ID1000500B
CONVOLUTION IP-CORE USER MANUAL

1. DESCRIPTION

The Convolution IP-core objective is giving the convolution of two signals. One of them will be stored inside the module, while the other is given by the device that's interacting with the IP-core.
After receiving the size of the output memory, it's data and the start command, this IP-core will convolute the output (Y) and input (X) memories and give the result in another output memory (Z).

1.1. CONFIGURABLE FEATURES

Software configurations	**Description**
Size of memory Y	Contains output memory Y number of registers.

1.2. TYPICAL APPLICATION

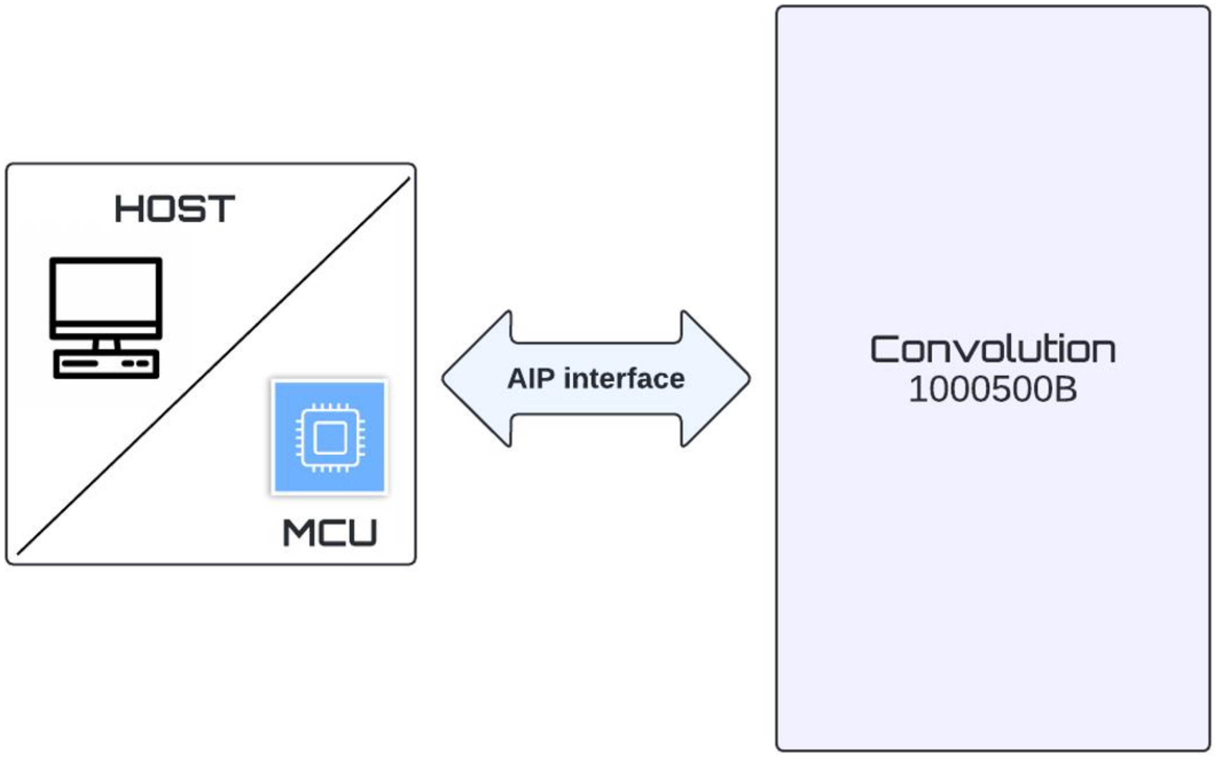


Figure 1.1 IP Convolution connected to a host

2. CONTENTS

2.1. List of figures

2.2. List of tables

3. INPUT/OUTPUT SIGNAL DESCRIPTION

Table 1 IP Convolution input/output signal description

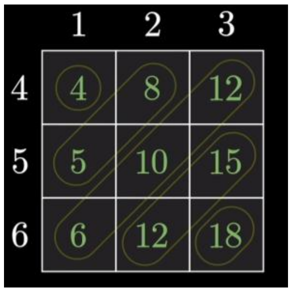
Signal	Bitwidth	Direction	Description
		General signals	
clk	1	Input	System clock
rst_a	1	Input	Asynchronous system reset, low active
en_s	1	Input	Enables the IP Core functionality
		AIP Interface	
data_in	32	Input	Input data for configuration and processing
data_out	32	Output	Output data for processing results and status
conf_dbus	5	Input	Selects the bus configuration to determine the information flow from/to the IP Core
write	1	Input	Write indication, data from the data_in bus will be written into the AIP Interface according to the conf_dbus value
read	1	Input	Read indication, data from the AIP Interface will be read according to the conf_dbus value. The data_out bus shows the new data read.
start	1	Input	Initializes the IP Core process
int_req	1	Output	Interruption request. It notifies certain events according to the configurated interruption bits.
		Core signals	

4. THEORY OF OPERATION

The Convolution core makes the convolution operation of two different signals after receiving a start processing command, the signals registers are stored in an internal memory (X) and the other in an external one (Y). This operation is performed with an algorithm consisting of obtaining the sum of a matrix diagonals products. Say, we have two signals.

$$x = (1,2,3) \text{ and } y = (4,5,6)$$

Well, one solution can be reached by putting the registers in the exterior of a matrix and put the result of multiplying one another, and after that, doing a summatory of the diagonals of the matrix.



So, the result is

$$z = (4,\ 5 + 8,\ 6 + 10 + 12,\ 12 + 15,\ 18) = (4,\ 13,\ 28,27,18)$$

Once the IP module finishes the operation, it will notify it by sending a ‘done’ signal.

5. AIP interface registers and memories description

5.1. Status register

Config: STATUS
Size: 32 bits
Mode: Read/Write.

This register is divided in 3 sections, see Figure 5.1:

- **Status Bits**: These bits indicate the current state of the core.
- **Interruption Flags:** These bits are used to generate an interruption request in the *int_req* signal of the AIP interface.
- **Mask Bits**: Each one of these bits can enable or disable the interruption flags.

Status Register

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17	16	15 14 13 12 11 10 9	8	7 6 5 4 3 2 1	0
	Mask Bits		***Status Bits***		***Interrupt/Clear Flags***	
Reserved	Reserved	MSK	Reserved	BSY	Reserved	DN
		rw		r		rw

Figure 5.1 IP Dummy status register

Bits 31:24 – Reserved, must be kept cleared.

Bits 23:17 – Reserved Mask Bits for future use and must be kept cleared.

Bit 16 – **MSK:** mask bit for the DN (Done) interruption flag. If it is required to enable the DN interruption flag, this bit must be written to 1.

Bits 15:9 – Reserved Status Bits for future use and are read as 0.

Bit 8 – **BSY**: status bit "**Busy**".
Reading this bit indicates the current IP Dummy state:
0: The IP Dummy is not busy and ready to start a new process.
1: The IP Dummy is busy, and it is not available for a new process.

Bits 7:1 – Reserved Interrupt/clear flags for future use and must be kept cleared.

Bit 0 – **DN**: interrupt/clear flag "**Done**"
Reading this bit indicates if the IP Dummy has generated an interruption:
0: interruption not generated.
1: the IP Dummy has successfully finished its processing.
Writing this bit to 1 will clear the interruption flag DN.

5.2. Configuration memory Y's size register

Config: CONFREG

Size: 32 bits
Mode: Write

This register is used to configure the external input memory Y's number of registers. This register must be configured before the core starts convoluting. See Figure 6.2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RESERVED																											SIZE_Y [4:0]				
																											w	w	w	w	w

Figure 5.2 Configuration memory Y's size register.

Bits 31:5 – Reserved, must be kept cleared.

Bit 4:0 – **SIZE_Y**: describes the number of registers that appear in external input memory Y.

5.3. Input data memory Y

Config: MEM_Y
Size: 32 bits
Mode: Write

This memory is used to store registers of a signal that is about to be convoluted with another one. The size of this memory manages 32-bit words.

5.4. Output data memory Z

Config: MEM_Z
Size: 32 bits
Mode: Read

This memory is used to store the processed data by the IP Convolution. After the IP Module completes its processing, the data stored in this memory will be the result of the convolution of the input data memory Y and the internal memory X. The size of this memory manages 32-bit words.

6. PYTHON DRIVER

The file *ID1000500B.py* contains the convolution class definition. This class is used to control the IP Convolution core for python applications.

6.1. Usage example

In the following code a basic test of the IP Convolution core is presented. First, it is required to create an instance of the convolution object class. The constructor of this class requires the network address and port where the IP is connected, the communication port, and the path where the configs csv file is located.

Thus, the communication with the IP Convolution will be ready. In this code, the registers of the input memory Y are randomly generated and later passed through a conv function, which returns the results of the convolution, stored in memZ.

```
if __name__=="__main__":
    import sys, random, time, os
    logging.basicConfig(level=logging.DEBUG)

    aip_mem_size = 8
    size_mem_y   = 5
    connector = '/dev/ttyACM0'
    nic_addr = 1
    port = 0
    csv_file = '/home/pablo/convo_core/ID1000500B_config.csv'
    convolutioner = 0

    try:
        convolutioner = convolution(connector, nic_addr, port, csv_file)
        ##convolutioner.reset()
    except:
        e = sys.exc_info()
        print('ERROR: ', e)

    #===========================================
    random.seed(64)
    memY = [random.randrange(2 ** 8) for i in range(0, size_mem_y)]
    memZ = convolutioner.conv(memY)
    convolutioner.finish()
    logging.info("DONE!")
```

6.2. Methods

6.2.1. Constructor

```
def __init__(self, connector, nic_addr, port, csv_file):
     ## Pyaip object
     self.__pyaip = pyaip_init(connector, nic_addr, port, csv_file)
     if self.__pyaip is None:
         logging.debug(error)
     ## Array of strings with information read
     self.__pyaip.reset()
     self.dataRX = []
     ## IP Convolution IP-ID
     self.IPID = 0
     self.__getID()
     self.__clearStatus()
     logging.debug(f"IP Convolution controller created with IP ID
     {self.IPID:08x}")
```

Creates an object to control the IP Convolution in the specified network address.

Parameters:

• connector (string):	Communications port used by the host.
• nic_addr (int):	Network address where the core is connected.
• port (int):	Port where the core is connected.
• csv_file (string):	IP Convolution csv file location.

6.2.2. conv

```
def conv(self, memY):
  # Enable interruptions
  self.__enableINT()
  # Write Conf Reg with size of memory Y
  self.__writeSizeY(len(memY))
  # Write memory: MEM_Y
  self.__writeData(memY)
  # Start IP
  self.__startIP()
  # Show status
  self.__status()
  # Clear Interrups
  #self.__disableINT()
  # Wait for Done Flag
  self.__waitInt()
  print('end waiting')
  # Read memory: MEM_Z
  memZ = self.__readData(64)
  # Clear flags
  self.__clearStatus()
  # Show status
  self.__status()
  return memZ
```

Does the convolution operation. The interruptions are enabled with `enableINT`, the size of memY is stored in the configuration register with `writeSizeY`, the input memory Y is written with `writeData`. After that `startIP` sends the start signal and `waitInt` lets the program continue once the 'done' flag is raised. The results obtained by `readData` are stored in memZ, finally, `clearStatus` clears the flag.

6.2.3. writeData

```
def __writeData(self, data):
```

Write data in the IP core input memory.

Parameters:

• data (List[int]):	Data to be written.

6.2.4. readData

```
def __readData(self, size):
```

Read data from the IP core output memory.

Parameters:

• size (int):	Size of the register in which all data will be saves.

Returns:

• List[int]	Data read from the output memory.

6.2.5. startIP

```
def __startIP(self):
```

Start processing in IP Convolution.

6.2.6. writeSizeY

```
def __writeSizeY(self, sizeY):
```

Set memory Y's number of registers.

Parameters:

• sizeY (int):	Number of registers of memory Y.

6.2.7. enableINT

```
def enableINT(self):
```

Enable IP Convolution interruptions (bit DONE of the STATUS register).

6.2.8. disableINT

```
def disableINT(self):
```

Disable IP Convolution interruptions (bit DONE of the STATUS register).

6.2.9. status

```
def status(self):
```

Show IP Convolution status.

6.2.10. clearStatus

```
def __clearStatus(self):
```

Clears status.

6.2.11. waitINT

```
def waitINT(self):
```

Wait for the completion of the process.

6.2.12. getID

```
def __getID(self):
```

Obtains ID of core

6.2.13. finish

```
def finish(self):
```

Finishes connection pyaip.

7. C DRIVER

In order to use the C driver, it is required to use the files: *id1000500b.h, id1000500b.c, id1000500b_conv_core.h and id1000500b_conv_core.c,* that contain the driver functions definition and implementation. The functions defined in this library are used to control the IP Dummy core for C applications.

7.1. Usage example

In the following code a basic test of the IP Convolution core is presented.

```
CODE SAMPLE:


// Example code of the convolution driver for the ID1000500B core
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include "id1000500b_conv_core.c"
/*************************************************************************/

/*************************************************************************/
/* DEFINITION OF PARAMETERS */

#define CONFIG_PATH "/home/pablo/convo_core/ID1000500B_config.csv"
#define USB_PORT    "/dev/ttyACM0"
#define SIZE_X      5
/*************************************************************************/

int main(void)
{

/*************************************************************************/
    // config init
    uint8_t nic_addr  = 1;
    uint8_t port = 0;
    // convolution process
    uint8_t sizeX = 5;
    uint8_t dataX[SIZE_X] = {0x00, 0x01, 0x02, 0x03, 0x04};
    uint16_t result[AMOUNT_RESULT_MAX] = {0};
/*****************************************************************************/

    // initialize the core
    id1000500b_init(USB_PORT, nic_addr, port, CONFIG_PATH);

    printf("\nData in mem_Y to be sent");
    printf("\nTX Data\n");
    for(uint32_t i=0; i<SIZE_X; i++){
        printf("%08X\n", dataX[i]);
    }

    // print the result before
    printf("\nresult Data before processing: [");
    for(int i=0; i<AMOUNT_RESULT_MAX; i++){
        printf("0x%08X", result[i]);
        if(i != AMOUNT_RESULT_MAX-1){
            printf("\n, ");
        }
    }
    printf("]\n\n");

    // call the task to be executed
    conv(dataX, sizeX, result);

    // print the result
    // printf("\nresult Data: [");
    // for(int i=0; i<AMOUNT_RESULT_MAX; i++){
    //     printf("0x%08X", result[i]);
    //     if(i != AMOUNT_RESULT_MAX-1){
    //         printf("\n, ");
    //     }
    // }
    // printf("]\n\n");

    printf("\n\nPress key to close ... ");
    return SUCCESS;

}
```

7.2. Driver functions

7.2.1. id1000500b_init

```
int32_t id1000500b_init(const char *connector, uint_8 nic_addr, uint_8 port,
const char *csv_file)
```

Configure and initialize the connection to control the IP Dummy in the specified network address.

Parameters:

• connector:	Communications port used by the host.
• nic_addr:	Network address where the core is connected.
• port:	Port where the core is connected.

• csv_file:	IP Dummy csv file location.

Returns:

• int32_t	Return 0 whether the function has been completed successfully.

7.2.2. id1000500b_writeData

```
int32_t id1000500b_writeData(uint32_t *data, uint32_t data_size, uint_8
nic_addr, uint_8 port)
```

Write data in the IP Convolution core input memory.

Parameters:

• data:	Pointer to the first element to be written.
• data_size:	Number of elements to be written.
• nic_addr:	Network address where the core is connected.
• port:	Port where the core is connected.

Returns:

• int32_t	Return 0 whether the function has been completed successfully.

7.2.3. id1000500b_readData

```
int32_t id1000500b_readData(uint32_t *data, uint32_t data_size, uint_8
nic_addr, uint_8 port)
```

Read data from the IP Convolution core output memory.

Parameters:

• data:	Pointer to the first element where the read data will be stored.
• data_size:	Number of elements to be read.
• nic_addr:	Network address where the core is connected.
• port:	Port where the core is connected.

Returns:

• int32_t	Return 0 whether the function has been completed successfully.

7.2.4. id1000500b_startIP

```
int32_t id1000500b_startIP(uint_8 nic_addr, uint_8 port)
```

Start processing in IP Convolution core.

Parameters:

• nic_addr:	Network address where the core is connected.
• port:	Port where the core is connected.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.5. id1000500b_enableINT

```
int32_t id1000500b_enableINT(int_8 nic_addr, uint_8 port)
```

Enable IP Convolution core interruptions (bit DONE of the STATUS register).

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.6. id1000500b_disableINT

```
int32_t id1000500b_disableINT(int_8 nic_addr, uint_8 port)
```

Disable IP Convolution core interruptions (bit DONE of the STATUS register).

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.7. id1000500b_status

```
int32_t id1000500b_status(int_8 nic_addr, uint_8 port)
```

Show IP Convolution core status.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.8. id1000500b_waitINT

```
int32_t id1000500b_waitINT(void)
```

Wait for the completion of the process.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.9. id1000500b_finish

```
int32_t id1000500b_finish(void)
```

Finishes all processes in the core.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.10. Id1000500b_getID

```
int32_t id1000500b_getID(uint32_t *id)
```

Ask for the ID to the IP Core.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.11. Id1000500b_clearStatus

```
int32_t id1000500b_clearStatus(void)
```

Clears the status register.

Returns:

- int32_t Return 0 whether the function has been completed successfully.

7.2.12. conv

```
int32_t conv(uint8_t *X, uint8_t sizeX, uint16_t result)
```

Operates the convolution with the parameters.

Returns:

- int32_t Return 0 whether the function has been completed successfully or –1 if some fault was committed.

7.2.13. check_if_valid_size

```
int32_t check_if_valid_size(uint8_t sizeX)
```

Verifies if the length of the dataX array (input array).

Returns:

- int32_t Return 0 whether the function has been completed successfully or –1 if some fault was committed.

7.2.14. set_parameters

```
void set_parameters(int32_t *X, uint8_t *sizeX,)
```

Writes the array input and uses it length as sizeX.

Returns:

- nothing.

7.2.15. run_convolution

```
void run_convolution(void)
```

Executes the necessary tasks to run the convolution

Returns:

- nothing.

7.2.16. read_result

```
void read_result(uint32_t *result)
```

Writes the result of the processed data in the array result.

Returns:

- nothing.

7.2.17. stop_core

```
void stop_core(void)
```

Kills the processes to reuse the core any time later.

Returns:

- nothing.