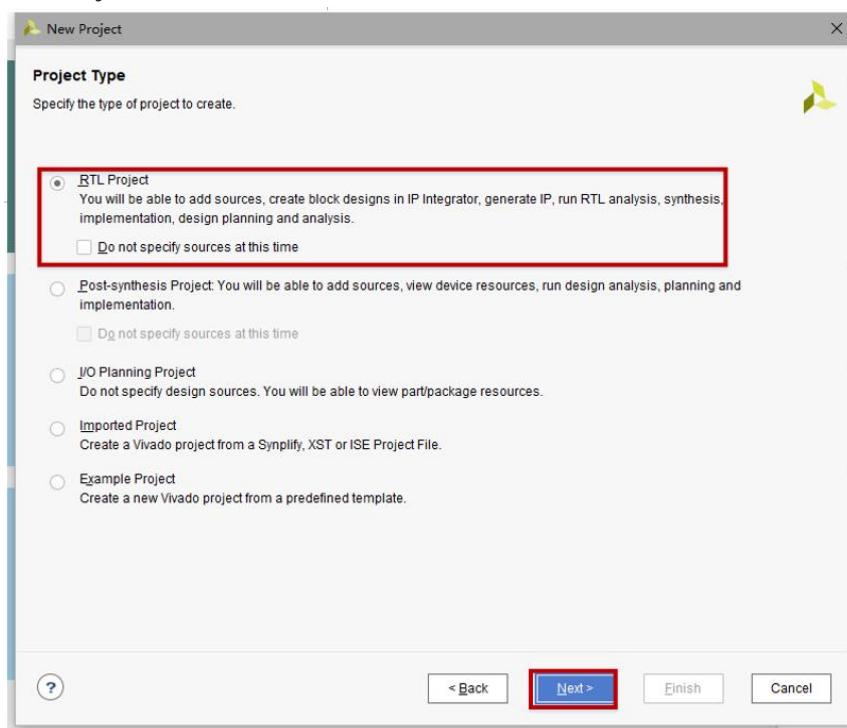


图 3-25 选择工程类型

点击 Add Direction, 在弹出的对话框中选择 “/Task1/rtl/” , 如图 3-26 所示, vivado

将会把 rtl 文件夹下所有 verilog 文件添加至工程，最后点击 Next。

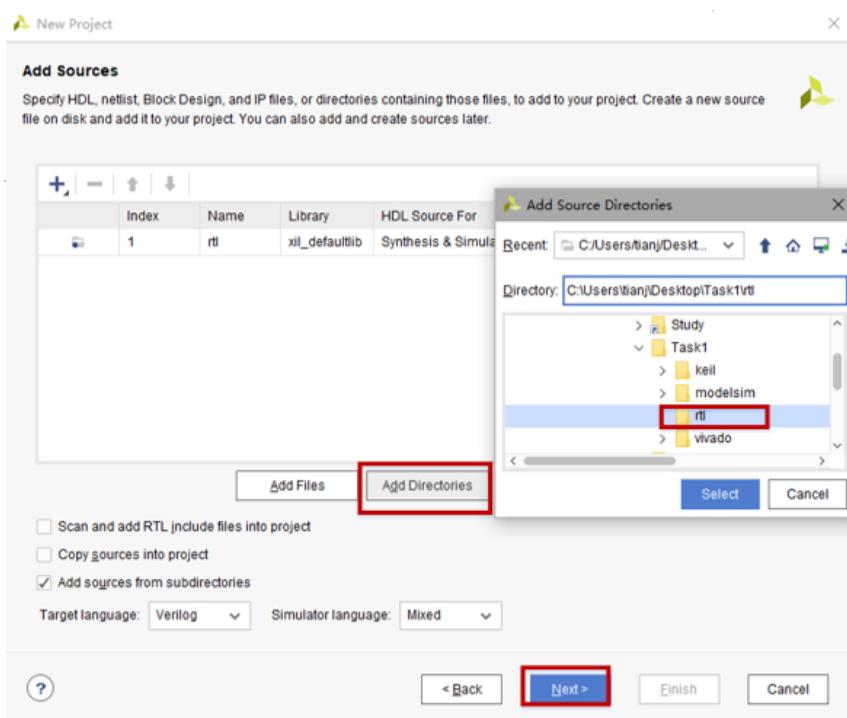


图 3-26 添加源文件地址

选择创建约束文件，并将其命名为 pin，最后点击 Next，如图 3-27。

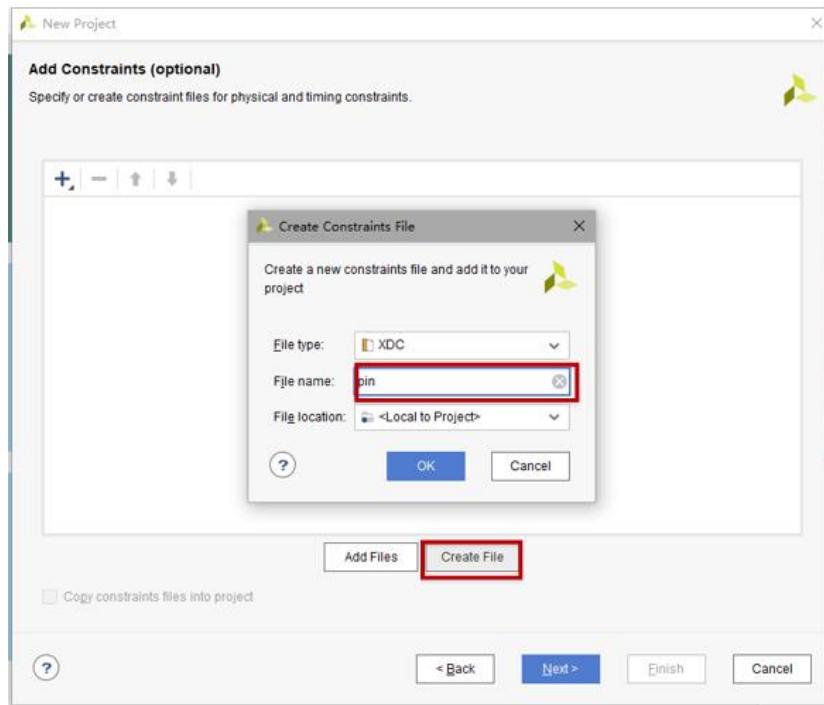


图 3-27 创建约束文件

选择芯片型号，首先在 Family 处选择 Artix-7，并在下方列表中选择 xc7a35tftg256-

2，最后点击 Next，如图 3-28。

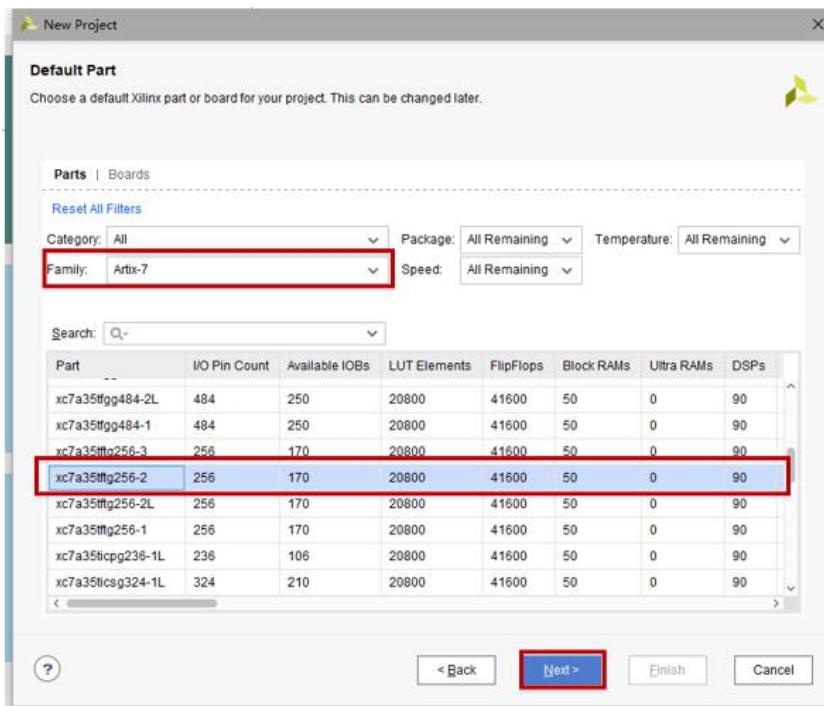


图 3-28 选择芯片型号

一路点击 Next 最后点击 Finish，完成 vivado 工程的创建。点击 Project Manager 中 Source 栏中的 constrain，双击刚才创建的 pin.xdc，如图 3-29。

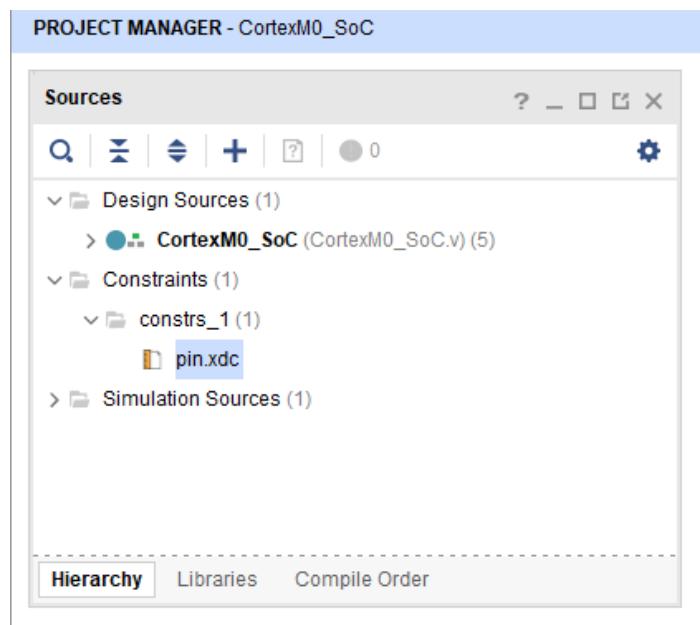


图 3-29 选择约束文件

在文本编辑界面内，编写如下所示的管脚约束文件，并保存。

```
##clk

set_property PACKAGE_PIN D4 [get_ports clk]

set_property IOSTANDARD LVCMS33 [get_ports clk]

##RSTn

set_property PACKAGE_PIN T9 [get_ports RSTn]

set_property IOSTANDARD LVCMS33 [get_ports RSTn]

##DEBUGGER

set_property PACKAGE_PIN H14 [get_ports SWDIO]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]  
  
set_property PACKAGE_PIN H12 [get_ports SWCLK]  
  
set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]  
  
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]
```

点击 vivado 左侧导航栏中下方的 Generate Bitstream, vivado 将会一次执行 RTL

ANALYSIS、SYNTHESIS、IMPLEMENTATION、Generate Bistream，如图 3-30。



图 3-30 vivado 工作开始

等待完成后，点击左侧导航栏中 Open Implemented Design->Schematic，可以看到 SoC 经过 vivado 布局布线后的电路图，如图 3-31 所示。

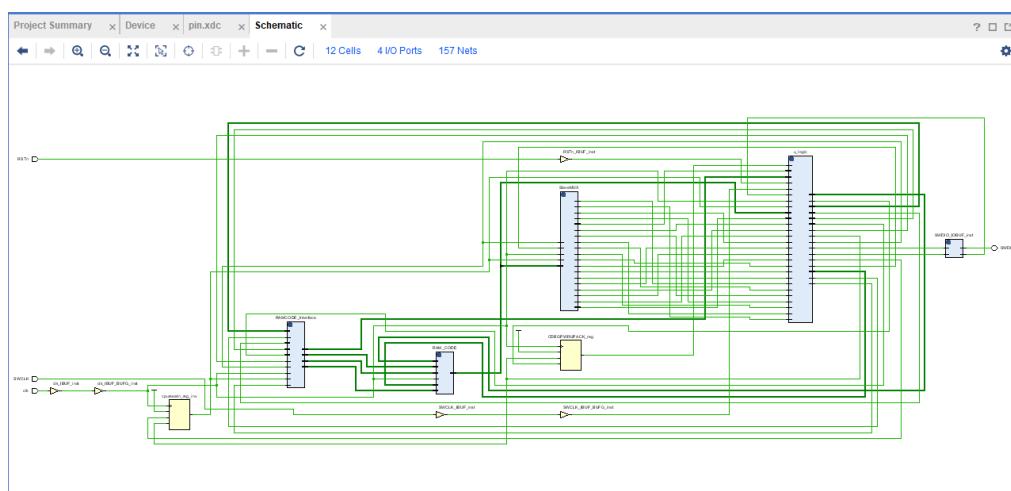


图 3-31 Schematic

然后点击 Schmatic 旁边的 Device，可以看到 FPGA 内部具体的资源利用情况，如图 3-32 所示。

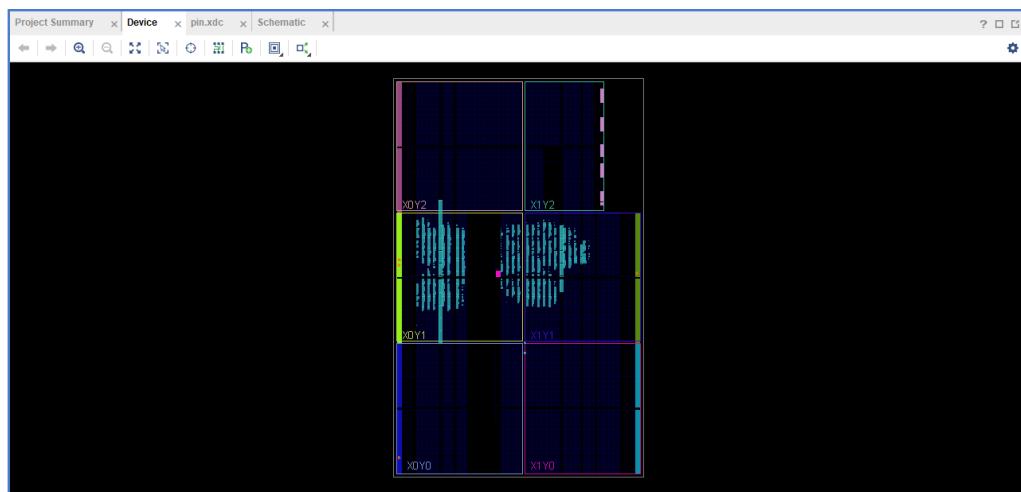


图 3-32 Device

通过观察之前编写的约束代码，可以看到 SoC 的复位信号连接在 T9 开关上（注意，此复位信号在系统内为低有效，因此在接下来的调试过程中需要把 T9 开关掰上去），调试信号连接在 H14、H12 两个引脚上。开发板上已经集成了 CMSIS-DAP 的调试器，所以将 SWDIO 和 SWCLK 两个端口约束到 H14 和 H12 引脚以后，便可以将 Debugger 连接至 PC。连接方式如图 3-33 所示。



图 3-33 Debugger 连接图

最后，在左侧导航栏下方 PROGRAM AND DEBUG 栏内，展开 Open Hardware Manager、Open Target->Auto Connect、Program Device->xc735t\_0，等待比特流下载完成，如图 3-34。

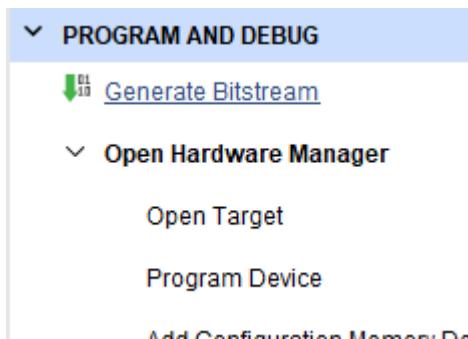


图 3-34 下载比特流

此时打开 Keil 工程，选择 Debugger Setting，看到图 3-35 所示的 IDCODE 则表示 Debugger 与 CPU 成功连接。

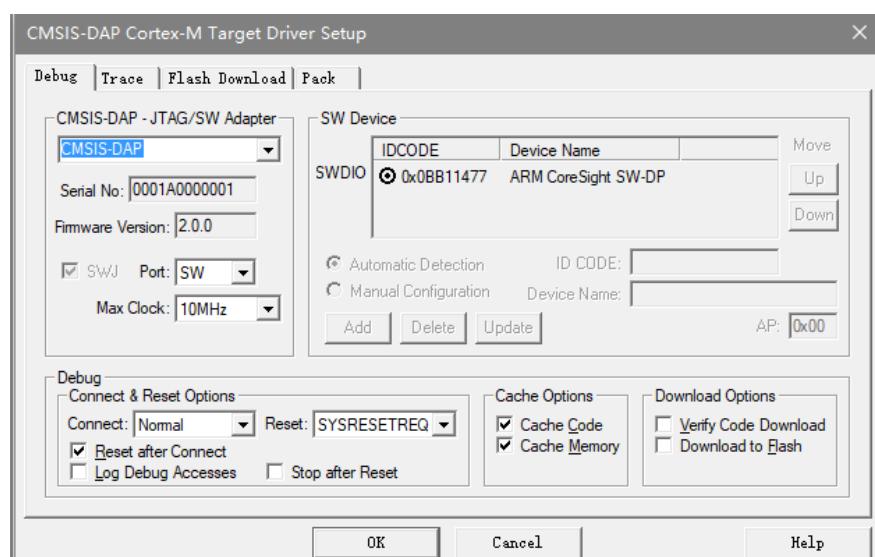


图 3-35 IDCODE 信息

点击 Start Debug, Keil 成功进入 Debug, 按 F11 进行单步调试, 可以看到 CPU 按照汇编代码开始正常运行, R0,R1 寄存器正常赋值, 如图 3-36。

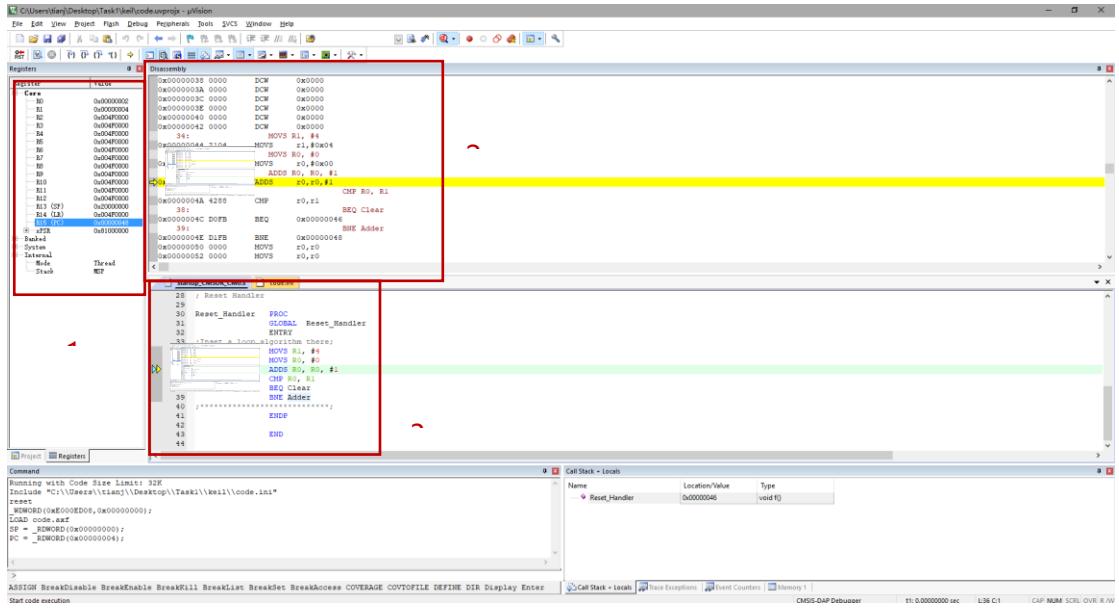


图 3-36 Debug

在红框 1 处为 CPU 内部寄存器的值, 能够实时显示每一步寄存器值的变化, 而在红框 2 处为当前执行的汇编代码以及代码地址, 在红框 3 处显示的是当前执行的源代码, 这里的源代码既可以是汇编代码也可以是 C 语言代码。根据之前汇编语言编写的循环计数, 在红框 1 处的 R1 的值保持为 4, 而 R0 则应该在 0-4 之间循环计数。

至此, 本小节结束, 在本小节中, 我们完成了最小 SoC 的设计, 完成 AHB 总线扩展, 编写了 BRAM 以及 RAM 总线接口, 编写了汇编代码并成功仿真与调试, SoC 的整个设计流程大致如此。接下来的几个小节将会继续深入 SoC 的搭建, 在步骤细节上也不再花费大量的篇幅, 并以此小节作为参考。

### 3.2. 实验二: 数据存储器与流水灯

堆栈对于一个处理器是至关重要的，没有堆栈就无法完成函数调用与参数传递等，因此我们需要设计一个数据存储器用来存放堆栈，在本小节中，我们将添加一个简单的外设—流水灯，最终实现如图 3-37 所示的 SoC。

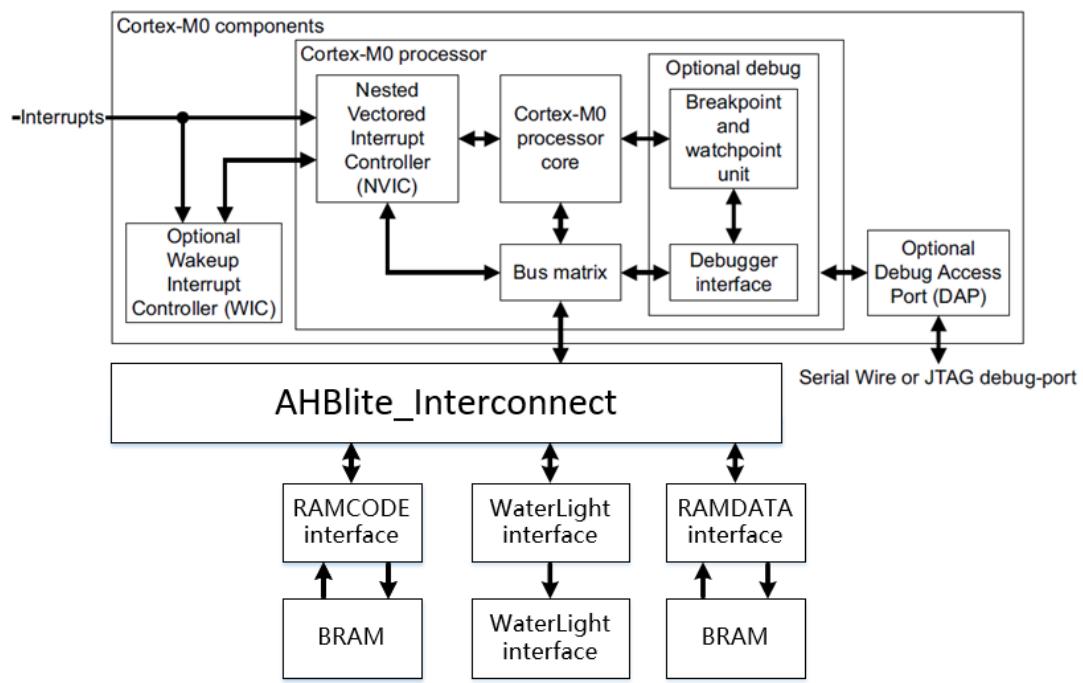


图 3-37 实验二搭建的 SoC

### 3.2.1. 硬件部分

首先向 SoC 中添加 RAMDATA，参考实验一添加 RAMCODE 步骤。在顶层模块中已经添加好 RAMDATA 总线接口以及对应的 Block RAM，需要完成将 RAMDATA 总线接口接入总线扩展模块中预留的 P1 接口。

第一步，在“AHBlite\_Decoder.v”中 RAMDATA (Port 1) 端口参数将其使能。

```
/*RAMDATA enable parameter*/

parameter Port1_en = 0,
/*
******/
```

改为：

```
/*RAMDATA enable parameter*/  
  
parameter Port1_en = 1,  
  
/*******************/
```

第二步，根据第二章所述的 memory map，RAMDATA 的总线编码为 0x20000000-0x2000ffff，因为对于一次总线操作，只要地址总线的高 16 位为 0x2000，则 Decoder 认为这是一次对数据存储器的操作，进而生成数据存储器总线选择信号。在译码部分插入 RAMDATA 的译码器代码。

```
//0x20000000-0x2000ffff  
  
/*Insert RAMDATA decoder code there*/  
  
assign P1_HSEL = 1' b0;  
  
/*******************/
```

改为：

```
//0x20000000-0x2000ffff  
  
/*Insert RAMDATA decoder code there*/  
  
assign P1_HSEL = (HADDR[31:16] == 16'h2000) ? Port1_en : 1'b0;  
  
/*******************/
```

文件 “CortexM0\_SoC/Task2/rtl/WaterLight.v” 为一个设计好的流水灯，流水灯总共设计为 4 个模式，由一个 8bit 输入作为选择：

- 全灭模式：所有灯灭掉
- 左移模式：同一时间点亮一个灯，依次左移，对应输入为 0x01
- 右移模式：同一时间点亮一个灯，依次右移，对应输入为 0x02
- 闪烁模式：所有灯同时闪烁，对应输入为 0x03

同时，流水灯闪烁速度由一个 32bit 输入作为控制。这里规定，流水灯模式控制寄存器地址为 0x40000000，流水灯速度控制寄存器地址为 0x40000004。

流水灯外设模块如图 2-38。

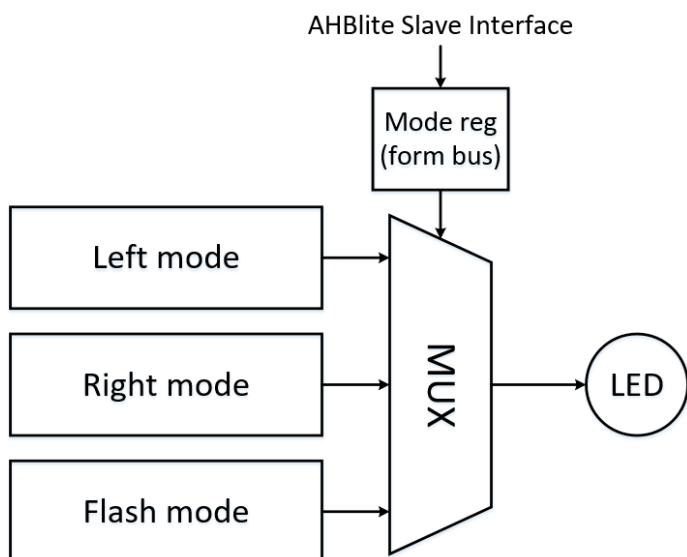


图 3-38 流水灯结构

同理，向 SoC 中添加 WaterLight，参考实验一添加 RAMCODE 步骤。在顶层模块中已经添加好流水灯总线接口以及对应的硬件代码，需要完成将流水灯总线接口接入总线扩展模块中预留的 P2 接口。

第一步，在“ AHBlite\_Decoder.v ”中 WaterLight (Port 2) 端口参数将其使能。

```
/*WaterLight enable parameter*/
```

```
parameter Port2_en = 0,
```

```
*****
```

**改为：**

```
/* WaterLight enable parameter*/
```

```
parameter Port2_en = 1,
```

```
*****
```

第二步，根据第二章所述的 memory map，WaterLight 的总线编码为 0x40000000-0x4000000f，因为对于一次总线操作，只要地址总线的高 28 位为 0x4000000，则 Decoder 认为这是一次对流水灯的操作，进而生成流水灯总线选择信号。在译码部分插入流水灯的译码器代码。

```
//0x40000000 WaterLight MODE
```

```
//0x40000004 WaterLight SPEED
```

```
/*Insert WaterLight decoder code there*/
```

```
assign P2_HSEL = 1' b0;
```

```
*****
```

**改为：**

```
//0x40000000 WaterLight MODE
```

```
//0x40000004 WaterLight SPEED

/*Insert WaterLight decoder code there*/

assign P2_HSEL = (HADDR[31:4] == 28'h4000000) ? Port2_en : 1'b0;

/*******************/
```

接下来，需要在顶层模块中将流水灯总线接口与数据存储器总线接口分别与总线扩展模块的 P1、P2 端口连接，流水灯和数据存储器以及它们的总线接口都已经在顶层模块“CortexM0\_SoC.v”中例化完成，只需要将其总线接口的连线部分补充完毕，实现正确的端口连接。

第一步，将数据存储器 RAMDATA 总线接口与总线扩展接口 P1 端口连接。

```
/* Connect to Interconnect Port 1 */

.HCLK          (clk),
.HRESETn       (cpuresetn),
.HSEL          /*Port 1*/,
.HADDR         /*Port 1*/,
.HPROT         /*Port 1*/,
.HSIZE         /*Port 1*/,
.HTRANS        /*Port 1*/,
.HWDATA        /*Port 1*/,
.HWRITE        /*Port 1*/,
.HRDATA        /*Port 1*/,
.HREADY        /*Port 1*/,
```

```
.HREADYOUT      /*Port 1/),  
  
.HRESP          /*Port 1/),  
  
.BRAM_ADDR     (RAMDATA_ADDR),  
  
.BRAM_RDATA   (RAMDATA_RDATA),  
  
.BRAM_WDATA   (RAMDATA_WDATA),  
  
.BRAM_WRITE   (RAMDATA_WRITE)  
  
/*******************/
```

**改为：**

```
/* Connect to Interconnect Port 1 */  
  
.HCLK          (clk),  
  
.HRESETn       (cpuresetn),  
  
.HSEL          (HSEL_P1),  
  
.HADDR         (HADDR_P1),  
  
.HPROT         (HPROT_P1),  
  
.HSIZE         (HSIZE_P1),  
  
.HTRANS        (HTRANS_P1),  
  
.HWDATA        (HWDATA_P1),  
  
.HWRITE        (HWRITE_P1),  
  
.HRDATA        (HRDATA_P1),  
  
.HREADY        (HREADY_P1),
```

```
.HREADYOUT      (HREADYOUT_P1),  
  
.HRESP          (HRESP_P1),  
  
.BRAM_ADDR     (RAMDATA_ADDR),  
  
.BRAM_RDATA    (RAMDATA_RDATA),  
  
.BRAM_WDATA    (RAMDATA_WDATA),  
  
.BRAM_WRITE    (RAMDATA_WRITE)  
  
/*******************/
```

第二步，将流水灯 WaterLight 总线接口与总线扩展接口 P2 连接。

```
/* Connect to Interconnect Port 2 */  
  
.HCLK           (clk),  
  
.HRESETn        (cpuresetn),  
  
.HSEL           /*Port 2*/),  
  
.HADDR          /*Port 2*/),  
  
.HPROT          /*Port 2*/),  
  
.HSIZE          /*Port 2*/),  
  
.HTRANS         /*Port 2*/),  
  
.HWDATA         /*Port 2*/),  
  
.HWRITE         /*Port 2*/),  
  
.HRDATA         /*Port 2*/),  
  
.HREADY         /*Port 2*/),  
  
.HREADYOUT     /*Port 2*/),
```

```
.HRESP      /*Port 2*/),  
  
.WaterLight_mode (WaterLight_mode),  
  
.WaterLight_speed (WaterLight_speed)  
  
/***************************/
```

改为：

```
/* Connect to Interconnect Port 2 */  
  
.HCLK      (clk),  
  
.HRESETn   (cpuresetn),  
  
.HSEL      (HSEL_P2),  
  
.HADDR     (HADDR_P2),  
  
.HPROT     (HPROT_P2),  
  
.HSIZE     (HSIZE_P2),  
  
.HTRANS    (HTRANS_P2),  
  
.HWDATA   (HWDATA_P2),  
  
.HWRITE    (HWRITE_P2),  
  
.HRDATA   (HRDATA_P2),  
  
.HREADY    (HREADY_P2),  
  
.HREADYOUT (HREADYOUT_P2),  
  
.HRESP     (HRESP_P2),  
  
.WaterLight_mode (WaterLight_mode),
```

```
.WaterLight_speed (WaterLight_speed)
```

```
/***************************/
```

### 3. 2. 2. 汇编

按照实验一所述，新建 keil 工程，编写汇编程序，让流水灯流起来。

在 “/Task2/keil/startup\_CMSDK\_CM0.s” 文件中，程序进入 WaterLight 段后，R0 存储流水灯模式，R1 存储模式转换计数器地址，R2 存储流水灯模式寄存器地址，R4 存储计数器时间。当流水灯切换到当前模式后，程序跳转至 delay 段开始计时，流水灯模式保持不变知道计数器计时完成，程序返回，流水灯模式改变，循环往复。

需要注意的是，为了方便仿真观察，我们将流水灯模式转换间隔时间设置得非常小，在接下来的上板调试时，需要重新修改 R4 的值，使流水灯模式转换时间保持在 10s 左右。

在汇编文件中补充 delay 段，利用 R4 计数至 10 后返回实现 delay 功能。

```
; Finish function delay
```

```
.....
```

改为：

```
; Finish function delay
```

```
delay ADDS R1, R1, #1
```

```
LDR R4, =0x10
```

CMP R4, R1

BNE delay

BX LR

.....

汇编代码修改完成后，点击编译，自动生成目标文件。

### 3.2.3. Modelsim 仿真

按照实验一所述，新建 modelsim 工程，将之前编写的 verilog 文件添加进工程，编译并开始仿真，观察图 2-40 所示的波形，可以看到流水灯速度正确设置，流水灯模式在周期性地改变。

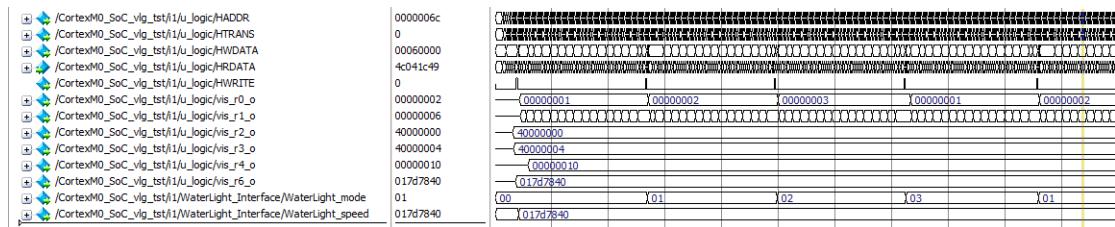


图 3-39 仿真波形图

### 3.2.4. 调试

由于调试是进行的单步调试，因此暂时不必对 R4 的值进行调整，当前 R4 的值决定了每 30 步左右流水灯模式改变一次。

将新编写的 verilog 文件添加进 vivado 工程中，将 led 灯添加进管脚约束，综合布局布线后生成比特流文件并下载进 FPGA 中。

```
##clk
set_property PACKAGE_PIN D4 [get_ports clk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##RSTn

set_property PACKAGE_PIN T9 [get_ports RSTn]

set_property IOSTANDARD LVCMOS33 [get_ports RSTn]

##DEBUGGER

set_property PACKAGE_PIN H14 [get_ports SWDIO]

set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]

set_property PACKAGE_PIN H12 [get_ports SWCLK]

set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]

##led

set_property PACKAGE_PIN P9 [get_ports {LED[0]}]

set_property PACKAGE_PIN R8 [get_ports {LED[1]}]

set_property PACKAGE_PIN R7 [get_ports {LED[2]}]

set_property PACKAGE_PIN T5 [get_ports {LED[3]}]

set_property PACKAGE_PIN N6 [get_ports {LED[4]}]

set_property PACKAGE_PIN T4 [get_ports {LED[5]}]

set_property PACKAGE_PIN T3 [get_ports {LED[6]}]

set_property PACKAGE_PIN T2 [get_ports {LED[7]}]
```

```
set_property PACKAGE_PIN R1 [get_ports LEDclk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports LEDclk]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
```

按照第一节讲述的方法连接调试器与 PC，打开 Keil 进行调试，调试结果如图 2-41 所示。

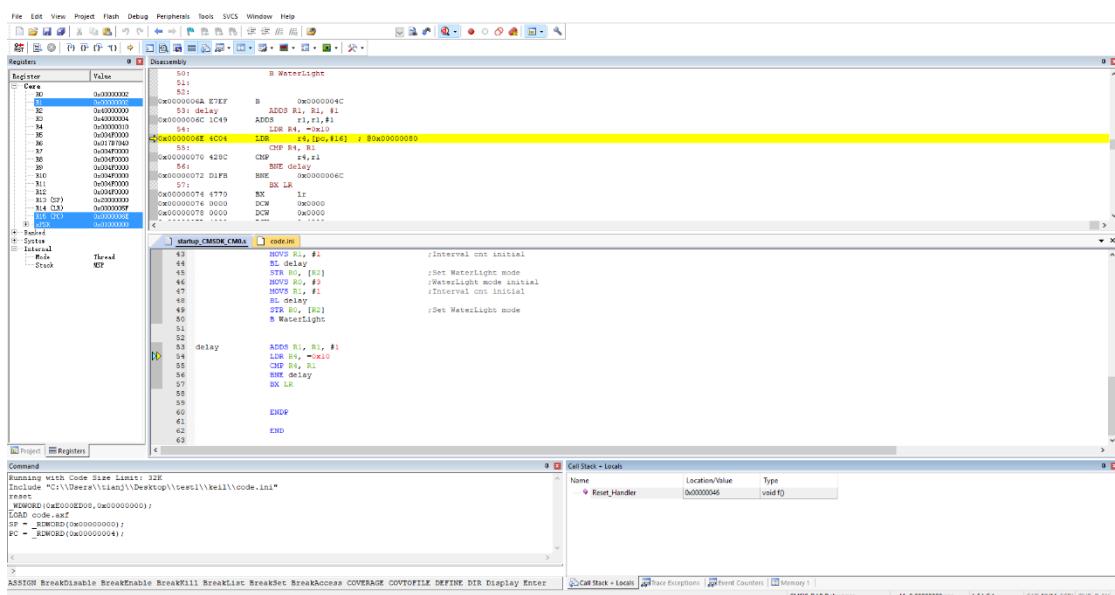


图 3-40 调试

观察 FPGA 上的 LED 灯，能看到其跟随程序的运行做出对应的反应，在 Keil 的调试界面也能看到各个寄存器正确的赋值，至此，本小节的实验完成了。

本小节完善了 SoC 的设计，补充了数据存储器及其总线接口，完成了第一个外设的设计。

### 3.2.5. 附加题一：蜂鸣器

文件 “Task2/rtl/buzzer.v” 为已经编写完成的蜂鸣器控制模块，类似于流水灯，其一共有四个模式，受输入 mode 寄存器控制：

- Mode[1:0]为 0，则静音
- Mode[1:0]为 1，则每一秒叫一次
- Mode[1:0]为 2，则每两秒叫一次
- Mode[1:0]为 3，则每四秒叫一次

请根据参照实验二流水灯模块，编写蜂鸣器总线接口，修改相应代码，编写汇编程序，实现蜂鸣器在三个模式中循环播放。

要求：蜂鸣器控制寄存器地址  $0x40010000 + \text{你的选课号} * 4$ （例如选课好号 20，则你需要实现的地址为  $0x40010050$ ）。

提示：完成此附加题需要进行如下的步骤：

- 参考 AHBlite\_WaterLight.v 模块编写 AHBlite\_Buzzer.v，实现总线接口
- 修改 AHBlite\_Interconnet.v，在其顶层添加一组外设端口 P4，并参考前三个端口完成 P4 端口信号的赋值，包括在 Decoder 以及 SlaveMUX 两个模块中添加额外的端口 P4
- 在顶层文件 CortexM0\_SoC.v 中添加蜂鸣器以及总线接口，并连接在总线扩展模

块的 P4 端口上

- 在顶层文件的端口添加蜂鸣器输出，并在管脚约束文件中添加对应的管脚约束

### 3.3. 实验三：中断与 C 语言编程

本小节以添加 UART 外设模块实验为基础，讲解 CPU 的中断处理以及使用 C 语言高效地编程。

根据 ARMv6-M 架构参考手册以及 Cortex-M0 用户手册，CPU 中断处理过程如下：

- CPU 接收到中断信号 (IRQ、NMI、Systick 等等)
- 将 R0,R1,R2,R3,R12,LR,PC,xPSR 寄存器入栈，如图 3-41
- 根据中断信号查找中断向量表（对应汇编启动代码中的\_Vector 段），跳转至中断处理函数，如图 3-42
- 中断处理函数执行完成后，利用链接寄存器返回，寄存器出栈，PC 跳转

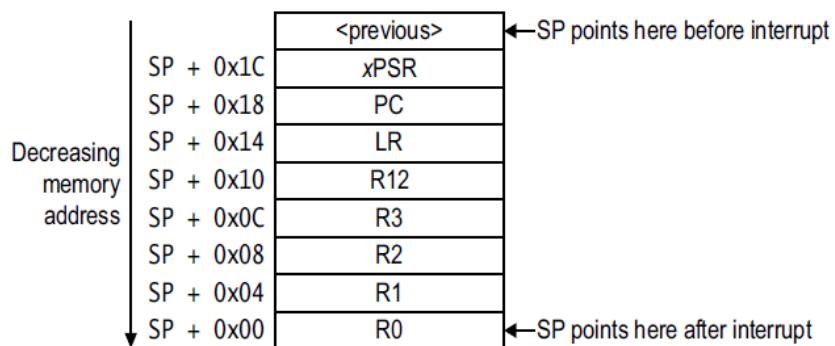


图 3-41 寄存器入栈

Exception number <sup>a</sup>	IRQ number <sup>a</sup>	Exception type	Priority	Vector address <sup>b</sup>	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable <sup>e</sup>	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable <sup>e</sup>	0x00000038	Asynchronous
15	-1	SysTick <sup>c</sup>	Configurable <sup>e</sup>	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above <sup>d</sup>	0 and above	IRQ	Configurable <sup>e</sup>	0x00000040 and above <sup>f</sup>	Asynchronous

图 3-42 中断向量表

实验三最终实现的 SoC 如图 3-43 所示。

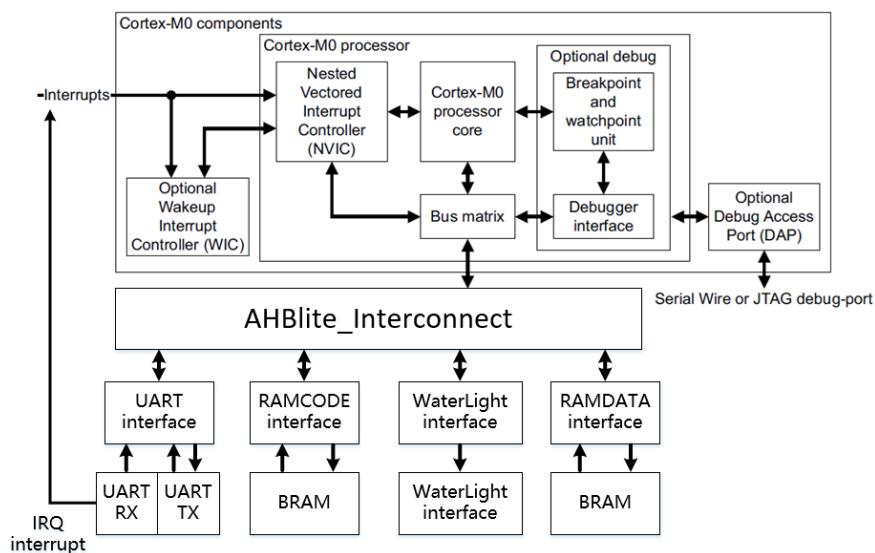


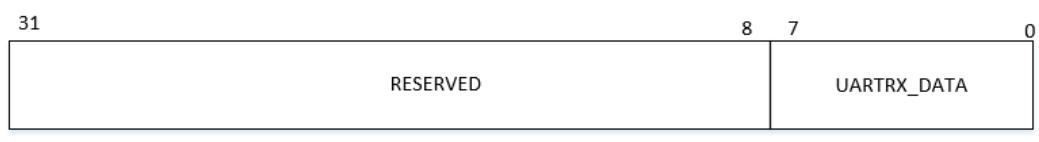
图 3-43 实验三实现的 SoC

### 3.3.1. 硬件部分

UART 外设主要由三部分组成：

- UARTRX: 用于接收数据，数据接收完成后向总线输入接收到的数据值并向 IRQ 中断产生一个时钟周期的脉冲。
- UARTTX: 用于发送数据，内部包含有一个缓冲器 (FIFO) 用以缓冲总线传来的数据，并通过总线提供 FIFO 满状态的状态寄存器，CPU 需要根据此寄存器判断是否可写。

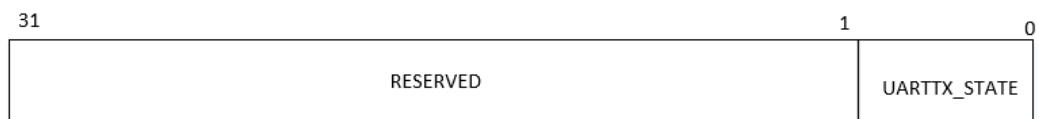
UART 对应着总线上三个寄存器，及三个 word 的地址空间，三个寄存器格式如图 3-44 所示。



(a) UARTRX数据寄存器



(b) UARTTX数据寄存器



(c) UARTTX状态寄存器

图 3-44 UART 寄存器格式

UART 具体代码提供在“/Task3/rtl/”文件夹下,请读者自行阅读, UART 结构如图 3-45 所示。

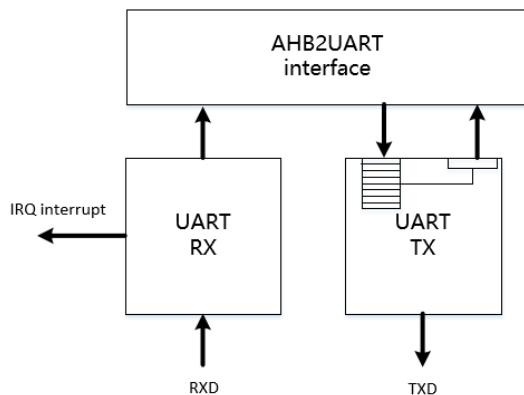


图 3-45 UART 模块结构图

将 UART 作为外设连接在总线外设 P3 端口,首先需要在 Decoder 中将 P3 的译码比较器添加进去,并在端口参数处将其使能;在顶层模块中已经完成例化 UART 各个模块以及总线接口,只需要将其总线接口与总线扩展模块 P3 端口连接。具体步骤与实验一、实验二相同,不再赘述。

最后,将 UART 连接上 IRQ 中断。

```
/*Connect the IRQ with UART*/  
  
assign IRQ = 32'b0;  
  
/***************************/
```

改为:

```
/*Connect the IRQ with UART*/  
  
assign IRQ = {31'b0,interrupt_UART};
```

```
*****
```

还得在实验二的 vivado 约束文件的基础上增加 UART 管脚的约束。

```
##UART

set_property PACKAGE_PIN F12 [get_ports TXD]
set_property IOSTANDARD LVCMOS33 [get_ports TXD]
set_property PACKAGE_PIN F13 [get_ports RXD]
set_property IOSTANDARD LVCMOS33 [get_ports RXD] ##clk
set_property PACKAGE_PIN D4 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports clk]

##RSTn
set_property PACKAGE_PIN T9 [get_ports RSTn]
set_property IOSTANDARD LVCMOS33 [get_ports RSTn]

##DEBUGGER
set_property PACKAGE_PIN H14 [get_ports SWDIO]
set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]
set_property PACKAGE_PIN H12 [get_ports SWCLK]
set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]
```

```
##led

set_property PACKAGE_PIN P9 [get_ports {LED[0]}]

set_property PACKAGE_PIN R8 [get_ports {LED[1]}]

set_property PACKAGE_PIN R7 [get_ports {LED[2]}]

set_property PACKAGE_PIN T5 [get_ports {LED[3]}]

set_property PACKAGE_PIN N6 [get_ports {LED[4]}]

set_property PACKAGE_PIN T4 [get_ports {LED[5]}]

set_property PACKAGE_PIN T3 [get_ports {LED[6]}]

set_property PACKAGE_PIN T2 [get_ports {LED[7]}]

set_property PACKAGE_PIN R1 [get_ports LEDclk]

set_property IOSTANDARD LVCMOS33 [get_ports LEDclk]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]

set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
```

### 3.3.2. 启动代码与 C 语言编程

我们需要根据 CMSIS 提供的启动代码重新完成自己的启动代码，具体代码见 “/Task3/keil/startup\_CMSDK\_CM0.s”。

与之前的汇编代码不同的是，我们在复位处理函数内调用了 \_mian 函数，此函数的作用是将堆栈初始化后跳转至 C 语言中的 mian 函数，而最后一段 \_user\_initial\_stackheap 则是初始化堆栈过程的一部分。初始化堆栈的具体过程由编译器提供，无需人为添加。

在中断处理的地方可以看到，当 UART 中断发生后，CPU 会根据 \_Vector 中的中断地址跳转到 UART 中断处理函数，在这个函数里面，首先人为地将寄存器入栈，然后跳转至 C 语言中的 UARThandle 函数，执行完成后寄存器出栈并返回。

然后，我们需要定义外设的地址，以及自己实现的函数，参考 CMSIS 编写自己头文件。具体代码见 “/Task3/keil/code\_def.h”。

```
#include <stdint.h>

.....
//UART DEF

typedef struct{

    volatile uint32_t UARTRX_DATA;

    volatile uint32_t UARTRX_STATE;

    volatile uint32_t UARTRX_DATA;

}UARTType;

#define UART_BASE 0x40000010
```

```
#define UART ((UARTType *)UART_BASE)
```

```
void SetWaterLightMode(int mode);
```

```
.....
```

第一行<stdint.h>头文件提供了结构体以及结构体运算符“->”的支持，高效地利用结构体定义外设地址，能够有效地减少代码量，节约存储空间。

下面以 UART 为例讲解结构体与基地址的使用。首先我们根据之前 UART 硬件部分设计，UART 在地址空间能有三个寄存器，分别为 UARTRX\_DATA、UARTTX\_STATE、UARTTX\_DATA，它们的地址分别为 0x40000010、0x40000014、0x40000018。三个寄存器在内存空间中是连续的三个字（word），因此在结构体中定义三个寄存器时需要按照它们地址的顺序依次定义，并且类型为 32bit 的 uint32\_t。之后再定义 UART 的基地址为 0x40000010。这样一来，当我们使用结构体中第一个元素时，它的地址则为基地址+0；第二个地址为基地址+4；第三个地址为基地址+8 依次类推。完全符合我们在硬件时定义的地址。

然后，我们需要完成函数的实现，具体见 “/Task3/keil/code\_def.c”

```
#include "code_def.h"

#include <string.h>

......

char ReadUARTState()

{

    char state;
```

```
state = UART -> UARTRX_STATE;

return(state);

}
```

```
char ReadUART()

{
    char data;

data = UART -> UARTRX_DATA;

return(data);

}
```

```
void WriteUART(char data)

{
    while(ReadUARTState());

UART -> UARTRX_DATA = data;

}
```

```
void UARTString(char *stri)

{
    int i;

for(i=0;i<strlen(stri);i++)

{
```

```
    WriteUART(stri[i]);  
}  
}  
  
void UARTHandle()  
{  
    int data;  
  
    data = ReadUART();  
  
    UARTString("Cortex-M0 : ");  
  
    WriteUART(data);  
  
    WriteUART('\n');  
}
```

在实现 UART 打印字符串时，我们并没有使用常见的重定向 printf、scanf 函数来实现，而是通过自己编写 UARTString 函数来实现。需要注意的是，在 WriteUART 函数里面，我们首先调用的时 ReadUARTState 函数，通过这个函数读取 UART 发送端口缓冲区是否为满，（满为 1，否则为 0），只有当其缓冲器未满时才进行写操作。

最后，编写主文件，具体在 “/Task3/keil/main.c” 。

```
#include "code_def.h"  
  
#include <string.h>  
  
#include <stdint.h>
```

```
.....  
  
int main()  
{  
    //interrupt initial  
    NVIC_CTRL_ADDR = 1;  
  
    //UART display  
    UARTString("Cortex-M0 Start up!\n");  
  
    .....  
  
}
```

在 main 函数中，首先对中断使能进行设置，相关知识请读者阅读相关手册文档，然后使用 UART 打印 “Cortex-M0 Start up” 字符串，然后进入流水灯控制，在每次改变流水灯模式后都用 UART 打印一串字符。

### 3. 3. 3. 调试与运行

打开 Keil 工程将编写好的文件添加至工程中，并在如下图所示的设置中取消勾选

“Don’t Search Standard Libraries”，然后编译，如图 3-46。

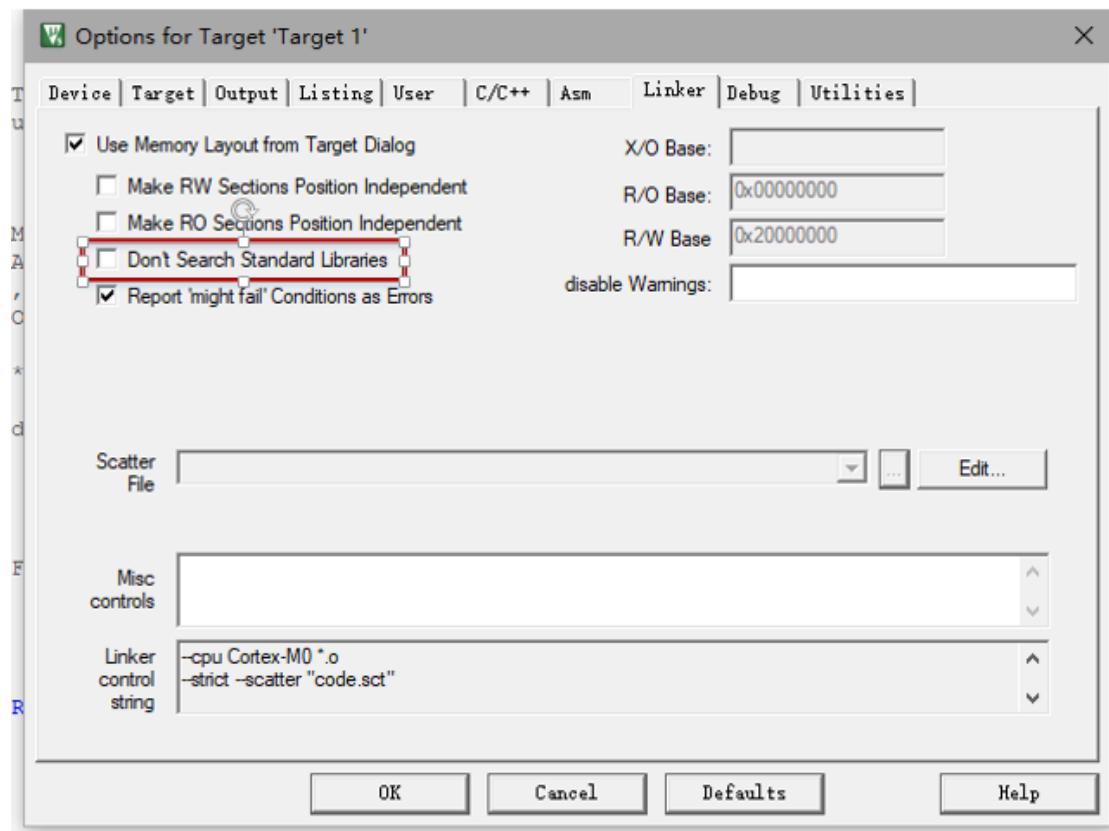


图 3-46 取消勾选

在 Vivado 中添加 UART 相关的 verilog 文件至工程中，在管脚约束中添加 RXD 与 TXD，综合布局布线并生成比特流文件，下载至 FPGA 中。

打开串口调试软件与 Keil，在 Keil 中点击开始调试，并开始连续运行，观察串口调试软件信息，如图 3-47。

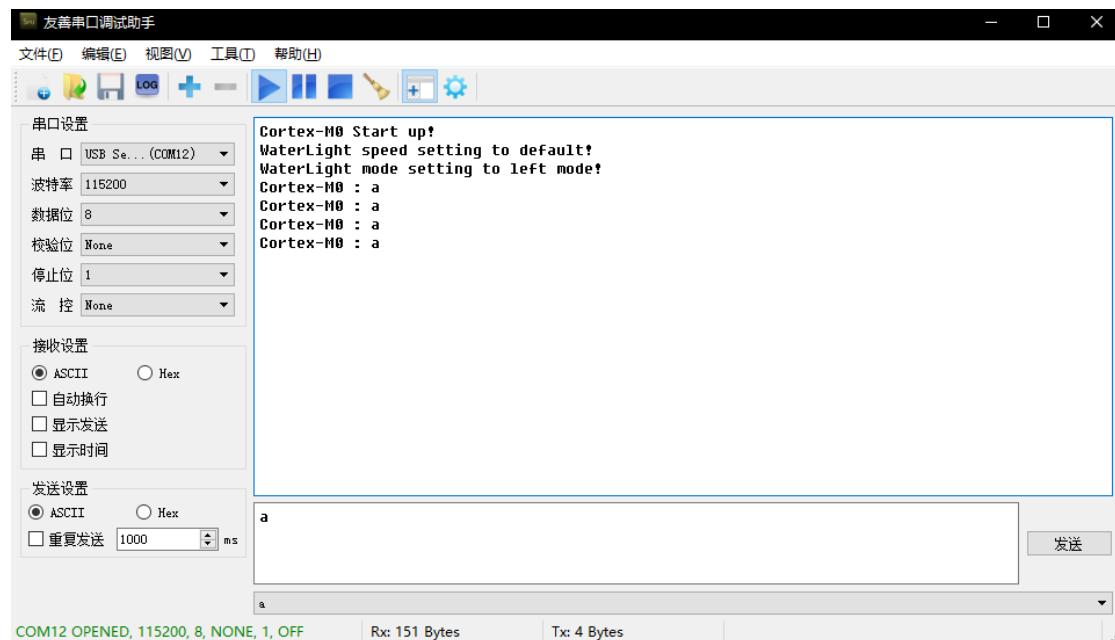


图 3-47 UART 接收信息

可以看到，CPU 启动后正常运行了程序，在通过串口调试软件向 CPU 发送字符后能成功显示字符串，说明 UART 中断设置成功。

### 3.3.4. 附加题二：UART 控制流水灯

通过修改 Keil 工程中的文件，实现通过 UART 接收字符控制流水灯模式：通过串口调试软件发送以 16 进制发送字符 0x01，流水灯进入左移模式；通过串口调试软件发送 16 进制字符 0x02，流水灯进入右移模式；通过串口调试软件发送 16 进制字符 0x03，流水灯进入闪烁模式；

提示：进入 code\_def.c 中利用 ReadUART()、SetWaterLightMode()两个函数修改 UARTHandle()函数。

### 3. 4. 实验四：初探虚拟仪器及 DAC

实验四为演示性实验，通过板载 DAC 实现简单的波形产生并通过虚拟仪器平台观察波形。其中 CortexM0\_SoC/Task4/source 目录下存放的是生成波形数据的 C 代码，其余文件夹和之前实验类似。

设计简单波形产生模块，通过连续读取 ROM 中的波形值输出到 DAC 实现波形的生成，将波形产生模块添加到 SoC 中步骤与之前实验相同：修改总线译码部分，添加总线接口“AHBlite\_WaveformGenerator.v”，在顶层模块中将外设模块连入。

**注意：需要修改“WaveformROM.v”模块中 readmemh 函数的地址。此地址在 source 文件夹下名为 sin\_Xilinx.hex，并且此地址必须为文件的绝对路径，其分隔符为‘/’而不是‘\’。**

具体操作步骤如下。

创建 Vivado 工程，添加 CortexM0\_SoC/Task4/rtl 目录下代码，添加管脚约束文件。

```
##clk

set_property PACKAGE_PIN D4 [get_ports clk]

set_property IOSTANDARD LVCMOS33 [get_ports clk]

##RSTn

set_property PACKAGE_PIN T9 [get_ports RSTn]

set_property IOSTANDARD LVCMOS33 [get_ports RSTn]

##DEBUGGER

set_property PACKAGE_PIN H14 [get_ports SWDIO]

set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]

set_property PACKAGE_PIN H12 [get_ports SWCLK]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]
set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]
##led
set_property PACKAGE_PIN P9 [get_ports {LED[0]}]
set_property PACKAGE_PIN R8 [get_ports {LED[1]}]
set_property PACKAGE_PIN R7 [get_ports {LED[2]}]
set_property PACKAGE_PIN T5 [get_ports {LED[3]}]
set_property PACKAGE_PIN N6 [get_ports {LED[4]}]
set_property PACKAGE_PIN T4 [get_ports {LED[5]}]
set_property PACKAGE_PIN T3 [get_ports {LED[6]}]
set_property PACKAGE_PIN T2 [get_ports {LED[7]}]
set_property PACKAGE_PIN R1 [get_ports LEDclk]
set_property IOSTANDARD LVCMOS33 [get_ports LEDclk]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[7]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[6]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[5]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[4]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {LED[0]}]
```

```
#dac

set_property IOSTANDARD LVCMOS33 [get_ports DACdata[0]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[1]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[2]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[3]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[4]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[5]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[6]]
set_property IOSTANDARD LVCMOS33 [get_ports DACdata[7]]

set_property IOSTANDARD LVCMOS33 [get_ports WRn]
set_property IOSTANDARD LVCMOS33 [get_ports CSn]
set_property IOSTANDARD LVCMOS33 [get_ports DACsel]

set_property PACKAGE_PIN G1 [get_ports DACdata[0]]
set_property PACKAGE_PIN G2 [get_ports DACdata[1]]
set_property PACKAGE_PIN F2 [get_ports DACdata[2]]
set_property PACKAGE_PIN E1 [get_ports DACdata[3]]
set_property PACKAGE_PIN L3 [get_ports DACdata[4]]
set_property PACKAGE_PIN K1 [get_ports DACdata[5]]
set_property PACKAGE_PIN K2 [get_ports DACdata[6]]
```

```
set_property PACKAGE_PIN J1 [get_ports DACdata[7]]
```

```
set_property PACKAGE_PIN H1 [get_ports WRn]
```

```
set_property PACKAGE_PIN H2 [get_ports CSn]
```

```
set_property PACKAGE_PIN J3 [get_ports DACsel]
```

注意，实验三中 keil 调试时取消了对图 3-48“Don’t Search Standard Libraries”的勾选，在实验四中需要再次勾选这个选项。

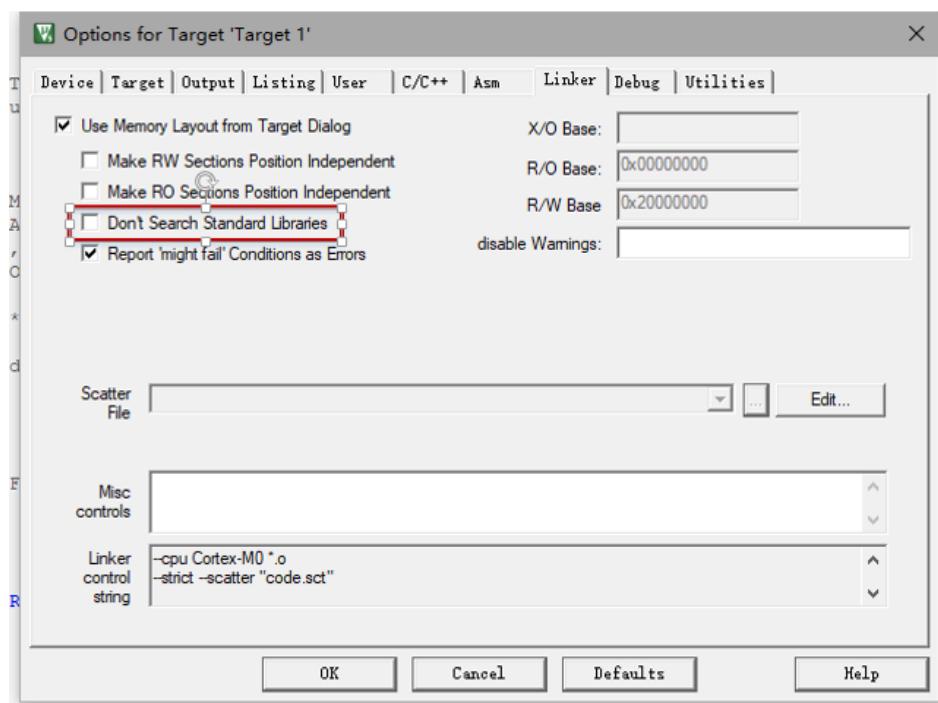


图 3-48 keil 工程设置

综合上板后通过 keil 调试，向波形产生器外设发送控制信号，并在通过虚拟平台观察波形。

图 3-49 为波形图，当 cpu 发送的控制信号为 0 时，不产生波形，控制信号为 1 时，有“正弦”波形产生。由于 DAC 精度和波形产生方法原因，波形美观度有所欠缺。

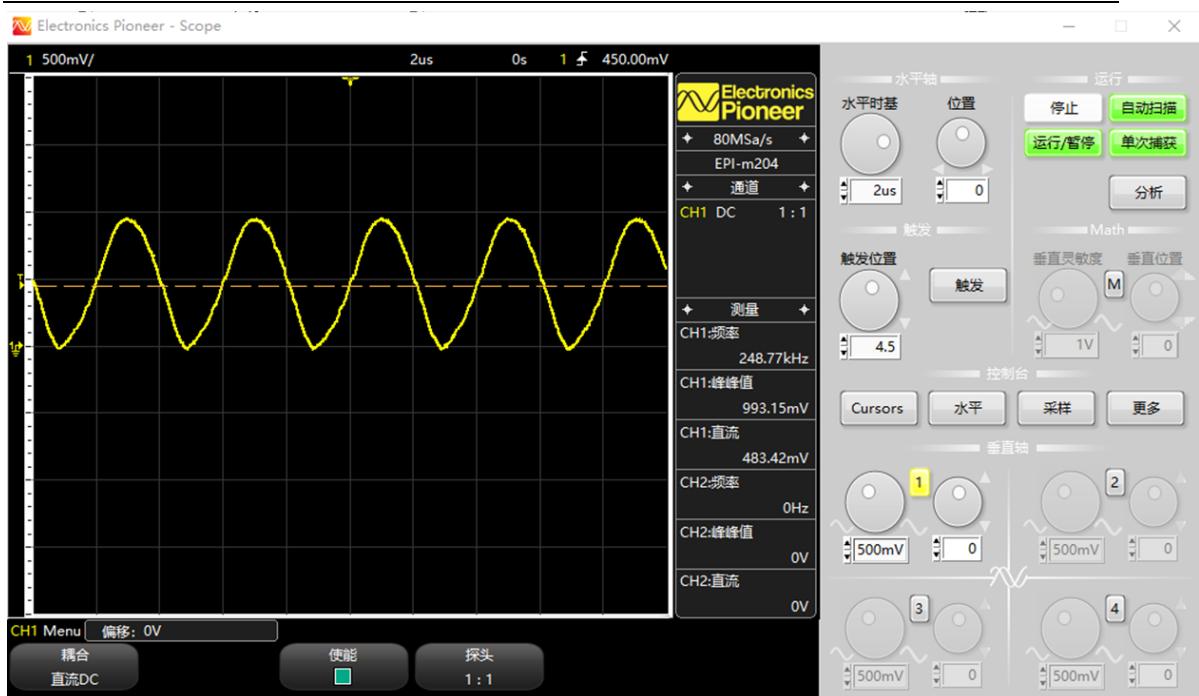


图 3-49 控制有效时输出波形图

### 3.5. 实验五：点亮 LCD 显示屏

#### 3.5.1. 显示屏简介

本次实验使用 2.4 寸 TFT LCD 液晶显示模块，驱动芯片为 ILI9341，该芯片的数据手册可以在 CortexM0\_SoC/docs/LCD\_DS 下找到。

LCD 屏的读写时序如图 3-50 和图 3-51 所示，在软件模拟时序时作为参考。

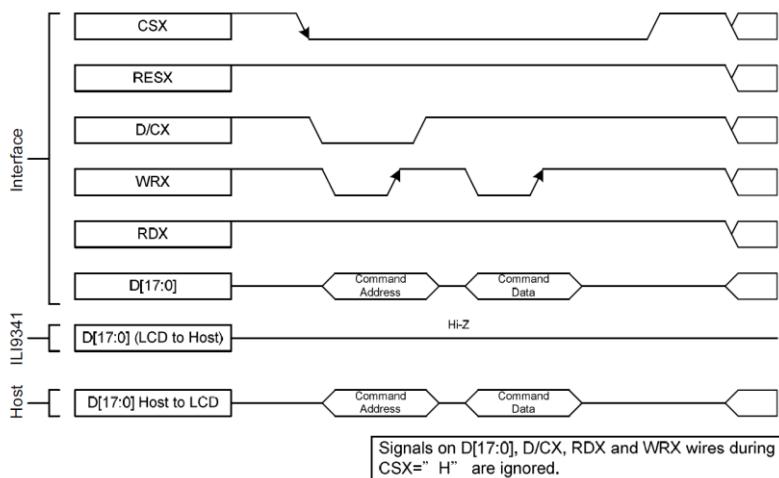


图 3-50 摄像头写时序

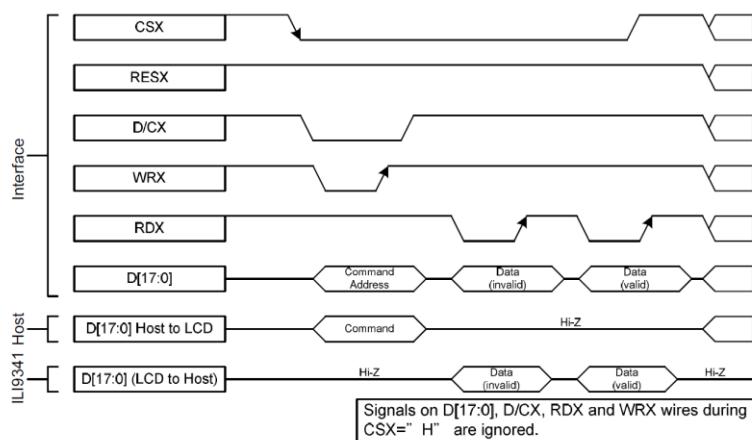


图 3-51 摄像头读时序

在本次实验中，将通过软件代码模拟读/写时序，简化硬件设计，增强设计灵活性。

### 3.5.2. 硬件部分

由于实验使用软件模拟读/写时序，所以外设的接口其实就是 GPIO 接口，相对来说较为简单。设计 GPIO 接口模块步骤和之前添加外设步骤类似。

第一步，在“AHBlite\_Decoder.v”中 LCD (Port 4) 端口参数将其使能。

```
/*LCD enable parameter*/
parameter Port4_en = 0
```

```
*****
```

改为

```
/*LCD enable parameter*/  
  
parameter Port4_en = 1  
  
*****
```

第二步，在 memory map 中我们设置 LCD 的总线编码为 0x40050000-0x40050100，

所以修改在译码部分插入 LCD 的译码器代码。

```
//0X40050000 LCD  
  
/*Insert LCD decoder code there*/  
  
*****
```

改为

```
//0X40050000 LCD  
  
/*Insert LCD decoder code there*/  
  
assign P4_HSEL = (HADDR[31:16] == 16'h4005) ? Port4_en : 1'b0;  
  
*****
```

第三步，在顶层模块中将 LCD 接口和总线扩展模块连接。

```
AHBlite_LCD_LCD_Interface(  
  
/* Connect to Interconnect Port 4 */
```

```
.HCLK           (clk),  
  
.HRESETn       (cpuresetn),  
  
.HSEL          /* Port4 */,  
  
.HADDR         /* Port4 */,  
  
.HPROT         /* Port4 */,  
  
.HSIZE         /* Port4 */,  
  
.HTRANS        /* Port4 */,  
  
.HWDATA        /* Port4 */,  
  
.HWRITE        /* Port4 */,  
  
.HRDATA        /* Port4 */,  
  
.HREADY        /* Port4 */,  
  
.HREADYOUT    /* Port4 */,  
  
.HRESP         /* Port4 */,  
  
.LCD_CS        (LCD_CS),  
  
.LCD_RS        (LCD_RS),  
  
.LCD_WR        (LCD_WR),  
  
.LCD_RD        (LCD_RD),  
  
.LCD_RST       (LCD_RST),  
  
.LCD_DATA      (LCD_DATA),  
  
.LCD_BL_CTR   (LCD_BL_CTR)  
  
/**************************/  
);
```

改为

```
AHBlite_LCD LCD_Interface(  
    /* Connect to Interconnect Port 4 */  
  
    .HCLK          (clk),  
    .HRESETn      (cpuresetn),  
    .HSEL          (HSEL_P4),  
    .HADDR         (HADDR_P4),  
    .HPROT         (HPROT_P4),  
    .HSIZE         (HSIZE_P4),  
    .HTRANS        (HTRANS_P4),  
    .HWDATA        (HWDATA_P4),  
    .HWRITE        (HWRITE_P4),  
    .HRDATA        (HRDATA_P4),  
    .HREADY        (HREADY_P4),  
    .HREADYOUT    (HREADYOUT_P4),  
    .HRESP         (HRESP_P4),  
    .LCD_CS       (LCD_CS),  
    .LCD_RS       (LCD_RS),  
    .LCD_WR       (LCD_WR),  
    .LCD_RD       (LCD_RD),
```

```
.LCD_RST          (LCD_RST),  
.LCD_DATA         (LCD_DATA),  
.LCD_BL_CTR       (LCD_BL_CTR)  
/******************/  
);
```

最后在顶层模块中声明 LCD 相关的输入信号。

```
module CortexM0_SoC (  
    input      wire      clk,  
    input      wire      RSTn,  
    inout     wire      SWDIO,  
    input      wire      SWCLK,  
    // LCD  
    // insert LCD ports  
    output     wire      TXD,  
    input      wire      RXD  
);
```

改为

```
module CortexM0_SoC (  
    input      wire      clk,
```

```
input      wire      RSTn,  
  
inout     wire      SWDIO,  
  
input      wire      SWCLK,  
  
// LCD  
  
// insert LCD ports  
  
output    wire      LCD_CS,  
  
output    wire      LCD_RS,  
  
output    wire      LCD_WR,  
  
output    wire      LCD_RD,  
  
output    wire      LCD_RST,  
  
output    wire [15:0] LCD_DATA,  
  
output    wire      LCD_BL_CTR,  
  
output    wire      TXD,  
  
input     wire      RXD  
);
```

在 Vivado 中新建工程，添加设计文件，在之前实验的管脚约束文件基础上添加 LCD。

```
##UART  
  
set_property PACKAGE_PIN F13 [get_ports RXD]  
  
set_property IOSTANDARD LVCMOS33 [get_ports RXD]  
  
set_property PACKAGE_PIN F12 [get_ports TXD]  
  
set_property IOSTANDARD LVCMOS33 [get_ports TXD]
```

```
##clk

set_property PACKAGE_PIN D4 [get_ports clk]

set_property IOSTANDARD LVCMOS33 [get_ports clk]

##RSTn

set_property PACKAGE_PIN T9 [get_ports RSTn]

set_property IOSTANDARD LVCMOS33 [get_ports RSTn]

##DEBUGGER

set_property PACKAGE_PIN H14 [get_ports SWDIO]

set_property IOSTANDARD LVCMOS33 [get_ports SWDIO]

set_property PACKAGE_PIN H12 [get_ports SWCLK]

set_property IOSTANDARD LVCMOS33 [get_ports SWCLK]

set_property CLOCK_DEDICATED_ROUTE FALSE [get_nets SWCLK]

##lcd

set_property PACKAGE_PIN A3 [get_ports LCD_CS]

set_property PACKAGE_PIN D3 [get_ports LCD_RS]

set_property PACKAGE_PIN B4 [get_ports LCD_WR]

set_property PACKAGE_PIN A4 [get_ports LCD_RD]

set_property PACKAGE_PIN B5 [get_ports LCD_RST]

set_property PACKAGE_PIN C8 [get_ports LCD_BL_CTR]

set_property PACKAGE_PIN A5 [get_ports LCD_DATA[0]]

set_property PACKAGE_PIN B6 [get_ports LCD_DATA[1]]
```

```
set_property PACKAGE_PIN B7 [get_ports LCD_DATA[2]]  
  
set_property PACKAGE_PIN A7 [get_ports LCD_DATA[3]]  
  
set_property PACKAGE_PIN C4 [get_ports LCD_DATA[4]]  
  
set_property PACKAGE_PIN E5 [get_ports LCD_DATA[5]]  
  
set_property PACKAGE_PIN D5 [get_ports LCD_DATA[6]]  
  
set_property PACKAGE_PIN D6 [get_ports LCD_DATA[7]]  
  
set_property PACKAGE_PIN C6 [get_ports LCD_DATA[8]]  
  
set_property PACKAGE_PIN E6 [get_ports LCD_DATA[9]]  
  
set_property PACKAGE_PIN C7 [get_ports LCD_DATA[10]]  
  
set_property PACKAGE_PIN D8 [get_ports LCD_DATA[11]]  
  
set_property PACKAGE_PIN D9 [get_ports LCD_DATA[12]]  
  
set_property PACKAGE_PIN C9 [get_ports LCD_DATA[13]]  
  
set_property PACKAGE_PIN D10 [get_ports LCD_DATA[14]]  
  
set_property PACKAGE_PIN A8 [get_ports LCD_DATA[15]]  
  
  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_CS]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_RS]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_WR]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_RD]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_RST]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_BL_CTR]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[0]]
```

```
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[1]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[2]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[3]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[4]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[5]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[6]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[7]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[8]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[9]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[10]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[11]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[12]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[13]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[14]]  
  
set_property IOSTANDARD LVCMOS33 [get_ports LCD_DATA[15]]
```

### 3. 5. 3. 软件部分

在实验三中我们已经学习了如何通过 C 语言编程访问外设，类似的，在“code\_def.h”中，定义 LCD 相关变量结构体。

```
//LCD DEF  
  
typedef struct {  
  
    volatile uint32_t LCD_CS; // 0x40050000
```

```

volatile uint32_t LCD_RS; // 0x40050004

volatile uint32_t LCD_WR; // 0x40050008

volatile uint32_t LCD_RD; // 0x4005000C

volatile uint32_t LCD_RST; // 0x40050010

volatile uint32_t LCD_BL_CTR; // 0x40050014

volatile uint32_t LCD_DATA[16]; // 0x40050018-0x40050054

}LCDType;

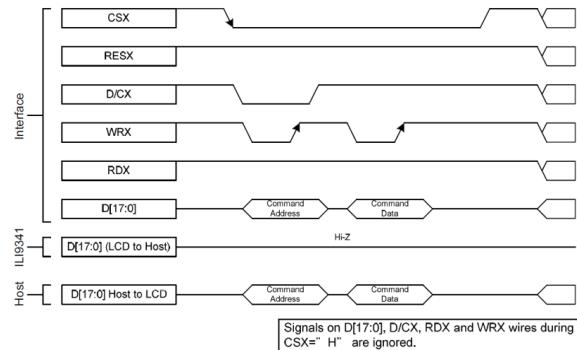
#define LCD_BASE 0x40050000

#define LCD ((LCDType *)LCD_BASE)

```

这样便可以通过 `LCD->LCD_xxx` 操作 LCD 相关的信号。

下图是图 3-51 屏幕写时序图，用 `LCD_WR_REG` 和 `LCD_WR_DATAx` 函数模拟时序。



```

// WRITE REG FUNCTION

void LCD_WR_REG( uint16_t data ) {

    LCD_RS_CLR;

    LCD_CS_CLR;

```

```
delay(500);

MakeData( data );

LCD_WR_CLR;

delay(500);

LCD_WR_SET;

LCD_CS_SET;

}

// WRITE DATA

void LCD_WR_DATA( uint16_t data ) {

LCD_RS_SET;

LCD_CS_CLR;

delay(500);

MakeData( data );

LCD_WR_CLR;

delay(500);

LCD_WR_SET;

LCD_CS_SET;

}
```

可以看到,在写寄存器的时候需要先将 CSX 和 D/CX(在 CortexM0\_SoC 中为 LCD\_RS, 选择写寄存器/写数据) 拉低, 经过 delay 后, 传送数据, WR 拉低, 写完成后再经过一个 delay 将 WR 和 CS 拉高。

与写寄存器不同的是，写数据时先将 D/CX 拉高，其余过程都一样。

注：LCD\_XX\_SET 和 LCD\_XX\_CLR 分别将信号设为 1 和 0，在“code\_def.h”中通过宏定义实现。LCD 相关的函数均是基于这两个函数实现的。

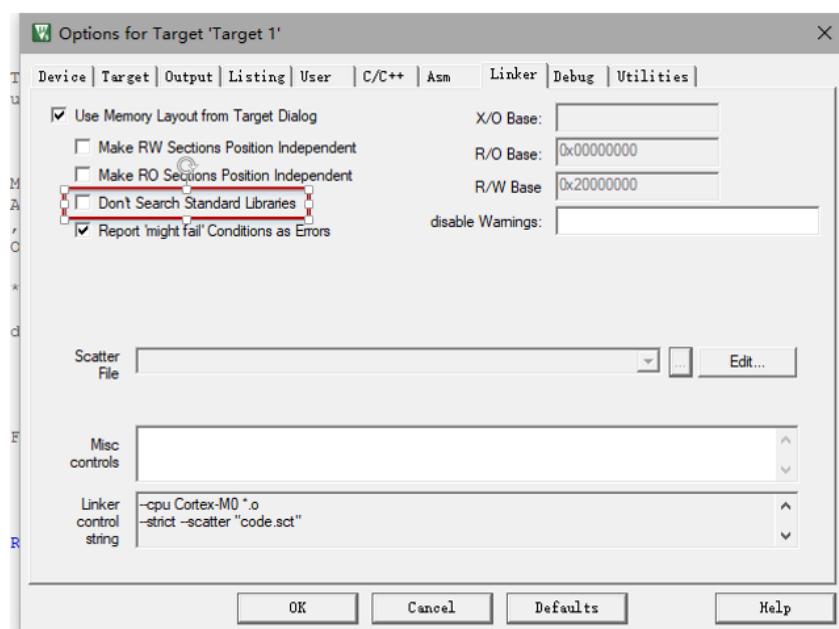
在 CortexM0\_SoC/Task5/keil/main.c 的 main 函数中，首先执行系统初始化函数 SYSInit()，目的是使能中断和初始化 systick。systick 是 Cortex-M 处理器中的一个简单定时器，主要用于延时，比如 main()中的 delay 函数，便是用 Systick 实现。在本实验中不做过多介绍，关于 Systick 的详细介绍可以参考 Joseph Yiu 所著的《The Definitive Guide to ARM Cortex-M0 and Cortex-M0+ Processors》。

在系统初始化以后，先将 LCD 的 Reset 信号拉低，保持一段时间后拉高，对 LCD 进行硬复位。在 while 循环中，通过两层 for 循环实现坐标的扫描，每次通过调用 LCD\_Fill 函数对一块方形区域进行颜色填充。

```
int main()
{
    uint16_t x, y;
    uint8_t dx, dy;
    SYSInit();
    LCD->LCD_RST = 0;
    delay(5000000);
    LCD->LCD_RST = 1;
    x = y = 0;
    dx = dy = 20;
    LCD_Init();
```

```
while (1) {  
  
    x = 0; y = 0;  
  
    for (x = 0; x < 240; x += dx) {  
  
        for (y = 0; y < 320; y += dy) {  
  
            LCD_Fill(x, y, x + dx, y + dy, RED);  
  
            delay(50000000);  
  
            LCD_Clear(WHITE);  
  
        }  
  
    }  
  
}  
  
}
```

在 keil 中进行工程设置时和之前实验一样，注意取消勾选 “Don't Search Standard Library”。



最终效果红色方框不停地移动，图 3-52 和图 3-53 所示红色方框移动到两个不同位置。

(网状波纹属于手机拍摄问题，与显示无关)

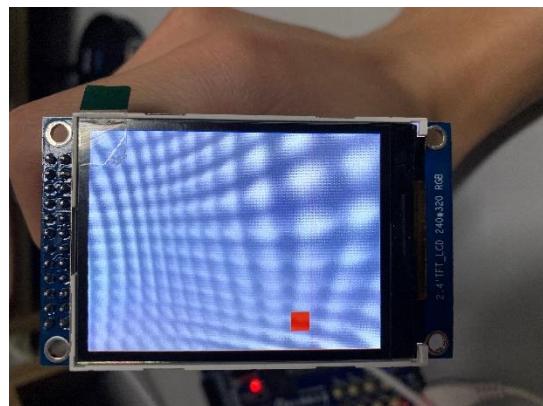


图 3-52 第一次拍摄红色方框位置

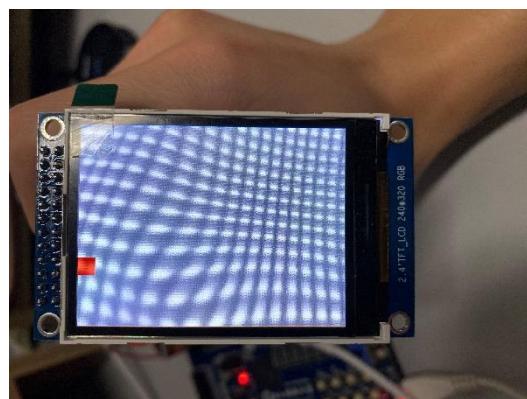


图 3-53 第二次拍摄红色方框位置

### 3.6. 本章小结

至此 SoC 搭建完成，本章以三个实验的方式，以 Cortex-M0 的裸核一步一步搭建出一个完整的 SoC，讲解了 CPU 的启动过程、中断处理过程等。并通过 LCD 显示屏实验，说明丰富 SoC 的一些基本方法及调试技巧，有助于熟练掌握如何在 SoC 上拓展更多需要的外设，如摄像头等。在 DAC 波形产生演示性实验中进一步熟悉了向 SoC 添加外设的步骤同时也学习了虚拟平台的使用。

## 第四章 终极测试

利用 LCD\_Fill 函数可以在 LCD 屏幕上的一块方形区域内填充参数指定的颜色，如实验四中所述。现在添加四个按键，分别对应上下左右，不按下按键时，屏幕上显示一个红色小方块，当按下对应方向的按键时，红色小方块朝对应方向移动。移动长度，方块边长根据自己喜好设置，便于观察即可。

为了实现这种效果，有 2 种大概的思路，一是设计一个键盘接口，通过总线去读键盘的信号值，进行分支判断进行上下左右或静止不动操作；二是将键盘信号连到 CPU 的中断信号，并设计中断处理函数，对按下不同键产生的不同中断进行中断处理，实现上下左右或静止不动的功能。

## 第五章 常见问题汇总

发生错误或者遇到问题首先不要慌，授人以鱼不如授人以渔，处理问题的大致流程

如下：

- 第一步：百度
- 第二步：谷歌
- 第三步：问同学
- 第四步：问助教
- 若 QUARTUS 或者 VIVADO 报错，右键错误代码点击 HELP

下面总结实验中可能遇到的常见问题。

### 5.1. 软件包安装问题

- 软件安装包是否在中文路径或者非法路径（非法字符）下
- 软件安装地址是否在中文路径或者非法路径（非法字符）下
- 是否是杀毒软件在装怪
- 软件安装地址空间是否充足
- 安装 CMSDK\_CM0 包的时候发生错误，首先新建工程，看看处理器列表中有没有 CMSDK\_CM0，若存在就不需要再安装扩展包

### 5.2. Verilog 或汇编编译错误（Syntax Error）

- 若按照指导书完成的代码编写，则主要检查是否有中文标点
- Keil 汇编代码编译错误 error:A1163E，检查汇编代码格式是否正确，标签(Label)顶格

写，指令进位写

- Keil 命令无法执行 Error: CreateProcess failed，添加命令时是否正确添加，是否存在中文标点，横线前是否有空格
- 若能正确编译生成.axf 文件而不能生成.hex 或者.txt 文件，检查 keil 工程与.axf 文件是否在同一文件夹下

### 5. 3. Modelsim 仿真错误

- 若使用 Modelsim SE 版本，在开始仿真时不能勾选 Enable optimization，否则会导致无电路模块，如图 5-1
- 仿真时 HRDATA 从复位释放后便为 Z 或者 X，检查 Block\_RAM.v 中 readmemh 函数引用的文件地址是否正确，路径是否存在中文或者非法字符，Keil 是否成功编译生成.axf 以及.hex 文件
- 仿真时 HRDATA 复位后先正常后变为 Z 或者 X，参考第二章，观察 IPSR 以及 LOCKUP 信号，分析错误原因