Nuclei™ N100 系列

处理器内核快速集成手册

版权声明

联系我们

若您有任何疑问，请通过电子邮件 support@nucleisys.com 联系芯来科技。

修订历史

版本号	修订日期	修订的章节	修订的内容
1.0	2019/5/10	N/A	1. 初始版本
2.0	2019/9/4	2	1. 修改文件包层次结构 2. 修改集成说明
3.0	2019/12/20	2.1,2.12	1. 添加逻辑综合目录和步骤
4.0	2019/1/10	N/A	1. 修订了若干笔误，重塑了文档章节结构

目录

表格清单

图片清单

1. 交付文件介绍

1.1. 交付文件压缩包

N100 交付的文件内容为一个文件压缩包，简介如表 1-1 所示。

表 1-1 交付文件压缩包介绍

文件包	内容简介	详细内容
n100_rls_pkg.tar.gz	包含交付 Verilog RTL 加密源代码，代码配置与生成工具、配套 Testbench、配套 SoC 原型、仿真环境与 FPGA 原型的源码文件压缩包。	详细内容介绍请参见本文档后续章节。

Nuclei N100 系列交付的文件正式版可通过与芯来公司取得联系授权获得。用户可以在芯来科技官网免费获取评估版。用户在得到压缩包后，可使用如下命令在 Linux 系统中进行解压，生成 n100_rls_pkg 文件夹：

```
tar -xzvf n100_rls_pkg.tar.gz
```

下文将介绍 Nuclei N100 系列交付的 n100_rls_pkg 文件压缩包的详细内容。

1.2. 文件包层次结构

n100_rls_pkg 文件包的文件层次结构如下所示。

```
n100_rls_pkg
    |n100                          // 存放 配置与生成工具 以及 RTL 的目录
        |----design                 // N100 核和配套 SoC 原型的 RTL 目录
            |----core               // 存放 N100 Core 的 加密 RTL 代码
```

```
        |----fab              // 存放配套 SoC 总线 bus fabric 的 RTL 代码
        |----subsys           // 存放配套 SoC 子系统文件的 RTL 代码
        |----mems             // 存放配套 SoC 的 memory 模块的 RTL 代码
        |----perips           // 存放配套 SoC 外设 peripherals 模块的 RTL 代码
        |----soc              //存放配套 SoC 顶层文件的 RTL 代码
    |----configs              // 配置与生成工具
        |----core_gen         // core_gen 工具，实现配置以及代码生成，
                              //   见第 4 章，了解该工具的使用
        |----env.sh           // bash 环境变量设置脚本
        |----env.csh          // csh 环境变量设置脚本
        |----private.pem      // 解密所用私钥（正式版需联系芯来科技，单独提供）
        |----check.txt        // 环境检查文件
        |----libc.so.6        // 库依赖文件
        |----rtl.vf           // Core 的 Verilog RTL file list
|n100_cct
    |----riscv-tests          // 存放一些测试用例的目录
    |----tb                   // 存放 Verilog TestBench （测试平台）的目录
    |----vsim                 // 运行 Verilog 仿真的目录
                                  参见第 5 章了解如何进行 Verilog 仿真。
        |----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
                                  参见第 3.2 节了解如何进行 FPGA 构建整个 SoC 原型。
    |----syn                    //存放综合脚本的目录。
```

注意：

- 以上 N100 只是代称，针对具体的发布型号，会换成对应 core 的名字，如 N101。

1.3. Core 源代码的命名规则

N100 系列不同型号的 Core 的源代码文件名前缀不一样，譬如 N101 内核的文件名和模块名的前缀均为“n101_”，其他型号的核前缀同理。

1.4. Core 的模块层次结构

以 N101 处理器内核为例，其模块层次的划分如图 1-1 所示，要点如下。

- n101_core_wrapper 为整个处理器内核的顶层，包含如下主要组件
  - n101_core：为处理器内核的总体部分。
  - n101_rst_ctrl：用于将外界的异步复位信号进行同步使之变成“异步置位同步释放”的复位信号。
  - n101_dbg_top：处理 JTAG 接口和相关的调试功能。
- n101_ucore 位于 n101_core 层次之下，为处理器内核的主体逻辑部分。
- 除了 n101_ucore 之外，在 n101_core 层次结构之下还包含了如下主要组件：
  - n101_clk_ctrl：用于控制处理器各个主要组件的自动时钟门控。
  - n101_clic_top：内核私有的中断控制单元。
  - n101_tmr_top：内核计时器单元。

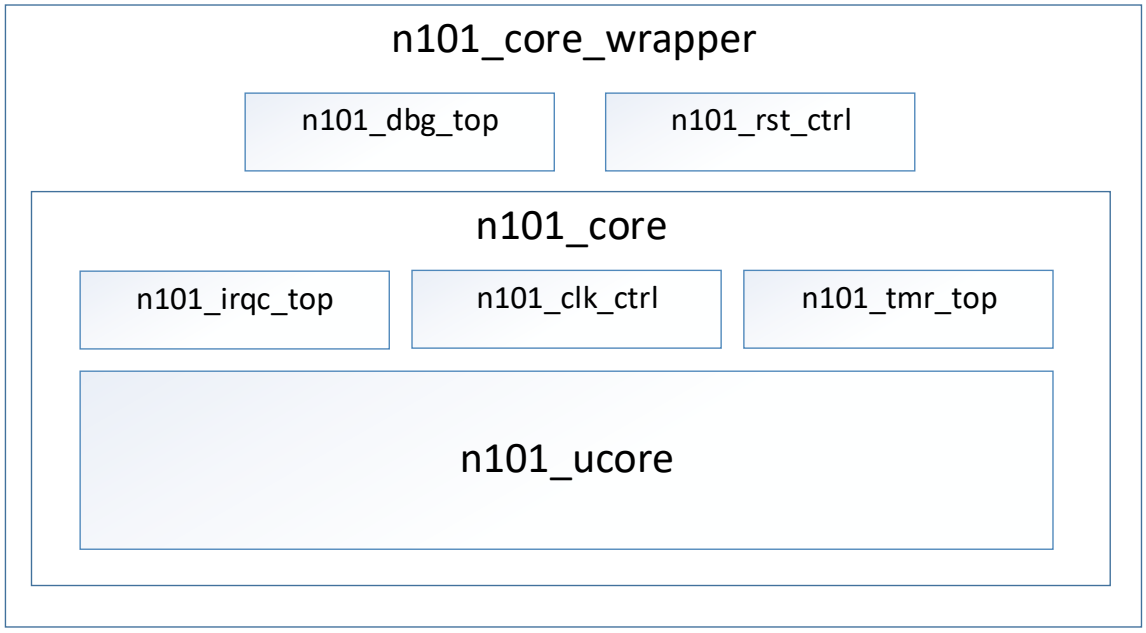


图 1-1 N101 处理器内核的设计模块结构

2. CPU 顶层集成

本节对 N100 系列处理器内核集成至 SoC 中需要注意的若干方面进行简要介绍。

2.1. 时钟关系

有关 N100 系列处理器内核的时钟关系，请参见《Nuclei_N100 系列简明数据手册》中的“N100 系列时钟域介绍”章节。

2.2. 接口关系

有关 N100 系列处理器内核的接口描述，请参见《Nuclei_N100 系列简明数据手册》中的“N100 系列接口简介”章节。

2.3. 地址映射

有关 N100 系列处理器内核的存储器地址映射分配，请参见《Nuclei_N100 系列简明数据手册》中的“N100 系列地址空间分配”章节。

3. 配套 SoC 原型与软件开发环境

3.1. 配套 SoC 原型

如果仅仅交付处理器内核而没有配套 SoC，那么为了能够使用该内核，用户需要花费不少精力来构建完整的 SoC 平台、FPGA 平台。为了方便用户快速地上手使用，N100 系列内核配套了完整的简单 SoC 原型，如图 3-1 所示（以 N101 内核为例）。基于此 SoC 原型，可以快速实现完整的 SoC 原型软硬件平台，有关此配套 SoC 的详细介绍请参见单独文档《Nuclei_N100 系列配套 SoC 介绍》。

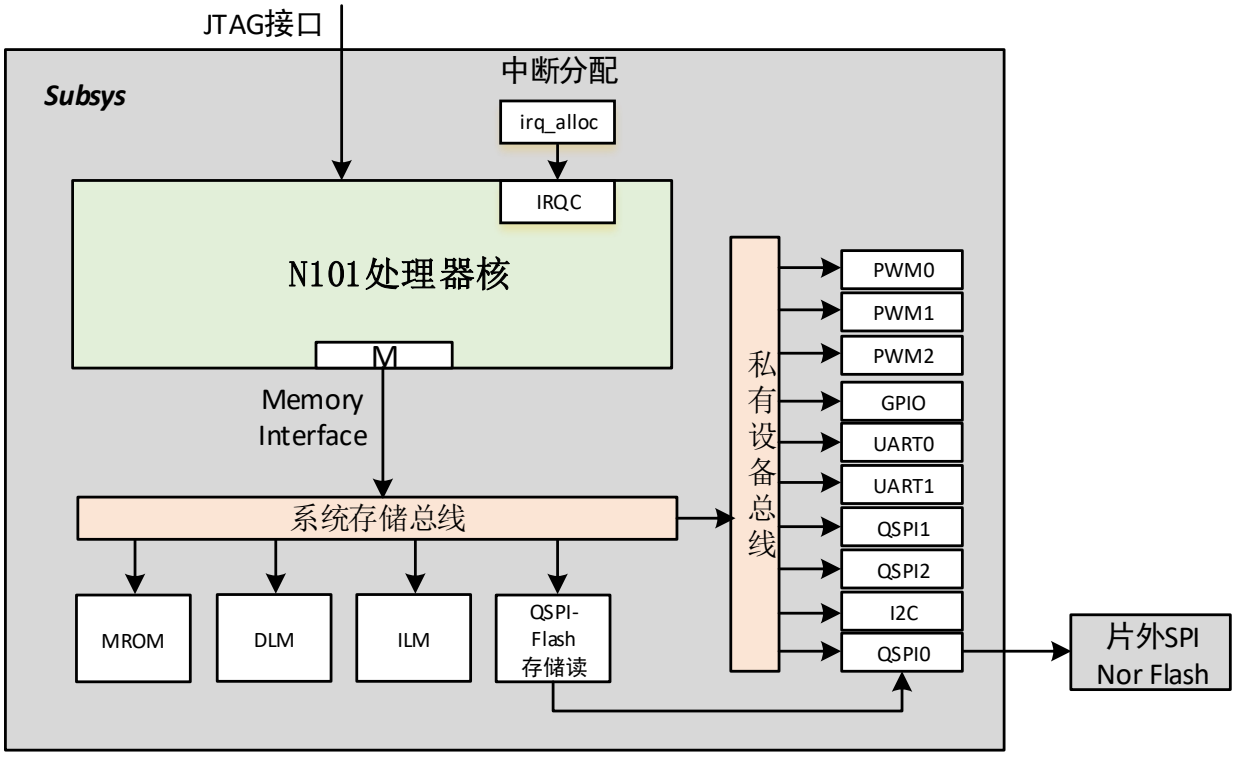


图 3-1 N100 系列处理器内核（以 N101 为例）的配套 SoC 结构

3.2. 配套 FPGA 评估板和 JTAG 调试器

芯来科技定制了专用的 FPGA 评估板（HBird Eval Kit）。上节描述的配套 SoC 原型（包含 N100 系列处理器内核）可以被整体实现在 HBird Eval Kit 上成为 SoC FPGA 原型开发板。芯来科技还定制了专用的 JTAG 调试器（HBird Debugger Kit），用于在 FPGA 评估板上对 N100 系列进行调试。

有关芯来科技定制的专用 JTAG 调试器（HBird Debugger Kit）和专用 FPGA 评估板（HBird Eval Kit）的详细介绍请参见参见单独文档《Nuclei_N100 系列配套 FPGA 实现》。

3.3. 配套软件开发套件（SDK）

N100 系列提供软件开发套件（SDK，Software Development Kit）。用户可以基于上述的 SoC FPGA 原型开发板，结合 SDK，进行完整的嵌入式软件开发和调试，详细介绍请参见参见单独文档《Nuclei_N100 系列 SDK 使用说明》。

注意：SDK 使用 Linux Makefile 进行项目和文件的组织，适用于开发水平较高的用户。

3.4. 配套可视化软件集成开发环境（IDE）

N100 系列提供图形化集成开发环境（IDE，Integrated Development Environment）。用户可以基于上述的 SoC FPGA 原型开发板，结合 IDE，进行完整的嵌入式软件开发和调试，详细介绍请参见参见单独文档《Nuclei_N100 系列 IDE 使用说明》。

注意：IDE 使用可视化集成开发环境进行项目和文件的组织，适用于习惯图形化开发环境的用户。

4. 可配置代码的生成与编译

4.1. core_gen 工具操作说明

为了方便客户，比较容易的配置 N100 系列 core，芯来科技开发了 core_gen 工具，实现对 core 的配置以及代码的生成。

在 n100_rls_pkg/n100/configs 目录下，有如下文件：

- core_gen：core_gen 工具
- private.pem：代码解密所需 key（需联系芯来科技，单独提供）
- check.txt：环境检查文件
- env.sh：环境变量设置脚本
- env.csh：csh 环境变量设置脚本
- libc.so.6：库依赖文件

注意：

- private.pem，check.txt，libc.so.6 文件请勿修改，以免代码无法正常生成

在执行 core_gen 工具之前，需要设置环境变量：

- bash 环境：source env.sh
- csh 环境：source env.csh

以上脚本（env.sh）会自动设置相应环境变量：

- PROJ_SRC_ROOT: n100_rls_pkg 所在目录
- PROJ_NAME: 对应 core 的名字
- PROJ_GEN_ROOT：RTL 代码生成目录，默认在 n100_rls_pkg/n100_cct 目录下，如果想将代码生成在其他目录，可以修改该环境变量。

设置完环境变量之后，直接执行 ./core_gen，启动代码配置与生成工具。启动之后，界面如下图所示。界面显示的配置选项，与《Nuclei_N100 系列简明数据手册》文档中“N100 系列内核配置参数选项”章节中的介绍一致。关于每个选项配置的说明，请参阅《Nuclei_N100 系列简明数据手册》。

以下以 n101 core 为例，对该工具使用进行说明。

图 4-1 core_gen 启动配置界面

选项后的特殊字符解释如下：

- 当选项后续有 --->，表示这个配置有子菜单，输入回车或者空格，进入到子菜单配置界面。
- 进入到子菜单配置界面后，可输入方向键“<”，回到上一层子菜单。

如进入 TIMER 子菜单，其配置界面如图 4-2 所示：

图 4-2 TIMER 配置子菜单

选项后的特殊字符解释如下：

- [*] 表示该配置，是被选择，输入空格，取消选择。
- [ ] 表示该配置，未被选择，输入空格，进行选择。
- -*- 表示该配置，是固定配置，不可配。
- () 中的内容，表示该选项的配置值。如果带有（NEW），表示此次配置是使用默认配置值，如果用户修改了配置值，则该（NEW）字符会消失。

对于一些选项，需要手动输入。如配置 DEBUG 基地址，如图 4-3 所示。在该配置选项上，输入回车或者空格，进入配置输入界面如图 4-4 所示。在配置输入界面，输入想要配置的地址值。输入回车，即配置成功。

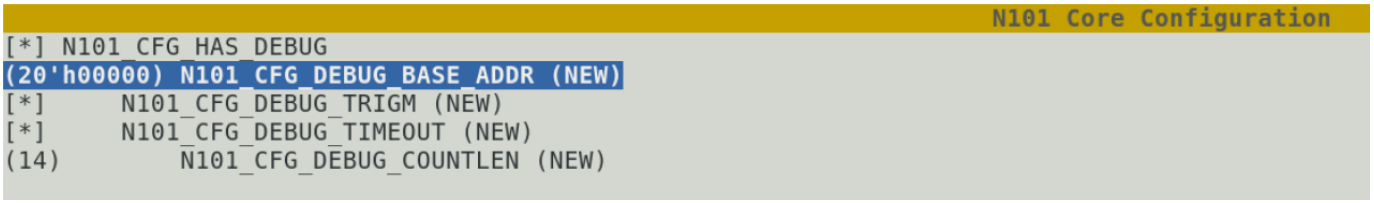


图 4-3 DEBUG 基地址配置

图 4-4 DEBUG 基地址输入配置

对于某些选项，需要手动输入，但是输入的条件有限制。如配置外部中断个数，如图 4-5 所示，Range 属性要求输入的值处于 1-29 范围之内。如果输入的值超过了要求的范围之内，会提示错误信息，如图 4-6 所示。

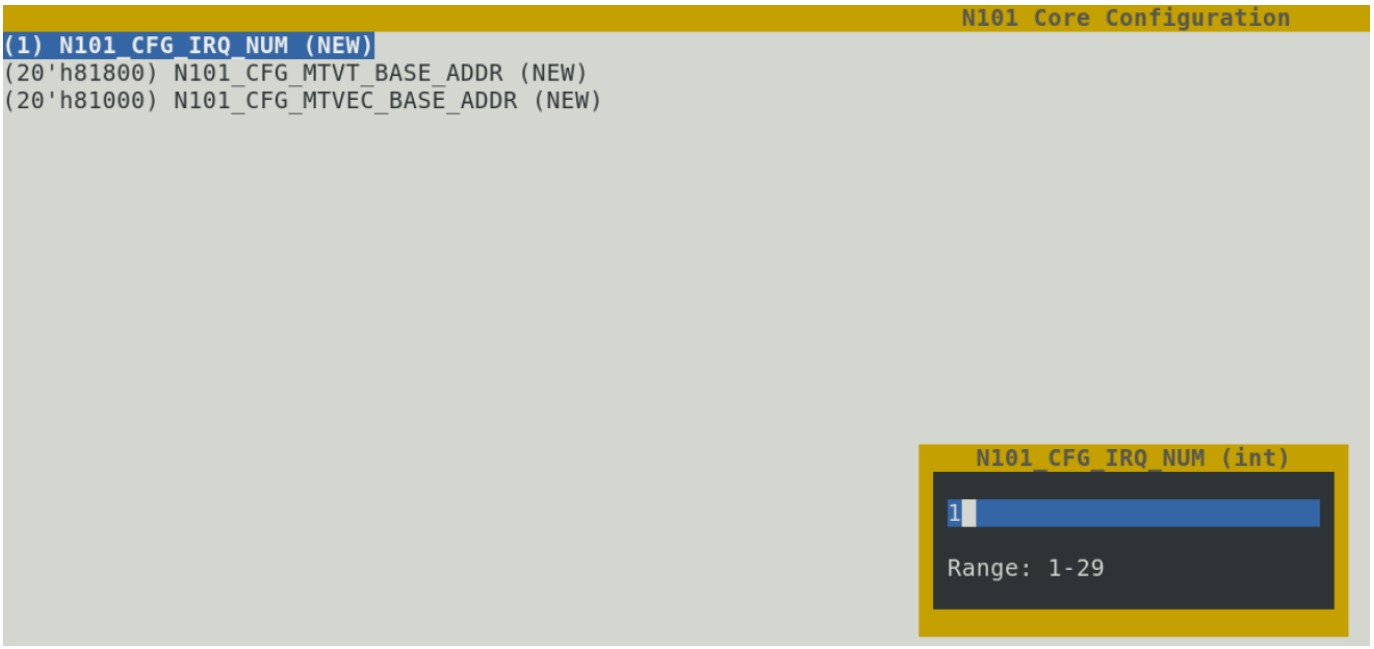


图 4-5 中断数配置

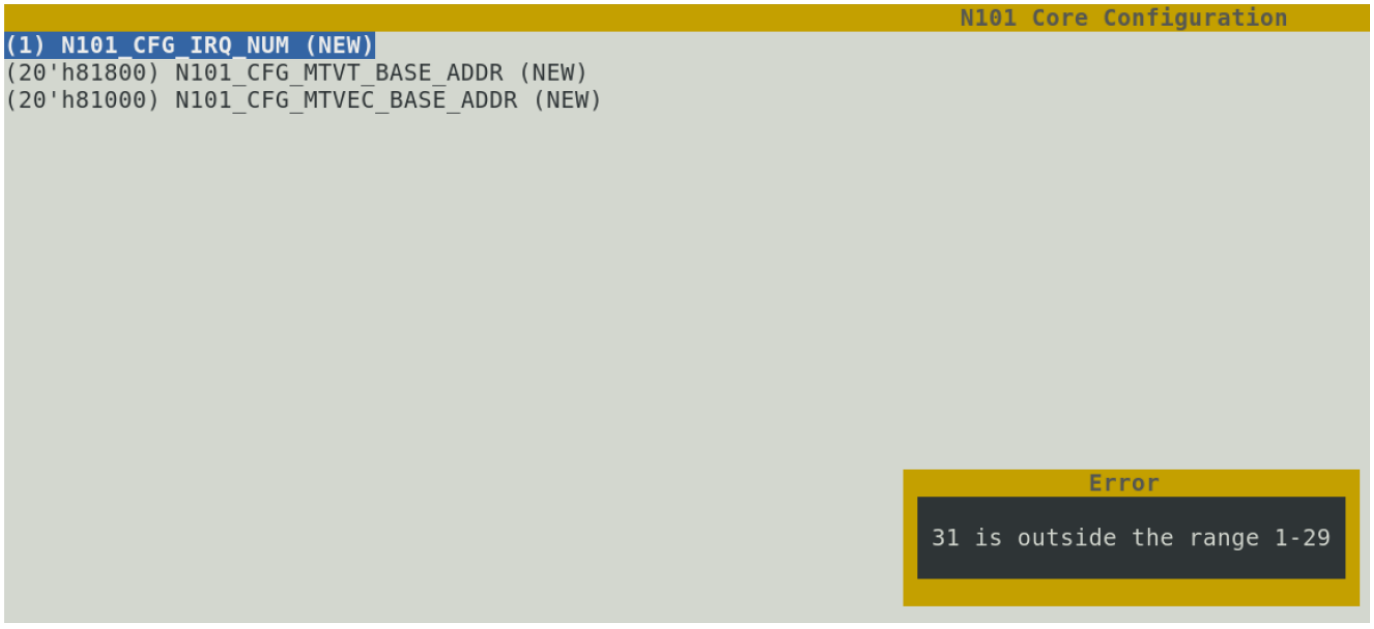


图 4-6 配置错误

配置完毕之后，输入 q，保存并退出。退出之后，core_gen 工具，会自动进行代码生成，代码生成需要花费几分钟，请耐心等待。生成的代码将放置在$PROJ_GEN_ROOT 目录下。注意：

- 代码配置生成完毕后，生成的代码将放置在$PROJ_GEN_ROOT 目录下，该目录里面包含了大量的 RTL 源代码，其中包含了处理器内核和配套 SoC 原型的所有可综合 Verilog RTL 源代码。如果仅仅需要内核相关的代码，则仅需关注 core 这个目录下的源代码。

配置完毕之后，会在当前目录下，生成 .config 文件。当下次在配置的时候，会加载该.config，继承上一次的配置。如果删除掉该文件，那么使用默认配置。

4.2. 查看和编译 Verilog RTL 代码

假设用户想快速查看 N100 系列处理器核配置生成的源代码，可以使用如下步骤进行。注意：

- 以下 N100 只是代称，正式版本发布的时候，可能换成对应 core 的名字，如 N101。

// 注意：在操作之前，需要安装 RISC-V GCC 编译工具链。该工具链可以从我们的官网进行下载。下载地址为：https://www.nucleisys.com/download.php，用户可点击该网页中 RISC-V GNU 工具链下的下载按钮进行下载。下载完毕后，解压到 Linux 系统中，解压后可以看到生成的文件夹下有一个 bin 子文件夹，用户需要将此 bin 子文件夹的路径添加到 PATH 环境变量中。

// 步骤一：按照上节所述使用 core_gen 工具进行配置和代码生成。

//首先将 n100_rls_pkg 解压至本机 Linux 环境中。然后配置并生成 RTL 代码，使用如下命令：

```
cd n100_rls_pkg/n100/configs
source env.sh
./core_gen
```

// 执行 core_gen 之后，会弹出图形化界面，用户在图形化界面上，配置 core 的参数
// 配置完毕后，按 q 退出并保存，之后开始进行代码生成
// 代码生成需要花费几分钟，请耐心等待
// 代码生成在 n100_rls_pkg/n100_cct/n100 目录中

// 步骤二：编译 RTL，使用如下命令：

```
cd n100_rls_pkg/n100_cct/vsim
    // 进入到 n100_cct/vsim 目录下

make install
    // 安装环境

make compile
```

```
    // 编译 RTL
```

// 步骤三：查看 RTL 代码，可使用如下两个命令：

```
make verilog
    // 该命令查看所有的 Verilog 源代码，会自动加载所有的 Testbench 和 Verilog 源代码（例化了整个配套 SoC 原型，包含处理器内核）。

make verilog_core
    // 该命令仅查看 Core 的 Verilog 源代码，会自动加载 n100_cct/n100 目录下的 RTL。
```

5. 仿真运行简单汇编测试用例

5.1. 自测试用例简介

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，存放于以下目录。

```
n100_rls_pkg
    |----n100_cct
        |----riscv-tests
                |----isa_origs      // 存放一些测试用例源码的目录
```

这些测试程序均由汇编语言编写，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 5-1 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相等则程序直接使用 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, 0xffffffffffffffff );
  TEST_RR_OP( 15, add, 0xfffffffffffffffe, 0xffffffffffffffff, 0xfffffffffffffffd );

  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 5-1 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。

5.2. 自测试用例文件

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 n100_cct 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。

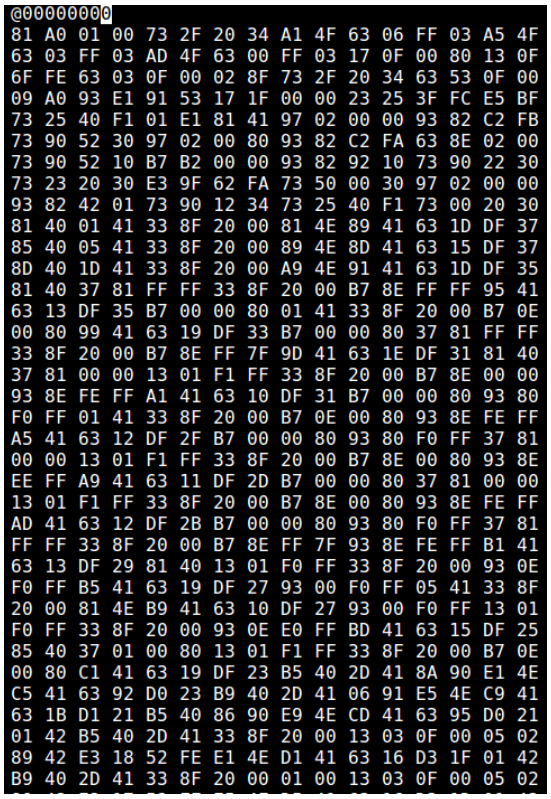
```
n100_cct
    |----riscv-tests        // 存放一些测试用例的目录
        |----isa
            |----generated                  // 编译好的 tests 文件夹
                |----rv32ui-p-addi          // 编译出的 elf 文件
                |----rv32ui-p-addi.dump     // 反汇编文件
                |----rv32ui-p-addi.verilog  // 可被 Verilog 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 5-2 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 5-2　rv32ui-p-addi 的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 5-3 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 5-3 Verilog 的 readmemh 函数可读入文件内容片段

5.3. 测试平台简介

在 n100_cct 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。

```
n100_cct
    |----tb                    // 存放 Verilog TestBench （测试平台）的目录
        |----tb_*.v            // 简单地 Verilog TestBench 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 SoC 的 Instruction Memory（由 Verilog 编写的二维数组充当 SRAM 行为模型）。
- 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 5-4 所示。

注意：用户在将 N100 系列集成在不同产品的 SoC 之中时，也可以将相关 tb_*.v 也集成在 SoC 中，以便于在 SoC 环境之中运行自测试用例。

```
@(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("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
end
else begin
    $display("~~~~~~~~~~~~~~~~~ TEST_FAIL ~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~######    ##       #    #     ~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#        #  #      #    #     ~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#####   #    #     #    #     ~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       ######     #    #     ~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       #    #     #    #     ~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~#       #    #     #    ######~~~~~~~~~~~~~~~~");
    $display("~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
end
```

图 5-4 Testbench 中打印测试用例的结果

5.4. 运行测试用例

假设用户想使用 N100 系列内核源代码运行基于 Verilog 的仿真测试程序，可以使用如下步骤进行。

// 注意：在运行测试用例之前，需要安装 RISC-V GCC 编译工具链。该工具链可以从我们的官网进行下载。下载地址为：https://www.nucleisys.com/download.php，用户可点击该网页中 RISC-V GNU 工具链下的下载按钮进行下载。下载完毕后，解压到 Linux 系统中，解压后可以看到生成的文件夹下有一个 bin 子文件夹，用户需要将此 bin 子文件夹的路径添加到 PATH 环境变量中。

// 步骤一：将 n100_rls_pkg 解压至本机 Linux 环境中。

// 步骤二：生成 test：

```
cd n100_rls_pkg/n100_cct/vsim
        // 进入到 n100_cct 目录文件夹下面的 vsim 目录。
```

```
make clean
        // 清除当前目录，以保证干净的工作目录。

make install
        // 运行该命令，会将 n100_cct 目录下的 test 进行编译，以及必要 tb 的生成。


// 步骤三：编译 RTL：

make compile
        // 编译所有的 Verilog 代码


// 步骤四：运行 testcase（测试用例），使用如下命令：
make run_test TESTNAME=rv32ui-p-add
        // 注意：make run_test 将执行
        // riscv-tests/isa/generated 目录中的一个 testcase “rv32ui-p-add”。
        // 如果希望运行所有的回归测试，请参见步骤四。

make wave TESTNAME=rv32ui-p-add
        //查看该 test 执行后的波形。


// 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 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
        // 该命令将收集步骤四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。
```

注意：

- 以上 N100 只是代称，真正发布的时候，会换成对应 core 的名字，如 N101。
- 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。

6. 仿真运行复杂 C 测试用例

6.1. 运行 C 程序

假设用户想在仿真环境中运行 C 语言编写的程序，那么需要借助 N100 系列的 SDK 进行。有关 SDK 的详细介绍请参见《Nuclei_N100 系列 SDK 使用说明》。

以 SDK 中的示例程序（Demo_irqc）为例，可以使用如下步骤进行。注意：下列步骤以 N101 为例，因此命令行中使用 CORE=n101。

// 步骤一：进入 n100-sdk 目录。

// 步骤二：在 n100-sdk 目录下对用于仿真的示例程序进行编译，使用如下命令：

```
make dasm PROGRAM=demo_irqc CORE=n101 DOWNLOAD=ilm SIMULATION=1
```

// 步骤三：进入 n100_rls_pkg 目录，将上述步骤生成的 software/demo_irqc 文件夹拷贝到 n100_rls_pkg/n100_cct/vsim 目录中，命令如下：

```
cp n100-sdk/software/demo_irqc  n100_rls_pkg/n100_cct/vsim -rf
```

// 步骤四：在 n100_cct 的 vsim 目录下对测试程序进行仿真。

```
cd n100_rls_pkg/n100_cct/vsim
make run_test TESTCASE=$PWD/demo_irqc/demo_irqc
    //运行拷贝过来的 demo_irqc 示例程序
```

7. 逻辑综合

7.1. 逻辑综合 Verilog 代码

如果需要对 N100 系列进行逻辑综合，可以按照以下步骤进行，以 N101 为例：

// 步骤一：进入 n101_cct/syn 目录。

// 步骤二：在 n101_cct/syn 目录下修改 Makefile 脚本，主要包括工艺库路径，Design 路径，频率等。

// 步骤三：综合脚本预处理

```
make install
    //将 Makefile 中的相关变量替换到综合脚本中。
```

// 步骤四：综合

```
make syn
    //最后的生成文件和报告文件在 syn_<CORE>_config_<freq>_<lib>/reports 下面。
```

上述综合仅仅给出最简单的综合参考环境。同时，如果要进行量产综合，在进行综合之前，需注意如下事项。

7.2. 注意事项

- 为了达到较好的面积和时序，推荐用户采用打平层次结构（Flatten Hierarchy）的方式进行逻辑综合。
- 处理器内核完全为纯数字电路逻辑。但是用户需要将源代码文件中的门控时钟逻辑替代成为具体工艺库下的门控时钟单元。
    - 以 N101 内核为例，时钟门控单元的源代码位于源文件的模块 n101_clkgate 中，请用户在 n101_rls_pkg/n101_cct/design/core 目录自行搜索该模块。