ID1000500A
CONVOLUTION IP-CORE USER MANUAL

1. DESCRIPTION

The Convolution IP-core is the system representing an algorithm that combines two functions to describe the overlap between the two functions.

After receiving a start command, this Ip-core performs the convolution algorithm of the data stored in the input memories X and Y and saves the resulting data in the output memory Z.

1.1. CONFIGURABLE FEATURES

Software configurations	**Description**
Processing size	Size of input memories.
Input Memory	Initial address of the input memory. Sets the base address from where data will be read.
Output Memory	Initial address of the output memory. Sets the base address to where data will be written.
Transfer size	Size of data blocks to transfer in each operation. Allows handling different block sizes, adjusting the amount of data transferred per cycle.
Operation Mode	Operation mode, such as single or continuous transfer Selects the mode to perform a single data transfer or continuous transfers until stopped.

1.2. TYPICAL APPLICATION

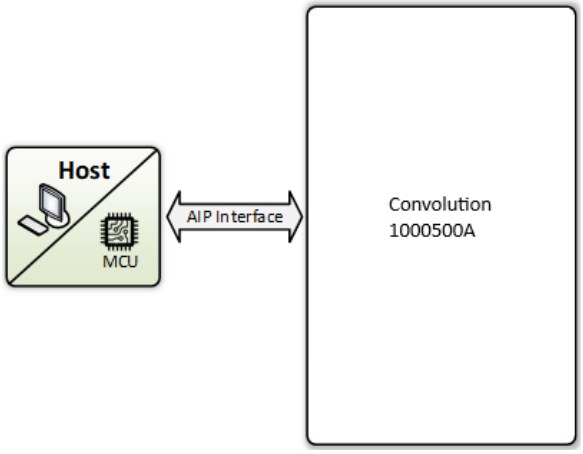

Figure 1.1 IP Convolution connected to a host

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

Introduction to convolution

Convolution is a mathematical operation that combines two functions to describe the overlap between them. Convolution takes two functions, "slides" one over the other, multiplies the values of the functions at all points of overlap, and adds the products to create a new function. This process creates a new function that represents how the two original functions interact with each other.

Formally, the convolution is an integral that expresses the amount of overlap of a function, f(t) when it is shifted over another function, g(t)

$$(f*g)(t)\approx_{\text{def}}\int_{\infty}^{-\infty} f(\tau)g(t-\tau)dr$$

Depending on the application of convolution, functions can be replaced with signals, images, or other types of data. Convolution and its applications can be implemented in several ways.

Input Matrix:

This is the matrix on which the convolution will be performed. In image processing, this matrix represents the image in the form of pixels.

Kernel (Filter):

A kernel is a small matrix that slides over the input matrix. Each kernel position produces a value in the output matrix based on a dot product operation between the elements of the kernel and the elements of the input matrix it covers.

Output Matrix:

The resulting matrix after applying the convolution operation. Each element of the output matrix is the result of the dot product of the kernel with a submatrix of the input matrix.

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 of 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 Convolution 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 delay register

Config: Csize
Size: 32 bits
Mode: Write

This register is used to configure the X and Y memory sizes before the core starts convolving them. See Figure 5.2

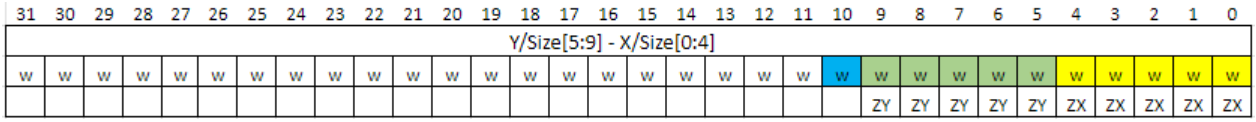

Figure 5.2 Configuration size register.

Bits 9:5 – **SizeY:** size value. The size value can be any value in the range 0x00001-0xFFFF.

Bit 4:0 – **SizeX:** size value. The size value can be any value in the range 0x00001-0xFFFF.

5.3. Input data memory

Config: MdataX
Size: 32 bits
Mode: Write

This memory is used to store data to be processed by the IP Convolution core.

Config: MdataY
Size: 32 bits
Mode: Write

This memory is used to store data to be processed by the IP Convolution core.

5.4. Output data memory

Config: MdataZ
Size: 64 bits
Mode: Read

This memory is used to store processed data by the IP Convolution. After the IP Convolution completes its processing, the data stored in this memory will be the same as the input data memory.

6. PYTHON DRIVER

The file *ID1000500A.py* contains the convolution class definition. This class is used to control the IP Convolution core for python applications.
This Python code aims to interact with a hardware device that uses the pyaip protocol to read and write data, start processes, and check their status. Let's see how it accomplishes this in an organized fashion:
The code imports the libraries needed to communicate with the device via the pyaip protocol.

The connection parameters are set, including the port, the network address of the device and the location of a configuration file (possibly device-specific). Initialize communication with the device using the pyaip_init function. The device is reset using the aip.reset() function.

The device identifier (ID) is read and displayed. The device status (STATUS) is read and displayed.
An example data set (MemX and MemY) is defined. The data is written to the device memory at locations MdataX and MdataY using the aip.writeMem function. The code also displays the written contents.

A sample configuration size (Size) is defined.
The size is written to the device configuration register Csize using the aip.writeConfReg function. The code also displays the written content.
The process is started on the device with the aip.start() function.
The device status (STATUS) is read back and displayed on the screen.
10 bytes of data are read from the device memory at location MdataZ using the aip.readMem function.
The code also displays the content read.

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 Convolution 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 inputs memories are written with array X and array Y by using the writeData method. Then the startIP method is used to start core processing. Finally, the waitINT method is used to wait the activation of the DONE flag, and after that, the output data is read with the readData method.

```
import sys, random, time, os
from ID1000500A import convolution
from ipdi.ip.pyaip import pyaip, pyaip_init, Callback

logging.basicConfig(level=logging.INFO)
connector = '/dev/ttyACM0'
csv_file =
'/home/a/Documents/HDL/ID1000500A_AlejandroPardo_LeonelGallo/config/ID1000500A_config.csv'
addr = 1
port = 0
aip_mem_size = 32

X = [0x00000001, 0x00000002, 0x00000003, 0x00000004, 0x00000003, 0x00000007, 0x00000006,
0x0000000A, 0x00000005, 0x00000008]
```

```
Y = [0x00000003, 0x00000003, 0x00000005, 0x00000006, 0x00000007]

try:
    driver = convolution(connector, addr, port, csv_file)
     logging.info("Test Convolution: Driver created")

     driver.status()
except:
    logging.error("Test Dummy: Driver not created")
    sys.exit()

driver.disableINT()

driver.writeData('MdataX', X)
logging.info(f'TX MemX Data: {[f"{x:08X}" for x in X]}')

driver.writeData('MdataY', Y)
logging.info(f'TX MemY Data: {[f"{x:08X}" for x in Y]}')

driver.startIP()
driver.waitInt()

conv_result = driver.conv(X,Y)
logging.info(f'Convolution result: {[f"{x:08X}" for x in conv_result]}')

driver.status()
driver.finish()

logging.info("The End")
```

6.2. Methods

6.2.1. Constructor

```
def __init__(self, connector, nic_addr, port, csv_file):
```

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 Dummy csv file location.

6.2.2. writeData

```
def writeData (self, mem_name, data):
```

Write data in the IP Convolution input memory.

Parameters:

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

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.3. readData

```
def readData (self, mem_name, size) :
```

Read data from the IP Convolution output memory.

Parameters:

- size (int): Communications port used by the host.

Returns:

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

6.2.4. startIP

```
def startIP(self):
```

Start processing in IP Convolution.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.5. enableINT

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

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

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.6. disableINT

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

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

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.7. status

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

Show IP Convolution status.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.8. waitINT

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

Wait for the completion of the process.

Returns:

- bool An indication of whether the operation has been completed successfully.

6.2.9. conv

```
def conv(self, X, Y):
```

X and Y convolution processing.

Parameters:

- X (List[int]): X data for convolution.
- Y (List[int]): Y data for convolution.

Returns:

- Z(List[int]) result of the convolution.

7. C DRIVER

In order to use the C driver, it is required to use the files: *ID1000500A_driver.h, ID1000500A_driver.c* that contain the driver functions definition and implementation. The functions defined in this library are used to control the IP convolution core for C applications.

7.1. C driver

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

```
#include <stdio.h>
#include <stdlib.h>
//#include <conio.h> // getch
#include "ID1000500A_driver.h"

#define SIZEX 10
#define SIZEY 5
#define SIZECONF 1
#define AUX SIZEX+SIZEY-1

int main()
{
    const char *connector = "/dev/ttyACM0";
    uint8_t nic_addr = 1;
    uint8_t port = 0;
    const char *csv_file =
"/home/a/Documents/HDL/ID1000500A_AlejandroPardo_LeonelGallo/config/ID1000500A_config.csv";

    if (id1000500A_init(connector, nic_addr, port, csv_file) < NULL)
        {
        printf("Failed to initialize the IP Controller.\n");
        return -1;
```

```
    }

        // Show initial ID and status
    id1000500A_status();

        // Data to write to MemX
        uint8_t  MemX[SIZEX]  =  {0x00000001,  0x00000002,  0x00000003,  0x00000004,  0x00000003,
0x00000007, 0x00000006, 0x0000000A, 0x00000005, 0x00000008};
        printf("\nWrite memory: MdataX\n");
    id1000500A_writeData(MemX, SIZEX);

        // Data to write to MemY
    uint8_t MemY[SIZEY] = {0x00000003, 0x00000003, 0x00000005, 0x00000006, 0x00000007};
    printf("Write memory: MdataY\n");
    id1000500A_writeData(MemY, SIZEY);

        // Configuration register for size
    uint32_t Size[SIZECONF] = {0x000000AA};
    id1000500A_writeData(Size, SIZECONF);
    printf("Write configuration register: Csize\n");

    // Convert MemX and MemY to uint8_t arrays
    uint8_t MemX_uint8[10];
    for (int i = 0; i < 10; i++) {
        MemX_uint8[i] = (uint8_t) MemX[i];
    }

    uint8_t MemY_uint8[5];
    for (int i = 0; i < 5; i++) {
        MemY_uint8[i] = (uint8_t) MemY[i];
    }

        // Start IP
    printf("Start IP\n\n");
    id1000500A_startIP();

    // Show status after start
    id1000500A_status();

    // Calculate the convolution of MemX and MemY
    uint16_t MemZ[AUX];
    conv(MemX_uint8, SIZEX, MemY_uint8, SIZEY, MemZ);

        // Read data from MemZ
    printf("\nMemZ Data: [");
    for(int i = 0; i < AUX; i++) {
        printf("0x%08X", MemZ[i]);
        if(i != AUX-1) {
            printf(", ");
        }
    }
    printf("]\n");

    // Finish
    id1000500A_finish();

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

    return 0;

}
```

7.2. Driver functions

7.2.1. id1000500A_init

```
int32_t id1000500A_init(const char *connector, uint_8 nic_addr, uint_8 port,
const char *csv_file)
```

Configure and initialize the connection to control the IP convulution 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. id1000500A_writeData

```
int32_t id1000500A_writeData(uint32_t *data, uint32_t data_size, uint_8
nic_addr, uint_8 port)
```

Write data in the IP convolution 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. id1000500A _readData

```
int32_t id1000500A_readData(uint32_t *data, uint32_t data_size, uint_8
nic_addr, uint_8 port)
```

Read data from the IP convolution 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. id1000500A_startIP

```
int32_t id1000500A_startIP(uint_8 nic_addr, uint_8 port)
```

Start processing in IP convolution.

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. id1000500A_enableINT

```
int32_t id1000500A_enableINT(int_8 nic_addr, uint_8 port)
```

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

Returns:

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

7.2.6. id00001001_disableINT

```
int32_t id1000500A_disableINT(int_8 nic_addr, uint_8 port)
```

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

Returns:

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

7.2.7. id1000500A_status

```
int32_t id1000500A_status(int_8 nic_addr, uint_8 port)
```

Show IP convolution status.

Returns:

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

7.2.8. id1000500A_waitINT

```
int32_t id1000500A_status(int_8 nic_addr, uint_8 port)
```

Wait for the completion of the process.

Returns:

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

7.2.9. id1000500A_finish

```
int32_t id1000500A_finish(void)
```

Finishes the IP processing by performing necessary cleanup operations to properly finalize the IP state and release any resources used.

Returns:

- int32_t Returns 0 if the function has been completed successfully.

7.2.10. id1000500A_conv

```
int32_t id1000500A_finish(unit8_t *X, unit8_t sizeX, uint8_t *Y, unit8_t sizeY, unit16_t *resut)
```

Performs a convolution operation on two input arrays, X and Y. The sizes of these arrays are sizeX and sizeY, respectively. The result of the convolution is stored in the result array, which should be pre-allocated with a size of sizeX + sizeY - 1.

Returns:

- void: This function does not return a value.