
蜂鸟 E203 快速上手说明

Content

1	PREFACE	3
1.1	REVISION HISTORY	3
2	运行 VERILOG 仿真测试	4
2.1	E203 开源项目的代码层次结构.....	4
2.2	E203 开源项目的测试用例（SELF-CHECK TESTCASE）	5
2.2.1	riscv-tests 自测试用例.....	5
2.2.2	编译 ISA 自测试用例.....	6
2.3	E203 开源项目的测试平台（TESTBENCH）	8
2.4	在 VERILOG TESTBENCH 中运行测试用例	10
3	蜂鸟 E203 开源 SOC	13
4	搭建 FPGA 原型平台	18
4.1	FPGA 开发板和项目介绍.....	18
4.2	生成 MCS 文件烧写 FPGA	20
4.3	JTAG 调试器.....	26
4.4	FPGA 原型平台 DIY 总结	29
5	运行和调试软件示例	30
5.1	HBIRD-E-SDK 简介	30
5.1.1	HBird-E-SDK 代码结构.....	30
5.2	使用 HBIRD-E-SDK 开发和运行示例程序.....	31
5.3	使用 GDB 和 OPENOCD 调试示例程序	34
6	运行更多示例程序和 BENCHMARKS	35
7	移植和运行 FREERTOS.....	35
8	WINDOWS IDE 开发工具	36

1 Preface

1.1 Revision History

Date	Version	Author	Change Summary
Oct 20,2018	0.1	Bob Hu	Initial version

注意：

- 本文档对蜂鸟 E203 处理器内核以及 RISC-V 指令集架构的介绍尚不够详细，在中文书籍《手把手教你设计 CPU：RISC-V 处理器》和《RISC-V 架构与嵌入式开发快速入门》中对其进行深入浅出地系统讲解。感兴趣的用户可以自行搜索书籍。
- 本文档对 SoC 的各外设的介绍尚不够详细，在中文书籍《RISC-V 架构与嵌入式开发入门指南》中进行深入浅出的系统讲解。感兴趣的用户可以自行搜索此书。

2 运行 Verilog 仿真测试

本章将介绍开源 E203 项目如何运行 Verilog 仿真测试平台。

2.1 E203 开源项目的代码层次结构

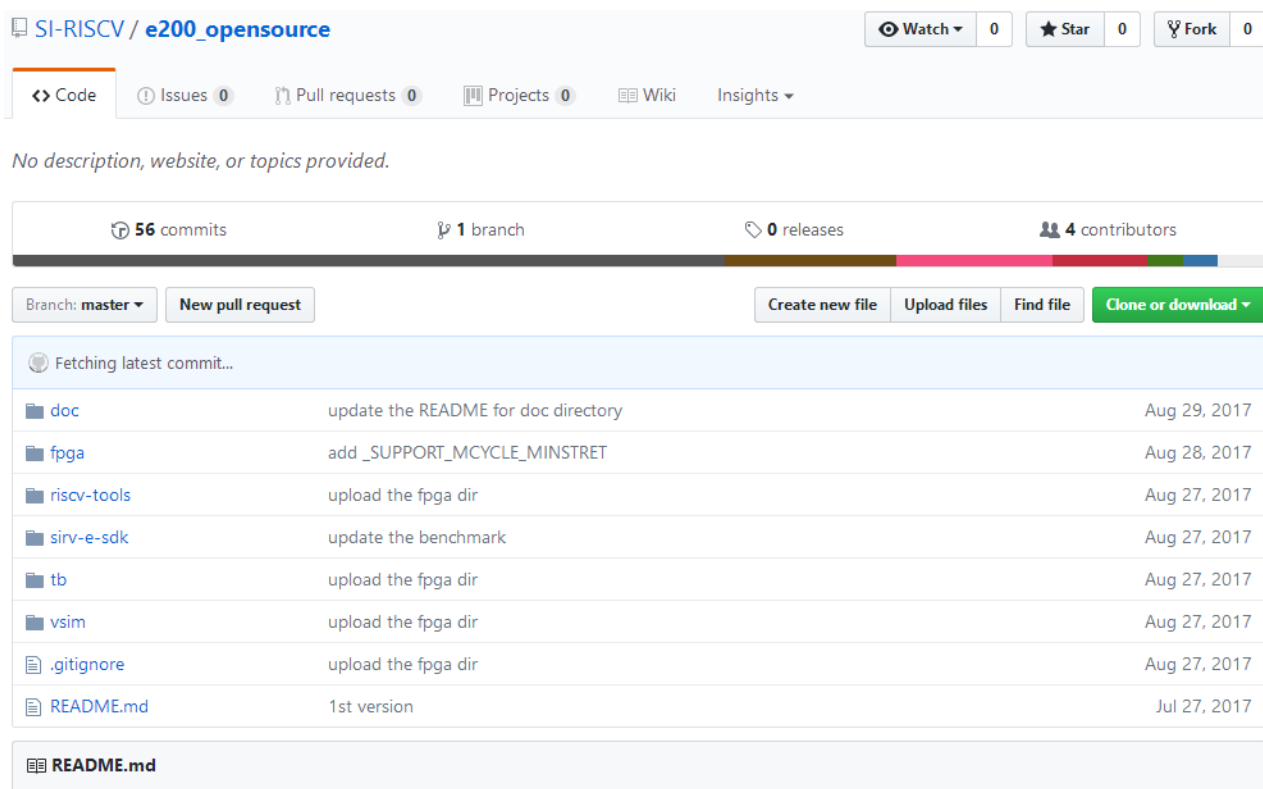


图 2-1 蜂鸟 E203 开源项目 Github 网址

蜂鸟 E203 开源项目的平台托管于著名网站 Github。E203 开源的项目网址为 https://github.com/SI-RISCV/e200_opensource 如图 2-1 所示。

在该网址的 e200_opensource 目录下，文件的层次结构如下所示。

```
e200_opensource
|----rtl                // 存放 RTL 的目录
|      |----e203        // E203 核和 SoC 的 RTL 目录
|      |      |----general // 存放一些公用的通用 RTL 代码
|      |      |----core   // 存放 E203 Core 的 RTL 代码
|      |      |----fab    // 存放总线 bus fabric 的 RTL 代码
```

```

|-----subsys      // 存放完整子系统顶层的 RTL 代码
|-----mems        // 存放 memory 模块的 RTL 代码
|-----perips      // 存放外设 peripherals 模块的 RTL 代码
|-----debug       // 存放 debug 相关模块的 RTL 代码
|-----fpga        // 存放 FPGA 实现的 RTL 代码
|-----tb          // 存放 Verilog TestBench (测试平台) 的目录
|-----tb_top.v    // 简单地 Verilog TestBench 顶层文件
|-----vsim        // 运行仿真的目录
|-----bin         // 存放脚本的文件夹子
|-----Makefile    // 运行的 Makefile
|-----run         // 运行目录
|-----fpga        // 存放 FPGA 项目和脚本的目录
|-----riscv-tools // 存放所需 riscv-tools 的目录
|-----riscv-fesvr // 用于编译指令模拟器 Spike 的源代码
|-----riscv-isa-sim // 用于编译指令模拟器 Spike 的源代码
|-----riscv-tests // 存放一些测试用例的目录
|-----doc         // 保存文档的目录
|-----README.md   // 说明文件

```

rtl 目录下包含了大量的源代码，主要为 E203 Core 的 Verilog RTL 源代码，和配套的 SoC 组件文件。E203 内核的 RTL 详细请参见《手把手教你设计 CPU——RISC-V 处理器篇》，配套的 SoC 的信息和代码分析请参见下一章。

2.2 E203 开源项目的测试用例（Self-Check TestCase）

上节中所述的 riscv-tools 与 RISC-V 架构正式维护的 riscv-tools 项目同名（Github 网址 <https://github.com/riscv/riscv-tools>）。正式维护的 riscv/riscv-tools 目录下包括了所有的 RISC-V 所需的软件工具，其中主要是 GNU ToolChain（源文件超过 1G，因此下载需要相当长的时间）。

而 e200_opensource 目录（https://github.com/SI-RISCV/e200_opensource）下的 riscv-tools 目录仅仅包含编译 Spike 所需的源代码和 riscv-tests，我们放置该目录于此是因为正式维护的 riscv/riscv-tools 在不断的更新，而 e200_opensource 下的 riscv-tools 仅需用于支持运行自测试用例（Self-Check TestCase），因此无需使用最新版本，并且进行了适当的修改（譬如，在 riscv-tests 里面添加了更多的测试用例和生成更多的 log 文件），同时还去除了 GNU ToolChain，使得文件夹的大小很小，方便用户快速的下载使用。

2.2.1 riscv-tests 自测试用例

所谓自测试用例（Self-Check Testcase）是一种具备自我检测运行成功还是失败的测试程序。riscv-test 是由 RISC-V 架构开发者维护的项目，这里面有一些测试处理器是否符合指令集架构定义的测试程序（<https://github.com/riscv/riscv-tests/tree/master/isa>），这些测试程序均由汇编语言编写。

这些汇编测试程序里面用某些宏定义组织成程序点，测试指令集架构中定义的指令，如图 2-2 所示，测试 add 指令（源代码文件为 isa/rv64ui/add.S），通过让 add 指令执行两个数据的相加（譬如 0x00000003 和 0x00000007），设定它期望的结果（譬如 0x0000000a）。然后使用比较指令加以判断，假设 add 指令的

执行结果的确与期望的结果相等则程序继续执行，假设与期望的结果不想等则程序直接使用 `jump` 指令跳到 `TEST_FAIL` 地址。假设所有的测试点都通过了，则程序一直执行到 `TEST_PASS` 地址。

```
RVTEST_CODE_BEGIN
#-----
# Arithmetic tests
#-----

TEST_RR_OP( 2, add, 0x00000000, 0x00000000, 0x00000000 );
TEST_RR_OP( 3, add, 0x00000002, 0x00000001, 0x00000001 );
TEST_RR_OP( 4, add, 0x0000000a, 0x00000003, 0x00000007 );

TEST_RR_OP( 5, add, 0xffffffffffff8000, 0x0000000000000000, 0xffffffffffff8000 );
TEST_RR_OP( 6, add, 0xffffffff80000000, 0xffffffff80000000, 0x00000000 );
TEST_RR_OP( 7, add, 0xffffffff7fff8000, 0xffffffff80000000, 0xffffffffffff8000 );

TEST_RR_OP( 8, add, 0x0000000000007fff, 0x0000000000000000, 0x0000000000007fff );
TEST_RR_OP( 9, add, 0x000000007fffffff, 0x000000007fffffff, 0x0000000000000000 );
TEST_RR_OP( 10, add, 0x0000000080007ffe, 0x000000007fffffff, 0x0000000000007fff );

TEST_RR_OP( 11, add, 0xffffffff80007fff, 0xffffffff80000000, 0x0000000000007fff );
TEST_RR_OP( 12, add, 0x000000007fff7fff, 0x000000007fffffff, 0xffffffffffff8000 );

TEST_RR_OP( 13, add, 0xffffffffffffffff, 0x0000000000000000, 0xffffffffffffffff );
TEST_RR_OP( 14, add, 0x0000000000000000, 0xffffffffffffffff, 0x0000000000000001 );
TEST_RR_OP( 15, add, 0xffffffffffffffe, 0xffffffffffffffff, 0xffffffffffffffff );

TEST_RR_OP( 16, add, 0x0000000080000000, 0x0000000000000001, 0x000000007fffffff );

#-----
# Source/Destination tests
#-----

TEST_RR_SRC1_EQ_DEST( 17, add, 24, 13, 11 );
TEST_RR_SRC2_EQ_DEST( 18, add, 25, 14, 11 );
TEST_RR_SRC12_EQ_DEST( 19, add, 26, 13 );
```

图 2-2 riscv-tests 测试用例 add.S 片段

在 `TEST_PASS` 的地址，程序将设置 `x3` 寄存器的值为 1，而在 `TEST_FAIL` 的地址，程序将 `x3` 寄存器的值设置为非 1 值。因此最终可以通过判断 `x3` 的值来界定程序的运行结果到底是成功了还是失败了。

2.2.2 编译 ISA 自测试用例

`riscv-tests` 中的这些指令集架构（ISA）测试用例都是使用汇编语言编写，为了在仿真阶段能够被处理器执行，需要将这些汇编程序编译成二进制代码。在 `e200_opensource` 的以下目录（`generated` 文件夹）下，已经预先上传了一组编译成的可执行文件和反汇编文件，以及能够被 Verilog 的 `readmemh` 函数读入的文件。

```
e200_opensource
|----riscv-tools          // 存放所需 riscv-tools 的目录
|----riscv-tests          // 存放一些测试用例的目录
|----isa
|----generated            // 编译好的 tests 文件夹
|----rv32ui-p-addi        // 编译出的 elf 文件
|----rv32ui-p-addi.dump   // 反汇编文件
|----rv32ui-p-addi.verilog // 可被 Verilog 的 readmemh
                           // 函数读入的文件
.....
```

反汇编文件（譬如 rv32ui-p-addi.dump）的内容如图 2-3 所示。

```
rv32ui-p-add:      file format elf32-littleriscv

Disassembly of section .text.init:

80000000 <_start>:
80000000:  a081                j      80000040 <reset_vector>
80000002:  0001                nop

80000004 <trap_vector>:
80000004:  34202f73           csrr    t5,mcause
80000008:  4fa1                li     t6,8
8000000a:  03ff0663           beq    t5,t6,80000036 <write_tohost>
8000000e:  4fa5                li     t6,9
80000010:  03ff0363           beq    t5,t6,80000036 <write_tohost>
80000014:  4fad                li     t6,11
80000016:  03ff0063           beq    t5,t6,80000036 <write_tohost>
8000001a:  80000f17           auipc  t5,0x80000
8000001e:  fe6f0f13           addi   t5,t5,-26 # 0 <_start-0x80000000>
80000022:  000f0363           beqz   t5,80000028 <trap_vector+0x24>
80000026:  8f02                jr     t5
80000028:  34202f73           csrr    t5,mcause
8000002c:  000f5363           bgez   t5,80000032 <handle_exception>
80000030:  a009                j      80000032 <handle_exception>

80000032 <handle_exception>:
80000032:  5391e193           ori    gp,gp,1337

80000036 <write_tohost>:
80000036:  00001f17           auipc  t5,0x1
8000003a:  fc3f2523           sw     gp,-54(t5) # 80001000 <tohost>
8000003e:  bfe5                j      80000036 <write_tohost>

80000040 <reset_vector>:
80000040:  f1402573           csrr    a0,mhartid
80000044:  e101                bnez   a0,80000044 <reset_vector+0x4>
80000046:  4181                li     gp,0
80000048:  00000297           auipc  t0,0x0
8000004c:  fbc28293           addi   t0,t0,-68 # 80000004 <trap_vector>
80000050:  30529073           csrw   mtvec,t0
80000054:  80000297           auipc  t0,0x80000
80000058:  fac28293           addi   t0,t0,-84 # 0 <_start-0x80000000>
8000005c:  00028e63           beqz   t0,80000078 <reset_vector+0x38>
80000060:  10529073           csrw   stvec,t0
```

图 2-3 反汇编文件内容片段

Verilog 的 readmemh 函数能够读入的文件（譬如 rv32ui-p-addi.verilog）内容如图 2-4 所示。

```

@00000000
81 A0 01 00 73 2F 20 34 A1 4F 63 06 FF 03 A5 4F
63 03 FF 03 AD 4F 63 00 FF 03 17 0F 00 80 13 0F
6F FE 63 03 0F 00 02 8F 73 2F 20 34 63 53 0F 00
09 A0 93 E1 91 53 17 1F 00 00 23 25 3F FC E5 BF
73 25 40 F1 01 E1 81 41 97 02 00 00 93 82 C2 FB
73 90 52 30 97 02 00 80 93 82 C2 FA 63 8E 02 00
73 90 52 10 B7 B2 00 00 93 82 92 10 73 90 22 30
73 23 20 30 E3 9F 62 FA 73 50 00 30 97 02 00 00
93 82 42 01 73 90 12 34 73 25 40 F1 73 00 20 30
81 40 01 41 33 8F 20 00 81 4E 89 41 63 1D DF 37
85 40 05 41 33 8F 20 00 89 4E 8D 41 63 15 DF 37
8D 40 1D 41 33 8F 20 00 A9 4E 91 41 63 1D DF 35
81 40 37 81 FF FF 33 8F 20 00 B7 8E FF FF 95 41
63 13 DF 35 B7 00 00 80 01 41 33 8F 20 00 B7 0E
00 80 99 41 63 19 DF 33 B7 00 00 80 37 81 FF FF
33 8F 20 00 B7 8E FF 7F 9D 41 63 1E DF 31 81 40
37 81 00 00 13 01 F1 FF 33 8F 20 00 B7 8E 00 00
93 8E FE FF A1 41 63 10 DF 31 B7 00 00 80 93 80
F0 FF 01 41 33 8F 20 00 B7 0E 00 80 93 8E FE FF
A5 41 63 12 DF 2F B7 00 00 80 93 80 F0 FF 37 81
00 00 13 01 F1 FF 33 8F 20 00 B7 8E 00 80 93 8E
EE FF A9 41 63 11 DF 2D B7 00 00 80 37 81 00 00
13 01 F1 FF 33 8F 20 00 B7 8E 00 80 93 8E FE FF
AD 41 63 12 DF 2B B7 00 00 80 93 80 F0 FF 37 81
FF FF 33 8F 20 00 B7 8E FF 7F 93 8E FE FF B1 41
63 13 DF 29 81 40 13 01 F0 FF 33 8F 20 00 93 0E
F0 FF B5 41 63 19 DF 27 93 00 F0 FF 05 41 33 8F
20 00 81 4E B9 41 63 10 DF 27 93 00 F0 FF 13 01
F0 FF 33 8F 20 00 93 0E E0 FF BD 41 63 15 DF 25
85 40 37 01 00 80 13 01 F1 FF 33 8F 20 00 B7 0E
00 80 C1 41 63 19 DF 23 B5 40 2D 41 8A 90 E1 4E
C5 41 63 92 D0 23 B9 40 2D 41 06 91 E5 4E C9 41
63 1B D1 21 B5 40 86 90 E9 4E CD 41 63 95 D0 21
01 42 B5 40 2D 41 33 8F 20 00 13 03 0F 00 05 02
89 42 E3 18 52 FE E1 4E D1 41 63 16 D3 1F 01 42
B9 40 2D 41 33 8F 20 00 01 00 13 03 0F 00 05 02

```

图 2-4 Verilog 的 readmemh 函数可读入文件内容片段

如果用户想修改汇编程序的源代码并需要重新编译，请参见《手把手教你设计 CPU——RISC-V 处理器篇》第 17 章了解详细步骤。

2.3 E203 开源项目的测试平台（TestBench）

在 e200_opensource 的如下目录已经创建了一个简单的由 Verilog 编写的 TestBench 测试平台。

```

e200_opensource
|----tb                // 存放 Verilog TestBench (测试平台) 的目录
|----tb_top.v          // 简单地 Verilog TestBench 顶层文件

```

在测试平台中主要的功能如下：

- 例化 DUT 文件，生成 clock 和 reset 信号。
- 根据运行命令解析出测试用例的名称， 并使用 Verilog 的 readmemh 函数读入相应的文件（譬如 rv32ui-p-addi.verilog）内容，然后使用文件中的内容初始化 ITCM（由 Verilog 编写的二维数组充当行为模型），如图 2-5 所示。
- 在运行结束后分析该测试用例是否执行成功，在 Testbench 的源文件中对 x3 寄存器的值进行判断，如果 x3 的值为 1，则意味着通过，向终端上将打印 PASS 字样，否则将打印 FAIL 字样。如图 2-6 所示。


```

integer i;

reg [7:0] itcm_mem [0:(`E200_ITCM_RAM_DP*8)-1];
initial begin
    $readmemh({testcase, ".verilog"}, itcm_mem);

    for (i=0;i<(`E200_ITCM_RAM_DP);i=i+1) begin
        `ITCM.mem_r[i][00+7:00] = itcm_mem[i*8+0];
        `ITCM.mem_r[i][08+7:08] = itcm_mem[i*8+1];
        `ITCM.mem_r[i][16+7:16] = itcm_mem[i*8+2];
        `ITCM.mem_r[i][24+7:24] = itcm_mem[i*8+3];
        `ITCM.mem_r[i][32+7:32] = itcm_mem[i*8+4];
        `ITCM.mem_r[i][40+7:40] = itcm_mem[i*8+5];
        `ITCM.mem_r[i][48+7:48] = itcm_mem[i*8+6];
        `ITCM.mem_r[i][56+7:56] = itcm_mem[i*8+7];
    end

    $display("ITCM 0x00: %h", `ITCM.mem_r[8'h00]);
    $display("ITCM 0x01: %h", `ITCM.mem_r[8'h01]);
    $display("ITCM 0x02: %h", `ITCM.mem_r[8'h02]);
    $display("ITCM 0x03: %h", `ITCM.mem_r[8'h03]);
    $display("ITCM 0x04: %h", `ITCM.mem_r[8'h04]);
    $display("ITCM 0x05: %h", `ITCM.mem_r[8'h05]);
    $display("ITCM 0x06: %h", `ITCM.mem_r[8'h06]);
    $display("ITCM 0x07: %h", `ITCM.mem_r[8'h07]);
    $display("ITCM 0x16: %h", `ITCM.mem_r[8'h16]);
    $display("ITCM 0x20: %h", `ITCM.mem_r[8'h20]);

end

```

图 2-5 使用 Verilog 的 readmemh 函数读入文件初始化 ITCM

```

@(pc_write_to_host_cnt == 32'd8)

$display("-----");
$display("-----");
$display("----- Test Result Summary -----");
$display("-----");
$display("-----");
$display("TESTCASE: %s -----", testcase);
$display("-----Total cycle_count value: %d -----", cycle_count);
$display("-----The valid Instruction Count: %d -----", valid_ir_cycle);
$display("-----The test ending reached at cycle: %d -----", pc_write_to_host_cycle);
$display("-----The final x3 Reg value: %d -----", x3);
$display("-----");

if (x3 == 1) begin
$display("----- TEST_PASS -----");
$display("-----");
$display("----- #####      ##      ###      ### -----");
$display("----- #      #      #      # -----");
$display("----- #      #      #      # -----");
$display("----- #####      #####      # -----");
$display("----- #      #      #      # -----");
$display("----- #      #      #      # -----");
$display("----- #      #      #      # -----");
$display("-----");
end
else begin
$display("----- TEST_FAIL -----");
$display("-----");
$display("----- #####      ##      #      # -----");
$display("----- #      #      #      # -----");
$display("----- #####      #      #      # -----");
$display("----- #      #####      #      # -----");
$display("----- #      #      #      # -----");
$display("----- #      #      #      # -----");
$display("-----");
end
end

```

图 2-6 Testbench 中打印测试用例的结果

2.4 在 Verilog TestBench 中运行测试用例

假设用户想使用 E203 源代码运行基于 Verilog 的仿真测试程序，可以使用如下步骤进行。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置：

- (1) 使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
- (2) 由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统。有关如何安装 VMware 以及 Ubuntu 操作系统本文不做介绍，有关 Linux 的基本使用本文也不做介绍，请用户自行查阅资料学习。

// 步骤二：将 e200_opensource 项目下载到本机 Linux 环境中，使用如下命令：

```

git clone https://github.com/SI-RISCV/e200_opensource.git
// 经过此步骤将项目克隆下来，本机上即可具有如前文所述完整的 e200_opensource 目
// 录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩写指代。

```

// 步骤三：编译 RTL 代码，使用如下命令：

```

cd <your_e200_dir>/vsim

```

```

// 进入到 e200_opensource 目录文件夹下面的 vsim 目录。

make install CORE=e203
// 运行该命令指明需要为 e203 进行编译，该命令会在 vsim 目录下生成一个 install
// 子文件夹，在其中放置所需的脚本，且将脚本中的关键字设置为 e203。

make compile
// 编译 Core 和 SoC 的 RTL 代码
// 注意：在此步骤之中，编译 Verilog 代码需要使用到仿真器工具，在 github 上的 Makefile
// 中使用的是免费的 iverilog 工具，如果需要使用商业 EDA 的用户需要自行修改 Makefile 中的
// 内容，如图 2-2 所示。
// 对于免费的 iverilog 工具如何安装请用户在互联网上自行搜索。

// 步骤四：运行默认的一个 testcase ( 测试用例 )，使用如下命令：
make run_test
// 注意：在此步骤中，运行仿真需要使用仿真器工具，在 github 上的 Makefile 中此部分空缺，
// 实际运行的是“echo PASS”命令打印一个虚假的 PASS 到 log 文件中。用户需要使用真
// 正的仿真器运行仿真得到真实的运行结果，如图 2-7 所示。
// 注意：make run_test 将执行 e200_opensource/ riscv-tools/
// riscv-tests/isa/generated 目录中的一个默认 testcase，如果希望运行所有的
// 回归测试，请参见步骤五。

// 步骤五：运行回归 ( regression ) 测试集，使用如下命令：
make regress_run
// 注意：这使用 e200_opensource/ riscv-tools/riscv-tests/isa/generated
// 目录中 testcases，逐个的运行 testcase。

// 步骤六：查看回归测试结果：
make regress_collect
// 该命令将收集步骤五中运行的测试集的结果，将打印若干行的结果，每一行对应一个测
// 试用例，如果那个测试用例运行通过，那一行则打印的 PASS，如果运行失败，那一行则
// 打印的 FAIL。如图 2-8 所示。

```

```

RTL_V_FILES      := $(wildcard ${VSRC_DIR}/*.v)
TB_V_FILES       := $(wildcard ${VTB_DIR}/*.v)

# The following portion is depending on the EDA tools you are using, Please add them by yourself
according to your EDA vendors

SIM_TOOL         := #To-ADD: to add the simulatoin tool

SIM_OPTIONS      := #To-ADD: to add the simulatoin tool options

SIM_EXEC         := #To-ADD: to add the simulatoin executable
SIM_EXEC         := @echo "Test Result Summary: PASS" # This is a fake run to just direct print PASS info
to the log, the user need to actually replace it to the real EDA command

WAV_TOOL         := #To-ADD: to add the waveform tool
WAV_OPTIONS      := #To-ADD: to add the waveform tool options
WAV_PFIX         := #To-ADD: to add the waveform file postfix

```

图 2-7 编译工具的设置

```

PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-simple/rv32ui-p-simple.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-beq/rv32ui-p-beq.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-scall/rv32mi-p-scall.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-lbu/rv32ui-p-lbu.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-jal/rv32ui-p-jal.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-fence_i/rv32ui-p-fence_i.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-bge/rv32ui-p-bge.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-ori/rv32ui-p-ori.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-breakpoint/rv32mi-p-breakpoint.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-sh/rv32ui-p-sh.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-mcsr/rv32mi-p-mcsr.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-sll/rv32ui-p-sll.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32uc-p-rvc/rv32uc-p-rvc.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-srli/rv32ui-p-srli.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-sra/rv32ui-p-sra.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-lb/rv32ui-p-lb.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-or/rv32ui-p-or.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-slli/rv32ui-p-slli.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-lui/rv32ui-p-lui.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-addi/rv32ui-p-addi.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-illegal/rv32mi-p-illegal.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-and/rv32ui-p-and.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-ma_addr/rv32mi-p-ma_addr.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-xori/rv32ui-p-xori.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-bgeu/rv32ui-p-bgeu.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-ma_fetch/rv32mi-p-ma_fetch.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32ui-p-sw/rv32ui-p-sw.log
PASS /home/zhenbohu/jx_work/e200_opensource/vsim/run/rv32mi-p-csr/rv32mi-p-csr.log

```

图 2-8 运行回归测试的结果示例

注意，以上的回归测试只是运行 `riscv-tests` 中提供的非常基本的自测试汇编程序，并不能达到充分验证处理器核的效果，因此如果用户修改了处理器的 Verilog 源代码而仅仅运行以上的回归测试将无法保证处理器的功能完备正确性。

3 蜂鸟 E203 开源 SoC

对于一个处理器核,还需要配套的 SoC 才能具备完整的功能。蜂鸟 E203 内核不仅仅完全开源了 Core 的实现,还搭配完整的开源 SoC 平台,请参见《蜂鸟 E203 开源 SoC 简介》了解更多 SoC 的介绍与信息。

蜂鸟 E203 开源 SoC 的代码结构如下所示。

```
e200_opensource
|----rtl                                // 存放 RTL 的目录
|----e203                              // E203 核和 SoC 的 RTL 目录
|----general                          // 存放一些公用的通用 RTL 代码
|----core                            // 存放 E203 Core 的 RTL 代码,列举主要模块如下,全部模块参见 Github
|----config.v                        // 参数配置文件
|----e203_biu.v                      // BIU 模块
|----e203_reset_ctrl.v              // Core 的 Reset Control 模块
|----e203_clk_ctrl.v                // Core 的 Clock Control 模块
|----e203_cpu_top.v                 // Core 的顶层模块
|----e203_cpu.v                     // Core 去除了 SRAM 之后的逻辑顶层模块
|----e203_core.v                    // Core 的主体逻辑模块
|----e203_dtcn_ctrl.v               // DTCM 的控制模块
|----e203_itcm_ctrl.v               // ITCM 的控制模块
|----e203_exu.v                     // Core 内部 EXU 单元顶层模块
|----e203_ifu.v                     // Core 内部 IFU 单元顶层模块
|----e203_lsu.v                     // Core 内部 LSU 单元顶层模块
|----e203_srams.v                   // Core 的所有 SRAM 的顶层模块
|----e203_itcm_ram.v                // ITCM 的 SRAM 模块
|----e203_dtcn_ram.v                // DTCM 的 SRAM 模块
|----fab                            // 存放总线 bus fabric 的 RTL 代码
|---- sirv_icblto4_bus.v             // 将 1 组 ICB 总线转换成 4 路 ICB 总线
|---- sirv_icblto8_bus.v             // 将 1 组 ICB 总线转换成 8 路 ICB 总线
|---- sirv_icblto16_bus.v            // 将 1 组 ICB 总线转换成 16 路 ICB 总线
|----subsys                          // 存放完整子系统顶层的 RTL 代码
|---- e203_subsys_top.v              // 子系统的顶层
|---- e203_subsys_main.v             // 子系统的主体部分(可关电)顶层
|---- e203_subsys_plic.v             // PLIC 顶层
|---- e203_subsys_clint.v            // CLINT 顶层
|---- e203_subsys_mems.v             // 子系统的存储部分顶层
|---- e203_subsys_perips.v           // 子系统的外设部分顶层
|----mems                            // 存放 memory 模块的 RTL 代码
|----perips                          // 存放外设 peripherals 模块的 RTL 代码
|---- sirv_aon*.v                   // Always-on(电源常开)部分模块
|---- sirv_clint*.v                 // CLINT 的模块
|---- sirv_flash_qspi*.v            // Flash 专用的 QSPI 模块
|---- sirv_gpio*.v                  // GPIO 的模块
|---- sirv_plic*.v                  // PLIC 的模块
|---- sirv_pmu*.v                   // PMU 的模块
|---- sirv_pwm16*.v                 // 16bits 精度的 PWM 模块
|---- sirv_pwm8*.v                  // 8bits 精度的 PWM 模块
|---- sirv_qspi_lcs*.v              // 1 个 CS 选通的 QSPI 模块
|---- sirv_qspi_4cs*.v              // 4 个 CS 选通的 QSPI 模块
|---- sirv_qspi*.v                  // 其他 QSPI 子模块
|---- sirv_rtc*.v                   // RTC 模块
|---- sirv_uart*.v                  // UART 模块
|---- sirv_wdog*.v                  // WatchDog 模块
|----debug                          // 存放 debug 相关模块的 RTL 代码
|----soc                            // 存放 SoC 顶层的 RTL 代码
|----e203_soc_top.v                 // SoC 顶层
```

各个主要的代码模块简述如下：

- `general` 目录主要用于存放一些通用的 Verilog RTL 模块供整个 SoC 公用，譬如一些 DFF（D 触发器寄存器）定义文件，ICB 总线的基础模块等等。
- `core` 目录主要用于存放处理器核模块的 Verilog RTL 代码。其模块间的层次结构如图 3-1 所示，`e203_cpu_top` 是蜂鸟 E203 处理器核的顶层，其接口请参见源代码注释。
- `fab` 目录主要实现 SoC 中 ICB Bus Fabric 模块的 Verilog RTL 代码。`serv_icb1to4_bus.v`，`serv_icb1to8_bus.v` 或者 `serv_icb1to16_bus.v` 实际例化调用了 `serv_icb_splt` 模块将一组 ICB 总线按照地址区间分发成为 4 组，8 组或者 16 组 ICB 总线。
- `subsys` 目录包含了 SoC 的主体顶层模块的 Verilog RTL 代码，其中 `e203_subsys_top` 是事实上的 SoC 顶层文件，它例化了 Main Domain 模块（`e203_subsys_main.v`）和 Always-on Domain 模块（`e203_aon_top.v`）。其模块间的层次结构如图 3-1 所示。`e203_subsys_mems` 模块实现了系统存储总线（System Memory Bus），通过调用例化 `serv_icb1to8_bus` 模块并且配置其参数的方式来配置每个从设备的地址区间，如图 3-2 所示。`e203_subsys_perips` 模块实现了系统设备总线（System Peripheral Bus），通过调用例化 `serv_icb1to16_bus` 模块并且配置其参数的方式来配置每个从设备的地址区间，如图 3-3 所示。除了已经实现的从设备，还预留了地址区间实现外部存储（`sysmem`），外部外设（`sysper`）和外部快速 IO（`sysfio`）总线接口，如图 3-4 所示。
- `mems` 目录主要用于存放 memory 模块的 Verilog RTL 代码，由于 Memory 的具体实现依赖于芯片生产加工厂（foundry）譬如 SIMC 或者 TSMC 的 Memory 宏单元，因此本文件夹下的 Verilog RTL 代码仅仅是行为模型。
- `perips` 目录主要用于存放各种外设（Peripherals）模块的 Verilog RTL 代码，譬如 GPIO，UART，SPI 等。大部分的 Peripherals 的 Verilog RTL 代码是直接复制于 SiFive 的 Freedom E310 项目中 Chisel 语言生成的出的 Verilog RTL 代码，在此基础上将其 TileLink 总线接口修改成了 ICB 总线接口，如图 3-5 中所示的 GPIO 模块 ICB 总线接口。
- `debug` 目录包含了 SoC 中有关 debugger 调试器模块的 Verilog RTL 代码。
- `soc` 目录主要用于存放 SoC 顶层模块的 Verilog RTL 代码。`e203_soc_top.v` 是一个简单地顶层 SoC Wrapper 模块，将 `e203_subsys_top` 进行例化。另外由于 `e203_subsys_top` 模块输出的 `sysmem`，`sysfio` 和 `sysmem` 总线在此 FPGA SoC 中并没有连接任何外部从设备。为了防止软件程序访问到这些总线接口的地址区间无任何返回而挂死，在 `e203_soc_top` 顶层模块中将这 ICB 总线的 Command Channel 信号直接反接到其 Response Channel，同时将 Response Channel 中的其他返回信号连接成常数 0，如图 3-6 所示。

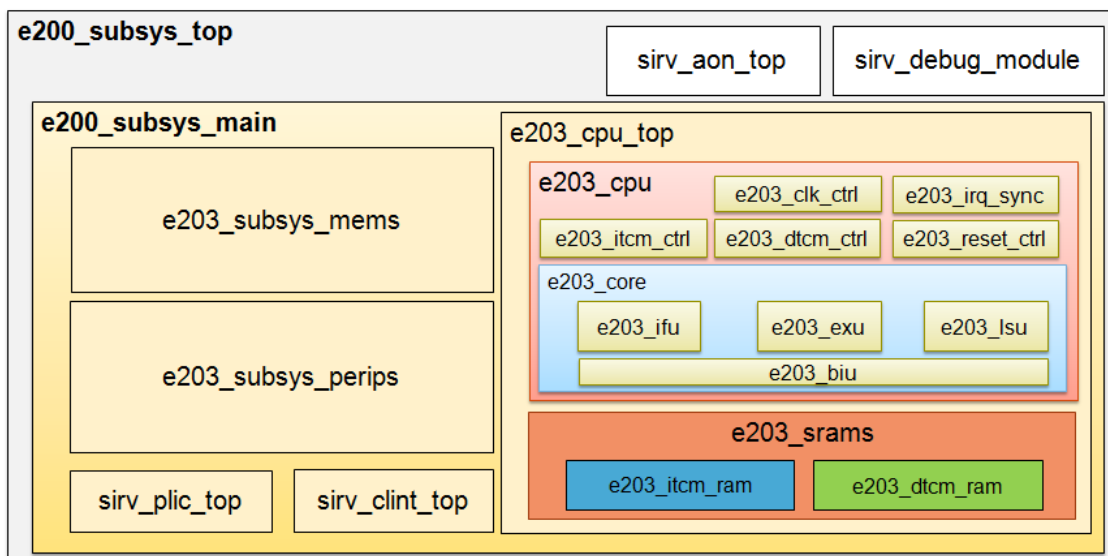


图 3-1 Subsys 模块层次结构图

```

// There are several slaves for Mem bus, including:
// * DM      : 0x0000 0000 -- 0x0000 0FFF
// * MROM    : 0x0000 1000 -- 0x0000 1FFF
// * OTP-RO  : 0x0002 0000 -- 0x0003 FFFF
// * QSPI0-RO : 0x2000 0000 -- 0x3FFF FFFF
// * SysMem   : 0x8000 0000 -- 0xFFFF FFFF

sirv_icblto8_bus # (
    .AW      (32),
    .DW      (`E200_XLEN),
    .SPLT_FIFO_OUTS_NUM (1), // The Mem only allow 1 oustanding
    .SPLT_FIFO_CUT_READY (1), // The Mem always cut ready
    // * DM      : 0x0000 0000 -- 0x0000 0FFF
    .00_BASE_ADDR      (32'h0000_0000),
    .00_BASE_REGION_LSB (12),
    // * MROM    : 0x0000 1000 -- 0x0000 1FFF
    .01_BASE_ADDR      (32'h0000_1000),
    .01_BASE_REGION_LSB (12),
    // * OTP-RO  : 0x0002 0000 -- 0x0003 FFFF
    .02_BASE_ADDR      (32'h0002_0000),
    .02_BASE_REGION_LSB (17),
    // * QSPI0-RO : 0x2000 0000 -- 0x3FFF FFFF
    .03_BASE_ADDR      (32'h2000_0000),
    .03_BASE_REGION_LSB (29),
    // * SysMem   : 0x8000 0000 -- 0xFFFF FFFF
    .04_BASE_ADDR      (32'h8000_0000),
    .04_BASE_REGION_LSB (31),

```

图 3-2 e203_subsys_mems 代码片段


```

// The total address range for the PPI is from/to
// *****0x1000 0000 -- 0x1FFF FFFF
// There are several slaves for PPI bus, including:
// * AON      : 0x1000 0000 -- 0x1000 7FFF
// * PRCI     : 0x1000 8000 -- 0x1000 8FFF
// * OTP      : 0x1001 0000 -- 0x1001 0FFF
// * GPIO     : 0x1001 2000 -- 0x1001 2FFF
// * UART0    : 0x1001 3000 -- 0x1001 3FFF
// * QSPI0    : 0x1001 4000 -- 0x1001 4FFF
// * PWM0     : 0x1001 5000 -- 0x1001 5FFF
// * UART1    : 0x1002 3000 -- 0x1002 3FFF
// * QSPI1    : 0x1002 4000 -- 0x1002 4FFF
// * PWM1     : 0x1002 5000 -- 0x1002 5FFF
// * QSPI2    : 0x1003 4000 -- 0x1003 4FFF
// * PWM2     : 0x1003 5000 -- 0x1003 5FFF
// * SysPer   : 0x1100 0000 -- 0x11FF FFFF

sirv_icblt016_bus # (
    .AW      (32),
    .DW      (`E200_XLEN),
    .SPLT_FIFO_OUTS_NUM (1), // The peripherals only allow 1 outstanding
    .SPLT_FIFO_CUT_READY (1), // The peripherals always cut ready
    // * AON      : 0x1000 0000 -- 0x1000 7FFF
    .00_BASE_ADDR (32'h1000_0000),
    .00_BASE_REGION_LSB (15),
    // * PRCI     : 0x1000 8000 -- 0x1000 8FFF
    .01_BASE_ADDR (32'h1000_8000),
    .01_BASE_REGION_LSB (12),
    // * OTP      : 0x1001 0000 -- 0x1001 0FFF
    .02_BASE_ADDR (32'h1001_0000),
    .02_BASE_REGION_LSB (12),
    // * GPIO     : 0x1001 2000 -- 0x1001 2FFF
    .03_BASE_ADDR (32'h1001_2000),
    .03_BASE_REGION_LSB (12),
    // * UART0    : 0x1001 3000 -- 0x1001 3FFF
    .04_BASE_ADDR (32'h1001_3000),
    .04_BASE_REGION_LSB (12),

```

图 3-3 e203_subsys_perips 代码片段

```

////////////////////////////////////
// The ICB Interface to Private Peripheral Interface
//
// * Bus cmd channel
output sysper_icb_cmd_valid,
input sysper_icb_cmd_ready,
output [`E200_ADDR_SIZE-1:0] sysper_icb_cmd_addr,
output sysper_icb_cmd_read,
output [`E200_XLEN-1:0] sysper_icb_cmd_wdata,
output [`E200_XLEN/8-1:0] sysper_icb_cmd_wmask,
output sysper_icb_cmd_lock,
output sysper_icb_cmd_excl,
output [1:0] sysper_icb_cmd_size,
//
// * Bus RSP channel
input sysper_icb_rsp_valid,
output sysper_icb_rsp_ready,
input sysper_icb_rsp_err,
input sysper_icb_rsp_excl_ok,
input [`E200_XLEN-1:0] sysper_icb_rsp_rdata,

`ifdef E200_HAS_FIO //{
////////////////////////////////////
// The ICB Interface to Fast I/O
//
// * Bus cmd channel
output sysfio_icb_cmd_valid,
input sysfio_icb_cmd_ready,
output [`E200_ADDR_SIZE-1:0] sysfio_icb_cmd_addr,
output sysfio_icb_cmd_read,
output [`E200_XLEN-1:0] sysfio_icb_cmd_wdata,
output [`E200_XLEN/8-1:0] sysfio_icb_cmd_wmask,
output sysfio_icb_cmd_lock,
output sysfio_icb_cmd_excl,
output [1:0] sysfio_icb_cmd_size,
//

```

图 3-4 e203_subsys_top 代码片段


```

// Description:
// The top level module of gpio
//
// =====

module sirv_gpio_top(
    input    clk,
    input    rst_n,

    input          i_icb_cmd_valid,
    output         i_icb_cmd_ready,
    input  [32-1:0] i_icb_cmd_addr,
    input          i_icb_cmd_read,
    input  [32-1:0] i_icb_cmd_wdata,

    output         i_icb_rsp_valid,
    input          i_icb_rsp_ready,
    output  [32-1:0] i_icb_rsp_rdata,

    output  gpio_irq_0,
    output  gpio_irq_1,
    output  gpio_irq_2,
    output  gpio_irq_3,
    output  gpio_irq_4,
    output  gpio_irq_5,
    output  gpio_irq_6,

```

图 3-5 GPIO 模块顶层的 ICB 总线接口

```

.sysper_icb_cmd_valid (sysper_icb_cmd_valid),
.sysper_icb_cmd_ready (sysper_icb_cmd_ready),
.sysper_icb_cmd_read  (),
.sysper_icb_cmd_addr  (),
.sysper_icb_cmd_size  (),
.sysper_icb_cmd_wdata (),
.sysper_icb_cmd_wmask (),
.sysper_icb_cmd_lock  (),
.sysper_icb_cmd_excl  (),

.sysper_icb_rsp_valid (sysper_icb_cmd_valid),
.sysper_icb_rsp_ready (sysper_icb_cmd_ready),
.sysper_icb_rsp_err   (1'b0 ),
.sysper_icb_rsp_excl_ok(1'b0),
.sysper_icb_rsp_rdata (32'b0),

.sysfio_icb_cmd_valid(sysfio_icb_cmd_valid),
.sysfio_icb_cmd_ready(sysfio_icb_cmd_ready),
.sysfio_icb_cmd_read  (),
.sysfio_icb_cmd_addr  (),
.sysfio_icb_cmd_size  (),
.sysfio_icb_cmd_wdata (),
.sysfio_icb_cmd_wmask (),
.sysfio_icb_cmd_lock  (),
.sysfio_icb_cmd_excl  (),

.sysfio_icb_rsp_valid(sysfio_icb_cmd_valid),
.sysfio_icb_rsp_ready(sysfio_icb_cmd_ready),
.sysfio_icb_rsp_err   (1'b0 ),
.sysfio_icb_rsp_excl_ok(1'b0),
.sysfio_icb_rsp_rdata (32'b0),

```

图 3-6 e203_soc_top 顶层 PER 总线接口的连接

4 搭建 FPGA 原型平台

我们为蜂鸟 E203 开源 SoC 定制了专用的 FPGA 原型开发板和 JTAG 调试器，E203 开源项目基于该 FPGA 开发板，使用蜂鸟 E203 开源 SoC 搭建完整的原型平台与示例。

FPGA 原型主要分为两部分：FPGA 开发板，和调试器。接下来章节分别予以介绍。完整的 FPGA 开发板原型（包括 FPGA 开发板和调试器）如图 4-1 所示。

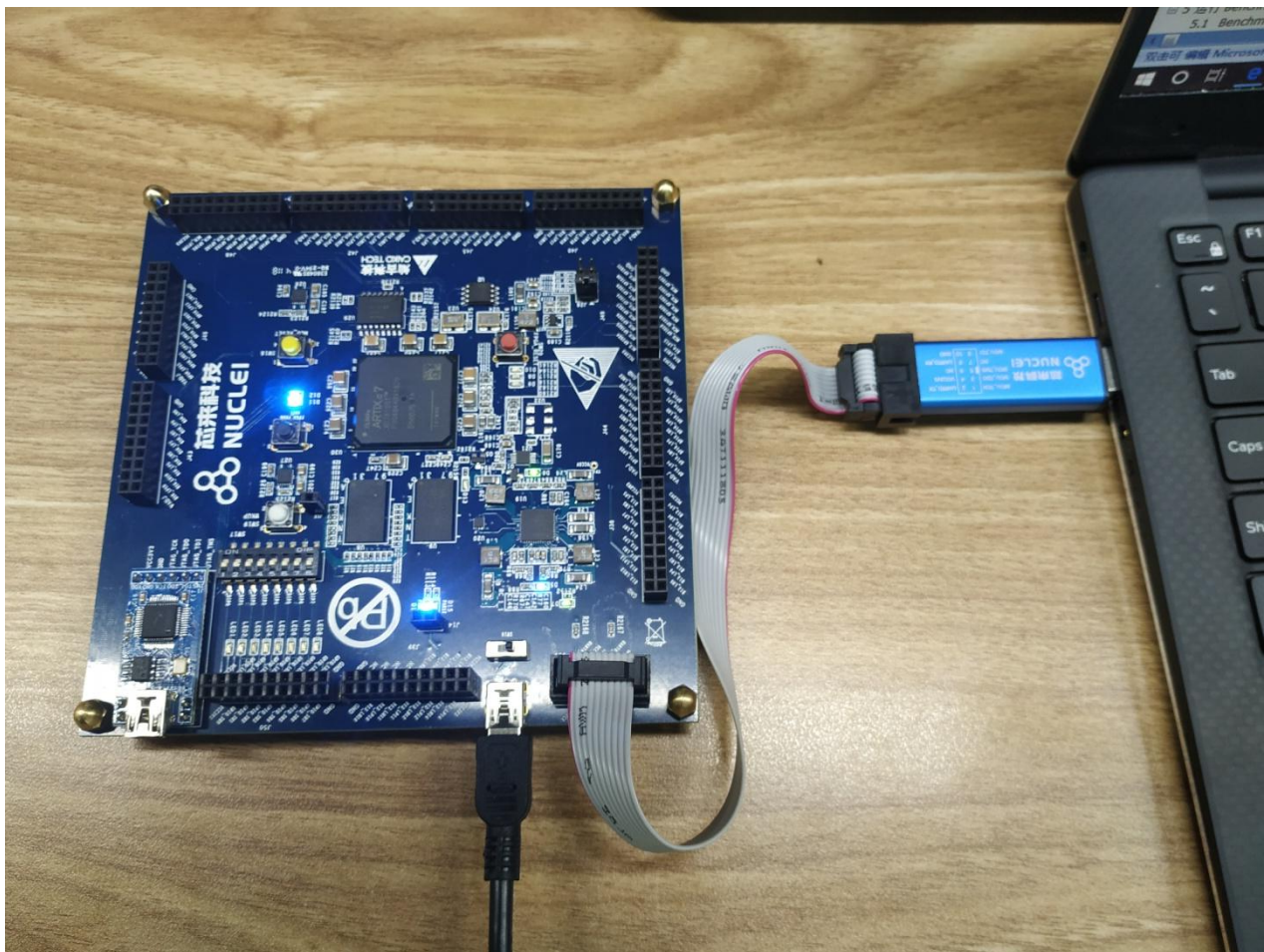


图 4-1 FPGA 开发板原型（包括 JTAG 调试器）

4.1 FPGA 开发板和项目介绍

蜂鸟 E203 专用 FPGA 开发板是一款低成本的入门级 Xilinx FPGA 开发板，如图 4-2 所示。该开发板不仅可以用于一块 FPGA 开发板作为电路设计使用，同时由于其预烧了蜂鸟 E203 开源 SoC（包括 E203 内核），因此其可以直接作为一块 MCU SoC 原型开发板进行嵌入式软件开发。

有关此 FPGA 开发板的详细介绍请参见《蜂鸟 FPGA 开发板和 JTAG 调试器介绍》。

若想购买此开发板，用户可以在 E203 开源项目的 Github 网页上
(https://github.com/SI-RISCV/e200_opensource/tree/master/boards) 了解此开发板的购买渠道。

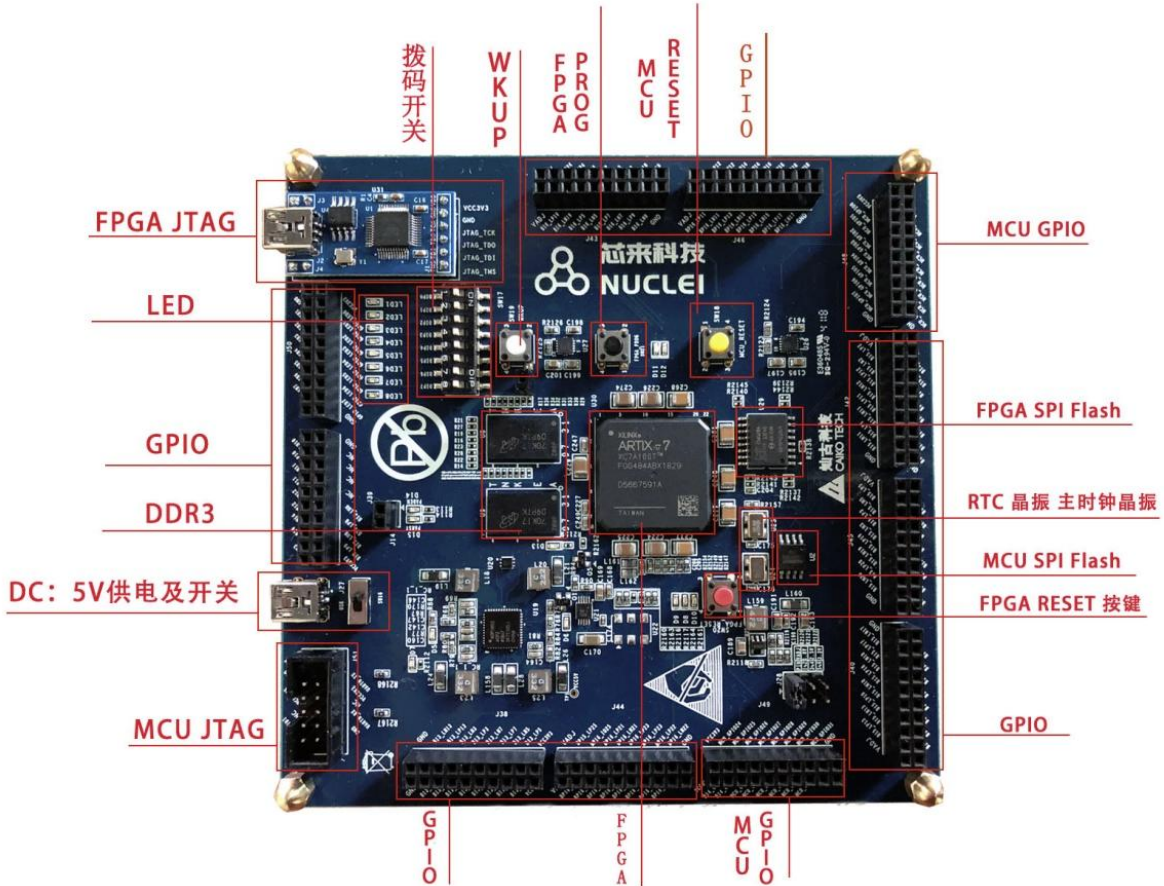


图 4-2 蜂鸟 E203 专用 FPGA 开发板

E203 开源项目 FPGA 项目相关的代码结构如下所示。

```
e200_opensource
|----fpga
|----nucleikit
|----constrs
|---- nuclei-master.xdc // 主约束文件
|----Makefile           // Makefile 脚本
|----script             // 存放运行脚本的文件夹
|----src                // 存放 Verilog 源代码的文件夹
|----system.org         // FPGA 系统的顶层模块
|----Makefile           // Makefile 脚本
```

FPGA 项目通过 Makefile (fpga/common.mk 文件) 将添加一个特殊的宏 `FPGA_SOURCE` 至 Core 的宏文件中，如图 4-3 所示。所以最终用于编译 FPGA 比特流的 RTL 源代码必须包含此宏(`FPGA_SOURCE`)。

```

19 # Install RTLs
20 install:
21     mkdir -p ${PWD}/install
22     cp ${PWD}/../rtl/${CORE} ${INSTALL_RTL} -rf
23     cp ${FPGA_DIR}/src/system.org ${INSTALL_RTL}/system.v -rf
24     sed -i 's/e200/${CORE}/g' ${INSTALL_RTL}/system.v
25     sed -i 'li\`define FPGA_SOURCE\`' ${INSTALL_RTL}/core/${CORE}_defines.v
26
27 EXTRA_FPGA_VSRCS :=
28 verilog := $(wildcard ${INSTALL_RTL}/*.v)
29 verilog += $(wildcard ${INSTALL_RTL}/*.v)
30

```

图 4-3 FPGA 项目宏定义文件中添加 FPGA_SOURCE

在 FPGA 的顶层模块（system.org）中除了例化 SoC 的顶层（e203_soc_top）之外，主要是使用 Xilinx 的 I/O Pad 单元例化顶层的 Pad。另外使用 Xilinx 的 MMCM 单元生成时钟。注意：SoC 的 Main Domain 使用的 MMCM 产生的高速时钟连接到 SoC 的 hfextclk，Always-On Domain 使用的是开发板上的低速实时时钟（32.768KHz）。

蜂鸟 E203 开源 SoC 的顶层 I/O Pad 经过 FPGA 的约束文件（nuclei-master.xdc）进行约束使之连接到 FPGA 芯片外部的引脚上面（譬如将 JTAG I/O 约束到开发板的 MCU_JTAG 插口引脚上）。

4.2 生成 mcs 文件烧写 FPGA

在前文中介绍了 E203 开源项目的 SoC 整体架构和 Verilog RTL 代码，为了使得该 SoC 能够真正运行在 FPGA 硬件上，需要将其编译成为 bitstream 文件然后烧录到 FPGA 中去。可以使用如下步骤进行编译和烧录。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置：
 （1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
 （2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
 有关如何安装 VMware 以及 Ubuntu 操作系统本文不做介绍，有关 Linux 的基本使用本文也不做介绍，请用户自行查阅资料学习。

// 步骤二：安装 Xilinx Vivado 软件至此虚拟机 Linux 操作系统中。有关如何安装 Xilinx Vivado 软件本文不做介绍，请用户自行查阅资料了解。

// 步骤三：将 e200_opensource 项目下载到本机 Linux 环境中，使用如下命令：

```

git clone https://github.com/SI-RISCV/e200_opensource.git
// 经过此步骤将项目克隆下来，本机上即可具有如前文所述完整的 e200_opensource 目
// 录文件夹，假设该目录为<your_e200_dir>，后文将使用该缩写指代。

```

// 步骤四：设置需要编译的 Core 的具体型号，使用如下命令：

```

cd <your_e200_dir>/fpga
// 进入到 e200_opensource 目录文件夹下面的 fpga 目录。

```



```
make install CORE=e203
// 运行该命令指明需要为 e203 内核进行编译, 该命令会在 fpga 目录下生成一个
// install 子文件夹, 在其中放置 Vivado 所需的脚本, 且将脚本中的关键字设置为 e203。
```

// 步骤五：生成 bitstream 文件或者 mcs 文件 (推荐使用 mcs 文件), 使用如下命令：

```
make bit
// 运行该命令将调用 Vivado 软件对 Verilog RTL 进行编译生成 bitstream 文件
// 生成的 bitstream 文件名和路径为
// <your_e200_dir>/fpga/nucleikit/obj/system.bit
// 该 bitstream 文件则可以使用 Vivado 软件的 Hardware Manager 功能将
// system.bit 烧录至 FPGA 中去。

// 熟悉 Vivado 和 Xilinx FPGA 使用的用户应该了解, bitstream 文件烧录到 FPGA 中
// 去之后 FPGA 不能掉电, 因为一旦掉电之后 FPGA 烧录的内容即丢失, 需要重新使用
// Vivado 的 Hardware Manager 进行烧录方能使用。
// 为了方便用户使用, Xilinx 的 Arty 开发板可以将需要烧录的内容写入开发板上的
// Flash 中, 然后在每次 FPGA 上电之后通过硬件电路自动将需要烧录的内容从外部的
// Flash 中读出并烧录到 FPGA 之中 ( 该过程非常的快, 不影响用户使用 )。由于 Flash
// 是非易失性的内存, 具有掉电后仍可保存的特性, 因此意味着将需要烧录的内容写入
// Flash 后, 每次掉电后无需使用 Hardware Manager 人工重新烧录 ( 而是硬件电路
// 快速自动完成 ), 即等效于, FPGA 上电即可使用。
// 有关此特性的详细原理与描述, 本文不做赘述, 请用户自行参阅 Arty 开发板手册。

// 为了能够将烧录 FPGA 的内容写入 Flash 中, 需要生成 mcs 文件, 使用如下命令：
make mcs
// 运行该命令将调用 Vivado 软件对 Verilog RTL 进行编译生成 mcs 文件
// 生成的 mcs 文件名和路径为
// <your_e200_dir>/fpga/nucleikit/obj/system.mcs
// 该 mcs 文件则可以使用 Vivado 软件的 Hardware Manager 功能将
// system.mcs 烧录至 FPGA 开发板中的 Flash 中去。
```

如何使用 Vivado 的 Hardware Manager 烧写 mcs 文件至 FPGA 开发板上的 Flash 中去, 参考如下步骤。

// 前提步骤 1：将 FPGA 开发板的“FPGA JTAG 接口”通过 USB 连接线与电脑的 USB 接口连接。开发板的“FPGA JTAG 接口”的位置请参见图 4-2 中标注所示。

// 前提步骤 2：将 FPGA 开发板的“DC : 5V 供电及开关”通过 USB 连接线与电脑 USB 接口或者电源插座连接, 并将“开关”拨开, 对 FPGA 开发板进行供电。开发板的“DC : 5V 供电及开关”的位置请参见图 4-2 中标注所示。

// 步骤一：打开 Vivado 软件。

// 步骤二：打开 Hardware Manager, 会自动连接 FPGA 开发板 (如果前提步骤 1 操作正确)。如图 4-4 和图 4-5 所示。

// 步骤三：右键 FPGA Device, 选择“Add Configuration Memory Device”。如图 4-6 所示。

// 步骤四：选择如下参数的 Flash, 如图 4-7 所示：

```
Part n25q128-3.3v
Manufacturer Micron
Family n25q
Type spi
Density 128
Width x1 x2 x4
```

// 步骤五：弹出“Do you want to program the configuration memory device now?”, 选择 OK

// 步骤六：在弹出的窗口中的<Configuration file>对话框中选择添加

<your_e200_dir>/fpga/nucleikit/system.mcs，然后选择 OK，则开始烧写 Flash，可能会花费几十秒的时间等待。

// 步骤七：一旦烧写 Flash 成功，则可以通过按开发板上的“FPGA_PROG”按键触发硬件电路使用 Flash 中的内容对 FPGA 重新进行烧录。

注意：FPGA 烧写成功之后，则可以无需再连接“FPGA JTAG 接口”的 USB 连接线。

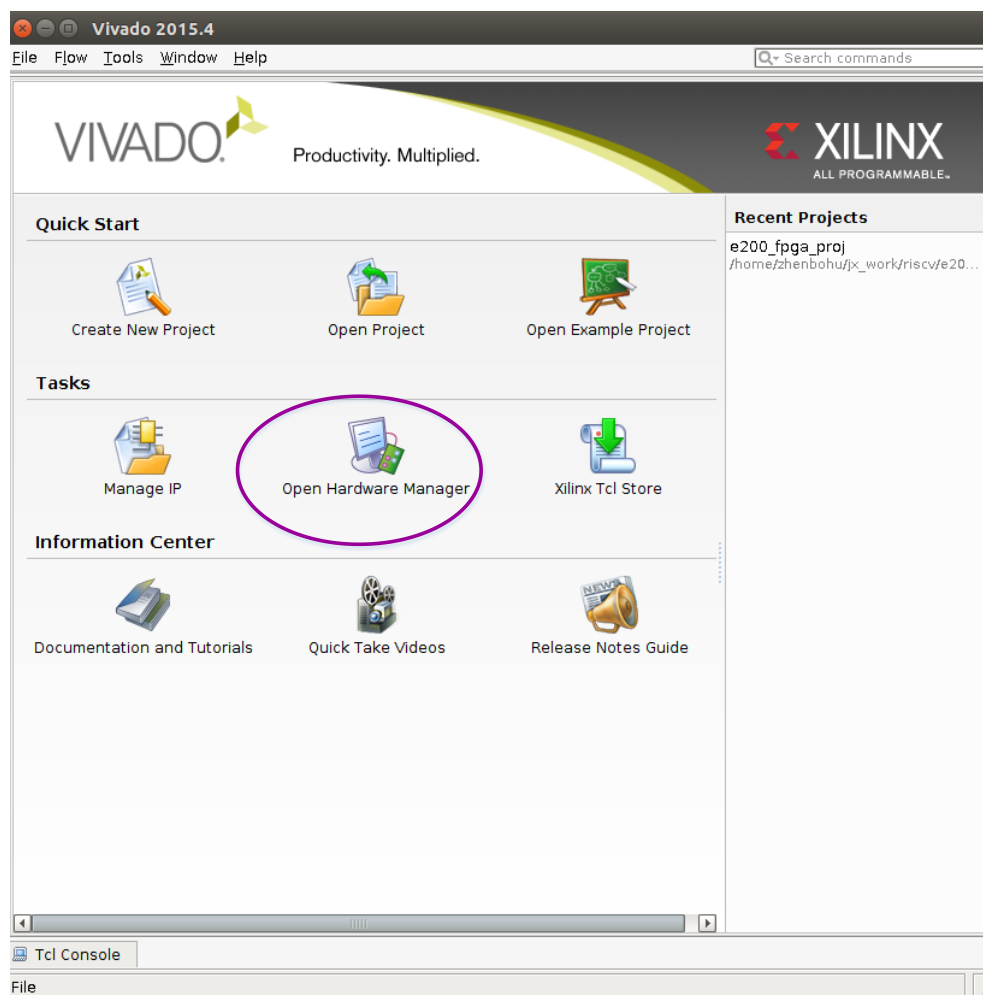


图 4-4 打开 Vivado Hardware Manager

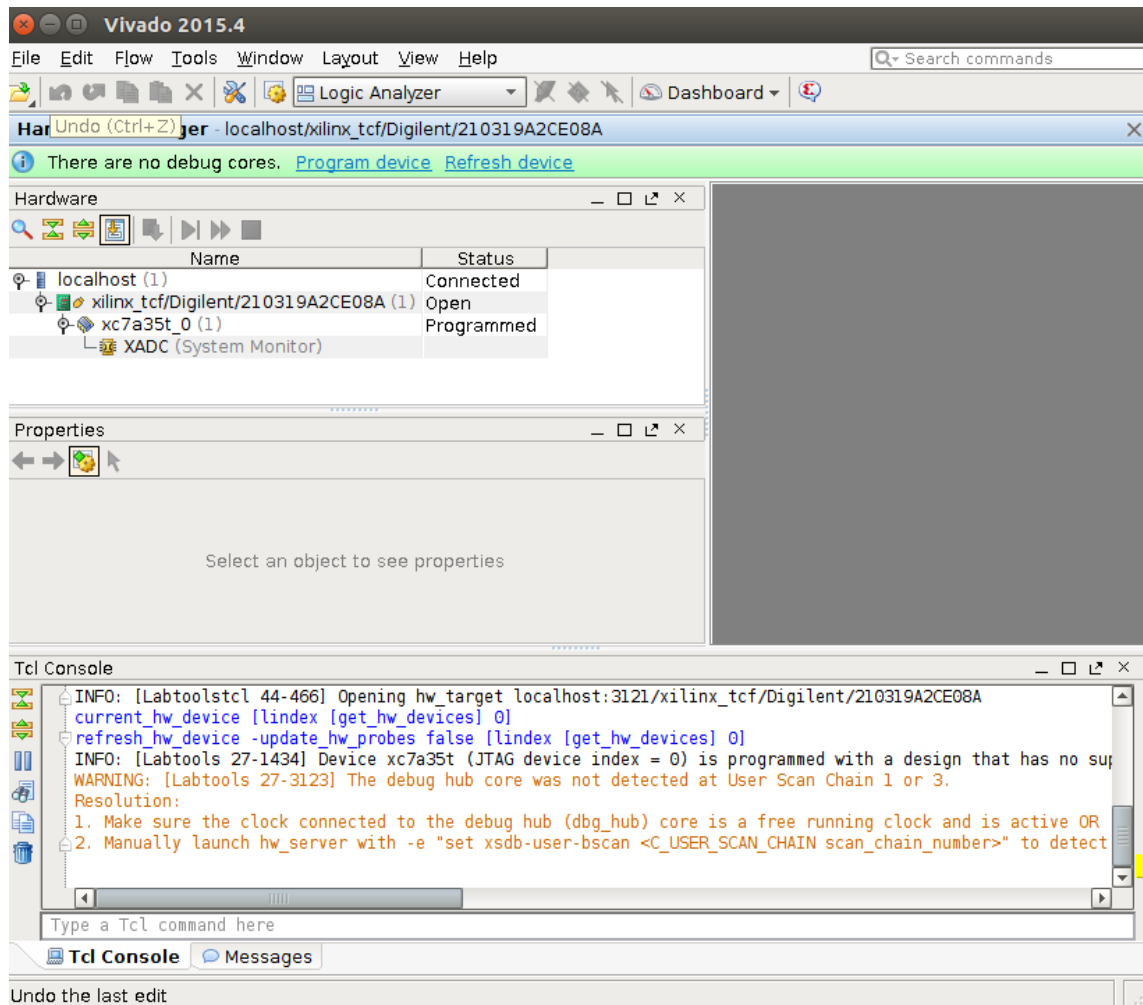


图 4-5 使用 Vivado Hardware Manager 连接 Arty 开发板

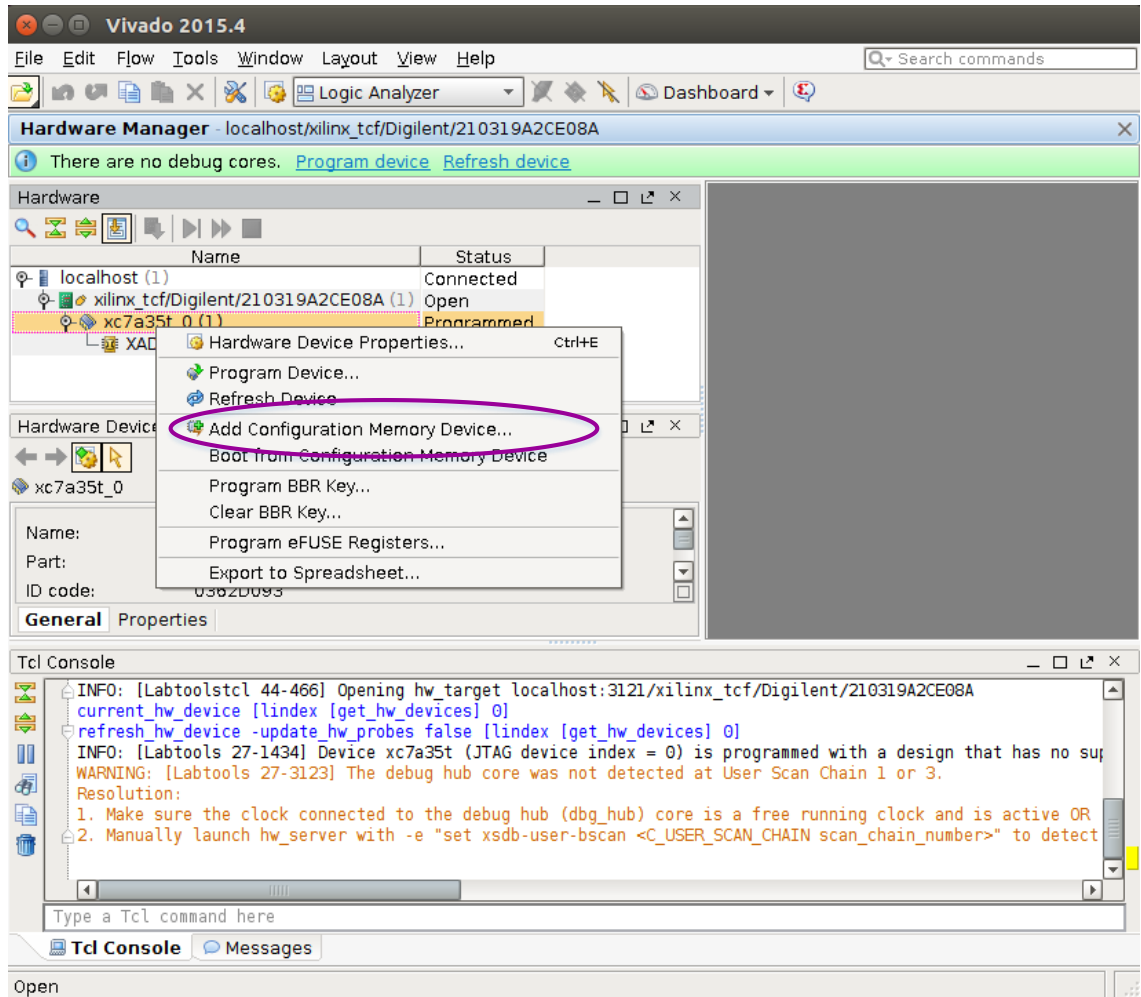


图 4-6 FPGA Device 选择 Add Configuration Memory Device

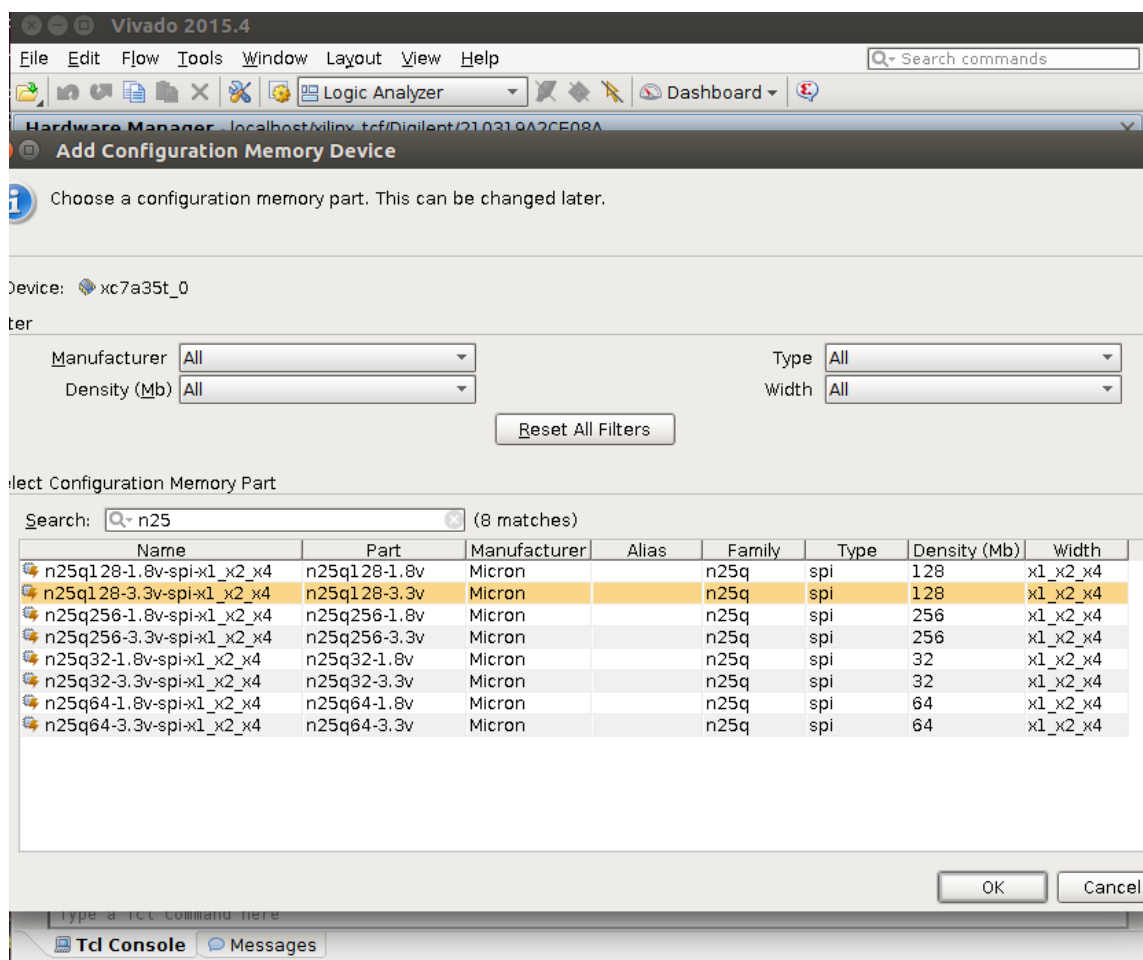


图 4-7 选择 Flash 类型

4.3 JTAG 调试器

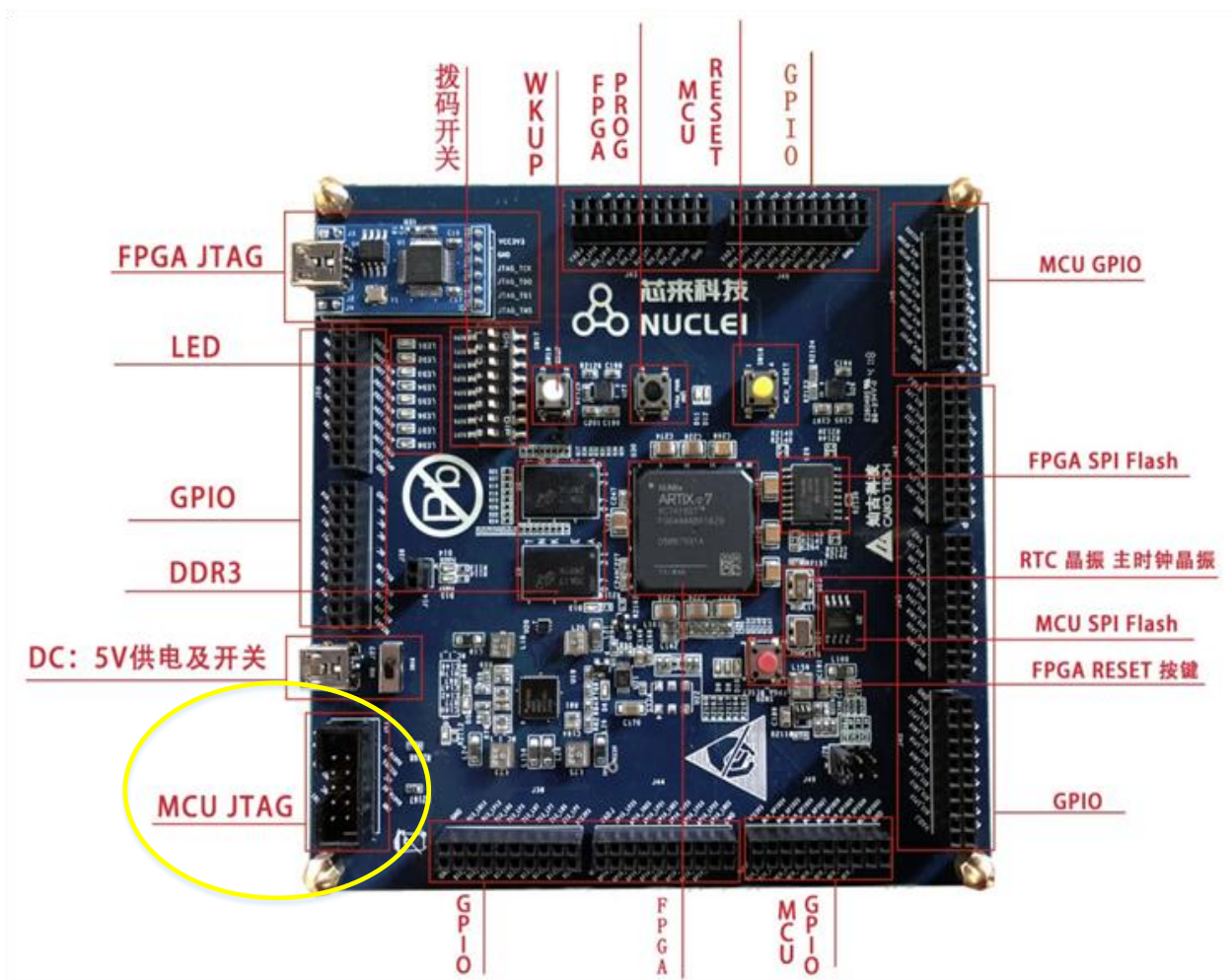


图 4-8 蜂鸟 E203 专用 FPGA 开发板的 MCU_JTAG 插槽

为了支持使用 GDB 进行交互式调试或者通过 GDB 动态下载程序到处理器中运行，需要为 FPGA 原型平台配备一个 JTAG 调试器（JTAG Debugger），E203 内核支持通过标准的 JTAG 接口对其进行调试，且 SoC 顶层 JTAG 的 I/O Pad 连接到了 FPGA 芯片的 pin 脚上，而该组 pin 脚在 E203 专用 FPGA 开发板上实际被连接到 MCU_JTAG 插槽上，如图 4-8 中黄色圆圈所示。

我们为 E203 内核定制了专用的“JTAG 调试器”，如图 4-9 中黄色圆圈所示。

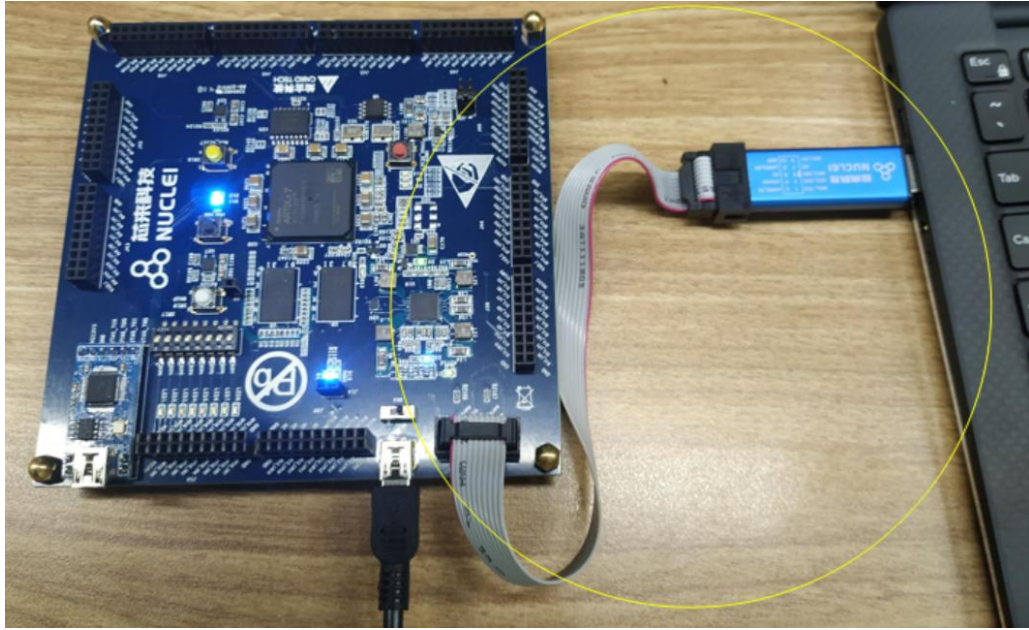


图 4-9 蜂鸟 E203 专用的 JTAG 调试器

有关此 JTAG 调试器的详细介绍请参见《蜂鸟 FPGA 开发板和 JTAG 调试器介绍》。

若想购买此 JTAG 调试器，用户可以在 E203 开源项目的 Github 网页上 (https://github.com/SI-RISCV/e200_opensource/tree/master/boards) 了解此 JTAG 调试器的购买渠道。

由于“JTAG 调试器”将其与上游主机 PC 的 USB 接口连接，因此上游 PC 的 USB 端口需要正确的设置以保证其有正确的权限。以 Ubuntu 16.04 为例，可以使用如下步骤进行配置。

// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置：

- (1) 使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。
 - (2) 由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统
- 有关如何安装 VMware 以及 Ubuntu 操作系统本文不做介绍，有关 Linux 的基本使用本文也不做介绍，请用户自行查阅资料学习。

// 步骤二：使用“JTAG 调试器”将主机 PC 与 FPGA 开发板连接，如图 4-9 中圆圈所示。注意使该 USB 接口被虚拟机的 Linux 系统识别（而非被 Windows 识别），如图 4-10 中圆圈所示，若 USB 图标在虚拟机中显示为高亮，则表明 USB 被虚拟机中 Linux 系统正确识别（而非被 Windows 识别），将 FPGA 开发板通电。

// 步骤三：使用如下命令查看 USB 设备的状态：

```
lsusb      // 运行该命令后会显示如下信息。
...
Bus 001 Device 029: ID 15ba:002a Olimex Ltd. ARM-USB-TINY-H JTAG interface
```

// 步骤四：使用如下命令设置 udev rules 使得该 USB 设备能够被 plugdev group 所访问：

```

sudo vi /etc/udev/rules.d/99-openocd.rules
    // 用 vi 打开该文件，然后添加以下内容至该文件中，然后保存退出。
# These are for the Olimex Debugger for use with Arty Dev Kit
SUBSYSTEM=="usb", ATTR{idVendor}=="15ba",
ATTR{idProduct}=="002a", MODE="664", GROUP="plugdev"
SUBSYSTEM=="tty", ATTRS{idVendor}=="15ba",
ATTRS{idProduct}=="002a", MODE="664", GROUP="plugdev"

// 步骤五：使用如下命令查看该 USB 设备是否属于 plugdev group:

ls /dev/ttyUSB*           // 运行该命令后会显示类似如下信息。
/dev/ttyUSB0 /dev/ttyUSB1

ls -l /dev/ttyUSB1        // 运行该命令后会显示类似如下信息。
crw-rw-r-- 1 root plugdev 188, 1 Nov 28 12:53 /dev/ttyUSB1

// 步骤六：将你自己的用户添加到 plugdev group 中：

whoami
    // 运行该命令能显示自己用户名，假设你的自己用户名显示为 your_user_name
    // 运行如下命令将 your_user_name 添加到 plugdev group 中
sudo usermod -a -G plugdev your user name

// 步骤七：确认自己的用户是否属于 plugdev group：

groups           // 运行该命令后会显示类似如下信息。
... plugdev ...
    // 只要从显示的 groups 中看到 plugdev 则意味着自己的用户属于该组，表示设置成功

```

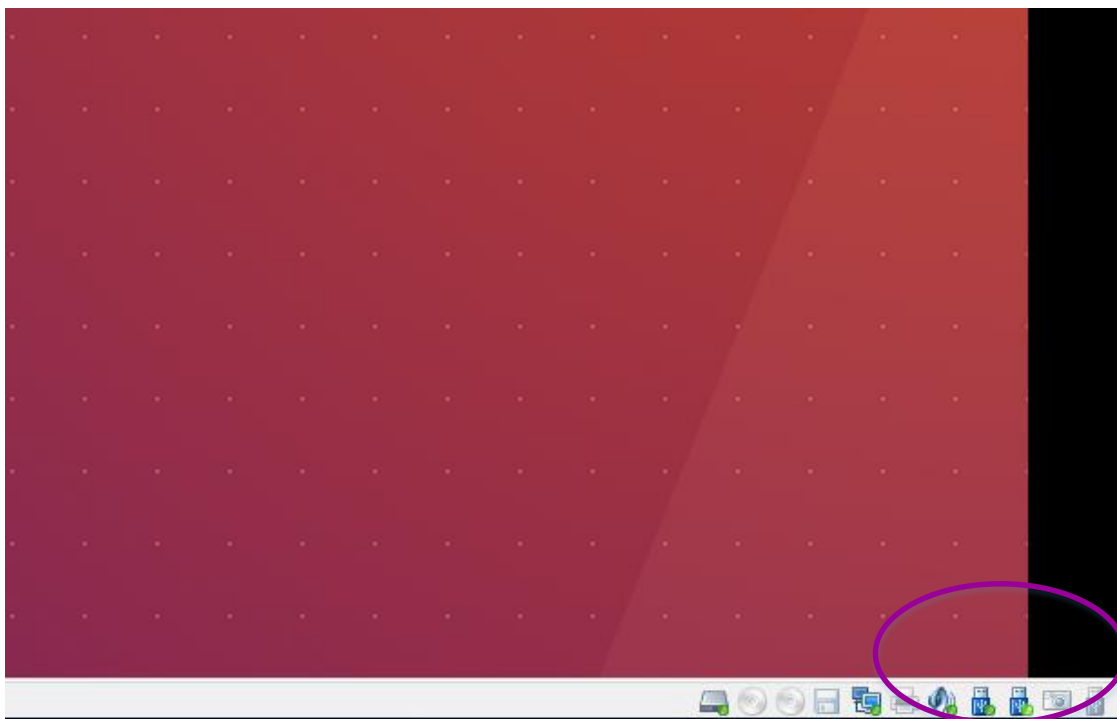


图 4-10 虚拟机 Linux 系统识别 USB 图标

在下一章将介绍利用该 JTAG 调试器如何使用 GDB 软件对蜂鸟 E203 SoC 原型进行程序下载或远程调试。

4.4 FPGA 原型平台 DIY 总结

至此，我们将以上论述的加以总结，为了能够搭建完整的 FPGA 原型平台，用户需要做如下硬件的准备：

- 购买一块蜂鸟 E203 专用 FPGA 开发板
- 购买一块蜂鸟 E203 专用 JTAG 调试器

用户需要做如下软件的准备：

- 推荐安装 VMware 虚拟机且安装 Linux 操作系统于虚拟机中
- 安装 Xilinx 的 Vivado 软件

在下一章将介绍如何使用烧录后的 FPGA 原型平台运行真正的软件示例。

5 运行和调试软件示例

本章将介绍如何使用烧录后的 FPGA 原型平台运行真正的软件示例。

5.1 HBird-E-SDK 简介

为了让用户能够轻松的使用起蜂鸟 E203 内核开发软件，E203 开源项目也配套了一个软件开发套件（Software Development Kit，SDK）。为了方便用户理解，本文将此 SDK 称之为“HBird-E-SDK”。

蜂鸟 E203 专门维护一个独立的 Github 仓库（<https://github.com/SI-RISCV/hbird-e-sdk>）作为管理和维护 HBird-E-SDK，并且在中文书籍《RISC-V 架构与嵌入式开发入门指南》的第 11 章中进行了深入浅出的系统讲解。感兴趣的用户可以自行搜索此书。

HBird-E-SDK 并不是一个软件，它本质上是由一些 Makefile、板级支持包（Board Support Package，BSP）、脚本和软件示例组成的一套开发环境。HBird-E-SDK 基于 Linux 平台，使用标准的 RISC-V GNU 工具链对程序进行编译，使用 OpenOCD+GDB 将程序下载到硬件平台中并进行调试。HBird-E-SDK 主要包含如下两个方面的内容：

- (1) 板级支持包（Board Support Package，BSP）。
- (2) 若干软件示例。

5.1.1 HBird-E-SDK 代码结构

HBird-E-SDK 平台（<https://github.com/SI-RISCV/hbird-e-sdk>）的代码结构如下：

```
hbird-e-sdk          // 存放 hbird-e-sdk 的目录
|----bsp             // 存放板级支持包 (Board Support Package) 的目录
|----hbird-e200      // 存放蜂鸟 E203 配套 SoC 的 BSP 文件
|----env             // 存放一些基本的支持性文件
|----drivers         // 存放底层设备驱动文件
|----include         // 存放一些头文件
|----stubs           // 存放移植 newlib 的底层桩函数
|----tools           // 存放一些工具脚本文件
|----software        // 存放示例程序的源代码
|----hello_world     // Hello World 示例程序
|----demo_gpio       // GPIO 和中断的示例程序
|----demo_iasm       // 内嵌汇编示例程序
|----dhrystone       // Dhrystone 跑分程序
|----coremark        // Coremark 跑分程序
|----FreeRTOSv9.0.0  // FreeRTOS 示例程序
|----work            // 存放工具链的目录
|----Makefile        // 主 Makefile 文件
```


各个主要的目录简述如下。

- software 目录主要用于存放软件示例，包括基本的 hello_world 示例、demo_gpio 示例、demo_iasm 示例、dhrystone 跑分程序、CoreMark 跑分程序和 FreeRTOS 示例程序。每个示例均有单独的文件夹，包含了各自的源代码、Makefile 和编译选项（在 Makefile 中指定）等。
- bsp/hbird-e200/drivers 目录主要用于存放驱动程序代码，譬如 PLIC 模块的底层驱动函数和代码。
- bsp/hbird-e200/include 目录主要用于存放包含 SoC 中外设模块的寄存器地址等参数的头文件。
- bsp/hbird-e200/stubs 目录主要用于存放一些移植 Newlib 所需的底层桩函数的具体实现。
- bsp/hbird-e200/env 目录主要用于存放一些基本的支持性文件，简述如下：
 - board.h: 定义了开发板上管脚或者按键相关的宏定义。
 - platform.h: 定义了 SoC 平台相关的宏定义。
 - common.mk: 调用 GCC 进行编译的 Makefile 脚本，也会指定编译相关的选项。
 - encoding.h: 存放编码和常数的宏定义。
 - entry.S: 异常和中断入口函数。
 - init.c: 系统上电初始化函数。
 - link_flash.lds: 将程序存放在 Flash 中，上电后上载至 ITCM 中进行执行的链接脚本。
 - link_flashxip.lds: 将程序存放在 Flash 中直接进行执行的链接脚本。
 - openocd_hbird.cfg: 使用蜂鸟 JTAG 调试器的 OpenOCD 配置文件。
- start.S: 系统上电启动的引导程序。

在中文书籍《RISC-V 架构与嵌入式开发入门指南》第 11 章中对 HBird-E-SDK 进行了深入浅出的系统讲解。感兴趣的用户可以自行搜索此书。

5.2 使用 HBird-E-SDK 开发和运行示例程序

E203 开源项目提供一个典型的示例程序 demo_gpio 可运行于前文中介绍的 FPGA 开发板上（烧写了蜂鸟 E203 开源 SoC），使用 HBird-E-SDK 平台按照如下步骤可以运行。

```
// 步骤一：准备好自己的电脑环境，可以在公司的服务器环境中运行，如果是个人用户，推荐如下配置：  
（1）使用 VMware 虚拟机在个人电脑上安装虚拟的 Linux 操作系统。  
（2）由于 Linux 操作系统的版本众多，推荐使用 Ubuntu 16.04 版本的 Linux 操作系统  
有关如何安装 VMware 以及 Ubuntu 操作系统本文不做介绍，有关 Linux 的基本使用本文  
也不做介绍，请用户自行查阅资料学习。
```

```
// 步骤二：将 HBird-E-SDK 项目下载到本机 Linux 环境中，使用如下命令：
```

```
git clone https://github.com/SI-RISCV/hbird-e-sdk  
// 经过此步骤将项目克隆下来，本机上即可具有如前文所述完整的 hbird-e-sdk 目  
// 录文件夹，假设该目录为 <your_sdk_dir>，后文将使用该缩写指代。
```

```
// 步骤三：由于编译软件程序需要使用到 GNU 工具链，假设使用完整的 riscv-tools 来自己编译 GNU 工具链则费  
时费力，因此本文档推荐使用预先已经编译好的 GCC 工具链。我们已经将工具链上传至网盘，网盘具体地址记载于  
hbird-e-sdk 项目( https://github.com/SI-RISCV/hbird-e-sdk )的 prebuilt_tools 目录下的 README
```

中，用户可以在网盘中的“RISC-V Software Tools/RISC-V_GCC_201801_Linux”目录下载压缩包 `gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz` 和 `gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz`，然后按照如下步骤解压使用(注意：上述链接网盘上的工具链可能会不断更新，用户请注意自行判断使用最新日期的版本，下列步骤仅为特定版本的示例)。

```
cp gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz ~/
cp gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz ~/
//将两个压缩包均拷贝到用户的根目录下

cd ~/
tar -xzf gnu-mcu-eclipse-riscv-none-gcc-7.2.0-2-20180111-2230-centos64.tgz
tar -xzf gnu-mcu-eclipse-openocd-0.10.0-6-20180112-1448-centos64.tgz
// 进入根目录并解压上述两个压缩包，解压后可以看到一个生成的 gnu-mcu-eclipse 文件夹

cd <your_sdk_dir>
// 进入 hbird-e-sdk 目录文件夹
mkdir -p work/build/openocd/prefix
// 在 hbird-e-sdk 目录下创建上述这个 prefix 目录
cd work/build/openocd/prefix
// 进入到 prefix 该目录

ln -s ~/gnu-mcu-eclipse/openocd/0.10.0-6-20180112-1448/bin bin
// 将用户根目录下解压的 OpenOCD 目录下的 bin 目录作为软链接链接到该 prefix 目录下

cd <your_sdk_dir>
// 再次进入到 hbird-e-sdk 目录文件夹
mkdir -p work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix/
// 在 hbird-e-sdk 目录下创建上述这个 prefix 目录
cd work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix
// 进入到 prefix 该目录

ln -s ~/gnu-mcu-eclipse/riscv-none-gcc/7.2.0-2-20180111-2230/bin bin
// 将用户根目录下解压的 GNU Toolchain 目录下的 bin 目录作为软链接链接到
// 该 prefix 目录下
```

注意：此步骤完成工具链的安装之后，后续开发程序示例无需重复执行此步骤。

// 步骤四：按照第 4 章中所述方法，准备好蜂鸟 E203 专用 FPGA 开发板，并将 bitstream 文件或者 mcs 文件烧录至 FPGA 中待命，且用 JTAG 调试器将 FPGA 开发板与主机 PC 连接，并确保 JTAG 调试器的 USB 接口被虚拟机 Linux 系统正确识别。

// 步骤五：编译 demo_gpio 示例程序，使用如下命令：

```
cd <your_sdk_dir>
// 进入 hbird-e-sdk 目录文件夹

make dasm PROGRAM=demo_gpio NANO_PFLOAT=0
//注意：由于 Demo_GPIO 程序的 printf 函数不需要输出浮点数，上述选项 NANO_PFLOAT=0 指明 newlib-nano 的 printf 函数无需支持浮点数，请参见《RISC-V 架构与嵌入式开发快速入门》第 11 章了解相关信息。
//注意：此处没有指定 Makefile 中的 DOWNLOAD 选项，则默认采用“将程序从 Flash 上载至 ITCM 进行执行的方式”进行编译，请参见《RISC-V 架构与嵌入式开发快速入门》第 11 章了解相关信息。
```

// 步骤六：将编译好的 demo_gpio 程序下载至 FPGA 原型开发板中，使用如下命令：

```
cd <your_sdk_dir>
```



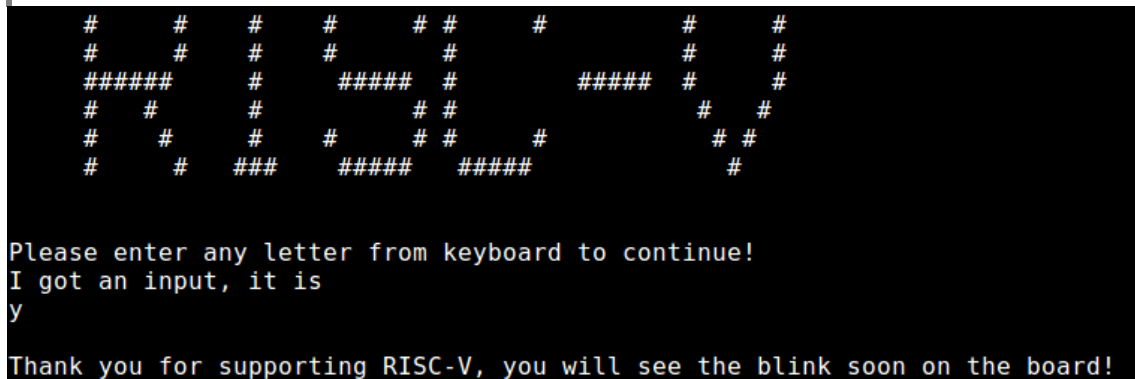
```
// 进入 hbird-e-sdk 目录文件夹

make upload PROGRAM=demo_gpio

// 步骤七：在 FPGA 原型平台上运行 demo_gpio 程序：

// 由于 demo_gpio 示例程序将通过 UART 打印一个字符串到主机 PC 的显示屏上。
// 因此需要先将串口显示终端准备好，打开另外一个 Ubuntu 的命令行终端，使用如下命令。
sudo screen /dev/ttyUSB1 115200
// 该命令将设备 ttyUSB1 设置为串口显示的来源，波特率为 115200
// 若该命令执行成功的话，Ubuntu 的该命令行终端将被锁定，用于显示串口发送的字符。
// 若该命令无法执行成功，请确保已按照第 4.3 节中所述方法将 USB 的权限设置正确。

// 将主机 PC 的串口显示终端准备好之后，则仅需按 FPGA 原型开发板上的 RESET 按键
// 即可。
按 FPGA 开发板上的 RESET 按键，则处理器复位开始执行 demo_gpio 程序，并将 Log 字符打印至主机 PC 的串口显示终端上，如图 5-1 所示。然后用户可以输入任意字符（譬如字母 y），程序继续运行，开发板上将会以固定频率进行闪灯。
```



```
# # # # # # # #
# # # # # # # #
##### # ##### # #
# # # # # # # #
# # # # # # # #
# # ### ##### ##### #

Please enter any letter from keyboard to continue!
I got an input, it is
y

Thank you for supporting RISC-V, you will see the blink soon on the board!
```

图 5-1 运行 Demo_GPIO 示例后于主机串口终端上显示信息

5.3 使用 GDB 和 OpenOCD 调试示例程序

GDB (GNU Project Debugger)，是 GNU 工具链中的调试软件。GDB 是一款应用非常广泛的调试工具，能够用于调试 C、C++、Ada 等等各种语言编写的程序，它提供如下功能：

- 下载或者启动程序
- 通过设定各种特定条件来停止程序
- 查看处理器的运行状态，包括通用寄存器的值，内存地址的值等
- 查看程序的状态，包括变量的值，函数的状态等
- 改变处理器的运行状态，包括通用寄存器的值，内存地址的值等
- 改变程序的状态，包括变量的值，函数的状态等

GDB 可以用于在主机 PC 的 Linux 系统中调试运行的程序，同时也能用于调试嵌入式硬件，在嵌入式硬件的环境中，由于资源有限，一般的嵌入式目标硬件上无法直接构建 GDB 的调试环境（譬如显示屏和 Linux 系统等），这时可以通过 GDB+GdbServer 的方式进行远程（remote）调试，通常而言 GdbServer 在目标硬件上运行，而 GDB 则在主机 PC 上运行。

为了能够支持 GDB 对其进行调试，蜂鸟 E203 使用 OpenOCD 作为其 GdbServer 与 GDB 进行配合。OpenOCD (Open On-Chip Debugger) 是一款开源的免费调试软件，由社区共同维护，由于其开放开源的特点，众多的公司和个人使用其作为调试软件，支持大多数主流的 MCU 和硬件开发板。为了能够完全支持 GDB 的功能，在使用 GCC 对源代码进行编译的时候，需要使用 -g 选项，例如：`gcc -g -o hello hello.c`。该选项会将调试所需信息加入编译所得的可执行程序中，因此该选项会增大可执行程序的大小，因此在正式发布的版本中通常不使用该选项。

GDB 虽然可以使用一些前端工具实现图形化界面，但是更常见的是使用命令行直接对其进行操作。常用的 GDB 命令介绍以及如何 GDB 和 OpenOCD 对蜂鸟 E203 内核进行调试的详细步骤，请参见中文书籍《RISC-V 架构与嵌入式开发快速入门》第 11 章了解详细信息。

6 运行更多示例程序和 Benchmarks

衡量处理器的一个重要指标便是功耗，另外一个重要指标便是性能。而对于处理器性能的评估，需要依赖跑分程序（Benchmarks）来完成。

在处理器领域的 Benchmarks 非常众多，有某些个人开发的程序，也有某些标准组织，或者商业公司开发的 Benchmarks，本文在此不加以一一枚举。在嵌入式处理器领域最为知名和常见的 Benchmarks 为 Dhrystone 和 CoreMark。

Dhrystone 和 CoreMark 和更多其他的示例程序的详细介绍，以及如何在 HBird-E-SDK 平台运行的详细步骤，请参见中文书籍《RISC-V 架构与嵌入式开发快速入门》第 12 章了解详细信息。

7 移植和运行 FreeRTOS

FreeRTOS 是著名的开源实时操作系统（RTOS），FreeRTOS 完全免费，具有源码公开、可移植、可裁剪、任务调度灵活等特点，可以方便地移植到各种 MCU 上运行。有关 FreeRTOS 的详细介绍，以及如何在 HBird-E-SDK 平台运行的详细步骤，请参见《蜂鸟 E203 移植 FreeRTOS》。

8 Windows IDE 开发工具

一款高效易用的集成开发环境（Integrated Development Environment，IDE）对于任何 MCU 都显得非常重要，软件开发人员需要借助 IDE 进行实际的项目开发与调试。ARM 架构的 MCU 目前占据了很大的市场份额，ARM 的商业 IDE 软件 Keil 也非常深入人心，很多嵌入式软件工程师均对其非常熟悉是商业 IDE 软件（譬如 Keil）存在着授权以及收费的问题，各大 MCU 厂商也会推出自己的免费 IDE 供用户使用，譬如瑞萨的 e2studio 和 NXP 的 LPCXpresso 等，这些 IDE 均是基于开源的 Eclipse 框架，Eclipse 几乎成了开源免费 MCU IDE 的主流选择。

Eclipse 平台采用开放式源代码模式运作，并提供公共许可证（提供免费源代码）以及全球发布权利。Eclipse 本身只是一个框架平台，除了 Eclipse 平台的运行时内核之外，其所有功能均位于不同的插件中。开发人员既可通过 Eclipse 项目的不同插件来扩展平台功能，也可利用其他开发人员提供的插件。一个插件可以插入另一个插件，从而实现最大程度的集成。

Eclipse IDE 平台具备以下几方面的优势：

■ 社区规模大

Eclipse 自 2001 年推出以来，已形成大规模社区，这为设计人员提供了许多资源，包括图书、教程和网站等，以帮助他们利用 Eclipse 平台与工具提高工作效率。Eclipse 平台和相关项目、插件等都能直接从 eclipse.org 网站下载获得。

■ 持续改进

Eclipse 的开放式源代码平台帮助开发人员持续充分发挥大规模资源的优势。Eclipse 在以下多个项目上不断改进。

- 平台项目——侧重于 Eclipse 本身。
- CDT 项目——侧重于 C/C++ 语言开发工具。
- PDE 项目——侧重于插件开发环境。

■ 源码开源

设计人员始终能获得源代码，总能修正工具的错误，它能帮助设计人员节省时间，自主控制开发工作。

■ 兼容性

Eclipse 平台采用 Java 语言编写，可在 Windows 与 Linux 等多种开发工作站上使用。开放式源代码工具支持多种语言、多种平台以及多种厂商环境。

■ 可扩展性

Eclipse 采用开放式、可扩展架构，它能够与 ClearCase、SlickEdit、Rational Rose 以及其他统一建模语言（UML）套件等第三方扩展协同工作。此外，它还能与各种图形用户接口（GUI）编辑器协同工作，并支持各种插件。

请参见《RISC-V 架构与嵌入式开发快速入门》书籍第 13 章，其中详细介绍了如何使用基于 MCU Eclipse IDE 的 Windows 开发调试环境对蜂鸟 E203 内核进行软件开发和调试。