

Silicon Integrated

蜂鸟 E200 系列 Core & SoC 原型

快速使用说明

Content

1 PREFACE	3
1.1 REVISION HISTORY	3
2 运行 VERILOG 仿真测试	4
2.1 E200 项目的代码层次结构.....	4
2.2 E200 项目的测试用例（SELF-CHECK TESTCASE）	5
2.2.1 <i>riscv-tests</i> 自测试用例.....	6
2.2.2 编译 ISA 自测试用例.....	6
2.3 E200 项目的测试平台（TESTBENCH）	10
2.4 在 VERILOG TESTBENCH 中运行测试用例	11
3 实现 SOC 平台	14
3.1 FREEDOM E310 SoC 简介	14
3.2 SIRV-E200-SoC 简介	16
3.2.1 SIRV-E200-SoC 组成结构.....	16
3.2.2 SIRV-E200-SoC 代码结构.....	21
3.2.3 SIRV-E200-SoC 自定义总线.....	26
3.3 SIRV-E200-SoC FPGA 原型平台	32
3.3.1 FPGA 开发板	32
3.3.2 生成 mcs 文件烧写 FPGA	37
3.3.3 JTAG 调试器	39
3.3.4 FPGA 原型平台 DIY 总结.....	43
4 运行和调试软件 DEMO	45
4.1 FREEDOM-E-SDK 平台简介	45
4.2 SIRV-E-SDK 简介	46
4.2.1 SIRV-E-SDK 代码结构.....	47
4.3 使用 SIRV-E-SDK 运行示例程序	48
4.4 使用 GDB 和 OPENOCD 调试示例程序	52
4.5 其他软件工具	56
5 运行 BENCHMARKS.....	58
5.1 BENCHMARKS 简介	58
5.2 运行 DHRYSTONE BENCHMARK.....	58
5.3 运行 COREMARK BENCHMARK	62
5.4 总结与比较	64

1 Preface

1.1 Revision History

Date	Version	Author	Change Summary
Sep 29, 2017	0.1	Bob Hu	Initial version

2 运行 Verilog 仿真测试

本章将介绍开源 E200 平台如何运行 Verilog 仿真测试平台。

2.1 E200 项目的代码层次结构

The screenshot shows the GitHub repository page for the project 'SI-RISCV/e200_opensource'. The repository has 56 commits, 1 branch, 0 releases, and 4 contributors. The README.md file is visible at the bottom.

File	Description	Date
README.md	1st version	Jul 27, 2017
.gitignore	upload the fpga dir	Aug 27, 2017
vsim	upload the fpga dir	Aug 27, 2017
tb	upload the fpga dir	Aug 27, 2017
riscv-tools	upload the benchmark	Aug 27, 2017
fpga	add _SUPPORT_MCYCLE_MINSTRET	Aug 28, 2017
doc	update the README for doc directory	Aug 29, 2017

图 2-1 蜂鸟 E200 Github 网址

蜂鸟 E200 的平台托管于著名网站 Github。E200 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，主要为目前的若干 Core 的 Verilog RTL 源代码，譬如 E203，和配套的 SoC 组件文件。每个 Core 的有一个专属的文件夹目录存放期 Verilog RTL 源代码，有关 E201/E203/E205 核的区别请参见文档《Hummingbird_E200_Series_Core_SoC_Introduction.pdf》。配套的 SoC 的信息和代码分析请参见下一章。

2.2 E200 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, 0xffffffffffff80000000, 0xffffffffffff80000000, 0x00000000 );
TEST_RR_OP( 7, add, 0xffffffffffff7fff8000, 0xffffffffffff80000000, 0xffffffffffff8000 );

TEST_RR_OP( 8, add, 0x0000000000007fff, 0x0000000000000000, 0x00000000000007fff );
TEST_RR_OP( 9, add, 0x000000007fffffff, 0x000000007fffffff, 0x0000000000000000 );
TEST_RR_OP( 10, add, 0x0000000080007ffe, 0x000000007fffff, 0x00000000000007fff );

TEST_RR_OP( 11, add, 0xffffffffffff80007fff, 0xffffffffffff80000000, 0x00000000000007fff );
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, 0xfffffffffffffffffe, 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 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 96 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

用户如果修改了汇编程序的源代码需要重新编译，在以下文件目录下可运行以下命令，编译出的文件将被重新生成在 e200_opensource/riscv-tools/riscv-tests/isa/generated 目录中。

```

// 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：为了防止后续步骤中出现错误，预先最好将很多工具包先安装在 Ubuntu 16.04 系统中，使用如下命令：

```

sudo apt-get install autoconf automake autotools-dev curl device-tree-compiler
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo
gperf libtool patchutils bc zlib1g-dev

```

// 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：由于编译汇编程序需要使用到 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，因此本文推荐使用预先已经编译好的 GCC 工具链。用户可以在链接
http://pan.baidu.com/s/1eSD0COM 下载压缩包 riscv32_unkown_elf_gcc6.1.0.tar.gz，然后按照如下
步骤解压使用。

```

```

cp riscv32_unkown_elf_gcc6.1.0.tar.gz ~/
    // 将压缩包拷贝到用户的根目录下

cd ~/
tar -xzvf riscv32_unkown_elf_gcc6.1.0.tar.gz
    // 进入根目录并解压该压缩包

cd <your_e200_dir>/
    // 进入到 e200_opensource 的目录文件夹

mkdir -p ./riscv-tools/prebuilt_tools/prefix/bin
    // 在 e200_opensource 目录下创建上述这个 bin 目录

cd ./riscv-tools/prebuilt_tools/prefix/bin/
    // 进入到这个新建的 bin 目录下

ln -s ~/riscv32_unkown_elf_gcc6.1.0/bin/*
    // 将用户根目录下解压压缩包中 bin 目录下的所有可执行文件作为软链接链接到
    // 该 ./riscv-tools/prebuilt_tools/prefix/bin/ 目录下

// 步骤五：在接下来的步骤中可能会出现如下错误：
"Syntax error:Bad fd number"
// 这个错误可能是由于在 Ubuntu 16.04 中，/bin/sh 被链接到了 /bin/dash 而不
// 是 /bin/bash。如果果真如此，可以用以下命令进行修改：
sudo mv /bin/sh /bin/sh.orig
sudo ln -s /bin/bash /bin/sh

// 步骤六：使用如下命令编译出 Spike ( 指令模拟器 ) 和 riscv-tests：
cd <your_e200_dir>/riscv-tools
    // 进入到 e200_opensource 目录下的 riscv-tools 文件夹

./build-e200-spike-rvtests.sh
    // 运行该脚本将编译出指令模拟器 Spike 和 riscv-tests。
    // 如果运行该步骤没有出现错误，那么一个可执行文件 spike 将被生成在
    // <your_e200_dir>/riscv-tools/prebuilt_tools/prefix/bin/ 目录下，
    // 一些相关的库文件也
    // 将被生成在 <your_e200_dir>/riscv-tools/prebuilt_tools/prefix/
    // 目录下。
    // 可以通过在 <your_e200_dir>/riscv-tools/prebuilt_tools/prefix/bin/
    // 目录下运行“spike -version”来确认此 spike 是否能够被正确执行。

// 步骤七：在以下文件目录下可运行以下命令，编译出的文件将被重新生成在 e200_opensource/
riscv-tools/riscv-tests/isa/generated 目录中。
e200_opensource
    |---riscv-tools
        |---riscv-tests
            |---isa          // 在该目录下运行命令 source regen.sh
            |--- regen.sh

```

2.3 E200 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

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 E200 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。
// 如果需要为其他型号的 core 进行设置，则仅需更改 CORE 的参数。譬如假设设置需要编
// 译的 Core 型号为 e225fd，则使用命令 make install CORE=e225fd

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 CORE=e203
    // 注意：这使用 e200_opensource/riscv-tools/riscv-tests/isa/generated
    // 目录中 E203 Core 的 testcases，逐个的运行 testcase。
    // 如果需要为其他型号的 core 运行回归测试，则仅需更改 CORE 的参数。譬如假设设置需
    // 要运行回归测试的 Core 型号为 e225fd，则使用命令 make regress CORE=e225fd

// 步骤六：查看回归测试结果：
make regress_collect CORE=e203
    // 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。
    // 如果需要为其他型号的 core 收集回归测试结果，则仅需更改 CORE 的参数。譬如假设设
    // 置需要收集回归测试结果的 Core 型号为 e225fd，则使用命令
    // make regress_collect CORE=e225fd

```

```

# 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 simulation tool
SIM_TOOL      := iverilog # this is a free solution here to use iverilog to compile the code

SIM_OPTIONS   := #To-ADD: to add the simulation tool options
SIM_OPTIONS   := -o vvp.exec -I "${VSRP_DIR}/core/" -D FPGA_SOURCE=1 -g2005 # this is a free
solution here to use iverilog to compile the code

SIM_EXEC      := #To-ADD: to add the simulation executable
#SIM_EXEC     := vvp ${RUN_DIR}/vvp.exec -none # The free vvp is toooooo slow to run, so just
comment it out, and replaced with the fake way below
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_PREFIX    := #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.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 实现 SoC 平台

对于一个处理器核，还需要配套的 SoC 才能具备完整的功能，本章将介绍一款开源的 Freedom E310 SoC，并介绍蜂鸟 E200 开源处理器如何配套使用该 SoC 进行工作。

3.1 Freedom E310 SoC 简介

Freedom E300 平台由 SiFive 公司推出（SiFive 公司是由伯克利大学发明 RISC-V 架构的几个主要发起人创办的商业公司）。

Freedom Everywhere E310-G000（简称 Freedom E310）是使用 Freedom Everywhere E300 平台配置出的一款特定配置的 SoC，并且 SiFive 将此 SoC 的代码完全开源，用户可以在 Github 上（<https://github.com/sifive/freedom>）下载其源代码。Freedom E310 SoC 基于 Rocket Core，架构配置为 RV32IMAC 架构，配备 16KB 的指令 Cache 与 16KB 的数据 SRAM（Scratchpad），硬件乘除法器，调试（Debug）模块，丰富的外设如 PWM、UART、SPI 等，其结构框图如图 3-1 所示。

基于 Freedom E310 SoC（增加一些模拟电路模块后），SiFive 公司使用 TSMC CL018G 180nm 工艺成功流片（Tapeout）生产出实际芯片，能够运行到 320MHz 以上的主频。并且，基于此芯片 SiFive 公司开发制造了一款信用卡大小的硬件开发板 HiFive1，如图 3-2 所示。HiFive1 开发板是目前市场上最早在售硬件开发板，已经被很多 RISC-V 的爱好者与公司使用。据称开售后在短短几个月之内便在全球范围内售出数千块之多，而中国大陆是销量仅次于美国的第二大畅销地。

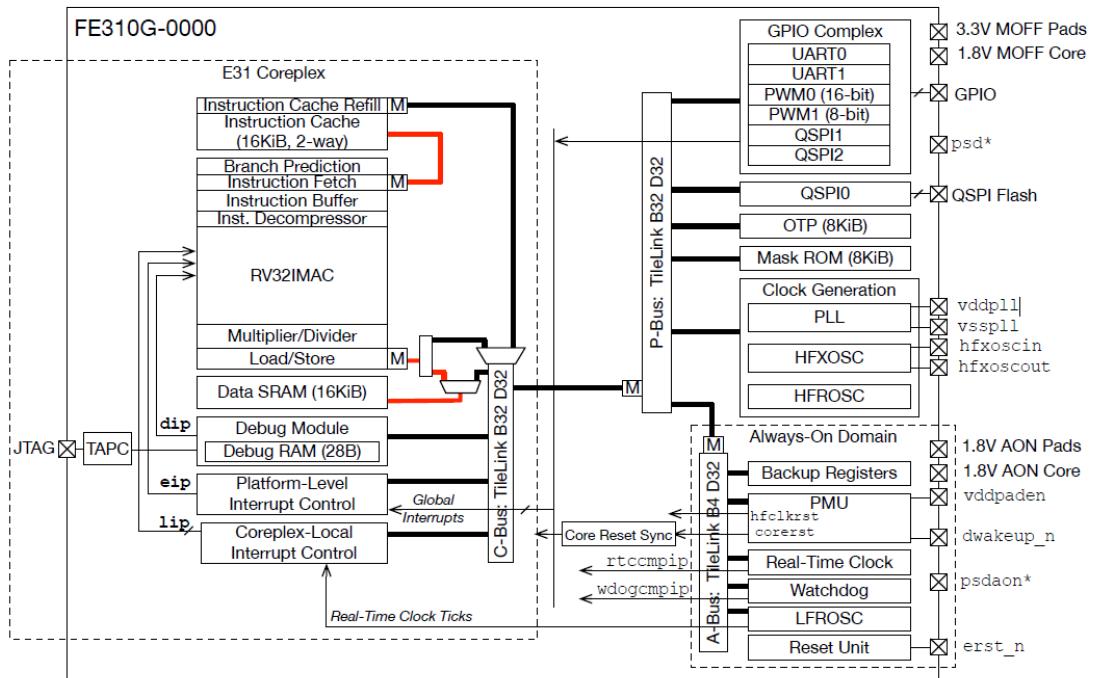


图 3-1 Freedom 310 SoC 结构图

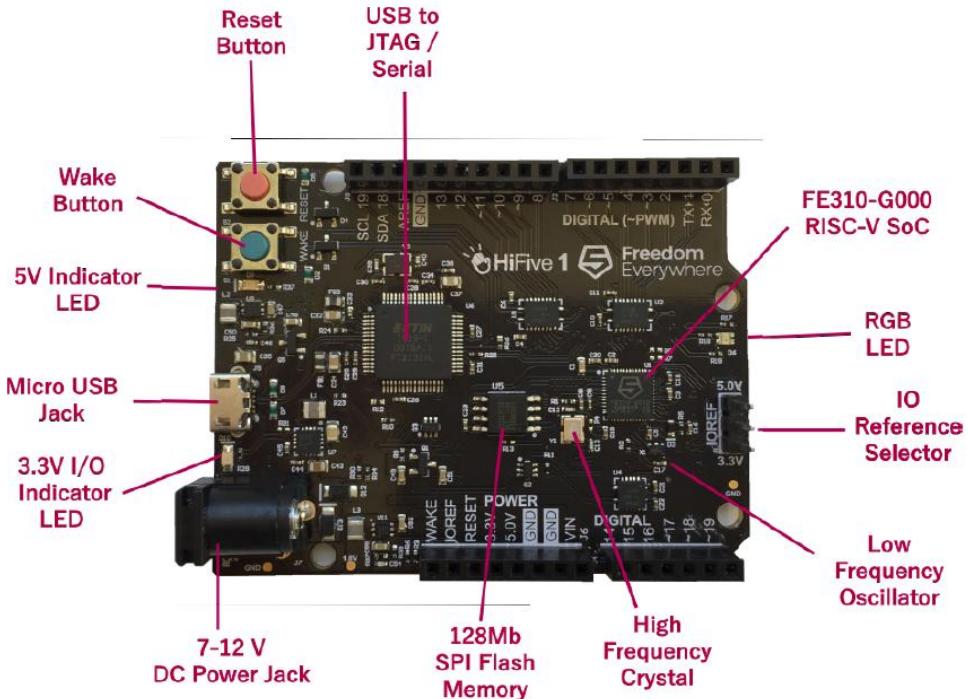


图 3-2 HIFIVE 开发板结构图

3.2 SIRV-E200-SoC 简介

3.2.1 SIRV-E200-SoC 组成结构

Freedom E310 是一款非常优秀的 SoC，非常感谢 SiFive 公司将其开源。E200 项目也是以 Freedom E310 SoC 为蓝本，在其基础上做了如下主要修改：

- 将其中的 Rocket Core 替换成为蜂鸟 E200 处理器核系列
- 将其使用的 TileLink 总线（一种伯克利自定义的总线）替换成为蜂鸟 E200 处理器使用的 ICB 总线（Internal Chip Bus 为 E200 项目自定义内部总线，关于此总线介绍见后文）。
注意：虽然对原 SoC 的总线进行了修改，但是总线的地址分配表仍然完全与原始的 Freedom E310 SoC 一致，如表 3-1 所示。因此从软件的角度来看，可以认为修改后的 SoC（虽然使用的是蜂鸟 E200 处理器核系列）与原始的 Freedom E310 几乎完全兼容，软件可以很方便移植运行。
- 保留了所有的其他系统 IP，譬如 UART，SPI，PWM 等等，但是直接使用其可综合的 Verilog 代码，无需使用 Chisel 进行编译转换。

表 3-1 SIRV-E200-SoC 地址分配表

总线分组	组件	地址区间	描述
Core 直属	CLINT	0x0200_0000 ~ 0x0200_FFFF	Core Local Interrupt Controller 模块寄存器地址区间
	PLIC	0x0C00_0000 ~ 0x0FFF_FFFF	Platform Level Interrupt Controller 模块寄存器地址区间
	ITCM	0x8000_0000 ~ 取决于 ITCM 配置大小	ITCM 地址区间
	DTCM	0x9000_0000 ~ 取决于 DTCM 配置大小	DTCM 地址区间
系统存储总线接口	Debug Module	0x0000_0000 ~ 0x0000_0FFF	注意：Debug Module 主要用于调试器使用，普通软件程序不会使用此区间
	Mask-ROM	0x0000_1000 ~ 0x0000_1FFF	注意：e200 项目由于是 FPGA 原型，因此 Mask-ROM 代码为一行为模型
	Off-Chip QSPIO Flash Read	0x2000_0000 ~ 0x3FFF_FFFF	详见下一节

	On-Chip OTI Read	0x0002_0000 ~ 0x0003_FFFF	注意: e200 项目由于是 FPGA 原型, 并未实际提供 OTP 模块, 仅分配了此地 址连接一空模块。
	外部存储 (sysmem)	0x8000_0000 ~ 0xFFFF_FFFF (如果配置了 ITCM 和 DTCM, 在此需 要去除其二者空间)	详见下一节 注意: E200 并未使用此接 口, 将其悬空接零
私有设备总线接口 (总区间为 0x1000_0000 ~ 0x1FFF_FFFF)	Always-On	0x1000_0000 ~ 0x1000_7FFF	Always-on 模块包含 PMU, RTC, WatchDog。详见下 一节。
	PRCI	0x1000_8000 ~ 0x1000_8FFF	Power, Reset, Clock, Interrupts (PRCI) 模块, 注意: e200 项目由于是 FPGA 原型, 并未实际提供 此模块, 仅分配了此地址连 接一空模块。
	OTP	0x1001_0000 ~ 0x1001_0FFF	注意: e200 项目由于是 FPGA 原型, 并未实际提供 此 OTP 模块, 仅分配了此 地址连接一空模块。
	GPIO	0x1001_2000 ~ 0x1001_2FFF	详见下一节
	UART0	0x1001_3000 ~ 0x1001_3FFF	详见下一节
	QSPI0	0x1001_4000 ~ 0x1001_4FFF	详见下一节
	PWM0	0x1001_5000 ~ 0x1001_5FFF	详见下一节
	UART1	0x1002_3000 ~ 0x1002_3FFF	详见下一节
	QSPI1	0x1002_4000 ~ 0x1002_4FFF	详见下一节
	PWM1	0x1002_5000 ~ 0x1002_5FFF	详见下一节
	QSPI2	0x1003_4000 ~ 0x1003_4FFF	详见下一节
	PWM2	0x1003_5000 ~ 0x1003_5FFF	详见下一节
	外部设备 (sysper)	0x1100_0000 ~ 0x11FF_FFFF	详见下一节 注意: E200 并未使用此接 口, 将其悬空接零
快速 IO 接口		0xF000_0000 ~ 0xFFFF_FFFF	注意: E200 并未使用此接 口, 将其悬空接零

为了方便用户理解区别，本文将“修改后的 SoC（使用的是蜂鸟 E200 处理器核系列）”称之为“SIRV-E200-SoC”。SIRV-E200-SoC 结构如图 3-3 所示。

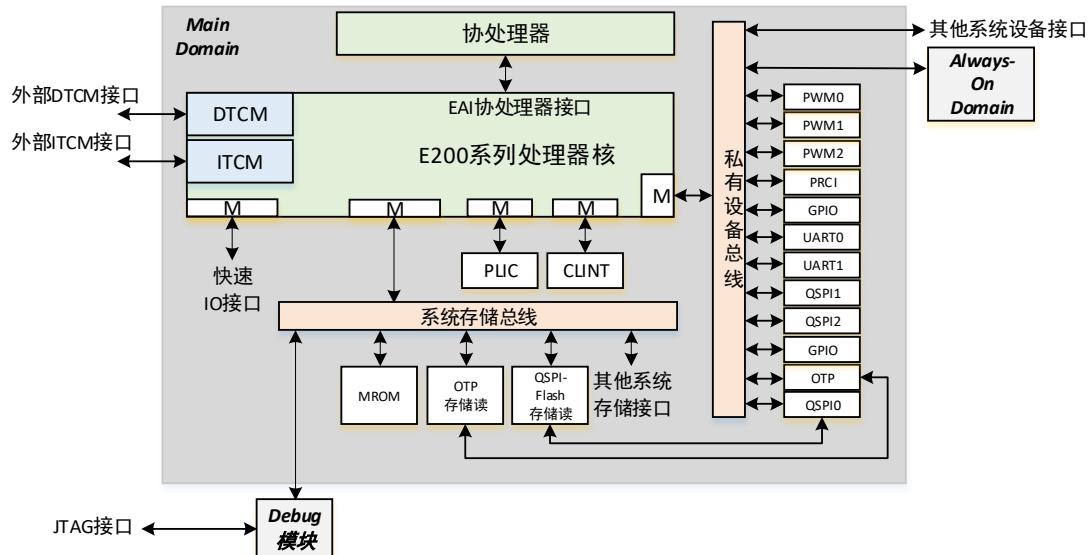


图 3-3 SIRV-E200-SoC 结构图

SIRV-E200-SoC 中主要的模块介绍如下：

■ **Clocks:**

整个 SoC 从时钟域 (Clock Domains) 上分为两个大的部分：主体部分 (Main Domain)，和电源常开部分 (Always-on Domain)。其中的 Main Domain 使用高速的时钟，而 Always-on Domain 使用低速的时钟。

注意：由于 SIRV-E200-SoC 原型是在 FPGA 上实现的原型平台，因此其并没有原 Freedom E310 SoC 文档中提及的 Clock Generation 模块（包含了 ASIC 工艺的 PLL 模块等），而是使用了 FPGA 提供的 Clock 和 Reset 生成 IP，相关信息会在下一节加以论述。

■ **CLINT:**

全称为 Core-Local Interrupt Controller（处理器核局部中断控制器），主要实现 RISC-V 架构手册中规定的标准计时器 (Timer) 和软件中断功能。

■ **PLIC:**

全称为 Platform-Level Interrupt Controller（平台中断控制器），主要实现 RISC-V 架构手册中规定的 PLIC 功能，该 PLIC 能够支持多个中断源，并且每个中断可以配置中断优先级。PLIC 的中断来源如表 3-2 所示，包括 UART、SPI、GPIO 等等。所有的这些中断经过 PLIC 仲裁后，生成一根最终的中断信号通给处理器核作为其外部中断信号。

■ **JTAG:**

标准 (1149.1) JTAG 连接模块用于连接系统外部调试器 (Debugger) 与内部的调试模块 (Debug Module)。

■ **Debug Module:**

调试模块，用于支持外部 JTAG 通过该模块调试处理器核，是的处理器核能够通过 GDB 对其进行交互式调试，譬如设置断点，单步执行等调试功能。

■ **Quad-SPI Flash:**

专用于连接外部 Flash 的 Quad-SPI (QSPI) 接口。指令和数据均可以存储于外部的 Flash 之中，并且该 QSPI 接口还可以被软件配置成为 eXecute-In-Place 模式，在此模式下，Flash 可以被当作一段只读区间直接被当做内存读取。在默认上电之后，QSPI 即处于该模式之下，由于 Flash 掉电不丢失的特性，因此可以将系统的启动程序存放于外部的 Flash 中，然后处理器核通过 eXecute-In-Place 模式的 QSPI 接口直接访问外部 Flash 加载启动程序启动。

■ **GPIO:**

全称为 General Purpose I/O，用于提供一组 32 I/O 的通用输入输出接口。每个 I/O 可用被软件配置为输入或者输出，如果是输出可以设置具体的输出值。每个 I/O 还可以被配置为 IOF (Hardware I/O Functions)，也就是将 I/O 供 SoC 内部的其他模块复用，譬如 SPI, UART, PWM 等等。GPIO 的 32 个 I/O 被 SoC 内部模块的复用分配如表 3-3 所示，其中每个 I/O 均可以供两个内部模块复用，软件可以通过配置每个 I/O 使其选择 IOFO 或者 IOF1 来选择信号来源。另外，每个 GPIO 的 I/O 均作为一个中断源连接到 PLIC 的中断源上。

■ **QSPI:**

除了上述专用于 Flash 的 QSPI 接口之外，SoC 还有两个独立的 QSPI 接口控制器。一个 QSPI 使用四个片选信号 (Chip Selects)，一个 QSPI 使用一个片选信号。两个 QSPI 均使用 GPIO 的 IOF 功能与外界通信。

■ **UART:**

全称为 Universal Asynchronous Receiver-Transmitter (通用异步接收-发射器)。SoC 有两个独立的 UART，两个 UART 均使用 GPIO 的 IOF 功能与外界通信。

■ **PWM:**

全称为 Pulse-Width Modulator (脉宽调节器)。SoC 有三个独立的 PWM，其中两个是 16 比特的精度，另外一个是 8 比特的精度。

■ **WatchDog:**

全称为 Watch Dog Timer (看门狗计数器)，该计数器位于 Always-on Domain 中，因此使用低速时钟进行计数，并且可以通过配置其计数的目标值产生中断。

■ **RTC:**

全称为 Real-Time Counter (实时计数器)，该计数器位于 Always-on Domain 中，因此使用低速时钟进行计数，并且还能产生中断。

■ PMU:

全称为 Power Management Unit (电源管理单元)，用于控制 SoC 的电源管理。整个 SoC 除了 WatchDog、RTC、PMU 等模块处于 Always-on Domain 之外，其他 Main Domain 可以在 PMU 的控制下被置于断电状态以节省功耗，或者重新唤醒等等。

表 3-2 PLIC 的中断分配

Interrupt Number	Source
0	<i>No Interrupt</i>
1	wdogcmp
2	rtccmp
3	uart0
4	uart1
5	qspi0
6	qspi1
7	qspi2
8	gpio0
...	
39	gpio31
40	pwm0cmp0
...	
44	pwm0cmp3
45	pwm1cmp0
...	
48	pwm1cmp3
49	pwm2cmp0
...	
52	pwm2cmp3

表 3-3 GPIO 的接口分配

Pin Number	IOF0	IOF1
0		PWM0_0
1		PWM0_1
2	QSPI1:SS0	PWM0_2
3	QSPI1:SD0/MOSI	PWM0_3
4	QSPI1:SD1/MISO	
5	QSPI1:SCK	
6	QSPI1:SD2	
7	QSPI1:SD3	
8	QSPI1:SS1	
9	QSPI1:SS2	
10	QSPI1:SS3	PWM2_0
11		PWM2_1
12		PWM2_2
13		PWM2_3
14		
15		
16	UART0:RX	
17	UART0:TX	
18		PWM1_1
19		PWM1_0
20		PWM1_2
21		PWM1_3
22		
23		
24	UART1:RX	
25	UART1:TX	
26	QSPI2:SS	
27	QSPI2:SD0/MOSI	
28	QSPI2:SD1/MISO	
29	QSPI2:SCK	
30	QSPI2:SD2	
31	QSPI2:SD3	

以上仅对每个模块功能进行简述，由于上述外设模块重用自 **Freedom E310 SoC** 且软件完全兼容，用户可以参阅 **Freedom E310 SoC** 的技术文档了解其细节。为了方便用户查看，**Freedom E310** 的相关文档也在 **e200_opensource** 文档目录 (https://github.com/SI-RISCV/e200_opensource/doc) 目录下存储了一份，分别为 **SiFive-E300-platform-reference-manual-v1.0.1.pdf** 与 **SiFive-E310-G000-manual-v1.0.1.pdf**，其中有对每个模块的详细功能描述与详细配置寄存器描述供用户参阅。

3.2.2 SIRV-E200-SoC 代码结构

SIRV-E200-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_dtcm_ram.v // DTCM 的 SRAM 模块
|----fab           // 存放总线 bus fabric 的 RTL 代码
|---- sirv_icb1to4_bus.v // 将 1 组 ICB 总线转换成 4 路 ICB 总线
|---- sirv_icb1to8_bus.v // 将 1 组 ICB 总线转换成 8 路 ICB 总线
|---- sirv_icb1to16_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_1cs*.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
|----fpga          // 存放 FPGA 实现的 RTL 代码
|---- clkdivider.v          // 时钟分频电路
|----e203_fpga_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 3-4 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。 sirv_icb1to4_bus.v, sirv_icb1to8_bus.v 或者 sirv_icb1to16_bus.v 实际例化调用了 sirv_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。
注意：虽然使用了 ICB 总线，但是总线的地址分配表仍然完全与原始的 Freedom E310 SoC 一致，如表 3-1 所示。因此从软件的角度来看，SIRV-E200-SoC 与原始的 Freedom E310 完全兼容，软件可以进行无缝移植运行。
- 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 3-4 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 sirv_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 3-5 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 sirv_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 3-6 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 3-7 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 SMIC 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 3-8 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。
- fpga 目录主要用于存放 FPGA 项目所需模块的 Verilog RTL 代码。其中 clkdivider.v 是一个简单地利用寄存器进行时钟分频的模块，分频系数是 256；而 e203_fpga_soc_top.v 是一个简单地顶层 SoC Wrapper 模块，将 e203_subsys_top 进行例化。另外由于前文中提到的 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_fpga_soc_top 顶层模块中将这些 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 3-9 所示。

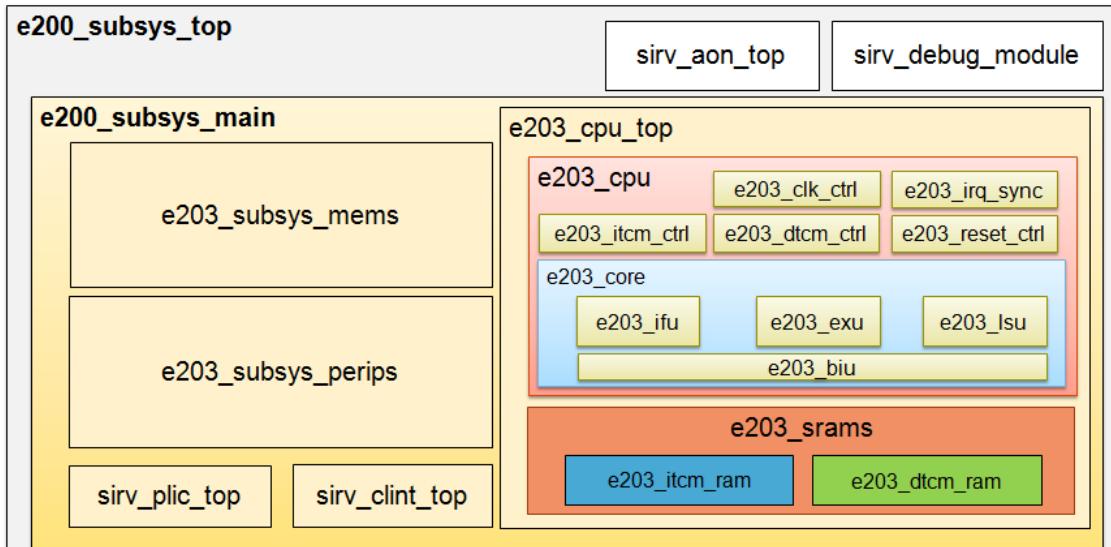


图 3-4 Subsys 模块层次结构图

```

// 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 outstanding
    .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 3-5 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_icblto16_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 3-6 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 3-7 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 3-8 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 3-9 e203_fpga_soc_top 顶层 PER 总线接口的连接

3.2.3 SIRV-E200-SoC 自定义总线

蜂鸟 E200 处理器开发过程中定义了一种自定义总线协议 ICB (Internal Chip Bus)，用于蜂鸟 E200 处理器核内部使用，同时也可作为 SoC 中的总线使用。ICB 总线的初衷是为了能够尽可能地结合 AXI 总线和 AHB 总线的优点，兼具高速性和易用性，它具有如下特性：

- 相比 AXI 和 AHB 而言，ICB 的协议控制更加简单，仅有两个独立的通道，如图 3-10 所示，读和写操作共用地址通道，共用结果返回通道。
- 与 AXI 总线一样采用分离的地址和数据阶段。
- 与 AXI 总线一样采用地址区间寻址，支持任意的主从数目，譬如一主一从，一主多从，多主一从，多主多从等拓扑结构。
- 与 AHB 总线一样每个读或者写操作都会在地址通道上产生地址，而非像 AXI 中只产生起始地址。
- 与 AXI 总线一样支持地址非对齐的数据访问，使用字节掩码 (Write Mask) 来控制部分写操作。
- 与 AXI 总线一样支持多个滞外交易 (Multiple Outstanding Transaction)。
- 与 AHB 总线一样不支持乱序返回乱序完成。反馈通道必须按顺序返回结果。
- 与 AXI 总线一样非常容易添加流水线级数以获得高频的时序。
- 协议非常简单，易于桥接转换成其他总线类型，譬如 AXI，AHB，APB 或者 TileLink 等总线。

对于蜂鸟 E200 处理器核这样的低功耗处理器而言，ICB 总线能够被用于几乎所有的相关场合，包括：作为内部模块之间的接口，SRAM 模块接口，低速设备总线，系统存储总线等等。

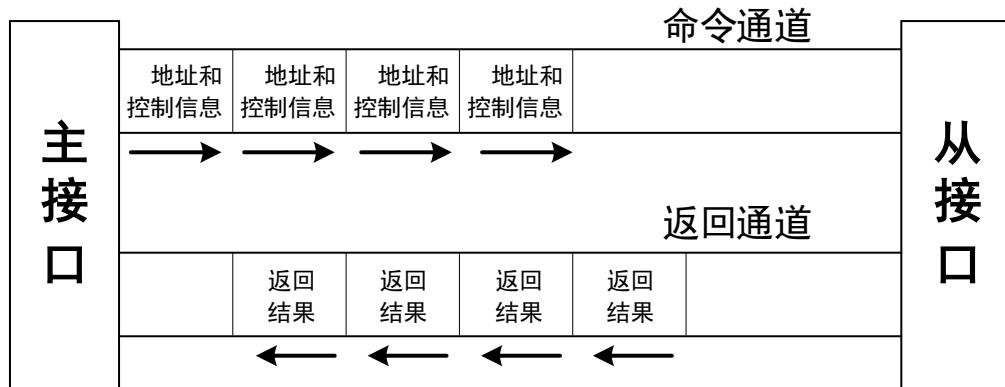


图 3-10 ICB 总线通道结构

3.2.3.1 ICB 总线协议信号

ICB 总线主要包含 2 个通道，如图 3-10 所示。ICB 总线信号列表如表 3-4 所示：

- 命令通道（Command Channel）
Command Channel 主要用于主设备向从设备发起读写请求。
- 返回通道（Response Channel）
Response Channel 主要用于从设备向主设备返回读写结果。

表 3-4 ICB 总线信号

通道	方向	宽度	信号名	介绍
Command Channel	Output	1	icb_cmd_valid	主设备向从设备发送读写请求信号
	Input	1	icb_cmd_ready	从设备向主设备返回读写接受信号
	Output	32	icb_cmd_addr	读写地址
	Output	1	icb_cmd_read	读或是写操作的指示。读则以总线宽度（譬如 32 位）为单位读回一个数据。写则靠字节掩码（icb_cmd_wmask）控制写数据的大小（Size）。
	Output	32	icb_cmd_wdata	写操作的数据，数据的摆放格式与 AXI 协议一致
	Output	4	icb_cmd_wmask	写操作的字节掩码，掩码的摆放格式与 AXI 协议一致
Reponse Channel	Input	1	icb_rsp_valid	从设备向主设备发送读写反馈请求信号
	Output	1	icb_rsp_ready	主设备向从设备返回读写反馈接受信号
	Input	32	icb_rsp_rdata	读反馈的数据，数据的摆放格式与 AXI 协议一致
	Input	1	icb_rsp_err	读或者写反馈的错误标志

3.2.3.2 ICB 总线协议时序

本节将描述 ICB 总线的若干典型时序。

- 如图 3-11 所示：主设备向从设备通过 ICB 的 Command Channel 发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。从设备在同一个周期返回读结果且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

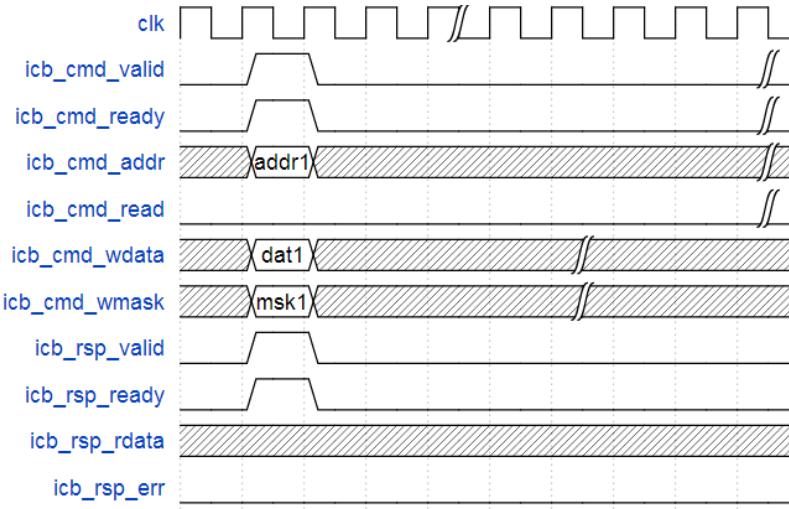


图 3-11 写操作同一周期返回结果

- 如图 3-12 所示：主设备向从设备通过 ICB 的 Command Channel 发送读操作请求（icb_cmd_read 为高），从设备立即接收该请求（icb_cmd_ready 为高）。从设备在下一个周期返回读结果且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

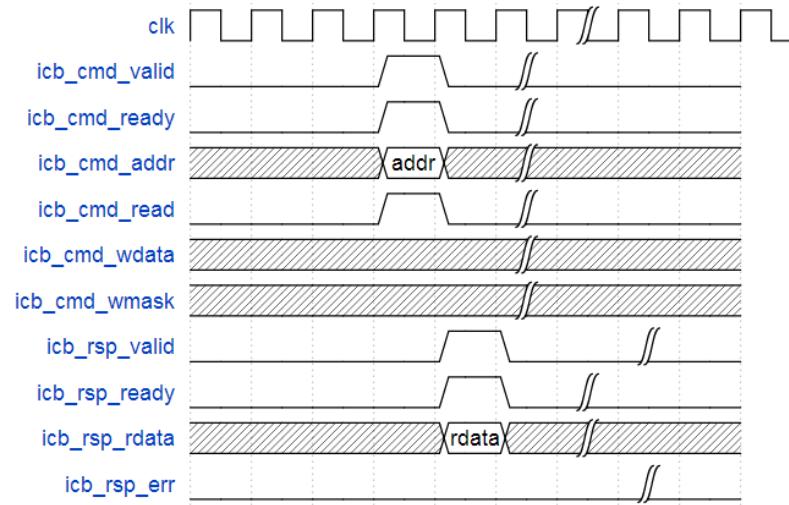


图 3-12 读操作下一周期返回结果

- 如图 3-13 所示：主设备向从设备通过 ICB 的 Command Channel 发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。从设备在下一个周期返回读结果且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

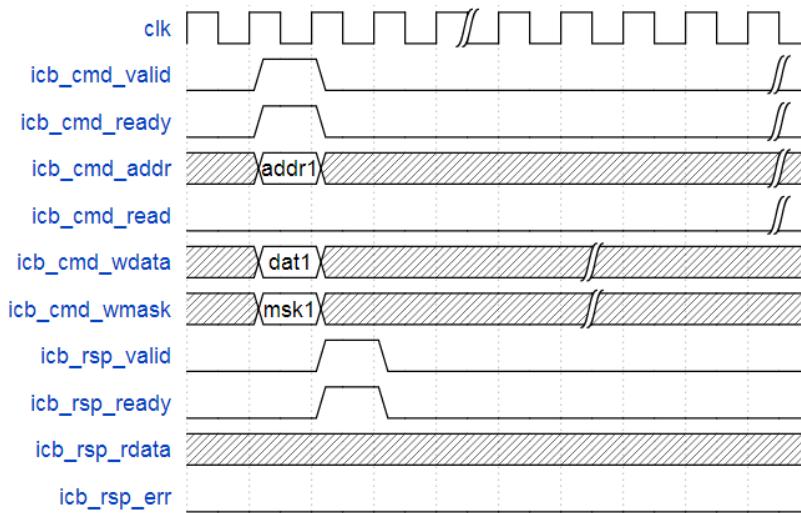


图 3-13 写操作下一周期返回结果

- 如图 3-14 所示：主设备向从设备通过 ICB 的 Command Channel 发送读操作请求（icb_cmd_read 为高），从设备立即接收该请求（icb_cmd_ready 为高）。从设备在四个周期后返回读结果且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

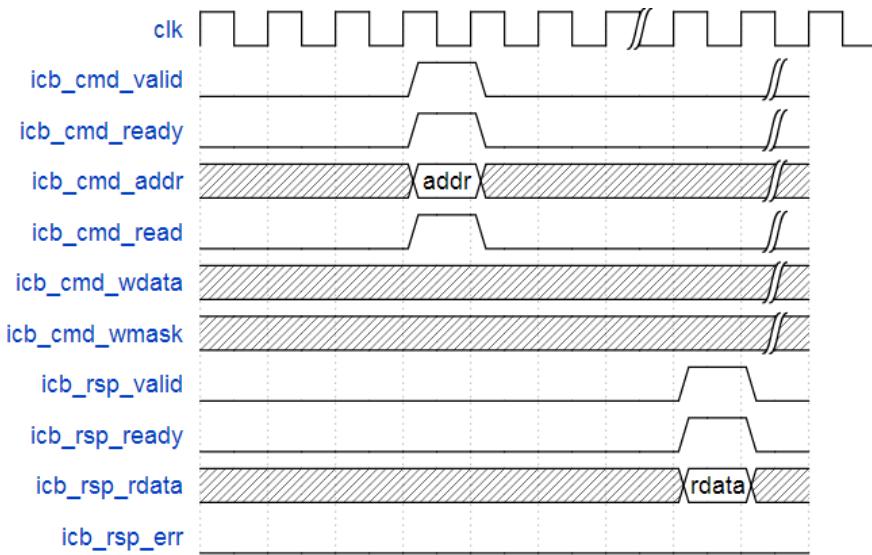


图 3-14 读操作四个周期返回结果

- 如图 3-15 所示：主设备向从设备通过 ICB 的 Command Channel 发送写操作请求（icb_cmd_read 为低），从设备立即接收该请求（icb_cmd_ready 为高）。从设备在四个周期后返回结果且结果正确（icb_rsp_err 为低），主设备立即接收该结果（icb_rsp_ready 为高）。

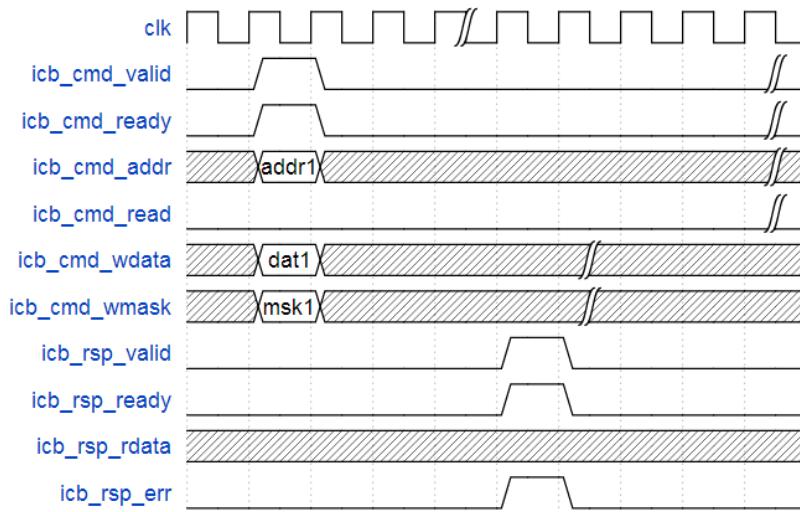


图 3-15 写操作四个周期返回结果

- 如图 3-16 所示：主设备向从设备通过 ICB 的 Command Channel 连续发送四个读操作请求（**icb_cmd_read** 为高），从设备均立即接收该请求（**icb_cmd_ready** 为高）。从设备在四个周期后连续返回四个读结果，其中前三个结果正确（**icb_rsp_err** 为低），第四个结果错误（**icb_rsp_err** 为高），主设备均立即接收此四个结果（**icb_rsp_ready** 为高）。

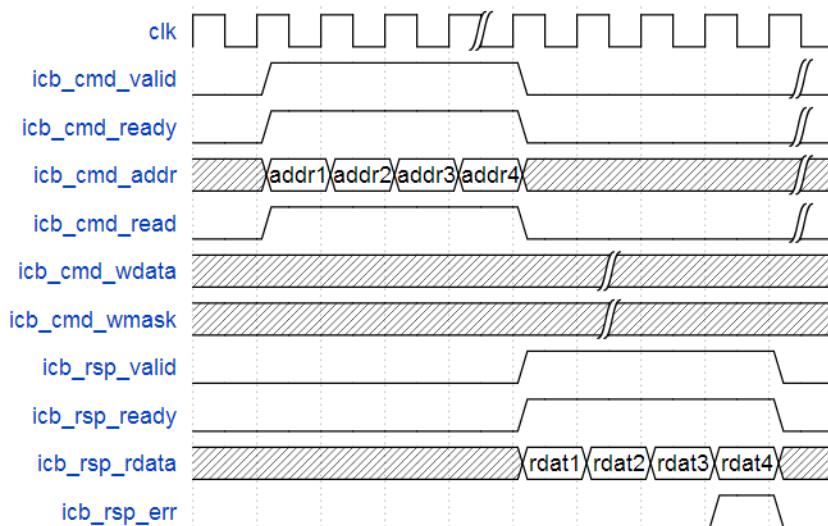


图 3-16 连续四个读操作均四个周期返回结果

- 如图 3-17 所示：主设备向从设备通过 ICB 的 Command Channel 连续发送四个写操作请求（**icb_cmd_read** 为低），从设备均立即接收该请求（**icb_cmd_ready** 为高）。从设备在四个周期后连续返回四个写结果，其中前三个结果正确（**icb_rsp_err** 为低），第四个结果错误（**icb_rsp_err** 为高），主设备均立即接收此四个结果（**icb_rsp_ready** 为高）。

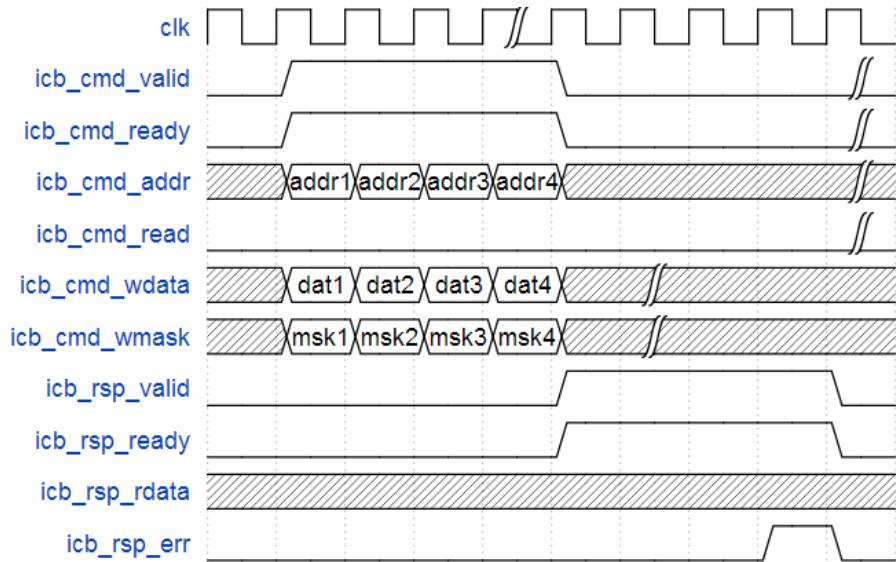


图 3-17 连续四个写操作均四个周期返回结果

- 如图 3-18 所示: 主设备向从设备通过 ICB 的 Command Channel 相继连续发送三个读和写操作请求。从设备立即接收了第一个和第三个请求, 但是第二个请求第一个周期并没有立即接受 (**icb_cmd_ready** 为低), 因此主设备一直将地址控制和写数据信号保持不变, 直到下一周期该请求被从设备接受 (**icb_cmd_ready** 为高)。从设备对于第一个和第二个请求都是在同一个周期就返回结果且被主设备立即接受, 但是对于第三个请求则是在下一个周期才返回结果, 并且主设备还没有立即接受 (**icb_rsp_ready** 为低), 因此从设备一直将返回信号保持不变, 知道下一周期该返回结果被主设备接受。

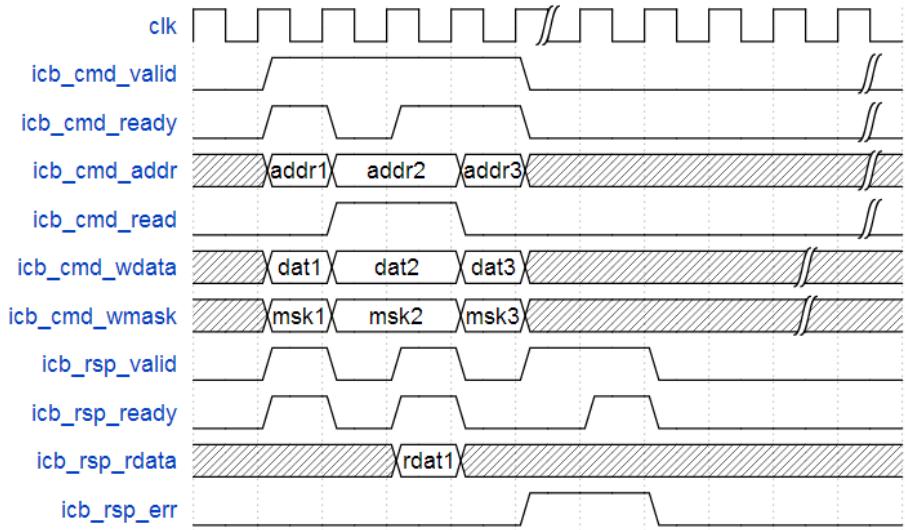


图 3-18 连续四个写操作均四个周期返回结果

3.3 SIRV-E200-SoC FPGA 原型平台

SiFive 公司的 Freedom E310 SoC 的平台提供完整的基于 Xilinx Artix-7 Arty FPGA Evaluation Kit 开发板的原型平台和示例。参考该示例，E200 项目同样基于该 FPGA 开发板，使用上节介绍的 SIRV-E200-SoC 搭建完整的原型平台与示例。

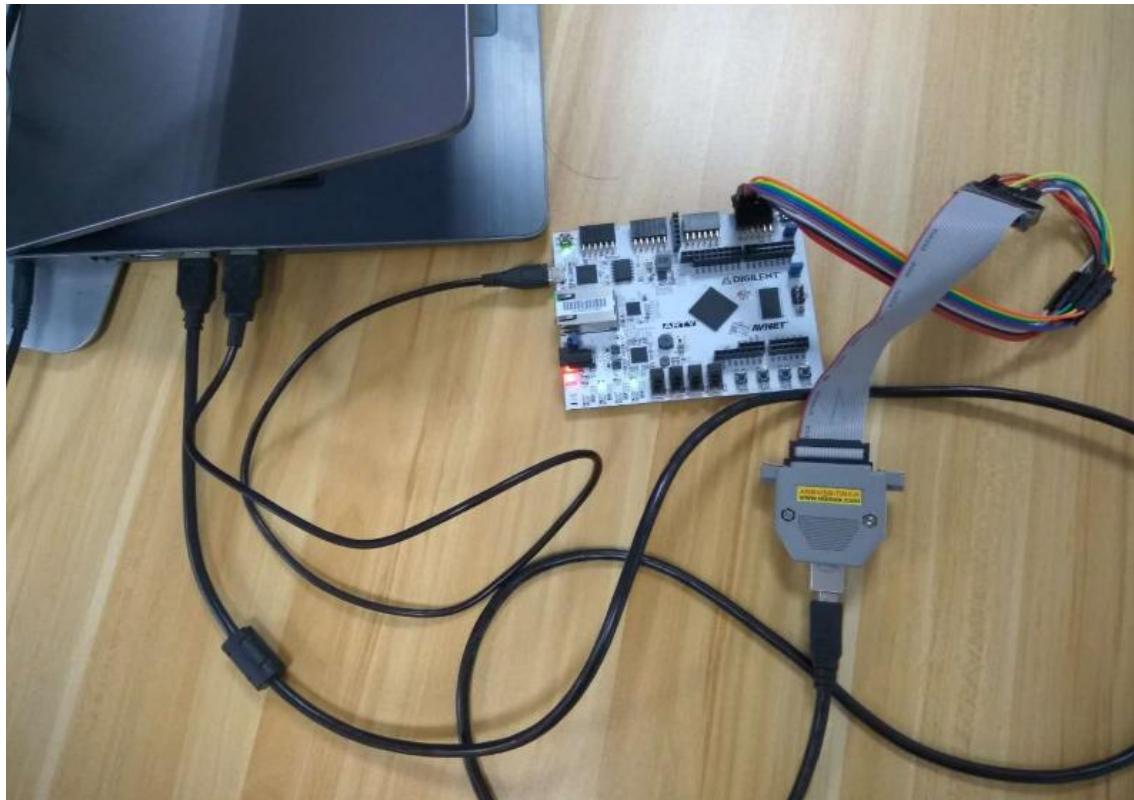


图 3-19 FPGA 开发板原型（包括调试器）

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 3-19 所示。

3.3.1 FPGA 开发板

FPGA 开发板使用 Xilinx Artix-7 35T Arty FPGA Evaluation Kit 开发板，该开发板是一款低成本的入门级 Xilinx FPGA 开发板，如图 3-20 所示。该开发板特别适用于电路设计规模不大的 FPGA 爱好者，其特性如下：

- 使用 Xilinx Artix-35T FPGA (xc7a35ticsg324-1L)
- FPGA 包含 33280 个逻辑单元，由 5200 个 Slices 组成。每个 Slice 包含 4 个 6 输入的 LUT 和 8 个 寄存器
- FPGA 包含 1800Kbits 的快速 Block RAM

- FPGA 包含五个时钟管理堆，每个均配备 PLL
- FPGA 包含 90 个 DSP slices
- FPGA 内部时钟频率可以高达 450MHz
- FPGA 提供片上模数转换（XADC）
- 支持通过 JTAG 或者 Quad-SPI 烧写 FPGA
- 开发板提供 256MB 的 DDR3L 接口，DDR3L 的数据宽度为 16 比特，频率为 667MHz
- 开发板提供 Quad-SPI 接口的 16MB Flash
- 开发板包含 USB 转 JTAG 电路，允许通过主机 PC 的 USB 口对 FPGA 的 JTAG 口进行烧录 FPGA
- 开发板支持通过 USB 接口供电或者通过单独的电源供电
- 开发板提供 10/100 Mbps 的以太网接口
- 开发板提供 UART 转 USB 电路，允许 FPGA 通过 UART 接口转 USB 接口与主机 PC 通信
- 开发板提供 4 个通用按键，1 个 Reset 按键，4 个 LED 灯，4 个 RGB LED 灯
- 开发板提供 4 个 Pmod connectors 和一组 Arduino/chipKIT Shield connector

用户可以在官方推荐的链接上购买此开发板 (<http://www.xilinx.com/products/boards-and-kits/arty.html>) 或者通过其他渠道购买。



图 3-20 Arty 开发板

注意：该 FPGA 开发板可以直接使用 USB 进行电源供电，或者单独使用电源线供电。出于简便，推荐直接使用 USB 线供电，使用 USB A to Micro-B Cable (即安卓手机充电器 USB 线) 对其进行供电即可，如图 3-21 所示。同时该 USB 线也会被用户将 FPGA 的 bitstream 文件（由 vivado 软件编译 Verilog 所得）烧录到 FPGA 芯片中。

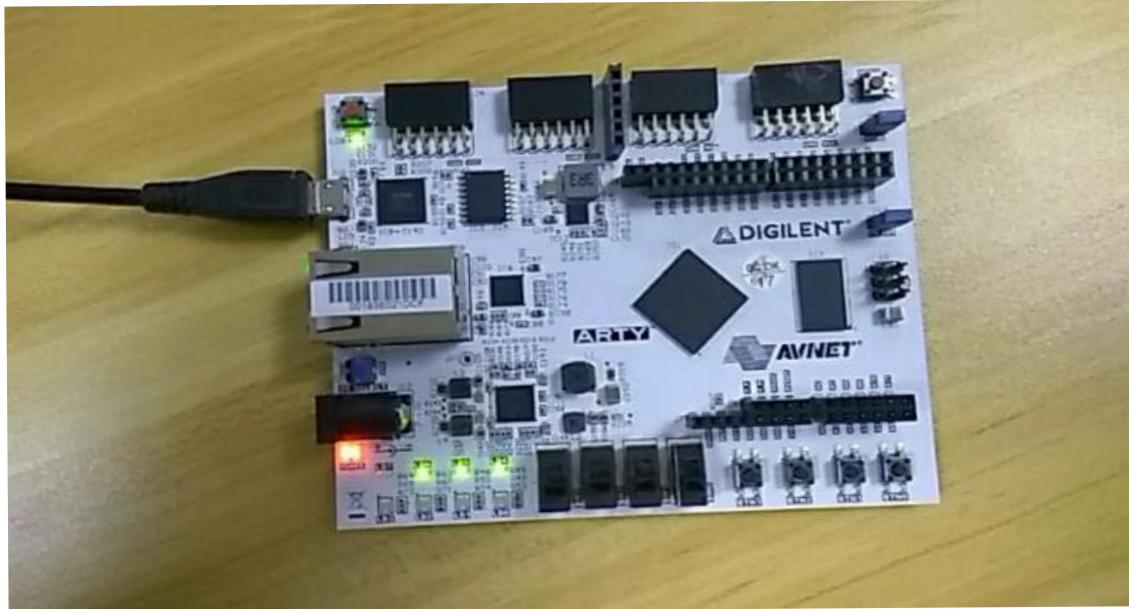


图 3-21 USB A to Micro-B Cable 对 FPGA 开发板供电

E200 项目 FPGA 项目相关的代码结构如下所示。

```
e200_opensource
|---fpga
    |---artydevkit
        |---constrs
            |--- arty-master.xdc // 存放 RTL 的目录
        |---Makefile // Arty 开发板的项目文件夹
        |---script // 存放约束文件的文件夹
        |---src
            |---system.v // 主约束文件
        |---Makefile // Makefile 脚本
    |---Makefile // 存放运行脚本的文件夹
    |---src
        |---Verilog 源代码的文件夹
    |---Makefile // 存放 Verilog 源代码的文件夹
    |---system.v // 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 3-22 所示。该宏定义会在 `e203_fpga_soc_top.v` 文件中控制 PC 复位值为 `0x2040_0000`，如图 3-23 所示，即 SoC 中对应的 QSPI Flash 基地址。因此在 FPGA 版本中处理器核上电复位后从外部的 Flash (`0x2040_000`) 开始执行程序。

```

CORE = e201
PATCHVERILOG ?= ""

base_dir := $(patsubst %/,%,$(dir $(abspath $(lastword $(MAKEFILE_LIST)))))

# Install RTLs
install:
	mkdir -p ${PWD}/install
	cp ${PWD}/../rtl/${CORE} ${INSTALL_RTL} -rf
	cp ${PWD}/artydevkit/src/system.org ${INSTALL_RTL}/system.v -rf
	sed -i 's/e200/${CORE}/g' ${INSTALL_RTL}/system.v
	echo 'define FPGA_SOURCE' >> ${INSTALL_RTL}/core/${CORE}_defines.v

EXTRA_FPGA_VSRCS :=
verilog := $(wildcard ${INSTALL_RTL}/*/*.v)
verilog += $(wildcard ${INSTALL_RTL}/*.v)

```

图 3-22 FPGA 项目宏定义文件中添加 FPGA_SOURCE

```

e200_subsys_top u_e200_subsys_top(
    .core_mhartid      (1'b0),
    `ifdef FPGA_SOURCE //{
        // This is the external QSPI flash base address
        .pc_rtvec          (32'h2040_0000),
    `else//{
        .pc_rtvec          (32'h0000_1000),
    `endif//}

    `ifdef E200_HAS_EAI //{
        .eai_icb_cmd_valid      (1'b0),
        .eai_icb_cmd_ready      (),
        .eai_icb_cmd_addr       (32'b0),
        .eai_icb_cmd_read       (1'b0),
        .eai_icb_cmd_wdata      (32'b0),
        .eai_icb_cmd_wmask      (4'b0),
        .eai_icb_cmd_lock       (1'b0),
        .eai_icb_cmd_excl       (1'b0),
        .eai_icb_cmd_size       (2'b0),

        .eai_icb_rsp_valid      (),
        .eai_icb_rsp_ready      (1'b0),
        .eai_icb_rsp_err         (),
        .eai_icb_rsp_excl_ok    (),
        .eai_icb_rsp_rdata      (),
    `endif//

```

图 3-23 宏定义 FPGA_SOURCE 控制 PC 的复位值

在 FPGA 的顶层模块 (system.v) 中除了例化 SoC 的顶层 (e203_fpga_soc_top) 之外，主要是使用 Xilinx 的 I/O Pad 单元例化顶层的 Pad，譬如 JTAG 接口 TDO 连接到名为 jd0 的 Pad 上，如图 3-24 所示。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 (hfclk)，来自于 MMCM 生成的 clk_out1，频率为 8.388MHz，该高速时钟经过 clkdivder 模块对其进行 256 分频，频率为 32.768kHz，如图 3-25 示。

```

IOBUF_jtag_TDI
(
    .O(iobuf_jtag_TDI_o),
    .IO(jd_4),
    .I(dut_io_pads_jtag_TDI_o_oval),
    .T(~dut_io_pads_jtag_TDI_o_oe)
);
assign dut_io_pads_jtag_TDI_i_ival = iobuf_jtag_TDI_o & dut_io_pads_jtag_TDI_o_ie;
PULLUP pullup_TDI (.O(jd_4));

wire iobuf_jtag_TDO_o;
IOBUF
#(
    .DRIVE(12),
    .IBUF_LOW_PWR("TRUE"),
    .IOSTANDARD("DEFAULT"),
    .SLEW("SLOW")
)
IOBUF_jtag_TDO
(
    .O(iobuf_jtag_TDO_o),
    .IO(jd_0),
    .I(dut_io_pads_jtag_TDO_o_oval),
    .T(~dut_io_pads_jtag_TDO_o_oe)
);
assign dut_io_pads_jtag_TDO_i_ival = iobuf_jtag_TDO_o & dut_io_pads_jtag_TDO_o_ie;

```

图 3-24 system 中 I/O Pad 例化

```

//=====================================================================
// Clock & Reset

wire SRST_n; // From FTDI Chip

mmcm ip_mmc
(
    .clk_in1(CLK100MHZ),
    .clk_out1(hfclk), // 8.388 MHz = 32.768 kHz * 256
    .clk_out2(), // 65 MHz
    .resetn(ck_rst),
    .locked(mmc_locked)
);

wire slowclk;
clkdivider slowclkgen
(
    .clk(hfclk),
    .reset(~mmc_locked),
    .clk_out(slowclk)
);

```

图 3-25 system 中的时钟生成

SIRV-E200-SoC 的顶层 I/O Pad 经过 FPGA 的约束文件 (arty-master.xdc) 进行约束使之连接到 FPGA 芯片外部真实的 Pin 脚上面，譬如 JTAG 的 Pad(jd0 ~ jd7)被连接到了 FPGA 芯片的 D4/D3 等 pin 脚上，如图 3-26 所示。

```

##Pmod Header JD

set_property -dict { PACKAGE_PIN D4      IOSTANDARD LVCMOS33 } [get_ports { jd_0 }];
#IO_L11N_T1_SRCC_35 Sch=jd[1]
set_property -dict { PACKAGE_PIN D3      IOSTANDARD LVCMOS33 } [get_ports { jd_1 }];
#IO_L12N_T1_MRCC_35 Sch=jd[2]
set_property -dict { PACKAGE_PIN F4      IOSTANDARD LVCMOS33 } [get_ports { jd_2 }];
#IO_L13P_T2_MRCC_35 Sch=jd[3]
#set_property -dict { PACKAGE_PIN F3      IOSTANDARD LVCMOS33 } [get_ports { jd_3 }];
#IO_L13N_T2_MRCC_35 Sch=jd[4]
set_property -dict { PACKAGE_PIN E2      IOSTANDARD LVCMOS33 } [get_ports { jd_4 }];
#IO_L14P_T2_SRCC_35 Sch=jd[7]
set_property -dict { PACKAGE_PIN D2      IOSTANDARD LVCMOS33 } [get_ports { jd_5 }];
#IO_L14N_T2_SRCC_35 Sch=jd[8]
set_property -dict { PACKAGE_PIN H2      IOSTANDARD LVCMOS33 } [get_ports { jd_6 }];
#IO_L15P_T2_DQS_35 Sch=jd[9]
#set_property -dict { PACKAGE_PIN G2      IOSTANDARD LVCMOS33 } [get_ports { jd_7 }];
#IO_L15N_T2_DQS_35 Sch=jd[10]

##USB-UART Interface (FTDI FT2232H)

set_property -dict { PACKAGE_PIN D10     IOSTANDARD LVCMOS33 } [get_ports { uart_rxd_out }];
#IO_L19N_T3_VREF_16 Sch=uart_rxd_out
set_property -dict { PACKAGE_PIN A9      IOSTANDARD LVCMOS33 } [get_ports { uart_txd_in }];
#IO_L14N_T2_SRCC_16 Sch=uart_txd_in

```

图 3-26 arty-master.xdc 中的 pin 脚约束

3.3.2 生成 mcs 文件烧写 FPGA

在前文中介绍了 E200 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 e200 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_core 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。

```

```
// 如果需要为其他型号的 core 进行设置，则仅需更改 CORE 的参数。譬如假设设置需要编  
// 译的 Core 型号为 e225fd，则使用命令 make install CORE=e225fd
```

```
// 步骤五：安装 Arty 开发板的 board part 到 Vivado 软件中。  
// 如果第一次使用 Vivado 软件为 Arty 开发板进行编译则可能出现如图 3-27 中所示的错误，这是因为没有为 Arty 开发板安装 board part。  
// 按照 Digilent 公司网址链接  
( https://reference.digilentinc.com/reference/software/vivado/board-files ) 中  
// 的说明安装 Arty 开发板的 board part 即可。
```

```
// 步骤六：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/artydevkit/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/artydevkit/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。
```

```
root@dawn-jobs:/home/dawn/jobs/freedom/fpga/e100artydevkit  
# set wrkdir [file join [pwd] obj]  
# set ipdir [file join $wrkdir ip]  
# set top {system}  
# create_project -part $part_fpga -in_memory  
# set_property -dict [list \  
#   BOARD_PART $part_board \  
#   TARGET_LANGUAGE {Verilog} \  
#   SIMULATOR_LANGUAGE {Mixed} \  
#   TARGET_SIMULATOR {XSim} \  
#   DEFAULT_LIB {xil_defaultlib} \  
#   IP_REPO_PATHS $ipdir \  
# ] [current project]  
ERROR: [Board 49-71] The board_part definition was not found for digilentinc.com:arty:part0:1.1.  
The project's board_part property was not set, but the project's part property was set to xc7a3sticsg324-1L. Valid board part values can be retrieved with the 'get_board_parts' Tcl command. Check if board.repoPaths parameter is set and the board_part is installed from the tcl app store.  
INFO: [Common 17-17] undo 'set_property'  
  
while executing  
"rdi::add_properties -dict {BOARD_PART digilentinc.com:arty:part0:1.1 TARGET_LANGUAGE Verilog SIMULATOR_LANGUAGE Mixed TARGET_SIMULATOR XSim DEFAULT_LIB xc7a3sticsg324-1L}"  
invoked from within  
"set_property -dict [list \  
  BOARD_PART $part_board \  
  TARGET_LANGUAGE {Verilog} \  
  SIMULATOR_LANGUAGE {Mixed} \  
  TARGET_SIMULATOR {XSim} \  
]
```

图 3-27 未安装 Arty 开发板的 board part 到 Vivado 软件中引起之错误

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。

```
// 步骤一：打开 Vivado 软件。  
// 步骤二：打开 Hardware Manager，自动连接 Arty 开发板。  
// 步骤三：右键 FPGA Device，选择“Add Configuration Memory Device”。  
// 步骤四：选择如下参数的 Flash：  
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/artydevkit/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“PROG”按键触发硬件电路使用 Flash 中的内容对 FPGA 重新进行烧录
```

3.3.3 JTAG 调试器

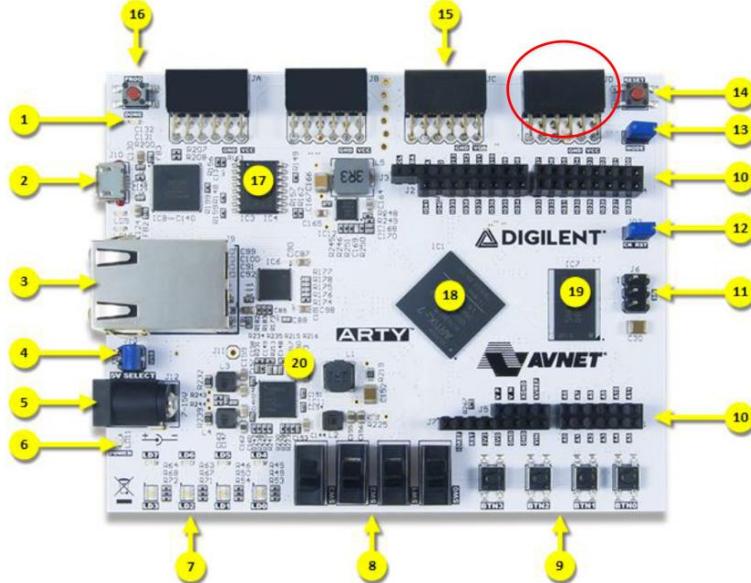


图 3-28 JTAG Pin 脚连接的 PMOD Header JD

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），在前文中曾经介绍了 E200 处理器核支持通过标准的

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 D4/D3 等 pin 脚上，而该组 pin 脚在 Arty 开发板上实际被连接到 PMOD Header JD 上，如图 3-28 中红色圆圈所示。

使用 GDB 调试进行嵌入式调试是一种常用的远程调试方式，GDB 软件运行在主机 PC 端，通过“硬件连接组件”连接主机 PC 与开发板，然后对开发板进行调试。连接主机 PC 与开发板之间的“硬件连接组件”便称之为“调试器”。目前商用的 MCU 均提供的“调试器”，具体的调试器类型与开发板的接口和类型有关（譬如 ARM Cortex M 系列采用 SWD 接口而非 JTAG 接口），本文在此不做过多赘述。

E200 项目参考了 SiFive 公司的 Freedom 310 开源 SoC 平台，提供完整的硬件调试参考方案。由于 E200 项目采用标准的 JTAG 接口，因此需要使用“JTAG 调试器”，该调试器通过 USB 转接线（USB A to B Cable）与上游主机 PC 的 USB 接口连接，同时通过 JTAG 接口与下游的 FPGA 开发板连接，如图 3-29 中黄色圆圈所示。

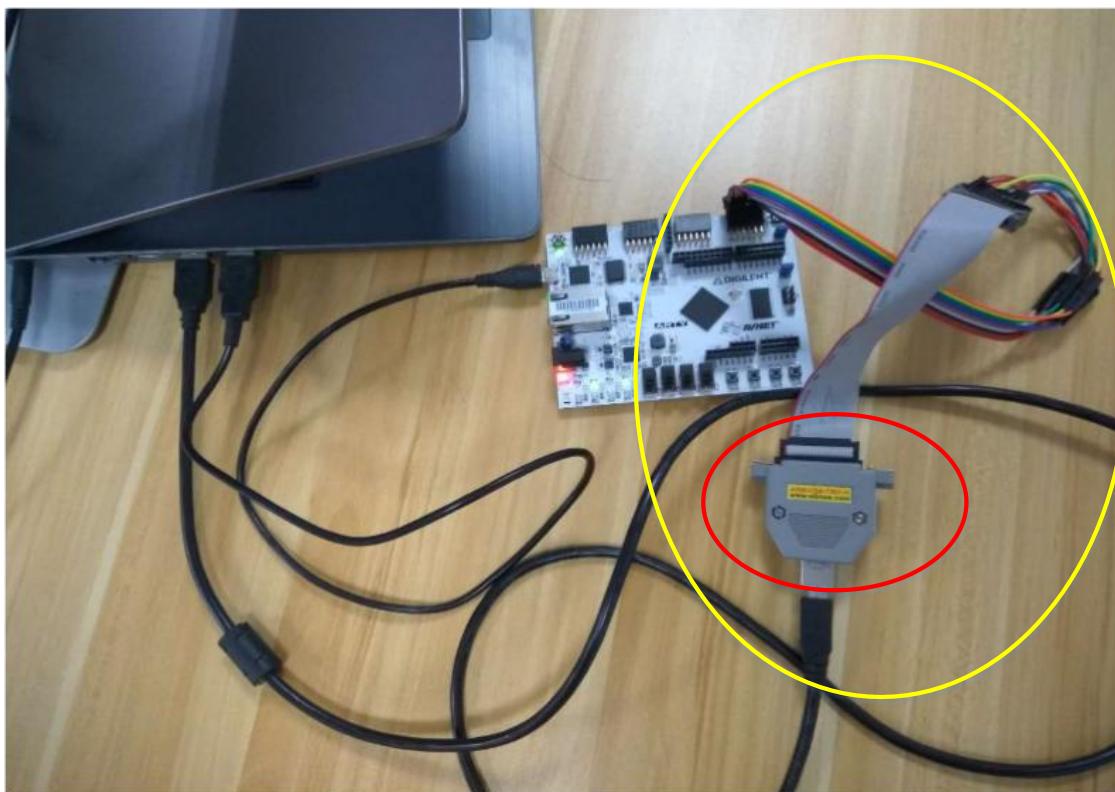


图 3-29 完整的 JTAG 调试器

该套 JTAG 调试器需要用户自己来 DIY 组装。

该调试器组件的核心为“Olimex ARM-USB-TINY-H”，这是一个硬件 JTAG 调试器，如图 3-29 中红色圆圈所示。

该调试器可以从官网推荐链接 (<https://www.olimex.com/Products/ARM/JTAG/ARM-USB-TINY-H/>) 购买，或者从其他的途径购买。

由于 Olimex ARM-USB-TINY-H 的输出口是双排管脚接口，因此需要使用公口转母口连接线（Male-To-Female Jumper Cables）将其与 FPGA 的 PMOD Header JD 连接，如图 3-30 所示。

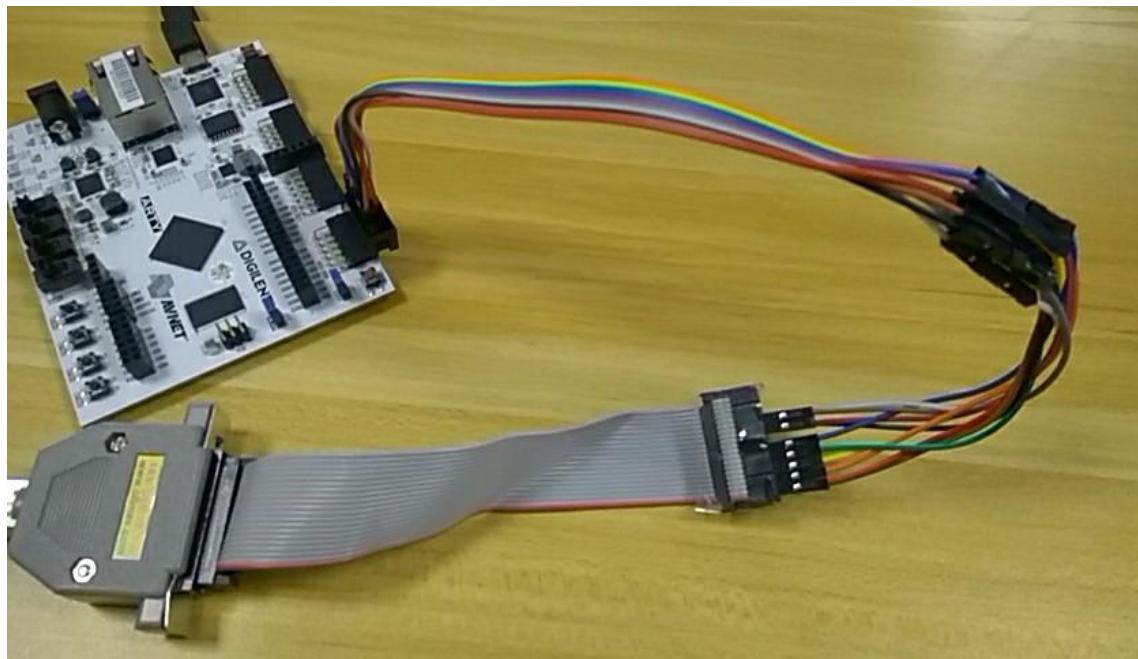


图 3-30 使用 Male-To-Female Jumper Cables 连接 JTAG 调试器与 FPGA 开发板

表 3-5 Olimex ARM-USB-TINY-H 输出的 20 根公口管脚颜色编号

	1 : VREF (red)	2 : VREF (brown)
	3 : nTRST (orange)	4
	5 : TDI (yellow)	6
	7 : TMS (green)	8
NOTCH	9 : TCK (blue)	10
NOTCH	11	12
	13 : TDO (purple)	14 : GND (black)
	15 : nRST (grey)	16 : GND (white)
	17	18
	19	20
	LED	

表 3-6 FPGA 的 PMOD Header JD 输入的 12 根母口管脚颜色编号

square pad	1 : TDO (purple)	7 : TDI (yellow)
	2 : nTRST (orange)	8 : TMS (green)
	3 : TCK (blue)	9 : nRST (grey)
	4	10
"GND"	5 : GND (black)	11 : GND (white)
"VCC"	6 : VREF (brown)	12 : VREF (red)

为了方便连线且防止出错，推荐使用不同颜色的公口转母口连接线进行连接。如表 3-5 中对 Olimex ARM-USB-TINY-H 输出的 20 根公口管脚进行了不同颜色的编号。如表 3-6 中对 FPGA 的 PMOD Header JD 输入的 12 根母口管脚进行了不同颜色的编号。在连线时则严格的使用对应颜色的连线逐一进行连接后(注意紧密连接防止接触不良)即宣告完成。

由于 Olimex ARM-USB-TINY-H 使用 USB 转接线 (USB A to B Cable) 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：将 FPGA 开发板通电（可以使用普通安卓手机 USB 充电器连接线供电，或者使用独立的电源供电）。使用  
USB A to B Cable 将主机 PC 与 FPGA 开发板连接。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 3-31 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）。
```

```
// 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
```

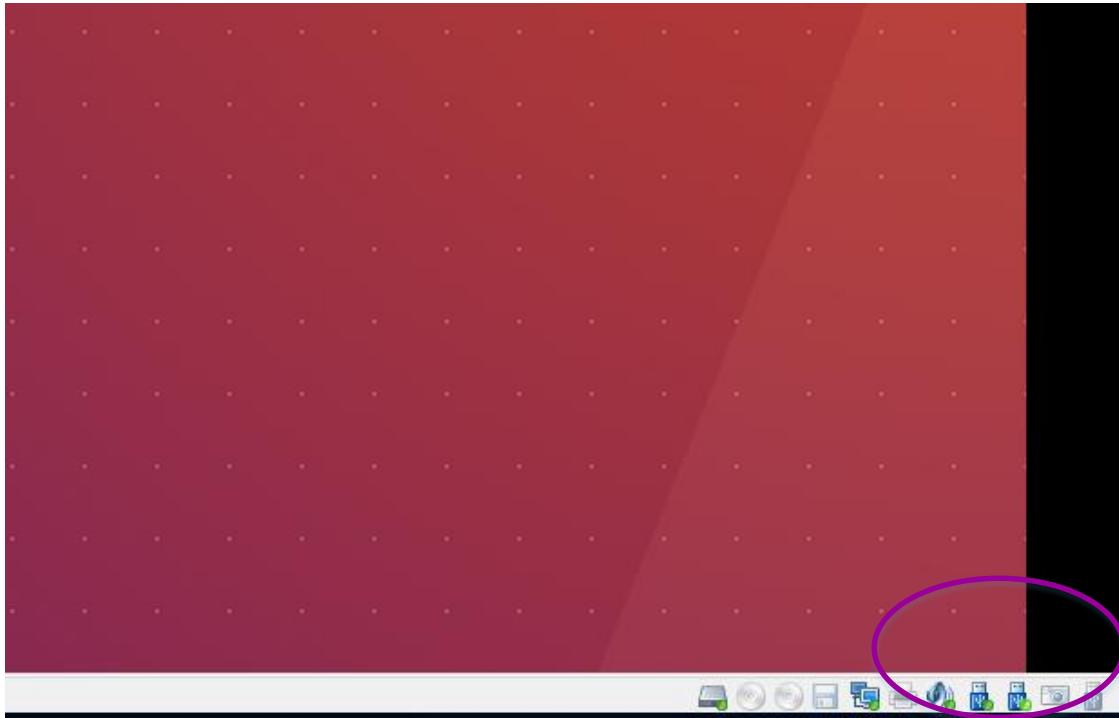


图 3-31 虚拟机 Linux 系统识别 USB 图标

对于 JTAG 调试器的硬件 DIY 工作至此便已完成。在下一章将介绍利用该 JTAG 调试器如何使用 GDB 软件对 E200 SoC 原型进行远程调试。

3.3.4 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 Xilinx Artix-7 35T Arty FPGA Evaluation Kit 开发板
- 购买一根 USB A to Micro-B Cable（即安卓手机充电器 USB 线）用于给 FPGA 开发板供电且烧录 FPGA
- 购买一块 Olimex ARM-USB-TINY-H Debugger
- 购买一根 USB A to B Cable 用于连接主机 PC 与 Olimex ARM-USB-TINY-H Debugger
- 购买一组 Male-To-Female Jumper Cables 用于连接 Olimex ARM-USB-TINY-H Debugger 与 FPGA 开发板

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 软件且安装 Arty 开发板的 board part

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。

4 运行和调试软件 Demo

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。

4.1 Freedom-E-SDK 平台简介

在前文中我们介绍了 SiFive 公司推出的开源 Freedom E310 SoC 平台，为了让用户能够非常容易的使用起 RISC-V 处理器开发软件，SiFive 公司不仅仅开源了其 SoC 平台，还开发了一套与之配套的软件开发平台，称之为 Freedom-E-SDK 平台，并且同样予之开源。

需要注意的是，Freedom-E-SDK 是并不是一个软件，它本质上是由一些 Makefile，BSP(板级支持包：Board Support Package)，一些脚本和软件示例组成了一套开发环境。其基于 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。因此，它主要包含如下几个方面的内容：

- RISC-V 软件工具链
- RISC-V 调试工具链
- BSP (板级支持包：Board Support Package)
- 若干软件示例

Freedom-E-SDK 的所有源代码均开源托管于 Github 网站上(<https://github.com/sifive/freedom-e-sdk>)，如图 4-1 所示。目前 Freedom-E-SDK 支持 SiFive 公司的所有硬件产品，包括 HiFive1 开发板，和若干 SiFive 公司的处理器 IP 和 FPGA 原型平台。

 FreedomStudio	E31 vectored_interrupt.c as a linked file	Jul 26, 2017
 bsp	changed synch trap entry to match other vectors	Jul 26, 2017
 openocd @ 45f2808	updated OpenOCD version	Aug 15, 2017
 riscv-gnu-toolchain @ f5fae1c	Update submodule to the 20170608 tagged releases	Jun 9, 2017
 software	blacklisted watchdog program for coreplexip-e51-arty board	Aug 15, 2017
 .gitignore	Add a SMP example	Jun 14, 2017
 .gitmodules	Use HTTPS when possible and speedup the first build (#40)	Jan 12, 2017
 LICENSE	Use full LICENSE text	Nov 28, 2016
 Makefile	Add linker scripts that target the scratchpad	Jun 14, 2017
 README.md	rename 'PREFIX' to 'PATH' for more intuitive naming	May 17, 2017
 regression.bash	Add a simple regression script	Jun 8, 2017

图 4-1 Freedom-E-SDK 的 Github 网站

4.2 SIRV-E-SDK 简介

Freedom-E-SDK 是一款非常优秀的嵌入式软件开发环境，非常感谢 SiFive 公司将其开源。为了让用户能够轻松的使用起蜂鸟 E200 处理器核开发软件，E200 项目也以 Freedom-E-SDK 为蓝本，在其基础上做了如下主要修改：

- 为不同的蜂鸟 E200 系列处理器核创建了 BSP 子目录。
- 去除了一些不必要的目录和文件。
- 去除了一些不相关的软件示例，对软件示例进行了若干修改。
- 去除了 GNU Toolchain 和 OpenOCD 源代码，无需编译生成工具链，而是使用预先编译好的工具链。

为了方便用户理解区别，本文将“修改后的 Freedom-E-SDK（服务蜂鸟 E200 处理器核系列）”称之为“SIRV-E-SDK”，但是非常感谢开源 Freedom-E-SDK 为社区所做的贡献。

值得强调的是，由于蜂鸟 E200 处理器和 SoC（SIRV-E200-SoC）由于与 SiFive 公司的 Freedom E310 SoC 完全软件兼容，因此理论上 Freedom-E-SDK 软件开发平台应该可以完全无缝的被移植到 SIRV-E200-SoC 上（事实也确实如此），之所以单独创建一个修改版的 SIRV-E-SDK，主要是出于如下原因：

- SIRV-E-SDK 的主要目的在于演示，演示 SIRV-E200-SoC 的 FPGA 原型平台运行示例程序。
- SIRV-E-SDK 由于删除了一些不相干的文件和目录，相比 Freedom-E-SDK 更加简洁，方便首次使用用户能够理解和上手。
- 原 Freedom-E-SDK 平台需要下载 GNU Toolchain 和 OpenOCD 的源代码然后编译生成工具链，由于其源代码体积非常巨大，整个过程耗时耗力。而 SIRV-E-SDK 则直接使用预先编译好的工具链，因此整个 SIRV-E-SDK 的代码量很小，方便用户快速的从 Github 上下载并搭建成型。
- 原 Freedom-E-SDK 平台是为了成为一个通用平台，因此在被不断的维护和更新，有更多的软件示例和功能在不断的添加。如果 E200 项目直接使用其源平台，则难免会出现某些更新带来的错误。因此 SIRV-E-SDK 则更追求稳定，一旦稳定后将停止修改，以方便用户能够稳定的使用 SIRV-E200-SoC 进行示例软件的移植和演示。

因此，如果用户想利用 SIRV-E200-SoC 开发更加丰富的软件，本文鼓励使用原始的 Freedom-E-SDK 软件平台加以移植（而不是 SIRV-E-SDK），因为 Freedom-E-SDK 有很多的软件开发者共同维护且不断更新，有很多丰富的软件例程，用户可以及时的与 Freedom-E-SDK 保持更新，共享其生态，享受其不断发展的结果。

4.2.1 SIRV-E-SDK 代码结构

SIRV-E-SDK 目录位于 e200_opensource 下的一个单独子目录，其代码结构如下所示。

```
e200_opensource
|---sirv-e-sdk          // 存放 sirv-e-sdk 的目录
|   |---bsp              // 存放板级支持包 (Board Support Package) 的目录
|   |---drivers           // 存放底层驱动代码的目录
|   |---env               // 存放不同平台的配置文件夹
|       |---sirv-e201-arty // 基于 E201 Core 平台的配置文件夹
|       |---sirv-e203-arty // 基于 E203 Core 平台的配置文件夹
|       |---sirv-e205-arty // 基于 E205 Core 平台的配置文件夹
|       |---sirv-e205f-arty // 基于 E205f Core 平台的配置文件夹
|       |---sirv-e205fd-arty // 基于 E205fd Core 平台的配置文件夹
|       |---sirv-e225fd-arty // 基于 E225fd Core 平台的配置文件夹
|   |---include            // 存放一些头文件
|   |---libwrap             // 存放一些库文件
|   |---tools               // 存放一些工具脚本文件
|   |---software            // 存放示例程序的源代码
|       |---demo_gpio        // GPIO 示例程序
|       |---coremark           // CoreMark 跑分程序
|       |---dhrystone          // Dhrystone 跑分程序
|   |---work                // 存放工具链的目录
|   |---Makefile             // 主 Makefile 文件
```

各个主要的代码模块简述如下：

- bsp/drivers 目录主要用于一些驱动程序代码，譬如 PLIC 模块的底层驱动函数和代码。
- bsp/include 目录下主要存放包含 SoC 中外设模块的寄存器地址等参数的头文件，如图 4-2 所示。
- bsp/libwrap 目录主要存放一些与硬件平台相关的底层库函数的具体实现，这是对于嵌入式开发平台为了能够支持完整的 C/C++ 标准库函数必须进行的移植工作。譬如，最典型的 printf 函数，由于在嵌入式平台中没有显示屏，常见的做法是将嵌入式开发板通过 UART 接口与主机 PC 的串口连接，然后将 printf 函数打印的信息通过主机 PC 的串口打印显示在主机的电脑屏幕上。因此需要将 printf 库函数调用的最底层函数 write 函数进行移植，最底层的 write 函数将向 UART 的某些寄存器发起写操作从而通过 UART 发送字符串至主机 PC 串口。bsp/libwrap 目录下存放的便是最底层函数的移植实现。
- bsp/env 目录主要用于存放不同 board 的相关支持包。譬如 bsp/env/sirv-e203-arty 文件夹存放的是使用 Arty FPGA 开发板（基于 E203 处理器核的 SoC）的支持包。另外，bsp/env 目录下还存放其他支持文件，譬如 common.mk 作为一个公用的 Makefile 脚本，start.S 作为程序的上电引导程序，和其他的若干头文件。
- software 目录主要存放软件示例，包括一个基本的 demo_gpio 示例，dhrystone 跑分程序和 CoreMark 跑分程序。每个示例均有其单独的文件夹，包含了各自的源代码，Makefile 和编译选项（在 Makefile 中指定）等。

由于本文侧重于介绍 CPU 的硬件使用，因此对于软件部分在此不做赘述，请用户自行阅读 SIRV-E-SDK 的脚本和代码了解其细节。

4.3 使用 SIRV-E-SDK 运行示例程序

E200 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 Xilinx Arty 开发板（基于 SIRV-E200-SoC），使用 SIRV-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 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：由于编译汇编示例程序需要使用到 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，因此本文推荐使用预先已经编译好的 GCC 工具链。用户可以在链接  
http://pan.baidu.com/s/1eSD0COM 下载压缩包 riscv32_unkown_elf_gcc6.1.0.tar.gz 和  
openocd_gcc6.1.0.tar.gz，然后按照如下步骤解压使用。
```

```
cp riscv32_unkown_elf_gcc6.1.0.tar.gz ~/  
cp openocd_gcc6.1.0.tar.gz ~/  
// 将两个压缩包均拷贝到用户的根目录下  
  
cd ~/  
tar -xzvf riscv32_unkown_elf_gcc6.1.0.tar.gz  
// 进入根目录并解压 GNU Toolchain 压缩包  
tar -xzvf openocd_gcc6.1.0.tar.gz  
// 解压 OpenOCD 压缩包  
  
cd <your_e200_dir>/sirv-e-sdk  
// 进入到 e200_opensource 的 sirv-e-sdk 目录文件夹  
mkdir -p work/build/openocd/prefix  
// 在 sirv-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 ~/openocd_gcc6.1.0/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_e200_dir>/sirv-e-sdk  
// 再次进入到 e200_opensource 的 sirv-e-sdk 目录文件夹  
mkdir -p work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix/  
// 在 sirv-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 ~/riscv32_unkown_elf_gcc6.1.0/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
```

```
// 步骤四：按照本文第 3.3 节中所述方法，准备好基于 SIRV-E200-SoC 的 Xilinx Arty FPGA 原型开发板，并
```

将 bitstream 文件或者 mcs 文件烧录至 FPGA 中，且用 JTAG 调试器将 Arty 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_e200_dir>/sirv-e-sdk  
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。  
  
make software PROGRAM=demo_gpio BOARD=sirv-e203-arty  
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译  
    // demo_gpio 示例程序。若需使用其他型号的 E200 系列 Core，譬如  
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。  
    // 程序若编译成功则显示如图 4-2 所示。
```

// 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_e200_dir>/sirv-e-sdk  
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。  
  
make upload PROGRAM=demo_gpio BOARD=sirv-e203-arty  
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载  
    // demo_gpio 示例程序至 FPGA 开发板中。若需使用其他型号的 E200 系列 Core，譬如  
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。  
    // 该过程的原理是调用 GDB 和 OpenOCD 软件通过 JTAG 调试器将编译好的 demo_gpio  
    // 程序下载到 E203 处理器核中。  
    // 程序若下载成功则显示如图 4-3 所示。
```

// 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 Log 符号到主机 PC 的显示屏上。  
// 因此需要先将串口显示终端准备好，在 Ubuntu 的命令行终端中使用如下命令。  
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 3.3.2 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 4-4 所示。
```

由于本文侧重于介绍 CPU 的硬件设计，因此对于软件部分在此不做赘述，请用户自行阅读 demo_gpio 的代码了解其程序细节。

```
rap/sys/times.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/libwrap/sys/sbrk.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/libwrap/sys/_exit.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/libwrap/misc/write_hex.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/work/build/riscv-gnu-toolchain/riscv32-unknown-elf-prefix/bin/riscv32-unknown-elf-gcc -O2 -fno-builtin-printf -DUSE_PLIC -DUSE_MTIME -g -march=rv32imc -mabi=ilp32 -mcmodel=medany -ffunction-sections -fdata-sections -fno-builtin-printf -fno-builtin-malloc -I/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/include -I/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/drivers/ -I/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env -I/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/start.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/entry.o demo_gpio.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/drivers/plic_driver.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/sirv-e203-arty/init.o /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/sirv_printf.o -o demo_gpio -Wl,--wrap=malloc -Wl,--wrap=free -Wl,--wrap=open -Wl,--wrap=lseek -Wl,--wrap=read -Wl,--wrap=write -Wl,--wrap=fstat -Wl,--wrap=stat -Wl,--wrap=close -Wl,--wrap=link -Wl,--wrap=unlink -Wl,--wrap=execve -Wl,--wrap=fork -Wl,--wrap=getpid -Wl,--wrap=kill -Wl,--wrap=wait -Wl,--wrap=isatty -Wl,--wrap=times -Wl,--wrap=sbrk -Wl,--wrap=_exit -L. -Wl,--start-group -lwrap -lc -Wl,--end-group -T /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/sirv-e203-arty/link.lds -nostartfiles -Wl,--gc-sections -Wl,--wrap=scanf -Wl,--wrap=malloc -Wl,--wrap=printf -Wl,--check-sections -L/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env  
make[1]: Leaving directory '/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/software/demo_gpio'  
zhenbohu@ubuntu:~/jx_work/e200_opensource/sirv-e-sdk$
```

图 4-2 编译 demo_gpio 程序成功后显示界面

```
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part: 0x0e31, ver: 0x1)  
riscv.cpu: target state: halted  
halted at 0x8000007e due to debug interrupt  
Loading section .init, size 0xb4 lma 0x20400000  
Loading section .text, size 0x159c lma 0x204000b4  
Loading section .rodata, size 0xaac lma 0x20401650  
Loading section .data, size 0x438 lma 0x204020fc  
riscv.cpu: target state: halted  
halted at 0x80000004 due to software breakpoint  
riscv.cpu: target state: halted  
halted at 0x80000004 due to software breakpoint  
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part: 0x0e31, ver: 0x1)  
riscv.cpu: target state: halted  
halted at 0x80000004 due to software breakpoint  
Start address 0x20400000, load size 9524  
Transfer rate: 7 KB/sec, 2381 bytes/write.  
halted at 0x20400004 due to step  
halted at 0x20400004 due to step  
riscv.cpu: target state: halted  
riscv.cpu: target state: halted  
halted at 0x20400004 due to step  
halted at 0x20400004 due to step  
shutdown command invoked  
shutdown command invoked  
A debugging session is active.  
  
Inferior 1 [Remote target] will be detached.  
  
Quit anyway? (y or n) [answered Y; input not from terminal]  
Remote communication error. Target disconnected.: Connection reset by peer.  
Successfully uploaded 'demo_gpio' to sirv-e203-arty.  
zhenbohu@ubuntu:~/jx_work/e200_opensource/sirv-e-sdk$
```

图 4-3 下载 demo_gpio 程序至 FPGA 开发板成功后显示界面

图 4-4 运行 demo_gpio 程序后在主机 PC 的串口显示终端上的 Log 字符

4.4 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。

表 4-1 GDB 常用命令

命令	介绍
load file	动态链入 file 文件，并读取它的符号表
jump	使当前执行的程序跳转到某一行，或者跳转到某个地址
info br	使用该指令可查看断点信息，br 是断点 break 的缩写，GDB 具有自动补齐功能，此命令等效于 info break
info source	使用该指令可查看当前程序的信息
info stack	使用该指令可查看程序的调用层次关系
list function-name	使用该指令可列出某个函数
list line-number	列出某行附近的代码
break function break line-number	在指定的函数，或者行号处设置断点
break *address	在指定的地址处设置断点。一般在没有源代码时使用
continue	恢复程序运行，直到遇到下一个断点
step	进入下一行代码的执行，会进入函数内部
step number	等效于连续执行 number 次 step 命令
next	执行下一行代码。但不会进入函数内部
next number	等效于连续执行 number 次 next 命令
until until line-number	继续运行直到到达指定行号，或者函数，地址等
stepi nexti	stepi/nexti 命令与 step/next 的区别在于其执行下一条汇编指令，而不是下一行代码（譬如 C/C++中的一行代码）
x address	打印指定内存地址中的值
p variable	打印指定变量的值

为了能够支持 GDB 对其进行调试，Freedom E310 SoC 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。通

过编写 OpenOCD 的底层驱动文件能够使其通过 JTAG 接口连接 Freedom E310 SoC 并利用其硬件调试特性对其进行调试。

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 命令介绍如表 4-1 所示。

可按照如下步骤使用 GDB 和 OpenOCD 对 Xilinx Arty 开发板（基于 SIRV-E200-SoC）进行调试。

```
// 步骤一 ~ 步骤四：与第 4.3 节中描述的运行 demo_gpio 示例程序步骤一 ~ 步骤四相同。  
  
// 步骤五：使用 GDB+OpenOCD 远程调试 FPGA 原型开发板，使用如下命令：  
  
cd <your_e200_dir>/sirv-e-sdk  
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。  
  
    // 使用如下命令打开 OpenOCD  
make run_openocd PROGRAM=demo_gpio BOARD=sirv-e203-arty  
    // 运行该命令会使用 bsp/env 目录下的 sirv-e203-arty 板级支持包中 OpenOCD 的  
    // 配置文件来打开 OpenOCD，并与 Arty FPGA 开发板相连。  
    // 如果该步骤执行成功，则如图 4-5 所示。  
  
    // 由于命令行界面已经被 OpenOCD 挂住，因此需要重新开启一个新的 Terminal 终端，  
    // 然后使用如下命令打开 GDB  
make run_gdb PROGRAM=demo_gpio BOARD=sirv-e203-arty  
    // 运行该命令会自动打开 GDB 来调试 demo_gpio 示例程序。  
    // 如果该步骤执行成功，则进入了 GDB 的调试命令行界面，如图 4-6 所示。  
  
// 步骤六：演示使用 GDB 命令：  
// 接下来便可使用 GDB 的常用命令进行调试。  
b main  
    // 在 main 函数的入口处设置断点。  
  
info b  
    // 查看目前程序设置的断点，显示如图 4-7 所示。  
  
x 0x20400000  
x 0x20400004  
x 0x20400008  
    // 查看内存地址 0x20400000/0x20400004/0x20400008 中的数值，显示如图 4-8  
    // 所示。  
info reg  
info reg mstatus  
    // 查看当前处理器的通用寄存器的值和 CSR 寄存器 mstatus 的值，显示如图 4-9 所示。  
  
jump main  
    // 从程序的_start 入口开始执行，将停于设置的第一个断点处，显示如图 4-10 所示。  
  
ni
```

```
// 单步执行，显示如图 4-11 所示。
continue
// 继续执行，将停于下一个断点处，若无断点则一直执行至程序结束处。
```

```
zhenbohu@ubuntu:~/jx_work/e200_opensource/sirv-e-sdk$ make run openocd PROGRAM=demo_gpio BOARD=sirv-e203-arty
work/build/openocd/prefix/bin/openocd -f bsp/env/sirv-e203-arty/openocd.cfg
Open On-Chip Debugger 0.10.0-dev-g9bab078 (2017-02-02-01:39)
Licensed under GNU GPL v2
For bug reports, read
    http://openocd.org/doc/doxygen/bugs.html
adapter speed: 1000 kHz
Info : auto-selecting first available session transport "jtag". To override use 'transport select <transport>'.
Info : clock speed 1000 kHz
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (<unknown>), part: 0x0e31, ver: 0x1)
Info : Examined RISCV core; XLEN=32, misa=0x40001105
riscv.cpu: target state: halted
halted at 0x80000007e due to debug interrupt
Info : Found flash device 'micron n25q128' (ID 0x0018ba20)
cleared protection for sectors 64 through 255 on flash bank 0
```

图 4-5 打开 OpenOCD 后的命令行界面

```
zhenbohu@ubuntu:~/jx_work/e200_opensource/sirv-e-sdk$ make run gdb PROGRAM=demo_gpio BOARD=sirv-e203-arty
/home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/work/build/riscv-gnu-toolchain/riscv32-unknown-elf/prefix/bin/riscv32-unknown-elf-gdb software/demo_gpio/demo_gpio -ex "set remotetimeout 240" -ex "target extended-remote localhost:3333"
GNU gdb (GDB) 7.12.50.20170109-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from software/demo_gpio/demo_gpio...done.
Remote debugging using localhost:3333
0x80000007e in main (argc=<optimized out>, argv=<optimized out>) at demo_gpio.c:291
291      GPIO_REG(GPIO_OUTPUT_VAL) ^= bitbang_mask;
(gdb) ■
```

图 4-6 GDB 的命令行界面

```
Remote debugging using localhost:3333
0x20400004 in _start ()
    at /home/zhenbohu/jx_work/e200_opensource/sirv-e-sdk/bsp/env/start.S:12
12          la gp, __global_pointer$
(gdb) b main
Breakpoint 1 at 0x80000000: file demo_gpio.c, line 239.
(gdb) info b
Num  Type            Disp Enb Address   What
1    breakpoint      keep y  0x80000000 in main at demo_gpio.c:239
(gdb) ■
```

图 4-7 GDB 显示设置的断点

```
12          la gp, __global_pointer$  
(gdb) b main  
Breakpoint 1 at 0x80000000: file demo_gpio.c, line 239.  
(gdb) info b  
Num      Type            Disp Enb Address     What  
1        breakpoint    keep y   0x80000000 in main at demo_gpio.c:239  
(gdb) x 0x20400000  
0x20400000 <_start>: 0x6fc01197  
(gdb) x 0x20400004  
0x20400004 <_start+4>: 0xc2818193  
(gdb) x 0x20400008  
0x20400008 <_start+8>: 0x6fc04117  
(gdb) x 0x2040000a  
0x2040000a <_start+10>: 0x01136fc0  
(gdb) █
```

图 4-8 通过 GDB 查看内存中的数据

```
(gdb) info reg  
ra          0x80000094      -2147483500  
sp          0x90003fc0      -1879031872  
gp          0x90001000      -1879044096  
tp          0x00000000      0  
t0          0x80000004      -2147483644  
t1          0x80000000      -2147483648  
t2          0x00000030      48  
fp          0x00000000      0  
s1          0x10012000      268509184  
a0          0x00000001      1  
a1          0x80006b8d      -2147456115  
a2          0x00000000      0  
a3          0x0000000a      10  
a4          0x10013000      268513280  
a5          0x00000000      0  
a6          0x80006b8d      -2147456115  
a7          0x77206000      1998610432  
s2          0x00000000      0  
s3          0x00000000      0  
s4          0x00000000      0  
s5          0x00000000      0  
s6          0x00000000      0  
s7          0x00000000      0  
s8          0x00000000      0  
s9          0x00000000      0  
s10         0x00000000      0  
s11         0x00000000      0  
t3          0x77206465      1998611557  
t4          0x20687000      543715328  
t5          0x20687469      543716457  
t6          0x65647000      1701081088  
pc          0x80000000      -2147483648  
(gdb) info reg mstatus  
mstatus      0x1800      6144  
(gdb) █
```

图 4-9 GDB 显示寄存器的值

```

0x20400004 < start+4>: 0xc2818193
(gdb) x 0x20400008
0x20400008 < start+8>: 0x6fc04117
(gdb) x 0x2040000a
0x2040000a < start+10>: 0x01136fc0
(gdb) jump main
Continuing at 0x80000000.

Breakpoint 1, main (argc=1, argv=0x80006b8d) at demo_gpio.c:239
239  {
(gdb) ■

```

图 4-10 GDB 显示程序停止于断点处

```

(gdb) ni
main (argc=1, argv=0x80006b8d) at demo_gpio.c:245
245      GPIO_REG(GPIO_OUTPUT_EN)  &= ~((0x1 << BUTTON_0_OFFSET) | (0x1 << BUTTON_1_OFFSET)
| (0x1 << BUTTON_2_OFFSET));
(gdb) ni
0x80000008      245      GPIO_REG(GPIO_OUTPUT_EN)  &= ~((0x1 << BUTTON_0_OFFSET) | (0x1 <<
BUTTON_1_OFFSET) | (0x1 << BUTTON_2_OFFSET));
(gdb) ni
0x8000000c      245      GPIO_REG(GPIO_OUTPUT_EN)  &= ~((0x1 << BUTTON_0_OFFSET) | (0x1 <<
BUTTON_1_OFFSET) | (0x1 << BUTTON_2_OFFSET));
(gdb) ni
239  {
(gdb) ni 10
250      GPIO_REG(GPIO_INPUT_EN)    &= ~((0x1<< RED_LED_OFFSET) | (0x1<< GREEN_LED_OFFSET)
| (0x1 << BLUE_LED_OFFSET)) ;
(gdb) ni
0x80000034      250      GPIO_REG(GPIO_INPUT_EN)    &= ~((0x1<< RED_LED_OFFSET) | (0x1<< GR
EEN_LED_OFFSET) | (0x1 << BLUE_LED_OFFSET)) ;
(gdb) ■

```

图 4-11 GDB 单步执行程序

4.5 其他软件工具

除了在 Linux 环境中使用命令行和脚本操作的 Freedom-E-SDK 软件平台之外，目前 RISC-V 还提供成熟的基于图形开发界面的集成开发环境（IDE，Integrated Development Environment），其中 SiFive 公司提供的免费 IDE 工具 Freedom Studio 便是优秀的代表。Freedom Studio 既有 Linux 版本，也有 Windows 版本，如图 4-12 所示。

由于 Linux 操作系统在工程领域更受推崇，其使用的命令行和脚本操作使得项目更加的具备可重现性和高效的自动化特性，因此本文更加推荐基于 Linux 环境的 Freedom-E-SDK 软件开发平台。有关基于 Windows 图形界面的 Freedom Studio 本文在此不做赘述，用户可以于 SiFive 网站(<http://www.sifive.com/>)自行查阅其使用详解。

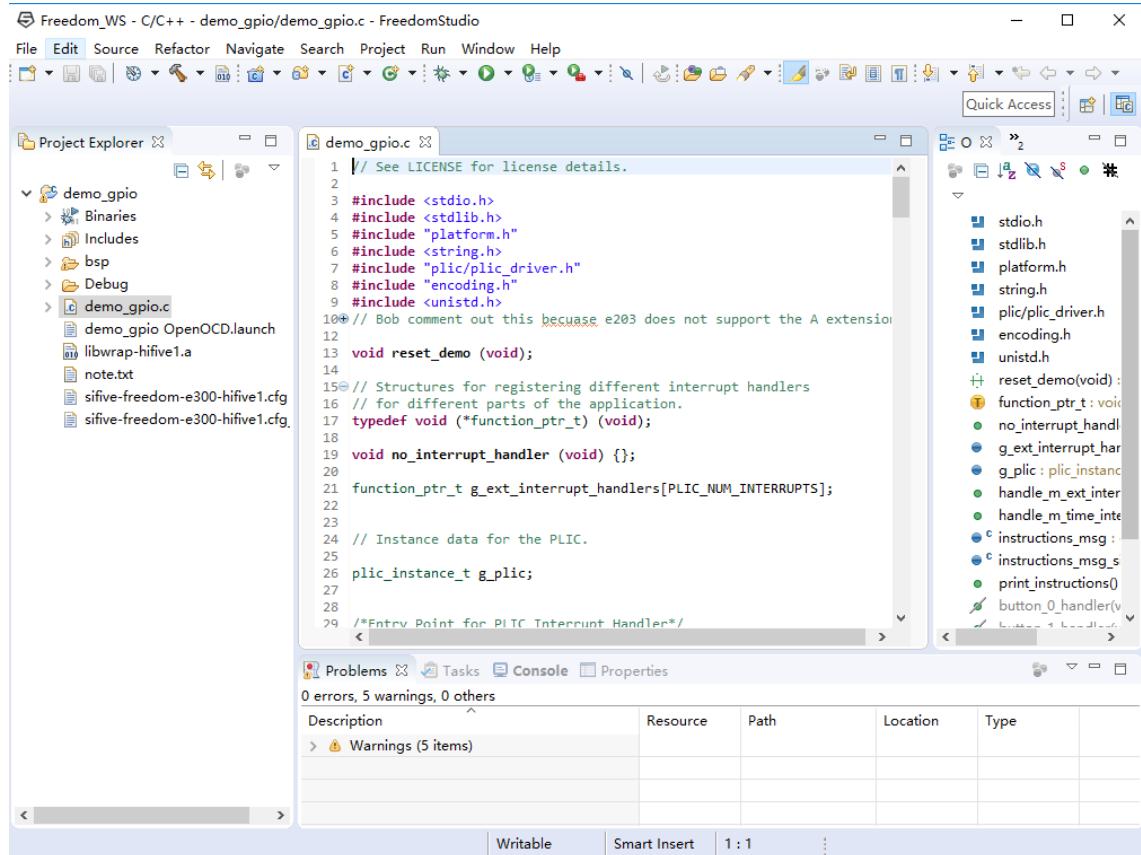


图 4-12 Windows 操作系统中带图形化开发界面的集成开发环境

5 运行 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。对于功耗的评估，其大致与处理器的硬件面积呈正比，蜂鸟 E200 是一款极为精简的小面积超低功耗处理器，面积要小于目前 ARM Cortex M 系列的最小的 Cortex M0+ 处理器核。而对于处理器性能的评估，需要依赖跑分程序（Benchmarks）来完成。

5.1 Benchmarks 简介

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。

5.2 运行 Dhystone Benchmark

Dhystone Benchmark 可运行于 Xilinx Arty 开发板（基于 SIRV-E200-SoC），使用 SIRV-E-SDK 平台按照如下步骤可以运行。

```
// 步骤一 ~ 步骤四：与第 4.3 节中描述的运行 demo_gpio 示例程序步骤一 ~ 步骤四相同。
```

```
// 步骤四：编译 Dhystone 示例程序，使用如下命令：
```

```
cd <your_e200_dir>/sirv-e-sdk
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。
make software PROGRAM=dhystone BOARD=sirv-e203-arty
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译
    // dhystone 示例程序。若需使用其他型号的 E200 系列 Core，譬如
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。
```

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

```
cd <your_e200_dir>/sirv-e-sdk
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。
```

```

make upload PROGRAM=dhystone BOARD=sirv-e203-arty
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载
    // dhystone 示例程序至 FPGA 开发板中。若需使用其他型号的 E200 系列 Core，譬如
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。
    // 该过程的原理是调用 GDB 和 OpenOCD 软件通过 JTAG 调试器将编译好的 dhystone
    // 程序下载到 E203 处理器核中。

```

// 步骤七：在 FPGA 原型平台上运行 Dhystone 程序：

```

// 由于 demo_gpio 示例程序将通过 UART 打印一个 Log 符号到主机 PC 的显示屏上。
// 因此需要先将串口显示终端准备好，在 Ubuntu 的命令行终端中使用如下命令。
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 3.3.2 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 Dhystone 程序，并将 Log 字符打印至主机 PC 的串口显示终端上。从其打印的结果我们可以看出 E203 处理器运行 Dhystone 跑分程序的结果 性能指标，如图 5-1 所示。

```

should be: (implementation-dependent)
Discr:      0
            should be: 0
Enum_Comp:  2
            should be: 2
Int_Comp:   17
            should be: 17
Str_Comp:   DHRYSTONE PROGRAM, SOME STRING
            should be: DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
Ptr_Comp:   -1879036832
            should be: (implementation-dependent), same as above
Discr:      0
            should be: 0
Enum_Comp:  1
            should be: 1
Int_Comp:   18
            should be: 18
Str_Comp:   DHRYSTONE PROGRAM, SOME STRING
            should be: DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:   5
            should be: 5
Int_2_Loc:   13
            should be: 13
Int_3_Loc:   7
            should be: 7
Enum_Loc:   1
            should be: 1
Str_1_Loc:   DHRYSTONE PROGRAM, 1'ST STRING
            should be: DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:   DHRYSTONE PROGRAM, 2'ND STRING
            should be: DHRYSTONE PROGRAM, 2'ND STRING

(*) User_Cycle for total run through Dhystone with loops 40000:
18040040
    So the DMIPS/MHz can be caculated by:
    10000000/(User_Cycle/Number_Of_Runs)/1757 = 1.261974 DMIPS/MHz

Progam has exited with code:0x00000000

```

图 5-1 E203 Core 运行 Dhystone Benchmark 程序后于主机 PC 的串口显示终端上显示分数

```
    should be: 5
Bool_Glob:
    should be: 1
Ch_1_Glob:
    should be: A
Ch_2_Glob:
    should be: B
Arr_1_Glob[8]:
    should be: 7
Arr_2_Glob[8][7]:
    should be: 40010
    should be: Number_Of_Runs + 10
Ptr_Glob->
    Ptr_Comp: -1879036832
    should be: (implementation-dependent)
Discr:
    should be: 0
Enum_Comp:
    should be: 2
Int_Comp:
    should be: 17
Str_Comp: DHRYSTONE PROGRAM, SOME STRING
    should be: DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
    Ptr_Comp: -1879036832
    should be: (implementation-dependent), same as above
Discr:
    should be: 0
Enum_Comp:
    should be: 1
Int_Comp:
    should be: 18
Str_Comp: DHRYSTONE PROGRAM, SOME STRING
    should be: DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:
    should be: 5
Int_2_Loc:
    should be: 13
Int_3_Loc:
    should be: 7
Enum_Loc:
    should be: 1
Str_1_Loc: DHRYSTONE PROGRAM, 1'ST STRING
    should be: DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc: DHRYSTONE PROGRAM, 2'ND STRING
    should be: DHRYSTONE PROGRAM, 2'ND STRING

(*) User_Cycle for total run through Dhystone with loops 40000:
19440052
    So the DMIPS/MHz can be caculated by:
    1000000/(User_Cycle/Number_Of_Runs)/1757 = 1.171091 DMIPS/MHz

Program has exited with code:0x00000000
```

图 5-2 E201 Core 运行 Dhystone Benchmark 程序后于主机 PC 的串口显示终端上显示分数

```

    should be: 1
Ch_1_Glob:      A
    should be: A
Ch_2_Glob:      B
    should be: B
Arr_1_Glob[8]:   7
    should be: 7
Arr_2_Glob[8][7]: 1000010
    should be: Number_Of_Runs + 10
Ptr_Glob->
    Ptr_Comp:      -1879036832
    should be: (implementation-dependent)
Discr:          0
    should be: 0
Enum_Comp:      2
    should be: 2
Int_Comp:       17
    should be: 17
Str_Comp:       DHRYSTONE PROGRAM, SOME STRING
    should be: DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
    Ptr_Comp:      -1879036832
    should be: (implementation-dependent), same as above
Discr:          0
    should be: 0
Enum_Comp:      1
    should be: 1
Int_Comp:       18
    should be: 18
Str_Comp:       DHRYSTONE PROGRAM, SOME STRING
    should be: DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:       5
    should be: 5
Int_2_Loc:       13
    should be: 13
Int_3_Loc:       7
    should be: 7
Enum_Loc:        1
    should be: 1
Str_1_Loc:       DHRYSTONE PROGRAM, 1'ST STRING
    should be: DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:       DHRYSTONE PROGRAM, 2'ND STRING
    should be: DHRYSTONE PROGRAM, 2'ND STRING

(*) User_Cycle for total run through Dhystone with loops 100000:
42000040
    So the DMIPS/MHz can be caculated by:
    1000000/(User_Cycle/Number_Of_Runs)/1757 = 1.355122 DMIPS/MHz

Program has exited with code:0x00000000

```

图 5-3 E204 Core 运行 Dhystone Benchmark 程序后于主机 PC 的串口显示终端上显示分数

图 5-1, 图 5-2 与图 5-3 分别的显示了 E201, E203 和 E204 Core 运行 CoreMark 所得跑分, 从中可以看出 E204 得分 1.355 DMIPS/Hz 高于 E201 的分数 1.171 DMIPS/Hz。这是因为 E204 Core 中使用了单周期的硬件乘法器和多周期的硬件除法器, 而 E201 Core 并没有硬件的乘法器和除法器。

5.3 运行 CoreMark Benchmark

CoreMark Benchmark 的运行方法类似，步骤如下。

```
// 步骤一 ~ 步骤四：与第 4.3 节中描述的运行 demo_gpio 示例程序步骤一 ~ 步骤四相同。  
  
// 步骤五：下载 CoreMark 的源代码  
// 由于未经 EEMBC 运行不得擅自转发 CoreMark 程序的源代码，因此需要用户自行于  
// EEMBC 网站上 (http://www.eembc.org/coremark/download.php) 下  
// 载 CoreMark 的程序源代码。  
// 将下载压缩包中的如下文件拷贝至 sirv-e-sdk/software/coremark 目录下。  
    core_list_join.c  
    core_main.c  
    coremark.h  
    core_matrix.c  
    core_state.c  
    core_util.c  
  
// 步骤六：编译 CoreMark 示例程序，使用如下命令：  
  
cd <your_e200_dir>/sirv-e-sdk  
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。  
  
make software PROGRAM=coremark BOARD=sirv-e203-arty  
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包编译  
    // coremark 示例程序。若需使用其他型号的 E200 系列 Core，譬如  
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。  
  
// 步骤七：将编译好的 CoreMark 程序下载至 FPGA 原型开发板中，使用如下命令：  
  
cd <your_e200_dir>/sirv-e-sdk  
    // 再次进入到 e200_opensource 目录文件夹下面的 sirv-e-sdk 目录。  
  
make upload PROGRAM=coremark BOARD=sirv-e203-arty  
    // 运行该命令指明需要使用 bsp/env 目录下的 sirv-e203-arty 板级支持包下载  
    // coremark 示例程序至 FPGA 开发板中。若需使用其他型号的 E200 系列 Core，譬如  
    // E225fd 处理器核，则使用 BOARD=sirv-e225fd-arty。  
    // 该过程的原理是调用 GDB 和 OpenOCD 软件通过 JTAG 调试器将编译好的 coremark  
    // 程序下载到 E203 处理器核中。  
  
// 步骤八：在 FPGA 原型平台上运行 CoreMark 程序：  
  
// 由于 demo_gpio 示例程序将通过 UART 打印一个 Log 符号到主机 PC 的显示屏上。  
// 因此需要先将串口显示终端准备好，在 Ubuntu 的命令行终端中使用如下命令。  
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 3.3.2 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 CoreMark 程序，并将 Log 字符打印至主机 PC 的串口显示终端上。从其打印的结果我们可以看出 E203 处理器运行 CoreMark 跑分程序的结果 性能指标如图 5-4 所示。

```

2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks       : 179668118
Total time (secs): 21
Iterations/Sec    : 18
Iterations        : 400
Compiler version  : GCC6.1.0
Compiler flags    : -O2 -fno-common -funroll-loops -finline-functions --param max-inline-insns
                   -auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4
Memory location   : STACK
seedcrc          : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x25b5
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 18 / GCC6.1.0 -O2 -fno-common -funroll-loops -finline-functions --param max-in
line-insns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4 / STACK

Print Personal Added Addtional Info to Easy Visual Analysis
(*) Assume SIRV (Silicion Integrated Developed RISC-V Core) running at 1 MHz
So the CoreMark/MHz can be caculated by:
(Iterations*1000000/total_ticks) = 2.226327 CoreMark/MHz

Program has exited with code:0x00000000

```

图 5-4 E203 Core 运行 CoreMark Benchmark 程序后于主机 PC 的串口显示终端上显示分数

```

2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks       : 295853718
Total time (secs): 35
Iterations/Sec    : 11
Iterations        : 400
Compiler version  : GCC6.1.0
Compiler flags    : -O2 -fno-common -funroll-loops -finline-functions --param max-inline-insns
                   -auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4
Memory location   : STACK
seedcrc          : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x25b5
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 11 / GCC6.1.0 -O2 -fno-common -funroll-loops -finline-functions --param max-in
line-insns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4 / STACK

Print Personal Added Addtional Info to Easy Visual Analysis
(*) Assume SIRV (Silicion Integrated Developed RISC-V Core) running at 1 MHz
So the CoreMark/MHz can be caculated by:
(Iterations*1000000/total_ticks) = 1.352019 CoreMark/MHz

Program has exited with code:0x00000000

```

图 5-5 E201 Core 运行 CoreMark Benchmark 程序后于主机 PC 的串口显示终端上显示分数

```

core running at Arty7 FPGA freq at 8388000 Hz
2K performance run parameters for coremark.
CoreMark Size      : 666
Total ticks       : 120224918
Total time (secs): 14
Iterations/Sec    : 27
Iterations        : 400
Compiler version  : GCC6.1.0
Compiler flags    : -O2 -fno-common -funroll-loops -finline-functions --param max-inline-inssns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4
Memory location   : STACK
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0x25b5
Correct operation validated. See readme.txt for run and reporting rules.
CoreMark 1.0 : 27 / GCC6.1.0 -O2 -fno-common -funroll-loops -finline-functions --param max-inline-inssns-auto=20 -falign-functions=4 -falign-jumps=4 -falign-loops=4 / STACK

Print Personal Added Addtional Info to Easy Visual Analysis
(*) Assume SIRV (Silicion Integrated Developed RISC-V Core) running at 1 MHz
So the CoreMark/MHz can be caculated by:
(Iterations*1000000/total_ticks) = 3.327097 CoreMark/MHz

Program has exited with code:0x00000000

```

图 5-6 E204 Core 运行 CoreMark Benchmark 程序后于主机 PC 的串口显示终端上显示分数

图 5-4, 图 5-5 与图 5-6 分别的显示了 E201, E203 和 E204 Core 运行 CoreMark 所得跑分, 从中可以看出 E204 得分 3.327 CoreMark/Hz 远远高于 E201 的分数 1.352 CoreMark/Hz。这是因为 E204 Core 中使用了单周期的硬件乘法器和多周期的硬件除法器, 而 E201 Core 并没有硬件的乘法器和除法器。

5.4 总结与比较

注意: 由于 Dhrystone 和 CoreMark 均只使用了整数运算类型, 因此并不能衡量浮点运算处理性能。

蜂鸟 E200 处理器核的 Benchmark 分数与主流的 ARM Cortex M 系列处理器的 Benchmark 分数对比如表 5-1 所示。

蜂鸟 E200 处理器核的面积仅仅和 ARM 最小的 Cortex M0+ 处理器核相当, 但通过表 5-1 的 Benchmark 分数可以看出, 蜂鸟 E203 功耗面积和性能均优于 ARM 的 Cortex M0+ 处理器核 (M0+ 是 ARM 最小面积的处理器核心), 蜂鸟 E204 功耗面积和性能均优于 ARM 的 Cortex M3 处理器核。

表 5-1 蜂鸟 E200 系列处理器核与 ARM Cortex M0+处理器核的 Benchmark 结果对比

Benchmarks	ARM Cortex-M0	ARM Cortex-M0+	ARM Cortex-M3	蜂鸟 E201	蜂鸟 E203	蜂鸟 E204
Dhrystone (DMIPS/MHz)	0.84 (Official) 1.21 (Max options)	0.94 (Official) 1.31 (Max options)	1.25	1.171	1.262	1.355
CoreMark (CoreMark/MHz)	2.33	2.42	3.32	1.352	2.226	3.327
最小配置逻辑门数 (K Gates)	12K	12K	36K	10K	12K	20K

注：

- (1) Cortex M0+的乘法器可以配置成单周期乘法器或多周期迭代乘法器，因此其 Dhrystone 性能数据有两组；CoreMark 性能数据采用单周期乘法或多周期乘法器信息不详。
- (2) 本表格中有关 ARM Cortex M 系列处理器核的性能数据来自于其他公开信息，非官方数据。请以最新 ARM 官方数据为准。