Nuclei™ Instruction Co-unit Extension

Copyright Notice

Contact Information

Should you have any problems with the information contained herein, please contact Nuclei System Technology by email support@nucleisys.com, or visit “Nuclei User Center” website http://user.nucleisys.com for supports or online discussion.

Revision History

Rev.	Revision Date	Revised Content
1.0	2019/8/20	1. Initial Release.
1.1	2019/12/20	1. Change Multi-Cycle rsp error to raise an exception(6.2).

Table of Contents

List of Tables

List of Figures

1. NICE Introduction

Nuclei Core Series supports configurable NICE (Nuclei Instruction Co-unit Extension) to support extensive customization and specialization. NICE allows customers to create user-defined instructions, enabling the integrations of custom hardware co-units that improve domain-specific performance while reducing power consumption. For example, artificial intelligence, could take the task of parameter training and model matching as an extension of the RISC-V kernel, which can enhance the performance of the entire system.

Co-unit connected by the NICE interface protocol (Hereinafter referred to as NICE-Core) is an independent module outside the Master Nuclei Core, so the NICE-Core can be turned off for reducing power while it is idle.

Nuclei Instruction Co-unit Extension supports the following features:

- Can be turned on/off through *mstatus* register.
- Support one-cycle transfer and multi-cycle transfer.
- Support 64-bit operation for one-cycle transfer when architecture is RV32 (please refer to configuration option in section 7.2)
- Support blocking mode and non-blocking mode for multi-cycle transfer，and at most 4 outstanding transfers for non-blocking mode.
- Support memory access with ICB protocol.
- Response error and memory access error observable and controllable.

2. NICE Instruction Format

NICE is a standard extension which means it should not conflict with any other RISC-V standard extensions. NICE instructions are also part of RISC-V base instruction set.

Table 2-1 RISC-V base opcode map, inst[1:0]=11

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (>32*b*)
00	LOAD	LOAD-FP	*Custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*Custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	⩾80*b*

In Table 2-1, major opcodes marked as *custom-0*, *custom-1*, *custom-2*, and *custom-3* are standard extensions and are recommended for custom instruction-set extensions within the base 32-bit instruction format. Principally, customers can use these four custom instruction groups for NICE extensions. Figure 2-1 shows the detail of NICE instruction format.

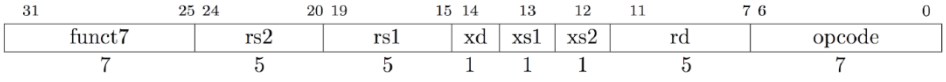

Figure 2-1 NICE instruction format

In Figure 2-1, once *xd* is set, the NICE instruction will write the result to destination register whose number is encoded in *rd* filed when the instruction retires, otherwise, no write-back operation will go on. If *xs1* or *xs2* is set, the NICE instruction will read the corresponding source register whose number is encoded in *rs1* or *rs2* field, otherwise, no source register will be used.

The highest 7 bits, *funct7*, can encode 128 instructions for each customer instruction group, so NICE support 4x128=512 extension instructions totally. Besides, NICE could get more instructions when *rs1*, *rs2* or *rd* filed is not used.

When the architecture is RV32 and it supports 64-bit operation for one-cycle transfer, setting bit 31th in NICE instruction means it is a 64-bit instruction with 64-bit integer of an even/odd pair

of registers specified by rs1(4,1), 64-bit integer of an even/odd pair of registers specified by rs2(4,1) and 64-bit integer of an even/odd pair of registers specified by Rd(4,1). *xd*, *xs1*, and *xs2* still control whether they are valid or not.

3. NICE Interface Descriptions

This chapter describes the NICE interface signals. It contains the following sections:

- *NICE global signals*
- *NICE request channel signals*
- *NICE one-cycle response channel signals*
- *NICE multi-cycle response channel signals*
- *NICE memory request channel signals*
- *NICE memory response channel signals*

3.1. NICE Global Signals

Name	Direction	Width	Description
nice_clk	Output	1	Clock for NICE-Core
nice_rst_n	Output	1	Reset for NICE-Core

3.2. NICE Request Channel Signals

Name	Direction	Width	Description
nice_req_valid	Output	1	This signal indicates Master Core sends a nice request. It should keep HIGH until nice_req_ready is set.
nice_req_ready	Input	1	This signal indicates NICE-Core can receive a nice request.
nice_req_instr	Output	32	The entire NICE instruction encoding from Master Core. It should keep stable until nice_req_ready is set.
nice_req_rs1	Output	32/64	The value of source register 1. It should keep stable until nice_req_ready is set.
nice_req_rs2	Output	32/64	The value of source register 2. It should keep stable until nice_req_ready is set.
nice_req_rs1_1	Output	32	The value of odd register in pair registers rs1(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when Nxxx_CFG_NICE_64BITS is configurated
nice_req_rs2_1	Output	32	The value of odd register in pair registers rs2(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when Nxxx_CFG_NICE_64BITS is configurated
nice_req_mmode	Output	1	This signal indicates the privilege mode of Master Core: 1: machine mode 0: user mode It should keep stable until nice_req_ready is set.

3.3. NICE One-Cycle Response Channel Signals

Name	Direction	Width	Description
nice_rsp_1cyc_type	Input	1	This signal indicates current NICE instruction is a one-cycle instruction and Master Core can get the result in one cycle.
nice_rsp_1cyc_dat	Input	32/64	The result from NICE-Core in one-cycle response
nice_rsp_1cyc_dat_1	Input	32	The value of odd register in pair registers rd(4,1), the high 32-bit in 64-bit data. Note: This signal exists only when Nxxx_CFG_NICE_64BITS is configurated
nice_rsp_1cyc_err	Input	1	This signal indicates current one-cycle response has an error and Master Core will take an illegal instruction exception when detecting this signal HIGH.

3.4. NICE Multi-Cycle Response Channel Signals

Name	Direction	Width	Description
nice_rsp_multicyc_valid	Input	1	This signal indicates NICE-Core sends a multi-cycle response. It should keep HIGH until nice_rsp_multicyc_ready is set.
nice_rsp_multicyc_ready	Output	1	This signal indicates Master Core can receive multi-cycle response.
nice_rsp_multicyc_dat	Input	32/64	The result from NICE-Core in multi-cycle response. It should keep stable until nice_rsp_multicyc_ready is set.
nice_rsp_multicyc_err	Input	1	This signal indicates current multi-cycle has an error and Master Core won't write the result to register file.

3.5. NICE Memory Request Channel Signals

Name	Direction	Width	Description
nice_icb_cmd_valid	Input	1	This signal indicates NICE-Core sends a memory access request to Master Core. It should keep HIGH until nice_icb_cmd_ready is set. Memory access request should be sent during a multi-cycle transfer.
nice_icb_cmd_ready	Output	1	This signal indicates Master Core can receive memory access request.
nice_icb_cmd_addr	Input	32/64	The address of memory access request. It should keep stable until nice_icb_cmd_ready is set.
nice_icb_cmd_read	Input	1	Write or Read of memory access requset: 0:Write 1:Read It should keep stable until nice_icb_cmd_ready is set.
nice_icb_cmd_wdata	Input	32/64	The write data of memory write request. It should keep stable until nice_icb_cmd_ready is set.

nice_icb_cmd_size	Input	2	The size of memory access request: 2'b00: byte 2'b01: half-word 2'b10: word 2'b11: reserved It should keep stable until nice_icb_cmd_ready is set. Note: NICE does not support misaligned access.
nice_icb_cmd_mmode	Input	1	The privilege mode of memory access requset. It should keep stable until nice_icb_cmd_ready is set.
nice_mem_holdup	Input	1	This signal helps NICE-Core occupy LSU pipe of Master Core for stalling next load and store instruction. This signal should be set one cycle after NICE-Core receives multi-cycle NICE instruction which includes memory operation and cleared after all memory accesses are done.

3.6. NICE Memory Response Channel Signals

Name	Direction	Width	Description
nice_icb_rsp_valid	Output	1	This signal indicates Master Core sends a memory access response to NICE-Core. It should keep HIGH until nice_icb_rsp_ready is set.
nice_icb_rsp_ready	Input	1	This signal indicates NICE-Core can receive memory access response.
nice_icb_rsp_rdata	Output	32/64	The read data of memory access. It should keep stable until nice_icb_rsp_ready is set.
nice_icb_rsp_err	Output	1	This signal indicates an error is detected during memory access of Master Core.

4. NICE Transfer

NICE instructions can be sent to NICE-Core only when mstatus.xs is NOT Zero, otherwise, an illegal instruction exception will be raised.

Before instruction sent to NICE-Core through the NICE interface, it is decoded by the Master Core and marked as a NICE instruction, at the same time rs1 and rs2 registers are read for the NICE interface if needed.

While NICE instruction has a dependency on previous unfinished instruction including common instruction or another NICE instruction, the pipeline would be stalled until the dependency is eliminated. With this mechanism, NICE instruction behaves just like a common instruction from the Master Core side.

NICE request channel confirms a transfer by *nice_req_valid* and *nice_req_ready* handshaking. *nice_req_valid* and other request information should keep stable until *nice_req_ready* signal is HIGH.

The NICE response might return through the one-cycle channel or multi-cycle channel, which depends on the implementation of the NICE-Core.

4.1. One-Cycle Response

If the NICE-Core can execute the instruction in one cycle, it sends the response to Master Core through one-cycle response channel, with or without data. In this way, *nice_rsp_1cyc_type* and *nice_rsp_1cyc_rdat* should keep for one cycle.

Figure 4-1 shows a one-cycle response with data.

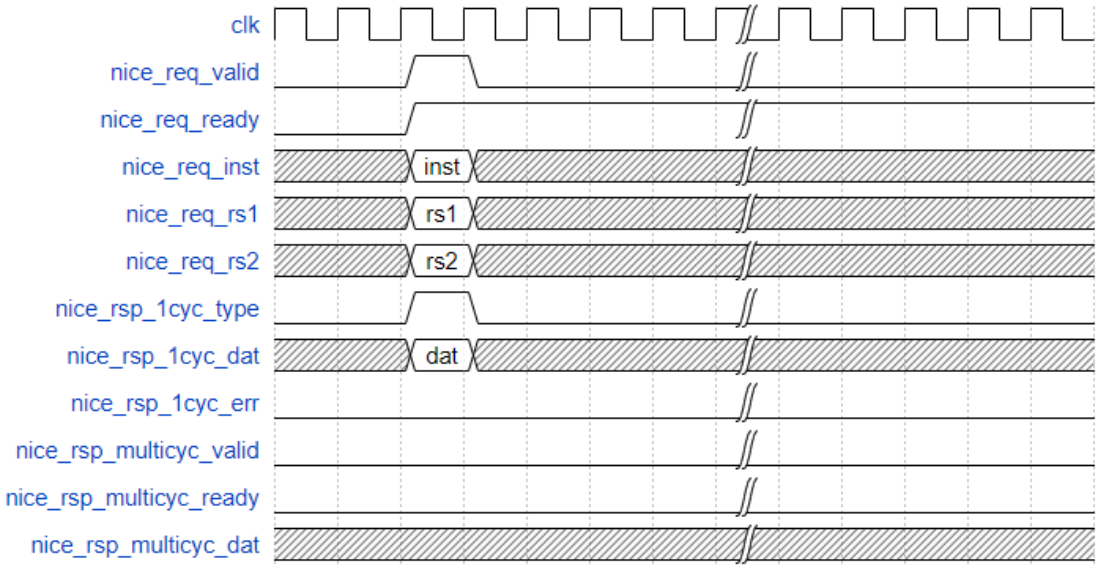

Figure 4-1 One-cycle response with data

4.2. Multi-Cycle Response

If NICE-Core needs more cycles to get the result(for example, huge computation or memory access. Please refer to chapter 5 for memory access operation), it sends the response to Master Core through the multi-cycle response channel.

There are two operation modes for NICE multi-cycle instruction: **blocking mode** and **non-blocking mode**. Blocking mode will clear *nice_req_ready* for stalling new NICE request until the current NICE transfer is done. Non-blocking mode, however, can receive a new NICE request no matter current NICE transaction is finished or not.

4.2.1. Multi-Cycle Blocking Mode

Figure 4-2 shows a multi-cycle blocking mode transfer with rs1, rs2, and rd valid. In this case, after the *nice_req_valid* and *nice_req_ready* handshake successfully, NICE-Core keeps *nice_req_ready* LOW until the data is ready to be written back.

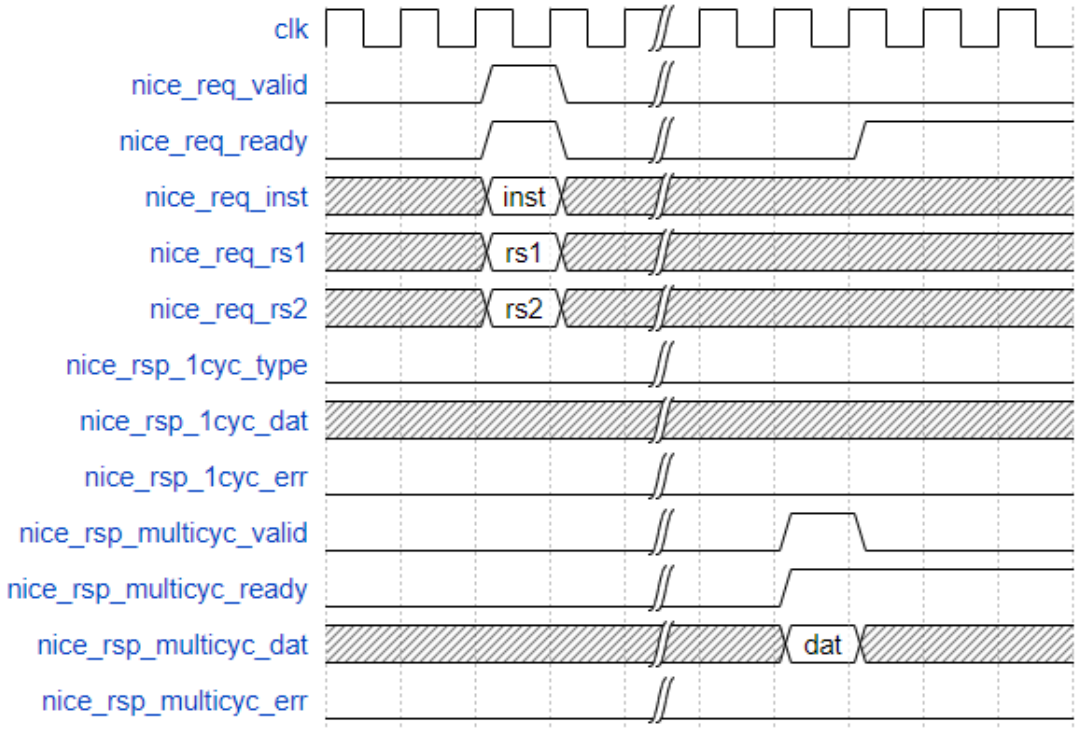

Figure 4-2 NICE multi-cycle blocking mode transfer

4.2.2. Multi-Cycle Non-Blocking Mode

Figure 4-3 shows four multi-cycle non-blocking mode transfers. In this case, NICE-Core can receive a new transfer while the previous transfer is still being processed. Each transfer needs four cycles to be done and sent the response, and because Master Core supports at most four outstanding transfer which is defined by RTL implementation, Master Core can send nice request contiguously without delay.

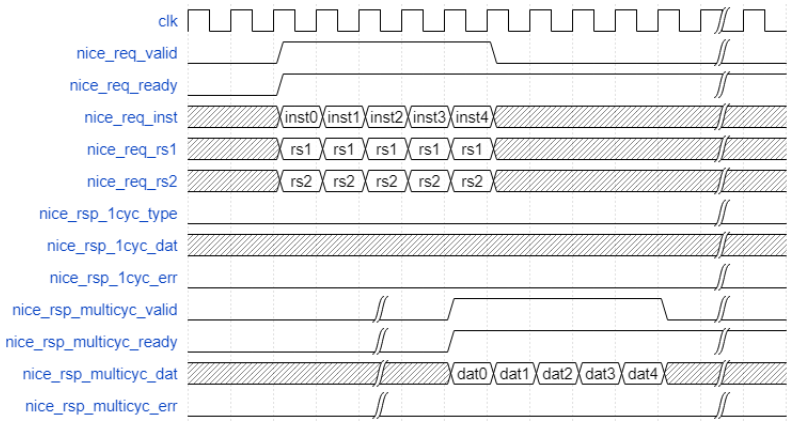

Figure 4-3 NICE multi-cycle blocking mode transfer without delay

Figure 4-4 shows four multi-cycle non-blocking transfer within eight cycles. In this case, NICE-Core can receive new transfer while the previous transfer is still being processed. Each transfer needs eight cycles to be done and sent response, but because Master Core supports at most four outstanding transfers, *nice_req_valid* must keep LOW while four NICE transfers are all being processed, until Master Core gets a response.

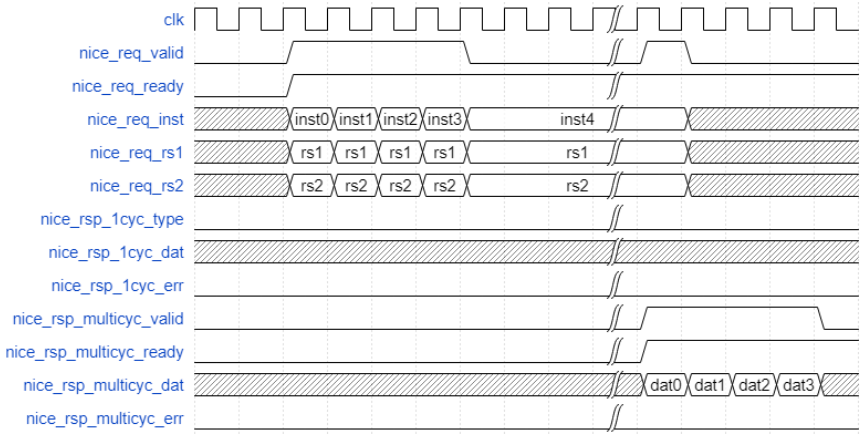

Figure 4-4 NICE multi-cycle blocking mode transfer with delay

5. NICE Memory Access

Generally, NICE-Core can access memory via the NICE interface, and the operation should be included in a multi-cycle transfer. While a multi-cycle transfer is going to access memory, the *nice_mem_holdup* should raise one cycle after *nice_req_valid* and *nice_req_ready* handshaking, and keep HIGH until NICE-Core finishes all nice memory accesses. This mechanism blocks the following load and store instruction, which can avoid some deadlock scenarios. With the help of *nice_mem_holdup*, NICE-Core can kick off one or several memory accesses at any time before the multi-cycle transfer is finished.

NICE accesses memory by ICB protocol. The ICB protocol contains command channel and response channel.

In the command channel, NICE-Core sends ICB request including *nice_icb_cmd_valid*, *nice_icb_cmd_addr*, *nice_icb_cmd_size* and *nice_icb_cmd_read*, then these signals are waiting for *nice_icb_cmd_ready* from Master Core. Once valid-ready handshakes successfully, Master Core processes the memory access operation with its LSU pipe.

In the response channel, Master Core sends *nice_icb_rsp_valid*, and *nice_icb_rsp_rdata* if it is a read operation, to NICE-Core and waits for *nice_icb_rsp_ready*.

nice_req_mmode indicates the current privilege mode in Master Core when a nice request is sent, and *nice_icb_cmd_mmode* should be the same mode with it when NICE-Core sends memory access request.

Note: NICE doesn't support misaligned memory access.

5.1. Single Memory Access Operation In Multi-Cycle Transfer

5.1.1. Single Memory Read Operation In Multi-Cycle Transfer

Figure 5-1 shows a single memory read operation in multi-cycle transfer.

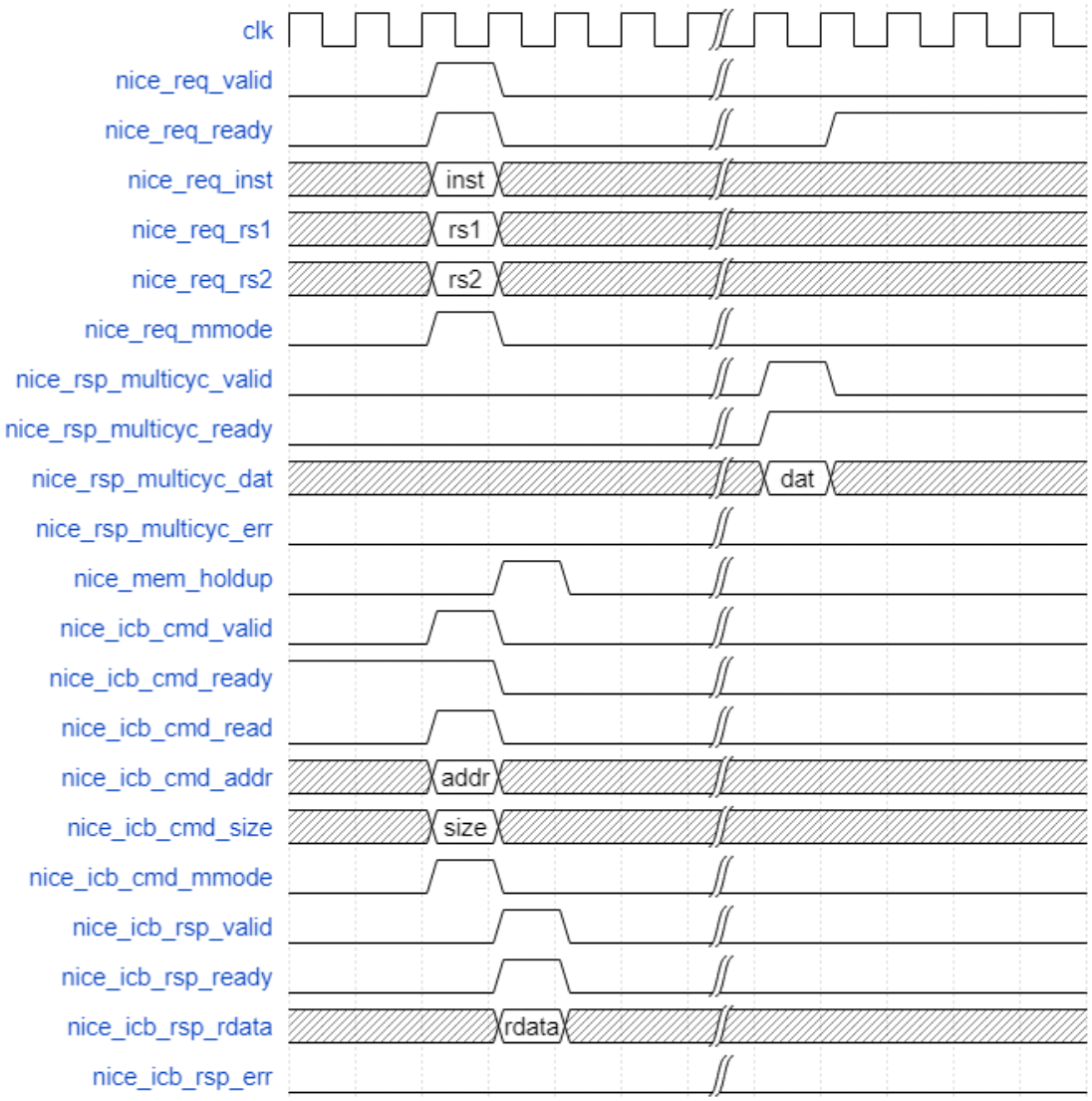

Figure 5-1 Single memory read operation in multi-cycle transfer

5.1.2. Single Memory Write Operation In Multi-Cycle Transfer

Figure 5-2 shows single memory write operation in multi-cycle transfer.

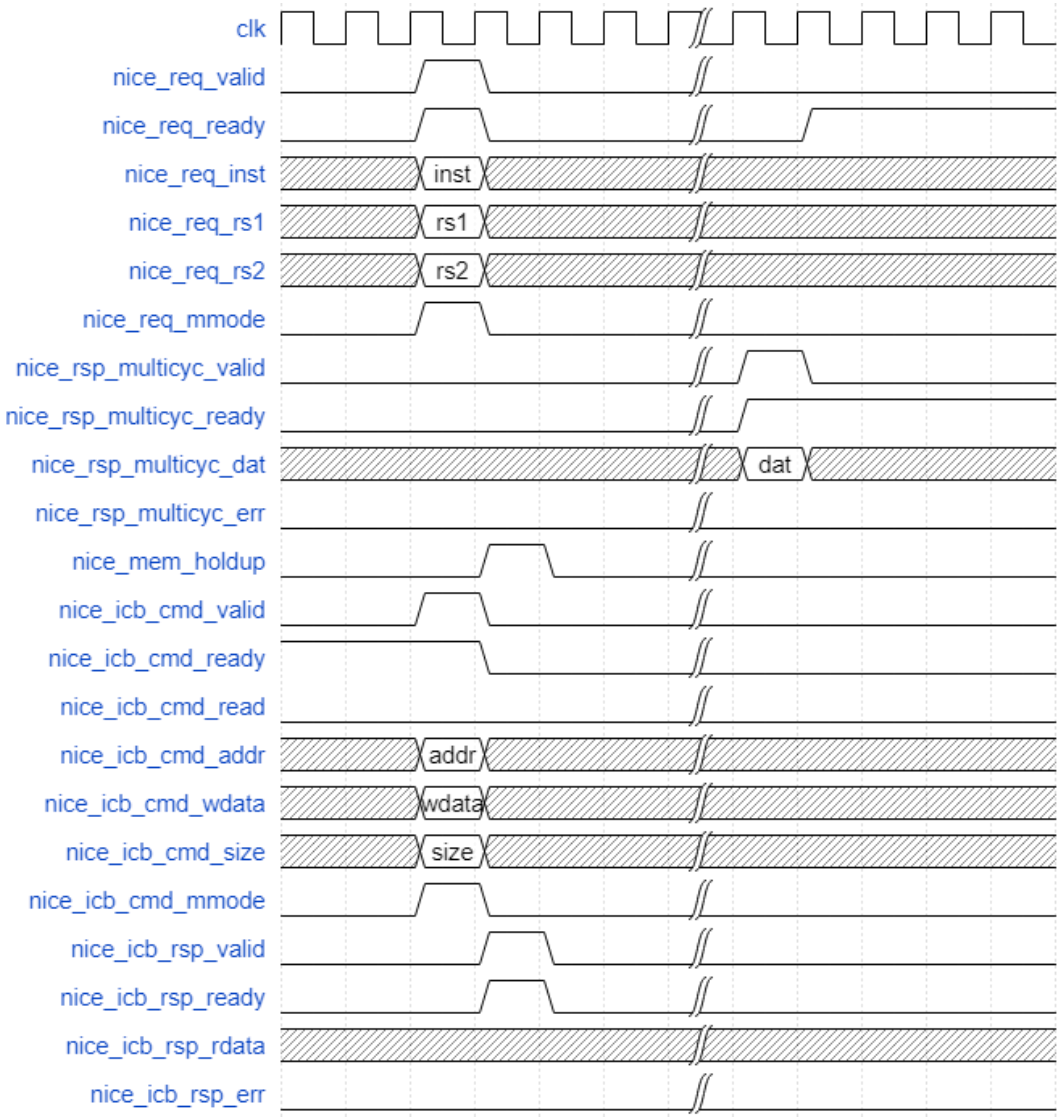

Figure 5-2 Single memory write operation in multi-cycle transfer

5.2. Multiple Memory Access Operation In Multi-Cycle Transfer

Figure 5-3 shows several memory accesses including read and write operations in a multi-cycle transfer. The Maximum num of ICB outstanding transfer depends on the implementation of NICE-Core.

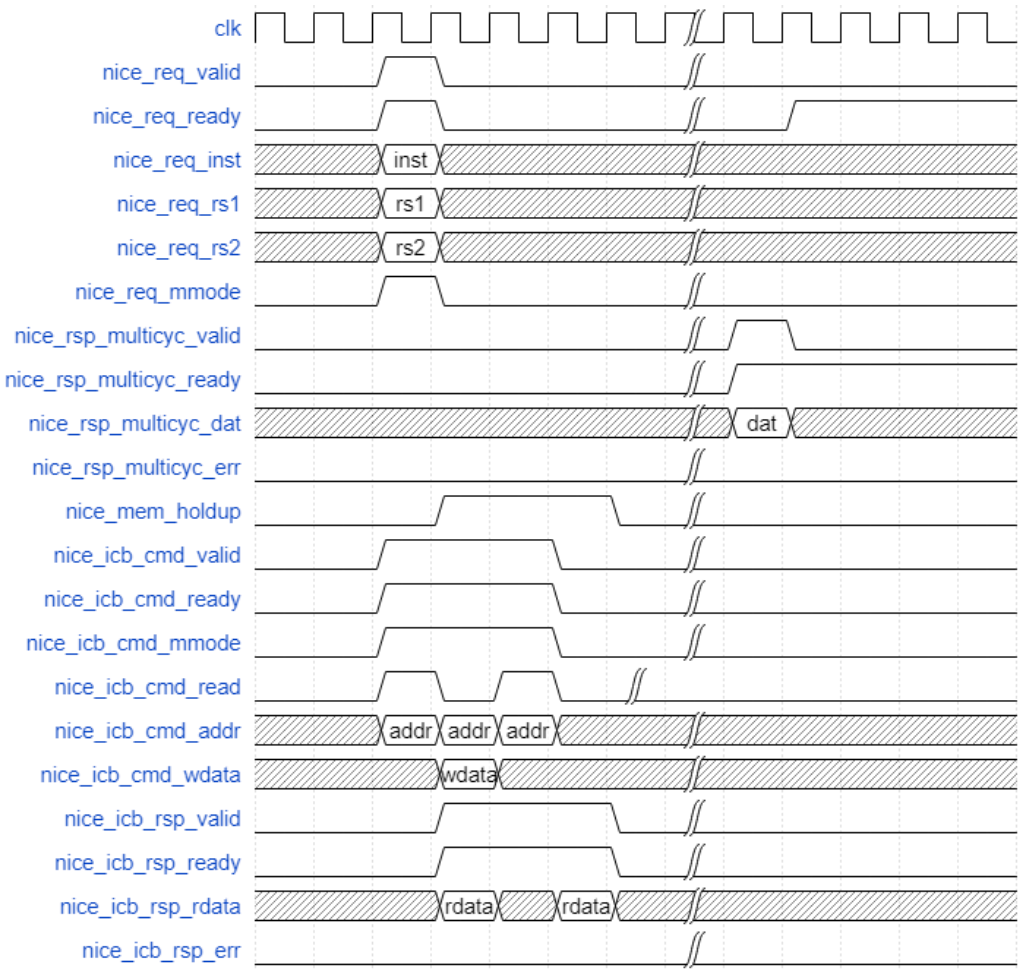

Figure 5-3 Several memory accesses including read and write operation

6. NICE Response Error

NICE-Core can send an error response to Master Core when it detects any error. There are two error types: one-cycle response error and multi-cycle response error.

6.1. One-Cycle Response Error

For one-cycle response error, it probably should be an illegal instruction found by NICE-Core. *nice_req_1cyc_err* signal will keep one cycle with *nice_req_1cyc_type*. When the Master Core receives the one-cycle response, an illegal instruction exception will be raised.

Figure 6-1 shows one-cycle response error.

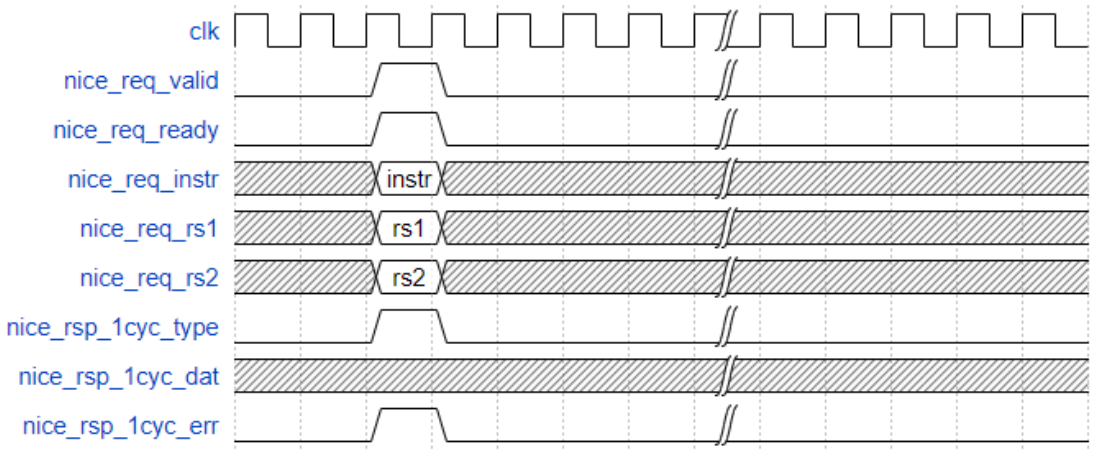

Figure 6-1 One-cycle response error

6.2. Multi-Cycle Response Error

For multi-cycle response error, it could be a memory access error or multi-cycle execution error. Figure 6-2 shows a memory access error scenario. Master Core sends *nice_icb_rsp_err* to NICE-Core for each corresponding memory access response, then NICE-Core sends one cycle *nice_rsp_multicyc_err* to Master Core at multi-cycle response phase.

While the Master Core receives *nice_rsp_multicyc_err*, the result won't be written back to register file. Additionally, a load access fault exception will be raised(Exception Code =5) and *mdcause* will be set to 3.

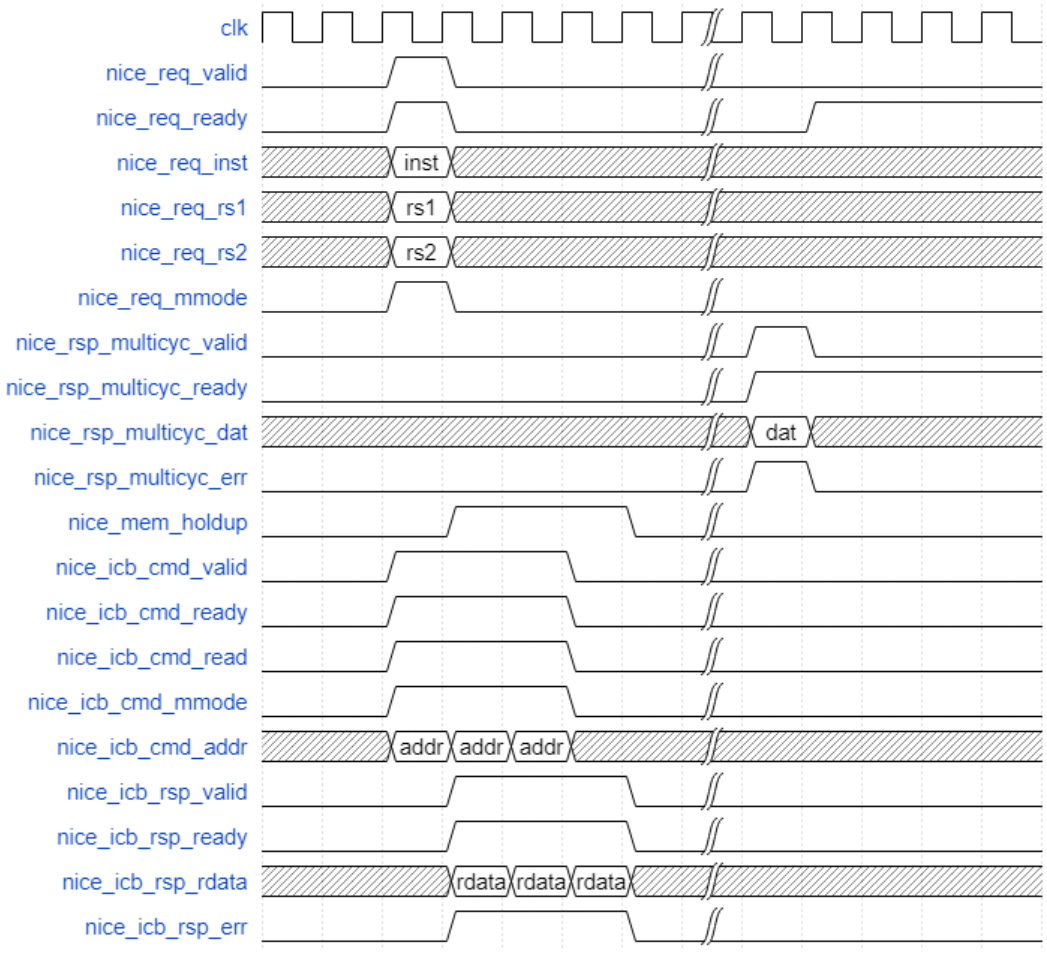

Figure 6-2 Multi-cycle response error

7. NICE Configuration

7.1. Nxxx_CFG_HAS_NICE

Nxxx_CFG_HAS_NICE is a global option to configure the implementation of NICE. It should be noted that for N200 Core Series N200_CFG_REGFILE_2WP must be defined when NICE is configured.

7.2. Nxxx_CFG_NICE_64BITS

Nxxx_CFG_NICE_64BITS is an RV32 option to support 64-bits source and destination for one-cycle transfer in. This option requires additional read and write ports in regfile, hence DSP option must be configured.

8. NICE Demo Introduction

This chapter shows a NICE demo to explain the usage of NICE. The NICE demo includes hardware and software parts. For hardware, the NICE-Core is included in Nuclei RTL package and the main function is to sum a 3x3 matrix for each row and column. For software, please refer to section 8.2.

The demo demonstrates a few instructions for mainly introducing the function of the NICE-Core and how to use these extended NICE instructions in software as well as the compiler.

8.1. Instruction In NICE Demo

This NICE Demo implements the following 3 instructions for NICE-Core.

Table 8-1 Instructions for NICE-Core

Instruction	Description	Encoding
CLW	Load 12-byte data from memory to row buffer.	■ opcode:0x7b, custom3 ■ xd:0, no write-back register ■ xs1:1, rs1 is vaild for load address ■ xs2:0, rs2 is invalid ■ funct7:1
CSW	Store 12-byte data from row buffer to memory.	■ opcode:0x7b, custom3 ■ xd:0, no write-back register ■ xs1:1, rs1 is valid for store address ■ xs2:0 ■ funct7:2
CACC	Sums a row of the matrix, and columns	■ opcode:0x7b, custom3

	are accumulated automatically.	■ xd:1, rd is valid for write-back register ■ xs1:1, rs1 is valid for the first address of a row ■ xs2:0, rs2 is invalid ■ funct7:6

In this NICE-Core, there is a 12-byte row buffer for saving the accumulated results of three columns. Before the operation of summing the matrix, the row buffer should be cleared with CLW instruction.

CACC instruction loads and accumulates all elements of a row one by one from memory and the result will be written back to register file directly. In addition, the columns are accumulated automatically for each CACC instruction and the results are saved in the row buffer in the NICE core. Figure 8-1 shows the behavior of CACC instruction

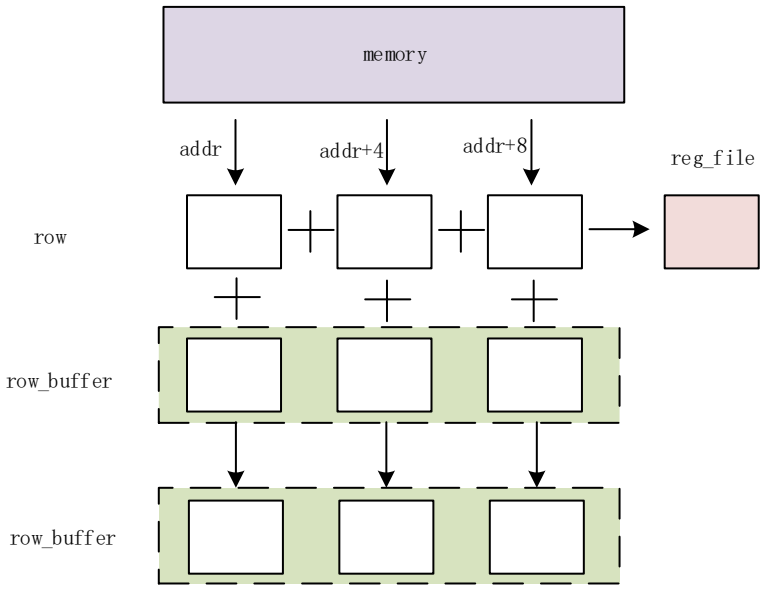

Figure 8-1 The behavior of CACC instruction

After all summing operations are done, the row buffer could be written back to memory by CSW instruction.

8.2. NICE Software Environment

8.2.1. SDK Environment

The software environment used in this demo follows the Nuclei public SDK environment(https://github.com/nucleisys/nuclei-n-sdk). Users could download the public software environment which contains some basic demo tests. After that, users can build the software environment based on the existing hello_world project, or create a new one according to the hello_world directory structure. This demo takes the first one, and names the project demo_nice.

8.2.2. Inline Assembly For User-defined Instruction

In this section, the CACC instruction will be an example to illustrate the usage of inline assembly. From section 8.1, we know that CACC is an R-type instruction and Figure 8-2 shows its format.

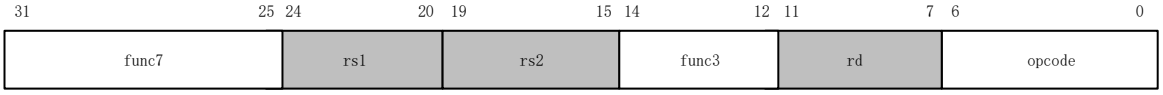

Figure 8-2 R-type instruction format

All user-defined instructions in the assembler are implemented by the pseudo instruction *.insn*. Following is the usage of pseudo instruction *.insn* for R-type.

```
.insn  r  opcode,  func3,  func7 ,  rd,  rs1,  rs2
```

For CACC instruction, the inline assembly function is as below.

```
inline int custom_rowsum(int addr)
{
 int rowsum;

 asm volatile (
    ".insn r 0x7b, 6, 6, %0, %1, x0"
```

```
    :"=r"(rowsum)
    :"r"(addr)
    );

  return rowsum;
}
```

custom _rowsum is the C function containing the CACC instruction. The function has an input and an output variable, respectively, corresponding to the starting address of a row of the matrix in the memory and the sum of the row elements.

The assembly part is contained in the asm volatile block. The *%0* indicates the output register whose value is reflected in the *rowsum* variable and register number is automatically assigned by the compiler. *%1* indicates the input register whose value is reflected in the *addr* variable and the register number is automatically assigned by the compiler.

'.insn' and *'r'* indicates this is a pseudo and R-type instruction. *'0x7b'* is the value of the opcode field, which means it is a NICE instruction belonging to custom3. The first *'6'* is the value of func3 field, which means the destination and source 1 register are valid. The second *'6'* is the value of func7 field, indicating it is a CACC instruction.

This inline assembly function will sum all elements in a row, and the result will be written back to register file directly. In addition, the columns are accumulated automatically for each *custom_rowsum* function and the results are saved in the buffer in the NICE core. The data of the buffer can be written back to memory with CSW instruction. Before the operation of summing the matrix, the buffer should be cleared with CLW instruction.

The following are the other two inline assembly functions.

CLW:

```
inline void custom_lbuf(int addr)
{
```

```
 int zero = 0;


 asm volatile (
    ".insn r 0x7b, 2, 1, x0, %1, x0"
:"=r"(zero)
:"r"(addr)
  );


}
```

CSW:

```
// custom sbuf
inline void custom_sbuf(int addr)
{
 int zero = 0;


 asm volatile (
    ".insn r 0x7b, 2, 2, x0, %1, x0"
:"=r"(zero)
:"r"(addr)
  );
```

8.2.3. Call Inline Assembly Function

The sum algorithm for row and column of the matrix is very simple. The conventional algorithm might be implemented by using two layers of for loops as shown below. *i* and *j* represent the number of rows and columns. *row_sum* and *col_sum* represent the sum of rows and columns.

```
// normal test case without NICE accelerator.
int normal_case(unsigned int array[ROW_LEN][COL_LEN])
{
  volatile unsigned char i=0, j=0;
```

```
  volatile unsigned int col_sum[COL_LEN]={0};
  volatile unsigned int row_sum[ROW_LEN]={0};
  volatile unsigned int tmp=0;
  for (i = 0; i < ROW_LEN; i++)
  {
    tmp = 0;
    for (j = 0; j < COL_LEN; j++)
    {
      col_sum[j] += array[i][j];
      tmp += array[i][j];
    }
    row_sum[i] = tmp;
  }
#ifdef _DEBUG_INFO_
  printf ("the element of array is :\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[0][i]); printf("\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[1][i]); printf("\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[2][i]); printf("\n\n");
  printf ("the sum of each row is :\n\t\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", row_sum[i]); printf("\n");
  printf ("the sum of each col is :\n\t\t");
  for (j = 0; j < COL_LEN; j++) printf("%d\t", col_sum[j]); printf("\n");
#endif
  return 0;
}
```

The NICE algorithm compresses some operations and reduces some unnecessary loading and writing back operations, hence it has a faster execution speed. The following code shows the NICE algorithm. *i* represents the number of rows. *row_sum* and *col_sum* represent the sum of rows and columns. All the above inline assembly functions can be called directly in the C function.

```
// teat case using NICE accelerator.
int nice_case(unsigned int array[ROW_LEN][COL_LEN])
{
  volatile unsigned char i, j;
  volatile unsigned int col_sum[COL_LEN]={0};
  volatile unsigned int row_sum[ROW_LEN]={0};

  custom_wsetup(COL_LEN-1);
  for (i = 0; i < ROW_LEN; i++)
  {
    row_sum[i] = custom_rowsum((int)array[i]);
  }
  custom_sbuf((int)col_sum);
#ifdef _DEBUG_INFO_
  printf ("the element of array is :\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[0][i]); printf("\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[1][i]); printf("\n\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", array[2][i]); printf("\n\n");
  printf ("the sum of each row is :\n\t\t");
  for (i = 0; i < ROW_LEN; i++) printf("%d\t", row_sum[i]); printf("\n");
  printf ("the sum of each col is :\n\t\t");
  for (j = 0; j < COL_LEN; j++) printf("%d\t", col_sum[j]); printf("\n");
#endif
  return 0;
}
```

In this demo, for comparing the performance, the conventional algorithm and the NICE algorithm are both tested.

8.2.4. Main Function And Makefile

The main function in demo_nice.c file mainly initializes the test environment, including:

- Matrix initialization
- Enable NICE-Core
- Enable performance evaluation function
- Call two sum functions, and report the result of performance comparison.

The makefile script in the project has some changes, including adding insn.c file, which contains all inline assembly function, and adding *-fgnu89-line* in CFLAGS to support inline assembly function.

There are only several files in the whole project: Makefile, demo_nice.c, insn.c and insn.h. Users can compile the project to get the executable file and make it run in the processor with the NICE-Core.

8.2.5. Result Analysis

Figure 8-3 shows the result of NICE demo test. The test contains both the conventional algorithm and the NICE optimization algorithm to calculate the 3x3 matrix. The optimization options are disabled and the debug information is enabled during the compilation process. As can be seen from the figure, the NICE optimization code functions correctly, and comparing to the conventional result, it has a significant reduction in the number of instructions and the operating cycle.

```
*********************************************
** begin to sum the array using ordinary add sum
the element of array is :
        10      20      30
        20      30      40
        30      40      50

the sum of each row is :
                60      90      120
the sum of each col is :
                60      90      120


*********************************************
** begin to sum the array using nice add sum
the element of array is :
        10      20      30
        20      30      40
        30      40      50

the sum of each row is :
                60      90      120
the sum of each col is :
                60      90      120
*********************************************
** performance list
         normal:
              instret: 21119, cycle: 29624
         nice  :
              instret: 20710, cycle: 29091
*********************************************
```

Figure 8-3 The result of NICE demo

Table 8-2 shows the performance at different optimization levels of the compiler toolchain. The O0+Debug indicates that the Debug information output is enabled and the compiler optimization option is disabled, and the remaining four items respectively correspond to the different optimization levels of the compiler tool with debug information disabled.

Table 8-2 Performance at different optimization level

00	O0+Debug		O0		O1		O2		O3	
01	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number	Instruction number	Cycle number
Convention	21119	29624	654	859	411	532	391	511	391	512
NICE	20710	29091	247	354	90	133	86	128	86	128

As can be seen from the table, the NICE core does improve the performance of the RISC-V core

in this application, and it is foreseeable that the larger the matrix, the better the performance.