ID1000500B
CONVOLUTION IP-CORE USER MANUAL

1. DESCRIPTION

The Convolution IP-core is a processing system and can be used as a reference example for convolution testing and demonstration purposes.

After receiving a start command, this IP-core uses data from its input memory and data from its internal ROM memory to generate the result of the convolution into its output memory. In addition, the size of the input memory can be configurated via software.

1.1. CONFIGURABLE FEATURES

Software configurations	**Description**
Input Memory Size	Input size less than 32 bits

1.2. TYPICAL APPLICATION

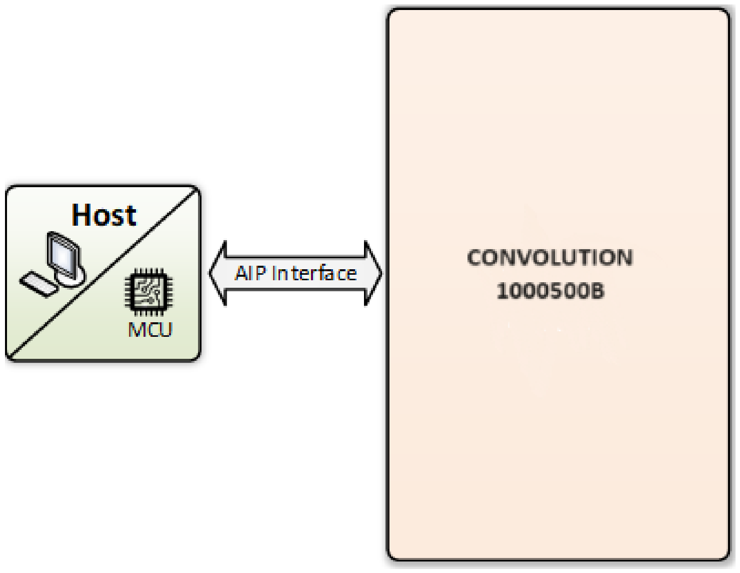

Figure 1.1 IP Convolution connected to a host

2. CONTENTS

1. DESCRIPTION 1
1.1. CONFIGURABLE FEATURES 1
1.2. TYPICAL APPLICATION 1
2. CONTENTS 2
2.1. List of figures 3
2.2. List of tables 3
3. INPUT/OUTPUT SIGNAL DESCRIPTION 4
4. THEORY OF OPERATION 5
5. AIP interface registers and memories description 6
5.1. Status register 6
5.2. Configuration Size Y register 7
5.3. Input data memory 7
5.4. Output data memory 7
6. PYTHON DRIVER 8
6.1. Usage example 8
6.2. Methods 9
6.2.1. Constructor 9
6.2.2. conv 9
6.2.3. writeData 9
6.2.4. readData 9
6.2.5. startIP 10
6.2.6. sizeY_config 10
6.2.7. finish 10
6.2.8. enableINT 10
6.2.9. disableINT 10
6.2.10. status 11
6.2.11. waitInt 11
6.2.12. getID 11
6.2.13. clearStatus 11

7. C DRIVER 11
7.1. Usage example 11
7.2. Driver functions 12
7.2.1. id00001001_init 12
7.2.2. id00001001_writeData 12
7.2.3. id00001001_readData 12
7.2.4. id00001001_startIP 13
7.2.5. id00001001_enableDelay 13
7.2.6. id00001001_disableDelay 13
7.2.7. id00001001_enableINT 13
7.2.8. id00001001_disableINT 14
7.2.9. id00001001_status 14
7.2.10. id00001001_waitINT 14

2.1. List of figures

Figure 1.1 IP Convolution connected to a host 1
Figure 5.1 Basic schematic of the IP Core block with the ipm block. .... **¡Error! Marcador no definido.**
Figure 5.2 IP Accelerator initialization and selection of the configs file. .... **¡Error! Marcador no definido.**
Figure 5.3 Expected result after processing data with the Ip Convolution. .. **¡Error! Marcador no definido.**
Figure 6.1 IP Convolution status register 6
Figure 6.2 Configuration input size register 7

2.2. List of tables

Table 1 IP Convolution input/output signal description 4

3. INPUT/OUTPUT SIGNAL DESCRIPTION

Table 1 IP Dummy 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 uses the data from its input and the data from its internal ROM to generate the convolution between the input data and the ROM data and stores the outcome into the output memory after receiving a start processing command. This processing is executed as soon as the command is received, however, the size of the input data must be configurated to properly function. The operation of the IP Convolution it is based on the following expression

$$(f * g)(t) \approx {}_{def}\int \infty - \infty f(\tau)g(t-\tau)dr$$

Convolution is a mathematical operation that combines two functions to describe the overlap between them. Convolution takes two functions and “slides” one of them over the other, multiplying the function values at each point where they overlap, and adding up the products to create a new function. This process creates a new function that represents how the two original functions interact with each other.

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 Convolution state:
0: The IP Convolution is not busy and ready to start a new process.
1: The IP Convolution 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 Convolution has generated an interruption:
0: interruption not generated.
1: the IP Convolution has successfully finished its processing.
Writing this bit to 1 will clear the interruption flag DN.

5.2. Configuration Size Y register

Config: CREG_CONF_SIZEY
Size: 32 bits
Mode: Write

This register is used to configure the size of the input data before the core starts with the process of the convolution. See Figure **5.2**

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
																											Size [4:0]				
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

Figure 5.2 Configuration size Y register.

Bits 4:0 – **SIZE Y:** The size of the input data can be any value in the range 0x00000001-0x00000020. A value of 0 is not possible.

Bit 31:5 – **NOT USED**.

5.3. Input data memory

Config: MMEM_Y_IN
Size: Nx32 bits (N=32,64,128)
Mode: Write

This memory is used to store data to be processed by the IP Convolution core. The size of this memory is set as a hardware parameter before the synthesis. It has support for storing 32, 64 and 128, 32-bit words.

5.4. Output data memory

Config: MMEM_Z_OUT
Size: Nx64 bits (N=32,64,128,256)
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 result of the convolution. The size of this memory is set as a hardware parameter before the synthesis. It has support for storing 32, 64, 128, and 256 32-bit words.

6. PYTHON DRIVER

The file *id1000500B.py* contains the conv_core 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 conv_core 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 method conv operates the entire convolution process. It only needs the input data as a parameter, in this case the input data is fixed in a List dataY. Internally the input memory is written by using the writeData method. Then, the SizeY_config method is used to configurate the input data size, and 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. The conv method returns a List with the result of the convolution, that it is store in a List dataZ.

```
import sys, random, time, os
from id00001001 import dummy
from ipdi.ip.pyaip import pyaip, pyaip_init, Callback

if __name__ == "__main__":
    import sys, random, time, os

    logging.basicConfig(level=logging.INFO)
    connector = '/dev/ttyACM0'
    csv_file = '/home/ingemtz/project_SoC/IP_MODULE/ID1000500B_config.csv'
    addr = 1
    port = 0

    data_Y = [0x0000001B, 0x0000001C, 0x00000018, 0x00000028, 0x00000028]
    modeloOro  =  [0x0000001B,  0x000000A3,  0x000000F5,  0x00000160,  0x0000022F,  0x000002CE,
0x0000032B,
                 0x000003B1,   0x00000411,   0x0000044B,   0x000003B4,   0x000002A8,   0x000001E0,
0x000000C8]

    try:
        ipm = conv_core(connector, addr, port, csv_file)
        logging.info("Test Convolution: Driver created")
    except:
        logging.error("Test Convolution: Driver not created")
        sys.exit()

    dataZ = ipm.conv(data_Y)
    print(f'data_Z Data: {[f"{x:08X}" for x in dataZ]}\n')

    for x, y in zip(modeloOro, dataZ):
        logging.info(f"TX: {x:08x} | RX: {y:08x} | {'TRUE' if x == y else 'FALSE'}")

    ipm.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 Dummy 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. conv

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

Operates the convolution process

Parameters:

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

Returns:

• List[int]	Convolution result data read from the output memory.

6.2.3. writeData

```
def writeData(self, dataY):
```

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.

6.2.4. readData

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

Read data from the IP Dummy output memory.

Parameters:

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

Returns:

List[int]	Data read from the output memory.

6.2.5. startIP

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

Start processing in IP Convolution.

Returns:

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

6.2.6. sizeY_config

```
def sizeY_config (self, size_Y_config):
```

Configurate the input data size in IP convolution processing.

Parameters:

size_Y_config (int):	Size of the input data

Returns:

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

6.2.7. finish

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

Disable the connection between the bridge and the host.

Returns:

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

6.2.8. 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.

6.2.9. 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.

6.2.10. status

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

Show IP Convolution status.

Returns:

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

6.2.11. waitInt

```
def waitInt(self):
```

Wait for the completion of the process.

Returns:

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

6.2.12. getID

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

6.2.13. clearStatus

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

Clears IP Dummy interruptions (bit DONE of the STATUS register).

7. C DRIVER

In order to use the C driver, it is required to use the files: *id00001001.h, id00001001.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 Dummy core is presented.

```
CODE SAMPLE:
```

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 Convolution 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 Convolution 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 Dummy 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)
```

Read data from the IP Dummy output memory.

Parameters:

• data:	Pointer to the first element where the read data will be stored.
• data_size:	Number of elements to be read.

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.

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 _configSizeY

```
int32_t id1000500b_ configSizeY (uint32_t *data, uint32_t data_size)
```

Configurate the input data size in the configuration register.

Parameters:

• data:	Pointer to the first element where the read data will be stored.
• data_size:	Number of elements to be read.

Returns:

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

7.2.6. Id1000500b _clearIntDone

```
int32_t id1000500b_clearIntDone(void)
```

Clears IP Dummy interruptions (bit DONE of the STATUS register).

Returns:

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

7.2.7. Id1000500b _enableINT

```
int32_t id1000500b_enableINT(void)
```

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

Returns:

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

7.2.8. Id1000500b _disableINT

```
int32_t id1000500b_disableINT(void)
```

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

Returns:

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

7.2.9. Id1000500b _status

```
int32_t id1000500b_status(void)
```

Show IP Dummy status.

Returns:

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

7.2.10. Id1000500b _waitINT

```
int32_t id1000500b_status(void)
```

Wait for the completion of the process.

Returns:

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

7.2.11. Id1000500b _finish

```
int32_t id1000500b_finish(void)
```

Disable the connection between the bridge and the host.

Returns:

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