

---

## ID1000500B CONVOLUTION IP-CORE USER MANUAL

### 1. DESCRIPTION

The Convolution IP-core is an essential tool for applications requiring signal processing, such as data analysis, and more. This system performs convolution operations on data stored in memory, applying filters and effects to enhance data quality and analysis.

Starting with an input signal "Y", the coprocessor performs processing and stores the results in another memory "Z". This process involves applying filters by multiplying and summing products between the values of the input signal and a fixed signal stored internally.

The Convolution IP-core is a versatile and fundamental tool for real-time data processing, providing an effective solution for applications requiring fast and accurate data filtering and analysis.

#### 1.1. CONFIGURABLE FEATURES

Software configurations	Description
Size Y	defines the size of the data input for processing, allowing the module to be tailored to different application requirements

#### 1.2. TYPICAL APPLICATION

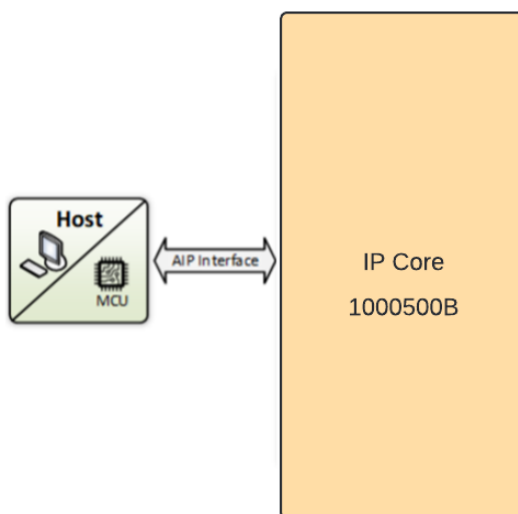


Figure 1.1 IP Core connected to a host

---

## 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 Y.....	7
5.4.	Output data memory Z.....	7
6.	PYTHON DRIVER.....	8
6.1.	Usage example.....	8
6.2.	Methods.....	9
6.2.1.	Constructor.....	9
6.2.2.	writeData.....	9
6.2.3.	readData.....	9
6.2.4.	startIP.....	9
6.2.5.	enableINT.....	10
6.2.6.	disableINT.....	10
6.2.7.	status.....	10
6.2.8.	waitINT.....	10
6.2.9.	conv.....	11
6.2.10.	SizeY_config.....	11
7.	C DRIVER.....	12
7.1.	Usage example.....	12
7.2.	Driver functions.....	13

---

---

7.2.1.	ID1000500B_init.....	13
7.2.2.	ID1000500B_writeData.....	13
7.2.3.	ID1000500B_readData.....	13
7.2.4.	ID1000500B_startIP .....	14
7.2.5.	ID1000500B_enableDelay.....	14
7.2.6.	ID1000500B_disableDelay .....	14
7.2.7.	ID1000500B_enableINT .....	15
7.2.8.	ID1000500B_disableINT .....	15
7.2.9.	ID1000500B_status .....	15
7.2.10.	ID1000500B_waitINT .....	15
7.2.11.	conv.....	16

## 2.1. List of figures

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

## 2.2. List of tables

Table 1	IP Convolution input/output signal description.....	4
---------	---	---

### 3. INPUT/OUTPUT SIGNAL DESCRIPTION

Table 1 Convolution IP Core input/output signal description

Signal	Bitwidth	Direction	Description
<b>General signals</b>			
Clk	1	Input	System clock
rst_a	1	Input	Asynchronous system reset, low active
en_s	1	Input	Enables the IP Core functionality
<b>AIP Interface</b>			
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 configured interruption bits.
<b>Core signals</b>			

## 4. THEORY OF OPERATION

Implementation of a convolution coprocessor, where the data from the Y-signal (shifted signal) are from an external 32-address the Y signal (shifted signal) is from an external 32-address memory and the H signal (fixed signal) is from an internal memory in the data from the H signal (fixed signal) is from an internal memory in the convolution.

Manually the convolution resolution uses the following formula:

$$Z[n] = \sum_{k=0}^{M-1} H[k] \cdot Y[n-k]$$

Where:

**$Z[n]$**  → Convolution result at position “n”.

**$H[k]$**   $\rightarrow$  Fixed signal at position “k”.

$Y[n - k]$  → The signal of the input shifted by “ $k$ ” positions with respect to “ $n$ ”.

Convolution in the coprocessor involves multiplication and addition of products between the values of the fixed signal and the shifted values of the input signal using the above formula. This process is repeated at each “n” position of the output signal.

The result of the convolution will have  $n + m - 1$  product sums. That is to say will be obtained at the output, a vector of this size where:

$n \rightarrow$  Number of data in fixed signal "H".

$m \rightarrow$  Number of data in the input signal “Y”.

The process is responsible for shifting the input signal, using its positions in the vector, and multiplying it with the fixed signal, in the same way using the positions, it only the multiplication of the numbers in position that crosses, as shown in the following image, and performs the in the following image, and the corresponding sum of products is performed.

			H0	H1	H2	...	Hn-1					Z0 =	H0 * Y0
Ym-1	...	Y1	Y0									Z1 =	H0 * Y1 + H1 * Y0
		Ym-1	Y1	Y0								Z2 =	H0 * Y... + H1 * Y1 + H2 * Y0
			Ym-1	Y1	Y0							Z3 =	H0 * Ym-1 + H1 * Y... + H2 * Y1 + H... * Y0
				Ym-1	Y1	Y0						Z4 =	H1 * Ym-1 + H2 * Y... + H... * Y... + Hn-1 * Y1
					Ym-1	Y1	Y0					Z5 =	H2 * Ym-1 + H... * Y... + Hn-1 * Y...
						Ym-1	Y1	Y0				Z... =	H... * Ym-1 + Hn-1 * Y...
							Ym-1	Y1	Y0			Zm+n-1 =	Hn-1 * Ym-1

## 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

Figure 5.1 Convolution IP Core 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 “**B**usy”.

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 Size Y register

Config: CONFIG\_REGISTER

Size: 5 bits

Mode: Write

This register is used to configure Size Y. 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_Y [4:0]				
RESERVED																											w	w	w	w	w

Figure 5.2 Configuration Size Y register.

## 5.3. Input data memory Y

Config: MEMORY\_Y

Size: Nx32 bits (N=32, 64, 128, 256)

Mode: Write

MEMORY\_Y serves as the input memory for the convolution coprocessor, storing the data to be processed. Its size is configurable, supporting storage capacities of 32, 64, 128, or 256 words of 32 bits each. The mode of operation is set to write, allowing data to be written into the memory for processing.

## 5.4. Output data memory Z

Config: MEMORY\_Z

Size: Nx32 bits (N=32, 64, 128, 256)

Mode: Read

MEMORY\_Z acts as the output memory for the convolution coprocessor, storing the processed data. After the convolution operation is completed, the results stored in MEMORY\_Z will be the output data. Its size is configurable, supporting storage capacities of 32, 64, 128, or 256 words of 32 bits each. The mode of operation is set to read, allowing the processor to read the processed data from the memory.

## 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 `disable_INT` method is then called to disable interruptions. The `conv` method is used to perform a convolution operation on the input data `data_Y`, and the result is stored in `data_Z`. The output data is printed and compared to the expected data `modeloOro`, logging whether each pair of transmitted (TX) and received (RX) values match. The `status` method is used to display the status of the IP Core, and finally, the `finish` method is called to finish the connection.

```
import logging, time
from ipdi.ip.pyaip import pyaip, pyaip_init
from ID1000500B import conv_core

logging.basicConfig(level=logging.INFO)
connector = '/dev/ttyACM0'
csv_file =
'/home/anette/intelFPGA_lite/22.1std/quartus/HDL/ID1000500B/ID1000500B_config.csv'
addr = 1
port = 0

data_Y = [0x00000001, 0x0000000B, 0x00000004, 0x000000FE, 0x000000FF]
modeloOro=[0x0000FFFF, 0x0000FFF7, 0x00000017, 0x00000044, 0x0000003C, 0x0000006C,
0x00000006, 0x0000FFE2, 0x0000007A, 0x00000035, 0x0000FFF4, 0x0000FFF8, 0x0000FFFE, 0x0000FFFF]

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()

ipm.disableINT()

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.status()

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. writeData

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

Write data in the IP Dummy 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, 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.

### 6.2.4. startIP

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

Start processing in IP Dummy.

**Returns:**

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

### 6.2.5. enableINT

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

Enable IP Dummy 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 Dummy 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 Dummy 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, Y):
```

Configures and controls the IP Core to perform a convolution operation with the input data provided.

**Returns:**

- List Returns the list of data resulting from the convolution operation performed by the IP Core.

### 6.2.10. SizeY\_config

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

Configures the size of Y and enables it.

**Returns:**

- None

## 7. C DRIVER

In order to use the C driver, it is required to use the files: *ID1000500B.h*, *ID1000500B.c* that contain the driver functions definition and implementation. The functions defined in this library are used to control the Convolution IP-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 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 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, uint_8
nic_addr, uint_8 port)
```

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.
- 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 Dummy.

**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\_enableDelay**

```
int32_t ID1000500B _enableDelay(uint_32 msec, int_8 nic_addr, uint_8 port)
```

Set and enable delay in IP Dummy processing.

**Parameters:**

- `msec`: Number of milliseconds of delay.
- `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.

**7.2.6. ID1000500B\_disableDelay**

```
int32_t ID1000500B _disableDelay(int_8 nic_addr, uint_8 port)
```

Disable delay in IP Dummy processing.

**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.

### 7.2.7. ID1000500B\_enableINT

```
int32_t ID1000500B _enableINT(int_8 nic_addr, uint_8 port)
```

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(int_8 nic_addr, uint_8 port)
```

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(int_8 nic_addr, uint_8 port)
```

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(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.

### 7.2.11. conv

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

Configures and controls the IP Core to perform a convolution operation with the input data provided.

{

**Returns:**

- List Returns the list of data resulting from the convolution operation performed by the IP Core.