芯来科技
NUCLEI

Nuclei™ N200 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

目录

表格清单

图片清单

1. N200 系列交付简介

Nuclei N200 系列处理器的特性简介如下：

- N200 系列处理器核采用 2 级流水线结构，通过一流的处理器架构设计，实现业界最高的能效比与最低的成本。
- N200 系列处理器核支持 RISC-V 指令集，支持 RV32I/M/A/C/F/D 等指令子集的配置组合。
- N200 系列处理器核提供标准的 JTAG 调试接口，以及成熟的软件调试工具。
- N200 系列处理器核提供标准的 RISC-V 的 GCC 编译工具链，以及 Linux 与 Windows 图形化软件开发工具。
- N200 系列处理器核配套的原型 SoC 提供紧耦合系统 IP 模块，包括 UART、QSPI、PWM 等，以及 Ready-to-Use 的 SoC 平台与 FPGA 原型演示系统。

有关 Nuclei N200 系列的详细介绍请参见《Nuclei_N200 系列简明数据手册》。

1.1. N200 系列交付文档

Nuclei N200 系列交付的配套文档可通过芯来公司的网站下载或者与芯来公司取得联系授权获得。Nuclei N200 系列交付的配套文档如表 1-1 中所示。

表 1-1 交付配套文档

编号	文档名称	内容简介
0	《Nuclei_N200 系列快速集成手册》	本文档，介绍 N200 系列处理器内核 IP 进行交付后快速集成的使用说明。
1	《Nuclei_N200 系列简明数据手册》	介绍 Nuclei N200 系列处理器内核 IP 的详细内容。
2	《Nuclei_N200 系列指令架构手册》	介绍 Nuclei N200 系列处理器内核支持的指令集和架构详细内容。
3	《Nuclei_N200 系列快速应用手册》	介绍 Nuclei N200 系列处理器内核如何快速地进行应用程序开发。

4	《Nuclei_N200 系列配套 SoC 介绍》	介绍 Nuclei N200 系列处理器内核配套的原型 SoC 详细内容。
5	《Nuclei_N200 系列配套 FPGA 实现》	介绍 Nuclei N200 系列处理器内核配套的原型 FPGA 开发板详细内容。
6	《Nuclei_N200 系列 SDK 使用说明》	介绍 Nuclei N200 系列处理器内核配套的基于 Linux 环境的软件开发套件（Software Development Kit）。
7	《Nuclei_N200 系列 IDE 使用说明》	介绍 Nuclei N200 系列处理器内核配套的基于 Windows 环境的 Eclipse IDE。
8	《Nuclei_N200 系列 NICE 使用说明》	介绍 Nuclei N200 系列处理器内核如何进行自定义指令的扩展和使用。

1.2. N200 系列交付代码

N200 交付的文件内容为一个文件压缩包，简介如表 1-2 所示。

表 1-2 交付文件压缩包介绍

文件包	内容简介	详细内容
n200_rls_pkg.tar.gz	包含交付 Verilog RTL 源代码，配套 Testbench、配套原型 SoC、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第 2 章。

Nuclei N200 系列交付的文件压缩包可通过与芯来公司取得联系授权获得。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 n200_rls_pkg 文件夹：

```
tar -xzvf n200_rls_pkg.tar.gz
```

2. N200 系列文件包详细介绍

本章将介绍 Nuclei N200 系列交付的 n200_rls_pkg 文件压缩包的详细内容。

2.1. 文件包层次结构

n200_rls_pkg 文件包的文件层次结构如下所示（以 N201 内核为例）。

```
n200_rls_pkg
    |----rtl                    // 存放 RTL 的目录
        |----n201               // N201 核和配套原型 SoC 的 RTL 目录
            |----core           // 存放 N201 Core 的 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
    |----tb                     // 存放 Verilog TestBench （测试平台）的目录
        |----tb_*.v             // 简单的 Verilog TestBench 顶层文件
    |----vsim                   // 运行 Verilog 仿真的目录。
                                    参见第 2.10 节了解如何进行 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 2.6 节了解如何进行 FPGA 重现整个原型 SoC。
    |----riscv-tools
        |----riscv-tests        // 存放一些测试用例的目录
```

注意：

- 在上述 rtl 目录的文件夹（譬如 n201）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原型 SoC 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。

2.2. Core 源代码的命名规则

N200 系列不同型号的 Core 的源代码文件名前缀不一样，譬如 N201 内核的文件名和模块名的前缀均为“n201_”，其他型号的核前缀同理。

2.3. Core 的模块层次结构

以 N201 处理器内核为例，其模块层次的划分如图 2-1 所示，要点如下。

- n201_core 为整个处理器内核的顶层。
- n201_ucore 位于 n201_core 层次之下，为处理器内核的主体逻辑部分。
- 除了 n201_ucore 之外，在 n201_core 层次结构之下还包含了如下主要组件：
  - n201_clk_ctrl：用于控制处理器各个主要组件的自动时钟门控。
  - n201_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。
  - n201_dbg_top：处理 JTAG 接口和相关的调试功能。
  - n201_clic_top：内核私有的中断控制单元。
  - n201_tmr_top：内核计时器单元。

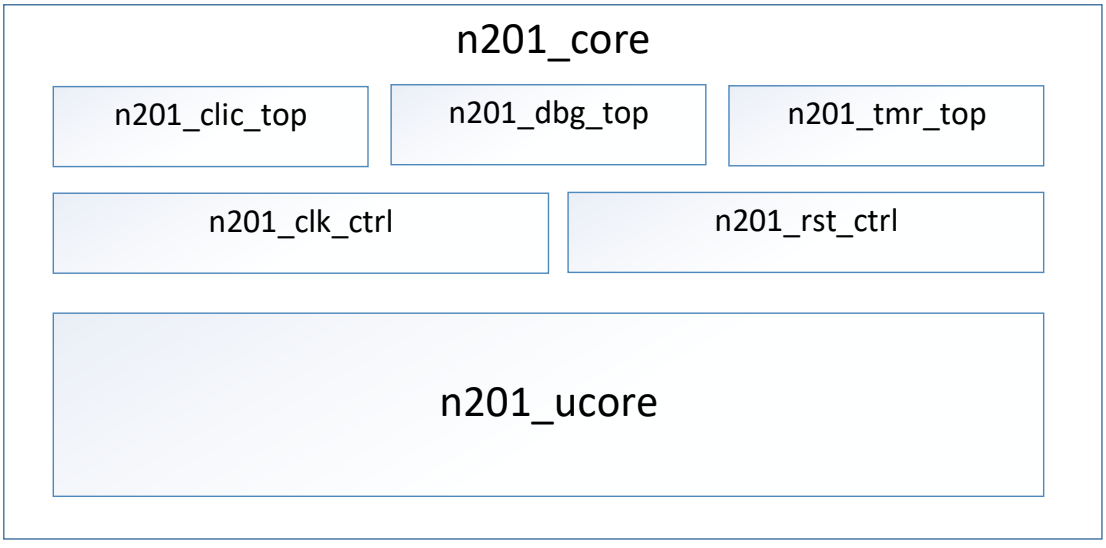

图 2-1 N201 处理器内核的设计模块结构

2.4. CPU 顶层集成

本节对 N200 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。

2.4.1. 时钟关系

有关 N200 系列处理器内核的时钟关系，请参见《Nuclei_N200 系列简明数据手册》中的“N200 系列时钟域介绍”章节。

2.4.2. 接口关系

有关 N200 系列处理器内核的接口描述，请参见《Nuclei_N200 系列简明数据手册》中的“N200 系列接口简介”章节。

2.4.3. 地址映射

有关 N200 系列处理器内核的存储器地址映射分配，请参见《Nuclei_N200 系列简明数据手册》中的“N200 系列地址空间分配”章节。

2.5. 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，N200 系列内核配套了完整的简单原型 SoC，如图 2-2 所示（以 N201 内核为例）。基于此原型 SoC，可以快速实现完整的 MCU 原型 SoC 平台，有关此配套 SoC 的详细介绍请参见单独文档《Nuclei_N200 系列配套 SoC 介绍》。

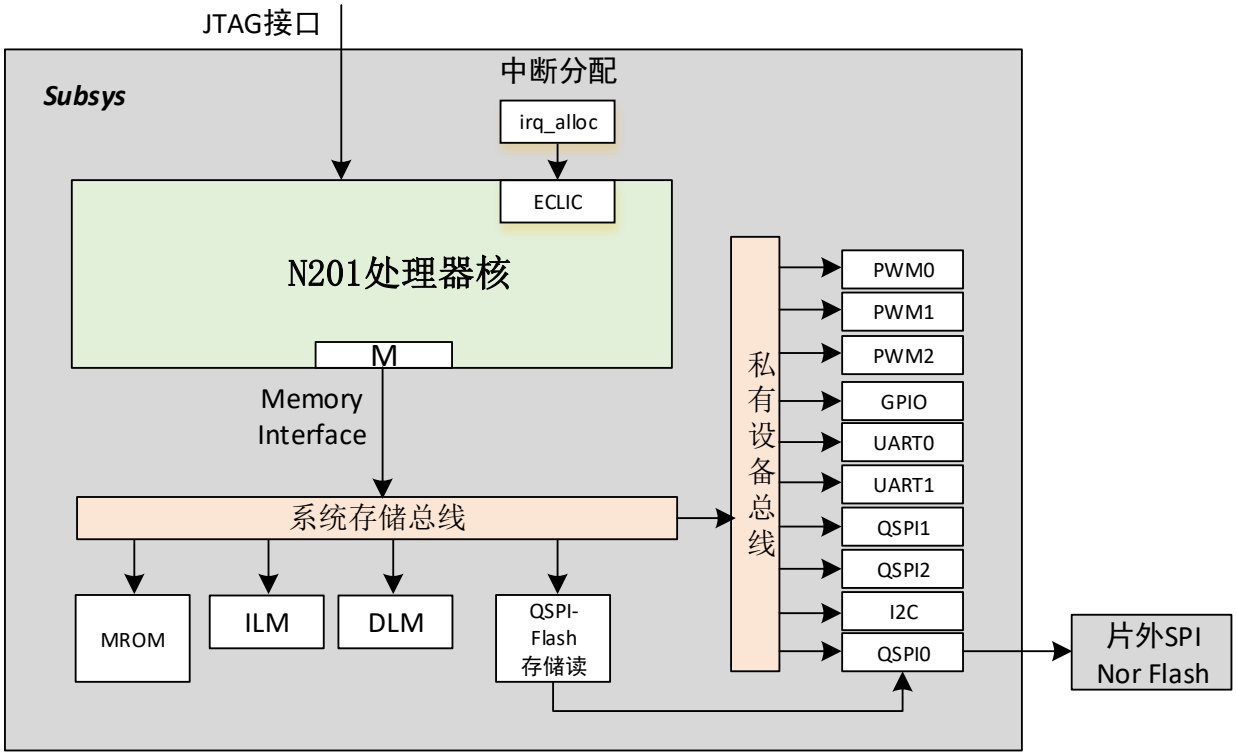

图 2-2 N200 系列处理器内核（以 N201 为例）的配套 SoC 结构

2.6. 配套 FPGA 原型

基于上节描述的配套原型 SoC，N200 系列定制了专用的 FPGA 开发板。N200 系列和配套原型 SoC 可以被整体实现在 FPGA 开发板上成为 MCU 原型平台。N200 系列还定制了专用的 JTAG 调试器，用于在 FPGA 开发板上对 N200 系列进行调试。

有关 N200 系列定制的专用 JTAG 调试器和专用 FPGA 开发板的详细介绍请参见参见单独文档《Nuclei_N200 系列配套 FPGA 实现》。

2.7. 配套基于 Linux 的 SDK

基于配套的原型 SoC，N200 系列提供基于 Linux 的软件开发套件（SDK，Software Development Kit），详细介绍请参见参见单独文档《Nuclei_N200 系列 SDK 使用说明》。

2.8. 配套基于 Windows 的 IDE

基于上述配套的原型 SoC，提供基于 Windows 的图形化集成开发环境（IDE，Integrated Development Environment），详细介绍请参见参见单独文档《Nuclei_N200 系列 IDE 使用说明》。

2.9. 查看和编译 RTL 代码

假设用户想快速查看 N200 系列处理器核的源代码，可以使用如下步骤进行。注意：下列步骤以 N201 为例，因此命令行中使用 CORE=n201。如果是其他型号的 Core（譬如 N205），则仅需更改 CORE 的参数，譬如 CORE=n205。

// 步骤一：将 n200_rls_pkg 解压至本机 Linux 环境中。

// 步骤二：生成并编译 RTL 代码，使用如下命令：

```
cd n200_rls_pkg/vsim
        // 进入到 n200_rls_pkg 目录文件夹下面的 vsim 目录。

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

make install CORE=n201
        // 运行该命令会在 vsim 目录下生成一个 install 子文件夹，在其中放置所需的 testbench
        // 文件，且将 testbench 中的关键字设置为 n201。
        //
        // 注意：
        //    (1) 该命令会在 vsim 目录下生成一个 run 子文件夹，在此文件夹下生成一个名为
        //          “rtlonly_flist”的文件，里面列举了所有的 Core 相关的 RTL 源代码文件。
        //
```

// 步骤三：查看 RTL 代码，使用如下命令：

```
make verilog CORE=n201
        // 查看所有的 Verilog 源代码，该命令会自动加载所有的 Testbench 和 Verilog 源代码（例化了整个配套 SoC，包含处理器内核）。

make verilog_core CORE=n201
```

```
        // 查看仅仅 Core 的源代码，该命令会自动加载 run 目录下的 core_flist
        // 文件列表以查看 Core 的源代码。


// 步骤四：编译 RTL 代码，使用如下命令：

        // 如果想编译仅仅 Core 的 RTL 源代码，则使用如下命令：
make compile_core CORE=n201
```

2.10. 运行 Verilog 仿真测试

2.10.1. 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。

```
n200_rls_pkg
    |----riscv-tools
        |----riscv-tests
            |----isa      // 存放一些测试用例的目录
```

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 2-3 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, 0x0000000000000001 );
  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 2-3 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。

2.10.2. 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 n200_rls_pkg 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。

```
n200_rls_pkg
    |----riscv-tools
        |----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 2-4 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 2-4 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 2-5 所示。

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

图 2-5 Verilog 的 readmemh 函数可读入文件内容片段

2.10.3. 测试平台简介

在 n200_rls_pkg 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。

```
n200_rls_pkg
    |----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 2-6 所示。

注意：用户在将 N200 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 2-6 Testbench 中打印测试用例的结果

2.10.4. 运行测试用例

假设用户想使用 N200 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：下列步骤以 N201 为例，因此命令行中使用 CORE=n201。如果是其他型号的 Core（譬如 N205），则仅需更改 CORE 的参数，譬如 CORE=n205。

// 步骤一：将 n200_rls_pkg 解压至本机 Linux 环境中。

// 步骤二：生成并编译 RTL 代码，使用如下命令：

```
cd n200_rls_pkg/vsim
        // 进入到 n200_rls_pkg 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 CORE=n201
        // 运行该命令会在 vsim 目录下生成一个 install 子文件夹，在其中放置所需的 testbench
```

```
        // 文件，且将 testbench 中的关键字设置为 n201。

make compile CORE=n201
        // 编译所有的 Verilog 代码


// 步骤三：运行默认的一个 testcase（测试用例），使用如下命令：
make run_test CORE=n201 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。

        // 如果想查看该 test 执行后的波形，使用如下命令
make wave CORE=n201 TESTNAME=rv32ui-p-add


// 步骤四：运行回归（regression）测试集，使用如下命令：
make regress_run CORE=n201
        // 注意：这使用 riscv-tests/isa/generated
        // 目录中 N201 Core 的 testcases，逐个的运行 testcase。


// 步骤五：查看回归测试结果：
make regress_collect CORE=n201
        // 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。

2.10.5. 运行 C/C++程序

假设用户想在仿真环境中运行 C/C++语言编写的程序，那么需要借助 n200-sdk 进行。有关 n200-sdk 的详细介绍请参见《Nuclei_N200 系列 SDK 使用说明》。

以示例程序（Demo_ECLIC）为例，可以使用如下步骤进行。注意：下列步骤以 N201 为例，

因此命令行中使用 CORE=n201。如果是其他型号的 Core（譬如 N205），则仅需更改 CORE 的参数，譬如 CORE=n205。

// 步骤一：进入 n200-sdk 目录。

// 步骤二：在 n200-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_eclic CORE=n201 DOWNLOAD=ilm SIMULATION=1
```

// 步骤三：进入 n200_rls_pkg 目录，将上述步骤生成的 software/demo_eclic 文件夹拷贝到 n200_rls_pkg/vsim 目录中，命令如下：

```
cp n200-sdk/software/demo_eclic  n200_rls_pkg/vsim -rf
```

// 步骤四：在 n200_rls_pkg 的 vsim 目录下对测试程序进行仿真。

```
cd n200_rls_pkg/vsim
make run_test TESTCASE=$PWD/demo_eclic/demo_eclic
    //运行拷贝过来的 demo_eclic 示例程序
```

2.11. 逻辑综合 Verilog 代码

如果需要对 N200 系列进行逻辑综合，在进行综合之前，需注意如下事项：

- 以 N201 内核为例，n201_core 层次下完全为数字逻辑。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 N201 内核为例，时钟门控单元的源代码位于源文件 n200_rls_pkg/rtl/n201/core/n201_clkgate.v 中。