ID00001001
DUMMY IP-CORE USER MANUAL

1. DESCRIPTION

The Dummy IP-core is the most basic processing system and can be used as a reference example for testing and demonstration purposes.

After receiving a start command, this IP-core only copies data from its input memory to its output memory. In addition, the processing delay can be configurated via software.

1.1. CONFIGURABLE FEATURES

Software configurations	**Description**
Processing delay	Delay in milliseconds before the data transfer operation completes. Delay in milliseconds before the data transfer operation completes.
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

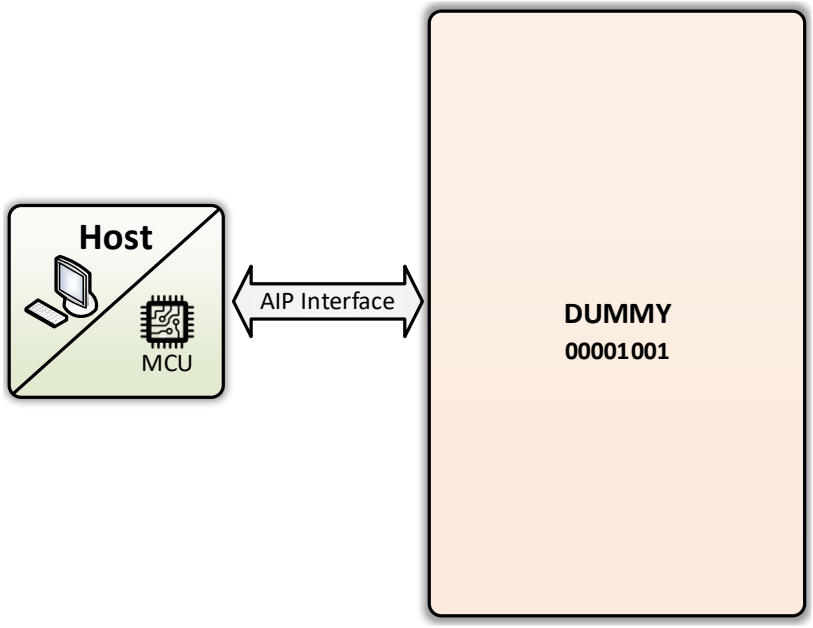

Figure 1.1 IP Dummy connected to a host

2. CONTENTS

2.1. List of figures

2.2. List of tables

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 Dummy core copies the data from its input to its output memory after receiving a start processing command. This processing is executed as soon as the command is received, however, a delay in milliseconds can be enabled to retard the data movement between memories. A millisecond delay is calculated by using a simple counter that generates an internal flag in the core when the counter reaches a threshold calculated as

$$millisecond\_count = clk_{freq} * 0.001,$$

where clk_{freq} is the operating clock frequency in Hz of the IP-Core. For demonstration purposes and to maintain this core as simple as possible, this value was set internally for a 50 MHz clock, which is commonly used in commercial FPGAs. Therefore, each time the internal counter reaches the threshold a single millisecond is detected. With this setup, software functions can be used to set different delays in milliseconds.

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 (rw)	Reserved	Reserved	BSY (r)	Reserved	DN (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 delay register

Config: CDELAY
Size: 32 bits
Mode: Write

This register is used to configure a delay time in milliseconds before the core starts copying data from the input to the output data memory. See Figure 5.2

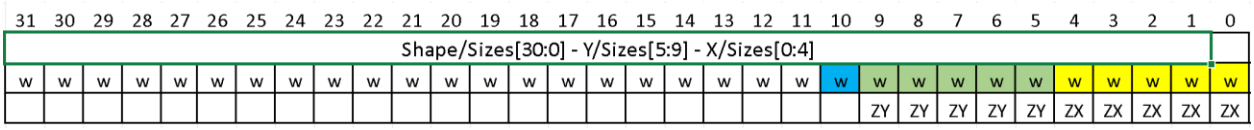

Figure 5.2 Configuration delay register.

Bits 31:1 – **DELAY:** DELAY count value. The DELAY value can be any value in the range 0x00000001-0xFFFFFFFF. A value of 0 is possible, meaning that no delay will occur.

Bit 0 – **ENA**: delay enable. When ENA is set to 1, a delay of DELAY milliseconds elapses before the core starts its processing.

0: delay counter disabled.

1: delay counter enabled.

5.3. Input data memory

Config: MDATAIN
Size: Nx32 bits (N=32,64,128,256)
Mode: Write

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

5.4. Output data memory

Config: MDATAOUT
Size: Nx32 bits (N=32,64,128,256)
Mode: Read

This memory is used to store processed data by the IP Dummy. After the IP Dummy completes its processing, the data stored in this memory will be the same as the input data memory. 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 *id00001001.py* contains the dummy class definition. This class is used to control the IP Dummy 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.

```
from ipdi.ip.pyaip import pyaip, pyaip_init

import sys

try:
    connector = '/dev/ttyACM0'
    nic_addr = 1
    port = 0
    csv_file =
'/home/a/Documents/HDL/ID1000500A_AlejandroPardo_LeonelGallo/config/ID1000500A_config.csv'

    aip = pyaip_init(connector, nic_addr, port, csv_file)

    aip.reset()

    #========================================
    # Code generated with IPAccelerator

    ID = aip.getID()
    print(f'Read ID: {ID:08X}\n')

    STATUS = aip.getStatus()
    print(f'Read STATUS: {STATUS:08X}\n')

    MemX = [0x00000001, 0x00000002, 0x00000003, 0x00000004, 0x00000003, 0x00000007]

    print('Write memory: MdataX')
    aip.writeMem('MdataX', MemX, 6, 0)
    print(f'MemX Data: {[f"{x:08X}" for x in MemX]}\n')

    MemY = [0x00000003, 0x00000003, 0x00000005, 0x00000006, 0x00000007]

    print('Write memory: MdataY')
    aip.writeMem('MdataY', MemY, 5, 0)
    print(f'MemY Data: {[f"{x:08X}" for x in MemY]}\n')

    Size = [0x000000A6]

    print('Write configuration register: Csize')
    aip.writeConfReg('Csize', Size, 1, 0)
    print(f'Size Data: {[f"{x:08X}" for x in Size]}\n')

    print('Start IP\n')
    aip.start()

    STATUS = aip.getStatus()
    print(f'Read STATUS: {STATUS:08X}\n')

    print('Read memory: MdataZ')
    MemZ = aip.readMem('MdataZ', 10, 0)
    print(f'MemZ Data: {[f"{x:08X}" for x in MemZ]}\n')

    print('Clear INT: 0')
    aip.clearINT(0)

    STATUS = aip.getStatus()
    print(f'Read STATUS: {STATUS:08X}\n')

    #========================================

    aip.finish()

except:
    e = sys.exc_info()
```

```
    print('ERROR: ', e)

    aip.finish()
    raise
```

6.1. Usage example

In the following code a basic test of the IP Dummy core is presented. First, it is required to create an instance of the dummy object class. The constructor of this class requires the network address and port where the IP Dummy is connected, the communication port, and the path where the configs csv file is located. Thus, the communication with the IP Dummy will be ready. In this code, the input memory is written with random data by using the writeData method. Then, the enableDelay method is used to set a delay of 2000 milliseconds, 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.

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

logging.basicConfig(level=logging.INFO)
connector = 'COM7'
csv file = 'E:/id00001001.csv'
addr = 1
port = 0
aip_mem_size = 32

try:
    dmy = dummy(connector, addr, port, csv_file)
    logging.info("Test Dummy: Driver created")
except:
    logging.error("Test Dummy: Driver not created")
    sys.exit()

random.seed(1)

dmy.disableINT()

WR = [random.randrange(2**32) for i in range(0, aip_mem_size)]

dmy.writeData(WR)
logging.info(f"Data generated with {len(WR):d}")
logging.info(f'TX Data {[f"{x:08X}" for x in WR]}')

dmy.enableDelay(2000)

dmy.startIP()

dmy.waitInt()

RD = []
RD = dmy.readData(aip_mem_size)
logging.info(f'RX Data {[f"{x:08X}" for x in RD]}')

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

dmy.disableDelay()

dmy.enableINT()
dmy.status()

dmy.disableINT()
```

```
dmy.status()

dmy.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. enableDelay

```
def enableDelay(self, msec):
```

Set and enable delay in IP Dummy processing.

Parameters:

- msec (int): Number of milliseconds of delay.

Returns:

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

6.2.6. disableDelay

```
def disableDelay(self):
```

Disable delay in IP Dummy processing.

Returns:

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

6.2.7. 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.

6.2.8. 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.

6.2.9. status

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

Show IP Dummy status.

Returns:

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

6.2.10. waitINT

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

Wait for the completion of the process.

Returns:

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

6.2.11. pyaip_init

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

Initializes the aip object to control the IP Dummy device.

Returns:

- connector (string): Communications port used by the host.
- nic_addr (int): Network address where the kernel is connected.
- port (int): Port where the kernel is connected.
- csv_file (string): Location of the IP Dummy configuration CSV file.

6.2.12. pyaip_init

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

Initializes the aip object to control the IP Dummy device.

Returns:

- connector (string): Communications port used by the host.
- nic_addr (int): Network address where the kernel is connected.
- port (int): Port where the kernel is connected.
- csv_file (string): Location of the IP Dummy configuration CSV file.

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. id00001001_init

```
int32_t id00001001_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. id00001001_writeData

```
int32_t id00001001_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. id00001001_readData

```
int32_t id00001001_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. id00001001_startIP

```
int32_t id00001001_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. id00001001_enableDelay

```
int32_t id00001001_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. id00001001_disableDelay

```
int32_t id00001001_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. id00001001_enableINT

```
int32_t id00001001_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. id00001001_disableINT

```
int32_t id00001001_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. id00001001_status

```
int32_t id00001001_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. id00001001_waitINT

```
int32_t id00001001_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.