Logarithmic Interconnect IP

Authors

I.Loi

Summary

This document describes the functional specification of Logarithmic Interconnect used in the P2012 project. Functional specification, IP configuration and usage will be explained in this document.

TABLE OF CONTENTS

1. Introduction

This document describes the functional specifications of the Logarithmic Interconnect, which is part of the P2012 project. It also describes the configuration of the IP, and gives some guidelines on the correct instantiation in a design.

The Logarithmic Interconnect implements a parametric Multi-Core Crossbar to connect multiple Master devices to multiple Slave devices (typically Memory banks. It has been designed to both sustain full bandwidth and to provide minimum latency (one clock cycle). For this reason it is mostly suitable for Tight Coupled Data Memory (TCDM) context (like shared scratchpad), where multiple and equal memory banks, are attached in the slave ports to form the entire TCDM memory space.

By interleaving the address with fine granularity (typically word level interleaving), it is possible to spread transaction among all the available memory banks (SRAMs), therefore hitting the access contentions in case of multiple accesses. Using a number of memory banks that is the double of the number of master ports, is enough to drastically reduce access collisions and to show to the Master an available bandwidth which is close to 100% of the process throughput, like each processor having a private memory; but the whole TCDM is shared so the benefit is to avoid data replication, and to simplify and build a more efficient mechanism to provide a multicore synchronization.

To keep the Interconnect fast and power efficient, a logarithmic approach has been adopted to build the IP, where arbitration primitives are used to build a sort of hybrid binary tree.

In conclusion, this IP offers an efficient and flexible solution to build a homogeneous or heterogeneous multi-core interconnect that can be instantiate as a soft IP (though coreKit tools) or directly with pure RTL.

This IP not only provides the RTL of the Log Interconnect, but also a preliminary test environment to verify functionality, a set of script and guidelines to implement the block (synthesis and place and route).

2. Features and Benefits

- Fully Synchronous design
- Configurable number of master and slave ports
- Configurable Data Bus width and Address Bus width
- Fine grain or Corse grain or user defined address interleaving
- Independent and decoupled Request and Response channels
- Request/grant based flow control
- Single clock cycle Latency network transversal
- Test and set supported
- Distributed Round robin arbitration
- High Priority and Low Priority channels
- Deterministic Load and Store latency
- Fully Synthesizable

3. IP Description

The Logarithmic Combinational Network (see Figure 3-1) implements a parametric Multi-Core X-Bar to connect Master devices to Slave devices (typically Memory banks). The figure shows X + Y master ports and Z slave ports. The X ports have higher priority with respect to the Y ports, and among both X and Y ports a round robin arbitration scheme is adopted. Between X and Y ports a fixed priority scheme is assumed. For this reason bandwidth hungry Processing Elements are directly connected to the X ports, while other peripherals that does not often request for memory accesses are connected to Y ports. The Z ports are used to attach slave devices, or more in general memories.

The restriction that is mandatory for the current configuration is that each memory cut which is connected to these ports must be exactly the same, or more in detail, it should be provide same number of memory cells and same datawidth.

As introduced in the previous introductive section, the network is capable to serve master transaction in a single cycle, and in case of load, the interconnect returns the read data the clock cycle after the read is asserted and granted. This feature is very important when the latency plays a crucial role, and it allows using this IP in a huge variety of cases.

3.1. Flow Control

The flow control used in this IP is a credit base protocol. It is based on two signals, namely a ***request*** and a ***grant***. The master that wants to initialize a transfer (either a load or a store) must assert a request (set to 1) and wait for a grant. The logarithmic interconnect computes in the grant in the same clock cycle, and resolves each possible conflict. Each master that asserts a request must trigger the grant on its input port to figure out if the transaction is gone or is stalled. If the master triggers a low grant (0), it means that the transaction has been not accepted, so it must keep the request and data signal on its output ports. In case it triggers a high grant (1) it must deassert the request, or can inject a new transaction. In summary, there are two possible states for the master that makes request: GONE or STALLED. Since the system is fully synchronous, the trigger events are based on the rising edge of the clock.

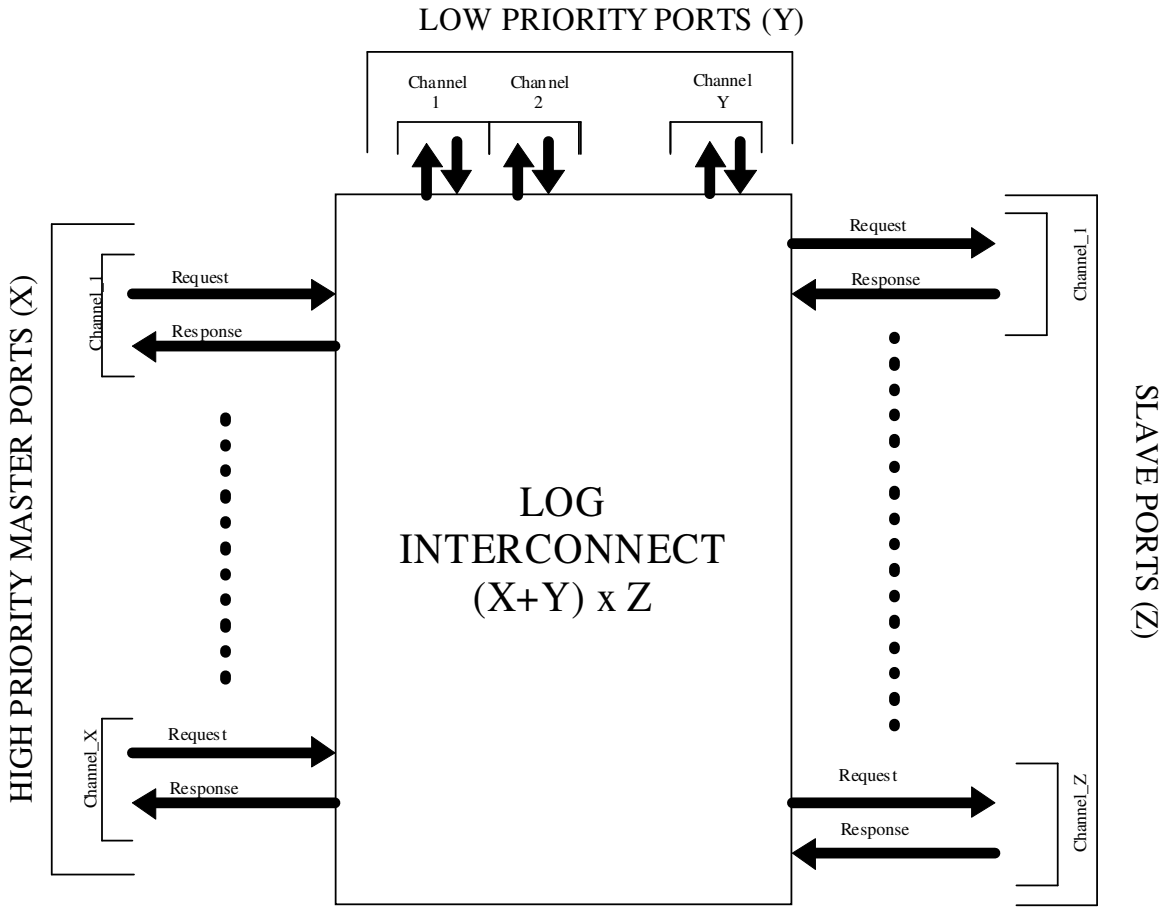

Figure 3-1: Overview for a (X+Y) x Z logarithmic interconnect

3.2. Communication Protocol

This subsection provides a description of the signals that compose both master and slave ports. In order to avoid bottleneck, this set of signals

have bee accurately selected in order reduce design complexity, therefore reducing overall latency and increasing the power efficiency. The Table 3-1 resumes the whole set of signals for both master side and slave side. As we can see in the table, signals are grouped in Master and salve sections, and inside we find response and request groups. This separation is made because the IP is designed with request channel and response channel completely decoupled.

	PIN NAME	WIDTH	DIRECTION	DESCRIPTION
	clk	1	input	System Clock
	rst_n	1	input (active low)	System Asynchronous reset
Master Side				
REQUEST	data_wdata_i	DATA_WIDTH * (X+Y)	input	Data Bus for Write
	data_addr_i	ADDR_WIDTH * (X+Y)	input	Address Bus
	data_type_i	(X+Y)	input (active low)	LOAD Flag (if high, Request is a load, if Zero is a store)
	data_req_i	(X+Y)	input (active high)	request Flag (Valid Request)
	data_be_i	BYTE_ENABLE_BIT* (X+Y)	input (check the memory protocol, the log interconnect do not check this signal)	Data byte enable: eg 0000-> 32bit 0011 or 1100 ->16 bit 0111, … , 1110 ->8bit
	data_gnt_o	(X+Y)	output (active high)	Grant signal (memory is busy or internal contentions). Master must wait till grant is high before injecting next command.
RESPONSE	data_r_rdata_o	DATA_WIDTH * (X+Y)	output	Data Bus for Read
	data_r_valid_o	(X+Y)	output (active high)	Valid response for Load or Store

				If comes after a load, it indicated that the rdata is valid. After a store indicated that It command has been accepted
Slave Side				
REQUEST	data_req_o	Z	output (active high)	request Flag (Valid Request)
	data_add_o	Z * ADDR_MEM_WIDTH	output	Address Bus
	data_type_o	Z	output	Load store Flag (if high, Request is LOAD, if zero a store)
	data_wdata_o	DATA_WIDTH * Z	output	Data Bus for Write
	data_be_o__i	BYTE_ENABLE_BIT * (X+Y)	output (check the memory protocol, the log interconnect do not check this signal)	Data byte enable (STORE): 0000-> 32bit 0011 or 1100 ->16 bit 0111, … , 1110 ->8bit
RESPONSE	data_r_rdata_i	DATA_WIDTH * Z	input	DATA bus for read data.

Table 3-1: List of the IP IO ports

Let's consider now the master side. The request group is composed by flow control and datapath signals. The former are the request and the grant, the former are address, write data, byte enable and type. We selected only these four signals (datapath) to reduce the design complexity, because one of the goals of this IP is to get the minimum latency and maximum power efficiency.

The response group is composed by only two signals: the data_r_valid is the control, and the data_r_rdata is the datapah. As we can see, there is no grant coming from the processor to the response ports, because we assume that master can always accept read data, and this happens always the cycle after the logarithmic interconnect accepts a LOAD.

In the salve side, we assume a set of signal that better matches a pure memory model, so there are no control signals, except the request that can be used as chip select of memories. The datapath signals are the same of the master side.

3.3. IP Configuration

The IP can be reconfigured changing the parameters that are listed in the parameter.v file in the RTL folder. The Table 3-2 resumes the main parameters that can be changed by the user. A brief description is also provided. Some name will be changed in the next release.

Parameter Name	DESCRIPTION	Allowed range
N_MASTER	Number of High priority master channels	1,2,4,8,16 ….
N_DMA	Number of Low priority master channels	0,1,2,4,8 …
N_SLAVE	Number of memory cuts (Equal size)	1,2,4,8,16,32,……
DATA_WIDTH	Data width	32,64,128….
ADDR_WIDTH	Address Width	32 (to be check other config)
WORD_INTERLEAVING	Enable word interleaving (DATA_WIDTH bits)	TRUE of FALSE
MEM_CUT_SIZE	Memory cut size in KB	1,2,4,8,16 ……
TEST_SET_BIT	address bit used to Test and set purpose	ADDR_OFFSET < x < ADDR_WIDTH-1

Table 3-2: Parameter List Description

Having a system with several degrees of freedom like this IP involves checking the functionality for every set of parameters defined. For this reason all the possible configuration have not been yet checked. In the next release all the possible legality check will be added with a sort of debug message generating at compile time to avoid the instantiation of illegal parameters. The coreKit development model that is on going has already some of these checks.

3.4. Cell List

The design is coded using the SystemVerilog Language. The main advantage using this language in place of VHDL or Verilog is to unify the verification and the design environment in a single language. By the way, the IP is composed by several hierarchical modules, used to build a homogeneous structure. The Table 3-3 resumes the list of cells used in the IP. A brief description is also provided.

CELL NAME	Language	Comments
XBAR_TCDM	SystemVerilog	Top level of the log interconnect
ResponeBlock_2CH	SystemVerilog	Response Block (when 2 Channels are used)
RequestBlock_2CH	SystemVerilog	Request Block (when 2 Channels are used)
ResponeBlock_1CH	SystemVerilog	Response Block (when1 Channel is used)
RequestBlock_1CH	SystemVerilog	Request Block (when 1 Channels is used)
ResponseTree	SystemVerilog	Collects response from memories
ArbitrationTree	SystemVerilog	collects request from processor
AddressDecoder_Req	SystemVerilog	Decode request address
AddressDecorer_Resp	SystemVerilog	Decode response ID
MUX2_REQ	SystemVerilog	Muxes 2 Channels
TestAndSet	SystemVerilog	Implements atomic operation *test and set*

Table 3-3: List of the cells used in the IP

4. Detailed IP Description

This section provides a detailed IP description, block by block. The design will be explored with a top down approach, first is described the top level (XBAR_TCDM) and the description will be focused on sub modules, until the leaf modules will be reached. For each module, a block diagram is provided and some considerations are derived from it.

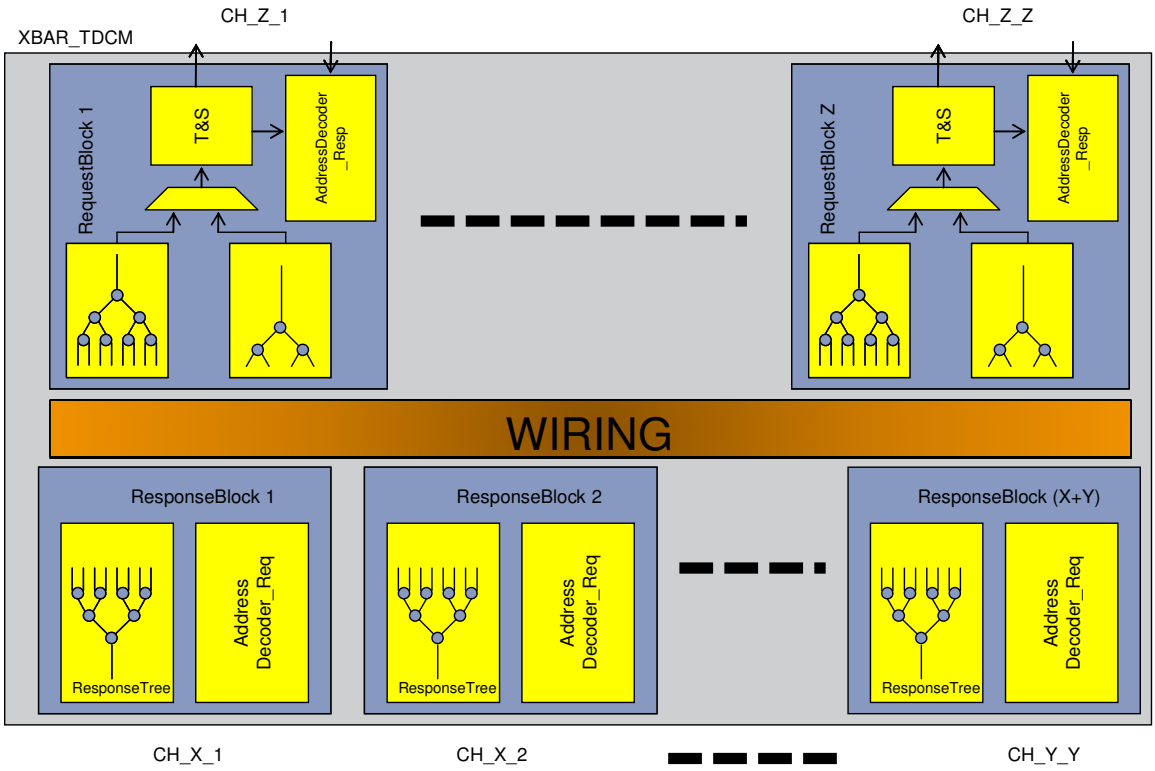

Figure 4-1: Block diagram of the top level XBAR_TCDM

4.1. XBAR_TCDM

The XBAR_TCDM is the top level of the logarithmic interconnect. It exposes the IO pins listed in Table 3-1, and it is composed by response and routing block, as shown in Figure 4-1. Each master port is connected to an instance of the ResponseBlock cell, while each slave is connected to an instance of the RequestBlock cell.

In total there are X+Y ResponseBlocks and Z RequestBlock where X stands for number of High priority masters, Y means for the number of low priority masters, and Z stands for the number of slaves.

Depending on the value of Y we can have a single channels log interconnect (Y = 0) or a dual channel Log interconnect. In case of dual channel the request block include some logic to handle the second channel, while in case of single channel (Y = 0) the RequestBlock is entirely made by the X_Channel arbitration tree and the Test and set interface. For clarity, if Y = 0, the request block cell is called RequestBlock_1CH, while in case of dual channel it is called RequestBlock_2CH.

The master request datapath signals (address, write data, type and byte enable) are directly connected to request block, while flow control signals are filtered based on address and traffic collisions. The response datapath signal (read data) is directly connected to ResponseBlock, while the control signal (response request) is filtered base on the sender ID.

For this reason, the master and slave datapath pins have respectively a fan out of Z and X+Y. Control signals have always fanout of 1.

The filtering is done inside the address decoders in Request and Response Block.

4.2. RequestBlock_2CH

The RequestBlock_2CH cell, depicted in **Errore. L'origine riferimento non è stata trovata.**, is in charge to collect incoming requests to a particular slave, resolve any conflict in a combinational way and prepare the response flow control to be compatible with the response channel. It is composed by two arbitration trees, one for the High Priority Channel, and one for the Low Priority Channel, a multiplexer to muxes the two flows, and a test and set interface (test and set is directly connected to slave).

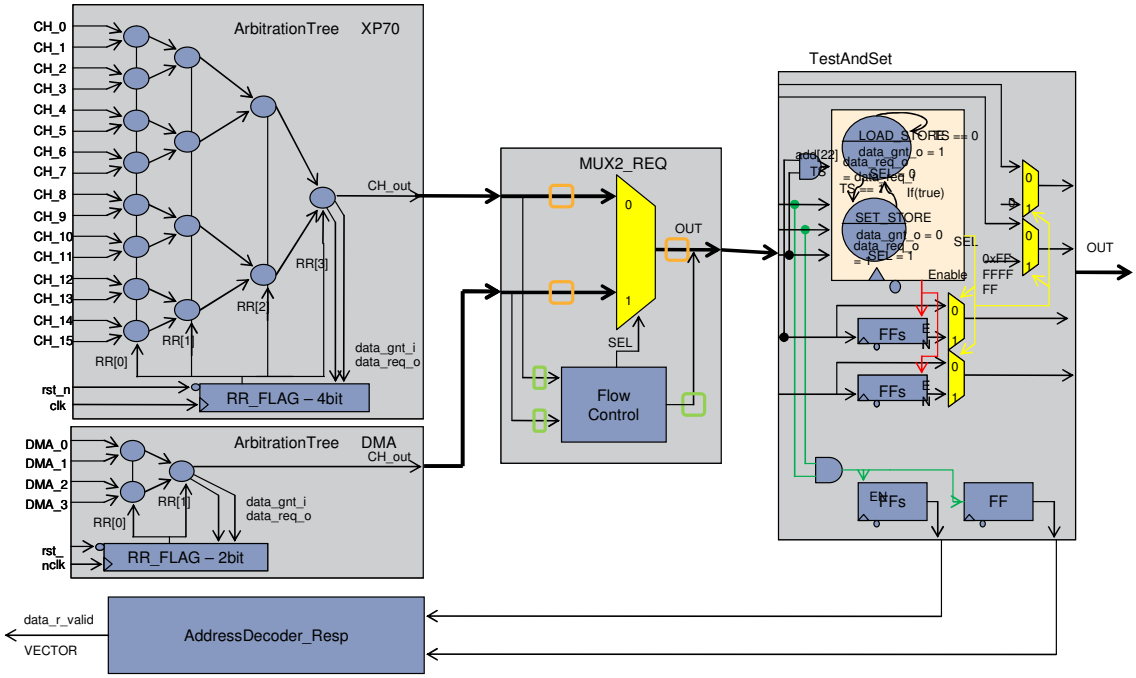

Figure 4-2: Block description of the Request Block _2CH cell for X=16, Y=4

The RequestBlock_2CH in **Errore. L'origine riferimento non è stata trovata.** is a particular case where X is 16 and Y is 4 and for sake simplicity, we named the X port as XP70 ports (processor) and Y ports to DMAs. Since X and Y are not equal, the two arbitration trees are different (same cell but different input parameter). The arbitration trees resolve any conflicts, granting only one request per clock cycle per slave. The second

stage is made by a multiplexer with embed a fixed priority arbiter inside. The two winner requests at the output of the two arbitration trees are connected to a Multiplexer (MUX2_REQ), which grants any request of the High priority channel. If any high priority is coming, the MUX2_CH checks for the low priority channel. The output of this block is sent to the test and set interface which implements the atomic operation composed mainly used for processor synchronization, and made by load followed by a store.

In addition, the Test and set interface is the common point between request and response channels therefore it is in charge of handling the back-route information: for this purpose the PID is attached on each master transaction, then transported until the test and set, and finally used to as address for delivery the responses.

4.3. RequestChannel_1CH

The RequestBlock_1CH cell, depicted inFigure 4-3, is in charge to collect incoming requests to a particular slave, and resolve any conflict in a combinational way. It is composed by one arbitration tree, (only the High Priority Channel is present) and a test and set interface (test and set is directly connected to slave)

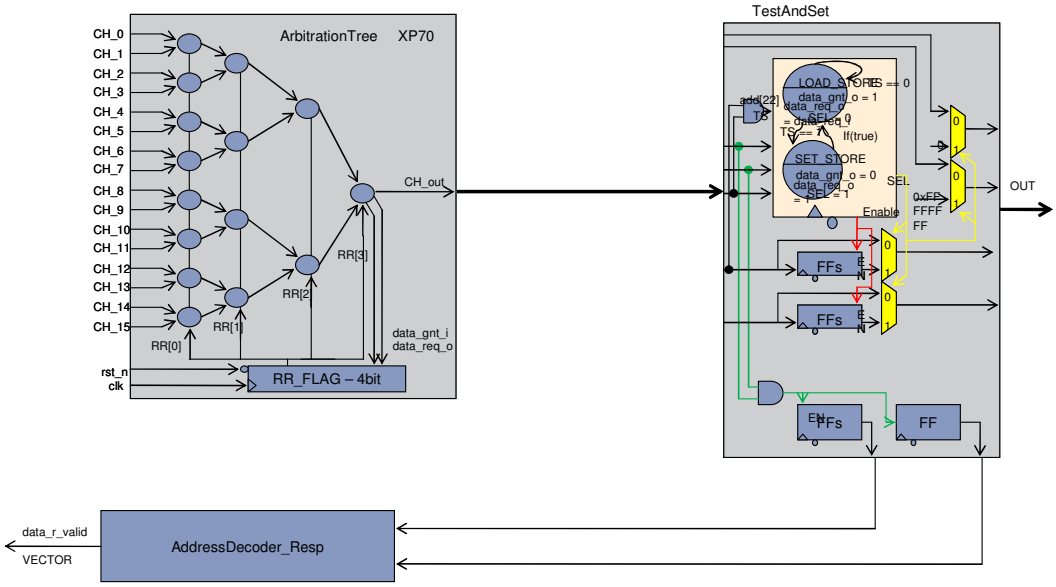

Figure 4-3: Block description of the Request Block _1CH cell for X=16, Y=0

The functionality is exactly the same of the RequestChannel_2CH, and the only difference is that the second arbitration tree and the mux are not needed. For the overview of this block please refer to subsection 4.2.

4.4. Arbitration Tree

This block (depicted in Figure 4-4: Arbitration tree Block diagram) is the heart of the IP, and is in charge of taking as input all the possible requests (X or Y) from any master attached on a particular channel (High Priority or Low Priority), handling the flow control and propagating the winner request to the output port (CH_out in the figure).

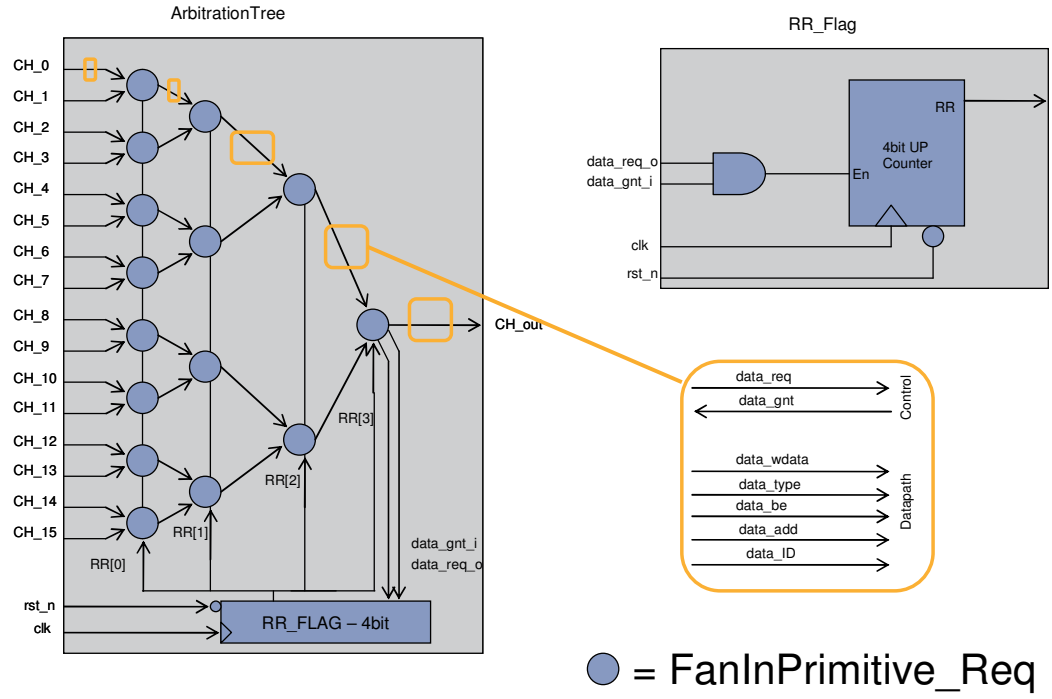

Figure 4-4: Arbitration tree Block diagram

The inputs of this block are the Master outputs (except for flow control and ID signals), and the path from any input to the output is completely combinational. It is built using arbitration primitives called FanInPrimititve_Req which will be discusses in the next subsection. The number of primitives arbitration tree is a function of the number of master attached. For each arbitration tree there are X+1 FanInPrimitive_Req, and in case of double channel (Y≠0) for each slave there are X+Y+2 FanInPrimitive_req and in total, in the entire IP there are Z*(X+Y+2) primitives.

It is important to note that, each RequestBlock (therefore each ArbitrationTree) is dedicated for a specific slave port (which means a range of memory addresses).

The purpose of this block is to propagate to the output port only one request among the X (or Y) possible input requests. The winner port must provide to the respective master a high grant, while the other port must provide a low grant (note that for this low grant is related to this specific slave).

The entire arbitration is resolved in distributed blocks considering the request available in the inputs port, and propagating the winner request to the successive stage (on each stage the number of winner candidates are halved). There are a number of stages that are function of the number of masters attached on the ArbitrationTree.

$$X\_stages = \log_2(X)$$

$$Y\_stages = \log_2(Y)$$

On each master port (datapath group), there is an additional signal that is not provided by the master: the data_ID. This signal is generated in the AddressDecoder_Req, and is a fixed label, different between each master, and is used to calculate the response address (on which master the response must be delivered?). The data ID as well the other datapath signals are just transported, no logical operation are made on this group of signal.

The arbitration, based on the round robin scheme, is taken on each primitive. An N bit counter (where N is the log2(X or Y)) is used to generate and delivery the Round Robin Flag used determine the priority inside each primitive. Therefore each stage receives only one of the N bits generated by the N-bit counter.

4.5. RR_Flag_req

The RR_Flag_req (refer to Figure 4-4) is simply a up-stoppable N-bit counter where the enable event is simply the condition

$$enable = data\_req\_o \,\&\, data\_gnt\_i$$

The Figure 4-5 shows the timing diagram for N = 4. The counter is incremented each clock cycle only when the enable is high. This particular behaviour allows switching the flag only when there is a transmitted command to the slave, therefore better balancing the traffic and ensuring a uniform bandwidth distribution across the masters.

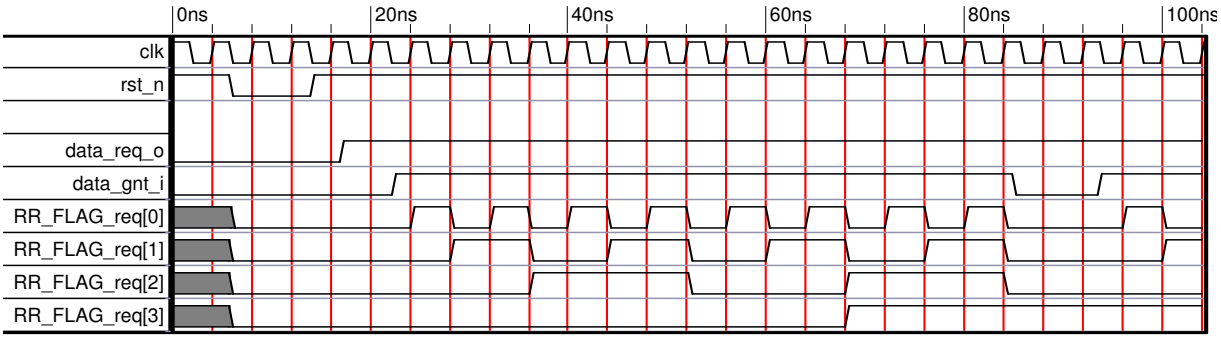

Figure 4-5: Timing diagram for the RR_Flag_req (stoppable up-counter) cell

4.5.1. Arbitration details

To better understand the arbitration scheme let's consider the example depicted in Figure 4-6. Let's suppose that there are only 4 masters, so 2 level of arbitration primitives are needed. A 2 bit counter is enough to generate the proper RR_Flag_req.

At Cycle one let's assume that the RR flag is 10, so the max priority is on channel 1. If there is no request on channel 1, the second priority is sets on channel 0. Again, if there are no request on channel 1 and 0, the third priority then is on channel 3. If none of these channels are making request, the fourth priority is set to channel 2. Max to min priorities are [1,0,3,2]

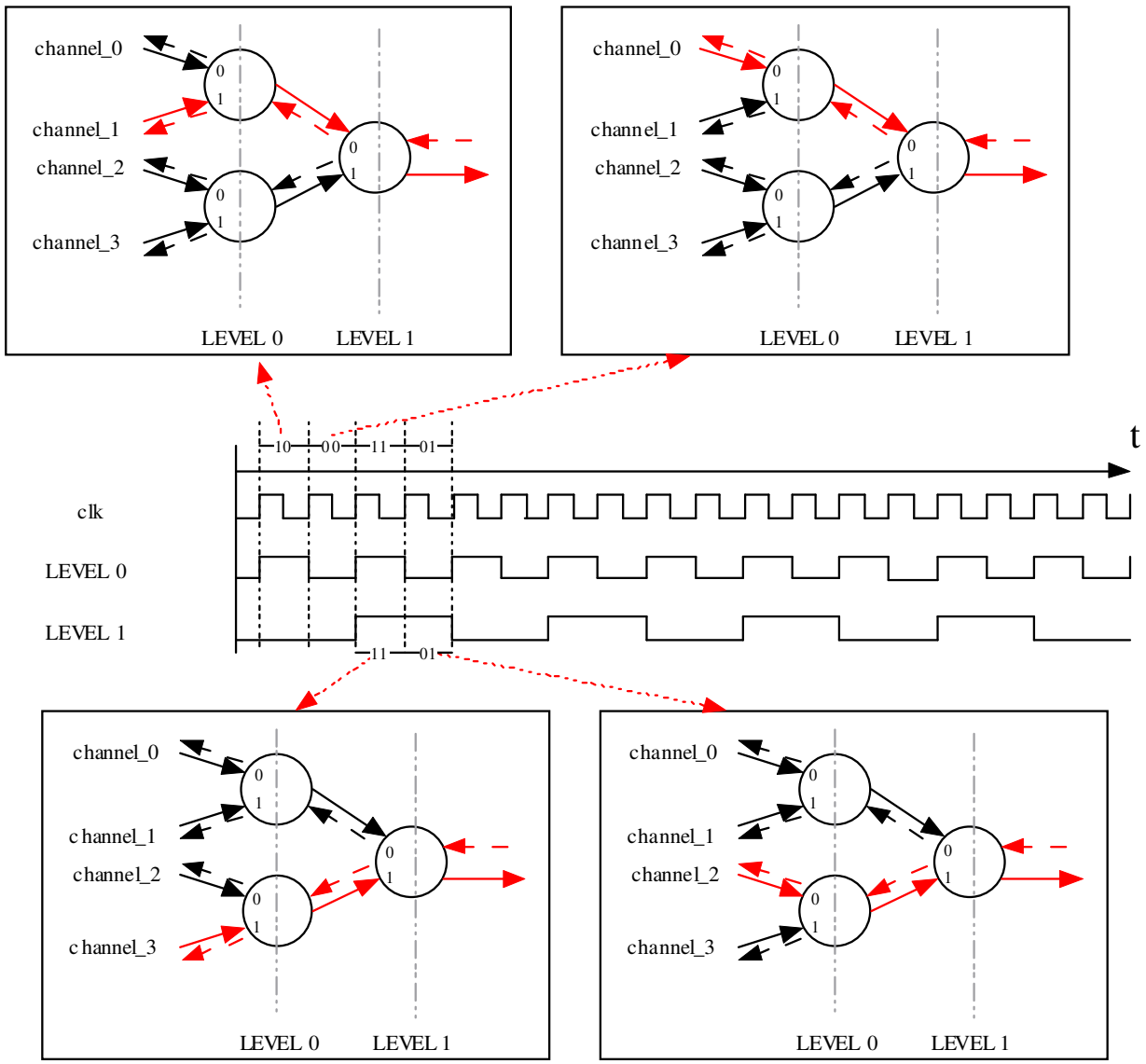

Figure 4-6: Arbitration details in case of X = 4 (enable = 1)

Every clock cycle where a valid request is granted, the counter is incremented. When the counter reaches the max values, it starts again from zero, in a cyclic manner.

In summary, the arbitration priorities are:

At cycle 1 (RR=10) the max priority is set to channel 1; max to min priorities are [1,0,3,2]

At cycle 2 (RR=00) the max priority is set to channel 0; max to min priorities are [0,1,2,3]

At cycle 3 (RR=11) the max priority is set to channel 3; max to min priorities are [3,2,1,0]

At cycle 4 (RR=01) the max priority is set to channel 2; max to min priorities are [2,3,0,1]

If we consider a generic case with N=16 different input ports, the priorities can be summarized as shown in Table 4-1. In this example, given a specific RR flag, the table resumes the max to min priority, for the relative arbitration tree. As can be noticed, for each particular row there is only one occurrence of each a master port, so this means that the available bandwidth is balanced among the available master attached on it

RR_FLAG	***PRIORITY (CH) → MAX to MIN***															
L0,L1,L2,L3	MAX	→	→	→	→	→	→	→	→	→	→	→	→	→	→	MIN
0000	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1000	1	0	3	2	5	4	7	6	9	8	11	10	13	12	15	14
0100	2	3	0	1	6	7	4	5	10	11	8	9	14	15	12	13
1100	3	2	1	0	7	6	5	4	11	10	9	8	15	14	13	12
0010	4	5	6	7	0	1	2	3	12	13	14	15	8	9	10	11
1010	5	4	7	6	1	0	3	2	13	12	15	14	9	8	11	10
0110	6	7	4	5	2	3	0	1	14	15	12	13	10	11	8	9
1110	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
0001	8	9	10	11	12	13	14	15	0	1	2	3	4	5	6	7
1001	9	8	11	10	13	12	15	14	1	0	3	2	5	4	7	6
0101	10	11	8	9	14	15	12	13	2	3	0	1	6	7	4	5
1101	11	10	9	8	15	14	13	12	3	2	1	0	7	6	5	4
0011	12	13	14	15	8	9	10	11	4	5	6	7	0	1	2	3
1011	13	12	15	14	9	8	11	10	5	4	7	6	1	0	3	2
0111	14	15	12	13	10	11	8	9	6	7	4	5	2	3	0	1
1111	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Table 4-1: Priority table for X (or Y) = 16

4.6. FanInPrimitive_Req

The FanInPrimitive_Req (shown in Figure 4-7) is the basic building block used to build the Arbitration Tree. It is composed by a multiplexer driven by a control block that handles also the output request and grants. The two ways of the multiplexer comprise a set of the datapath signal made by address, write data, byte enable, type and ID. Depending of the value of the SEL signal, the mux connect the output to one of the two inputs. At the same time the control logic assert a low grant for the looser channel, and for the winner channel it assign the input grant that comes from the next primitive stages. This is fundamental to avoid more than one grant in the same cycle related to the same slave. **Only one master is allowed to access one of the available slaves in the same clock cycle!!!**

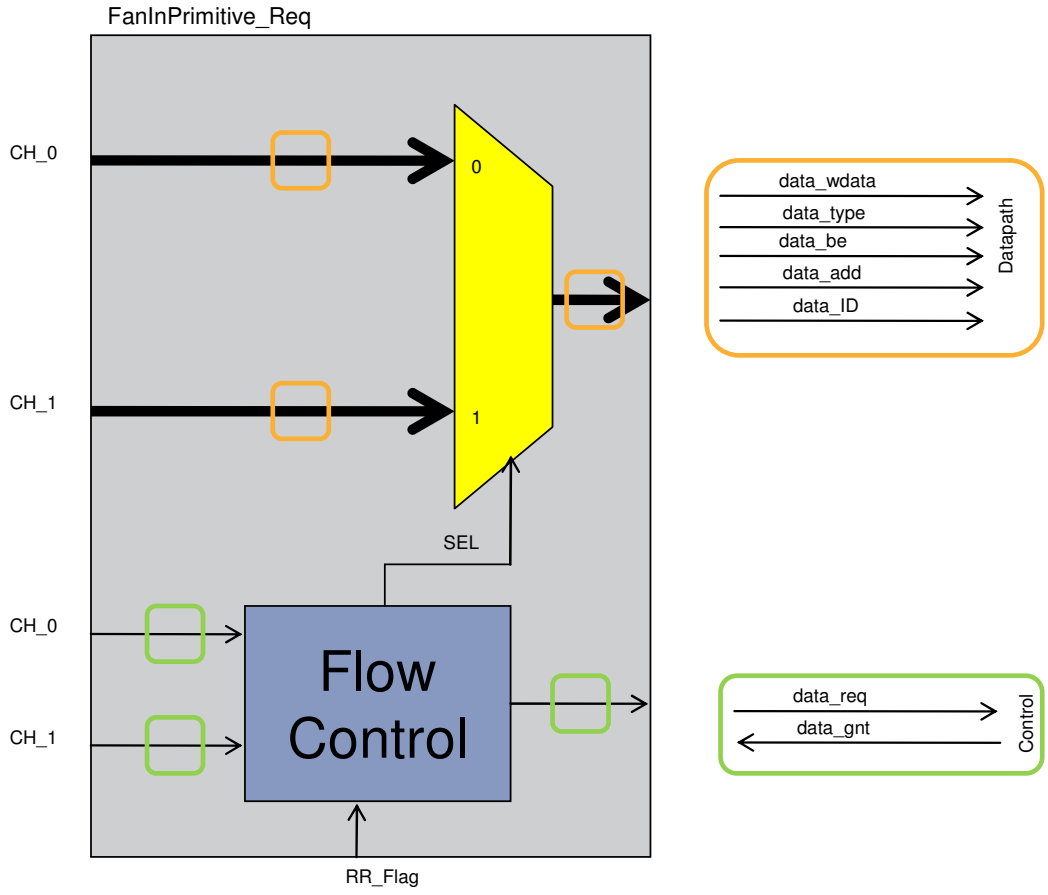

Figure 4-7: Detail of the FanInPrimitive_Req

The output request is computed as the or of the input request

$$data\_req\_o = data\_req0\_i \mid data\_req1\_i$$

While the output grants are calculated with more complex logic conditions

$$data\_gnt0\_o = [(\text{data\_req0\_i} \,\&\, \sim \text{data\_req1\_i}) \mid (\text{data\_req0\_i} \,\&\, \sim \text{RR\_FLAG})] \,\&\, \text{data\_gnt\_i}$$

$$data\_gnt1\_o = [(\sim data\_req0\_i \,\&\, data\_req1\_i) \mid (data\_req1\_i \,\&\, RR\_FLAG)] \,\&\, data\_gnt\_i$$

$$SEL = \sim data\_req0\_i \mid (RR\_FLAG \,\&\, data\_req1\_i)$$

Table 4-2 shows the truth table for the FanInPrimitive_Req. The first the column are the flow control inputs of the block, while the last four are the generated outputs. The logic expressions reported above are simply derived with hand made reduction (using the Karnaugh map method).

data_req0_i	data_req1_i	RR_Flag	data_gnt0_o	data_gnt1_o	SEL	data_req_o
0	0	0	0	0	0 (dc)	0
0	0	1	0	0	0 (dc)	0
0	1	0	0	data_gnt_i	1	1
0	1	1	0	data_gnt_i	1	1
1	0	0	data_gnt_i	0	0	1
1	0	1	data_gnt_i	0	0	1
1	1	0	data_gnt_i	0	0	1
1	1	1	0	data_gnt_i	1	1

Table 4-2: FanInPrimitive_Req Truth table

4.7. MUX2_REQ

The MUX2_REQ (Figure 4-8) is a block used inside the RequestBlock only in the configuration where Y = 0. It includes a two channel multiplexer driven by a logic block that embeds a fixed priority encoder. The maximum priority is assigned to the first input (CH_0), so every request in this channel is forwarded to the output. In case there is no request on the first channel, the second channel is considered.

Table 4-3 depicts the truth table for this Block. The generated control signals (data_req_out, data_gnt0_o, data_gnt1_o and SEL) are resumed in the Figure 4-8, and they are derived by hand reduction of the truth table using the Karnaugh map reduction method. Similarly to the FanInRequest, the loser channel gets a low grant, while for the winner, the grant is equal to the input grant (data_gnt_i), this because the Test and set interface may be busy during the test and set operation (test and set consumes two cycles).

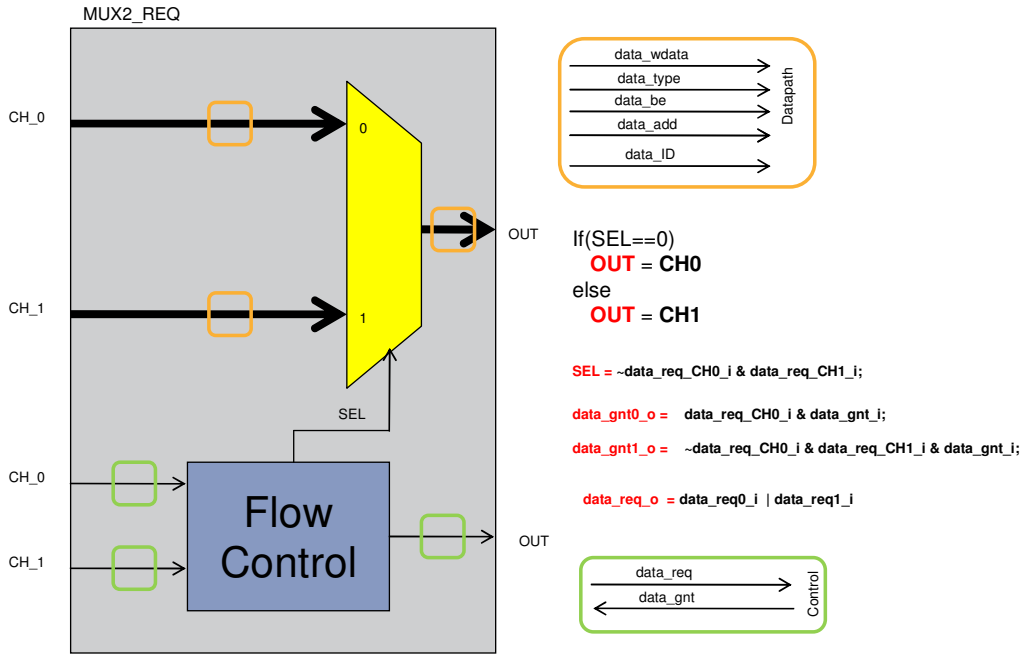

Figure 4-8: Block Diagram for the MUX2_REQ

data_req0_o	data_req1_o	data_gnt0_o	data_gnt1_o	SEL	data_req_o
0	0	0	0	0 (dc)	0
0	0	0	0	0 (dc)	0
0	1	0	data_gnt_i	1	1
0	1	0	data_gnt_i	1	1
1	0	data_gnt_i	0	0	1
1	0	data_gnt_i	0	0	1
1	1	data_gnt_i	0	0	1
1	1	data_gnt_i	0	0	1

Table 4-3: MUX2_REQ Truth table

4.8. TestAndSet

The TestAndSet interface is the block that:

- ✓ Handles the test and set atomic operation,
- ✓ Handles the flow control signals, leaving the slave transparent on it
- ✓ Handles the ID used to back-route the responses

A graphical representation is given in Figure 4-9. It is composed by a small Finite state machine that handles both request datapath and flow control signals, and a small logic to handle the valid response signal plus the response address used to back-route the read data and valid response, or simply in case of STORE a valid response, which means that the command has been executed.

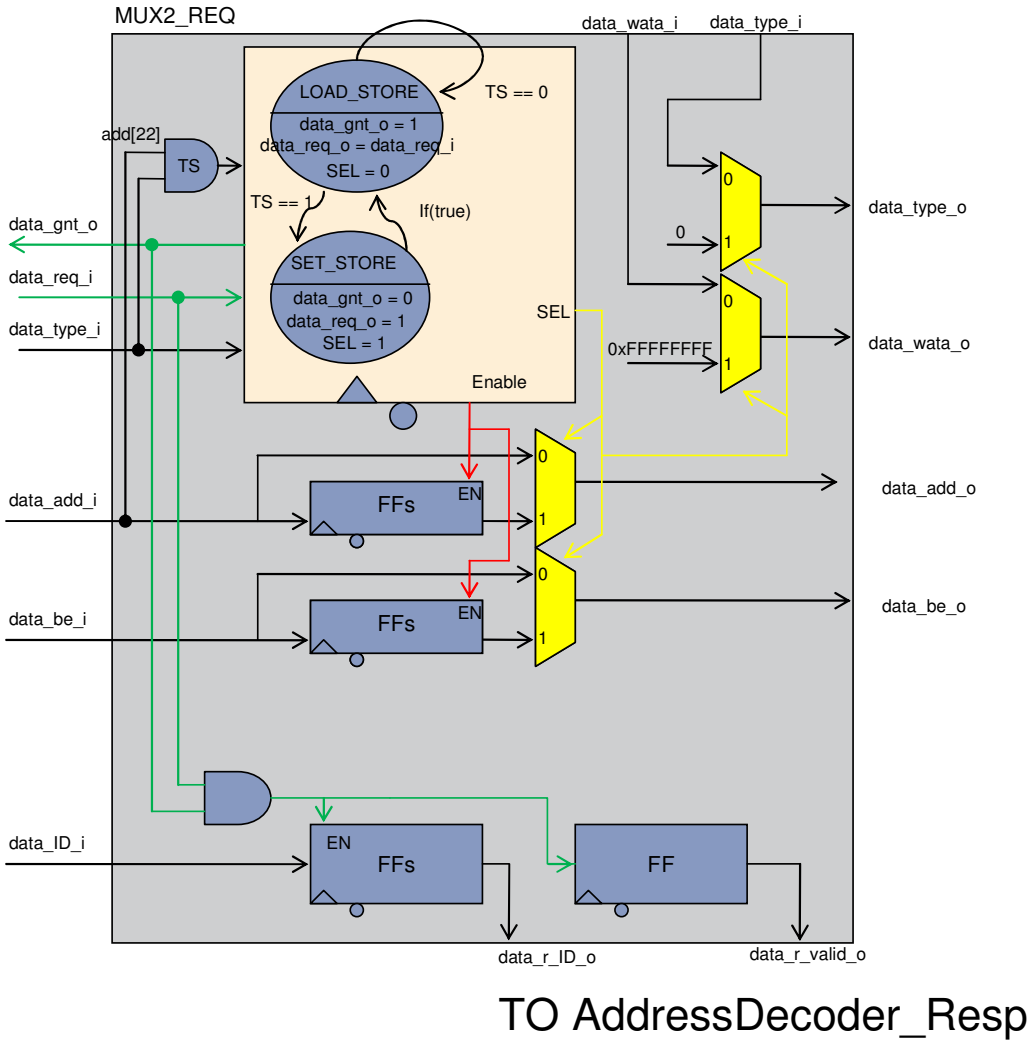

Figure 4-9: Block Diagram for the TestAndSet cell

The upper part in Figure 4-9 has been implemented entirely with a finite state machine where the details of this FSM are depicted in Figure 4-10. The Multiplexer shown in the Figure 4-9 are implicitly inferred in the FSM assigning the datapath outputs directly to the datapath inputs, or to the sampled inputs (used during the SET stage of the test and set). The data_gnt_o is generated from the state (Moore ouptut) while the request_out depends both on the current state and the input (Mealy output).

In Figure 4-10 the TestAndSet signal is internally generated starting from the data_add_i and the data_type. The condition to be satisfied in order to start this atomic operation is:

$$(data\_add\_i[TEST\_SET\_BIT] == 1) \,\&\, (data\_type\_i == LOAD) \,\&\, (data\_gnt\_o == 1'b1)$$

Test and set can be served only if the current state is LOAD_STORE

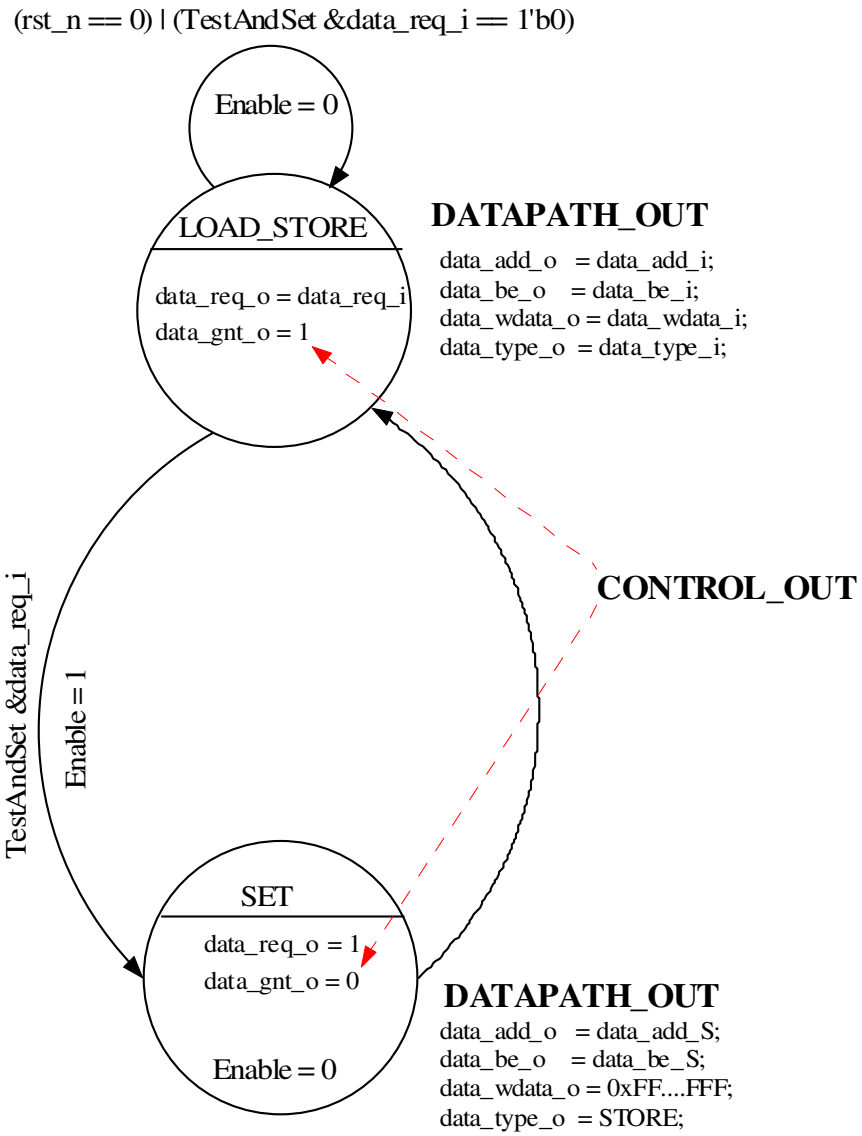

Figure 4-10: Implementation of the TestAndSet FSM

The normal operation is the Load/Store state, where the memory is accessed to perform loads or stores. The test and set is used for

synchronization purpose, more in detail to implements barriers to synchronize processors that works on a parallelized piece of code.

4.8.1. Test-and-set atomic operation

The **test-and-set** is an atomic instruction used to write to a memory location and return its old value as a single atomic (*i.e.* non-interruptible) operation. If multiple processes may access the same memory and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process is done.

When the master asserts a test and set, first the TestAndSet interface returns the value of the pointed memory cell, and then it performs a store writing a set of 1 using the byte enable provided during the fist step of this instruction.

For instance let's consider the following case: At Cycle 0 suppose that

Memory[0x00000000] = 0x0000_0000; //(i.e. suppose DATA_WIDTH = 32)

And processor 0 accesses this memory to perform a test and set with the following data:

data_be_o = LOAD

data_add_o = 0x00010000; // (i.e. suppose TEST_SET_BIT = 16)

data_be_o = 0x0; //(Consider active low mask)

At cycle 0 (consider no request collisions) the processor request reaches the memory asking for a LOAD; so at cycle 1 the memory returns this value:

data_r_rdata = 0x00000000

In the same cycle (cycle 1) the TestAndSet interface performs a write (the read id ongoing). Finally at cycle 2 the content of the memory is:

Memory[0x00000000] = 0xFFFFFFFF;

If another processor (i.e. processor 1) try to read the location Memory[0x00000000] it gets 0xFFFFFFFF, so this means that Processor 0 is currently locking the a process, and other processor must wait until processor 0 release the lock. This happens only when processor 0 performs a STORE on that location writing 0x00000000;

4.9. AddressDecorer_Resp

The AddressDecorer_Resp is a small address decoder that is in charge of generating the one hot valid response array. The inputs of this block are the data_r_ID_o (X+Y bit) and data_r_valid_o (single bit) that come from the TestAndSet interface while the output is data an array (X+Y bit).

To facilitate the generation of this signal (output), the data_r_ID is yet in the one-hot form, so the output is obtained simply with few and gates as shown in Figure 4-11

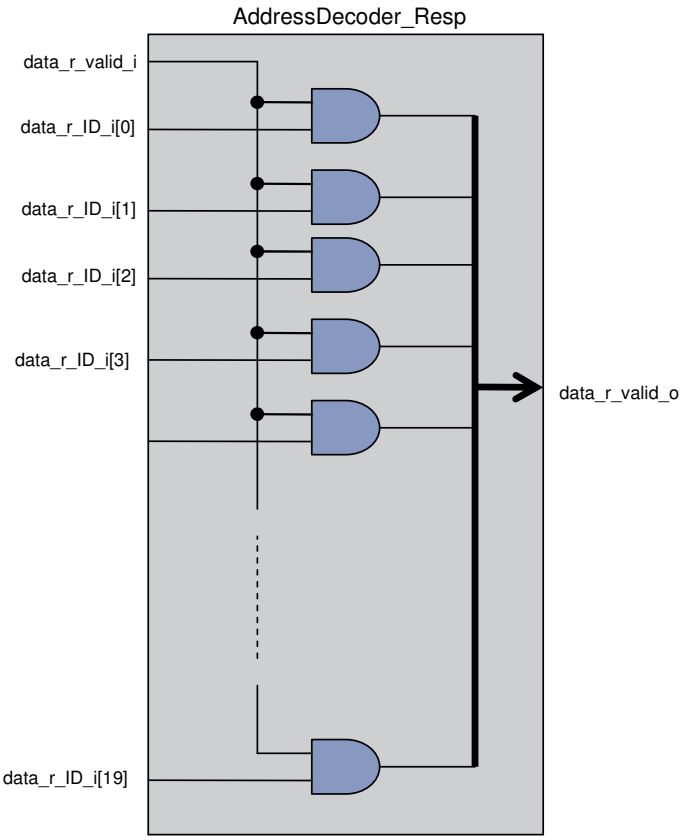

Figure 4-11: Implementation of the AddressDecorer_Resp cell

The one-hot implementation despite infers more resources than the binary version, simplify the decoding, achieving more performance in terms of less latency. Since the combinational logarithmic interconnect may be a critical component, it is fundamental to optimize by hand all the possible paths that may fall in the critical path.

The AddressDecorer_Resp was the last cell that composes the RequestBlock. In the next subsection the ResponseBlock sub modules will be analyzed.

4.10. ResponseBlock

The ResponseBlock cell, depicted in Figure 4-12 for the configuration X=16, Y=4 and Z=32, is in charge to collect incoming response from all the slaves and route them to the right master, and finally generating the proper request flow control signals (data_req, data_ID and multiplex the

grant array) to be dispatched to the several RequestBlock available in the IP.

It is composed by a single ResponseTree and an AddressDecoder_Req. Each ResponseTree port is composed by two signals, respectively the data_r_rdata and the data_r_valid. The former is directly driven by the output pin of the slave (read data of the memory), while the second comes out from the AddressDecoder_Resp.

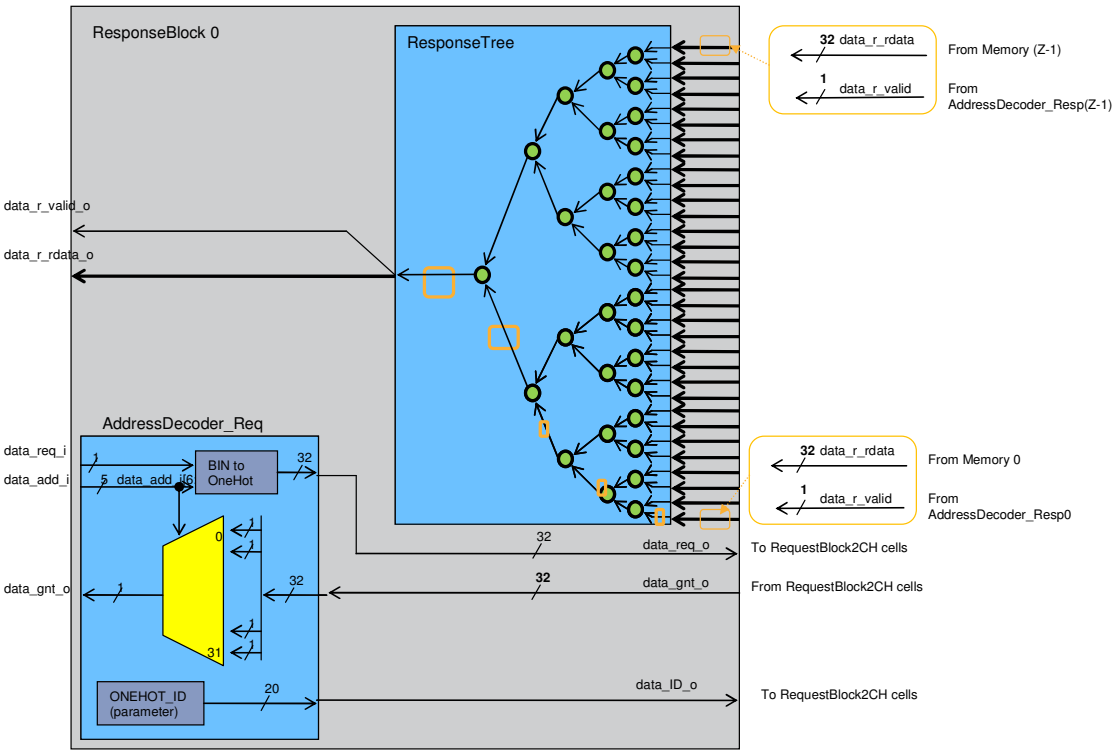

Figure 4-12: Block diagram of the ResponseBlock cell

There are many inputs as many slaves, so to better understand the wiring of these two inputs, let's consider the Figure 4-13. Each data_r_rdata output of the slaves is sent to a ResponseTree port. For istance, data_r_rdata from slave[0] is sent to ports[0] of each ResponseTree. data_r_rdata from slave[1] is sent to ports[1] of each ResponseTree and so on up to the last Slave. Regards the data_r_valid, if we consider the slave[0], the single bit data_r_valid generated by the TestAndSet is converted in one-hot format, so valid response is now an array made by X+Y bits. In Figure 4-13 we suppose X=16, Y=4 and Z =32, so the available 20 bits for the data_r_valid are sent in this order:

// Wiring for Slave[**0**]

TestAndSet[0].data_r_valid[0] → ResponseTree[0]. port[0].data_r_valid

TestAndSet[0].data_r_valid[1] → ResponseTree[1]. port[0].data_r_valid

....

TestAndSet[0].data_r_valid[19] → ResponseTree[1].port[0].data_r_valid

More in general the scheme to route signals is the following (Slave is denoted with ***i***)

$$\begin{cases} TestAndSet[i].data\_r\_valid[j] \Rightarrow \mathrm{Re}\,sponseTree[j].port[i].data\_r\_valid \\ i = 0 \rightarrow Z-1 \\ j = 0 \rightarrow (X+Y)-1 \end{cases}$$

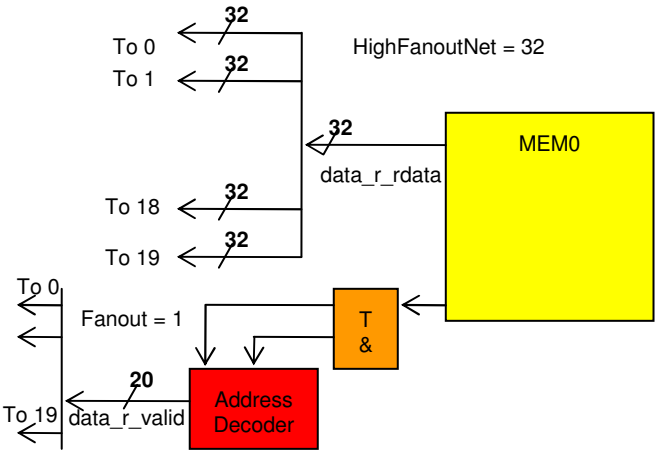

Figure 4-13: Detail of the global wiring for the response channels

In the next subsection is described the ResponseTree then the AddressDecoder_Req.

4.11. ResponseTree

The ResponseTree cell (shown in Figure 4-14 for Z = 32) is in charge of collecting all the possible responses that have for destination the master where this ResponseTree is attached (there are X+Y ResponseTrees in total) an route the request and read data up to the master side (left side). Since the Load and Store latency is deterministic (data_r_valid and related data_r_data come always the cycle after the request is grantes), there is no possibility that two or more responses have as destination the same master. In a nutshell, ***this interconnect does not allow that two responses arrive at the same time to the same destination*** (master), ***therefore there is no need of arbitration blocks, but only simply decoders***.

The ResponseTree is built as the ArbitrationTree, with the main differences as listed below:

- ✓ The number of inputs ports is equal to Z (number of slaves)
- ✓ The number of stage is equal to $\log_2(Z)$
- ✓ There is no arbitration, but only simpler decoders (no RR_Flag_Req)
- ✓ The primitives that composes the tree are called FanInPrimitive_Resp

The last bullet point means that the entire flow control is based on requests (grant signals are not present in this tree).

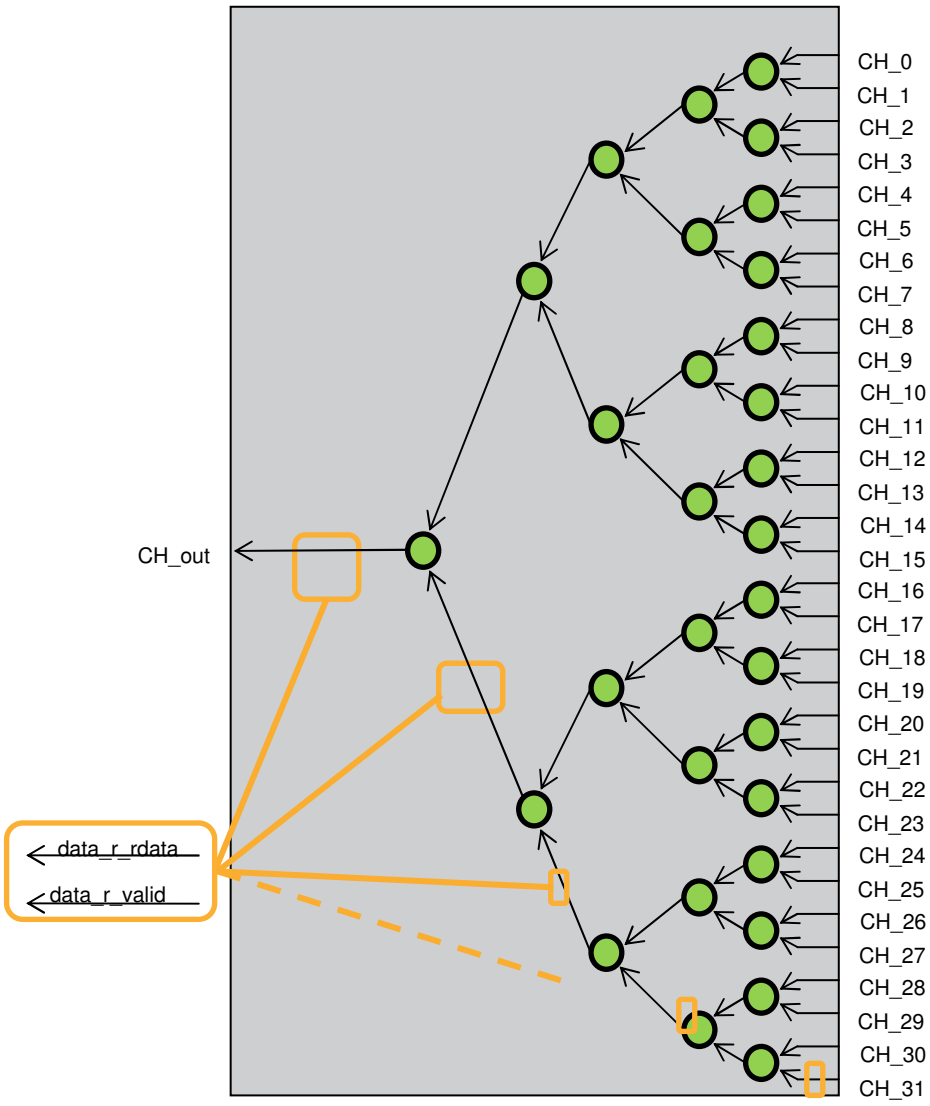

Figure 4-14: Detail of the ResponseTree for Z = 32

As listed before the ResponseTree in built as binary tree using the FanInPrimitive_Resp (Green circles in the Figure). At each stage the number of possible requests is halved and (with respect to the figure), after 5 stages (in general z=log2(Z)) one of the possible request (requests

are mutually exclusive) reaches the master side. Finally the CH_out port (data_r_rdata_o and data_r_valid_o) is directly connected to the master.

4.12. FanInPrimitive_Resp

The FanInPrimitive_Resp (shown inFigure 4-15) is the basic building block used to build the ResponseTree cell. It is composed by a multiplexer driven by the data_r_valid1_i. The two ways of the multiplexer carry the data_r_rdata (read data).

Since there are no cases of request collision, no arbiter is implemented, ad the only logic that we need to route the read_data is to enable one of the two multiplexer inputs accordingly to the relative response request (data_r_valid0_i or data_r_valid1_i). Since requests are mutually exclusive, the simplest and most efficient solution hadle this problem is to use one of the two requests as selector for the multiplexer. The output request is simply generated putting in or the two input requests.

As we can see the logic is kept as simple as possible to get the maximum performance (minimum latency for the response path).

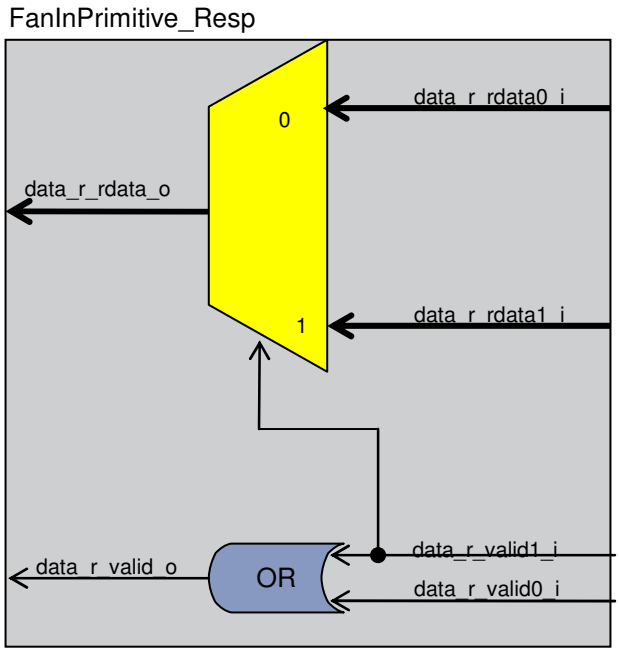

Figure 4-15: FanInPrimititve_Resp Implementation

The Table 4-4 shows the truth table of this cell, and as mentioned before the last row (both request are 1) in not possible so has been treated as don't care during the manual synthesis of the SEL and data_req_o.

data_r_valid1_i	data_r_valid1_i	SEL	data_r_valid_o
0	0	0 (dc)	0
0	1	1	1
1	0	0	1
1	1	impossibile (dc)	impossibile (dc)

Table 4-4: FanInPrimitive_Resp truth table.

4.13. AddressDecoder_Req

The AddressDecoder_Req is in charge of:

- ✓ Generating the one-hot request array (data_req) to be dispatched among the available RequestGroup
- ✓ Multiplex the incoming grant array
- ✓ Generate the one-hot ID

This cell is depicted in Figure 4-16 for a configuration of X=16, Y=4, Z=32, and the three different parts can be easily identified in the figure. The upper part comprises the BIN to ONEHOT encoder which takes z=Log2(Z) bits from the addres, and creates the respective address in the one-hot format.

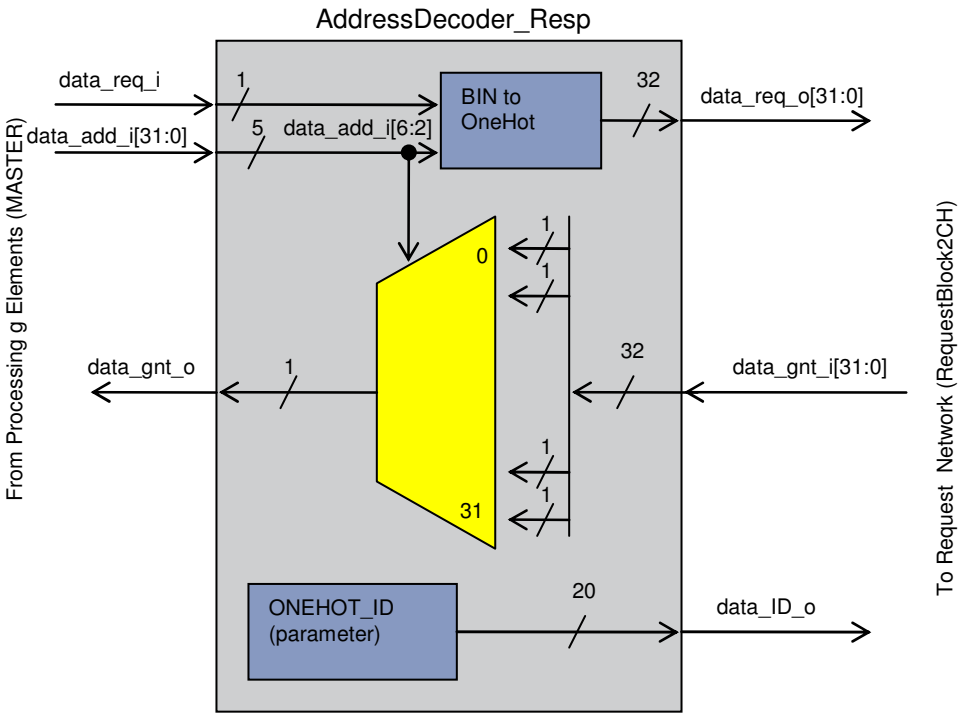

Figure 4-16: Block diagram for the AddressDecoder_Req (X=16, Y=4, Z=32)

The detail of the data_req_o generation is shown in Figure 4-17 for a configuration with Z = 32, ADDR_OFFSET= 2 and Z = 32. If the variable WORD_INTERLEAVING is set to “TRUE” (see parameters.v) then routing bits feeded to the BIN2ONEHOT are taken as follow

$$RoutingAddr = data\_add\_i[\log_2(Z) + ADDR\_OFFEST - 1 : ADDR\_OFFSET]$$

In case of WORD_INTERLEAVING set to “FALSE” then the routing address is calculated as follow:

$$RoutingAddr = data\_add\_i[TEST\_SET\_BIT - 1 : TEST\_SET\_BIT - \log_2(Z) - 1]$$

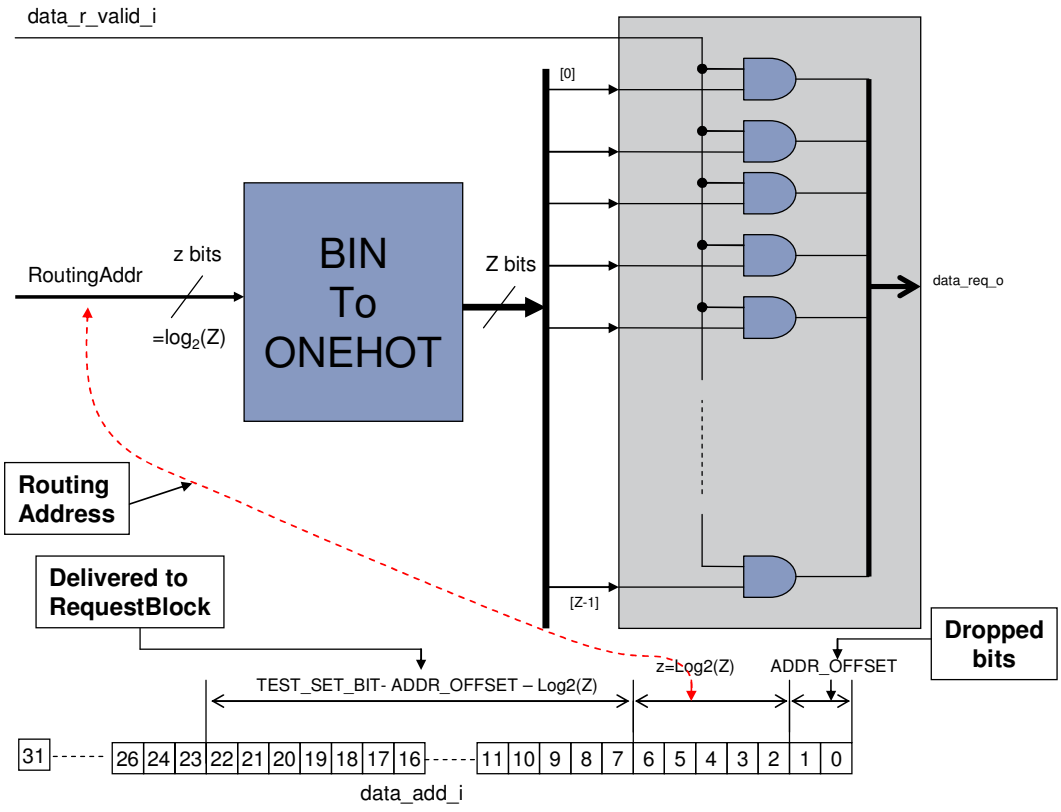

Figure 4-17: Detail of the OneHot data_req generation and address rule for the configuration ADDR_OFFSET=2, Z=32 and TEST_SET_BIT = 22

The RoutingAddr is used to route the requests first generating a one-hot Z bit signal then mixing it with the data_req_i and finally dispatching it to the several RequestBlocks. The request array is made by Z bits and each bit is dispatched following these rules:

$$\begin{cases} \text{Re}\,sponseBloc\,k[i].AdressDeco\,der\_\text{Re}\,q.data\_req\_o[j] \rightarrow \text{Re}\,questBlock\,[j].Arbitratio\,nTree\_CH\,0.port[i].data\_req\_i \\ j=0 \rightarrow Z-1 \\ i=0 \rightarrow X-1 \end{cases}.$$

and

$$\begin{cases} \text{Re}\,sponseBloc\,k[i].AdressDeco\,der\_\text{Re}\,q.data\_req\_o[j] \rightarrow \text{Re}\,questBlock\,[j].Arbitratio\,nTree\_CH\,1.port[i-X].data\_req\_i \\ j=0 \rightarrow Z-1 \\ i=X \rightarrow X+Y-1 \end{cases}.$$

The first rule is applied for all the AddressDecoder_Req attached to High Priority masters, while the second is related low priority masters (if any).

When a bit of the generated one-hot data_req_o array is set to one, it means that the current master wants to start a transfer (LOAD or WRITE) to the slave which is identified by the position of the “1” bit in the array.

The second part of the AddressDecoder_Req is made by the Grant multiplexing. This block receives Z grants, one for each memory, and the goal of this subblock is to select only the one related to the pending transfer. For this reason this cells uses the Routing address (pointed slave) to drive the Multiplexer that seleclts the proper grant bit of the array, sending it to the master. The log interconnect is capable to generate the grant in the same clock cycle the request is asserted, so the RoutingAddress is the right signal to be used for this purpose.

Finally the third sub-block is used to generate a label, called data_ID_o which is first propagated to the TestAndSet interface and then used to propagate back the read data (data_r_rdata) and the valid response (data_r_valid).

4.13.1. AddressInterleaving

The AddressDecoder_Req uses some bit of the address field to route the request to the right slave. For fine grain address interleaving the traffic is well balanced among the available slaves (i.e. memory banks). In case of coarse interleaving, the slave are mapped in different regions (i.e. the memory addresses are allocated in a contiguous way, and each bank hold an address region). Despite the fine interleaving balance the traffic, in some cases it also increases the memory contentions, so the designer have to select the right tradeoff. For example by chosing ADDR_OFFSET = 2, and let’s suppose that we have N_SLAVE=M=32 so 5 bits are used to route the request to the right slave.

WORD_LEVEL_INTERLEAVING

In this example the RoutingAddr is using the range [6:2] in the following way:

Address: → Destination

0x00000000 → SLAVE 0

0x00000004 → SLAVE 1

0x00000008 → SLAVE 2

0x0000000C → SLAVE 3

0x00000010 → SLAVE 4

0x00000014 → SLAVE 5

0x00000018 → SLAVE 6

0x0000001C → SLAVE 7

0x00000020 → SLAVE 8

0x00000024 → SLAVE 9

0x00000028 → SLAVE 10

0x0000002C → SLAVE 11

0x00000030 → SLAVE 12

0x00000034 → SLAVE 13

0x00000038 → SLAVE 14

0x0000003C → SLAVE 15

0x00000040 → SLAVE 16

0x00000044 → SLAVE 17

0x00000048 → SLAVE 18

0x0000004C → SLAVE 19

0x00000050 → SLAVE 20

0x00000054 → SLAVE 21

0x00000058 → SLAVE 22

0x0000005C → SLAVE 23

0x00000060 → SLAVE 24

0x00000064 → SLAVE 25

0x00000068 → SLAVE 26

0x0000006C → SLAVE 27

0x00000070 → SLAVE 28

0x00000074 → SLAVE 29

0x00000078 → SLAVE 30

0x0000007C → SLAVE 31

0x00000080 → SLAVE 0

0x00000084 → SLAVE 1

…….

BANK_LEVEL_INTERLEAVING

In this example the RoutingAddr is using the range [21:16] in the following way:

Address: → Destination

0x00000000 → SLAVE 0

……

0x0000FFFC → SLAVE 0

0x00010000 → SLAVE 1

……

0x0001FFFC → SLAVE 1

0x00020000 → SLAVE 2

……

0x0002FFFC → SLAVE 2

……

5. Timing Diagrams

This section gives some examples about the dynamic of the tranfers: we remind that the entire communication protocol is based on a simple flow control, where each tranfert must start asserting a request and holding the datapath signals until the master samples an high grant. The network is designed to sustain full bandwidth for conventional load/store operation. The case where the bandwidth is degrades involveses test-and-set operations, where two cycles are needed to complete the instruction.

The Figure 5-1 shows the timing diagram for a STORE followed by two successive LOADS. At time 6ns the master (first group of signals labelled as XP_SIDE) initiates the transfer asserting a request. At the same time it holds on its outputs the datapath signals (in this case it is performing a STORE) waiting to sample a high grant. Suddently, at time 8ns the masters samples the high grant, therefore it deasserts the request and since there are no new command and the previous operation was a STORE, the tranfert is done.

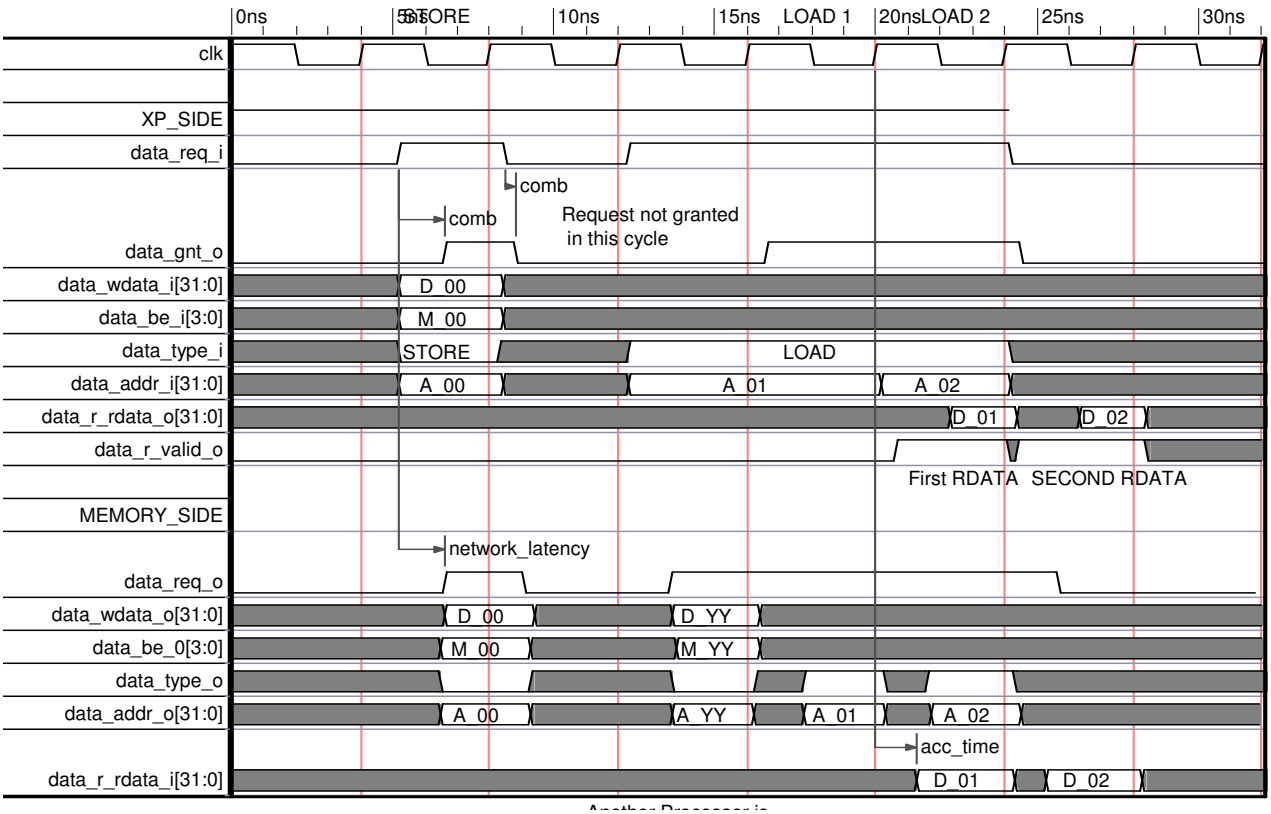

Figure 5-1: Timing Diagrams for Store and Loads

At time 12ns the master want to start a new operation (LOAD), therefore it assert the request the holds the datapath signal waiting for a high grant, but at time 16ns it samples a low grant, therefore it waits for the next clock cycle. In this clock cycle the memory is busy because another master is peforming a STORE.

At time 20ns it samples a high grant therefore since the master in this cycle (20to24) expects to receive the read data, but also can inject a new command. Since a second load is available the master in this timing window holds the request high, and changes the data_add_i. Before the end of this clock cycle, the master gets a high data_r_valid and the data_r_rdata. Since at tiem 24ns the master gets an high grant, it deasserts the the request, sample the read data, and waits the next clock cycle to sample the second load.

Figure 5-2 shows the logaritmich interconnect behaviour in case of test and set.

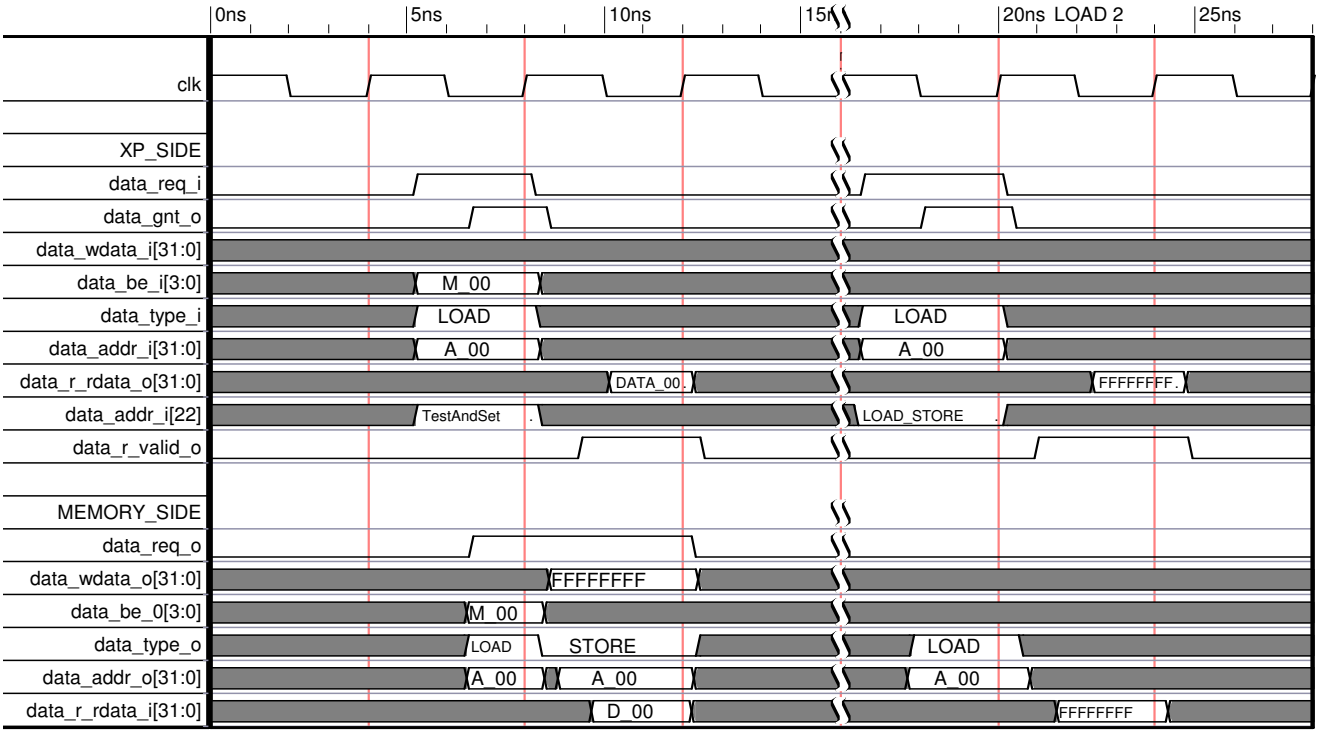

Figure 5-2: Timing Diagrams for test-and-set instruction followed by a LOAD

At time 5ns the master request for a LOAD at address A_00, with mask M_00, ans setting the TEST_SET_BIT of the address to “1”. At time 8ns the master gets a grant so the operation is committed, but the master in the following clock receive the read data that was previously stored in the address A_00. in the Window 8ns-12ns the TestAndSet interface assert a

SET overriding the content stored at address A_00 filling with “1” bits and depending of the M_00 that thmaster provided the cycle before.

At time 16ns the master perform a read on the address A_00 and in the successive clock cycle it receives the valued that the SET of the test-and-set wrote at time 12ns.