

Representing streaming data

	Organisatie / Organization	Datum / Date
Auteur(s) / Author(s): Eric Kooistra	ASTRON	2015
Controle / Checked: Andre Gunst	ASTRON	
Goedkeuring / Approval: Andre Gunst	ASTRON	
Autorisatie / Authorisation: Handtekening / Signature Andre Gunst	ASTRON	

© ASTRON 2015
All rights are reserved. Reproduction in whole or in part is
prohibited without written consent of the copyright owner.

Distribution list:

Group:	Others:
Hajee Pepping (HJP) Daniel van der Schuur (DS) Leon Hiemstra (LH) Alwin Zanting (AZ)	Andre Gunst (AG)

Document history:

Revision	Date	Author	Modification / Change
0.1	2015-04-23	E. Kooistra	Creation.
0.2	2015-04-28	E. Kooistra	Clarified use of valid, sop, eop, sync, BSN.
0.3	2015-05-12	E. Kooistra	Updated and extended after review with HJP, DS. Added more examples and a conclusion.
1.0	2015-05-28	E. Kooistra	Clarified functional and implementation aspects of the array notation. Added appendix with executable description in Python made by Daniel van der Schuur.

Table of contents:

1	Introduction.....	5
1.1	Purpose	5
1.2	Scope.....	5
2	Array notation	6
2.1	Definition.....	6
2.1.1	Example.....	6
2.2	Transpose operation.....	7
2.3	Time series data	8
2.3.1	Sampled data.....	8
2.3.2	Framed data	8
3	Representation in digital logic	9
3.1	Streaming interface	9
3.2	Data	9
3.3	Control strobes and meta-data	9
3.3.1	Sampled data – valid	9
3.3.2	Framed data – valid, sop, eop.....	10
3.3.3	Sync pulse – sync.....	10
3.3.4	Block sequence number – BSN.....	10
3.4	Flow control – ready, xon	10
3.5	DP streaming records.....	11
3.6	DP packetized framed data	11
4	Aspects of handling streaming data	12
4.1	Transportation of stream control versus local regeneration	12
4.2	Wormhole processing versus store-and-forward processing	12
4.3	Stream control inside the FPGA and at the external FPGA inputs	12
4.4	Parallel stream alignment	13
4.5	Concatenating parallel data versus serial multiplexing of data	13
4.6	Data packing and unpacking	13
5	Conclusion.....	14
6	Appendix: Using numpy to represent streaming data	15
6.1	Python script.....	15
6.2	Script result log	16

References:

- [1] "Timing in the Streaming Interface", ASTRON-RP-481, E. Kooistra
- [2] "Avalon Interface Specifications", mnl_avalon_spec.pdf, Altera
- [3] "Specification for module interfaces using VHDL records", ASTRON-RP-380, E. Kooistra
- [4] "DP Streaming Module Description (The ready signal of the streaming interface)", ASTRON-RP-382, E. Kooistra
- [5] "Data Path Packet Interface Specification", ASTRON-SP-042, E. Kooistra
- [6] "Uthernet interface specification", ASTRON-SP-041, E. Kooistra

Terminology:

ADC	Analogue to Digital Conversion
BSN	Block Sequence Number (timestamp)
DP	Data Path
eop	End of Packet (or frame, or block)
FIFO	First In First Out
FPGA	Field Programmable Gate Array
Im	Imaginary part of complex number
IO	Input Output
MM	Memory-mapped
Re	Real part of complex number
siso	Sink In Source Out (direction of the flow control)
sop	Start of Packet (or frame, or block)
sosi	Source Out Sink In (direction of the data)
UTC	Coordinated Universal Time

1 Introduction

1.1 Purpose

This document defines an array notation for streaming data that is useful for describing the streaming data in interface specification documents. Furthermore this document relates the array notation to how the streaming data can be modelled in Python and how it is represented in digital logic (e.g. VHDL).

1.2 Scope

This document specifies the array notation. Regarding the digital representation it provides a top level overview and refers to [1], [3] and [5] for the details. In this way this document provides a direct link between a compact notation for streaming data in a specification and the actual implementation in digital logic.

2 Array notation

2.1 Definition

To describe the streaming data it is useful to use the array notation:

Equation 1 $(int16) s_{a,b}[n][m]$

The subscript indices indicate parallel streams and the array index contains serial data in the stream. The order of the indices matters. The last subscript and the last array index varies the fastest. The type cast defines the unit size of the data in the stream. Indices can be combined or extra sub-indices can be created so:

Equation 2 $s_{a,b}[n][m] = s_{a,b}[t] = s_p[n][m] = s_p[t]$

where in this case the serial index t and the parallel index p are given by:

Equation 3 $t = n * M + m$

Equation 4 $p = a * B + b$

Default an index range starts at 0 and increments by 1. Other index ranges are possible too to describe a certain order of the elements in a block. Replicating of data can be represented by adding another sub-index, e.g. the result after K times replicating every element in $s[n]$ can be noted as $s[n][k]$.

The number of sub-indices that is used to index the elements of s depend on how the data in s has to be interpreted. At the functional level the minimum amount of indices should be used. At implementation level extra indices can be needed to reflect how the implementation details impact the streaming data. The distinction between parallel and serial indices can be a functional distinction e.g. because the data streams originate from parallel sources, but it may also be an implementation detail e.g. in case the processing needs to be distributed over parallel units. Therefore the streaming data of Equation 1 can also be represented as a single stream:

Equation 5 $s[a][b][n][m]$

The representation of Equation 5 does not show the distinction between parallel and serial, but the advantage is that it can be used in programming languages like Python to model the streaming data in a single variable as shown by `S_abnm` in the Python script in appendix 6.1.

2.1.1 Example

For example if indices a , b , n and m have length $A=2$, $B=3$, $N=2$, $M=4$ then the unrolled array has $P=A*B=6$ parallel streams (p) and $T=N*M=8$ values in series (t). Table 1 shows $s_{a,b}[n][m]$ in the format of $s_p[t]$:

s	t			0	1	2	3	4	5	6	7
	n			0	0	0	0	1	1	1	1
	m			0	1	2	3	0	1	2	3
p	a	b									
0	0	0		$s_{0,0}[0][0]$	$s_{0,0}[0][1]$	$s_{0,0}[0][2]$	$s_{0,0}[0][3]$	$s_{0,0}[1][0]$	$s_{0,0}[1][1]$	$s_{0,0}[1][2]$	$s_{0,0}[1][3]$
1	0	1		$s_{0,1}[0][0]$	$s_{0,1}[0][1]$	$s_{0,1}[0][2]$	$s_{0,1}[0][3]$	$s_{0,1}[1][0]$	$s_{0,1}[1][1]$	$s_{0,1}[1][2]$	$s_{0,1}[1][3]$
2	0	2		$s_{0,2}[0][0]$	$s_{0,2}[0][1]$	$s_{0,2}[0][2]$	$s_{0,2}[0][3]$	$s_{0,2}[1][0]$	$s_{0,2}[1][1]$	$s_{0,2}[1][2]$	$s_{0,2}[1][3]$
3	1	0		$s_{1,0}[0][0]$	$s_{1,0}[0][1]$	$s_{1,0}[0][2]$	$s_{1,0}[0][3]$	$s_{1,0}[1][0]$	$s_{1,0}[1][1]$	$s_{1,0}[1][2]$	$s_{1,0}[1][3]$
4	1	1		$s_{1,1}[0][0]$	$s_{1,1}[0][1]$	$s_{1,1}[0][2]$	$s_{1,1}[0][3]$	$s_{1,1}[1][0]$	$s_{1,1}[1][1]$	$s_{1,1}[1][2]$	$s_{1,1}[1][3]$
5	1	2		$s_{1,2}[0][0]$	$s_{1,2}[0][1]$	$s_{1,2}[0][2]$	$s_{1,2}[0][3]$	$s_{1,2}[1][0]$	$s_{1,2}[1][1]$	$s_{1,2}[1][2]$	$s_{1,2}[1][3]$

Table 1: Relation between the elements and indices of s

2.2 Transpose operation

A transpose operation is represented by swapping indices in the array notation. The swapping can be done between parallel indices, between series indices and also between parallel and series indices. For example if signal $s_{a,b}[n][m] = s_p[t]$ from Table 1 contains values 0:47 in the following order:

s	t			0	1	2	3	4	5	6	7
	n			0	0	0	0	1	1	1	1
	m			0	1	2	3	0	1	2	3
p	a	b									
0	0	0		0	1	2	3	4	5	6	7
1	0	1		8	9	10	11	12	13	14	15
2	0	2		16	17	18	19	20	21	22	23
3	1	0		24	25	26	27	28	29	30	31
4	1	1		32	33	34	35	36	37	38	39
5	1	2		40	41	42	43	44	45	46	47

Table 2: Example of $s_{a,b}[n][m] = s_p[t]$ with incrementing element values

then a transpose operation between index a and m yields signal $s_{m,b}[n][a] = s_p[t']$:

s	t'			0	1	2	3
	n			0	0	1	1
	a			0	1	0	1
p'	m	b					
0	0	0		0	24	4	28
1	0	1		8	32	12	36
2	0	2		16	40	20	44
3	1	0		1	25	5	29
4	1	1		9	33	13	37
5	1	2		17	42	21	45
6	2	0		2	26	6	30
7	2	1		10	34	14	38
8	2	2		18	42	22	46
9	3	0		3	27	7	31
10	3	1		11	35	15	39
11	3	2		19	43	23	47

Table 3: Example of transpose between index a and m of s

In the Python script in appendix 6.1 the variable S_{pt} prints Table 2 and after the transpose the variable T_{pt} prints Table 3 as shown in the script result log in appendix 6.2.

2.3 Time series data

2.3.1 Sampled data

Time series ADC sampled data can be represented with a single array index $[t]$.

2.3.2 Framed data

Framed data can be represented by extra array indices e.g. $[t][n]$ for frames with N data per frame or $[t][n][m]$ for frames with $N*M$ data per frame. The order of the $[n][m]$ indices defines the order in the frame.

Dependent on what is appropriate for the application the $[t]$ index may increment e.g. by 1 or by $N*M$ or by some other appropriate value. The $[t]$ index may relate directly to absolute time and increments 'forever'.

The blocks of data may also contain time series sample, e.g. for an integration interval. This is represented by separating the index $[t]$ into two time indices e.g. $[tt][ti]$ where the interval size of ti is TI so:

Equation 6: $t = tt * TI + ti$

Equation 7: $tt = t \text{ DIV } TI$
 $ti = t \text{ MOD } TI$

The index ti increments during each interval and restarts when the interval has expired. The index tt increments per interval and increments 'forever'. The purpose of describing the intervals in s is that now this interval can be transposed with some other subblock e.g. for the data stream $s[t][n] = s[tt][ti][n]$ the transpose between n and ti yields $s[tt][n][ti]$. Similar if after a transpose two time series indices become adjacent then they can be merged into a single time index using Equation 6.

2.3.2.1 Block sequence number

For framed data the $[t]$ index that increments 'forever' acts as a timestamp or block sequence number (BSN). The block size is equal to the number of data words that are in the next indices, so e.g. block size is N for $[t][n]$ and $N*M$ for $[t][n][m]$.

3 Representation in digital logic

3.1 Streaming interface

The DP streaming interface consists of:

1. Data
2. Meta-data control information
3. Backpressure flow control.

It follows the Avalon streaming interface specification [2] and has some additional fields (sync, BSN, xon).

3.2 Data

The streaming data is carried by a stream of data words of width ≥ 1 bit. The data can be:

1. Real
2. Complex

For real data the data can represent integers or arbitrary data. Integers can be signed or unsigned. Arbitrary data is typically represented as unsigned hexadecimal values. Arbitrary data can e.g. be packet header data or concatenated integer data. Complex data is integer data with a signed real and a signed imaginary field. The streaming data can also be floating point numbers, however most current DSP applications on FPGAs still use fixed point numbers that are represented by integers.

3.3 Control strobes and meta-data

Attached to the data there can be four control strobes:

1. valid
2. sync
3. sop
4. eop

Attached to the data there can be meta data fields that define additional information to relate the data:

Meta data	Valid at	Description
BSN	sop	Block sequence number that counts blocks of data (see section 3.3.4)
Channel	sop	Identifies multiple streams that are multiplexed within a single stream
Empty	eop	Indicates data ok or that some error occurred during the data
Error	eop	Count the number of unused data symbols in the last data word of a block

The following sub-sections discuss the control strobes and the BSN field in more detail.

3.3.1 Sampled data – valid

In digital logic sampled data only needs a valid strobe to indicate whether the data is valid at that clock cycle. If the data is valid at every clock cycle then even the valid strobe is not needed, because it is then always '1'. However typically a data valid strobe is needed, because the data valid strobe is needed when e.g.:

1. The digital clock frequency is larger than the data rate
2. The data crosses a clock domain
3. Data gaps are needed e.g. to insert a packet header or to pack data into another word width

3.3.2 Framed data – valid, sop, eop

For framed data logic it is convenient to mark the start of packet (sop) and end of packet (eop) by a sop and eop strobe. The sop and eop can only be active if the valid is also active. Between frames, so between the previous eop and a next sop, there can be no valid data cycles. The behavior of the valid during a frame depends on the constraints of the streaming component. There are two cases:

1. Ideally the valid may become inactive at any time so also during a frame
2. It may be required that the valid remains active during the entire frame, e.g. to ease the implementation of the component or e.g. because a protocol like Ethernet requires it.

3.3.3 Sync pulse – sync

The sync pulse is a periodic signal that aligns the streaming data to a starting time t_0 . The sync pulse period depends on the largest time-variant interval in the data path processing [1]. A sync pulse can be used for both sampled data as well as framed data. The sync can only be active if the valid is also active or if the sop is also active in case of framed data. Typically a sync interval contains multiple blocks of data so multiple sop. However it is also possible to have only one block per sync interval in which case the sop and sync become equivalent.

3.3.4 Block sequence number – BSN

The BSN is represented by a signal field that is valid at the sop strobe and is used to detect lost frames and to synchronize streams from different sources. Dependent on the application the BSN may be related directly to real word time (UTC) or it may merely be a block index that restarts after every sync interval.

Typically the BSN increments by 1. However dependent on the application it may also increment with another step size. A combination is also possible, e.g. the BSN at the sync represents UTC and the other BSN at the sop's during the rest of the sync interval represent an arbitrary block index that restarts at 0.

The UTC information of the BSN is typically not used inside the FPGA, because the DSP processing in the FPGA operates on a sync interval time scale. Therefore within and between the FPGAs it can also be sufficient to only pass on the sync and a BSN per sync interval and attach the UTC information only to the data when the data is output to the subsequent post processing on other systems. This scheme requires that the sync interval is sufficiently large such that the UTC information can be uniquely linked to each sync via memory-mapped (MM) control. The advantage of attaching the UTC related BSN already at the input of the system and then pass it along is that no further MM control to provide UTC information with the data is needed.

The BSN as block index within a sync interval typically requires somewhat more than 16 bit to be able to count $\sim 10^6$ blocks per sync interval. For UTC information the BSN needs to be about 64 bit to be able to cover > 10 years, because $10 \text{ years} / 2^{64} = 17 \text{ ps}$. Hence a suitable BSN size is 64 bit because that fits both purposes, without costing too much overhead.

3.4 Flow control – ready, xon

The streaming interface supports two levels of flow control [4]:

1. Ready, operates per clock cycle and relates to the valid
2. Xon, operates per block and relates to the sop and eop

The ready flow control is used to fine tune the input rate to the processing rate of the downstream components. No data should get lost due to ready remaining low for too long. Instead the xon flow control provides coarse flow control and is typically used to drop blocks of data in a controlled way. A component can use xon='0' e.g. when:

1. To avoid FIFO overflow
2. To (re)align between streams
3. During the initialization of a component (e.g. an external interface controller)

3.5 DP streaming records

The streaming data (data, re, im) and the control strobes (sync, valid, sop, eop) and meta-data (BSN, channel, empty, error) are represented as fields in the DP sosi record. The flow control (ready, xon) are represented as fields in the DP siso record [3].

3.6 DP packetized framed data

The framed data can be packetized according to the DP Packet Interface Specification [5]. This packet interface prepends the data with a header containing the information that is valid at the sop and it extends the data with a tail containing the information that is valid at the eop. The header starts with the channel field. The sync and BSN are also transported in the header whereby the most significant bit of the BSN field carries the sync bit. The BSN field is 64 bit, so bit [63] carries the sync and bits [62:0] carry the BSN value. Transporting only 63 bit of the 64-bit BSN value is still sufficient. The tail only contains the error field, The empty is not encoded explicitly, because the DP packet for stream has a known length.

4 Aspects of handling streaming data

4.1 Transportation of stream control versus local regeneration

The stream control consists of strobes (valid, sop, eop, sync) and meta-data index values (BSN) that are attached to the data. In general this stream control can be regenerated locally by locally counting the valid strobes provided that:

1. The first valid also marks the first sync and the first sop
2. The frame size is known
3. No data gets lost during the transport

However typically it is necessary to transport at least some of the stream control information along with the data because in practice data may get lost e.g. due to start up initialization effects, link errors or rare functional errors.

4.2 Wormhole processing versus store-and-forward processing

Wormhole routing implies that a component can already process and output the head of a block of streaming data while it has not yet received the tail data for this block. With store-and-forward routing the component first needs to receive also the tail data, before it starts process and output the block. The advantages of wormhole processing and routing are:

1. It does not introduce more latency to a data stream than needed
2. The block length can be very large, because it does not need to be stored
3. Minimal need for storage and for memory IO (FPGA RAM for FIFOs, external DDR3 memory)

Advantages of store-and-forward processing and routing are e.g.:

1. The tail information about the error field is known before the data is processed further.
2. A scheme of reading data from memory, processing it and writing it to memory forms a clear scheme for implementing a data path.

4.3 Stream control inside the FPGA and at the external FPGA inputs

Internally in the FPGA all components can and should assume that all strobes (valid, sop, eop, sync) and index values (BSN) that are attached to the data are correct. This implies that:

1. All components must function correctly
2. FIFOs must never overflow
3. External inputs must never pass on incomplete frames

4.3.1.1 No functional errors

Components need to be verified with dedicated tests to ensure that they function correctly under all circumstances, such that other components can rely on it without checking.

4.3.1.2 No FIFO overflow

FIFO overflow is considered to be a logical error and should never occur. FIFO overflow can be avoided using xon/xoff block based flow control. Default xon='1', but when the FIFO and subsequent downstream

components cannot accept another block then they can assert `xon='0'` to halt the input data stream. This may lead to blocks of data being dropped, but that is acceptable then.

4.3.1.3 External input block size checker

Errors on an external input link should never lead to a missing eop or an eop that occurs too early or too late. For external inputs an input checker component can ensure that only complete blocks of data are passed on to the internal data path processing. To do this the input checker needs to know the expected block size. For streaming applications the expected block size is typically known and fixed, so therefore the block length then does not need to be transported via the link [5], [6]. If the block length is transported then some header checksum field is also needed to ensure that the received block length value is indeed correct. The input checker only outputs a sop when it has detected a start of packet and it only outputs an eop when it has counted the expected number of data. In this way a corrupted or partially received packet can never cause incomplete blocks of data to enter internal data path processing. Blocks of data can get lost and the values of the received data may get corrupted, but that is acceptable then.

4.4 Parallel stream alignment

A set of parallel streams s_a may originate from up to A different sources. When similar streams from different sources need to be combined then they need to be aligned using the control information. The streams may be misaligned due to differences in latencies that can occur within the separate data paths. If the streams have not started at the same time then the first valid cannot be used to align the streams. The sop can be used to align the streams provided that the latency difference between the streams is less than the block period. If the latency difference between the streams is larger than a block period then also some range of BSN values needs to be accounted for to find alignment. If a less responsive alignment scheme is acceptable than the sync can be used to align data streams instead of the BSN, provided that the sync interval is larger than the latency difference between the streams.

4.5 Concatenating parallel data versus serial multiplexing of data

Parallel streams can be combined into a single stream by concatenating their data fields using the (VHDL) `'&'` operator, provided that:

1. the streams are aligned
2. the streams have the same data format and block format

To transport independent parallel streams via a single stream it is necessary to use multiplexer components that select the different inputs in some order. The original input streams are then identified by the channel field. The disadvantages of using multiplexer components are:

1. cost logic resources
2. may require input FIFOs if the input does not support back pressure
3. initial input latencies can be larger than the steady state latencies and need careful simulation to ensure no FIFOs will overflow
4. does not fit with wormhole routing if the multiplexing is done per block

4.6 Data packing and unpacking

If the streaming data width does not match the data width of an external interface, then the data words need to be resized or repacked. Resizing the data costs no logic resources but can be inefficient because the padded bits are also transported via the external link. Repacking the data costs logic resources but can achieve up to 100% efficiency on the link.

5 Conclusion

The array notation forms a clear link between specification and implementation of streaming data. The array notation is used in documentation. However the array notation can also be used in e.g. Python as shown in appendix 6, to have an executable specification model that can be simulated.

In the implementation the streaming data typically at least needs a valid strobe. The sync strobe is needed for synchronization and to be able to recover from lost data.

Most processing operates on blocks of data and for that it is convenient to pass along also a sop and eop. The block size may be redefined along the data path. E.g. a transpose operation may change the natural block size. Similar at the external interfaces the block size may be redefined to match the interface packet size.

The BSN is a block counter. The BSN may relate directly to UTC. Alternatively only the BSN at the sync relates to UTC. The sync is attached to the data at the input of the system and then passed along. It is convenient to also attach the UTC related BSN already at the input of the system.

To have minimal latency and minimal memory requirements the DP processing uses wormhole processing and routing where possible. The ready flow control supports backpressure to fine tune the input rate to a component without losing data. The xon flow control operates per block of data and is useful to drop data in a controlled way.

6 Appendix: Using numpy to represent streaming data

6.1 Python script

```
#####
#
# Copyright (C) 2015
# ASTRON (Netherlands Institute for Radio Astronomy) <http://www.astron.nl/>
# P.O.Box 2, 7990 AA Dwingeloo, The Netherlands
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program. If not, see <http://www.gnu.org/licenses/>.
#
#####

"""
Purpose:
. Numpy implementations corresponding to the examples in the document
Description:
. Similar to this file, firmware design documents could contain corresponding
  executable specifications.
Usage:
. python ASTRON_RP_1486_Representing_Streaming_Data.py
"""

import numpy as np

print '#####'
print '# Data: S[a][b][n][m]'
print '# . A=2, B=3, N=2, M=4'
print '#####'
S_abnm = np.arange(48).reshape((2,3,2,4))
print S_abnm, '\n'

print '#####'
print '# Data: S[p][t] (in table format)'
print '# . p = a*B+b'
print '# . t = n*M+m'
print '#####'
S_pt = S_abnm.reshape((2*3,2*4))
print S_pt, '\n'

print '#####'
print '# Data transposed : T[m][b][n][a] (swapped index a and m)'
print '# . A=2, B=3, N=2, M=4'
print '#####'
T_mbna = S_abnm.transpose(3,1,2,0)
print T_mbna, '\n'
```

```
print '#####'
print '# Data transposed : T[p][t] (in table format)'
print '#####'
T_pt = T_mbn.reshape(4*3,2*2)
print T_pt, '\n'
```

6.2 Script result log

```
#####
# Data: S[a][b][n][m]
# . A=2, B=3, N=2, M=4
#####
[[[ 0  1  2  3]
  [ 4  5  6  7]]

 [[ 8  9 10 11]
  [12 13 14 15]]

 [[16 17 18 19]
  [20 21 22 23]]]

 [[24 25 26 27]
  [28 29 30 31]]

 [[32 33 34 35]
  [36 37 38 39]]

 [[40 41 42 43]
  [44 45 46 47]]]]

#####
# Data: S[p][t] (in table format)
# . p = a*B+b
# . t = n*M+m
#####
[[ 0  1  2  3  4  5  6  7]
 [ 8  9 10 11 12 13 14 15]
 [16 17 18 19 20 21 22 23]
 [24 25 26 27 28 29 30 31]
 [32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47]]
```



```
#####
# Data transposed : T[m][b][n][a] (swapped index a and m)
# . A=2, B=3, N=2, M=4
#####
[[[ 0 24]
  [ 4 28]]

 [[ 8 32]
  [12 36]]

 [[16 40]
  [20 44]]]

[[[ 1 25]
  [ 5 29]]

 [[ 9 33]
  [13 37]]

 [[17 41]
  [21 45]]]

[[[ 2 26]
  [ 6 30]]

 [[10 34]
  [14 38]]

 [[18 42]
  [22 46]]]

[[[ 3 27]
  [ 7 31]]

 [[11 35]
  [15 39]]

 [[19 43]
  [23 47]]]

#####
# Data transposed : T[p][t] (in table format)
#####
[[ 0 24 4 28]
 [ 8 32 12 36]
 [16 40 20 44]
 [ 1 25 5 29]
 [ 9 33 13 37]
 [17 41 21 45]
 [ 2 26 6 30]
 [10 34 14 38]
 [18 42 22 46]
 [ 3 27 7 31]
 [11 35 15 39]
 [19 43 23 47]]
```