MANCHESTER 1824
The University of Manchester

TECHNISCHE UNIVERSITÄT DRESDEN

SpiNNaker2 datasheet version 0.03

August 19, 2021

Human Brain Project

SpiNNaker2 - a chip multiprocessor for neural network simulation. Datasheet.

Features

1. 152ARM Cortex M4F processors, each with:
   (a) 128 Kbytes of local code and data memory;
   (b) Memory Protection Unit (MPU)
   (c) DMA controller;
   (d) 4 timer/counters
   (e) vectored interrupt controller;
   (f) low-power 'wait for interrupt' mode.
2. Multicast communications router
   (a) 6 high-speed serial inter-chip bidirectional links;
   (b) 16,384 associative routing entries.
3. 2 LPDDR4 PHYs (16-Bit at maximum 2.4GBit/s)
   (a) up to 6.4 Gbytes/s sustained block transfer rate;
   (b) using PoP (Package-on-Package).
4. High-speed serial interface for host connection
5. Fault-tolerant architecture
   (a) defect detection, isolation, and function migration.
6. Boot, test and debug interfaces.
   (a) profiling counters

Introduction

SpiNNaker2 is a chip multiprocessor designed specifically for the real-time simulation of large-scale spiking neural networks. Each chip (along with its associated SDRAM chip) forms one node in a scalable parallel system, connected to the other nodes through high-speed serial links.

The processing power is provided through the multiple ARM Cortex M4F cores on each chip. In the standard model, each M4F models multiple neurons, with each neuron being a coupled pair of differential equations modeled in continuous 'real' time. Neurons communicate through atomic 'spike' events, and these are communicated as discrete packets through the on- and inter-chip communications fabric. The packet contains a routing key that is defined at its source and is used to implement multicast routing through an associative router in each chip.

One Quad Processor Element (QPE) on each SpiNNaker2 chip will normally perform system management functions; the communications fabric supports point-to-point packets to enable co-ordinated system management across local regions and across the entire system, and nearest-neighbor packets are used for system flood-fill boot operations and for chip debug.

Background

SpiNNaker2 was designed at the University of Manchester and the Technical University of Dresden within the EU-funded Human Brain Project in collaboration with ARM Limited. The work would not have been possible without EU funding, and the support of the EU and the industrial partners is gratefully acknowledged.

Intellectual Property rights

All rights to the SpiNNaker design are the property of the University of Manchester and TU Dresden with the exception of those rights that accrue to the project partners in accordance with the contract terms.

Disclaimer

The details in this datasheet are presented in good faith but no liability can be accepted for errors or inaccuracies. The design of a complex chip multiprocessor is a research activity where there are many uncertainties to be faced, and there is no guarantee that a SpiNNaker2 system will perform in accordance with the specifications presented here. The APT group in the School of Computer Science at the University of Manchester and the Chair for highly parallel VLSI systems and neuromorphic circuits at TU Dresden were responsible for all of the architectural and logic design of the SpiNNaker2 chip, with the exception of synthesizable components supplied by ARM Limited and others. All design verification was also carried out by the Manchester and Dresden groups. As such the industrial project partners bear no responsibility for the correct functioning of the device.

Error notification and feedback

Please email details of any errors, omissions, or suggestions for improvement to:
steve.furber@manchester.ac.uk.

Change history

version	date	changes
0.00	21/10/16	First draft
0.00	20/11/16	Still coming together
0.00	14/1/17	Yet more draft sections
0.01	7/6/18	Router and Comms Controller updates
0.02	7/6/18	Clean up
0.03	13/7/18	Extensive reorganisation

Contents

1 System architecture **5**
1.1 Routing 5
1.2 Time references 6
1.3 System-level address spaces 6

2 Chip Organization **7**
2.1 Block Diagram 7
2.2 System-on-Chip hierarchy 8
2.3 High-Level Goals 8
2.4 Register description convention 9

3 NoC **10**
3.1 Description 10
3.2 Register summary 11
3.3 Fault-tolerance 15
3.4 SDRAM DMA transfers 15
3.4.1 Features 15
3.4.2 Description 15
3.4.3 SDRAM reads and writes 16
3.4.4 SpiNNaker Router traffic 16
3.4.5 QPE to QPE DMA transfers 16
3.4.6 QPE to QPE reads and writes 17
3.5 Register summary 17
3.6 Fault-tolerance 17

4 Quad-core ARM processing subsystem (QPE) **19**
4.1 Features 19
4.2 Description 19
4.3 Quad-core ARM Cortex M4F subsystem organisation 20
4.4 The QPE DMA and NoC subsystem 20
4.5 Fault-tolerance 20

5 ARM Cortex M4F processing element (PE) **23**
5.1 Features 23
5.2 Organization 23
5.3 Address map 23
5.4 PE bus description 25
5.5 Fault-tolerance 25

6 Crossbar **27**

7 DMA Controller (DMAC) **28**
7.1 Features 28
7.2 Limitations 28
7.3 Operating modes 28
7.4 Memory-to-memory block transfers 28
7.4.1 Terminology 28
7.4.2 General 28
7.4.3 Read transfers 29
7.4.4 Write transfers 30
7.4.5 CRC 30
7.4.6 SDRAM transfers 30
7.4.7 QPE transfers 30
7.5 I/O transfers 31
7.6 Register summary 31
7.7 Fault-tolerance 31

8 Fixed-point Elementary Function Accelerator **32**
8.1 Features . . . 32
8.2 Description . . . 32
8.3 Implementation . . . 33
8.4 Accuracy . . . 33
8.5 Register summary . . . 33
8.6 Register details . . . 34
8.7 Fault-tolerance . . . 35

9 Random Number Generator **37**
9.1 Features . . . 37
9.2 Description . . . 37
9.3 Register summary . . . 37
9.4 Fault-tolerance . . . 38

10 Stochastic Rounding Accelerator **39**
10.1 Features . . . 39
10.2 Description . . . 39
10.3 Implementation . . . 39
10.4 Register summary . . . 39
10.5 Register details . . . 39
10.6 Fault-tolerance . . . 39

11 Machine Learning Accelerator (MLA) **44**
11.1 Features . . . 44
11.2 Overview . . . 44
11.3 Configuration and Command Registers . . . 44
11.4 Mode of Operation . . . 45
11.4.1 Matrix Multiplication . . . 45
11.4.2 Convolution . . . 45
11.5 ARM C code & Execution . . . 46

12 Counter/timer **49**
12.1 Features . . . 49
12.2 Register summary . . . 49
12.3 Register details . . . 50
12.4 Fault-tolerance . . . 51

13 Exchange - the PE communications switch **52**
13.1 Features . . . 52
13.2 Overview . . . 52
13.2.1 NoC interface . . . 52
13.2.2 Bus master interface . . . 53
13.3 Exception subunit . . . 54
13.3.1 Interrupt exceptions . . . 54
13.4 Register summary . . . 54
13.5 Register details . . . 55

14 Comms unit **58**
14.1 Features . . . 58
14.2 Overview . . . 58
14.2.1 Area map . . . 59
14.2.2 Packet transmitter . . . 59
14.2.3 Packet receiver . . . 60
14.2.4 Rx Module . . . 60
14.2.5 Messages . . . 62
14.3 Register summary . . . 62
14.4 Register details . . . 63
14.4.1 Transmitter . . . 63

14.4.2 Default receiver . . . 67
14.4.3 Messages . . . 70
14.4.4 Miscellaneous . . . 70
14.4.5 Receiver filters . . . 71
14.5 Bus bridge . . . 74
14.6 Monitoring . . . 75
14.7 Fault-tolerance . . . 75

15 NoC DMA Submodule (memDMA) 77
15.1 Features . . . 77
15.2 Description . . . 77
15.3 Register summary . . . 77
15.3.1 Flow control . . . 80
15.3.2 Local bus masters . . . 80

16 Bus bridge to NoC 81
16.1 Features . . . 81
16.2 Description . . . 81
16.2.1 SDRAM mapping . . . 81
16.2.2 Operation . . . 82

17 Response unit 83
17.1 Response packet generator submodule . . . 83

18 SpiNNaker Packet Router 85
18.1 Features . . . 85
18.2 Description . . . 86
18.3 Packet formats . . . 86
18.4 Control byte summary . . . 87
18.5 Debug access to neighbouring devices . . . 88
18.6 Internal organization . . . 89
18.7 Multicast (MC) router . . . 90
18.8 The core-to-core (C2C) router . . . 91
18.9 The nearest-neighbour (NN) router . . . 91
18.10 Time phase handling . . . 91
18.11 Packet error handler . . . 92
18.12 Register summary . . . 92
18.13 Register details . . . 93
18.14 Fault-tolerance . . . 102

19 SDRAM interface 104
19.1 Features . . . 104
19.2 DMA . . . 104
19.2.1 DMA Overview . . . 104
19.2.2 DMA SDRAM read . . . 105
19.2.3 DMA SDRAM write . . . 105
19.2.4 DMA SDRAM configuration . . . 106
19.3 Register summary . . . 106
19.4 Fault-tolerance . . . 106

20 Inter-chip transmit and receive interfaces 108
20.1 Features . . . 108
20.1.1 Key features for the Chip-to-Chip Link . . . 108
20.1.2 Key features for the LVDS AURORA link . . . 108
20.2 Configuration . . . 108
20.2.1 register selection . . . 108
20.3 Chip-to-Chip Link (C2C Link) . . . 109
20.3.1 C2C Link Transceiver . . . 109

21 Periphery **111**
21.1 Start-up Control . . . 111
21.2 Register File Interface . . . 111
21.3 GPIO MUX . . . 113
21.4 Clock Configuration . . . 116
21.5 Periphery Arm Cortex-M4 . . . 117
21.6 JTAG . . . 120
21.7 SPI . . . 122
21.7.1 NoC SPI . . . 122
21.7.2 SPI slave . . . 124
21.7.3 SPI master . . . 126
21.7.4 SPI flash start-up controller . . . 127
21.7.5 Spike SPI slave . . . 132
21.8 I2C . . . 135
21.8.1 I2C slave . . . 135
21.8.2 I2C master . . . 137
21.9 PWM . . . 141
21.9.1 PWM0 and PWM1 . . . 141
21.9.2 PWM2 . . . 143
21.9.3 GPIO debug output . . . 144
21.10 UART . . . 145
21.10.1 CUART . . . 145
21.10.2 Printf UART . . . 145
21.11 SDC Interface . . . 146

22 Host Interface **152**
22.1 UDP Routing . . . 152
22.1.1 Incoming Packets . . . 152
22.1.2 Outgoing Packets . . . 154
22.1.3 Packet Counters . . . 154
22.1.4 Frame ID Protocol . . . 154
22.2 UDT . . . 156
22.2.1 UDT Packet Type . . . 156
22.2.2 Channel Set up and Shut down . . . 157
22.2.3 Architecture . . . 157
22.2.4 UDT Configuration . . . 158
22.2.5 Buffer Memory . . . 159
22.3 Register summary . . . 159
22.4 GPIO MUX . . . 173
22.5 Register summary . . . 173
22.6 Fault-tolerance . . . 173

23 System Controller **174**
23.1 Features . . . 174
23.2 Register summary . . . 174
23.3 Fault-tolerance . . . 174

24 Watchdog timer **175**
24.1 Features . . . 175
24.2 Register summary . . . 175
24.3 Register details . . . 175

25 Power Management Architecture **178**

Appendix A Packaging **180**

Appendix B Input and Output signals **181**

Appendix C Electrical Specification **182**
C.1 Operating Temperature . . . 182
C.2 Power Supply . . . 182
C.3 Current consumption . . . 182

Appendix D Application Note **183**
D.1 External Components . . . 183
D.2 PCB Integration Guideline . . . 183

Appendix E SpiNNaker2 Address Map and Register Summary **184**
E.1 Core memories . . . 184

Appendix F NoC packet formats and header definitions **185**
F.1 NoC packet format . . . 185
F.2 NoC header definitions . . . 185
F.3 NoC packet formats . . . 189

1 System architecture

SpiNNaker2 is designed to form a node of a massively parallel system. The system architecture is illustrated in Fig. 1.

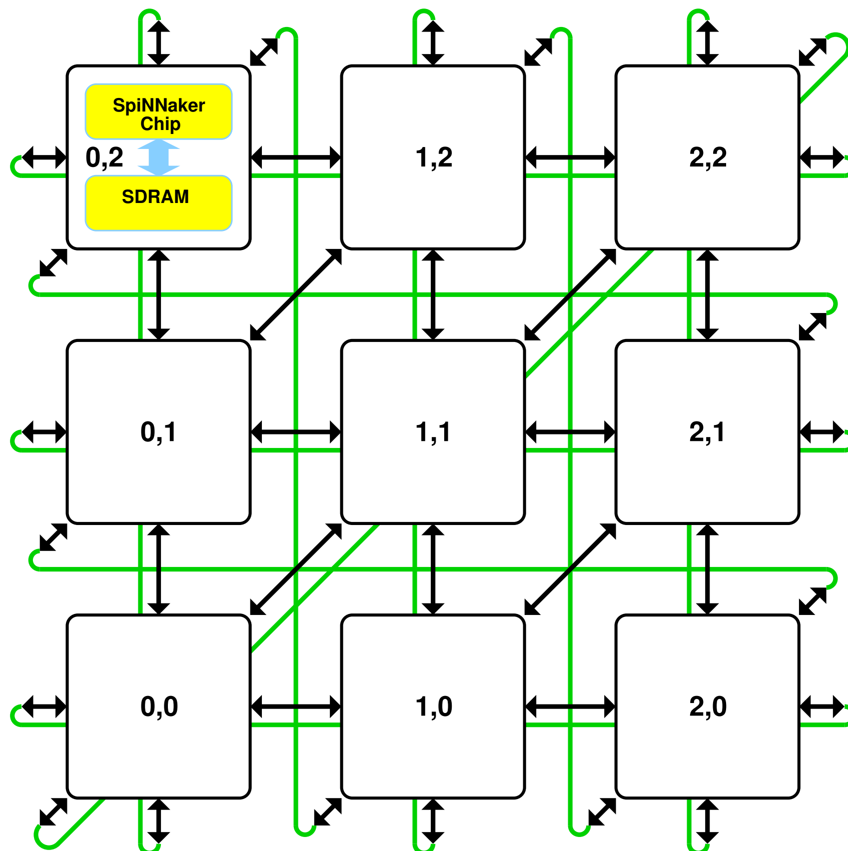


Figure 1: SpiNNaker2 system architecture

1.1 Routing

The nodes are arranged in a triangular mesh with bidirectional links to 6 neighbors. The system supports multicast packets (to carry neural event information, routed by the associative Multicast Router), core-to-core packets (to carry system management and control information, routed by table look-up) and nearest-neighbor packets (to support boot-time flood-fill and chip debug).

Deadlock avoidance

The communications system has potential deadlock scenarios because of the possibility of circular dependencies between links. The policy used here to prevent deadlocks occurring is:

1. **no Router can ever be prevented from issuing its output.**

The mechanisms used to ensure this are:

1. outputs have sufficient buffering and capacity detection so that the Router knows whether or not an output has the capacity to accept a packet;
2. where this fails the packet is 'dropped' to a Router register, and either re-inserted later by hardware or the Monitor Processor informed;

The expectation is that the communications fabric will be lightly-loaded so that blocked links are very rare. Where the operating system detects that this is not the case it will take measures to correct the problem by modifying routing tables or migrating functionality.

Errant packet trap

Packets that get mis-routed could continue in the system for ever, following cyclic paths. To prevent this all (apart from nearest-neighbor) packets are time stamped and a coarse global time phase signal is used to trap old packets. To minimize overhead the time stamp is 2 bits, cycling $00 \rightarrow 01 \rightarrow 11 \rightarrow 10$, and when the packet is two time phases old (time sent XOR time now = 0b11) it is dropped and an error flagged to the local Monitor Processor. The length of a time phase can be adapted dynamically to the state of the system; normally, timed-out packets should be very rare so the time phase can be conservatively long to minimise the risk of packets being dropped due to congestion.

1.2 Time references

Each processor has timer/counters driven from a 100MHz clock which can be used to support time reference signals, for example a 1ms interrupt could be used to generate the time input to the real-time neural models. Software may use this to generate the local time phase information.

Firefly synchronization

The local time phase, used for errant packet trapping, can be maintained across the system by a combination of local slightly randomized timers and local phase-locking using nearest-neighbor communication.

Time phase accuracy

If the system time phase is F and the skew is K (that is, all parts of the system transition from one phase to its successor within a time K), then a packet has at least $F - K$ to reach its destination and will be killed after at most $2F + K$. Thus, if we want to allow for a maximum packet transit time of $F - K = T$ and can achieve a minimum phase skew of K, then T and K are both system constants and we should choose $F = T + K$. The longest packet life is then $2T + 3K$.

1.3 System-level address spaces

The system incorporates different levels of component that must be enumerated:

1. Each Node (where a Node is a SpiNNaker chip plus SDRAM) must have a unique, fixed address which is used as the destination ID for a core-to-core packet, and the addresses must be organised logically for algorithmic routing to function efficiently.
2. Processors will be addressed relative to their host Node address, but this mapping will not be fixed as an individual Processor's role can change over time. Internal to a Node there is hard-wired addressing of each Processor for system diagnosis purposes, but this mapping will generally be hidden outside the Node.
3. The neuron address space is purely a software issue. Neurons should occupy an address space that identifies each Neuron uniquely within the domain of its multicast routing path. Where these domains do not overlap it is possible to reuse the same address, though this must be done with considerable care. Neuron addresses can be assigned arbitrarily; this can be exploited to optimize Router utilization (e.g. by giving Neurons with the same routing requirements related addresses so that they can be routed by the same Router entries).

2 Chip Organization

2.1 Block Diagram

The primary functional components of SpiNNaker2 are illustrated in Fig. 2.

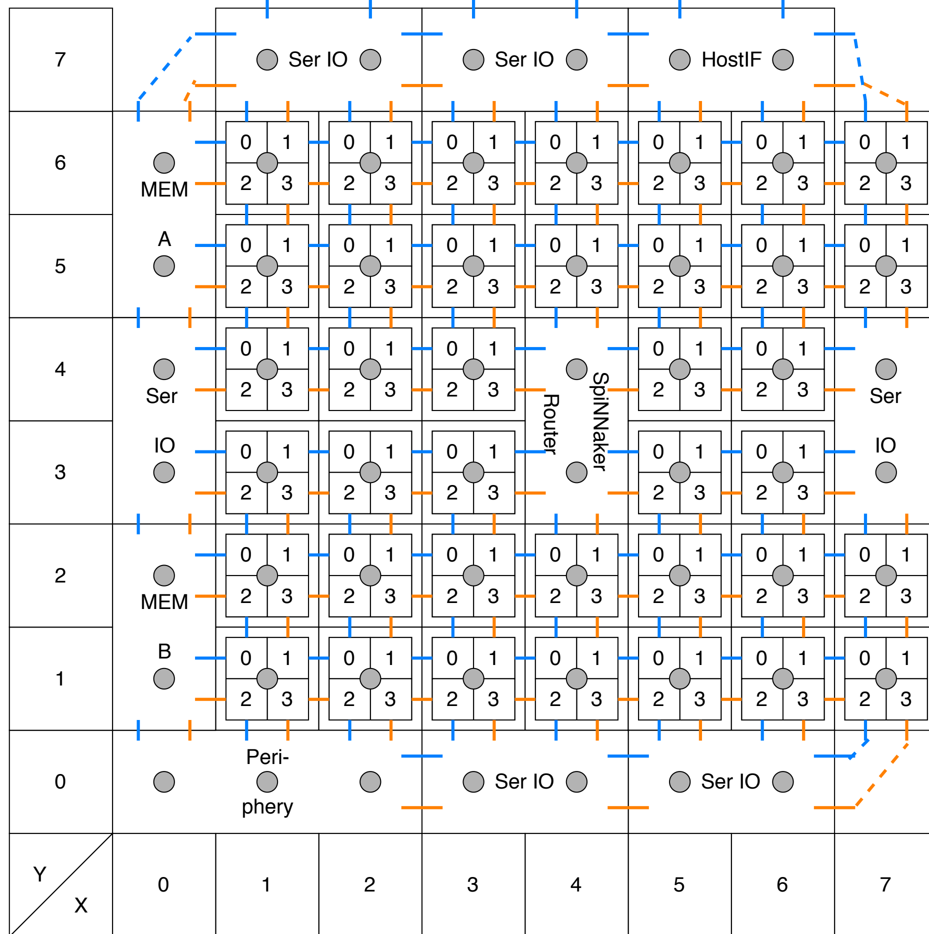


Figure 2: SpiNNaker2 NoC Topology

Each chip contains 38identical Quad-core Processing Elements. These processors are responsible for modeling one or more neuron populations - a population being a group of neurons with associated inputs and outputs (although some processors may be reserved as spares for monitoring or fault-tolerance purposes).

The Router is responsible for routing neural event packets both between the on-chip processors and from and to other SpiNNaker2 chips. The SerIO interface components are used to send and receive SpiNNaker packets to and from other SpiNNaker2 chips. Inputs from the various on- and off-chip sources are assembled into a serial stream which is then passed to the Router.

Various resources are accessible from the processor systems via the NoC. Each of the processors has access to the shared off-chip (but co-packaged) SDRAM, and various system components also connect through the NoC so that all processors have access to these components.

The sharing of the SDRAM is an implementation convenience rather than a functional requirement, although it may facilitate function migration in support of fault-tolerant operation.

2.2 System-on-Chip hierarchy

The SpiNNaker2 chip is viewed as having the following structural hierarchy, which is reflected throughout the organisation of this data sheet (see Fig. 2):

1. chip-wide NoC connectivity (Section 3)
   (a) 4.8 Gbytes/s bandwidth
      i. 192-bit flit with 128-bit payload
   (b) a regular array of NoC routers (Section 3)
      i. each with configuration registers
2. 38Quad-core Processing Elements (QPEs - Section 4)
   (a) NoC router (Section 3)
   (b) dynamic memory sharing between neighbor PEs
   (c) register file with controls for the QPE
   (d) four ARM Cortex M4F processor elements (PEs - Section 5), each with:
      i. 128Kbyte SRAM memory array
      ii. DMA controller (Section 7)
      iii. fixed-point exponential and logarithm accelerator (Section 8)
      iv. pseudo and true random number generator (Section 9)
      v. a machine learning accelerator (Section 11)
      vi. timer/counters (Section 12)
      vii. interrupt controller
      viii. DVFS (Dynamic Frequency and Voltage Scaling) and power switching (Section 25)
      ix. memory BIST (Section ??)
3. SpiNNaker packet router (Section 18)
   (a) multicast, core-to-core and nearest-neighbour routing functions
4. 2 LPDDR4 SDRAM memory controllers and PHYs (Section 19)
5. 6 bidirectional high-speed serial inter-chip links for SpiNNaker packets (Section 20)
6. bidirectional high-speed serial link to host machine (Section ??)
7. Periphery module with standard interfaces for boot, test and debug support (Section 21)

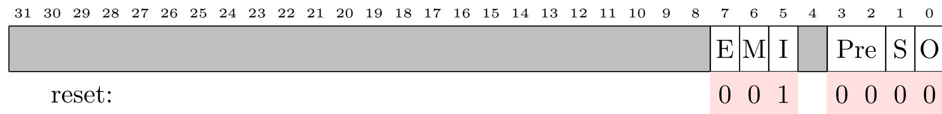
2.3 High-Level Goals

SpiNNaker2 is designed with the following high-level design objectives:

1. Power constraints
   (a) ¿10x SpiNNaker1 energy-efficiency
   (b) 1W node power, for small-scale mobile applications (e.g. drones)
2. Performance
   (a) ¿10x SpiNNaker1 within the same power budget: 1nJ per connection/s
3. Scalability: from single node to large-scale system
   (a) greater usability of small-scale system than SpiNNaker1
4. much better I/O capability than SpiNNaker1
   (a) 1Gbyte/s I/O bandwidth per card
5. Computational neuroscience support
   (a) support for 0.1ms time step simulation in real time
   (b) 10k efferent synapses per neuron
      i. 100k inputs per population with 10% sparse connectivity
   (c) neuron models: LIF; Izhikevich; AdExp; GIF; stochastic thresholds; multi-compartmental; dendritic branches
   (d) synapse models: single exponential; double exponential; alpha
   (e) plasticity models: STDP; LTP; LTD; BCPNN
   (f) other neutron features: gap junctions; neuromodulators
   (g) recording: spike data; synaptic weight changes?

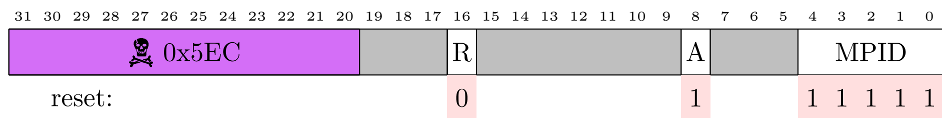
2.4 Register description convention

Registers are 32-bits (1 word) and are usually displayed in this datasheet as shown below:



1. The grey-shaded areas of the register are unused. They will generally read as 0, and should be written as 0 for maximum compatibility with any future functionality extensions.
2. Reset values, where defined, are shown against a red shaded background.

Certain registers in the System Controller have protection against corruption by errant code:



1. Here any attempt to write the register must include the security code 0x5EC in the top 12 bits of the data word. If the security code is not present the write will have no effect.

3 NoC

The SpiNNaker2 NoC (Network-on-Chip) is a shared communications resource that interconnects all of the processing and related sub-systems on the SpiNNaker2 chip.

3.1 Description

The NoC is organised as a 2D mesh with a NoC router associated with each QPE and additional NoC routers supporting the SDRAM controller and the SpiNNaker Router. A major concern with NoC design is traffic management to avoid congestion and, most importantly, deadlock scenarios. On the SpiNNaker2 NoC deadlock avoidance *within* the NoC is guaranteed by the use of static X-Y ordered routing, which ensures that no circular dependencies can arise in routing patterns. External traffic management ensures packets in the NoC can always exit unimpeded. Traffic management strategies are particular to the class of traffic, as described below; all these have the common characteristic that they do not allow packets to enter the NoC unless it is certain that they can leave although different traffic manages this in different ways.

NoC Router

Each QPE Quad-core processing element includes two independent NoC router as shown in Fig. 3.

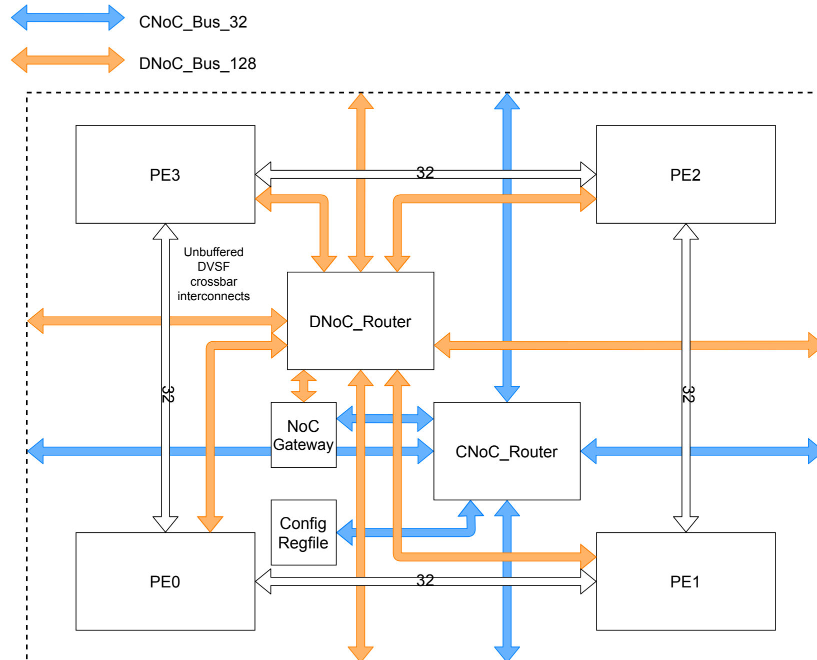


Figure 3: NoC QPE structure

The Configuration-NoC (CNoC) is using a 32 bit flit width and is running at reference frequency. It has main function of packet transport during chip initialisation e.g. for PLL configuration, however the CNoC can be used for management purposes during operation. For the main chip data transfers the Data-NoC (DNoC) is used. It uses 192 bit wide flits and runs at 500 MHz (change for final implementation) for maximum data throughput. Both NoCs operate on basis of the packet format described in Appendix F. A simplified NoC router structure is shown in Fig. 4.

Configuration NoC

Transfers in CNoC are done in a sequence of 32 bit packet parts. First the 32 bit NoC header is transferred followed by 32bit address and up to 128bit data. The CNoC router operates in a worm-hole flow control scheme: the route determined by packet header stays established until the last flit was transmitted. Each CNoC router port contains a combinational packet decoder which determines the destination port based on the router coordinate and packet destination field. As described before a x-y routing strategy was implemented. By configuration the algorithm can be set as y-first or x-first

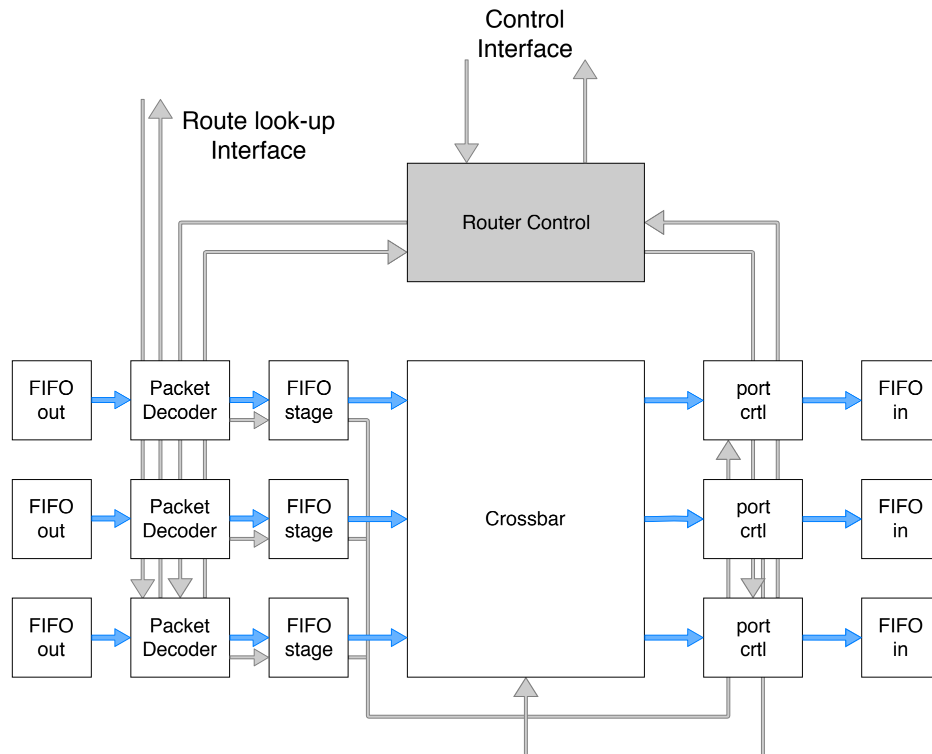


Figure 4: DNoC router structure

routing. To ensure fairness in packet transmission around-robin arbitration logic is implemented in each output port control block.

Data NoC

In DNoC flit size of 192bit is equal to the packet size. Meaning the whole packet is transferred through NoC in a parallel fashion. In contrast to worm-hole switching im CNoC a packet cannot block a route through the DNoC router. The packet route is determined by a combinational x-y routing algorithm (y-first and x-first). Additionally a look-up based route algorithm is implemented. Which allows to customize the global packet routing strategy. The DNoC router supports multicast at QPE-level. If the destination DNoC router receives a packet with more than one PE destination bits set it will replicate the packet to corresponding PE ports. As in CNoC router the output ports determine the next transferred packet based on a round-robin arbitration scheme.

NoC gateway

By using the same packet format in CNoC and DNoC it is possible to route a given packet using CNoC or DNoC depending on the C-bit in packet format. If the C-bit is set to one the packet will be routed via CNoC. In the case that the packet was started from a DNoC sourse it will change at soon as possible to CNoC via the NoC gateway. The packet is routed via CNoC until the destination xy coordinate and will change back to DNoC if necessary. By setting the C-bit to zero, the packet is routed via DNoC and is transferred to CNoC at destination if necessary.

3.2 Register summary

DNoC router enable

The DNoC router is disabled by default. To enable the router bit zero of register
LaTeX error: \getReg() failed to find requested information! must be set to one. It is possible to deactivate router ports individually in the case of physical link faults by setting port bits to zero. To save power during operation sel_we can be enabled. This prevents FIFO writing of unused data fields e.g. in the case of 32bit payload.

Register 3.1: DNOC_EN (0x00000014)

	unused	port	sel_we	router
Bits	31–11	10–2	1	0
Reset	—	0x1ff	0	0

port (RW🔒) router port enable
sel_we (RW🔒) selective fifo write enable
router (RW🔒) router enable

DNoC router decoder

The routing behaviour of the DNoC router can be changed with register 0x00000018 . There a re two basic mode: combinational and LUT mode selectable via bit 0. Combinational routing determines the router destination port by the packet destiantion and router position. Here two modes can be chosen for each router input: X-first or Y-first. For combinational mode packets with a non-existing destination ID can be dropped automatically by enabling the drop_en feature. If this happened the decode_error field is set by the router. The second routing mode is LUT mode. In LUT mode the destination port is the result of the destination coordinates and the lookup table entry of registers 0x0000001c to 0x00000028 .

Register 3.2: DNOC_DECODE (0x00000018)

	unused	decode_error	unused	xfirst	unused	drop_en	mode
Bits	31–25	24–16	15–13	12–4	3–2	1	0
Reset	—	—	—	0x1ff	—	1	0

decode_error (R🔒) DNoC router decoder error occured
xfirst (RW🔒) DNoC router decoder x-first routing (only active in combinational mode)
drop_en (RW🔒) DNoC router decoder drop faulty packets
mode (RW🔒) DNoC router decoder mode 0-combinational, 1-LUT

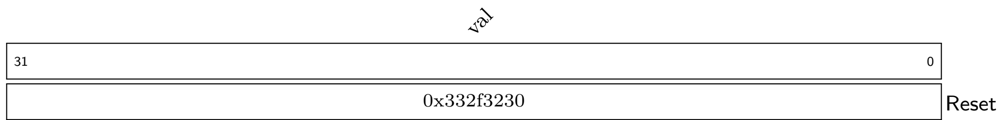
Register 3.3: DNOC_RT_LUT0 (0x0000001c)

	val
Bits	31–0
Reset	0x51504520

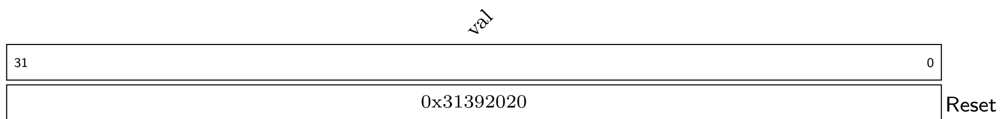
Register 3.4: DNOC_RT_LUT1 (0x00000020)

	val
Bits	31–0
Reset	0x52462730

Register 3.5: DNOC_RT_LUT2 (0x00000024)



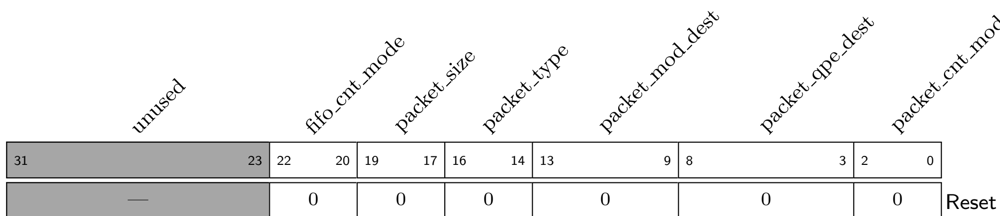
Register 3.6: DNOC_RT_LUT3 (0x00000028)



DNoC router statistics trigger modules

The DNoC router contains hardware to monitor NoC traffic during runtime. Each input and output port has counters which count depending on the configuration of register 0x0000002c .

Register 3.7: DNOC_STAT_MOD (0x0000002c)



fifo_cnt_mode	(RW🔒)	fifo counting mode;0:off; 1:full; 2:empty; 3:almost full; 4:read
packet_size	(RW🔒)	packet module destination to trigger on
packet_type	(RW🔒)	packet qpe destination to trigger on
packet_mod_dest	(RW🔒)	packet size to trigger on
packet_qpe_dest	(RW🔒)	packet type to trigger on
packet_cnt_mode	(RW🔒)	count if mached bit0:size, bit2:type, bit3:destination, all set: cnt any packet

DNoC router statistics reporting

Statistics counter values are periodically sent via CNoC to a desired address (region). The time between packet creation is determined by cnt_max of register 0x00000030 . The internal counter runs with sys_tick. The destination start address can be configured with register 0x00000034 and the last address by register 0x00000038 . The destination address is incremented if field packet_addr_incr_en is enabled.

Register 3.8: DNOC_STAT_TOP_CTRL (0x00000030)

unused	config_stop	config_mode	packet_addr_wrap_en	packet_addr_incr_en	mod_dest	qpe_dest	cnt_max	enable	
31 25	24	23 21	20	19	18 14	13 8	7 1	0	
—	—	0	0	0	0	0	0	0	Reset

config_stop (R🔒) stop: max address was reached
config_mode (RW🔒) bit2:reduced mode; bit1:fifo_cnt; bit0:packet_cnt, one-hot
packet_addr_wrap_en (RW🔒) wrap around (stop otherwise)
packet_addr_incr_en (RW🔒) increment address
mod_dest (RW🔒) packet mod desination
qpe_dest (RW🔒) packet qpe desination
cnt_max (RW🔒) cycle cnt max
enable (RW🔒) enable statistics module

Register 3.9: DNOC_STAT_TOP_START_ADDR (0x00000034)

val	
31 0	
0	Reset

val (RW) start address of stat. packet

Register 3.10: DNOC_STAT_TOP_ADDR_MAX (0x00000038)

val	
31 0	
0	Reset

val (RW) stop address of stat. packet

CNoC router enable

In contrast to DNoC router , the CNoC router is enabled by default. The CNoC router only supports combinational routing and X-/Y-decoding can be only enabled for all ports.

Register 3.11: CNOC_ROUTER_CTRL (0x0000003c)

unused	out_port_en	unused	port_en	decode_xfirst	enable	
31 22	21 16	15 8	7 2	1	0	
—	0x3f	—	0x3f	1	1	Reset

out_port_en	(RW🔒)	output port enable
port_en	(RW🔒)	decoder enable
decode_xfirst	(RW🔒)	enable x-first routing
enable	(RW🔒)	enable of CNoC router

3.3 Fault-tolerance

It is ensured on NoC level that packet loss cannot occur. The NoC is essentially a packet transfer from FIFO to FIFO. The NoC router is only transferring a packet if the local destination FIFO has space, otherwise packets are stalled. If FIFO space was feed by a consumer the NoC router will continue pushing the stalled packets. A packet can be lost on if the NoC stalls an the sender is still pushing, or if the consumer is discarding received packets.

Reconfiguration

1. TO BE DONE

3.4 SDRAM DMA transfers

3.4.1 Features

1. 6.4 Gbytes/s bandwidth between QPEs and SDRAM
   (a) bulk transfer by DMA bursts
   (b) individual word (etc.) access by PE processors
2. SpiNNaker packet delivery between QPEs and SpiNNaker Router
3. communication between QPEs
   (a) DMA between QPE memories
   (b) ‘remote’ reads and writes of QPE memories by any PE
4. event/interrupt signalling via configuration registers
5. guaranteed packet delivery
   (a) deadlock-free static routing

3.4.2 Description

The critical bandwidth demand arises from the requirement to enable the SDRAM controller to deliver data to/from a QPE at the SDRAM’s peak bandwidth of 6.4 Gbytes/s. The NoC supports this by virtue of carrying 128-bit (16-byte) flits at 400M/s, giving the NoC a peak bandwidth of 6.4 Gbytes/s on a single route.

Traffic management is the responsibility of the QPE requesting the transfer. The QPE has a central DMA Controller (DMAC) which is responsible for all block transfers between any of the QPE’s memory areas and the SDRAM. The primary algorithm is that each QPE is restricted to a single outstanding SDRAM command, and that command will be issued only when:

1. READS: there is a free buffer within the QPE to accept the incoming data burst;
2. WRITES: there is a buffer within the SDRAM controller to accept the outgoing write address and command. The SDRAM controller then pulls the write data from the source PE when it has buffer space to accommodate that data.

Each DMAC has multiple channels and can manage multiple ‘concurrent’ transfers although the individual transfer bursts are regulated to avoid initiating transfers which cannot leave the NoC; *bursts* from different channels may be interleaved.

In addition, SDRAM traffic moves horizontally within the 2D NoC framework to avoid any interference with the SpiNNaker Router traffic, which moves vertically.

3.4.3 SDRAM reads and writes

Individual PEs can issue individual read and write operations to the SDRAM. These will typically be low bandwidth and will observe the natural limit of one outstanding command per PE (four per QPE).

Because processor access is typically more latency critical, individual accesses will take priority over DMA bursts.

3.4.4 SpiNNaker Router traffic

SpiNNaker Router traffic comprises incoming and outgoing SpiNNaker packets, each of which is a single NoC flit.

Incoming packets: The SpiNNaker Router will send an incoming multicast packet as multiple individual packets, one to each QPE that should receive the packet. Within the QPE the packet is buffered; flow control is regulated by a credit arrangement with the Router (q.v.) where credit is returned via NoC tokens as the FIFO buffer is emptied. Credit return is done in 'blocks' to avoid acknowledging every incoming packet. This flow control prevents packets 'backing up' onto the NoC without the need to drop packets at the QPE, thus simplifying packet congestion handling by confining it to the Router.

"Packets in the FIFO are routed appropriately to the PEs, multicast packets being duplicated appropriately as determined by a 4-bit field in the NoC packet which is filled in by the Router. The incoming packet should raise an interrupt if it arrives when the input buffer is empty.

Incoming packets could, in principle, arrive at up to the full NoC capacity of 500M packets/s, though in practice the rate is unlikely to exceed 100M packets/s and will usually be much lower than this.

Traffic management relies on the incoming packets arriving at an individual QPE no faster than that can be placed into the input buffer(s) by the local DMA system. In the limit, packets may need to be dropped to keep things moving, though in practice this should not happen.

Outgoing packets: Outgoing packets are sent when a neuron spikes, and the total rate for the chip is likely to be in the region of 1M packets/s or lower, although when used to convey SDP packets this may be ten time higher. This is still much lower than the incoming packet rate.

[SDP – SpiNNaker Datagram Protocol – packets are used to initialise the machine and to set up application data structures, extract results, send application messages to the host, and such like. They are a higher-level protocol that uses multiple SpiNNaker point-to-point packets to deliver larger data payloads.]

Traffic management is achieved by:

1. prioritising internally-sourced packets within the SpiNNaker Router over externally-sourced packets, thereby ensuring that all internally-sourced packets can clear the Router even if they are all going out through the same chip-to-chip link

As noted above, SpiNNaker Router traffic moves vertically across the NoC, avoiding interference with SDRAM traffic, which moves horizontally.

3.4.5 QPE to QPE DMA transfers

QPE to QPE traffic will typically comprise large blocks of data being transferred between QPEs by DMA. This supports the "synapse-centric" refactoring of the SpiNNaker neuron model, and could amount to a significant total traffic of up to 3 Gbytes/s (with a 0.1 ms time step), but distributed across the SpiNNaker2 NoC.

In principle a memory-to-memory DMA in the chip-wide address space could be performed by a DMAC in *any* QPE; in practice only the DMAC associated with the source or destination QPE should be used. This can be enforced by limiting the destination address registers to a ('virtual') range of local addresses. DMA burst requests will only be issued when the DMAC – at QPE level – has a (temporary) buffer already available to receive the returned data burst at full speed from the NoC. (Writes to the SRAM – which may be in a somewhat slower clock domain and are subject to arbitration with PE activity – are likely to be significantly slower; thus the inter-QPE bandwidth seen by the user for any single transfer channel will be significantly less than the NoC peak bandwidth.)

The source QPE in a QPE-to-QPE DMA also has a data burst buffer dedicated to each potential transfer. When a burst is requested this buffer is filled using QPE local reads; once it is full and has

received the burst request indicating a destination buffer is prepared it can transfer the DMA burst across the NoC at full rate, knowing that it can be received without blocking.

There are different approaches to setting up the DMACs which depend partly on the hardware design and partly on software choices. At *transfer* (as opposed to burst) level transactions could be prearranged and triggered by sending some form of (short) 'message' between PEs or – as all DMACs will be visible to any PE – all programming could be handled from a single PE which could be at either the source or destination NoC node. **In all cases it is important that the resources are managed and a pair of buffers – one in the source and one in the destination QPE – are reserved for each logical QPE-to-QPE logical channel.** There are <TBC ????> channels in each DMAC. See the section on the QPE DMAC for full details.

3.4.6 QPE to QPE reads and writes

Individual PEs can address memory directly anywhere across the chip. These will typically be low bandwidth and will have little hardware management, so could cause problems if used carelessly?

The source QPE can restrict these to single outstanding transaction from each QPE (to any particular destination) which means that, if a QPE is able to buffer <**34-ish**> pending requests no further control protocol is needed.

3.5 Register summary

TO BE DONE - what configuration does the NoC require?

1. Resetting?
2. Its own ID/position (could be 'pin' configured on macrocell). The PEs need to be able to find this, too.
3. Some DMAC channel (resource) allocation: this is probably a software function although there needs to be some (centralised?) arbitration.
4. More . . .

3.6 Fault-tolerance

The NoC should guarantee delivery of all packets it accepts. This is very important because some packets are mediating processor bus cycles and that processing element will be stalled indefinitely. Given the limit on outstanding transactions imposed at the QPE level, this could, over time, 'take out' the entire QPE.

Packets do a significant amount of clock domain resynchronisations; each comes with a very small (but finite) probability of failure. If possible this should not risk losing a packet; producing a duplicate packet instead would be preferred if this is an option.

It would be possible to produce additional safeguards around NoC failure at the transaction source. An example would be a 'bus' time-out on a processor operation, resulting in an ARM 'abort' if a read or write operation had not responded within a given (programmable?) interval. (Subsequently arriving responses can be dropped, within limits.) This would divert the failed operation into an error recovery routine.

A watchdog mechanism (q.v.) provides a last line of defence by being able to reset and restart a failing PE.

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

4 Quad-core ARM processing subsystem (QPE)

SpiNNaker2 incorporates 38quad-core ARM processing subsystems (Quad Processor Elements - QPEs) which provide the computational capability of the device. Each of the 152individual cores is capable of generating and processing neural events, sending spikes and other packets via the NoC and, alternatively, of fulfilling the role of Monitor Processor.

Although able to operate independently, the cores within a QPE are able to share memory – with a small timing penalty – and share resources such as access to the system NoC.

4.1 Features

1. NoC router;
2. four synthesized Processing Element (PE) modules each with:
   (a) an ARM Cortex M4F processor with:
      i. single-precision floating-point hardware;
   (b) 128Kbyte SRAM;
   (c) fixed-point exponential, logarithm accelerators;
   (d) random number generators;
   (e) a machine learning accelerator;
   (f) extensive DMA functions.

4.2 Description

The QPE (Quad-core Processing Element) represents a node on the SpiNNaker2 NoC. Each QPE incorporates a NoC router and four PEs each containing an ARM Cortex M4F processor, DMA and Spike management engines.

The QPE uses a single NoC router for data routing, and the relevant connectivity between this and the DMA Controller and the 4 PEs is illustrated in Fig. 5.

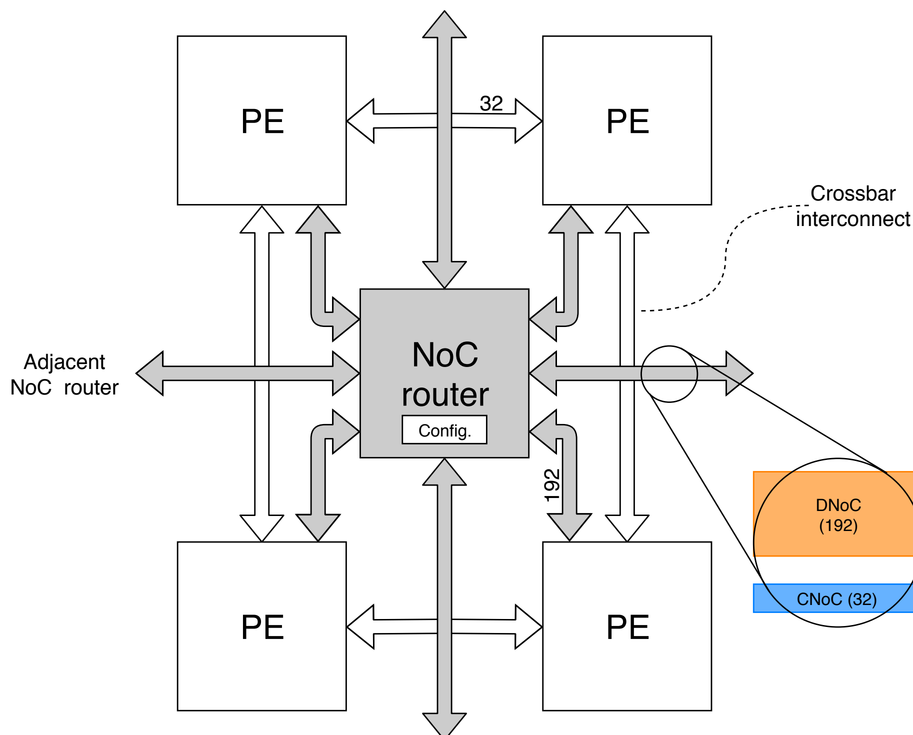


Figure 5: QPE internal routing

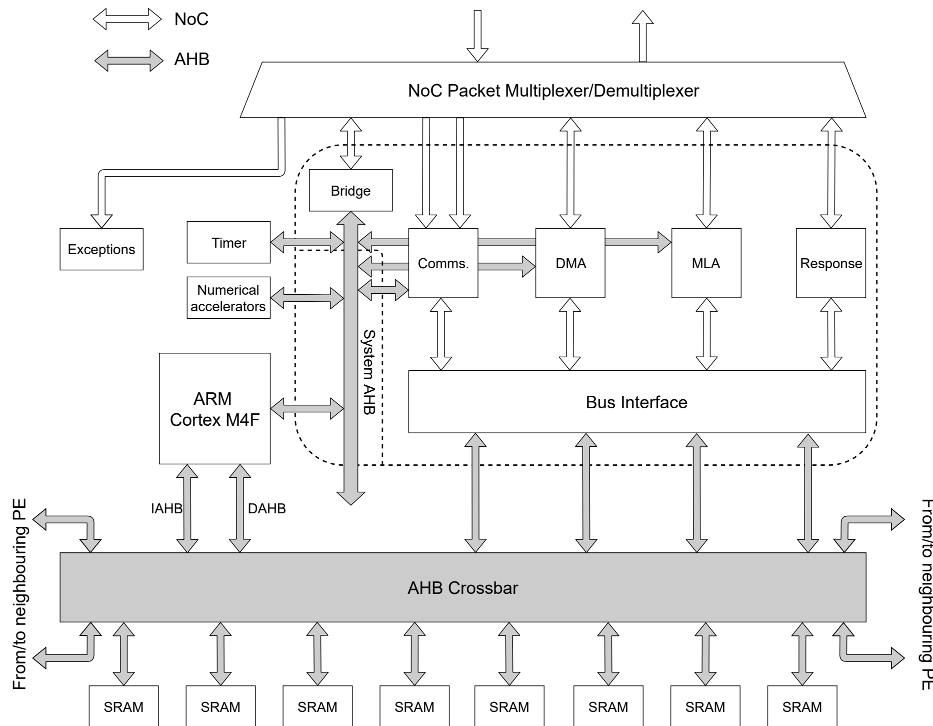


Figure 6: ARM Cortex M4F bus structure

4.3 Quad-core ARM Cortex M4F subsystem organisation

The Cortex M4 has uses several output buses in different areas of its address space. Fig. 6 shows the arrangement of the AHB interfaces which can connect the PEs locally. More details of the bus structure are given on page 23 which describes the PE structure; the Cortex-M4 is described in ARM document DDI 0439D.

The PEs within a QPE are linked by the NoC-Interfaces to the central data NoC-router and the local shared-memory interfaces. These two, parallel buses link with the PE's local memory through a crossbar switch. Part of the address space decodes to neighbours' local memories and a reference to this space is routed via the neighbour's crossbar switch, from where it is switched to the appropriate bank or RAM. There is a small timing penalty for this [* probably one wait state for writes and two for reads, but TBC *]. Additional penalties may result from contention for access to particular buses or RAM banks. It is therefore recommended that this mechanism is used for less frequent accesses and specifically not for commonly running code or frequently accessed data such as a stack.

Successful operation of this mechanism relies on the corresponding subsystems being driven by the same clock source. If different clock sources are employed, including separate clocks of nominally the same frequency, correct operation is not guaranteed.

4.4 The QPE DMA and NoC subsystem

Each of the four PEs within the QPE operates within its own DVFS domain, but there are central resources within the QPE that operate continuously at the maximum voltage and frequency. In particular, these include the NoC router and its associated DMA Manager.

In the noc-access module within the PE connected to the local AHB Bus of the ARM core are submodules for DMA and Spike processing. Those submodules can interact with the central DMA management module via NoC-packets to issue or receive requests and arbitrate their data transfers.

Fig. 7...

4.5 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

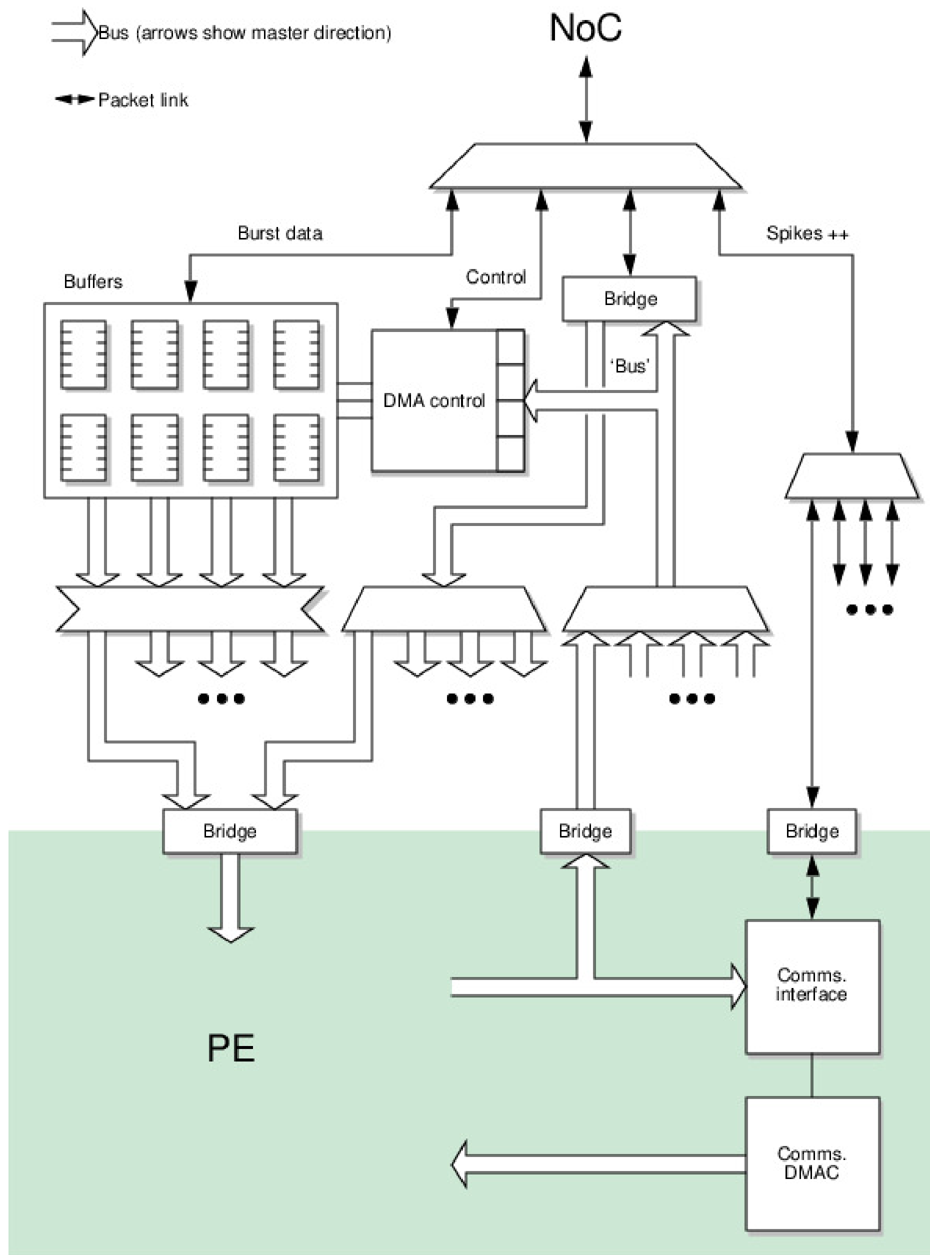


Figure 7: PE NoC interface

5 ARM Cortex M4F processing element (PE)

The ARM Cortex m4f (with its associated memory) forms the core processing resource in SpiNNaker.

5.1 Features

1. single-precision floating-point unit (FPU)
2. nested vectored interrupt controller (NVIC)
3. memory protection unit (MPU)
4. local AHB buses with:
   (a) 128Kbyte SRAM memory array
   (b) pseudo and true random number generator
   (c) fixed-point exponential and logarithm accelerator
   (d) machine learning accelerator
   (e) 4 timer/counters
5. local power management and test support:
   (a) DVFS (Dynamic Frequency and Voltage Scaling) and power switching
   (b) memory BIST
   (c) scan chain control
6. privileged access to protect configurations (see ARM M4 manual, Control Register)

5.2 Organization

See ARM DDI 0439D: the ARM Cortex M4 Technical Reference Manual.

5.3 Address map

Area	From	To	Size	Function
Local SRAM	0000_0000	0001_FFFF	128 KB	Tightly bound SRAM
Local SRAM	0002_0000	0007_FFFF	384 KB	Direct neighbours local SRAM
Local SRAM	0008_0000	000F_FFFF	512 KB	Remapped Quad local SRAM
System	3000_0000	DFFF_FFFF		Memory mapped System access via NoC
Private Peripheral Bus	E000_0000	E010_0000		Reserved by ARM
NMU	E010_0000	E010_01FF		NMU (Neural Multiplication Unit) KISS (Pseudo RNG)
NMU	E010_0200	E010_020F		NMU TRNG (True RNG) Config
NMU	E010_0210	E010_03FF		NMU TRNG Random
NMU	E010_0400	E010_05FF		NMU EXP
NMU	E010_0600	E0FF_FFFF		NMU SR
Timer 1	E100_0000	E100_001F		Timer 1
Timer 2	E100_0020	E100_003F		Timer 2
Comms Config	E200_0000	E2FF_FFFF		NoC Comms Controller
ML ACC	E300_0000	EFFF_FFFF		Machine Learning Accelerator
QPE NoC SRAM	F000_0000	F1FF_FFFF		QPE NoC SRAM (global)
QPE regfile	F200_0000	F5FF_FFFF		QPE regfile (global)
JIB management	F600_0000	E600_03FF		JIB management regfile (global)
UART 0	F600_0400	F600_04FF		UART 0 (global)
UART 1	F600_0500	F600_1FFF		UART 1 (global)
JIB Core Regfile	F600_2000	F?		Jib core regfile (global)

Tightly bound SRAM map

From	To	Function
0000_0000	0000_7FFF	Instructions
0000_8000	0000_FFFF	Data #0
0001_0000	0001_7FFF	Data #1
0001_8000	0001_FFFF	Data #2

The suggested arrangement is up to the software. The division into four banks (probably 2-way interleaved) is a recommendation. Programmers should try to avoid instruction and data accesses to the same bank as collisions will reduce performance although it should still function.

Other cores' memories within the quad appear above address 0002_0000. The core 'seen' in each space is its position in a cycle of cores/RAMs.

Designating the cores: {PE 0, PE 1, PE 2, PE 3} and the RAMs {RAM 0, RAM 1, RAM 2, RAM 3}:

Core	Base address	RAM	Speed
PE 0	0000_0000	RAM 0	Fast
PE 0	0002_0000	RAM 1	Slow
PE 0	0004_0000	RAM 2	Slow
PE 0	0006_0000	RAM 3	Slow
PE 1	0000_0000	RAM 1	Fast
PE 1	0002_0000	RAM 2	Slow
PE 1	0004_0000	RAM 3	Slow
PE 1	0006_0000	RAM 0	Slow
PE 2	0000_0000	RAM 2	Fast
PE 2	0002_0000	RAM 3	Slow
PE 2	0004_0000	RAM 0	Slow
PE 2	0006_0000	RAM 1	Slow
PE 3	0000_0000	RAM 3	Fast
PE 3	0002_0000	RAM 0	Slow
PE 3	0004_0000	RAM 1	Slow
PE 3	0006_0000	RAM 2	Slow

This gives typical software an identical view of the system from any core. All four cores may operate independently – or largely so – in their 'own' SRAM; two cores could shut down leaving an identical pair (say, PE0 & PE2) with extra RAM; a single core could operate with all the RAM, etc.

'Slow' access here means at least two cycles for a write operation and three cycles for a read operation (TBC), plus possibly penalties for collisions; access to other cores' spaces is therefore practicable in general code, if not overused.

Remapped Quad local SRAM

The the Quad's SRAM is aliased into another part of the SRAM map to facilitate other programming models. The SRAM is subdivided into 4 banks each:

Base address	PE 0	PE 1	PE 2	PE 3	Speed
`0008_0000`	RAM 0_0	RAM 1_0	RAM 2_0	RAM 3_0	Fast
`0008_8000`	RAM 0_1	RAM 1_1	RAM 2_1	RAM 3_1	Fast
`0009_0000`	RAM 1_0	RAM 2_0	RAM 3_0	RAM 0_0	Slow
`0009_8000`	RAM 1_1	RAM 2_1	RAM 3_1	RAM 0_1	Slow
`000A_0000`	RAM 2_0	RAM 3_0	RAM 0_0	RAM 1_0	Slow
`000A_8000`	RAM 2_1	RAM 3_1	RAM 0_1	RAM 1_1	Slow
`000B_0000`	RAM 3_0	RAM 0_0	RAM 1_0	RAM 2_0	Slow
`000B_8000`	RAM 3_1	RAM 0_1	RAM 1_1	RAM 2_1	Slow
`000C_0000`	RAM 0_2	RAM 0_2	RAM 0_2	RAM 0_2	Varies
`000C_8000`	RAM 0_3	RAM 0_3	RAM 0_3	RAM 0_3	Varies
`000D_0000`	RAM 1_2	RAM 1_2	RAM 1_2	RAM 1_2	Varies
`000D_8000`	RAM 1_3	RAM 1_3	RAM 1_3	RAM 1_3	Varies
`000E_0000`	RAM 2_2	RAM 2_2	RAM 2_2	RAM 2_2	Varies
`000E_8000`	RAM 2_3	RAM 2_3	RAM 2_3	RAM 2_3	Varies
`000F_0000`	RAM 3_2	RAM 3_2	RAM 3_2	RAM 3_2	Varies
`000F_8000`	RAM 3_3	RAM 3_3	RAM 3_3	RAM 3_3	Varies

This mapping intends to leave some fast 'local' memory – at the same address for all cores to aid software distribution – whilst providing a *contiguous* data space (`000C_0000 - 000F_FFFF`) with the same absolute map for all cores to aid pointer sharing et alia. Here two banks of 'local' memory have been chosen to facilitate easy, parallel code and data accesses at full speed.

Other alias variants may be included too.

Memory mapped system access

Area	From	To	Size	Function
Invalid	`0000_0000`	`2FFF_FFFF`		Local Memory/Peripherals/unused
QPE SRAM	`3000_0000`	`30FF_FFFF`	152x 128 kB	SRAM off all QuadPEs
QPE Regfiles	`3200_0000`	`3203_FFFF`	38x 1 kB	Register files of all QuadPEs
TBD	`3208_0000`	`5FFF_FFFF`		currently unused
SDRAM	`6000_0000`	`DFFF_FFFF`	2 GB	Memory of external SDRAM
Invalid	`E000_0000`	`FFFF_FFFF`		Local Memory/Peripherals/unused

5.4 PE bus description

The Cortex-M4 is master of three AHBs. Two of these, the 'ICode' and 'DCode' memory interfaces are at coincident addresses. These are led to a crossbar switch which allows parallel access to local SRAM providing there is no resource contention. To reduce the possibility of contention for RAM, the RAM is divided into four [**TBC**] addressable banks and further [** two or four? TBC **]-way interleaved within each bank. In addition to the 'ICode' and 'DCode' inputs the crossbar has bus connections leading from neighbouring PEs within the QPE – via the central crossbar switch – and an external master from the QPE for global addressing and high-speed DMA access, e.g. for SDRAM communications.

The majority of the address space is mapped via the 'System interface'. Accesses on this bus are regarded as non-local [** with a possible exception regarding local accelerators and peripheral devices? TBC **] and the bus is routed to the QPE central resources. The cycle time of this is difficult to predict as it involves clock domain crossing to the QPE and, in some cases communication to more distant units via the NoC.

The Cortex-M4 'Private Peripheral Bus' (PPB) – an APB interface is [**currently largely up for grabs; suggestions please! **].

Figure here?

5.5 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

6 Crossbar

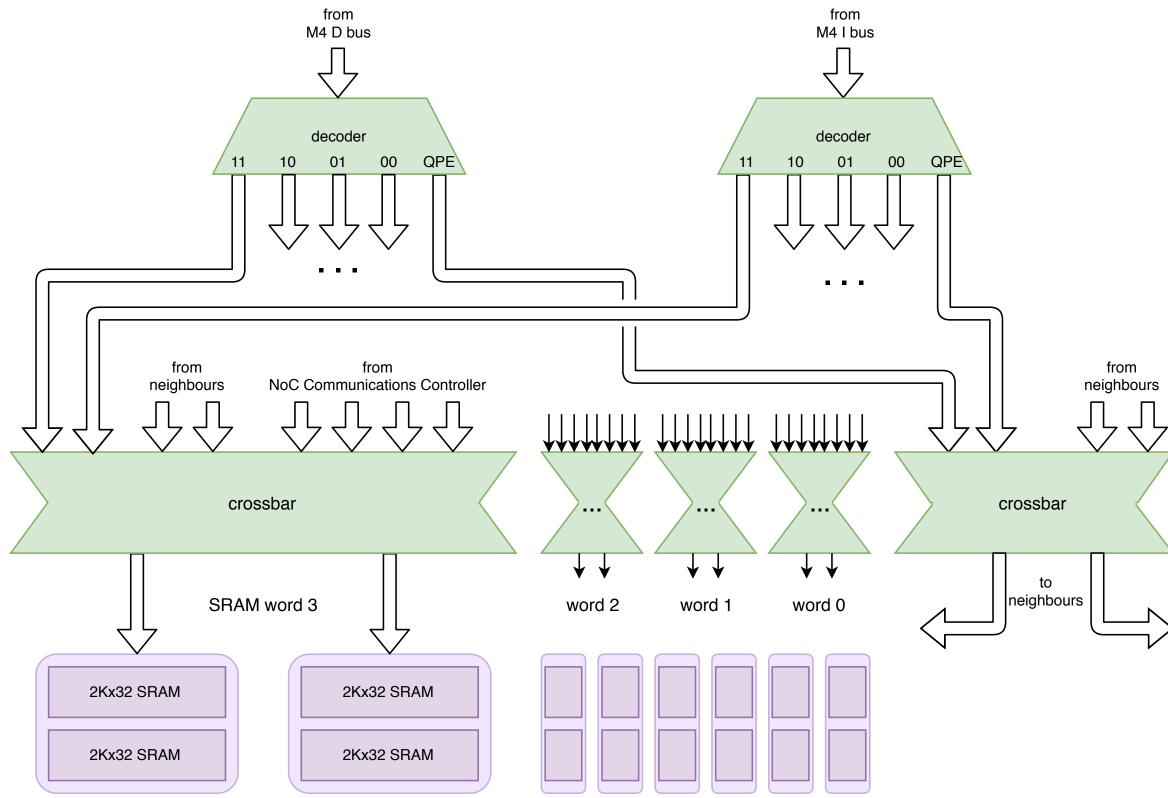


Figure 8: Crossbar organisation

In Fig. 8, the inputs on the individual crossbar elements are indicative of their priority, with higher priority on the left hand side. The crossbar does not support burst operation, as the slaves also do not. Bursts are handled like individual nonsequential transfers.

7 DMA Controller (DMAC)

Each QPE includes a DMA unit which supports a number of different communications modes.

7.1 Features

1. Coming soon!

7.2 Limitations

A number of limitations are imposed on the exact nature and number of DMA transfers to protect the NoC from overloading and packets 'backing up' from destinations. These are enforced firstly by the number of available DMA channels for the user and latterly by limiting outstanding transactions on the NoC.

A general strategy is to limit each QPE to a single unsolicited request to any given destination *possibly more aggressive for simplicity?* and ensure that there is space to accept as many such requests at each destination as there are possible senders. A higher limit or dynamic allocation has certain attractions but becomes considerably more complicated and the returns may not be great.

7.3 Operating modes

DMAC covers several(??) different classes of operation. These are largely (effectively) independent but all share the NoC interface and thus may compete for resources.

1. Memory-to-memory block DMA
2. I/O DMA
3. Bridging PEs to the NoC

7.4 Memory-to-memory block transfers

7.4.1 Terminology

Channel: one of several possible 'places' where a DMA *transfer* may be instigated. An illustrative picture would be to have four channels per QPE with one channel notionally 'owned' by each PE.

Transfer: a complete data movement from memory to memory of an arbitrary data size. Transfers are specified at software level.

Burst: a data 'item', typically consisting of several/many words with a well specified upper size. Multiple bursts will usually be required for one transfer. Bursts are scheduled by hardware.

Initiator: the DMAC which is programmed with the transfer data.

Target: a DMAC responding to a request generates by an *initiator* DMAC.

Buffer: a hardware-controlled data buffer capable of sourcing/sinking a *burst* at full NoC speed. Within the QPE data is moved at local speeds, thus more buffers than *channels* are needed if a *transfer* is not to be unduly delayed.
It is not yet clear if the same buffers should act for inputs and output; it is vital that every *initiator* DMAC ensures that at least one buffer is available (or in use) for *target* responses at all times, else deadlock could ensue.

7.4.2 General

Both the SRAM and SDRAM transfers have a lot in common. This, and the sections below, try to unify the commonality. The important issue is to coordinate buffers at each side of the NoC.

Imagine a transfer request to a DMAC at (initiator) software level defining a local and a remote address, a transfer length and a direction. There may be other, minor details such as CRCs, interrupt/signalling etc. but these can be subsumed in the message.

All potential, pending or progressing transfers are scheduled locally from a pool of possibilities. The pool includes all the active *block structured* transfers; it is limited by the number of channels.

Transfers are (logically) subdivided into data bursts, each burst being no longer than the agreed DMA burst buffer size. (Division may be done piecemeal as transfers proceed but it is convenient to think of the bursts from the start of the operation.)

Bursts are scheduled such that at any time only a single burst can be active between the local DMAC and any specific corresponding unit. Bursts between the local unit and *different* correspondents may be active simultaneously.

Figure 9 shows an overview of the actions of the corresponding DMACs for (nominal) read and write bursts. When a buffer is empty– either when its data has been transmitted of when it is written into local RAM – it is deallocated and recycled (i.e. it the bottom of each column in the figure).

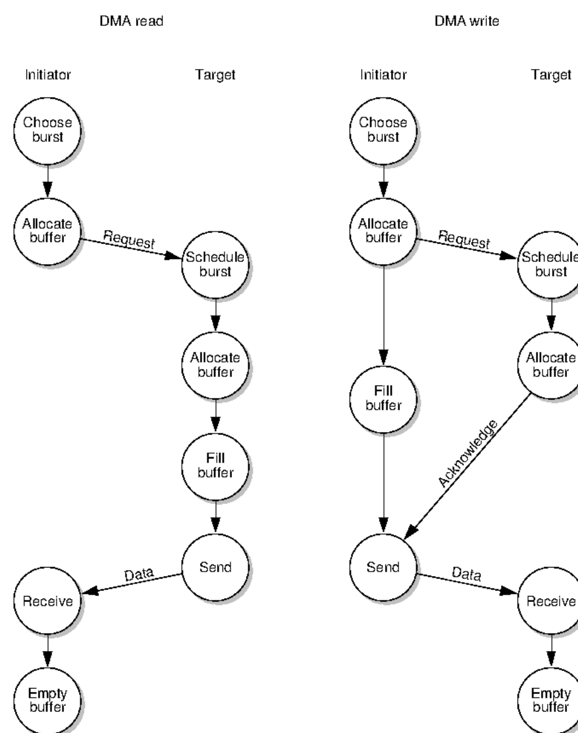


Figure 9: DMA NoC coordination

In addition to acting as an initiator, each DMAC must guarantee to accept as many burst requests from the NoC as there are possible (remote) initiators.

Several DMA *transfers* may be active simultaneously. The DMAC supports ***probably*** **four** transfer channels. If there are multiple active transfers the DMAC is responsible for scheduling the bursts and will interleave bursts from different channels as the local buffer-SRAM bandwidth is typically much lower than the NoC bandwidth so data will reside in buffers for some time. [**Could have some prioritisation if desired, here.**]

The transmission of *burst requests* (i.e. the initiation of a transaction) must be limited to avoid more request packets arriving at the destination (in this case the SDRAM interface) than it can accept from the NoC. **The simplest and therefore most likely rationing mechanism is to limit each QPE to a single outstanding transaction of this type.** This does not preclude multiple DMA *transfers* operating concurrently as the DMAC will contain several data buffers so a subsequent burst request can be sent as soon as one burst is received.

It is expected that DMA channels will be allocated in software, typically as one per PE although this is not enforced by the hardware.

7.4.3 Read transfers

1. From the possible pending bursts, one is chosen according to <some prioritisation scheme>with the limitation that this must not be to the same correspondent as any currently outstanding burst.

2. A read buffer is allocated for the incoming data.
3. A request is sent to the correspondent which includes the details of the data burst required. On receipt the correspondent schedules the burst amongst competing requests and fills its own buffer (as and when one is available).
4. The burst is sent and accepted at full NoC speed.

7.4.4 Write transfers

Write transfers are closely analogous to read transfers.

1. From the possible pending bursts, one is chosen according to <some prioritisation scheme>with the limitation that this must not be to the same correspondent as any currently outstanding burst.
2. A request is sent to the correspondent which includes the details of the pending data burst. On receipt the correspondent schedules the burst amongst competing requests and allocates its own buffer (as and when one is available).
3. The initiator fills a transmit buffer from its local memory, then waits for permission to transmit.
4. When a receive buffer is available, the target sends an acknowledgement, allowing the burst to be sent (as and when it is ready).
5. The burst is sent and accepted at full NoC speed.

7.4.5 CRC

**** TBC ****

7.4.6 SDRAM transfers

SDRAM transfers are the primary reason for the employment of DMA. The expected operations are dominated by reads from – and, to a much lesser extent, writes to – the SDRAM from individual PEs. DMA transfers are performed as bursts so that 'trains' of data are carried across the NoC to make best use of both NoC and SDRAM bandwidth.

The anticipated bandwidth requirements w.r.t SDRAM transfers is:

1. Reads <**some number of words per second**>.
2. Writes <**some lesser number of words per second**>.

7.4.7 QPE transfers

DMA between PE SRAMs is allowed to enable the passing of messages. In a similar manner to the SDRAM transfer the message (or a burst thereof) is assembled in a buffer in the transmitting QPE and sent when informed there is a free receiver buffer.

In principle there is no difference in the protocols in transferring to/from SDRAM or QPE SRAM.

A feature of such transfers is the ability to generate an interrupt (at receiver or sender) when they are complete, flagged on the last burst of a transfer.

In operation a (short) request packet from the sender can cause a receiver to reserve a buffer and reply when ready; transfers then occur when buffers at both ends are primed.

The bandwidth requirements w.r.t SRAM transfers is hard to predict and will depend on future software requirements. It is anticipated that individual transfers will be less frequent than SDRAM operations but there may be more of them running in parallel and independently. The nature on the NoC helps with this as its bandwidth is aimed at satisfying a peak demand on an individual path whereas inter-QPE traffic will occupy many different paths.

1. Reads <**some number of words per second**>.
2. Writes <**some lesser number of words per second**>.

7.5 I/O transfers

I/O transfers are primarily concerned with incoming spike packets. The spike receiver is basically a UART and can buffer several (**how many? q.v.**) incoming spikes locally. Like a UART it can have an associated DMA request (as an alternative to a Rx interrupt) which can cause incoming packets to be placed in SRAM. To assist with buffering a 'modulo' mechanism can be set up with the DMA address.

A mechanism for indicating to the appropriate PE that this has been done – presumably an interrupt – is needed. In this operating mode the DMAC *may* need to take responsibility for this. ** TBC **

7.6 Register summary

The size and local destination(/source) address of the transfer are programmable. The address is confined to within the (Q)PE so is expressed with a local alias rather than using the global address space. *Likely to be a PE ID plus an address within that PE.*

The 'remote' address in the transfer is also decomposed into a local address and a QPE/PE address. For convenience this is done in hardware so the register view specifies an address in the global address space.

Transfers are broken down into bufferable-sized bursts. Bursts are buffered off(/for) the NoC in the QPE so that reception(/transmission) is possible at full speed on the NoC itself. The burst length is programmable, up to the hardware buffer size of <**some number of**>words.

**** TBC ****

7.7 Fault-tolerance

All TBC

Note: there is an assumption that every NoC packet is delivered correctly. In the event of failure problems could multiply quite quickly. Some form of 'back-stop' – presumably a time-out mechanism – may be appropriate to trap problems. **** TBC ****

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

8 Fixed-point Elementary Function Accelerator

Accelerator for the calculation of exponential and natural logarithm functions is included as an AHB slave within each ARM Cortex M4F processor sub-system.

8.1 Features

1. Evaluate e^x or $log(x)$ (base e).
2. s16.15 and s0.31 fixed-point (input/output formats can be mixed)
3. Single-precision floating-point format (IEEE 754-2008) with subnormals on input and output is supported
4. Programmable accuracy (1 to 16 iterations)
5. 1-2 ulp accuracy and monotonicty in most of the available numerical formats
6. 7-22 clock cycles per operation, depending on accuracy
7. Saturation when arguments are out of range
8. Bus error when $x \leq 0$ for logarithm.

8.2 Description

The unit employs the standard s16.15 (accum), s0.31 (signed long fract) fixed-point or single-precision float formats for both operand and result. Results are accurate to one least-significant bit (LSB/ulp) for almost all operands, functions and numerical formats when compared to C double-precision function accuracies (see below for the summary). If the operand would yield a result that is too small for the fixed-point format, a zero (binary value: 0x00000000) is returned. If the operand would result in an overflow, the maximum value of the fixed-point data format (binary value 0x7FFFFFFF) is returned. When the accelerator is used in floating-point format, for exponential, if the argument is ± 0, then 1 is returned, if $-\infty$, then $+0$ is returned, if $+\infty$, then $+\infty$ is returned. For floating-point logarithm, if ± 0 is given, $-\infty$ is returned, if the argument is 1, $+0$ is returned, if the argument is negative, NaN is returned and bus error is signalled, if $+\infty$ is given, $+\infty$ is returned. In both floating-point functions, if NaN argument is given, NaN is returned. Input ranges of different functions and numerical formats are shown in tables 1 and 2.

s16.15 inputs (operands) to the exponential function unit are written to AHB address 0xE0100400. Results stored in the output registers are read in the same format from the same AHB address. With fixed-point formats, there is also an option to read an answer in a different format from the argument, this is achieved by reading from a different address. The unit stores up to four outputs internally, depending on the address used. The unit can have at most one value in calculation and it will hold the answer in the output registers until another calculation is requested. If there is no value in the calculation pipeline (e.g. after restarting the unit), a read access returns an undefined result.

There is no error monitoring implemented in the unit. Users have to ensure correct interfacing to the unit. The unit has an internal FSM that will ignore any requests for a new calculation when it is already working on another operation (inputs will be stalled by the AHB protocol).

Table 1: Approximate minimum and maximum ranges of values of *exp* operation with different 32-bit 2's complement formats. * - saturates to 0x0 below this range; † - saturates to 0x7fffffff above this range.

Format I/O	exp input range	exp output range
s16.15/s16.15	$-10.397...^*$ to $11.09...^\dagger$	0.00003... to 65534.5...
s0.31/s0.31	-1 to $(-2^{-31})^\dagger$	0.367... to 0.99...
s16.15/s0.31	$-21.487...^*$ to $(-2^{-15})^\dagger$	$\sim 2^{-31}$ to 0.99...
s0.31/s16.15	-1 to $1 - 2^{-31}$	0.367... to $\sim e$
float/float	$-103.278...$ to 88.722...	$\sim 2^{-149}(sub.)$ to $\sim 3.403 \times 10^{38}$

Table 2: Approximate minimum and maximum ranges of values of *ln* operation with different 32-bit 2's complement formats. ‡ - saturates to 0x80000000 below this range; † - saturates to 0x7fffffff above this range.

Format I/O	ln input range	ln output range
s16.15/s16.15	2^{-15} to $2^{16} - 2^{-15}$	$-10.397...$ to $11.09...$
s0.31/s0.31	$0.367...^{\ddagger}$ to $0.99...$	$-0.99...$ to $\sim -2^{-31}$
s16.15/s0.31	$0.367...^{\ddagger}$ to $\sim e^{\dagger}$	-1 to $0.99...$
s0.31/s16.15	2^{-31} to $0.99...$	$-21.487...$ to 0
float/float	$\sim 2^{-149}(sub.)$ to $\sim 3.403 \times 10^{38}$	$-103.278...$ to $88.722...$

8.3 Implementation

The main algorithm is based on the shift-and-add algorithm presented in chapter 8 of the book *Elementary Functions: Algorithms and Implementation, 3rd. ed.* by J-M. Muller. The architecture of the unit is shown in Fig. 10. The core part of the unit performs two iterations of the algorithm in one cycle and this is what defines the accuracy of the functions. One iteration roughtly finds two bits of the answer. Range reduction and range reconstruction units are reducing the values to the convergence domain of the iterative algorithm - refer to the book for more information.

8.4 Accuracy

The implementation was verified by sweeping over all meaningful operands for the full accuracy setting and a subset of arguments for lower accuracy settings, and measuring the error relative to the C double-precision math.h libraries. Tables 3-5 give accuracies of some of the configurations.

Table 3: Accuracy of *s16.15* functions for the different numbers of iterations N. $ulp = 2^{-15} = 0.000030517578125$ (absolute)

	exp			log		
N	Max error	Average	Monotonic	Max error	Average	Monotonic
16	1 ulp	0.477 ulp	Y	2ulp	0.5ulp	Y
14	8 ulp	0.564 ulp	Y	2ulp	0.5ulp	Y
12	125 ulp	3.172 ulp	Y	2ulp	0.5ulp	Y
8	30044 ulp	693.33 ulp	Y	2ulp	0.56ulp	N

Table 4: Accuracy of *s0.31* functions for the different numbers of iterations N. $ulp = 2^{-31}$ (absolute).

	exp			log		
N	Max error	Average	Monotonic	Max error	Average	Monotonic
16	2ulp	0.29ulp	Y	2ulp	0.54ulp	Y
14	9ulp	2.22ulp	Y	7ulp	2.32ulp	Y
12	127ulp	39.13ulp	Y	87ulp	34.3ulp	Y
8	32416ulp	9537ulp	N	22644ulp	8737ulp	N

8.5 Register summary

Base address: 0xE0100400.

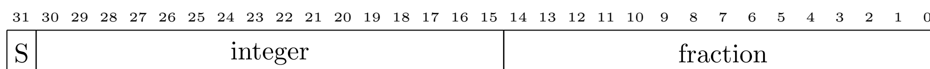
Table 5: Accuracy of single-precision floating-point functions for the different numbers of iterations N. Ulp value relative to the exponent of the numbers.

	exp			log		
N	Max error	Average	Monotonic	Max error	Average	Monotonic
16	1ulp	0.066ulp	Y	192ulp	0.25ulp	Y
12	1ulp	0.086ulp	Y	2043ulp	0.27ulp	Y
8	252ulp	15ulp	N	∼ 10Mulp	8.28ulp	N

Name	**Offset**	**R/W**	**Function**
r0: s16.15 exp 0	0x00	R/W	Operand/Result
r1: s16.15 exp 1	0x04	R/W	Operand/Result
r2: s16.15 exp 2	0x08	R/W	Operand/Result
r3: s16.15 exp 3	0x0C	R/W	Operand/Result
r4: s0.31 exp 0	0x10	R/W	Operand/Result
r5: s0.31 exp 1	0x14	R/W	Operand/Result
r6: s0.31 exp 2	0x18	R/W	Operand/Result
r7: s0.31 exp 3	0x1C	R/W	Operand/Result
r8: float exp 0	0x20	R/W	Operand/Result
r9: float exp 1	0x24	R/W	Operand/Result
r10: float exp 2	0x28	R/W	Operand/Result
r11: float exp 3	0x2C	R/W	Operand/Result
r12: unused			
r13: unused			
r14: unused			
r15: unused			
r16: s16.15 log 0	0x40	R/W	Operand/Result
r17: s16.15 log 1	0x44	R/W	Operand/Result
r18: s16.15 log 2	0x48	R/W	Operand/Result
r19: s16.15 log 3	0x4C	R/W	Operand/Result
r20: s0.31 log 0	0x50	R/W	Operand/Result
r21: s0.31 log 1	0x54	R/W	Operand/Result
r22: s0.31 log 2	0x58	R/W	Operand/Result
r23: s0.31 log 3	0x5C	R/W	Operand/Result
r24: float log 0	0x60	R/W	Operand/Result
r25: float log 1	0x64	R/W	Operand/Result
r26: float log 2	0x68	R/W	Operand/Result
r27: float log 3	0x6C	R/W	Operand/Result
r28: Accuracy control	0x70	R/W	Number of iterations to do (1-16, default is 16)

8.6 Register details

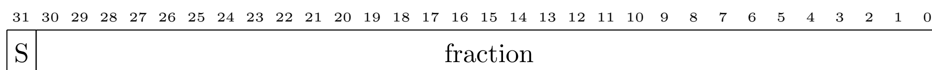
r0 - r3, r16-r19: Operand/Result



The functions of these fields are described in the table below:

Name	**bits**	**R/W**	**Function**
S	31	R/W	result/operand sign bit
integer	30:15	R/W	16-bit integer part of result/operand
fraction	14:0	R/W	15-bit fractional part of result/operand

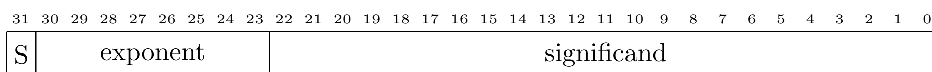
r4-r7, r20-23: Operand/Result



The functions of these fields are described in the table below:

Name	bits	R/W	Function
S	31	R/W	result/operand sign bit
fraction	30:0	R/W	31-bit fractional part of result/operand

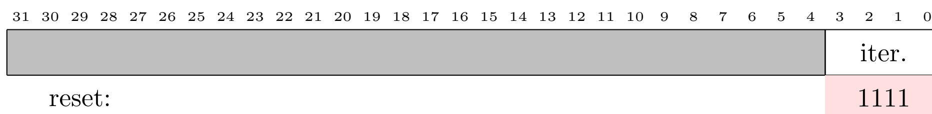
r8-r11, r24-27: Operand/Result



The functions of these fields are described in the table below:

Name	bits	R/W	Function
S	31	R/W	result/operand sign bit
fraction	30:23	R/W	exponent of the result/operand
significand	22:0	R/W	significand of the result/operaand

r28: Accuracy control



The functions of these fields are described in the table below:

Name	bits	R/W	Function
iterations	3:0	R/W	Number of iterations to do = value specified + 1

8.7 Fault-tolerance

Fault insertion

1. The fault-detection software could have an “insert fault” option.

Fault detection

1. Software could test random operand/result pairs from time to time.

Fault isolation

1. If this unit is faulty it should not be used.

Reconfiguration

1. Exponentials can be computed in software, or the functionality migrated to another processor subsystem.

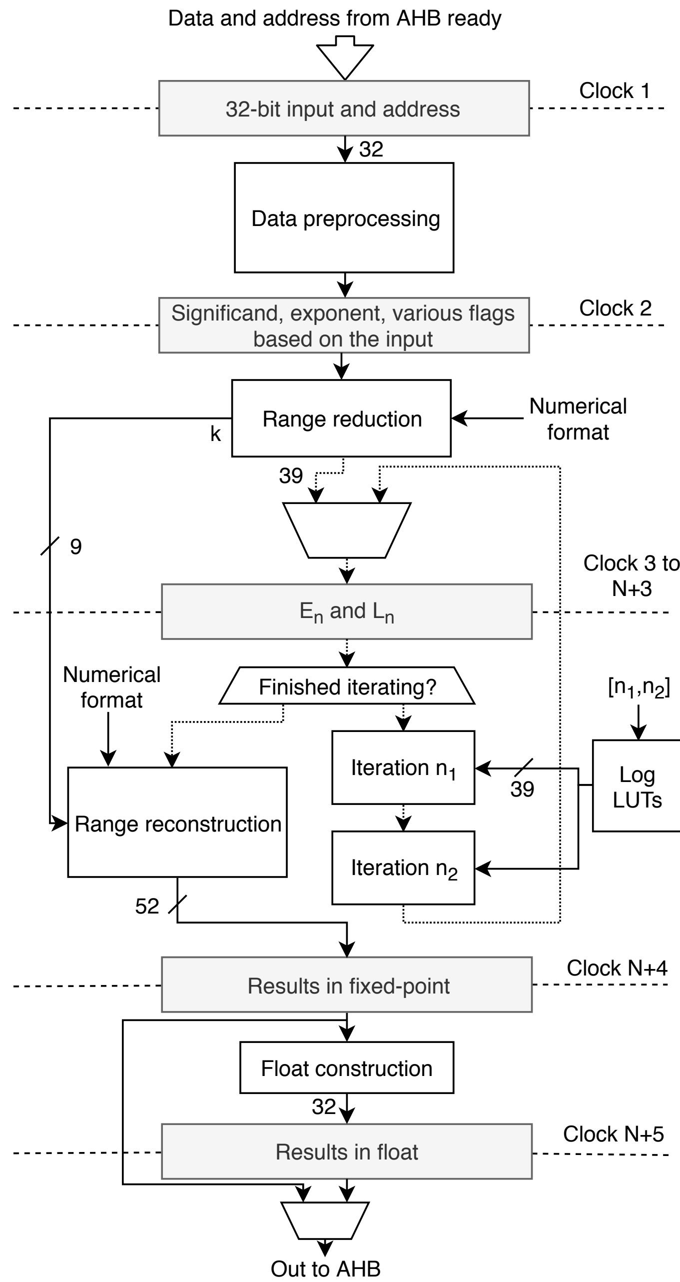


Figure 10: Architecture of the elementary function accelerator.

9 Random Number Generator

A pseudo-random number generator is included as AHB slave within the ARM processor sub-system. It provides high-quality pseudo random number generation and can optionally be randomized by a true random number signal.

True random numbers are generated using the phase frequency detection signals of the ADPLL clock generators. This true random number signal bus is globally available on the chip. It is used for randomization of the pseudo random number generator within the ARM subsystem in the Exec PM.

9.1 Features

1. a dedicated global True Random Number Generator (TRNG) with different ring oscillator architectures and simple postprocessing blocks
2. a set of global Pseudo Random Number Generators (PRNG) for stand alone use or as postprocessing elements for the TRNG
3. a 2-Bit True Random Bus (TRB) for distributing entropy between different sources and different receivers. Sources, that can be coupled onto any of the 2 wires are: PFD signals of all ADPLLs, outputs of oscillators of the TRNG, synchronized or unsynchronized values of the TRB itself.
4. an AHB TRNG slave for each ARM Core reading from the TRB.
5. an AHB PRNG slave for each ARM Core for stand alone use or operation with scrambling by TRB data.

9.2 Description

Pseudo-Random Number Generator (PRNG)

The PRNG gives access to apparently random (though repeatable sequences of) values with a minimal time overhead. Once set up, a 'random' value is provided simply by reading the appropriate register.

Two different PRNG algorithms [** TBC **] are available. These are both KISS ('Keep it Simple Stupid') algorithms for efficiency combined with reasonable 'randomness' and long repeat periods. One is based on 'KISS32', the second on 'KISS64' [** reference/description?? ** ** something on seed meaning/whole algorithm? **].

In addition to this the generators have multiple, concurrent 'seed' values so pseudo-random values from different parts of the cycles can be available; for instance different asynchronous threads or processes can have their own seeds which can provide a 'private' repeatable source.

A value is obtained by reading the appropriate output. This action prompts the hardware to derive the next number for that channel 'behind the scenes' so – unless reads are very close together – there will be no waiting the next time the register is read.

The outputs are available in two forms: the first is as a simple, 32-bit 'integer'; the second takes 24 bits from that value and recodes it into IEEE single precision (32-bit) floating point format. These are alternative representations and reading either causes a new value to be generated.

The floating point output format is appropriately normalised and coded to represent equally spaced values x such that $1 > x >= 0$ (i.e. a separation of approx. 6.10^{-8}) with the proviso that the '0' value is replaced with the (denormalised) minimum positive value (approx. 10^{-38}).

To initialise/reset the PRNG the various seed values for that sequence must first be written. It is then necessary to read the corresponding output register (discarding the value) to cause the seed to be used by the generator.

??? KISS32 sequences and ??? KISS64 sequences are available in each PRNG [** both TBC **].

True Random Number Generator

*** TO BE DONE ***

9.3 Register summary

Base address: 0x20000000.

Name	Offset	R/W	Function
r0: k32_0 s_0	0x00	R/W	kiss32_0 seed 0
r1: k32_0 s_1	0x04	R/W	kiss32_0 seed 1
r2: k32_0 s_2	0x08	R/W	kiss32_0 seed 2
r3: k32_0 s_3	0x0C	R/W	kiss32_0 seed 3
r4: k32_0 Rint	0x10	R	kiss32_0 random INT
r5: k32_0 Rflt	0x14	R	kiss32_0 random Float32
r6: k32_0 C	0x18	R/W	kiss32_0 seed carry
r8: k32_1 s_0	0x20	R/W	kiss32_1 seed 0
r9: k32_1 s_1	0x24	R/W	kiss32_1 seed 1
r10: k32_1 s_2	0x28	R/W	kiss32_1 seed 2
r11: k32_1 s_3	0x2C	R/W	kiss32_1 seed 3
r12: k32_1 Rint	0x30	R	kiss32_1 random INT
r13: k32_1 Rflt	0x34	R	kiss32_1 random Float32
r14: k32_1 C	0x38	R/W	kiss32_1 seed carry
r15: k64 s_0	0x40	R/W	kiss64 seed 0
r17: k64 s_1	0x44	R/W	kiss64 seed 1
r18: k64 s_2	0x48	R/W	kiss64 seed 2
r19: k64 s_3	0x4C	R/W	kiss64 seed 3
r20: k64 Rint	0x50	R	kiss64 random INT

9.4 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

10 Stochastic Rounding Accelerator

Accelerator for performing stochastic rounding of numbers to a specified bit position is included as an AHB slave within each ARM Cortex M4F processor sub-system.

10.1 Features

1. Rounding and saturation of 64, 32 and 16-bit values
2. Stochastic (using random numbers from MARS KISS 64) or round-to-nearest modes
3. Signed and unsigned arithmetic support
4. Configurable bit position to round (1 to 32 bits)
5. Support for single-precision floating-point to BFLOAT16 rounding
6. Unit works in 3 or 4 cycles (1 or 2 cycles for data input, 1 for rounding and 1 for output)

10.2 Description

Integer rounding and saturation is supported for 64-bit → 32-bit, 32-bit → 32-bit, 32 → 16-bit and 16-bit → 16-bit. Both signed and unsigned arithmetic is supported. Rounding is RTN (round-up on tie) and stochastic rounding using random numbers from MARS KISS 64. Bit position for rounding can be specified (1 to 32). Additionally, FLOAT → BFLOAT16 rounding and saturation is also supported. Unit works in 3 or 4 cycles (1 or 2 cycles for data input, 1 for rounding and 1 for output). For 64-bit inputs, little-endian sequence of word arrival is assumed. 4 threads are supported - both for using a separate PRNG channels in stochastic rounding and for preserving data in case interrupt happens before outputs are read. Input to the unit are written to AHB address 0xE0100600 plus an offset depending on the arithmetic and rounding mode, as shown below. Results stored in the output registers are read from the same AHB address.

10.3 Implementation

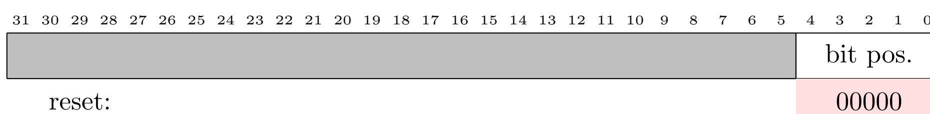
Figure 11 shows an architectural diagram of this accelerator. When the input number is ready, accelerator reads the configuration register which contains the number of bits to round. Residual is found and stochastic rounding is performed by adding a pseudo-random number from the appropriate channel of the MARS_KISS64 pseudo-random number generator, and then adding a carry-out to the value that will be returned. If round-to-nearest is requested, the most significant bit of the residual is added to the returned value. After rounding, saturation is performed by checking the top bits, taking into account whether signed or unsigned arithmetic is used (which is encoded in the address). Rounding mode is also encoded in the address.

10.4 Register summary

Base address: 0xE0100600. Tables 6, 7 and 8 contain registers available for using this accelerator. Encoding of the register naming is as follows: rounding mode (RN/SR), signed/unsigned arithmetic, bit width of the input (64/32/16), bit width of the output (32/16), top or bottom part of the 64-bit input (not required for 32 and 16-bit inputs) and thread number (0-3).

10.5 Register details

r88: Configuration of bits to round



reset: 00000

The functions of these fields are described in the table below:

Name	bits	R/W	Function
bits to round	4:0	R/W	Number of bits to round = value specified + 1

10.6 Fault-tolerance

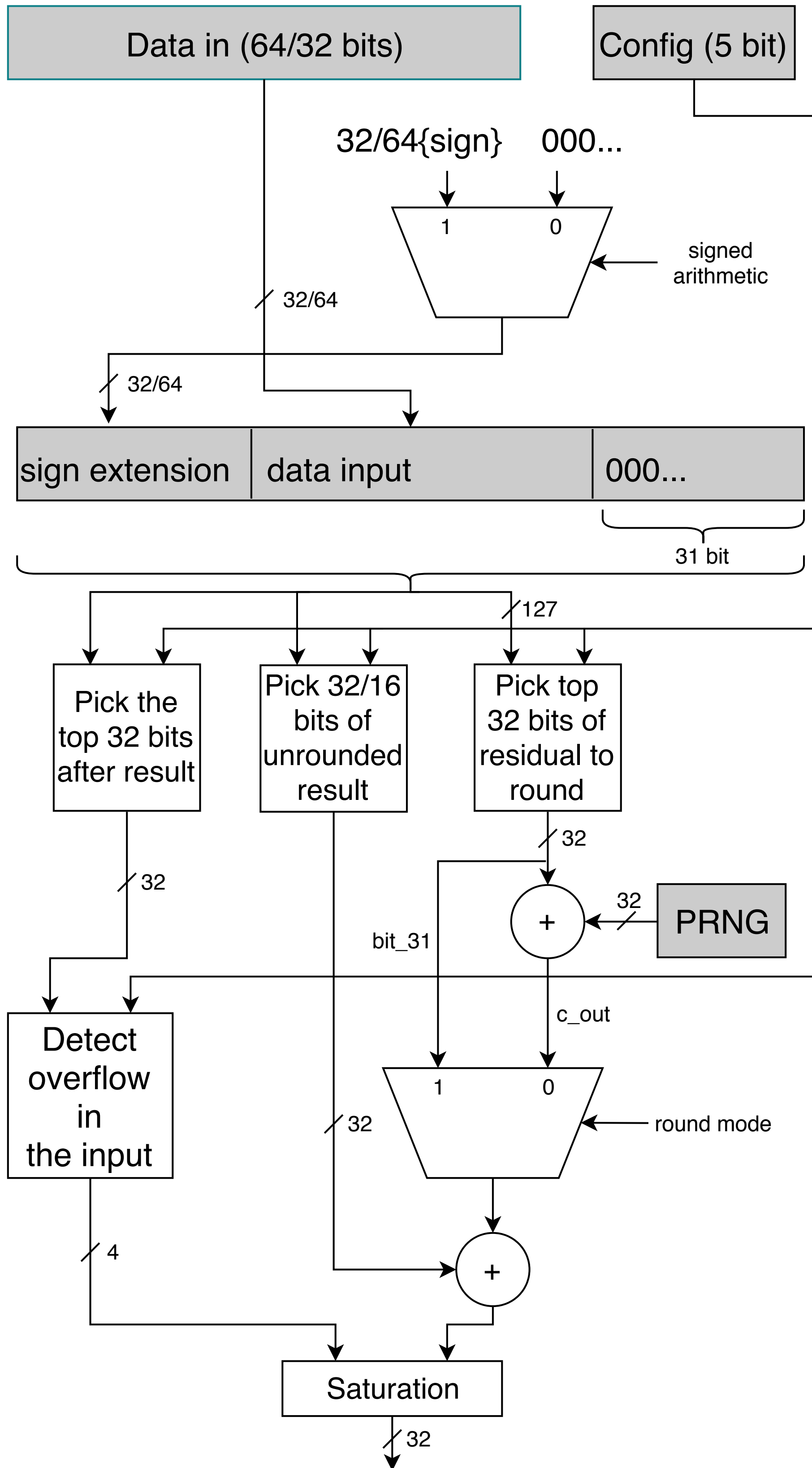


Figure 11: Architecture of the rounding accelerator.

Table 6: First set of registers of the rounding accelerator. Stochastic rounding (SR) registers are shown for all numerical formats.

Name	**Offset**	**R/W**	**Function**
r0: SR_U32_32 0	0x00	R/W	Operand/Result
r1: SR_U32_32 1	0x04	R/W	Operand/Result
r2: SR_U32_32 2	0x08	R/W	Operand/Result
r3: SR_U32_32 3	0x0C	R/W	Operand/Result
r4: SR_U32_16 0	0x10	R/W	Operand/Result
r5: SR_U32_16 1	0x14	R/W	Operand/Result
r6: SR_U32_16 2	0x18	R/W	Operand/Result
r7: SR_U32_16 3	0x1C	R/W	Operand/Result
r8: SR_U16_16 0	0x20	R/W	Operand/Result
r9: SR_U16_16 1	0x24	R/W	Operand/Result
r10: SR_U16_16 2	0x28	R/W	Operand/Result
r11: SR_U16_16 3	0x2C	R/W	Operand/Result
r12: SR_S32_32 0	0x30	R/W	Operand/Result
r13: SR_S32_32 1	0x34	R/W	Operand/Result
r14: SR_S32_32 2	0x38	R/W	Operand/Result
r15: SR_S32_32 3	0x3C	R/W	Operand/Result
r16: SR_S32_16 0	0x40	R/W	Operand/Result
r17: SR_S32_16 1	0x44	R/W	Operand/Result
r18: SR_S32_16 2	0x48	R/W	Operand/Result
r19: SR_S32_16 3	0x4C	R/W	Operand/Result
r20: SR_S16_16 0	0x50	R/W	Operand/Result
r21: SR_S16_16 1	0x54	R/W	Operand/Result
r22: SR_S16_16 2	0x58	R/W	Operand/Result
r23: SR_S16_16 3	0x5C	R/W	Operand/Result
r24: SR_FLOAT_BFLOAT16 0	0x60	R/W	Operand/Result
r25: SR_FLOAT_BFLOAT16 1	0x64	R/W	Operand/Result
r26: SR_FLOAT_BFLOAT16 2	0x68	R/W	Operand/Result
r27: SR_FLOAT_BFLOAT16 3	0x6C	R/W	Operand/Result

Table 7: Second set of registers of the rounding accelerator. Rounding to nearest (RN) registers are shown for all numerical formats.

Name	Offset	R/W	Function
r28: RN_U32_32 0	0x70	R/W	Operand/Result
r29: RN_U32_32 1	0x74	R/W	Operand/Result
r30: RN_U32_32 2	0x78	R/W	Operand/Result
r31: RN_U32_32 3	0x7C	R/W	Operand/Result
r32: RN_U32_16 0	0x80	R/W	Operand/Result
r33: RN_U32_16 1	0x84	R/W	Operand/Result
r34: RN_U32_16 2	0x88	R/W	Operand/Result
r35: RN_U32_16 3	0x8C	R/W	Operand/Result
r36: RN_U16_16 0	0x90	R/W	Operand/Result
r37: RN_U16_16 1	0x94	R/W	Operand/Result
r38: RN_U16_16 2	0x98	R/W	Operand/Result
r39: RN_U16_16 3	0x9C	R/W	Operand/Result
r40: RN_S32_32 0	0xA0	R/W	Operand/Result
r41: RN_S32_32 1	0xA4	R/W	Operand/Result
r42: RN_S32_32 2	0xA8	R/W	Operand/Result
r43: RN_S32_32 3	0xAC	R/W	Operand/Result
r44: RN_S32_16 0	0xB0	R/W	Operand/Result
r45: RN_S32_16 1	0xB4	R/W	Operand/Result
r46: RN_S32_16 2	0xB8	R/W	Operand/Result
r47: RN_S32_16 3	0xBC	R/W	Operand/Result
r48: RN_S16_16 0	0xC0	R/W	Operand/Result
r49: RN_S16_16 1	0xC4	R/W	Operand/Result
r50: RN_S16_16 2	0xC8	R/W	Operand/Result
r51: RN_S16_16 3	0xCC	R/W	Operand/Result
r52: RN_FLOAT_BFLOAT16 0	0xD0	R/W	Operand/Result
r53: RN_FLOAT_BFLOAT16 1	0xD4	R/W	Operand/Result
r54: RN_FLOAT_BFLOAT16 2	0xD8	R/W	Operand/Result
r55: RN_FLOAT_BFLOAT16 3	0xDC	R/W	Operand/Result

Table 8: Third set of registers of the rounding accelerator. 64-bit SR and RN registers are shown for all numerical formats as well as control register.

Name	**Offset**	**R/W**	**Function**
r56: SR_U64_32_bottom 0	0xE0	R/W	Operand/Result
r57: SR_U64_32_top 0	0xE4	R/W	Operand/Result
r58: SR_U64_32_bottom 1	0xE8	R/W	Operand/Result
r59: SR_U64_32_top 1	0xEC	R/W	Operand/Result
r60: SR_U64_32_bottom 2	0xF0	R/W	Operand/Result
r61: SR_U64_32_top 2	0xF4	R/W	Operand/Result
r62: SR_U64_32_bottom 3	0xF8	R/W	Operand/Result
r63: SR_U64_32_top 3	0xFC	R/W	Operand/Result
r64: SR_S64_32_bottom 0	0x100	R/W	Operand/Result
r65: SR_S64_32_top 0	0x104	R/W	Operand/Result
r66: SR_S64_32_bottom 1	0x108	R/W	Operand/Result
r67: SR_S64_32_top 1	0x10C	R/W	Operand/Result
r68: SR_S64_32_bottom 2	0x110	R/W	Operand/Result
r69: SR_S64_32_top 2	0x114	R/W	Operand/Result
r70: SR_S64_32_bottom 3	0x118	R/W	Operand/Result
r71: SR_S64_32_top 3	0x11C	R/W	Operand/Result
r72: RN_U64_32_bottom 0	0x120	R/W	Operand/Result
r73: RN_U64_32_top 0	0x124	R/W	Operand/Result
r74: RN_U64_32_bottom 1	0x128	R/W	Operand/Result
r75: RN_U64_32_top 1	0x12C	R/W	Operand/Result
r76: RN_U64_32_bottom 2	0x130	R/W	Operand/Result
r77: RN_U64_32_top 2	0x134	R/W	Operand/Result
r78: RN_U64_32_bottom 3	0x138	R/W	Operand/Result
r79: RN_U64_32_top 3	0x13C	R/W	Operand/Result
r80: RN_S64_32_bottom 0	0x140	R/W	Operand/Result
r81: RN_S64_32_top 0	0x144	R/W	Operand/Result
r82: RN_S64_32_bottom 1	0x148	R/W	Operand/Result
r83: RN_S64_32_top 1	0x14C	R/W	Operand/Result
r84: RN_S64_32_bottom 2	0x150	R/W	Operand/Result
r85: RN_S64_32_top 2	0x154	R/W	Operand/Result
r86: RN_S64_32_bottom 3	0x158	R/W	Operand/Result
r87: RN_S64_32_top 3	0x15C	R/W	Operand/Result
r88: Bits to round	0x160	R/W	Operand/Result

11 Machine Learning Accelerator (MLA)

11.1 Features

1. 16x4 Broadcast MAC array
2. 8bit unsigned integer multiplication
3. 29bit accumulation registers with saturation at maximum value
4. Get sources from and write target to AHB/NoC
5. Prefetch data from NoC to avoid waiting
6. Order of dimension loops for convolution adapted, so that addition of sub-elements not necessary
7. Maximum throughput:
   (a) Matrix multiplication:
      i. Operand A: 128bit/clk
      ii. Operand B: 128bit/4*clk
   (b) Convolution:
      i. Operand A: 128bit/clk (within loop #0: 32bit/4*clk)
      ii. Operand B: 128bit/4*clk

11.2 Overview

The accelerator lies within the communication control module (CommsCtrl) and provides a SIMD execution of matrix-matrix multiplication (MM) and 2d convolution (CONV). For now the precision of the input for both operations is 8bit but other resolutions are planned for future iterations. To control the accelerator a AHB slave connection is provided as connection to the ARM core within the PE. The accelerator accesses its input (MM: Matrix A, Matrix B, CONV: Filter/Weights Tensor A, Input Feature Map Tensor B) either from the local SRAM as AHB master within the same PE or from external sources over NoC. It then writes the result in the local SRAM. The complete operation (convolution or matrix multiplication) is performed by the block autonomously and throws an interrupt if the operation finished either successfully or with an positive error output signal. Required configuration and status information are listed in the following subsections.

11.3 Configuration and Command Registers

Configuration Word	Description
source_a	Source type (2 MSB) and address of operand A
source_b	Source type (2 MSB) and address of operand B
target	Address of result
rows_a	MM: Number of rows for operand A
cols_a_rows_b	MM: Number of columns for operand A, and number of rows of operand B
cols_b	MM: Number of columns for operand B
trigger_command	Execute Accelerator: 1- MM / 2- CONV
data_a*	Data for operand A, written via AHB
conv_params*	CONV: convolution parameters

Convolution Parameter	Description
in_batch_s	number of examples in one minibatch
in_pix_row_s	input feature map x dimension (horizontal)
in_pix_col_s	input feature map y dimension (vertical)
fil_pix_row_s	filter x dimension (horizontal)
fil_pix_col_s	filter y dimension (vertical)
in_channel_s	input channel (depth) size
out_channel_s	output channel (depth) size
stride_in_pix_row	horizontal stride
stride_in_pix_col	vertical stride
padding_mode	0- valid / 1- same / 2- manual

11.4 Mode of Operation

The accelerator assumes that the input data lies in correct layout within the addresses it is provided with. This subsection describes the process of execution with the help of the accelerator for both matrix multiplication and 2D convolution in detail. To control the accelerator over ARM core see 11.5.

11.4.1 Matrix Multiplication

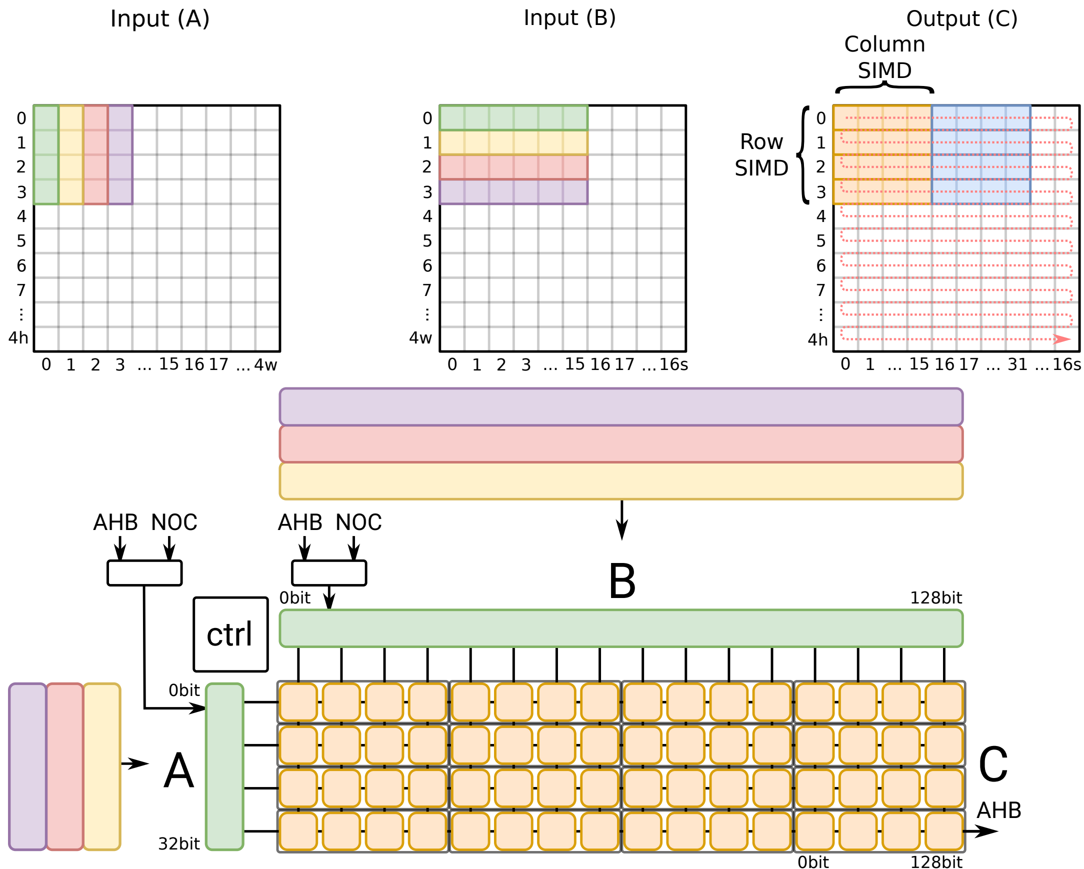


Figure 12: Matrix multiplication execution with the help of the accelerator.

The operation for matrix multiplication is fairly simple. The shape of operand A is defined as $(4h, 4w)$ while the shape of operand B is $(4w, 16s)$. Those shapes are the result of the internal structure of the MAC array and the memory alignment or data access of the spinnaker2 system. To mitigate memory accesses the shared dimension is treated as the inner loop.

11.4.2 Convolution

For the functionality of the accelerator look at the tensorflow implementation quantized_conv2d. For a good explanation of the convolution process (or cross-correlation) visit the Stanford course CS231. Input A are the weight/filters and Input B are the input feature maps. The feature maps must lie within the same PE as the accelerator, because they are only fetchable with the SRAM AHB port. The weights are prefetched by NoC. The order of execution is dictated by the convolution parameters with the exception of stride and padding (see 11.3). They span a 7-dimensional loop structure in the order described in table 9 with filter column as the innermost loop and minibatch size as the outer loop.

In the beginning specific parameters get calculated out of the provided convolution parameters (e.g. padding size for if VALID padding mode active). After the first data arrives at the accelerator the array executes one 16x4 multiply-accumulate operation. During loop #0 operator B, one input feature map row, is shifted for the next mac step and new weights are loaded as operator A. How

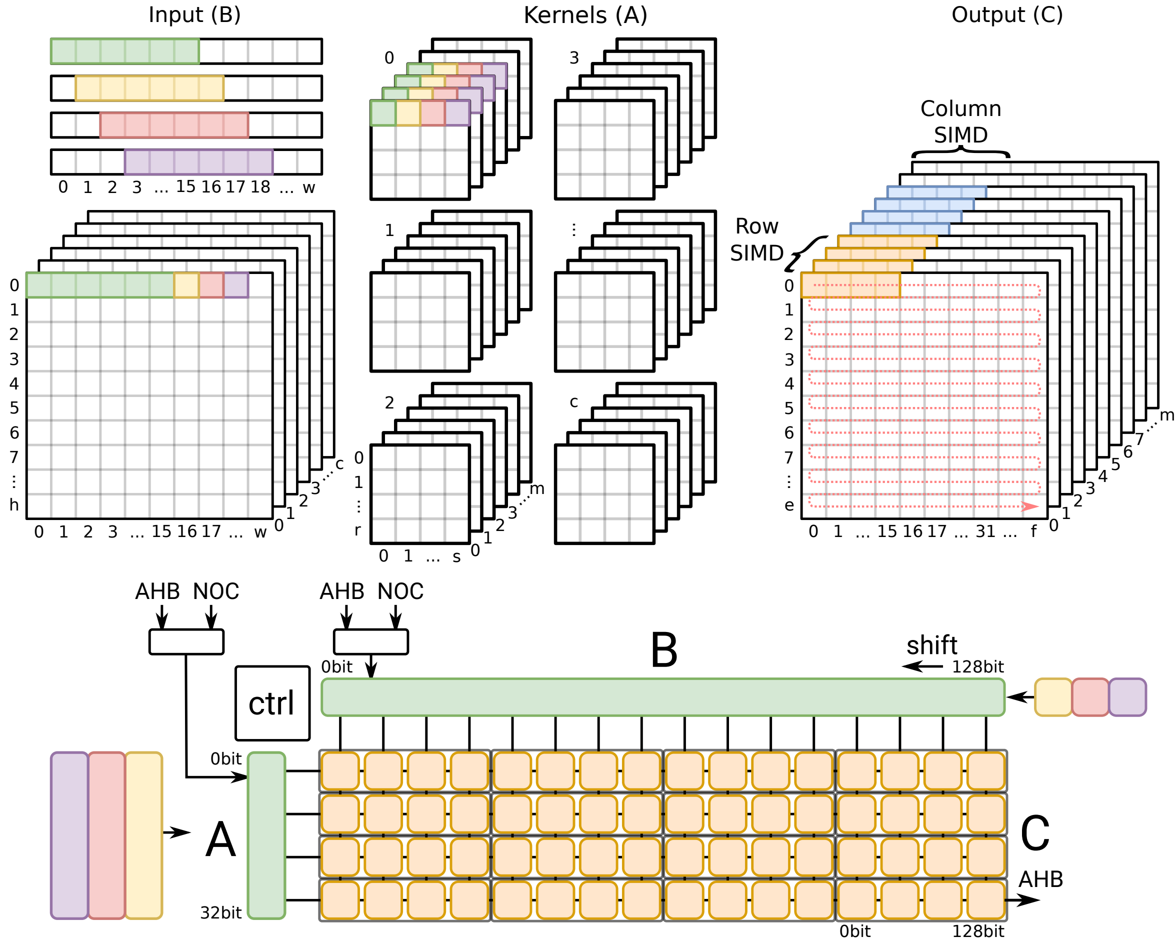


Figure 13: 2D Convolution execution with the help of the accelerator. The order of dimension loops are described in table 9.

often we fetch operator A depends on the element precision and the row count of the MAC array. If 8bit precision is used and the MAC array uses 4 rows one 128bit fetch can be used 4 times. The loop structure then calculates where to read the next relevant data with address offsets loop_inc. If the counter of loop #3 (output depth) increments the array is written into the target memory block (4 values with 128 bit output). The SIMD aspect of the mac array is bound to the output depth (M) of the array rows (4 simultaneous output channels) and to the filter row (S) of the array columns (16 simultaneous filter columns).

11.5 ARM C code & Execution

Since the accelerator is controlled by the ARM core we provide a C library, called ml-lib, as interface to send the correct signals. It is furthermore worth noting, that any device, having access to the shared NoC, may control any accelerator within the network. The functions listing 1 and listing 2 are of interest. See table 10 for the description of the parameters for matrix multiplication and table 11 for the function parameters of *execute_conv*.

```
bool execute_mm(uint32_t dim_row_a, uint32_t dim_col_a, uint32_t dim_col_b,
    uint32_t op_a_addr, uint32_t op_b_addr, uint32_t target_addr, uint16_t
    op_a_modid, bool op_a_use_noc);
```

Listing 1: ARM core function to exectute matrix multiplication with the accelerator

```
execute_conv(const struct conv_params cparams, uint32_t fmaps_addr, uint32_t
    weights_addr, uint16_t weights_modid, uint32_t target_addr);
```

Listing 2: ARM core function to exectute 2D convolution with the accelerator

loop	dimension	symbol	notes
#0	filter column	S	shift, SIMD x16 (array column)
#1	filter row	R	
#2	input depth	C	
#3	output depth	M	write back, SIMD x4 (array row)
#4	output column	F	
#5	output row	E	
#6	minibatch size	N	

Table 9: Loop order of the execution of the accelerator during 2d convolution.

parameter	description
dim_row_a	Number of row elements of matrix A
dim_col_a	Number of column elements of matrix A and number of row elements of matrix B
dim_col_b	Number of column elements of matrix B
op_a_addr	Address of first element of matrix A within hardware
op_b_addr	Address of first element of matrix B within hardware
target_addr	Address of first element of result matrix C within hardware
op_a_modid	If matrix A is fetched from NoC, this id specifies the used SRAM/DRAM block
op_a_use_noc	Set true if matrix A should be fetched with NoC

Table 10: Function parameters of *execute_mm*.

parameter	description
cparams	Struct of all necessary convolution parameters
fmaps_addr	Address of first element of the input feature map
weights_addr	Address of first element of the weight tensor
weights_modid	If the weight tensor is fetched from NoC, this id specifies the used SRAM/DRAM block
weights_addr	Address of first element of the resulting output tensor

Table 11: Function parameters of *execute_conv*.

```
uint8_t quad = 0x9; // quad = 001001 -> (1,1)
uint8_t pe = 1; // pe within quad
execute_mm(4, 5, 3, 0x120, 0x10000, 0x18000, (0x0000 | quad << 5 | 1 << ((2-pe) %
    4)), 1);
```

Listing 3: Example of one matrix multiplication executed by the internal ARM core

```
uint8_t quad = 0x9; // quad = 001001 -> (1,1)
uint8_t pe = 1; // pe within quad
struct conv_params cparams1 = CPARAMS_DEFAULT;
cparams1.in_batch_s         = 1;
cparams1.in_pix_row_s       = 4;
cparams1.in_pix_col_s       = 18;
cparams1.fil_pix_row_s      = 3;
cparams1.fil_pix_col_s      = 3;
cparams1.in_channel_s       = 1;
cparams1.out_channel_s      = 4;
cparams1.stride_in_pix_row  = 1;
cparams1.stride_in_pix_col  = 1;
cparams1.padding_mode       = PADDING_VALID;
execute_conv(cparams1, 0x8040, 0x8080, (0x0000 | quad << 5 | 1 << ((2-pe) % 4)), 0
    x80C0);
```

Listing 4: Example of one convolution executed by the internal ARM core

12 Counter/timer

Each processor element on a SpiNNaker chip has 4 counter/timers.

The counter/timers use two copied of the standard AMBA peripheral device described on page 4-24 of the AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003. The peripheral has been modified only in that the APB interface of the original has been replaced by an AHB interface for direct connection to the ARM968 AHB bus.

12.1 Features

1. the counter/timer unit provides two independent counters, for example for:
   (a) millisecond interrupts for real-time dynamics.
2. free-running and periodic counting modes:
   (a) automatic reload for precise periodic timing;
   (b) one-shot and wrapping count modes.
3. the counter clock (which runs at the processor clock frequency) may be pre-scaled by dividing by 1, 16 or 256.

12.2 Register summary

Base address: 0x21000000 (buffered write), 0x11000000 (unbuffered write).

User registers

The following registers allow normal user programming of the counter/timers:

Name	Offset	R/W	Function
r0: Timer1load	0x00	R/W	Load value for Timer 1
r1: Timer1value	0x04	R	Current value of Timer 1
r2: Timer1Ctl	0x08	R/W	Timer 1 control
r3: Timer1IntClr	0x0C	W	Timer 1 interrupt clear
r4: Timer1RIS	0x10	R	Timer 1 raw interrupt status
r5: Timer1MIS	0x14	R	Timer 1 masked interrupt status
r6: Timer1BGload	0x18	R/W	Background load value for Timer 1
r8: Timer2load	0x20	R/W	Load value for Timer 2
r9: Timer2value	0x24	R	Current value of Timer 2
r10: Timer2Ctl	0x28	R/W	Timer 2control
r11: Timer2IntClr	0x2C	W	Timer 2interrupt clear
r12: Timer2RIS	0x30	R	Timer 2raw interrupt status
r13: Timer2MIS	0x34	R	Timer 2masked interrupt status
r14: Timer2BGload	0x38	R/W	Background load value for Timer 2

Test and ID registers

In addition, there are test and ID registers that will not normally be of interest to the programmer:

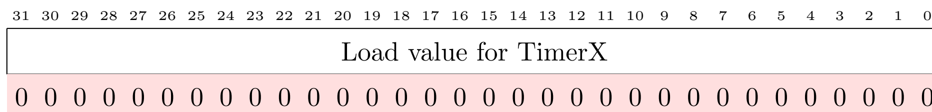
Name	Offset	R/W	Function
TimerITCR	0xF00	R/W	Timer integration test control register
TimerITOP	0xF04	W	Timer integration test output set register
TimerPeriphID0-3	0xFE0-C	R	Timer peripheral ID byte registers
TimerPCID0-3	0xFF0-C	R	Timer Prime Cell ID byte registers

See AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003, for further details of the test and ID registers.

12.3 Register details

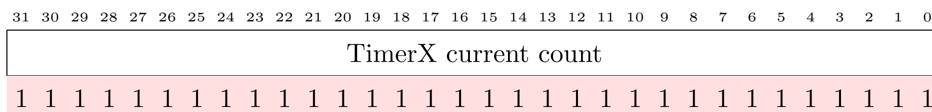
As both timers have the same register layout they can both be described as follows (X = 1 or 2):

r0/8: Timer X load value



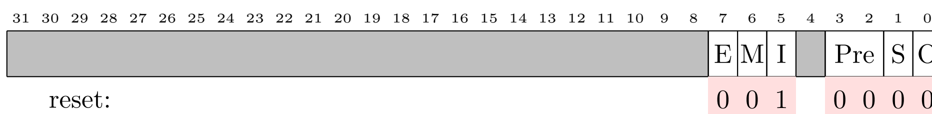
When written, the 32-bit value is loaded immediately into the counter, which then counts down from the loaded value. The background load value (r6/14) is an alternative view of this register which is loaded into the counter only when the counter next reaches zero.

r1/9: Current value of Timer X



This read-only register yields the current count value for Timer X.

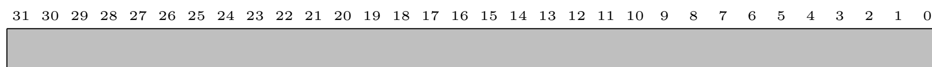
r2/10: Timer X control



The shaded fields should be written as zero and are undefined on read. The functions of the remaining fields are described in the table below:

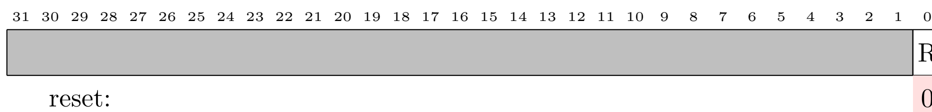
Name	**bits**	**R/W**	**Function**
E: Enable	7	R/W	enable counter/timer (1 = enabled)
M: Mode	6	R/W	0 = free-running; 1 = periodic
I: Int enable	5	R/W	enable interrupt (1 = enabled)
Pre: TimerPre	3:2	R/W	divide input clock by 1 (00), 16 (01), 256 (10)
S: Timer size	1	R/W	0 = 16 bit, 1 = 32 bit
O: One shot	0	R/W	0 = wrapping mode; 1 = one shot

r3/11: Timer X interrupt clear



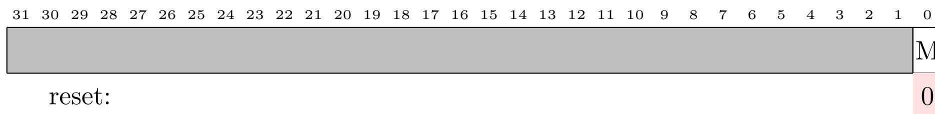
Any write to this address will clear the interrupt request.

r4/12: Timer X raw interrupt status



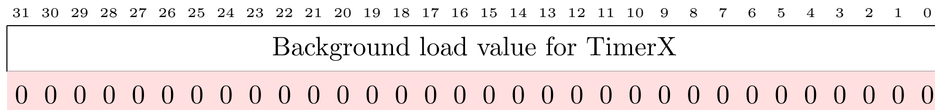
Bit zero yields the raw (unmasked) interrupt request status of this counter/timer.

r5/13: Timer X masked interrupt status



Bit zero yields the masked interrupt status of this counter/timer.

r6/14: Timer X background load value



The 32-bit value written to this register will be loaded into the counter when it next counts down to zero. Reading this register will yield the same value as reading register 0/8.

12.4 Fault-tolerance

Fault insertion

1. Disabling a counter (by clearing the E bit in its control register) will cause it to fail in its function.

Fault detection

1. Use the second counter/timer with a longer period to check the calibration of the first?

Fault isolation

1. Disable the counter/timer with the E bit in the control register; disable its interrupt output; disable the interrupt in the interrupt controller.

Reconfiguration

1. If one counter fails then a system that requires only one counter can use the other one.

13 Exchange - the PE communications switch

Each PE on SpiNNaker2 includes a communications Exchange which is responsible for transmitting and receiving packets to and from the communications network. This block contains all the NoC-connected units, which are outlined in this section and described in greater detail in their own sections.

Incoming packets may be routed directly to local SRAM using a dedicated IO DMA controller, in which case they do not appear in the 'default' receiver path below.

In addition, this unit handles some other packet functions – such as providing remote access to the PE SRAM – which are not directly visible at the software interface.

13.1 Features

1. Transparent bus bridge onto global address space via the NoC.
   (a) The PE processor's address space can be expanded to encompass the whole chip plus its accompanying SDRAM(s) with transparent load and store.
   (b) Write accesses may be fire-and-forget for improved performance.
2. NoC packet transmitter and receiver with software interface.
   (a) The transmitter allows any sort of NoC packet to be sent. The typical common purposes are:
      i. SpiNNaker {MC, C2C, NN} packet transmission to router.
      ii. Message/protocol packet transmission to other PEs.
      iii. Raising exceptions with other PEs.
      iv. Software, *non-blocking* request to read/write data in a remote PE.
   (b) The receiver includes DMA support for cyclic receive buffers in local SRAM; selectively on type, all of or partial incoming packets can be written to a cyclic buffer in local address space without software/interrupt intervention.
   (c) Packets not filtered to DMA are queued for software intervention; a selection of identifying interrupts (according to type) is available.
3. Memory-memory DMA controller for transfers to/from local RAM.
4. The Machine Learning Accelerator (MLA) can be fully controlled via NoC.
5. Exception receiver. Interrupts and resets on the PE can be asserted remotely via NoC packets without software intervention.
6. Bridge from NoC *into* local SRAM. This makes the local SRAM available as a globally addressable resource. This facility is independent from the local PE processor.

13.2 Overview

As shown in Fig. 14, the Exchange includes a number of sub-units.

Unit	Function(s)
Bridge	Allows System AHB to be extended as a master over the NoC.
Comms.	Provides software to packet transmit and receive interfaces; also has DMA interfaces into local bus.
DMA	Main memory-memory DMA unit for moving data blocks into and out of local RAM.
ML accelerator	MAC array to accelerate matrix multiplication and 2d-convolution.
Response	A bus bridge from the NoC *onto* the local PE RAM.

13.2.1 NoC interface

After the Tx FIFO, a multiplexer is employed to arbitrate sending requests from the Bridge (Peek/Poke), Tx FIFO, DMAm, ML-Acc and RespPkt. The priorities of the 5 are listed below from highest to lowest.

1. RespPkt
2. Bridge (Peek/Poke)

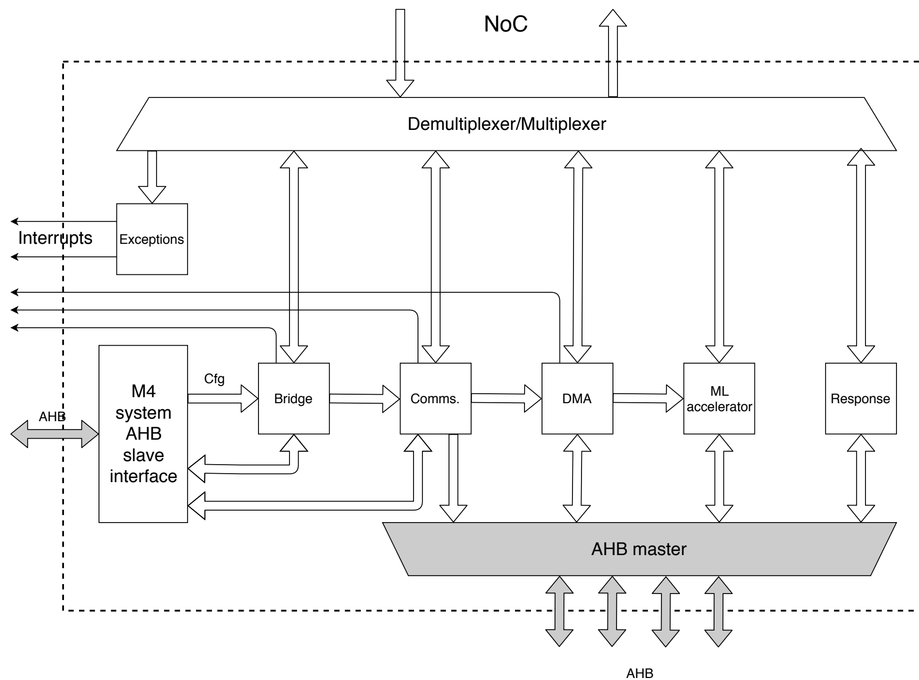


Figure 14: Exchange sub-units

3. Tx FIFO
4. DMAm
5. ML-Acc

The receiver is a demultiplexer to five NoC receiving submodules and a bus interface. The demultiplexer redirects input packets to these submodules according to their types.

1. **Exception** inputs.
2. Bridge (Peek/Poke) returned **responses**.
3. DMA returned **responses**.
4. ML-Accelerator DETAILS REQUIRED.
5. General, software packets including **SpiNNaker packets** and software messages.
   Some of these packets may be filtered and DMA-ed directly into the local SRAM; others are left to software.
6. Remote **requests** for local bus operations.

13.2.2 Bus master interface

To provide sufficient bandwidth to the local SRAM it is divided into two 64_KiB banks (using A[16]) which are each subdivided onto four word-interleaved buffers (using A[3:2]). There are four units which can be SRAM masters. To simplify the subsequent switch these are decoded (using A[3:2]) into parallel word-lanes and distributed appropriately to subunits of the crossbar switch Q.V. – cross reference. Each of the units which can be a bus master {comms, DMA, ML-accelerator, response} is capable of outputting a bus request on (up to) all the 32-bit AHB word lanes in a single cycle. See Fig. 15.

There is some additional alignment and sequencing needed when up to six, unaligned words are being saved by the comms unit.

The address interleaving is supported by the data in three of the four sources **being aligned at source (???)**; only the input from the comms unit, where packet length may vary and not all fields may, necessarily, be saved, causes some consternation; this is addressed within the comms unit boundary. *** There's still an issue of lane-swapping though. Note that a single word payload is always in NoC lane 3 (UofM numbering) whilst its address is arbitrary. ***

To reduce the latency of the critical path, which is the arbiter before each memory bank, a stage of pipeline registers are inserted between the 6-to-1 arbitration and crossbar (designed by Delong) before each memory bank. Because the two SpiNNaker DMAs only write data to local memory, do not read data, the pipeline stage doesn't lead to any performance decrease.

The DMAw_read registers are also implemented in the pipeline stage. DMAw_read stands for the address that finished writing. There are two DMAw_read registers, one for DMAs0 and the other for DMAs1. When a word is forwarded to the pipeline stage, a flag indicating the word belonging to DMAs0 or DMAs1 is also passed on. When the word is finished writing, the corresponding DMAw_read register increases by one. However, this increase is only a temporary increase.

It is possible that words belonging to one SpiNNaker Packet DMA finish writing out of order. Therefore, the increases at the 4 ports are stored temporarily, and DMAw_read always checks the 4 ports sequentially. That is, if DMAw_read starts from port 1, it checks port 1, then port 2, then port 3, then port 0, the port 1 ... When it checks the port, if there are temporary increases, the temporary increase will be decreased by one. If there is no temporary increase, it waits until an increase comes.

Attempt to summarise previous paragraphs: check for correctness.

The flow through the interface [for which units? All? Are there consequences for MLA?] is pipelined. It is essential that memory modifications are seen *deterministically* by the processor. In the comms unit (q.v. – cross reference) there are DMA registers {`spDMAw0`, `spDMAw1`} which indicate the position reached in writing to the ring-buffer. These are not updated until it is known the write is complete. Write operations may complete out of order across the different, interleaved ports; the `spDMAw*` registers, as read, are only updatedwhen *all* the write operations up to that point are known to be complete.

*** Don't have confirmation of such implications for other subunits at this time. ***

13.3 Exception subunit

The exception subunit is responsible for handling incoming contro/exception packets from the NoC. There are two classes of exception packets {reset, interrupt} which are dealt with in hardware and software, respectively.

13.3.1 Interrupt exceptions

An arriving exception packet can set one or more local interrupt requests. These requests remain set until actively cleared by software. Clear is effected by writing a '1' to the appropriate bit; writing a '0' has no effect. The processor cannot set these bits directly (although it could send itself an appropriate NoC packet).

IFF all the bits go directly to the VIC – which is not that unlikely – then this *could* be aliased with the general interrupt status register. In this case only the bits set from the external NoC packets need (and probably '*should*') be clearable locally.

13.4 Register summary

Base address: 0x????????

Access permissions? Should these be privileged access only?

Name	Offset	R/W	Function
r0: XXX	0x000	R/W	Overall control register
r1: YYY	0x004	R	Status register
r2: IEN	0x008	R/W	Interrupt enable register
r?: IRQ	0x???	R/W*	Global interrupt status register
r?: ZZZ	0x???	R/?	Global source register(s)

*** 32-bits of status may be inadequate ***

13.5 Register details

r0: XXX - Control register

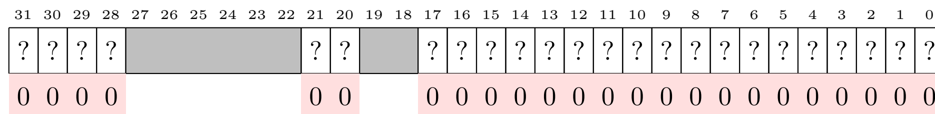
Enables, resets etc. Blah, blah.

r1: YYY - Status register

Consider ARM immediates when placing bits!

This is a suggested plan for the comms. interface interrupts. It does not take into account the memDMA etc. Nor does it encompass the system-level exception inputs.

Potential comms interrupts (possibly out of date):



May need to accommodate **more memDMA interrupts**.
Needs to accommodate **more MLA interrupts.**
Needs to accommodate **more (or fewer!) NoC interrupts.**

The functions of these fields are described in the table below:

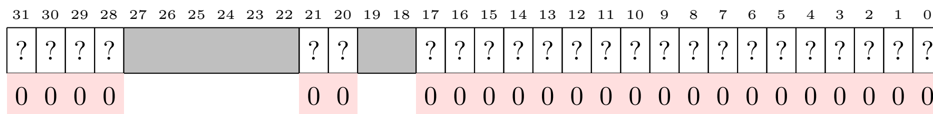
Name	bits	R/W	Function
?	31	R	IRQ_external_#3
?	30	R	IRQ_external_#2
?	29	R	IRQ_external_#1
?	28	R	IRQ_external_#0
?	21	R	IRQ_MLA_#1
?	20	R	IRQ_MLA_#0
?	17	R	IRQ_memDMA_error
?	16	R	IRQ_memDMA_complete
?	15	R	IRQ_spDMA_1_overrun
?	14	R	IRQ_spDMA_1_limit
?	13	R	IRQ_spDMA_1_full
?	12	R	IRQ_spDMA_1_not_empty
?	11	R	IRQ_spDMA_0_overrun
?	10	R	IRQ_spDMA_0_limit
?	9	R	IRQ_spDMA_0_full
?	8	R	IRQ_spDMA_0_not_empty
?	7	R	IRQ_Rx_overrun
?	6	R	IRQ_Rx_not_empty_info
?	5	R	IRQ_Rx_not_empty_NN
?	4	R	IRQ_Rx_not_empty_C2C
?	3	R	IRQ_Rx_not_empty_MC
?	2	R	IRQ_Rx_not_empty
?	1	R	IRQ_Tx_not_full
?	0	R	IRQ_Tx_empty

Interrupt	Meaning
IRQ_external_*	Reserved for NoC packet interrupt sources
IRQ_MLA_*	Reserved for MLA interrupt sources
IRQ_memDMA_error	Memory-memory DMA (memDMA) anomaly
IRQ_memDMA_complete	Memory-memory DMA (memDMA) completed transfer successfully
IRQ_spDMA_1_overrun	At least one packet to spDMA dropped
IRQ_spDMA_1_limit	spDMA RAM FIFO exceeds programmed fullness
IRQ_spDMA_1_full	spDMA RAM FIFO is full
IRQ_spDMA_1_not_empty	spDMA RAM FIFO has unprocessed entries
IRQ_spDMA_0_overrun	At least one packet to spDMA dropped
IRQ_spDMA_0_limit	spDMA RAM FIFO exceeds programmed fullness
IRQ_spDMA_0_full	spDMA RAM FIFO is full
IRQ_spDMA_0_not_empty	spDMA RAM FIFO has unprocessed entries
IRQ_Rx_overrun	At least one packet to Rx dropped
IRQ_Rx_not_empty_info	Next receiver packet is type 'message'
IRQ_Rx_not_empty_NN	Next receiver packet is type SpiNNaker-NN
IRQ_Rx_not_empty_C2C	Next receiver packet is type SpiNNaker-C2C
IRQ_Rx_not_empty_MC	Next receiver packet is type SpiNNaker-MC
IRQ_Rx_not_empty	Receiver contains at least one packet
IRQ_Tx_not_full	Transmitter has free capacity
IRQ_Tx_empty	Transmitter completely empty

More, specific 'not_empty' indicators??

r2: IEN - Interrupt enable register

Mirror the bit positions of any potential interrupts.



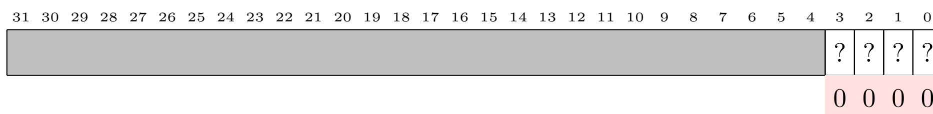
r?: IRQ - * Global interrupt status register

Consider ARM immediates when placing bits!

This register holds 'sticky' bits which are set by remote, events via 'exception' NoC packets. Such packets **set** any corresponding bits *only*. Bits can be cleared by writing '1's to the appropriate bits; writing '0's has no effect. [Note: this should be done *before* addressing any remote cause of the interrupt in case another packet is in transit.]

There needs to be some way of identifying the *source* of an interrupt where more than one source (such as a PE) is possible. This could be by software allocation — otherwise some means of associating the packet source (available in the NoC packet) with each potential interrupt is needed. (THIS COULD GET MESSY!) In principle, many PEs could make a request at the same time; thus a (big!) 2D bit array may be needed for status, with this register merely being the (in this case, read only) summary of the OR of these sources.

Initially there are four different requests known from the SpiNNaker router. *** THIS LIST WILL EXPAND! ***



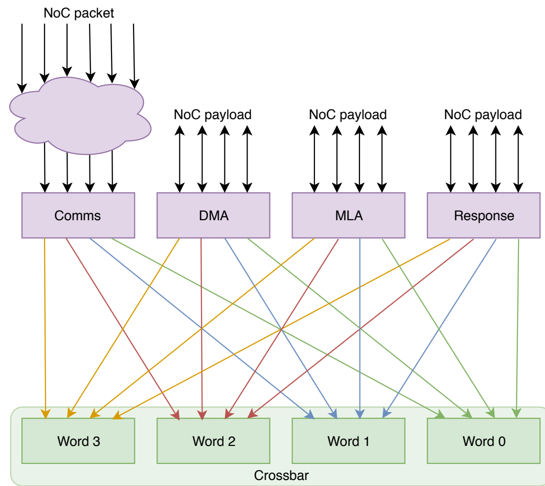


Figure 15: Exchange wiring

14 Comms unit

14.1 Features

1. Support for three SpiNNaker packet types:
   (a) multicast (MC) neural event packets routed by a key provided at the source;
   (b) core-to-core (C2C) packets routed by destination address to any core;
   (c) nearest-neighbour (NN) packets routed by arrival port;
      i. in both normal and peek/poke forms.
2. Support for software message packets to any on-chip PE.
3. Support for requesting and receiving data from any globally-addressable memory.
   (a) This provides non-blocking software support for remote read and write operations.
4. Output for sending exceptions to other processing units.
5. flexible packet features:
   (a) support for 40-bit packets with optional 32-, 64- and 128-bit payloads;
   (b) 2-bit time stamp (used by Routers to trap errant packets);
6. Two DMA engines to transfer selected incoming packet types into local SRAM.

14.2 Overview

This whole 'chapter' should, probably, be subdivided as shown in this (temporary?) subsection.

The comms unit structure is shown in figure 16. The transmitter is a fairly simple, UART-like transmitter, packaging data words from the AHB into NoC packets. The receiver performs a complementary function but there are programmable packet filters which allow particular classes of incoming NoC packets to be filtered. To relieve pressure from processor interrupts all received data is sent directly to the local RAM via DMA controllers which implements cyclic ring-buffers in the RAM. There are three such independent buffers: incoming NoC packets are offered to each in turn, the first two having filters which will trap out particular sorts of packet. The third buffer is the 'default' which handles packets which are not identified by the filters.

In addition to their filters, each buffer has the ability to discard *parts* of the packets. For example, it is envisioned that in operation with spiking networks the first filter would be set to trap MultiCast (MC) SpiNNaker packets with no payload (only) since these represent neuron firing and save (only) the key field from the NoC packet. The outcome would be a queue with 32-bit entries, each representing an incoming spike.

Figure 16: SpiNNaker2 NoC Topology

The processor may poll its input buffers periodically, be interrupted when a buffer is not empty or be interrupted when the buffer fills beyond a programmable amount. All buffer locations and sizes are software configurable.

A separate packet type can send 'messages', which are basically 'events'. There are 32 possible events which are asserted until cleared by the local software. Any of these can cause an interrupt or be left to be polled.

14.2.1 Area map

Base address: 0xE200_0000

Area	Offsets	Function
r0-r31	0x000-0x07C	Communications Transmitter
r32-r47	0x080-0x0BC	Default packet receiver
r48-r49	0x0C0-0x0C4	Message/interrupt receiver
r61-r63	0x0F4-0x0FC	General control
r64-r79	0x100-0x13C	Receiver filter 0
r80-r95	0x140-0x17C	Receiver filter 1
r96-r102	0x180-0x19C	Memory-memory DMA controller

Processor access to the device is via an AHB interface from the M4 System Bus; this allows configuration and data transfer from and to the M4. An additional *set* of *master* buses are used by the DMA channels to write data through the crossbar to the local SRAM. These provide four semi-independent, word-interleaved paths which can, in principle, operate in parallel.

14.2.2 Packet transmitter

The transmitter is responsible for assembling packets from input information and can derive some fields automatically: for example the packet payload presence/size is (normally) deduced as the payload words are input. Alternate transmitter addresses are offered so that data like the packet type do not need to be reloaded for every packet; it is possible to choose the control information from one of four preprogrammed registers. For example, it might be anticipated that one is reserved for neural spike outputs so a spike can be sent by a single 32-bit write of its source ID.

The Tx module can generate any type of NoC packet although in normal use only a subset of types would typically be initiated by this unit. (Care should be taken not to generate packets which may cause problems elsewhere in the system.) The Tx packet forming submodule receives data from the AHB slave interface, forms the packet and sends it into the Tx FIFO. The Tx FIFO is a conventional FIFO whose function is to release the Tx packet forming submodule quickly if the NoC Tx channel is blocked. The current FIFO is one entry (which may be uncomfortably short: JDG) long.

In the Tx packet forming submodule there are four sets of configuration registers, comprising {TCR*, TDR**, TKR*}.

These are all address aliases except for the TCRs, which are discrete registers. All the registers are of 32-bit width. The sets of registers are subscripted {a, b, c, d}. The intended operation is that commonly sent packet types can be configured in chosen TCRs which then do not need rewriting; the other values – which are expected to differ in every packet – are then written to the corresponding alias, thus saving an operation in each transmission.

Packet transmission is caused by writing to the appropriate TKR; this must therefore be the *last* word written. This then assembles the packet using the corresponding TCR for packet type, destination etc. and appends the size as determined by the *lowest numbered* (i.e. lowest offset) TDR written to since the last packet was sent. The anticipated software behaviour would therefore be writing the appropriate number of words – less the control which is preset in TCR – in ascending register order to the appropriate alias. This can be done using an ARM `STM` instruction.

When the TDR* registers are written, the payload size is deduced automatically from the lowest numbered ('lowest' address) TDR register written to. When a TKR* is written, the packet is formed and sent out automatically and the payload size counter is cleared. The payload size is selected from the values {0, 1, 2, 4} words. Although this mechanism can be overridden, and the size set manually, it is expected that payloads will generally differ in packet sequences.

Bits in the TCTL override the automatic counter for the different TCRs When TCTL[4] is set ('1'), and TCRa is selected (by writing TKRa), size comes from TCRa[31:29]; when TCTL[5] is set, and TCRb is selected, it comes from TCRb[31:29]; when TCTL[6] is set, and TCRc is selected, it comes from TCRc[31:29]; when TCTL[7] is set, and TCRd is selected, it comes from TCRd[31:29].

Such a mechanism is necessary to enable the explicit (as opposed to *implicit* – via the bridge) sending of packets such as requests to write a single byte. It is suggested that this mechanism – used in **Jib1** – is subsequently reviewed and simplified. Unfortunately there is not space to *dedicate* a control bit in a TCR solely to this function; one way could be to use a previously unused code in TCT[31:29] (e.g. '111') to designate 'use the packet count' although the question of what should be

read back in these bits is then raised. Another approach would be to overload some bit field(s) as there are always some which have no purpose in some packets. *** TBC ***

When the packet is of type '111' (i.e. a SpiNNaker packet) bits [161:160] of the packet are always derived from payload_size counter.

Request packets require a response which should be returned to the initiator. The initiator PE ID is fixed by the hardware and integrated automatically when an appropriate packet type is sent. The PE ID is not visible in the TCR (there is no room) but can be read in software from the Chip_ID register.

14.2.3 Packet receiver

Received packets are filtered and routed internally various ways. Packets with specific purposes, such as messages/interrupt events, are routed to the appropriate unit internally. Those which are not so recognised arrive in the packet receiver where they can be filtered for particular, programmed characteristics. All such received packets are moved into local SRAM by DMA but specific, recognised packets are routed to a their own areas. There are two programmable filters in series before remaining packets are placed in the 'default' buffer.

The filters are designed to look for SpiNNaker packets only? messages? other?.

The filters look for packets of given type(s) and payload size(s). These criteria are bit-selectable so several types or sizes may be included in one filter. Matching packets are diverted into the appropriate DMA channel for writing into SRAM.

Saving of the *fields* within the packet can also be controlled individually; thus the control word (akin to RCR) the 'key' word (RKR) and the set of payload words (as specified within the packet) can be saved or discarded individually.

> Example: a filter could be set to identify only multicast (MC) packets without a payload – these typically corresponding to neural spikes. From these only the 'key' field could then be selected for saving. The result would be a RAM buffer containing only the spike key IDs in successive addresses.

Figure 17: Block Diagram of SpiNNaker Packet DMA

Fig.10-5 shows the structure of SpiNNaker Packet DMA. The input packets firstly are filtered by a packet filter, which is configurable by the processor, then the packets go into a FIFO. There are two SpiNNaker DMAs below the FIFO, each one maintains a set of registers, which control the packet filter and the storing of wanted packets. The two DMAs commonly use a 192-bit AHB-like bus to access local memory. The order incoming packets are stored is header, payload[3], payload[2], payload[1], payload[0], address. If a field is configured not to be received the following fields move up in addresses.

In each SpiNNaker Packet DMA, There are some registers controlling its function. DMAc decides what kind of packets it receives. DMAs is the start address of the buffer in local memory. DMAe is the end address of the buffer. DMAr is the read address. DMAw is the write address. DMAl is the buffer count (in words) limit. RAM_buff_count is the RAM buffer count (in words too). RAM_buff_max is the maximum RAM_buff_count that has ever reached. DDC records the number of packets that are dropped (packets that are wanted but buffer is full). DCTL is the configuration register for this module itself.

14.2.4 Rx Module

`Fig.10-6 shows the diagram of default Rx submodule. Received Packets are first`

Figure 18: Block Diagram of default Rx submodule

filtered, then go to a packet FIFO. At the output of the FIFO, Packets are stored into registers and then interrupt the processor.

There are six 32-bit packet registers for storing the incoming packets. These are RCR, RKR, RDR3, RDR2, RDR1, and RDR0. The six registers correspond to the [191:0] bits of a packet sequentially. When the processor read out these registers, the RKR register should be read as the last one. When RKR is read, all registers are cleared, and the next packet can be stored into the registers.

There are two registers who are responsible for the configuration/status report of the default submodule: RCTL and RDC. When the packet registers are full, and a packet arrives. The default receiver can decide to wait until the previous packet is read out, or just drop the incoming packet. The choice is configurable by bit RCTL[17]. When it is configured to drop out the incoming packet, RDC counts the packets that are dropped.

The packet filter decides to accept what kind of packets coming into the FIFO and then registers. It can also reject all kinds of packets. The function of the packet filter is configurable by RCTL.

Displaced paragraph below:

There is a control register RCTL in the dRx module. With some bits of RCTL, we can control whether dRx accept packet types that should be directed to previous modules. That is, when one previous module is busy, for example, RespPkt, the incoming packet can be routed to dRx. However, if the dRx is configured to reject the packet, the NoC interface will be blocked and waiting until RespPkt receives it. *** EH?! NEEDS CLARIFICATION! ***

FIFO operation

The various receiver FIFOs are controlled through hardware-defined registers. They are confined to working with 32-bit words. The first and last word address (inclusive) is programmed into hardware registers and the received words are stored cyclically, between these, in ascending address order. The reset value is the lowest ('start') address. In operation the hardware DMA will insert words into the SRAM and then increment the software-visible write pointer; *this pointer should not, normally, be written to in software*. When reading from a buffer the software should load the word(s) of interest and, subsequently, write the new address to the read pointer. It is not necessary to write every word address in software; if several words (such as a 128-bit payload) is read at once then only a single write is needed and the hardware will know that all addresses up to that one are free. The DMA will not overwrite 'used' space in the buffer, exerting some (limited) backpressure on the NoC but, typically, discarding input if the buffer is full. An 'overrun' is indicated if discarding has occurred.

A FIFO is empty when its pointers are equal. For (software) simplicity the FIFO is 'full' when only a single word remains free, so the pointers are *only* equal when the FIFO is empty.

Each buffer's occupancy (in words used) is also calculated by hardware and made available to the programmer. There is also a user-specifiable soft limit on the pending contents of the buffer which can generate an interrupt if exceeded; this can act as a warning that the buffer needs urgent attention.

A monitor register retains the maximum capacity used in each buffer to assist with performance analysis and allow FIFO sizes to be optimised for particular circumstances.

14.2.5 Messages

Specific 'message' packets can be sent between processing elements. These are separated from other packets at reception and used to set appropriate bits in a hardware register. such bits remain set until cleared by software. Any of these bits can be enabled to assert an interrupt request. This primary intention of messages is to enable software-generated interrupts between PEs.

14.3 Register summary

Base address: 0x????????

Name	Offset	R/W	Function
r0: TCRa (Tx configuration)	0x000	R/W	Controls packet transmission
r1-4: $TDRa_i$ (Tx data)	0x004-010	R/W	Transmitter data payload [0:3]
r5: TKRa (Tx key)	0x014	R/W	Send packet type 'a'
r8-13: TCRb, $TDRb_i$, TKRb	0x020-034	R/W	Tx register variants/aliases - type 'b'
r16-21: TCRc, $TDRc_i$, TKRc	0x040-054	R/W	Tx register variants/aliases - type 'c'
r24-29: TCRd, $TDRd_i$, TKRd	0x060-074	R/W	Tx register variants/aliases - type 'd'
r31: TCTL (Tx control)	0x07C	R/W	Transmitter control register
r32: DMAc	0x080	R/W	dummy register, no function
r33: DMAs (Rx buffer start)	0x084	R/W	First buffer memory address
r34: DMAe (Rx buffer end)	0x088	R/W	Last buffer memory address
r35: DMAr (Rx buffer read)	0x08C	R/W	Buffer read pointer
r36: DMAw (Rx buffer write)	0x090	R(/W)	Buffer write pointer
r37: DMAl (Rx buffer limit)	0x094	R/W	Buffer soft limit (in words)
r38: RDBO (Rx buffer occupancy)	0x098	R	Current buffer occupancy (words)
r39: RDMax (Rx tidemark)	0x09C	R	Buffer maximum occupancy
r40: RDDC (Rx Drop Count)	0x0A0	R	No. of receiver packets discarded (overrun)
r47: RCTL (Rx control)	0x0BC	W	Receiver Control?? Register
r48: Message	0x0C0	R/W*	Message signals
r49: Message_IE	0x0C4	R/W	Message interrupt enables
r61: Receiver timeout	0x0F4	R/W	Stall cycles before Rx packet discarded
r62: Chip_ID	0x0F8	R	Chip ID
r63: GCTL	0x0FC	R/W	Global control register
r64: spDMAc0	0x100	R/W	spDMA channel 0 configuration
r65: spDMAs0	0x104	R/W	spDMA channel 0 start address
r66: spDMAe0	0x108	R/W	spDMA channel 0 end address
r67: spDMAr0	0x10C	R/W	spDMA channel 0 read address
r68: spDMAw0	0x110	R	spDMA channel 0 write address
r69: spDMAl0	0x114	R/W	spDMA buffer soft limit (words)
r70: DBO0	0x118	R	spDMA buffer occupancy (words)
r71: DMax0	0x11C	R	Maximum spDMA buffer occupancy reached (words) ('Tidemark')
r72: DDC0 (Drop Count)	0x120	R	No. of spDMA packets discarded (overrun) - channel #0
r79: DCTL0 (Control)	0x13C	R/W	spDMA channel 0 control
r80-95: spDMA{...})1	0x140-17C	R/W	spDMA channel 1: same as channel 0.
...			

A packet will contain a data payload if any of r1-4 – or their aliases – is written before r5; this can be performed using a single ARM STM instruction if desired. A 32-bit payload is written into r4, a 64-bit payload into r3 & r4, and a 128-bit payload into r1-r4 (the cases with the aliased addresses being similar). For compatibility with a receiver DMA unit (spDMA), if employed, it is recommended that r4 is always used for the *most* significant word (etc.), regardless of the payload size.

> Note: the Cortex M4 STM operations are *interruptable* so atomicity cannot simply be assumed.

There are four alternative *aliases* for the transmit registers, with the exception of the TCRs, which are unique: {a, b, c, d}. Thus, default control set-ups can be made in some TCRs and reused by addressing the appropriate alias.

14.4 Register details

14.4.1 Transmitter

Offsets 0x000-0x01F

r0, r8, r16, r24: TCR - transmit configuration register

These four registers are used for setting control information for transmitted packets. Unlike the accompanying TDR and TKR addresses, which are aliases for the *same* registers, the four TCRs are distinct. This allows four packet set-ups to persist, avoiding rewriting the values each time. (The TKR and any TDR contents are expected to be different, and therefore written, for every packet sent.)

A potential application might dedicate one TCR for sending MC packets, another for C2C packets carrying comms. traffic etc.

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
Pkt Sz	Dest X	Dest Y	Dest PE	C	Type	0 0 0	Froute	STy	SW	TS	B	S
0 0 0	0 0 0	0 0 0	0 0 0 0 0	0	0 0 0	0 0 0	0 0 0	0 0	0 0	0 0	0	0

The functions of these fields are described in the table below:

Name	bits	R/W	Function
Pkt Sz	31:29	R*/W	Data size as will be sent in the packet
Dest X	28:26	R/W	Destination NoC node address, X
Dest Y	25:23	R/W	Destination NoC node address, Y
Dest PE	22:18	R/W	Destination(s) within NoC node (details?)
C: Use CNOC	17	R/W	0 = use DNOC; 1 = use CNOC
Type	16:14	R/W	NoC packet type
	13:11	R	Read as 000
Froute	10:8	R/W	Set 'fake' route in SpiNNaker packet
STy: SpiNN Type	7:6	R/W	SpiNNaker packet type {00 = MC, 01 = C2C, 10 = NN}
SW: SpiNN SW	5:4	R/W	SW defined (if SpiNNaker packet)
TS: SpiNN TS	3:2	R/W	Time stamp if SpiNNaker packet and relevant
B: Buffered write	1	R/W	Only used in write requests
Rq Sz:	3:1	R/W	Size in type 0 packets:
S: Supervisor	0	R/W	Used in memory requests etc.

The Pkt Sz field reflects the bit code which will be sent in the packet. This may be explicit or (more usually) is derived from the 'counter' associated with writes to the TDRs; it is 'R*/W' in that the bits are writeable but will only *read back* here if the appropriate bit (7:4) (one bit per TCR) in the control register is set; if the bit is clear the value will reflect the automatically calculated field from the data writes. The value read is always the one which will be used in the transmited packet.

When automatically derived, the size will be the largest value since the previous packet transmission, as indicated in this table. Packet transmission (or reset) resets this to zero.

Written	Size	Meaning
None	000	No payload
$TDRx_0$	011	One word
$TDRx_1$	100	Two words
$TDRx_2$	101	Four words
$TDRx_3$	101	Four words

The "Dest" fields and "C" (bits 28:17) are used for routeing the packet across the NoC.
Type specifies the NoC packet type:

Code	Type
000	Read request
001	Read response
010	Write request
011	Write response
100	Control/exceptions
101	Protocol (SW defined) message
110	(SDRAM DMA: do not use)
111	SpiNNaker packet

Request size is only used in read request packets and determines the size of the payload(s) returned in the resulting response packets. Such packets are usually automatically generated by the bus-NoC bridge but can be sent by hand to read remote memory locations, the data being returned to the normal, software receiver.

The Froute (a.k.a. "link") field allows a packet to be sent to the router which *appears* to have come from one of the external links. Normally this field will be set to 7 (0b111) but can be set to a link number in the range 0 to 5 to achieve this.

Bits 7-2 of the appropriate TCR are used in the *control byte* if a SpiNNaker packet is sent. [cross-reference appropriate section]

STy is the SpiNNaker packet type:

Code	Type
00	MC packet
01	C2C packet
10	NN packet
11	—

The time stamp (where applicable) will be inserted by the local Router in the normal case where the Froute field is 111, otherwise the value here will be used. Only SW and, in the case of NN packets *** and other, local NoC packets ***, T and the control byte route will normally be defined by TCR.

The *size* field for a SpiNNaker packet is notionally derived from the *Dest Sz* field of the TCR, which is itself derived from which TDR registers have been written to since the previous NoC packet was dispatched.

The 'B' bit is used to determine if an outgoing write request will generate ('0') a response packet or not ('1').

If 'S' is set then packets are sent with the initiator's priority (determined from the write to TKR). If 'S' is clear then the packet is sent in user mode (i.e. S=0 in the packet.)

Some NoC packets include a source ID field; this is added automatically, if appropriate, as the transmitting unit's ID, which is determined in hardware by its physical position.

Any attempt to write to any transmitter register other than the transmitter control register (TCTL) when the transmitter is full will cause a data abort. Thus, there are two means of providing output flow control:

1. The transmit buffer ready status can be used, by polling or interrupt, to prevent buffer overrun.
2. A transmission can be attempted without first checking 'fullness'; an abort indicates that there was not room and the attempt failed.

Under most operating conditions it is expected that a transmitted packet will have reached the local transmitter FIFO within a single clock cycle; as the NoC will typically provide more bandwidth that is needed by several processors it is expected that aborts would be extremely rare in the second case and this can provide a much more efficient transmission mechanism in most situations.

These are the normal, legitimate TCR settings:

FIX TABLE HERE - FIXED 24/3/20 @@ correct nunber of unused bits?

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">Dest X</td><td colspan="3">Dest Y</td><td colspan="5">Dest PE</td><td>C</td><td colspan="3">0 0 0</td><td colspan="3">Pkt Sz</td><td colspan="10">unused</td><td>S</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">Dest X</td><td colspan="3">Dest Y</td><td colspan="5">Dest PE</td><td>C</td><td colspan="3">0 1 0</td><td colspan="12">unused</td><td>B</td><td>S</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">Dest X</td><td colspan="3">Dest Y</td><td colspan="5">Dest PE</td><td>C</td><td colspan="3">1 0 0</td><td colspan="12">unused</td><td>B</td><td>S</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">Dest X</td><td colspan="3">Dest Y</td><td colspan="5">Dest PE</td><td>C</td><td colspan="3">1 0 1</td><td colspan="13">unused</td><td>S</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">RouterX</td><td colspan="3">RouterY</td><td colspan="5">???</td><td>C</td><td colspan="3">1 1 1</td><td colspan="3">unused</td><td colspan="3">Froute</td><td colspan="2">0 0</td><td colspan="2">SW</td><td colspan="2">TS</td><td colspan="2">—</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">RouterX</td><td colspan="3">RouterY</td><td colspan="5">???</td><td>C</td><td colspan="3">1 1 1</td><td colspan="3">unused</td><td colspan="3">Froute</td><td colspan="2">0 1</td><td colspan="2">SW</td><td colspan="2">TS</td><td colspan="2">—</td></tr>
<tr><td colspan="3">unused</td><td colspan="3">RouterX</td><td colspan="3">RouterY</td><td colspan="5">???</td><td>C</td><td colspan="3">1 1 1</td><td colspan="3">unused</td><td colspan="3">Froute</td><td colspan="2">1 0</td><td>T</td><td colspan="3">route</td><td colspan="2">—</td></tr>
</table>

r1-4, r9-12, r17-20, r25-28: TDR*$_i$ - transmit data payload

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="32">32-, 64- or 128-bit data payload for sending with next packet</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

If data is written into TDR before a send key or dest ID is written into TKR, the packet initiated by writing to TKR will include the contents of TDR as its data payload, which may be 32-, 64- or 128-bits long as determined by the lowest TDR *index* (0-3) written to. If no data is written into TDR before a send key or dest ID is written into TKR the packet will carry no data payload. (The payload length setting can be reset via TCTL.)

There are four, discrete 32-bit components to the TDR as indicated by the subscript index. The highest subscript corresponds to the lowest register number, thus TDR*$_3$ appears as {r1, r9, r17, r25}, etc. The disparate TDR addresses are simple aliases for the *same set* of registers; it does not matter which alias is used in any operation. The aliases are provided so a packet, including its payload, can be sent with a single ARM `STM` instruction. In many cases (a) TCR does not need to be updated for a particular send operation so the `STM` can 'begin' at the highest desired TDR index (if any), work downwards and finish with a TKR alias.

r5, r13, r21, r29: TKR - send MC key or C2C dest ID & sequence code

*** Cross-reference packet description section. ***

This register (and its counterparts) allow the miscellaneous packet bits to be set up. Not all of the fields are used in a given packet. The bit fields can be divided into two groups: one of 15 bits which are used in the NoC transmission itself and the 17 bits which make up the balance of the headers. Any unused bits are sent as '0's.

The packet size may be specified here or an automatically derived from the store operations setting up the data field. However for certain packet types (notably read request (000) and write response (011)) which never legitimately carry payloads the size is forced to 'no payload', regardless.

The destination, 'C' and type fields are copied directly from this register.

Within the body of the packet, the source fields are hard-wired to the values for the particular PE. The source type is programmable.

Where appropriate, the 'B' bit is taken from bit 1 of the TCR.
Where appropriate, the 'S' bit is taken from bit 0 of the TCR.
Alternatively . . .
Where appropriate, the 'BE' bit is taken from bit 0 of the TCR.

– Specialist SpiNNaker packet stuff incomplete –

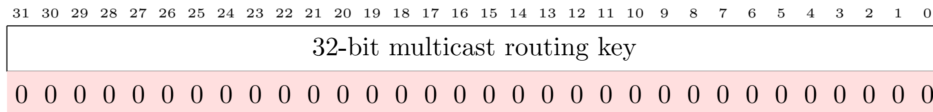
Resets to 0000_0001.

Writing to TKR causes a packet to be issued (with a data payload if TDR was written previously). The packet type is determined by bits in the TCR corresponding to the TKR alias which causes the packet to be sent. The appropriate TKR interpretation is the responsibility of the software.

TCR[16:14]	TCR[7:6]	Valid?	Packet type	TKR meaning
000	—	SrcT = 2'b01	Read request	Read address
001	—	No	Read response	(Read address)
010	—	SrcT = 2'b01	Write request	Write address
011	—	No	Write response	(Written address)
100	—	Yes	Control/Exception	Command ???
101	—	Yes	Message	Command
110	—	No	*undefined*	*undefined*
111	00	Yes	MC packet	AER key
111	01	Yes	C2C packet	PE address
111	10	Yes	NN packet Address/operation I forget :-}	
111	11	No	*undefined*	*undefined*

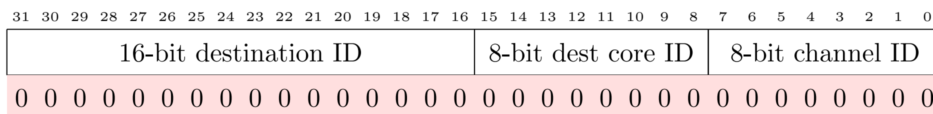
Multicast key

The 32-bit routing key is used by the associative multicast Routers to deliver the packet to the appropriate destination(s).



C2C destination address

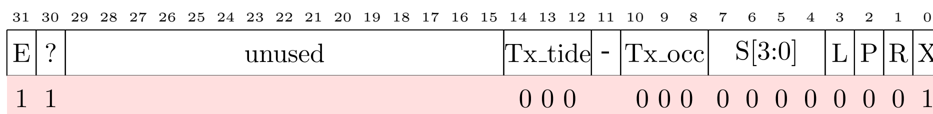
The value written into TKR should include the 16-bit address of the destination chip in bits [31:16], the destination core in bits [15:8], and the channel code (once established) in bits [7:0].



*** Other packet types. ***

r31: TCTL - Transmitter control bits

This register controls the overall function of the packet transmitter and gives access to its status. This is the only register in the packet transmitter which can be written to when the transmitter is 'full' (attempting to write to other registers will cause a data abort under that condition).



The functions of these fields are described in the table below:

Name	bits	R/W	Function
E!: ready	31	R	Tx buffer ready (not full)
?: empty	30	R	Tx buffer empty
Tx_tide	14:12	R	Maximum Tx FIFO occupancy Review: currently reduced to 1 bit which is **meaningless!**
Tx_count	10:8	R	Current Tx FIFO occupancy Review: currently reduced to 1 bit.
S[3:0]: size test	7:4	R/W	Force packet size field [3:0]
L: low tide	3	W	Flush maximum Tx occupancy Review: see above!
P: ????	2	W	Flush payload size setting
R: reset	1	W	Flush Tx FIFO buffer
X: stop	0	R/W	Disable transmission (at FIFO output)

*** More bits/functions?? ***

The transmitter will only output packets if X (Stop) is clear; this does not prevent accumulating packets in its internal buffer. The internal buffer can be flushed by writing a '1' to the appropriate control bit; setting X disables the output but does not, intrinsically, flush the buffer. Transmitter 'empty' indicates the internal FIFO has been cleared so all packets have left this unit; this is primarily intended for management purposes and is *not* exported as an interrupt. The 'Ready' status bit, which indicates that the transmitter is not full and will accept another packet is more useful in applications programming and is available as an interrupt.

The transmitter packet FIFO holds 1 (TBC) packet(s) in addition to the holding register. The instantaneous number of packets residing here is available for reading but this will typically change (reduce) rapidly. A separate field indicates the maximum value this count has reached, which is more useful in monitoring network congestion; this field can be cleared (to current occupancy) by writing a '1' to the 'L' bit in this register.

*** This feature made effectively redundant by the shrinking of the output FIFO. ***

If the transmitter becomes full, further *write* operations to any register other than TCTL will cause a data abort.

Writing a '1' to the P bit resets the payload length counter, as determined by which TDR registers have been written to. It has no obvious function in normal operation.

The 'S[3:0]' bits cause the substitution of the size field from the corresponding TCR to replace the (default) size as supplied by the payload_size counter.

14.4.2 Default receiver

Offsets 0x020-0x2F

This unit acts as a packet receiver for all packets which are not trapped by the other filters. It will store the packets – or a subset thereof – by DMA into a cyclic buffer in the local SRAM. The buffer's size and position are programmable.

r32: DMAc

This register exists, so all DMAs have a similiar register interface. Think about removing it later

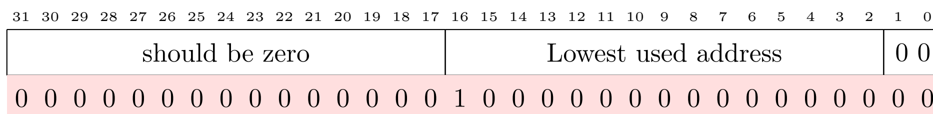
Resets to 0xFFFFFFFF.

r33: Buffer start address

First address used for cyclic buffer. The address is constrained to a address and allows addresses up to 0x1FFFC. The minimum size of the buffer is 6 words.

It is aligned as a byte address and will be the same as the addresses seen by the processor, directly.

It resets to 0x00010000.

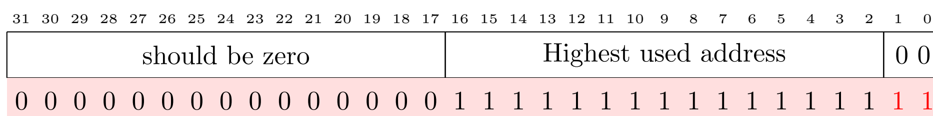


r34: Buffer end address

This is the last word address (inclusive) used in the cyclic buffer. It is subject to the same constraints as the buffer start address.

Although all the locations between the start and end of the buffer will be used, to avoid confusion with wrap around (at least) one word location will always be left free. Thus, setting start to 0x100 and end to 0x1FC would provide a useful maximum capacity of 0x3F words.

It resets to 0x0001FFFC.



r35: Buffer read pointer

This is the head of the cyclic buffer – a.k.a. the 'last' used (word) address (i.e. it will be the numerically *lower* pointer unless/until the write ('tail') pointer has wrapped around). It represents the address from (and above) which packet data resides in SRAM.

This pointer is to be managed entirely in software. Writing to this pointer frees up space 'behind' it. The expected sequence of operations would be expected to be:

1. read read- and write pointers
2. load data from SRAM
   (a) beginning at read pointer
   (b) any number of words, but
   (c) not *exceeding* write pointer
3. update and write back read pointer

It is a *software* responsibility to reset this pointer to the buffer start (r33) when it has *exceeded* the buffer end (r34).

The buffer write pointer will never 'lap' this pointer.

The two pointers being equal indicates an empty buffer. The two pointers can be reset to the start addresses by writing the appropriate command bit (in r47).

The minimum buffer size should never be less than 6 words, otherwise a single received packet would overwrite itself partially.

The reset value is 0x00000000.

31–17	16–2	1 0
should be zero	Next word address	0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0

r36: Buffer write pointer

This is the tail of the cyclic buffer – a.k.a. the 'next to be' used (word) address (i.e. it will be the the numerically *higher* pointer unless it has wrapped around (or be equal if the buffer is empty)). It holds the next (word) address in SRAM which will be written to by incoming packets.

Although it can be written to, normally this pointer is to be managed in hardware. When it has been updated any SRAM locations 'behind' it will be valid.

> (FORMERLY . . . Software writes are only possible when enabled by setting the relevant status bit in GCTL (r63).
> This restriction seems to have disappeared?)

The buffer write pointer will not 'lap' the read pointer. If the size of the incoming packet is sufficient to do this the write operation will be stalled or the packet dropped (depending on the action programmed in r47).

The two pointers being equal indicates an empty buffer. The two pointers can be reset to the start addresses by writing the appropriate command bit (in r47).

The reset value is 0x00000000.

31–17	16–2	1 0
should be zero	Oldest word address	0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0

r37: Buffer soft limit

This is a 15-bit register counting 32-bit *words*; it is RH aligned. If the number of occupied words in the buffer equals or exceeds this value the status is reported (in r47) and an interrupt may be caused.

The intended purpose of this register is to allow attention to be attracted if the buffer is getting full, before a problem can manifest.

The reset value is 0x4000.

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
zero																	Word count alert														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

r38: Buffer occupancy

This read-only, 15-bit register provides the number of *words* currently held in the SRAM buffer (derived from the difference in the write- and read pointers).

Inevitably, it resets to 0x0000.

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
zero																	Word count														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

r39: 'Tidemark' – the maximum fullness reached

This read-only, 15-bit register holds the maximum value of the buffer occupancy (r38) reached since it was reset. It can be reset by writing to the relevant bit in the RCTL (r47). The reset value is unknown, as they need valid start and end addresses, and read and write pointers setting up before they can be evaluated. The DMax value should be explicitly 'cleared' before it is first used.

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
zero																	Max. word count reached														
X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

r40: Dropped packet count

This read-only, 15-bit, saturating counter holds the number of *packets* which have been discarded due to an SRAM buffer overrun. It can be reset by writing to the relevant bit in the RCTL (r47).

The reset value is 0x0000.

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
zero																	Dropped packet count														
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

r47: RCTL – Control 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
NE	EF	L	O	S	unused										R	unused								?	K	H	P	un	L	C	U
0	0	0	0	0											0										0	0	0				1

Name	bits	R/W	Function
Not_empty	31	R	SRAM buffer not empty
Full	30	R	SRAM buffer full : indicates that an incoming packet cannot be accommodated. **
Limit	29	R	SRAM buffer limit reached : the buffer occupancy has reached or exceeded the soft limit.
Overflow	28	R	Packet(s) dropped : one or more packets have been discarded.
Stalled	27	R	Packet(s) waiting : one or more packets are queued in the NoC drop.
Reset	16	W	Writing a '1' to this bit resets the read and write pointers to the buffer start value.
	7	W?	* ** Latched? but unused? Maybe read masked too **
Key	6	R/W	Keep key: if '0' the packet 'key' field (second 32-bit word) will be discarded: if '1' the bit will be stored.
Header	5	R/W	Keep header: like 'keep key' but for the 'header' (first 32-bit) word.
Privilege	4	R/W	Privilege: the 'HPROT' value used for the memory transfers. (Probably never checked on this AHB.)
Low_tide	2	W	Writing a '1' to this bit resets the buffer 'tidemark' (to the current 'fullness').
Clear_drop	1	W	Writing a '1' to this bit clears the packet drop counter.
Unblock	0	R/W	Drop packets which can't be accepted **

Reset value is 0x00000001

14.4.3 Messages

Offsets 0x030-0x037

There are 32 possible 'messages' which can be sent, each as a single bit encoded in a word. Any subset of these may be enabled to assert the message interrupt.

Arriving messages set the appropriate bit(s) in the message register. These bits are 'sticky', remaining set until cleared by software. Clearing bits is achieved by writing to the message register with a '1' in the appropriate bit position(s); writing with a '0' bit has no effect.

r48: Pending messages

Reading this register returns a 32-bit word containing set bits for all the pending messages.

Writing a 32-bit pattern will *clear* any bits where these is a '1'. It is not possible to set bits directly in software.

The reset value is 0x00000000.

r49: Message interrupt enables

This read/write register specifies which messages can assert a message interrupt. A '1' bit enables the corresponding message bit to interrupt.

The reset value is 0x00000000.

14.4.4 Miscellaneous

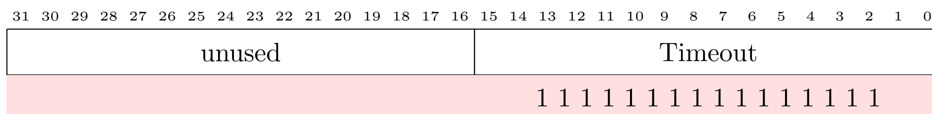
Offsets 0x038-0x03F

r61: Receiver timeout

This is a 16-bit (right justified) read/write register. The upper 16-bits read as **undefined**.

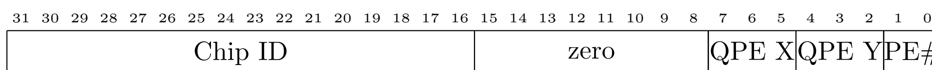
The value acts as a receiver timeout. When an incoming NoC packet is stalled a counter is started, incrementing every local clock cycle. If it reaches this value the packet is given to the default DMA.

The reset value is 0xFFFF.



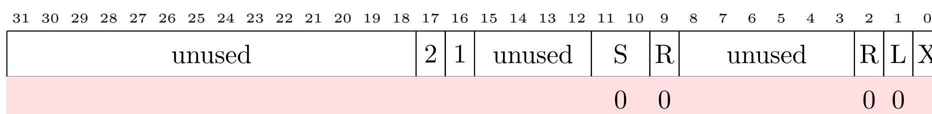
r62: Chip_ID

This register holds the unique identifier of the particular PE. The upper 16 bits are programmed elsewhere, on a chip-wide basis. The lowest 8 bits are hard-wired according to the PE's physical position on the chip.



Name	bits	R/W	Function
Chip ID	31:16	R	ID of this specific chip; writeable globally via an I/O register (TBC)
QPE X	7:5	R	Physical 'X' position of this QPE
QPE Y	4:2	R	Physical 'Y' position of this QPE
PE#	1:0	R	PE identity within QPE

r63: GCTL - Global/General Control register



Name	bits	R/W	Function
2: Disable Timer 2	17	R/W	Disable RX Timer 2
1: Disable Timer 1	16	R/W	Disable RX Timer 1
B: Buffered Write	11	R/W	Enable buffered write
S: global segment	10:9	R/W	global segment
R: resp mode	8	R/W	resp mode
T: test write	2	W	Activate test write
L: loopback	1	R/W	Connect transmitter to packet receiver

The loopback bit is for test purposes and should normally be left as '0'. Note that setting this bit will also affect the packet receiver's connections.

14.4.5 Receiver filters

Offset 0x100-0x17F

The Communications Controller has two additional receiver DMA channels; each can be programmed to transfer incoming packets meeting a particular specification directly into a cyclic buffer in local SRAM. Each DMA channel has programmable start and end word addresses for its SRAM buffer, read and write pointers and a control register which defines which sort(s) of packet it handles.

Each DMA buffer is circular and should typically be configured to be a multiple of the packet size that it stores. Packets occupy up to $2 + N$ words, where N is the number of words in the payload. Packets are logically subdivided into three fields: {control, payload and key/address/source}. There are separate enables so that any field can be independently retained or discarded. Words are written to ascending addresses, modulo the buffer size. The programmed DMA start address is the first used address; the programmed DMA end address is the last used address, thus both are included and will be written to.

The first (in order of ascending addresses) word from a received packet to be written will hold the 'control' word if this is to be retained. The second field (if enabled) will hold the data payload of

{0, 1, 2, 4} words (i.e. RDR), with the size being determined by the incoming packet; the last word (if enabled) will hold the 'key' or 'address' word. Initially the read and write pointers should be set to the buffer start address. Writing to the buffer is a hardware process; reading from the buffer is the responsibility of software, which should maintain the read pointer accordingly. Any requirement to write to the buffer which would cause the write pointer to catch up with the read pointer will stall and, potentially, cause back-pressure into the NoC or packet drop?. Like the default receiver there is an option to alleviate back-pressure by allowing overruns; again overrunning packets are counted but otherwise discarded.

r64, r80: spDMAc0-1 - DMA configuration registers

The DMA configuration registers define the sorts of packet(s) which each DMA channel handles. The packets transferred can be specified by type (MC, C2C or NN) and by payload length, although any or all payload lengths and any or all packet types can be handled by the same DMA channel.

31	30	29	28	27	26–22	21	20	19	18	17	16	15–11	10	9	8	7	6	5	4	3	2	1	0
E	F	L	V	?	unused	T	Z	X	A	X	R	unused	K	P	S	Q	D	W	Z	I	N	C	M
0	0	0	0	1		0	0	1	1	0	0		1	1	1	0	0	0	0	0	0	0	0

The functions of these fields are described in the table below:

Name	bits	R/W	Function
E!: empty	31	R	spDMA buffer not empty
F!: full	30	R	spDMA buffer full
L!: limit	29	R	spDMA buffer soft limit reached/exceeded
V!: overrun	28	R	spDMA overrun count non-zero
?: FIFO empty	27	R	Internal FIFO empty (read pointer == write pointer)
T: clear max.	21	W	spDMA 'tidemark' reset
Z: zero overrun	20	W	spDMA overrun count reset
A: stall	18	R/W	stall noc if buffer is full until buffer can accept again
R: reset	16	W	reset spDMA buffer read + write pointer
K: key	10	R/W	retain key field (RKR)
P: payload	9	R/W	retain payload field (RDR)
S: status	8	R/W	retain status field (RCR)
Q: 128-bit	7	R/W	enable spDMA on 128-bit payload packet
D: 64-bit	6	R/W	enable spDMA on 64-bit payload packet
W: 32-bit	5	R/W	enable spDMA on 32-bit payload packet
Z: no payload	4	R/W	enable spDMA on no payload packet
I: Info.	3	R/W	enable spDMA on information (message) packet received
N: NN	2	R/W	enable spDMA on NN packet received
C: C2C	1	R/W	enable spDMA on C2C packet received
M: MC	0	R/W	enable spDMA on MC packet received

The DMA channels are prioritised: if an incoming packet matches the specification of channel 0 it is handled by channel 0, else if it matches the specification of channel 1 it is handled by channel 1, and so on. If it does not match any channel specification it is directed to the Comms. Controller registers.

The same priorities apply to transfer to the SRAM. Channel 0 has the higher priority. This may be important in that the transfer – using a 32-bit bus – may, necessarily, be slower than the packet reception and a number of packets may be queued in the separate DMA unit hardware buffers.

An incoming packet which is transferred by DMA to SRAM will not generate interrupts via RCTL. There are three possible interrupts which can be generated, depending on the fullness of the SRAM buffer:

not Empty when the RAM buffer contains at least one word.

Full when the RAM buffer contains no more free space and the input is stalling.

Limit when the RAM buffer contains at least as many words as indicated in the programmable 'limit' register (spDMAl).

A DMA channel can be disabled by setting N, C and M to 0. This will prevent packets entering the unit; any packets already present will proceed normally unless deliberately flushed. The presence of packets in the DMA *hardware* buffer is indicated by the ?? bit, which will be 1 when there are no words waiting for transfer to RAM. This status bit is *not* exported for interrupt purposes because the software only needs informing about the words accessible in memory under most circumstances.

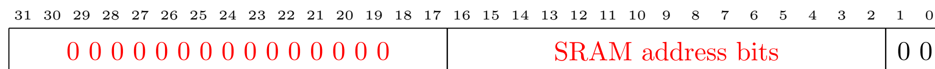
E, F, L and V reflect the levels on the DMA interrupt signals sent to the interrupt controller.

The {K, P, S} bits allow the selection of the packet fields which will be retained. A '1' bit means the appropriate field is stored.

> Example: setting spDMAc to 0x00000411 will filter out only MC packets without a payload and store the key word (only) in SRAM. This setting may be useful for incoming neural spikes.

r65-68, r81-84: spDMAs, e, r, w0; spDMAs, e, r, w - DMA start, end, read pointer and write pointer registers

These registers notionally hold 32-bit addresses defining the DMA channel start, end, read pointer and write pointers.



The upper 17 and the lowest 2 bits of these registers are hard-wired to zero, meaning the pointers are constrained to word addresses in the local SRAM. The justification allows easy software comparisons. Software cannot modify the write pointer, which is used by the DMA processor, except to reset it. Resetting can be explicit, through the F/R???? bit(s) in the appropriate spDMAc or implicit in that both pointers are reset to the spDMAs value when either spDMAs or spDMAe is written ???

The maximum used capacity of the SRAM buffer is one word less than the maximum space. Only when the buffer is empty will the read and write pointers be equal. This is for ease of software disambiguation.

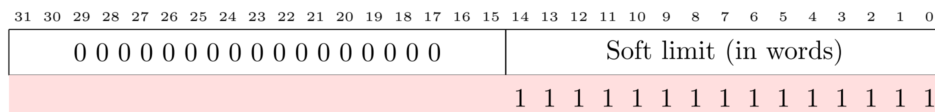
> Example: setting spDMAs to 0x00001000 and spDMAe to 0x0000101C will provide a maximum capacity of 7 words, although all 8 words will be written to over time as the buffer is cycled.

r69, r85: spDMAl - DMA buffer soft limit

These registers contain a number indicating how many unprocessed words are 'expected' in the appropriate DMA SRAM buffer. If the number of words present equals or exceeds this limit a status bit is set and an interrupt may be raised.

It is anticipated that this limit may be useful as a high-priority warning that the associated input buffer is approaching fullness, with potential consequences for stalling or overrunning.

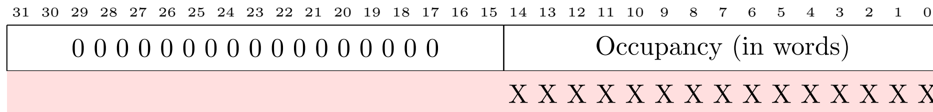
The reset value is the maximum value which means that this will not usually occur until the register is programmed with practical limit.



r70-71, r86-87: DBO - DMA Buffer Occupancy, DMax ('tidemark')

These 15 bit, read-only registers monitor the current 'fullness' of the associated SRAM buffer. The contents indicate a count, in words, of the distance between the write and read pointers. DBO represents the instantaneous state of the buffer; DMax represents the highest value DBO has reached since it was last reset: a buffer occupancy 'tidemark'.

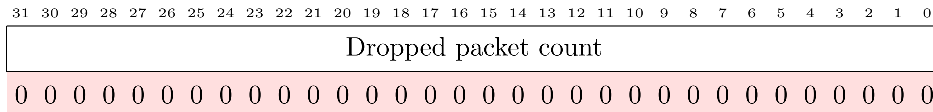
DMax is reset by writing a '1' to the 'T' bit in the DCTL register.



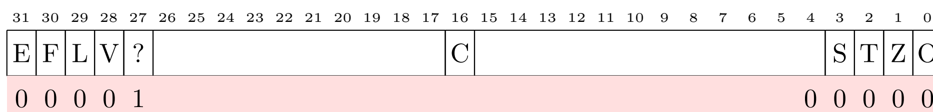
The implemented bits of these 'registers' are unknown after reset as they need valid start and end addresses, and read and write pointers setting up before they can be evaluated. The DMax value should be explicitly 'cleared' before it is first used (which will set it to the current value in DBO).

r72, r88: DDC - DMA Dropped packet Count

If the DMA channel is set to be able to drop packets in the event of an overrun the tally of dropped packets is kept here. This counter is reset by writing a '1' to the Z bit in DCTL. It saturates at 0xFFFFFFFF.



r79, r95: DCTL - DMA Control register

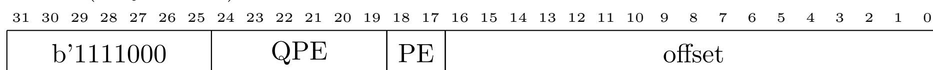


Name	bits	R/W	Function
E!: empty	31	R	spDMA buffer not empty
F!: full	30	R	spDMA buffer full
L!: limit	29	R	spDMA buffer soft limit reached/exceeded
V!: overrun	28	R	spDMA overrun count non-zero
?: FIFO empty	27	R	Internal *hardware* FIFO empty
C: flush	16	W	flush spDMA *hardware* buffer
S: Supervisor	4	R/W	Privilege of request (1 = supervisor)
T: clear max.	2	W	spDMA 'tidemark' reset
Z: zero overrun	1	W	spDMA overrun count reset
O: overrun enable	0	R/W	spDMA permit overrun

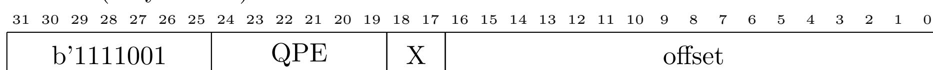
14.5 Bus bridge

Any PE has access to any SRAM location and the peripheral memory-mapped IO on the chip although there are perfomance penalties and some accesses will be slow. Logically, the addressable space can be divided into the SRAM and similar memory mapped IO devices and the (much larger) SDRAM. These are both reachable through (different) regions in the M4's address space. The SRAM is reached in the space between *** TBC ***; addresses in this area are separated into a physical NoC address and an offset within the particular device.

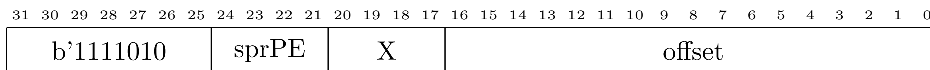
QPE SRAM (only DNoC):



Register Files (only CNoC):

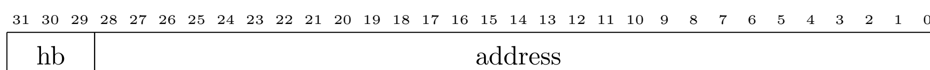


SpiNNaker Router (only DNoC)



sprPE	Router	PE
0	0	0
1	0	1
2	0	2
3	1	1
4	1	2
5	1	3

SDRAM (only DNoC):



Register field *global_segment[1]* (see *GCTL* register) selects the LPDDR4 interface. The address on the SDRAM interface is calculated by:

$$((hb - 1 + 8 \cdot global_segment[0]) << 29) + address$$

hb is in the range of 1-6 and overflowing possible.

This address space is not 'full'; some QPE addresses have no equivalent on the chip and this part of the address fundamentally reflects the physical NoC layout. Some 'QPE' addresses are not QPEs – notably the SpiNNaker router and the peripheral IO – and use the offsets in their own, specific ways.

Accesses are translated from the instruction on the M4 AHB so can be 8-, 16- or 32-bit operations. Usually the M4 will be stalled until the operation is complete; **note that this stall can be for a considerable time during which the M4 will be waiting and will *not respond to interrupts.*** To alleviate this, *write* operations can be buffered as 'fire and forget' * need to say how *; if this is done any bus errors wil be ignored.

As an alternative, such remote operations may be performed in software by sending an enquiry packet and reading the corresponding response at some later time.

SDRAM accesses are available in a similar fashion. These appear in the M4 address space in the address range 0x20000000-0xDFFFFFFF. Because the SDRAMs are larger than this available space they are 'paged', see GCTL register, global segment.

14.6 Monitoring

The communications unit contains a number of features to assist run-time monitoring.

1. The occupancy of the DMA RAM buffers is calculated in hardware (which saves software effort). More importantly, the *maximum* value this occupancy has reached (since it was last reset) is retained to allow periodic checking to see if the allocated buffer size is appropriate.

   A soft limit can be imposed on this count which will raise a status/interrupt signal if it is reached or exceeded, facilitating remedial action before the buffer is completely filled.
2. If receiver/spDMA overrunning is enabled, a count of any packets dropped is maintained.
3. Other monitor features? Timer/rates?

14.7 Fault-tolerance

Fault insertion

1. Software can cause the Communications Controller to misbehave in several ways including inserting dodgy routing keys, source IDs, destination IDs.

Fault detection

1. Attempts to overrun buffers or registers will cause data aborts.

Fault isolation

1. The Communications Controller is mission-critical to the local processing subsystem, so if it fails the subsystem should be disabled and isolated.
2. The ability to allow the various packet receivers to overrun can prevent back-pressure from the PE causing wider problems.and isolated.
   *** Perhaps there should be a remote (exception packet?) way of setting these bits – a bit like an interrupt – for remote alleviation of faults? ***

Reconfiguration

1. The local processing subsystem is shut down and its functions migrated to another subsystem on this or another chip. It should be possible to recover all of the subsystem state and to migrate it to a functional alternative.

15 NoC DMA Submodule (memDMA)

15.1 Features

1. write a block of data from PE's local SRAM to global memory
2. read a block of data from global memory to local PE's SRAM

15.2 Description

Fig. 19 shows the architecture of SDRAM DMA (memDMA). There is an AHB interface, an input FIFO, two Finite-State-Machines (FSMs) and a 128-bit master interface. The input FIFO is of depth of one packet. Between the two FSMs, one is responsible for reading – which moves remote data to local memory – and the other is responsible for writing – which moves local data to remote memory (SRAM or SDRAM).

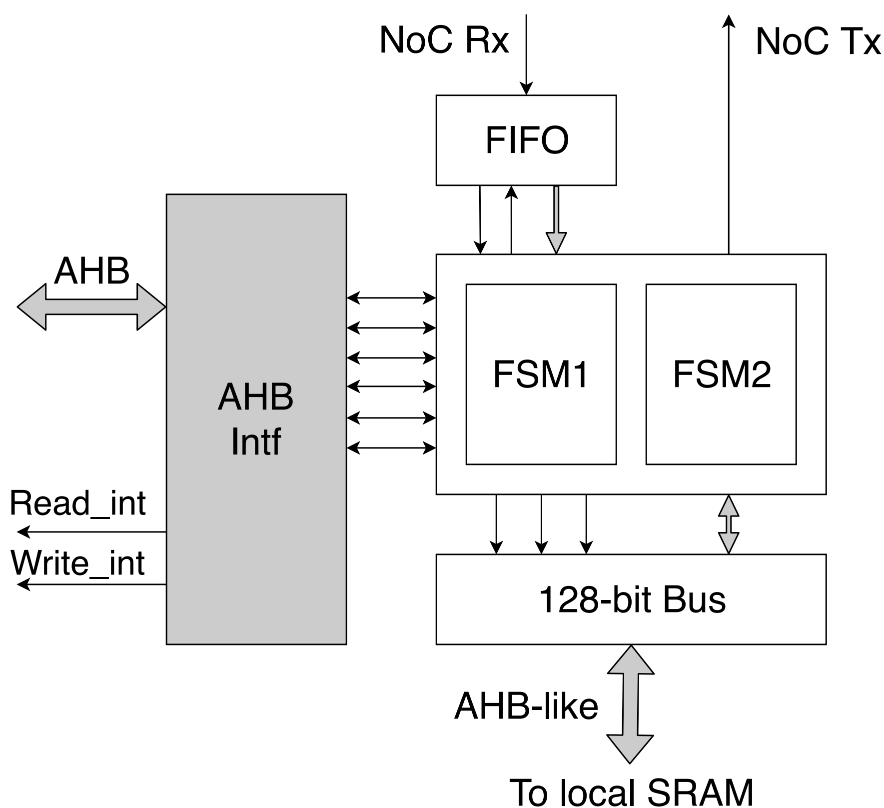


Figure 19: SDRAM DMA organisation

Before initiating a transfer task, some registers must be configured. The 'local_address' register points to the local memory, whose range is 0x00000-0x1FFFF. The 'remote_address' refers to remote global address, which is of 32-bit width. 'len' iss the size of the block to be transferred in words. Only word aligned operation is supported. The 'config' register controls the functions of SDRAM DMA: for example, 'dest_location' decides the Dest_QPE and Dest_PE position; 'C' decides the reading or writing packets are routed via the CNoC or DNoC; 'B' decides to perform a buffered write or not; 'S_remote' decides the remote counterpart performs a privileged access or not; 'S_local' decides the accessing to local memory with a privileged access or non-privileged access; 'mode' decides when to de-assert the request signal. Register 'done_count' shows the number of bytes that finished transferring. Register 'status' reports the running status or running results to processor. Register 'cmd' starts a transfer task or clears the status register.

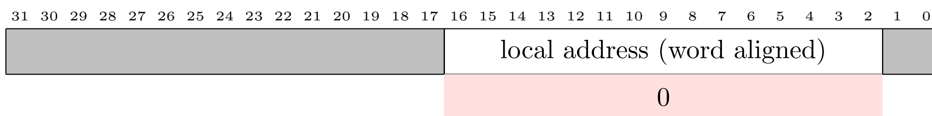
15.3 Register summary

The bit maps of register 'config', 'status' and 'cmd' AND MORE are listed below.

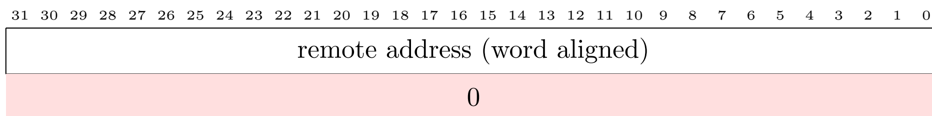
Name	Offset	R/W	Function
r96: local_address	0x180	R/W	Address within local SRAM
r97: remote_address	0x184	R/W	Address anywhere in chip
r98: len	0x188	R/W	length of transfer
r99: config	0x18C	R/W	DMA Configuration
r100: dma done	0x190	R	how many word transfers were done in this DMA transfer
r101: status	0x194	R	DMA status
r103: row mask	0x19C	R/W	row mask
r110: clear	0x1B8	W	Clear registers
r111: cmd	0x1BC	W	DMA command

*** More channels? Double buffering? Occupancy status? ***

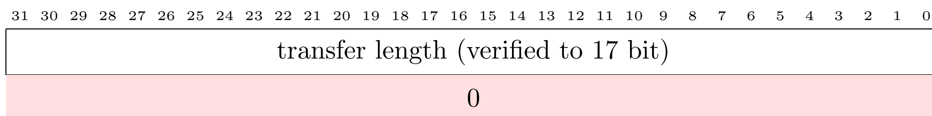
r96: local_address



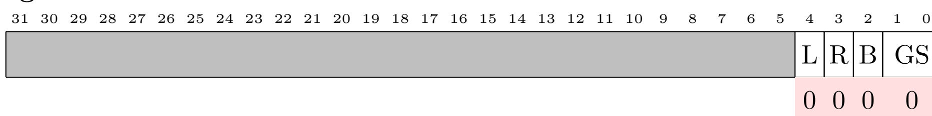
r97: remote_address



r98: len

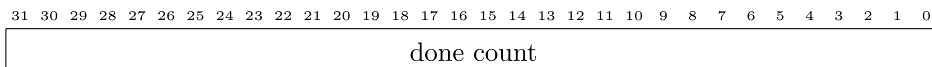


r99: config



Name	bits	R/W	Function
L: S_local	4	R/W	use local priviliged access
R: S_remote	3	R/W	use remote priviliged access
B: Buffered write	2	R/W	Use NoC with buffered write protocol
GS: global segment	1:0	R/W	select global address segment. GS[1]: 0: MEM A, 1: MEM B, GS[0]: upper / lower segment

r100: dma done



Name	bits	R/W	Function
done count	31:0	R	how many word transfers were done in this DMA transfer

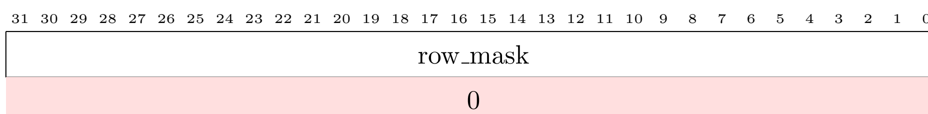
r101: status

Name	bits	R/W	Function
SD: SRAM DMA	15	R	SRAM DMA active
P: DMA	14	R	DMA active
D: Done	13	R/W	All DMA operation is done
OE: Output enable	12	R	output is enabled, no pending interrupts
WN: Write enable	11	R	DMA configuration is enabled
S: Running State	10:8	R	DMA running state (interpretation depending on read or write operation running). For states see table below.
WI: write interrupt	7	R	write_interrupt
RI: read interrupt	6	R	read_interrupt
WE: Write error	5	R	write_error
Wr: writing	4	R	writing
RE: Read error	3	R	read_error
R: Reading	2	R	reading
N: !Error	1	R	no_error
W: Working	0	R	working

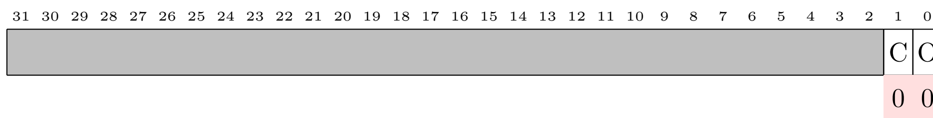
PE2PE		
Running State	**Read FSM**	**Write FSM**
0	IDLE	IDLE
1	WAITING_GNT	READ_REQ
2	READ_REQ	WAITING_RESP
3	WAITING_RESP	WAITING_GNT
4	WRITING_PKT	TX_REQ
5	WRITING_DONE	WRITE_RESP
6	CHECK_DONE	CHECK_DONE
7	WAITING_OUTPUT	WAITING_OUTPUT

PE2SDRAM		
Running State	**Read FSM**	**Write FSM**
0	IDLE	IDLE
1	READ_REQ	CMD_OUT
2	WAITING_RESP	WAITING_BLK_REQ
3	WRITING_PKT	READ_REQ
4	WRITING_DONE	WAITING_RESP
5	CHECK_DONE	TX_REQ
6	REQ_NXT_BLK	CHECK_DONE
7	-	FINISHED_CONFIRM
8	ABORT	ABORT
9	ABORT_RESP	ABORT_RESP
10	WAITING_OUTPUT	WAITING_OUTPUT

r103: row_mask

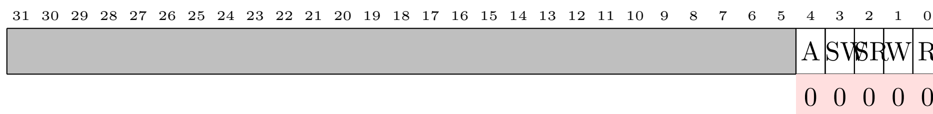


r110: clear



Name	bits	R/W	Function
C: Command	1	W	clear command
O: Output	0	W	clear output

r111: cmd



Name	bits	R/W	Function
A: Abort Command	4	W	abort current command (Only SDRAM DMA)
SW: SDRAM write	3	W	start SDRAM write
SR: SDRAM read	2	W	start SDRAM read
W: write	1	W	start DMA write
R: read	0	W	start DMA read

15.3.1 Flow control

To regulate the pressure of requests on the SDRAM controller(s) each **QPE** is restricted to a single outstanding DMA transaction. The four SDRAM DMAs in one QPE are therefore not allowed to send simultaneously. An arbiter is employed and every SDRAM DMA needs to request permission before sending each request or set of requests.

Request is asserted before sending. The 3-bit mode register decides when to de-assert the request signal. If mode[0] is set, the request signal is de-asserted as soon as this DMA is granted; if mode[1] is set, the request signal de-asserts only after all the response packets corresponding to the request packet just sent out are back; if mode[2] is set, request de-asserts only after the whole transfer task is finished.

The control of this module is implemented as four FSMs, one for reading and one for writing for SDRAM and DMA each. The FSMs are independent. However, they cannot run simultaneously.

A special command is the abort command. An initiated SDRAM read/write will be aborted. This command can be send, even if a read/write is running. The abortion takes time to happen, therefore the user must wait for the done signal.

15.3.2 Local bus masters

The local memory has a datawidth of 32bit, while a NoC packet has a size of 128bit. To still allow one cycle read/write accesses from NoC to the memory the virtual 128bit interface is physically split into 4 local masters, which have a datawidth of 32 each. Between two consecutive local masters the address offset is constant with a value of 4. For accesses where single datawords are not needed the corresponding local master will not initiate a transaction.

16 Bus bridge to NoC

16.1 Features

1. Supporting feature.
2. Interval and choc-ice.
3. Main feature.

16.2 Description

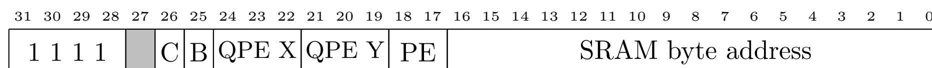
The bridge (Peek/Poke submodule) provides a method for the M4 to access remote registers/SRAM. A large part of the M4 **system bus** address space is bridged onto the NoC. The mechanism is implemented by suspending the M4 bus operation whilst a *request* packet is transmitted and a *response* packet is returned. Operation is therefore transparent to software except for the speed of the operation. There is a corresponding, hardware *response* unit in each PE which translates *request* packets into local SRAM cycles and thence to *response* packets.

To extend the address space . . . mapping table for SDRAM . . . is this present? It needs its REGISTERS DEFINING AND DOCUMENTING!

System bus address decode:

From	To	Meaning
2000_0000	DFFF_FFFF	SDRAM(s) via NoC
E000_0000	E00F_FFFF	local APB I/O
E010_0000	EFFF_FFFF	local AHB I/O
F000_0000	FFFF_FFFF	NoC-level bus

The **SUGGESTED** address translation for the PEs is as follows:
Bits [16:0] address the location within a PE SRAM.
Bits [18:17] address the PE within a QPE.
Bits [24:19] address the (notional) QPE. *** VERIFY ***



Where is the *write buffer* bit set up – if it is. Is this another address bit? A[25] suggested above. It would need to be in the SDRAM page tables too.
Similarly, suggesting A[26] for use CNoC, rather than DNoC.
What about 'S'? Below it says GCTL, but this seems the wrong approach; certainly breaks privilege limitation. Surely just bridge incoming request?

16.2.1 SDRAM mapping

It is anticipated that the SDRAM sizes available will exceed the 4 GiB address space by the time the final device is complete. To anticipate this, there is a simple mapping scheme in the SDRAM area (0x2000_0000-0xDFFF_FFFF). This area is divided into six 512 MiB pages and the physical addresses for these is looked up in a small, programmable table. The output form is a 34-bit byte address (plus a write-buffer bit??). The extra address bits are carried as part of the NoC unit address as it is anticipated that the SDRAM interface required will comprise four separate units.

The page table entries are accessible directly in I/O space at addresses 0xE???_????-0xE???_????.

A[31:29]	Entry address	Purpose
0 0 0	0xE???_??00	Unavailable – mapped to local buses
0 0 1	0xE???_??04	SDRAM extended page address (or NoC port?)
0 1 0	0xE???_??08	SDRAM extended page address (or NoC port?)
0 1 1	0xE???_??0C	SDRAM extended page address (or NoC port?)
1 0 0	0xE???_??10	SDRAM extended page address (or NoC port?)
1 0 1	0xE???_??14	SDRAM extended page address (or NoC port?)
1 1 0	0xE???_??18	SDRAM extended page address (or NoC port?)
1 1 1	0xE???_??1C	Unused – I/O and SRAM space

This mapping is not required for SRAM (and other) addresses as there is adequate address space available in the standard 4 GiB map.

DMA addressing uses unmapped, 34-bit 'physical' addresses. DMA accesses are always privileged(??) and always use DNoC(??).

16.2.2 Operation

When the processor reads from an address within this range, the Peek/Poke submodule de-asserts 'HREADY' signal, suspending the processor, and sends out a read request packet. Then it waits until the response packet arrives, translates the payload of the response packet into 'HRDATA', and asserts 'HREADY'.

The write process is similar although writes can be *buffered*, in which case no response packet is expected or returned and the processor is allowed to continue once the request packet has been formed.

The data bus of AHB is of 32-bit width, therefore, the read/write size of the NoC packet can only be 1 byte, 2 bytes or 1 word.

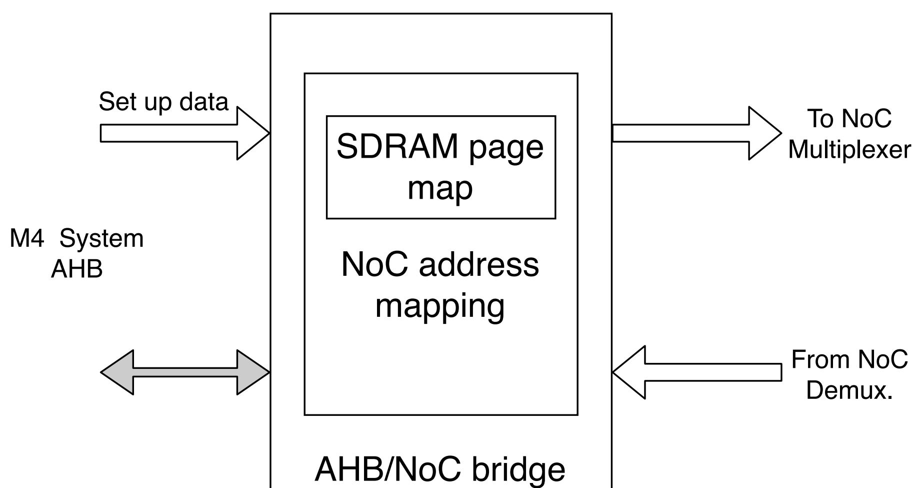


Figure 20: Block diagram of the of Peek/Poke response unit

The address the processor issues is a local address. However, The address in NoC packet is a global address. Peek/Poke submodule instantiates the NoC_addr_map module for the translation from local address to global address.

The bridge unit is using the B field of the GCTL register.

17 Response unit

This unit is not really part of the PE's communication system, being a bus bridge from the NoC *into* the PE's SRAM. It has no direct connection with the software on the appropriate PE.

Request packets arriving from the NoC are routed to this unit where they cause local bus activity – either read(s) or write(s). The appropriate **response** packet(s) are generated and routed back to the source interface, as specified in the request. Zero (in the case of a write, marked as 'buffered') to eight (128 byte read) response packets may be generated for each incoming request.

17.1 Response packet generator submodule

The response packet generator answers read request packets or write request packets, no matter the request packets come from Tx module, Peek/Poke submodule or SDRAM DMA submodule of another PE. In addition, response packet generator answers read request packets generated by ML-Acc, but doesn't answer write request packets generated by the ML-Acc.

There is one FIFO at the incoming direction as well as outgoing direction for NoC packets. Both FIFOs are of 1 packet depth.

When answering read request packets, there may be 1, 2, 4, or 8 response packets corresponding to 1 read request packet. When answering multiple packets, for example 8 packets, and bus error occurs during one of mid-packets, there are two options for the response packet generator:

(1) When bus error occurs, the reply stops; (2) The response packet generator answers all the requested packets. If one of the packets encounters bus error, then the corresponding response packet shows bus error (setting bit [160]).

Currently both the above options are implemented, and is selectable by a bit 'mode'. 'Mode' is the 8th bit of GCTL register, say, GCTL[8].

This doesn't work! If this is programmable then it cannot be done on a PE-by-PE basis because both correspondents need to agree on the behaviour. Either the behaviour needs to be set *at source* and conveyed in the *request packet* (Yuk!) or should be predetermined.

If the incoming write request packet requests a buffered write, that is, bit [161] is set, no response packet will be generated.

If the incoming read or write request packet is routed by CNoC, say, bit [177] is set, the response packet will also be routed by CNoC. And vice versa, if the incoming request packet is routed by DNoC, the response packet will also be routed by DNoC.

Okay – but how is this indicated? Is this another address bit?

Except bit 'mode' being configurable, the running of response packet generator is not visible and perceivable by the processor. It runs independently with the processor.

When answering the request packets, the response packet generator needs to access to local memory(SRAM). In order to access to memory efficiently, the interface is designed to be an AHB-like bus of 128-bit data width, which is transformed into 4 32-bit buses in the bus interface module.

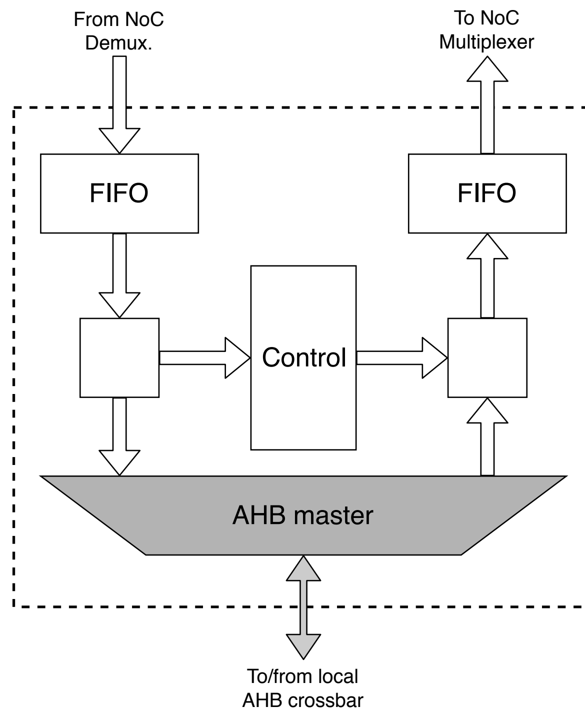


Figure 21: Block diagram of the response packet generator

18 SpiNNaker Packet Router

The Router is responsible for routing all packets that arrive at its inputs to one or more of its outputs. It is responsible for routing multicast neural event packets, which it does through an associative multicast router subsystem, core-to-core packets (for which it uses a look-up table), nearest-neighbour packets (using a simple algorithmic process), global read/write packets (using the C2C look-up table) and default routing (when a multicast packet does not match any entry in the multicast router).

Various error conditions are identified and handled by the Router, for example packet time-out and unroutable packets.

18.1 Features

1. Support for 4 packet types:
   (a) multicast (MC) neural event packets routed by a key provided at the source;
   (b) core-to-core (C2C) packets routed by destination address to any core on any chip;
   (c) nearest-neighbour (NN) packets routed by arrival port;
      i. in both normal and peek/poke forms.
   (d) global read/write (GRW) packets routed by destination address to any core on any chip and the reponse is routed by a default source address (configurable in each chip);
2. flexible packet features:
   (a) support for 40-bit packets with optional 32-, 64- and 128-bit payloads;
   (b) 2-bit time phase (used by Routers to trap errant packets);
3. 16,384 programmable associative multicast (MC) routing entries.
   (a) associative routing based on packet;
   (b) with flexible 'don't care' masking.
4. default routing of unmatched multicast packets.
5. look-up table routing of core-to-core (C2C) packets and global read/write (GRW) packets.
6. routing of nearest-neighbour (NN) packets.
7. independent programmable wait times for MC and C2C/NN/GRW packets.
8. hardware dropped packet handling
   (a) hardware dropped packet buffer;
   (b) hardware support for automatic dropped packet re-insertion;
   (c) software recovery for dropped packets.
9. pipelined implementation to route 1 packet per cycle (peak).
   (a) back-pressure flow control;
   (b) power-saving pipeline control.
10. out-of-order issue.
11. performance counters.
12. fault detection and handling:
    (a) expired time phase;
    (b) unroutable packet;
    (c) illegal packet;

18.2 Description

SpiNNaker packets arrive from other nodes via the link receiver interfaces and NoC packets from internal processor nodes and are presented to the router through its 6 NoC ports. These packets are decoded, arbitrated and transmitted by a front-end crossbar to different internal engines in parallel.

Each multicast packet contains an identifier which is used by the Router to determine which of the outputs the packet is sent to. These outputs may include any subset of the output links, where the packet may be sent via the respective link transmitter interface, and/or any subset of the internal processor nodes, where the packet is sent to the respective Communications Controller.

For a neural network application the identifier can be simply a number that uniquely identifies the source of the packet — the neuron that generated the packet by firing. This is 'source address routing'. In this case the packet need contain only this identifier, as a neural spike is an 'event' where the only information is that the neuron has fired. The Router then functions simply as a look-up table where, for each identifier, it looks up a routing word, where each routing word contains 1 bit for each destination (each link interface and each local processor) to indicate whether or not the packet should be passed to that destination.

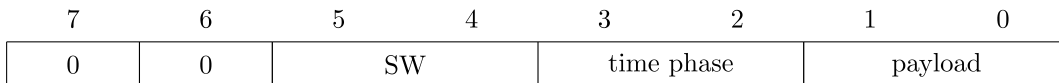
18.3 Packet formats

Neural event multicast (MC) packets (type 0)

Neural event packets include a control byte and a 32-bit routing key inserted by the source. In addition they may include an optional 32-, 64- or 128-bit payload:

8 bits	32 bits	32, 64 or 128 bits
control	routing key	optional payload

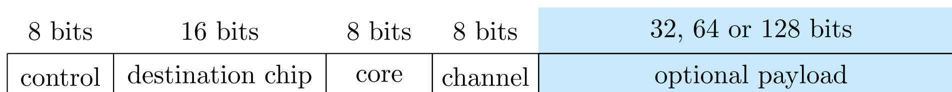
The 8-bit control field includes packet type (bits[7:6] = 00 for multicast packets), two software controlled bits, time phase and payload information:



7	6	5	4	3	2	1	0
0	0	SW		time phase		payload	

Core-to-core (C2C) packets (type 1)

Core-to-core packets include a 16-bit destination chip ID and an 8-bit destination core ID, an 8-bit channel ID (which is for software use), plus a control byte and an optional 32-, 64- or 128-bit payload:



8 bits	16 bits	8 bits	8 bits	32, 64 or 128 bits
control	destination chip	core	channel	optional payload

The C2C packets are directed to a specific core, rather than just the Monitor core, on a specific chip. It is assumed that a protocol will associate a particular channel number with a sender/receiver pair, and this will be initiated by the sender first sending a communication request that includes the sender (source) chip and core IDs as payload, to which the receiver will respond by allocating and sending back a channel ID, though of course protocol details are the responsibility of the software.

Here the 8-bit control field includes packet type (bits[7:6] = 01 for C2C packets), two software controlled bits that may be used for packet sequence information, time phase and payload information:

7	6	5	4	3	2	1	0
0	1	SW		time phase		payload	

Nearest-neighbour (NN) packets (type 2)

Nearest-neighbour packets include a 32-bit address or operation field, plus a control byte and an optional 32-, 64- or 128-bit payload:

8 bits	32 bits	32, 64 or 128 bits
control	address/operation	optional payload

Here the 8-bit control field includes packet type (bits[7:6] = 10 for NN packets), a 'peek/poke' or 'normal' type indicator (T), routing and payload information:

7	6	5	4	3	2	1	0
1	0	T	route			payload	

Global read/write (GRW) packets (type 3)

Global read/write packets include a 16-bit destination chip ID and an 8-bit source host ID, a 2-bit time phase, a 6-bit software controlled bits, plus a control byte and a 32-bit operation address in the payload:

8 bits	16 bits	8 bits	2	6 bits	32, 64 or 128 bits
control	destination chip	host ID	T	SW	payload

The GRW packets allow an on-chip initiator to read/write data from/to a specific address on any chip of the whole system. The 16-bit destination chip ID and the 32-bit target address are carried in the packet. The 8-bit host ID of the initialiator is also carried in the packet and will be used to return the read/write response. The 16-bit source chip ID is fixed but configurable in each chip. Compared with the C2C packets, GRW packets provide a hardware solution for transmitting data without software intervention. In contrast, the C2C packets provide a flexible software solution that can be used for data transmission and system messaging.

Here the 8-bit control field includes packet type (bits[7:6] = 11 and bits[5] = 0 for GRW packets), a 'router-to-router' flow control indicator (F), 3-bit operation code (OP) and payload information:

7	6	5	4	3	2	1	0
1	1	F	operation code			payload	

18.4 Control byte summary

The various fields in the control bytes of the different packet types are summarised below:

Field Name	bits	Function
payload	1:0	no (00) or 32-bit (01), 64-bit (10) or 128-bit (11) payload
time phase	3:2	phase marker indicating time packet was launched
route	4:2	NN only: information for the Router
SW	5:4	for software use - e.g. sequence numbers in C2C packets
T	5	NN only: packet type - normal (0) or peek/poke (1)
F	5	= 0 for router-to-router flow control; = 1 for GRW
operation code	4:2	global read/write packet operation code
packet type	7:6	= 00 for MC; = 01 for P2P; = 10 for NN; = 11 for GRW

payload

Indicates whether the packet has no data payload (= 00), or has a 32-bit (= 01), 64-bit (= 10) or 128-bit (= 11) payload. Peek/poke NN packets only support no data payload (= 0) or a 32-bit payload (= 1).

time phase

The system has a global time phase that cycles through 00 $\rightarrow$ 01 $\rightarrow$ 11 $\rightarrow$ 10 $\rightarrow$ 00. Global synchronisation must be accurate to within one time phase (the duration of which is programmable and may be dynamically variable). A packet is launched with a time phase equal to the current time phase, and if a Router finds a packet that is two time phases old (time now XOR time launched = 11) it will drop it to the local Monitor Processor. The time phase is normally inserted by the local Router at the packet source unless a packet is being inserted under unusual circumstances, such as the re-insertion of a dropped packet, when the sending core can specify the time phase to be used.

route

These bits are set at packet launch to the values defined in the control register. They enable a packet to be directed to a particular neighbour (0 - 6), broadcast to all or a subset (as defined in the Router r1 'NN broadcast' bits on page 93) of neighbours (7).

SW

This bit is used by software for packet sequence control and similar purposes.

T (NN packet type)

This bit specifies whether an NN packet is 'normal', so that it is delivered to the Monitor Processor on the neighbouring chip(s), or 'peek/poke', so that it performs a read or write access to the neighbouring chip's addressable resources.

F (router-to-router flow control)

This bit defines whether the packet is used for router-to-router flow control or GRW packets. $F = 1$ is reserved for hardware use and $F = 0$ is for GRW packets

operation code

These bits indicate whether the GRW packet is a 32-bit read (000), 64-bit read (001), 128-bit read (010), unbuffered write (011), buffered write(100), read response (101), write response (110) or error response (111).

packet type

These bits indicate whether the packet is a multicast (00), core-to-core (01) or nearest-neighbour (10) packet or global read/write (11).

18.5 Debug access to neighbouring devices

The 'peek' and 'poke' mechanism gives access to the NoC address space on any neighbouring device without processor intervention on that chip. To read a word (peek), include its address in a 'peek/poke' nearest neighbour packet output (i.e. with T = 1, no payload). Only word addresses are permitted. A write (poke) carries a 32-bit payload. Peeks and pokes would normally be done by a Monitor Processor although, in principle, any processor can output these packets.

In the case of a peek, the target device performs the appropriate access and returns a response on the corresponding link input. This is delivered to the processor designated as Monitor Processor in the local router. The response is a 'normal' NN packet which carries the requested word as a 32-bit payload. The address field is also returned for identification purposes with the lower two bits = 10; = 11 indicates that the access caused a bus error; = 01 indicates that the destination bus master is full and the request should be retransmitted later.

Writing ('poke') is similar; including a 32-bit payload in the outgoing packet causes that word to be written. A no payload normal response packet is returned which indicates the error status in the address field.

Only no or 32-bit payloads are supported by peek and poke NN packets.

18.6 Internal organization

The internal organization of the Router is illustrated in Fig. 22.

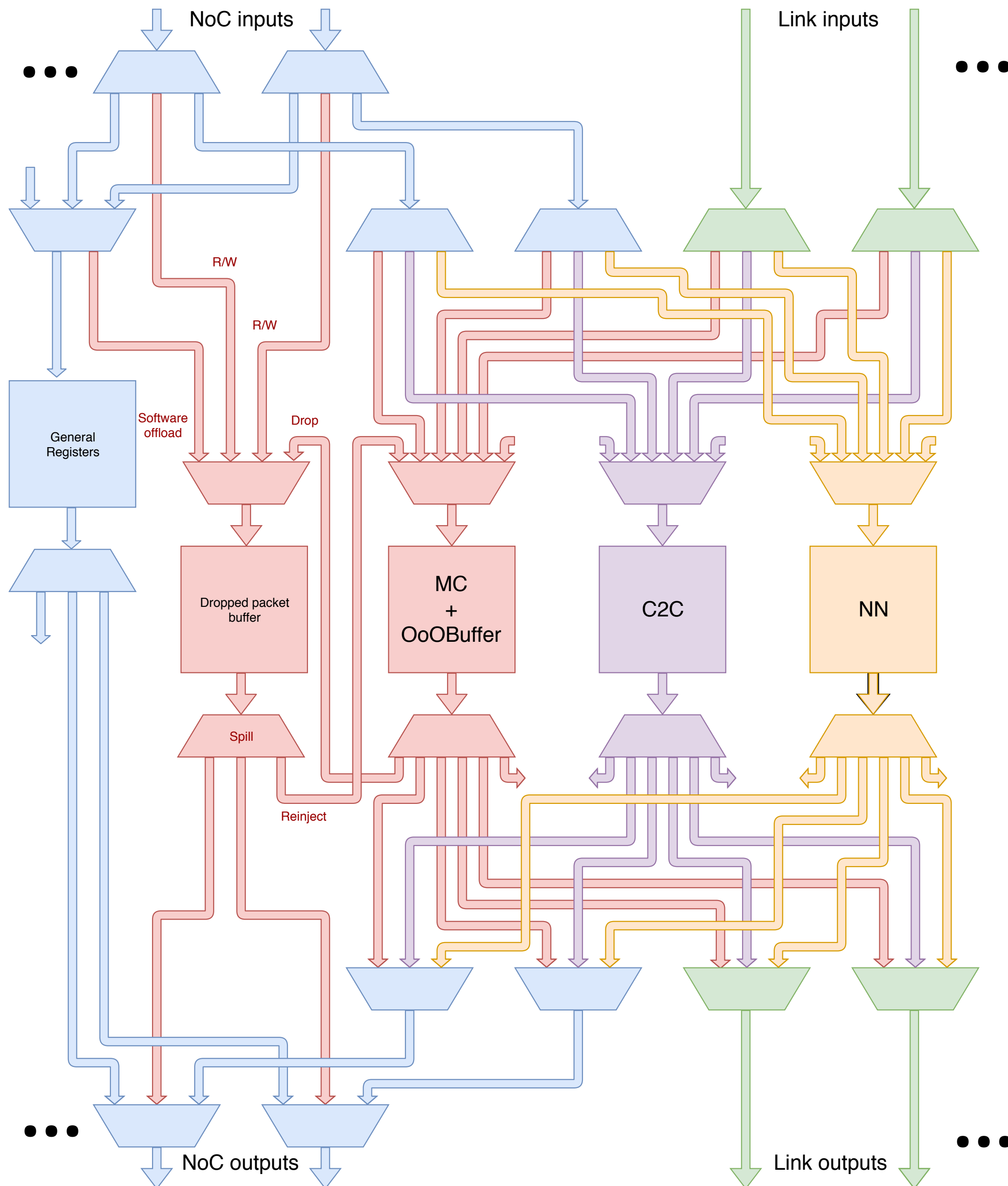


Figure 22: Router organisation

Packets are passed as complete 40- to 168-bit units from the inter-chip links and 64- to 192-bits from the NoC ports, together with the identity of the Rx interface that the packet arrived through (for nearest-neighbour, default routing and packet reinsertion). The input stage of processing is to identify errors and pass packets to appropriate units. The routing stage contains different routing engines – the multicast (MC) router is activated when the packet is of multicast type, the core-to-core (C2C) router handles core-to-core and gobal read/write packets while the NN router handles nearest-neighbour packets and contains a bus master. The output of the router stage is a vector of destinations to which the packet should be relayed.

18.7 Multicast (MC) router

The MC router uses the routing key in the MC packet to determine how to route the packet. The router has 16,384 look-up entries, each of which has a mask, a key value, and an output vector. The packet's routing key is compared with each entry in the MC router. For each entry it is first ANDed with the mask, then compared with the entry's key. If it matches, the entry's output vector is used to determine where the packet is sent; it can be sent to any subset (including all) of the local processors and the output links.

Thus, to programme an MC entry three writes are required: to the key, its mask and the corresponding vector. A mask of FFFFFFFF ensures all the key bits are used; if any mask bits are '0' the corresponding key bits should also be '0', otherwise the entry will not match. This can be exploited to ensure that unused entries are invalid. The effect of the various combinations of bit values in the mask[] and key[] regions is summarized in the table below:

key[]	mask[]	Function
0	0	don't care - bit matches
1	0	bit misses - entry invalidated
0	1	match 0
1	1	match 1

Thus a particular entry [i] will match only if:

1. wherever a bit in the mask[i] word is 1, the corresponding bit in the MC packet routing word is the same as the corresponding bit in the key[i] word, AND
2. wherever a bit in the mask[i] word is 0, the corresponding bit in the key[i] word is also 0.

Note that the MC Router CAM is not initialised at reset. Before the Router is enabled all CAM entries must be initialised by software. Unused mask[] entries should be initialised to 00000000, and unused key[] entries should be initialised to FFFFFFFF. This invalidates every bit in the word, ensuring that the word will miss even in the presence of minor component failures.

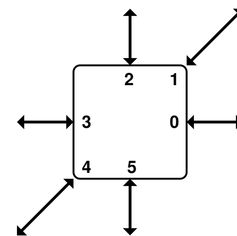
The matching is perfomed in a parallel ternary associative memory, with a two-stage RAM used to store the output vectors. The first stage of the RAM stores the output vectors for inter-chip links plus one bit to indicate whether there are any internal destinations. The second stage stores the output vectors for the 152internal PEs.

The associative memory can be set up so that more than one entry matches an incoming routing key; in this case the matching entry at the lowest address determines the output vector to be used. Multiple simultaneous matches can also be used to improve test efficiency.

If no entry matches an MC packet's routing key then default routing is employed - the packet is sent to the output link opposite the input link through which it arrived. Packets from local processors or external link cannot be default-routed; the router table must have a valid entry for every locally-sourced packet and the packet from external link, otherwise the packet will be unroutable.

The first stage MC output vector assignment is detailed in the table below:

MC vector entry	Output port	Direction
bit[0]	Tx0	East
bit[1]	Tx1	North-East
bit[2]	Tx2	North
bit[3]	Tx3	West
bit[4]	Tx4	South-West
bit[5]	Tx5	South
bit[6]	Tx6	external
bit[7]	internal	Local



The second stage MC output vector assignment similary extends the internal outputs to any or all of Processors 0 to 151 .

If any of the multicast packet's output links are blocked or unavailable the packet is stalled for a programmable 'wait' time (see 'r0: Router control register' on page 93). When the multicast out-of-order (OoO) issue function is enabled, the subsequent packets in the OoO buffer may be issued ahead of it if their output ports are all clear.

The out-of-order (OoO) routing mechanism is for issuing packets whose output routes are clear, which it detects using 'full' signals fed back from the individual destination output buffers. The packet which is at the front of the OoO buffer is issued if its outputs are clear; if it cannot be issued, the second packet in the queue is issued if possible; if not, the third, and so on, up to the capacity of the buffer.

If a packet remains unissued at the front of the the OoO buffer for a time defined by a programmable 'wait' counter it is 'dropped' to remove the risk of deadlock. Dropped packets are stored in a buffer, and are re-inserted when the Router pipeline has capacity to accept them. The Monitor Processor is informed of dropped packets, and they can also be counted using Router diagnostic facilities.

18.8 The core-to-core (C2C) router

The C2C router uses the 16-bit destination chip ID in a core-to-core packet to determine which output the packet should be routed to. There is a 3-bit entry for each of the 64K destination chip IDs. Each 3-bit entry is decoded to determine whether the packet is delivered to a local core (as identified by the core ID field) or one of the seven output links, as detailed in the table below:

C2C table entry	Output port	Direction
000	Tx0	East
001	Tx1	North-East
010	Tx2	North
011	Tx3	West
100	Tx4	South-West
101	Tx5	South
110	Tx6	external
111	core	Local

The 3-bit entries are packed into an 8K entry x 24-bit SRAM lookup table. The 24-bit words hold entries 0, 8, 16, ... in bits [2:0], 1, 9, 17, . . . in bits [5:3], etc.

18.9 The nearest-neighbour (NN) router

Nearest-neighbour packets are used to initialise the system and to perform run-time flood-fill and debug functions. The routing function here is to send 'normal' NN packets that arrive from outside the node (i.e. via an Rx link) to the Monitor Processor and to send NN packets that are generated internally to the appropriate output (Tx) link(s). This is to support a flood-fill load process.

In addition, the 'peek/poke' form of NN packet can be used by neighbouring systems to access NoC resources. Here an NN poke 'write' packet (which is a peek/poke type with a 32-bit payload) is used to write the 32-bit data defined in the payload to a 32-bit address defined in the address/operation field. An NN peek 'read' packet (which is a peek/poke type without a 32-bit payload) uses the 32-bit address defined in the address/operation field to read from the NoC and returns the result (as a 'normal' NN packet) to the neighbour that issued the original packet using the Rx link ID to identify that source. This 'peek/poke' access to a neighbouring chip's principal resources can be used to investigate a non-functional chip, to re-assign the Monitor Processor from outside, and generally to get good visibility into a chip for test and debug purposes.

As the peek/poke NN packets convey only 32-bit data payloads the bottom 2 bits of the address should always be zero. All peek/poke NN packets return a response to the sender, with the lower two bits = 10. The lower 2 bits will be set to 11 if there was a bus error at the target. The lower 2 bits will be set to 01 if the destination bus master is full. Peeks return a 32-bit data payload; pokes return no payload.

18.10 Time phase handling

The Router maintains a 2-bit time phase signal that is used to drop packets that are out-of date. The time phase logic operates as follows:

1. locally-generated packets will have the current time phase inserted (where appropriate);
2. a packet arriving from off-chip will have its time phase checked, and if it is two phases old it will be dropped (dropped, and copied to the Error registers).

18.11 Packet error handler

The packet error handler is a routing engine that simply flags the packet for dropping to the Error registers if it detects the following:

1. a packet that is two time phases old;
2. an unroutable MC packet from a local source and the external link;

A Monitor Processor can be interrupted to deal with packets dropped with errors.

18.12 Register summary

Base addresses: 0xF4600000 $port_0$**, 0xF4800000** $port_1$**, 0xF4A00000** $port_2$**, 0xF4000000** $port_3$**, 0xF4200000** $port_4$**, 0xF4400000** $port_5$

Name	Offset	R/W	Function
r0: control	0x00	R/W	Router control register
r1: route & throttle	0x04	R/W	NN route and output throttling control
r2: reinsert control	0x08	R/W	packet reinsertion control
r3: router status	0x0C	R	Router interrupt status
r4: error header	0x10	R	error packet control byte and flags
r5-8: error payload	0x14-0x20	R	error packet data payload
r9: error routing	0x24	R	error packet routing word
r10: error status	0x28	R	error packet status
r11: diag enables	0x2C	R/W	diagnostic counter enables
r12: profiling ctr ctl	0x30	R/W	profiling counters control
r13: cycle ctr	0x34	R/W	counts Router clock cycles
r14: busy cyc ctr	0x38	R/W	counts Router busy cycles
r15: no wt pkt ctr	0x3C	R/W	counts packets that do not wait to be issued
r16: drp iteration ctr	0x40	R/W	counts packets which drop more than once
r17: oooi ctr ctl	0x44	R/W	out-of-order issue counter control
r24: grw id	0x60	R/W	GRW default response chip ID
r25: oooi ctr	0x64	R/W	counts MC packets that issued out-of-order
r32: drop header	0x80	R	dropped packet register control byte and flags
r33-36: drop payload	0x84-0x90	R	dropped packet register data payload
r37: drop routing	0x94	R	dropped packet register routing word
r38: drop reg status	0x98	R	dropped packet register status
r39: ofl header	0x9C	R	offload packet control byte and flags
r40-43: ofl payload	0xA0-0xAC	R	offload packet control data payload
r44: ofl routing	0xB0	R	offload packet routing word
r45: drop buf status	0xB4	R	dropped packet buffer status
r46: total rein ctr	0xB8	R	total reinserted packet counter
r47: spkt wl-ctr	0xBC	R	maximum packets in drop buffer
r48: wd wl-ctr	0xC0	R	maximum words in drop buffer
r49: illegal pkt hdr	0xC4	R	illegal packet header
r50: ack intrpt	0xC8	R	acknowledge interrupt
r51: tcam blk en	0xCC	R/W	TCAM blocks control
r54: link control	0xD8	R/W	link control
rF[N]: diag filter	0x200-23C	R/W	diagnostic count filters (N = 0-15)
rC[N]: diag count	0x300-33C	R/W	diagnostic counters (N = 0-15)
rT0: tbist ctrl	0xF00	R/W	TCAM BIST control register
rT1: tbist data	0xF04	R/W	TCAM BIST data register
DP[1023:0]	0x8000	R/W	dropped packet buffer
C2C[8191:0]	0x10000	R/W	C2C Router routing entries (8 3-bit entries/word)
key[16,383:0]	0x20000	W	MC Router key values
mask[16,383:0]	0x40000	W	MC Router mask values
route0[16,383:0]	0x60000	R/W	MC Router routing word values
route1[16,383:0]	0x80000	R/W	MC Router routing word values

18.13 Register details

r0: Router control register

Bits	31:24	23:16	15:8	7:6	5	4	3	2	1	0
Field	wait[7:0]	sys_wait[7:0]	MP[7:0]	TP	W	S	E	R	F	V
Reset	1 0 0 0 0 0 0 0	1 0 0 0 0 0 0 0	0 0 1 0 0 1 0 0	0 0			0	0	0	0

The functions of these fields are described in the table below:

Name	bits	R/W	Function
wait[7:0]	31:24	R/W	wait time before dropping MC packet
sys_wait[7:0]	23:16	R/W	wait time before dropping C2C/NN/GRW packet
SLtx[6:0]	23:17	R/W	SpiNNaker output links mask
MP[7:0]	15:8	R/W	Monitor Processor ID number
TP	7:6	R/W	time phase (c.f. packet time stamps)
W	5	W	re-initialise wait counter
S	4	W	re-initialise sys wait counter
E	3	R/W	enable error packet interrupt
R	2	R/W	enable dropped packet register interrupt
F	1	R/W	enable dropped packet buffer full interrupt
V	0	R/W	enable dropped packet buffer not empty interrupt

The wait time (defined by wait[]) is stored in a floating point format to give a wide range of values with high accuracy at low values combined with simple implementation using a binary pre-scaler and a loadable counter. The 8-bit field is divided into a 4-bit mantissa M[3:0] = wait[3:0] and a 4-bit exponent E[3:0] = wait[7:4]. The wait time in clock cycles is then given by:

$$\text{wait} = (M + 16 - 2^{4-E}).2^E \text{ for } E \leq 4;$$
$$\text{wait} = (M + 16).2^E \text{ for } E > 4;$$

Note that wait[7:0] = 0x00 gives a wait time of zero (meaning that the packet waits for one cycle, and if it is not issued in that cycle it is immediately dropped), and the wait time increases monotonically with wait[7:0]; wait[7:0] = 0xFF is a special case and gives an infinite wait time - wait forever.

If the wait counter is in an infiite wait status, writing a 1 to W (bit[5]) or S (bit[4]) will cause the active counter to restart from the new value written to it. This enables the Monitor Processor to clear a deadlocked 'wait forever' condition. If 0 is written to W the counter will not restart but will use the new wait time value the next time it is invoked.

Note that the Router is enabled after reset. This is so that a neighbouring chip can peek and poke a chip that fails after reset using NN packets, to diagnose and possibly fix the cause of failure.

In addition, the SLtx field define which links are masked out. A 1 indicates the link is disabled, packets sent to disabled links are dropped and disappeared.

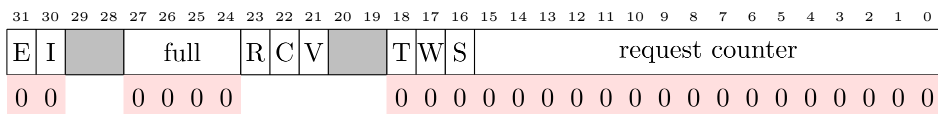
r1: route & output throttle control

This register defines the directions that a broadcast NN packet is sent through, enables the out-of-order issue buffer, stores the 2-bit SDRAM segment address and the 8-bit throttling cycles to control the output rate of the Router ports.

Bits	31:25	24:19	18	17:16	15:8	7:0
Field	NN broadcast		E	SD		Throttling cycles
Reset	0 1 1 1 1 1 1		0	0 0		0 0 0 0 0 0 0 0

The 8-bit throttling value defines minimum number of the router clock cycles between NoC packets sent to the same QPE. A packet may be delayed to restrict traffic entering the NoC. '0' indicates that packets may be issued on consecutive cycles, '1' that there is a single 'idle' cycle etc. In addition, the 'NN broadcast' bits[31:25] define which links an NN broadcast packet is sent through. A 1 indicates an active link, and bit[25] is for link 0, bit[26] link 1, etc.

r2: packet reinsertion control



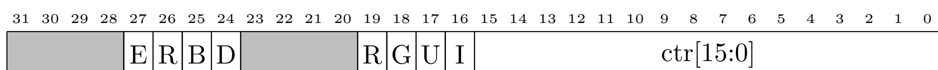
The function of these fields are described in the table below:

Name	bits	R/W	Function
E	31	R/W	enable packet drop to buffer mechanism
I	30	R/W	enable dropped packet re-insertion mechanism
full	27:24	R/W	sets dropped packet buffer full level
R	23	W	reset reinsert packets diagnostic counter
C	22	W	reset words water-level diagnostic counter
V	21	W	reset SpiNNaker packets water-level diagnostic counter
T	18	R/W	enable reinsert packets diagnostic counter
W	17	R/W	enable words water-level diagnostic counter
S	16	R/W	enable SpiNNaker packets water-level diagnostic counter
request counter	15:0	R/W	reinsertion request spacing

The ‘full’ point indicates that the buffer is regarded as *becoming* full; it is set in multiples of 128 words.

r3: router interrupt status

All Router interrupt request sources are visible in this register.



The functions of these fields are described in the table below:

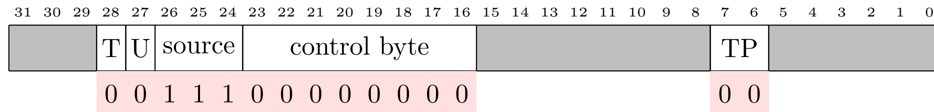
Name	bits	R/W	Function
E: error int	27	R	error interrupt sent
R: drop register int	26	R	drop register interrupt sent
B: drop buffer int	25	R	drop buffer interrupt sent
D: diagnostic int	24	R	diagnostic interrupt sent
R: error int	19	R	error interrupt active
G: drop register int	18	R	drop register interrupt active
U: drop buffer int	17	R	drop buffer interrupt active
I: diagnostic int	16	R	diagnostic interrupt active
ctr[15:0]	15:0	R	diagnostic counter interrupt active

The Router can generate four types of interrupt request packet that are sent to the monitor processor (as specified in r0) through the on-chip network: error interrupt, drop packet register interrupt, drop packet buffer interrupt and diagnostic counter event interrupt. These correspond to E, R, B, D respectively. The drop packet buffer interrupt is generated from the sub-interrupt events ‘full’ and ‘not empty’. These sub interrupts can be enabled in r0. The diagnostic interrupt is generated from the sub-interrupt events (ctr[15:0]).

The next interrupt packet will not be sent before the interrupt status register (r50) is written. The interrupt active status are cleared by reading their respective router interrupt status registers: r10, r38, r45 and reseting the diagnostic counters.

r4: error packet header

A packet which contains an error is moved to r4-10. Once a packet has been moved (indicated by bit[31] of r10 being set) any further error packet is ignored, except that it can update the sticky bits in r10 (and errors of the types specified in r0 are counted in r10).

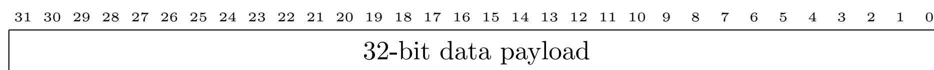


The functions of these fields are described in the table below:

Name	bits	R/W	Function
T: TP error	28	R	packet time phase error
U: unroutable	27	R	unroutable packet error
source	26:24	R	Rx source field of error packet
control byte	23:16	R	control byte of error packet
TP: time phase	7:6	R	time phase when packet received

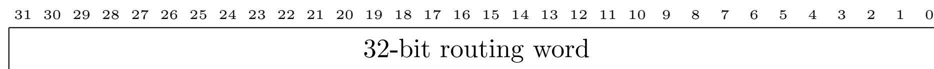
r5 - 8: error packet data payload

If the packet has a 32-bit payload this will be moved into r5 at 0x14:



A 64-bit payload will be moved into 0x14 and 0x18 {r5, r6}; a 128-bit into 0x14-0x20 {r5, r6, r7, r8}.

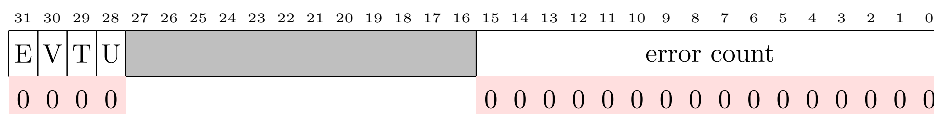
r9: error packet routing word



r10: error interrupt status

There is a single interrupt (packet) generated from this register. Reading this register acknowledges the interrupt and cause all fields of this register to reset, and then the interrupt can be generated again and sent. This regsiter is used for interrupt mechanism. Interrupt information polling can be done by reading r3.

This register counts error packets, including time phase and unroutable packet errors as enabled by r0[5:4]. The Monitor Processor resets r10[31:28] and the error count by reading its contents.

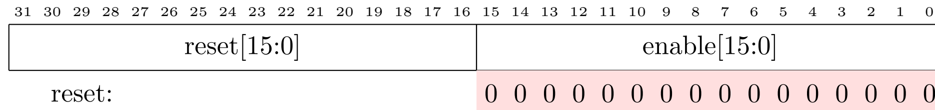


The functions of these fields are described in the table below:

Name	bits	R/W	Function
E: error	31	R	error packet detected
V: overflow	30	R	more than one error packet detected
T: TP error	29	R	packet time phase error (sticky)
U: unroutable error	28	R	unroutable packet error (sticky)
S: TP enable	17	R/W	enable count of packet time phase errors
N: UN enable	16	R/W	enable count of unroutable packet errors
error count	15:0	R	16-bit saturating error count

r11: diagnostic counter enable/reset

This register provides a single control point for the 16 diagnostic counters, enabling them to count events over a precisely controlled time period.



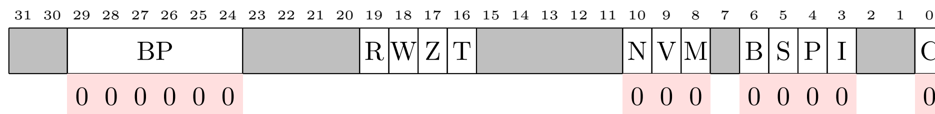
The functions of these fields are described in the table below:

Name	bits	R/W	Function
reset[31:16]	31:16	W	write a 1 to reset diagnostic counter15. . . 0
enable[15:0]	15:0	R/W	enable diagnostic counter 15. . . 0

Writing a 0 to reset[15:0] has no effect. Writing a 1 clears the respective counter.

r12: profiling counters control

This register contains the control bits of the profilling counters, the router cycle counter (r13), busy cycle counter (r14), the zero-wait packet counter (r15) and the iterating dropped packet counter (r16).

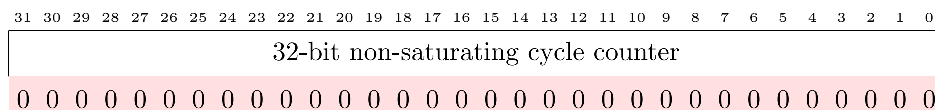


The functions of these fields are described in the table below:

Name	bits	R/W	Function
BP	28:24	R/W	monitor the backpressure at the 6 input ports
R	19	R/W	reset the iterating dropped packet counter (r16)
W	18	R/W	reset the zero-wait packet counter (r15)
Z	17	R/W	reset the busy cycle counter (r14)
T	16	R/W	reset router cycle counter (r13)
N	10	R/W	monitor the zero-wait status at NN output
V	9	R/W	monitor the zero-wait status at C2C output
M	8	R/W	monitor the zero-wait status at MC output
B	6	R/W	monitor the backpressure at the NN input
S	5	R/W	monitor the backpressure at the C2C input
P	4	R/W	monitor the backpressure at the MC input
I	3	R/W	enable the iterating dropped packet counter
C	0	R/W	enable the router cycle counter

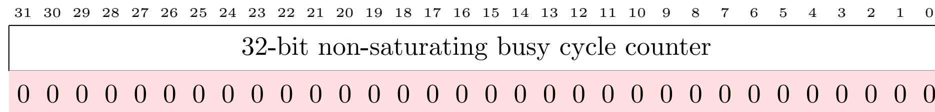
Writing a 0 to R, W Z or T has no effect. Writing a 1 clears the respective counter.

r13: cycle count



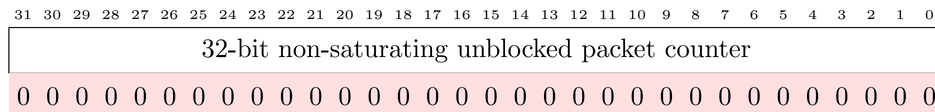
r13, when enabled by r12, simply counts the number of Router clock cycles.

r14: busy cycle count



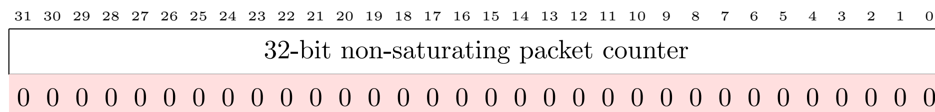
r14, controlled by r12, counts the number of Router wait cycles – cycles when backpressure occurs at the 6 Router input ports, the MC, C2C, NN routing engine inputs.

r15: zero-wait packet counter



r15 can be configured by r12 to count the number of packets which pass through undelayed at one of the MC, C2C, NN routing engine outputs.

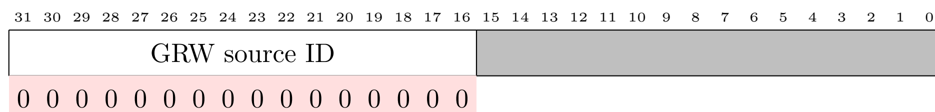
r16: iterating dropped packet counter



r16, when enabled by r12, counts the number of packets that are dropped more than one time.

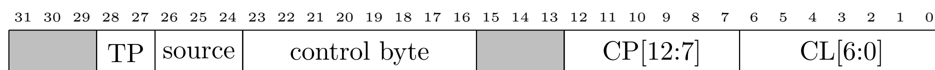
r24: default source ID for the GRW packets

This register stores the 16-bit default source ID that is used to return GRW responses.



r32: drop register header

If the dropped packet buffer is not enabled, a dropped packet will be copied to r32-38. Once a packet has been dropped (indicated by bit[31] of r38 being set) any further packet that is dropped is ignored, except that it can update the sticky bits in r38 (and can be counted by a diagnostic counter).

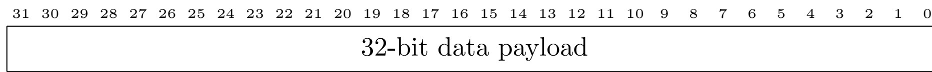


The function of these fields are described in the table below:

Name	bits	R/W	Function
TP	28:27	R	Router time phase when packet is dropped
source	26:24	R	Rx source field of dropped packet
control byte	23:16	R	control byte of dopped packet
CP[12:7]	12:7	R	congested ports
CL[6:0]	6:0	R	congested links

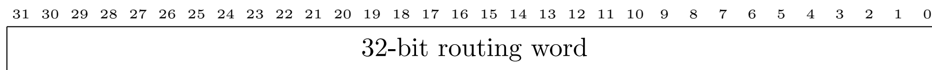
r33-36: drop register payload

If the packet has a 32-bit payload this will be copied into r33 at 0x84:



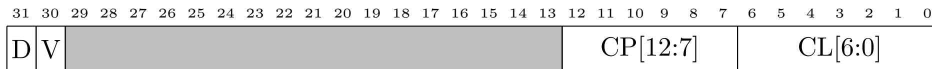
A 64-bit payload will be copied into 0x84 and 0x88; a 128-bit into 0x84-90.

r37: drop register routing word



r38: drop register interrupt status

There is a single interrupt (packet) generated from this register. Reading this register acknowledges the interrupt and cause the V field to reset, and then the interrupt can be generated again and sent. This regsiter is used for interrupt mechanism. Interrupt information polling can be done by reading r3.

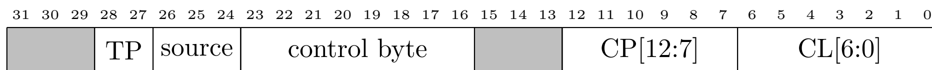


The functions of these fields are described in the table below:

Name	bits	R/W	Function
D:dropped	31	R	packet dropped
V:overflow	30	R	more than one packet dropped
CP[12:7]	12:7	R	congested ports(sticky)
CL[6:0]	6:0	R	congested links(sticky)

r39: software offload header

Reading this register causes the oldest packet in the H/W dropped packet buffer to be offloaded into r39-r44. If the dropped packet buffer is empty, a bus error will be returned.

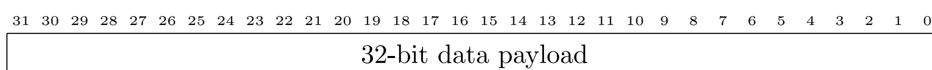


The function of these fields are described in the table below:

Name	bits	R/W	Function
TP	28:27	R	Router time phase when packet is dropped
source	26:24	R	Rx source field of dropped packet
control byte	23:16	R	control byte of dopped packet
CP[12:7]	12:7	R	congested ports
CL[6:0]	6:0	R	congested links

r40-43: software offload payload

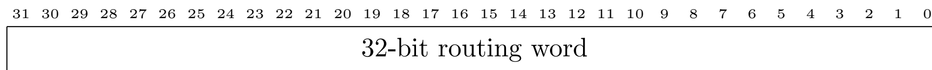
If the offloaded packet has a 32-bit payload this will be copied into r40 at 0xA0:



A 64-bit offloaded payload will be copied into 0xA0 and 0xA4; a 128-bit into 0xA0-AC.

r44: software offload routing word

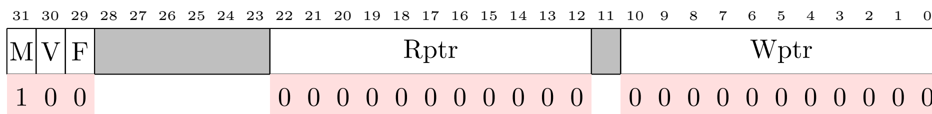
This field will be updated when the a valid dropped packet is offloaded by reading r39.



r45 (rDP): router dropped packet buffer interrupt status

There is a single interrupt (packet) generated from this register. Reading this register acknowledges the interrupt and cause the V field to reset, and then the interrupt can be generated again and sent. This register is used for interrupt mechanism. Interrupt information polling can be done by reading r3.

When dropping to buffer is enabled, dropped packets are automatically stored in a 2048 x 32-bit word RAM (DP[2047:0]) that operates as a circular buffer, from which they are subsequently automatically re-inserted into the Router by enabling the request counter in r2 when the Router is lightly loaded. rDP holds the read and write pointers for the circular buffer, and sundry status bits.



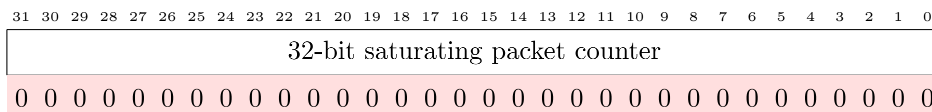
The functions of these fields are described in the table below:

Name	bits	R/W	Function
M	31	R	dropped packet buffer empty
V	30	R	dropped packet buffer overflow (sticky)
F	29	R	dropped packet buffer (approaching) full
Rptr	22:12	R	dropped packet buffer read pointer
D	11	R/W	enable packet drop to buffer mechanism
Wptr	10:0	R	dropped packet buffer write pointer

When the buffer is not able to accept the full dropped packet, the overflow bit V will be set.

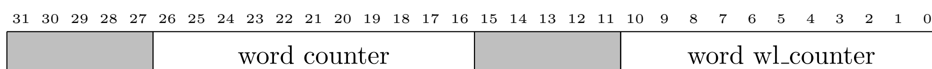
In general the dropped packet buffer will operate autonomously, though if there is any danger of it overflowing software should be invoked to off-load some of the contents to memory elsewhere. This can be done by reading r39 to cause a packet offloaded from the drop buffer. The read-out packets should then be re-inserted at an appropriate time by software.

r46: total reinsert packets counter



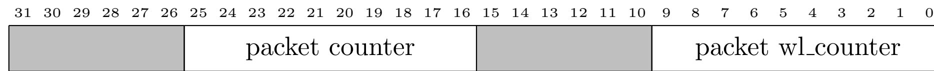
r46, when enabled by r2, simply counts the number of packets reinserted by hardware.

r47: wd water-level counter



The word counter counts the number of words in the dropped packet buffer. The corresponding water-level counter records the maximum number of words ever dropped in the buffer. When it is enabled in r2, if wd ctr is bigger than wd wlctr, wd wlctr will be updated.

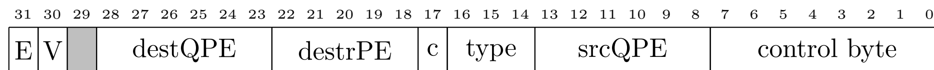
r48: spkt water-level counter



The SpiNNaker packet counter counts the number of SpiNNaker packets in the dropped packet buffer. The corresponding water-level counter records the maximum number of packets ever dropped in the buffer. When it is enabled in r2, if spkt ctr is bigger than spkt wlctr, spkt wlctr will be updated.

r49: unrecognised packets logging

This register logs the packet header information of the unrecognised packet received at the SpiNNaker ports for diagnostic purpose. For example, control and exception packet, protocol message and undefined SpiNNaker packet will be discarded and logged in this register. Once the first unrecognised packet header is recorded, the further packet is ignored, except that the overflow bit is updated. Reading this register causes it resets.

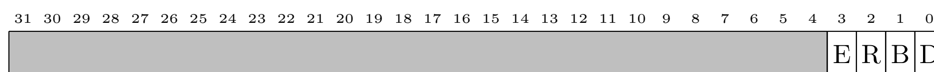


The functions of these fields are described in the table below:

Name	bits	R/W	Function
E:error	31	R	unrecognised packet detected
V:overflow	30	R	more than one unrecognised packet detected
destQPE	28:23	R	destination QPE field in the packet header
destrPE	22:18	R	destination PE in the packet header
c	17	R	C/D NoC selection bit in the packet header
type	16:14	R	NoC packet type in the packet header
srcQPE	13:8	R	source QPE filed in the packet header
control byte	7:0	R	lower 8-bit control field in the packet header

r50: interrupt acknowledge

There are four interrupts (packets) generated from the error packet, dropped packet register, dropped packet buffer and diagnostic counters. Writing 1 to E, R, B, or D acknowledges the error packet, dropped packet register, dropped packet buffer and diagnostic counters interrupts respectively. After the interrupt is acknolwedged, the interrupt (packet) can be generated again and sent. This register is used for the interrupt messaging mechanism. Interrupt information polling can be done by reading r3.

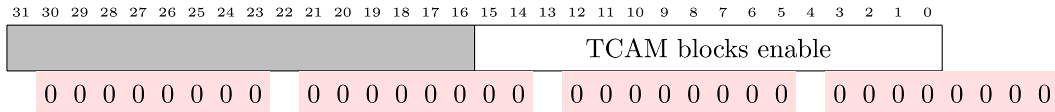


The functions of these fields are described in the table below:

Name	bits	R/W	Function
E	3	W	acknowledge the error packet interrupt
R	2	W	acknowledge the dropped packet register interrupt
B	1	W	acknowledge the dropped packet buffer interrupt
D	0	W	acknowledge the diagnostic counters interrupt

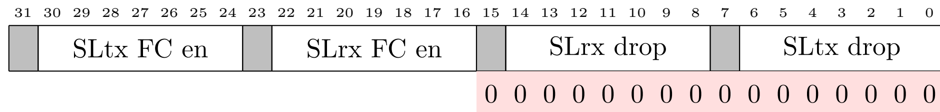
r51: TCAM blocks control

The TCAM consists of 16 sub blocks. Each block has 1K entries and can be enabled in this register.



r54: link control

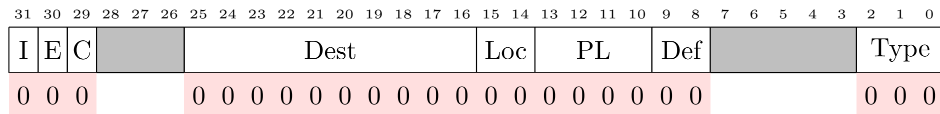
This registers contains 7-bit link transmitter flow control control enable, 7-bit link receiver flow control enable, 7-bit link receiver drop, and 7-bit link transmitter drop.



rF[N]: diagnostic filter control

The Router has 16 diagnostic counters (N = 0..F) each of which counts packets passing through the Router filtered on packet characteristics defined here. A packet is counted if it has characteristics that match with a '1' in each of the 6 fields. Setting all bits [24:10, 7:0] to '1' will count all packets.

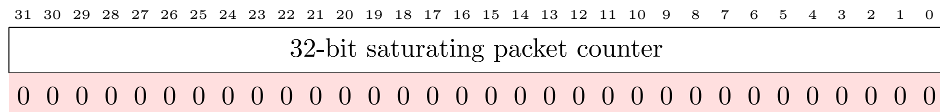
A diagnostic counter may (optionally) generate an interrupt on each count. The C bit[29] is a sticky bit set when a counter event occurs and is cleared whenever this register is read.



The functions of these fields are described in the table below:

Name	bits	R/W	Function
I	31	R	counter interrupt active: I = E AND C
E	30	R/W	enable interrupt on counter event
C	29	R	counter event has occurred (sticky)
Dest	25:16	R/W	packet dest (Tx link[6:0], MP, local ¬MP, dump)
Loc	15:14	R/W	local [x1]/non-local[1x] packet source
PL	13:10	R/W	packets without [1xxx] payload, or with 32-bit [x1xx], 64-bit [xx1x] or 128-bit [xxx1] payload
Def	9:8	R/W	default [x1]/non-default [1x] routed packets
Type	3:0	R/W	packet type: NN[1xx], C2C[x1x], MC[xx1]

rC[N]: diagnostic counters

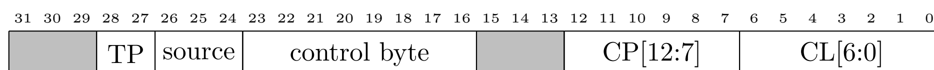


Each of these counters can be used to count selected types of packets under the control of the corresponding rFN. The counter can have any value written to it, and will increment from that value when respective events occur. If an event occurs as the counter is being written it will not be counted. To avoid missing an event it is better to avoid writing to the counter; instead, read it at the start of a time period and subtract this value from the value read at the end of the period to get a count of the number of events during the period.

DP[2047:0]: dropped packet buffer

When dropping to buffer and hardware packet reinsertion are enabled (r2), dropped packets will automatically be stored in this RAM buffer, and re-inserted into the Router. The dropped packet buffer is writeable. When dropping to buffer and hardware packet reinsertion are disabled, it can be used for general purpose. However, writing the buffer is not recommended when it is used for hard packet dropping and automatic reinsertion.

Each dropped packet is written as $2 + N$ words, where N is the number of words in the payload. The first word is the header word:



The functions of these fields are described in the table below:

Name	bits	Function
TP	30:29	time phase when packet dropped
source	26:24	Rx source field of dropped packet
control byte	23:16	dropped packet control byte
CP[12:7]	12:7	on-chip network port error caused packet drop
CL[6:0]	6:0	Tx link transmit error caused packet drop

If the packet had a payload, this is written into ascending address(es). The last word written to the buffer is the packet routing word. This order is designed to facilitate the read/write multiple ARM instructions

Sufficient information is stored to allow the packet to be re-inserted into the front of the Router pipeline and routed exactly as before.

18.14 Fault-tolerance

The Communications Router has some internal fault-tolerance capacity, in particular it is possible to map out a failed multicast router entry. This is a useful mechanism as the multicast router dominates the silicon area of the Communications Router.

There is also capacity to cope with external failures. In order to tolerate a chip failure several expedients can be employed on a local basis:

1. C2C packets can be routed around the obstruction;
2. MC packets with a router entry can be redirected appropriately.

In most cases, default MC packets cannot sensibly be trapped by adding table entries due to their (almost) infinite variety. To allow rerouting, these packets can be dropped to the Monitor Processor on a link-by-link basis using the diversion register. In principle they can then be routed around the obstruction as C2C payloads before being resurrected at the opposite side.

Should the Monitor Processor become overwhelmed, it is also possible to use the diversion register to eliminate these packets in the Router; this prevents them blocking the Router pipeline whilst waiting for a timeout and thus delaying viable traffic.

Fault insertion

1. TO BE DONE

Fault detection

1. packet time-phase errors.
2. packet unroutable errors (e.g. a locally-sourced multicast packet which doesn't match any entry in the multicast router).

Fault isolation

1. a multicast router entry can be disabled if it fails - see initialisation guidance above.

Reconfiguration

1. since all multicast router entries are identical the function of any entry can be relocated to a spare entry.
2. if a router becomes full a global reallocation of resources can move functionality to a different router.

19 SDRAM interface

The SDRAM interface connects the NoC to an off-chip LPDDR3 SDRAM device.

19.1 Features

1. TO BE DONE

19.2 DMA

The SDRAM interface contains a DMA controller for DMA data transfers from and to core's local SRAM.

19.2.1 DMA Overview

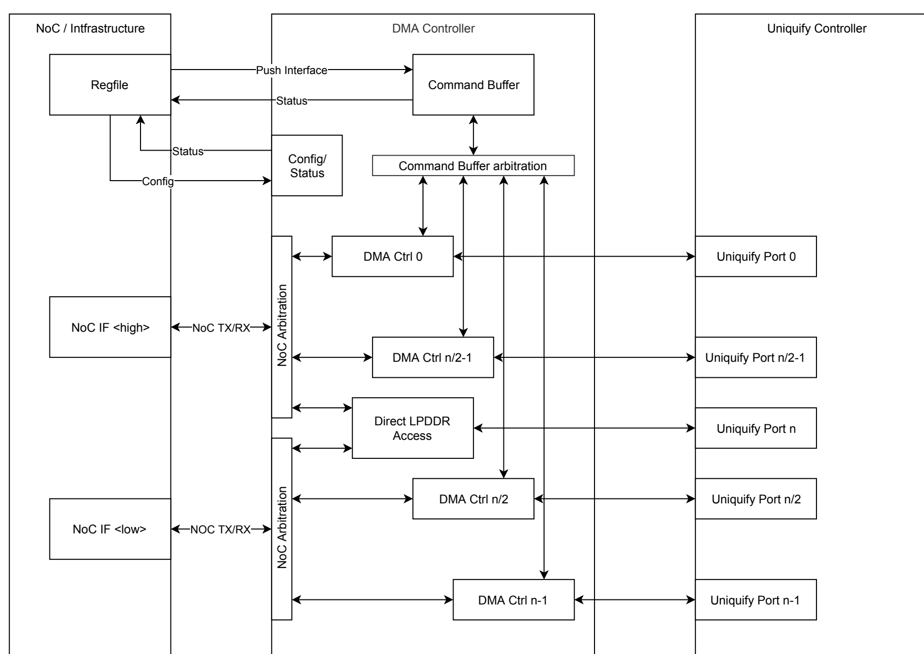


Figure 23: SDRAM DMA module overview

The DMA controller is configured via register-file (TO BE DONE). DMA requests are pushed into a command buffer via register-file (TO BE DONE: more details). A request consits of

1. request type (SDRAM read or write)
2. SDRAM start address (NoC memory-mapped)
3. PE SRAM start address (NoC memory-mapped)
4. transfer size in bytes (must be a multiple of 64).

TO BE DONE: register format, comms controller support for simplifying requests.

Commands of the buffer are distributed to n (TO BE DONE: actual number) DMA controllers that preferably handle the DMA requests on a close NoC Y coordinate. Additionally it is possible to directly access SDRAM via a NoC read or write request.

Packets from the NoC are visible to all DMA controllers on the same NoC port at the same time. The DMA controller must be able to recognize whether it is the destination of the packet based on its currently served request (PE, expected address). Only that controller is allowed to read the packet from the NoC.

NoC packets from the DMA controller need to be arbitrated round robin towards the NoC.

Commands from the command buffer need to be distributed to the individual DMA controllers. If only one controller is ready to serve a command that controller is selected. If multiptle controllers are ready on different NoC interfaces the one with the closer NoC y coordinate is selected. If multiple controllers with same NoC y coordinate are available the one with lowest ID is selected.

19.2.2 DMA SDRAM read

DMA SDRAM read sequence (see figure 24):

1. DMA read command data is pushed to command buffer via a register-file write.
2. Once the command is available at the output of the command buffer and a DMA controller is ready to serve it, it is popped from the buffer and served by that DMA controller.
3. The DMA controller issues one (or if necessary multiple) LPDDR read commands to its assigned interface at the SDRAM controller.
4. Read data from the SDRAM controller is worwarded as NoC writes to the PEs memory. The write requests are buffered writes but every n-th (configurable n) write and the last write are unbuffered to prevent NoC flooding. A certain (configurable) number of buffered writes following an intermediate unbuffered writes is allowed before the write response packet arrives to allow finetuning of data flow.
5. After the last write to the PEs memory, which must be an unbuffered write, and its write response's arrival at the DMA controller, an interrupt at the PE having issued the DMA request is triggered via a NoC write to the corresponding QPE's register file.

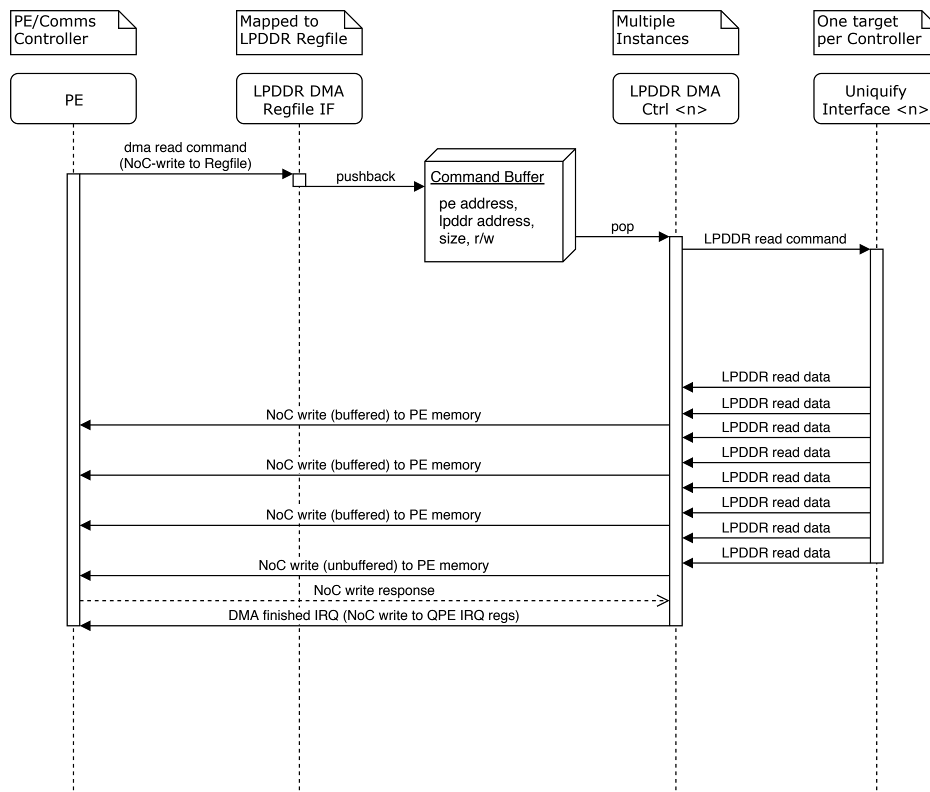


Figure 24: DMA SDRAM read transfer

19.2.3 DMA SDRAM write

DMA SDRAM write sequence (see figure 25):

1. DMA write command data is pushed to command buffer via a register-file write.
2. Once the command is available at the output of the command buffer and a DMA controller is ready to serve it, it is popped from the buffer and served by that DMA controller.
3. The DMA controller issues one (or if necessary multiple) LPDDR write commands to its assigned interface at the SDRAM controller.
4. The DMA controller issues one or multiple NoC read requests to the PE's SRAM. only a configurable number of outstanding read requests is allowed before arrival of read data from a previous request to prevent NoC flooding.

5. Arriving read response data from the PE is forwarded as SDRAM write data to the SDRAM controller.
6. When the SDRAM controller signals successful processing of the last write data sequence, an interrupt at the PE having issued the DMA request is triggered via a NoC write to the corresponding QPE's register file.

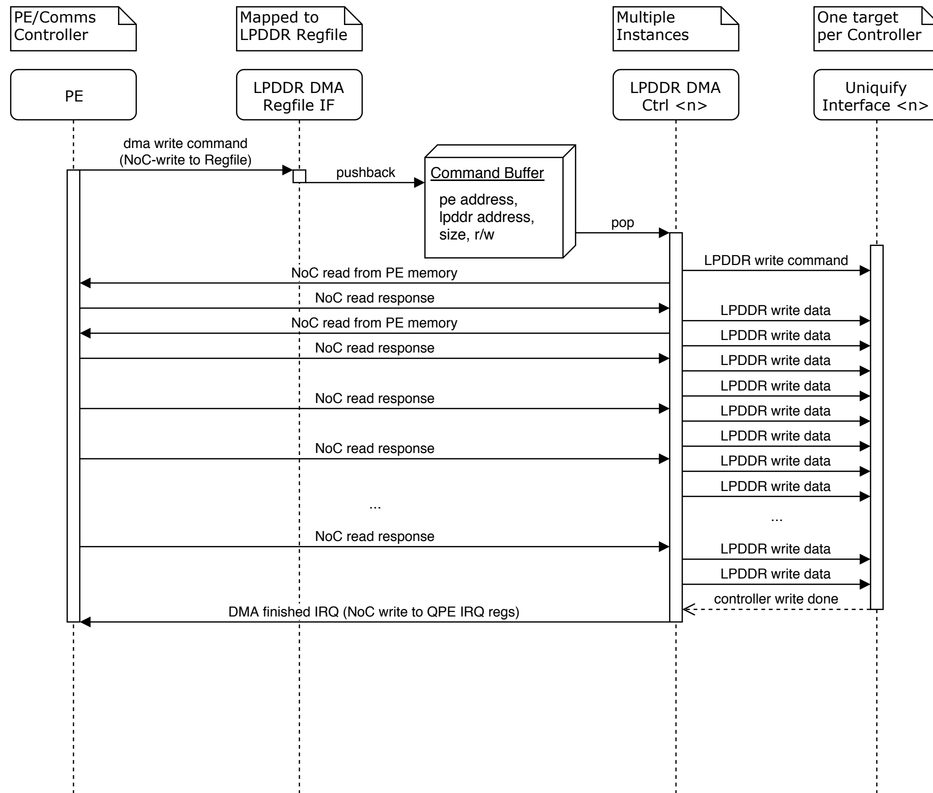


Figure 25: DMA SDRAM write transfer

19.2.4 DMA SDRAM configuration

1. number of outstanding read requests to PE memory for DMA SDRAM write transfer
2. number of buffered write requests before an unbuffered write request is issued for DMA SDRAM read transfer
3. number of buffered write requests before an outstanding write response packet is received
4. TO BE DONE

19.3 Register summary

TO BE DONE

19.4 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

20 Inter-chip transmit and receive interfaces

Inter-chip communication is implemented with high-speed serial communications that has been optimized for power with sparse traffic. The interfaces powered down in the absence of traffic, but have rapid recovery when traffic is presented.

20.1 Features

There are two modes for inter-chip communication available:

1. Chip-to-Chip (C2C) Link and
2. LVDS AURORA Link.

The C2C link is build for a near distance chip to chip communication and the LVDS AURORA link is build for longer distance board to board communication.

20.1.1 Key features for the Chip-to-Chip Link

1. gross data rate: 12.0 GBit/s
2. net data rate: 7.0 GBit/s
3. six data transmission lanes in both directions
   (a) 2 GBit/s per lane
   (b) double data rate
   (c) transmission line fundamental wave at 1GHz
4. power down when idle

20.1.2 Key features for the LVDS AURORA link

1. gross data rate: 2.00 GBit/s
2. net data rate: 933.33 MBit/s
3. one transmission lane in both directions
   (a) 2 GBit/s per lane
   (b) double data rate
   (c) transmission line works at 1GHz
   (d) aurora protocol with 8b10b line code

20.2 Configuration

20.2.1 register selection

There are five individual registers for the link configuration inside:

1. NoC registers
2. SpikeSedes registers
3. C2C registers
4. AURORA registers
5. LVDS register

An APB multiplexer is available to access the registers. The selection signal is generated as follows from the three MSBs of the signal apb_addr[11:0]:

1. 3'b000: NoC registers
2. 3'b001: SpikeSerdes registers
3. 3'b010: C2C LINK registers
4. 3'b011: AURORA LINK registers
5. 3'b100: LVDS SERDES register

20.3 Chip-to-Chip Link (C2C Link)

The Chip-to-Chip Link enables the inter-chip communication between nearest neighbor chips from the architectural point of view. Each chip includes six C2C links, each capable of transmitting data at a rate of 2Gbps per lane in one direction, resulting in a total data rate of 12Gbps in each direction. Figure 26 shows the basic data flow structure of two C2C link interfaces, which are located on two different chips.

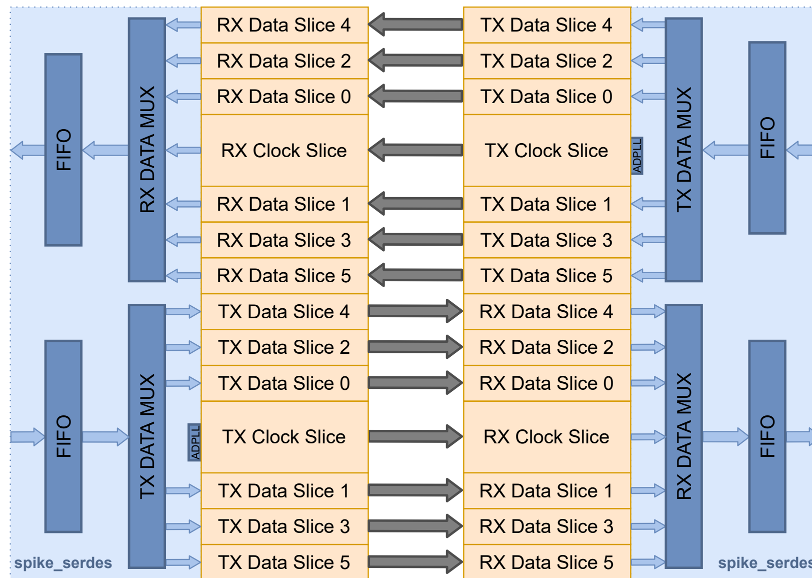


Figure 26: C2C Link: data flow between two interfaces

20.3.1 C2C Link Transceiver

Figure 27 depicts a simplified block level schematic of the C2C Link transceiver (orange colored boxes in figure 26). The C2C Link acts as a source synchronous link, which means that in addition to the data, the clock is also being sent from the TX to the RX. This allows a fast on and off switching, because, once recovered, the sampling phase does not shift, since there is no frequency offset between TX and RX. The data is being transmitted single-ended, whereas the clock is being transmitted differential. This results in a reduced sensitivity of the transmitted clock signal to interferences from other signals, which is necessary to sample the data signal correctly at the receiver. Due to the short transmission channel distances on the PCB, the data can be transmitted single-ended. This further decreases the power consumption of the transceivers.

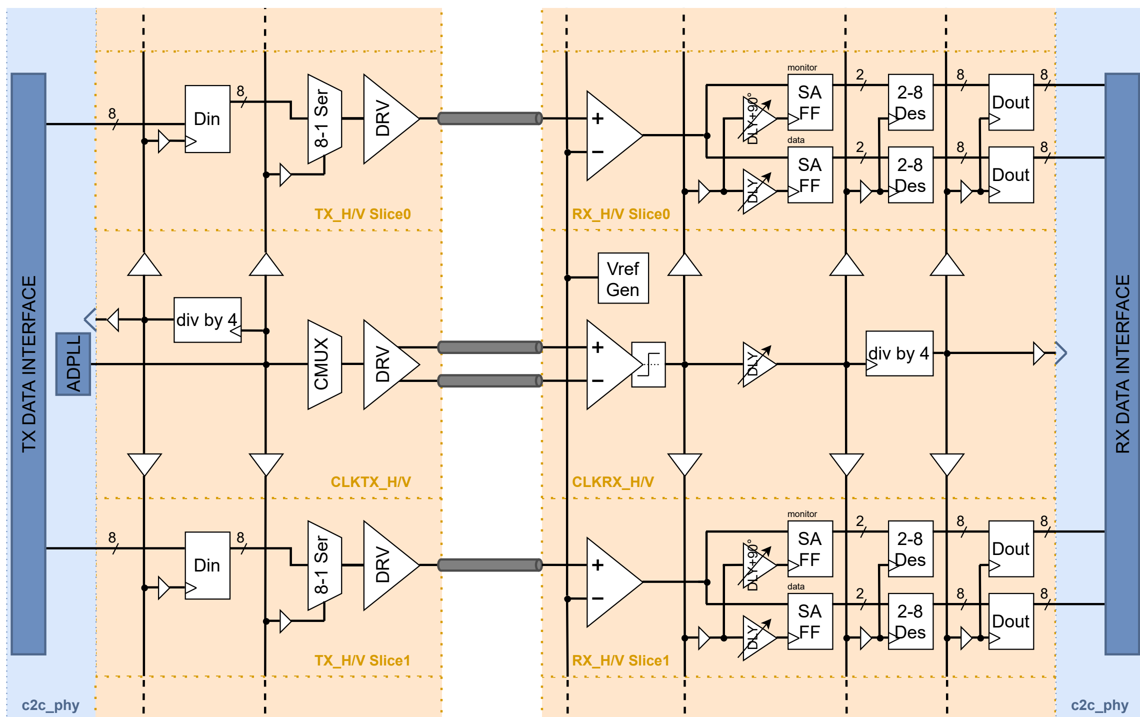


Figure 27: C2C Link Transceiver: Clock lane plus two data lanes

21 Periphery

The periphery module contains various IO interfaces for low-speed communication. It further contains a periphery processor for general purpose control tasks for chip boot up and operation. The periphery module is not body biased and clocked solely with the reference clock signal. It is operational directly after power up and reset de-assertion. The chip bootup is orchestrated by the periphery module.

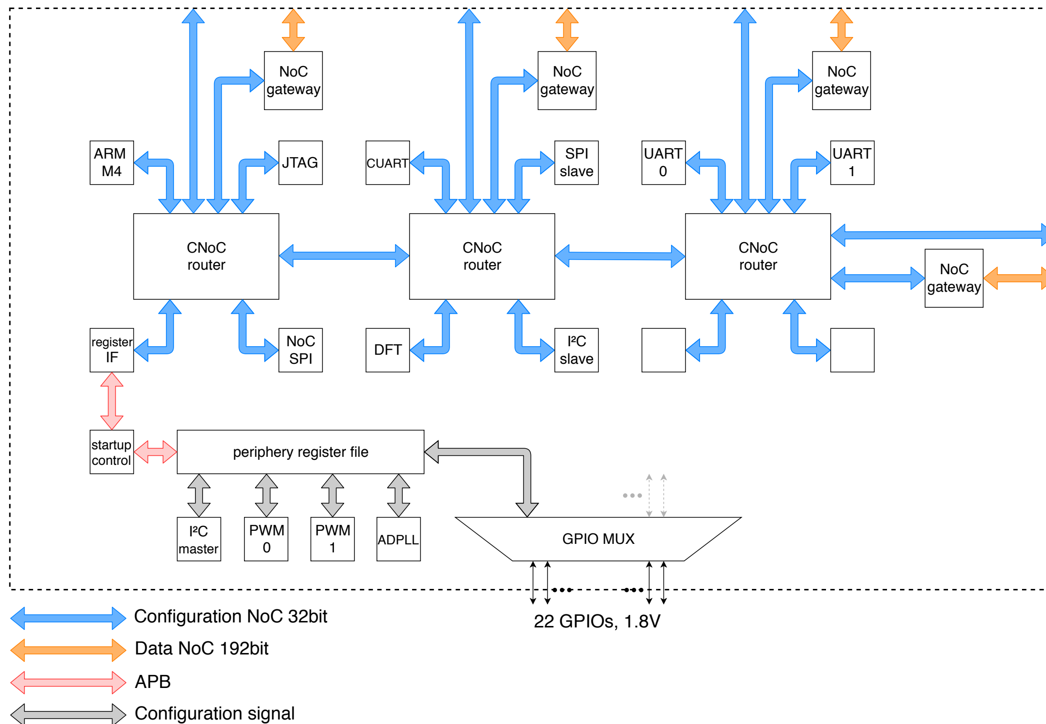


Figure 28: Periphery components

21.1 Start-up Control

SpiNNaker2 chip can be started in four different modes:

1. JTAG-only
2. I2C slave
3. SPI flash boot + JTAG
4. SPI slave + JTAG

The start-up mode is by bootstrap pins GPIO0 and GPIO1 after the reset release. After reset the start-up controller configures GPIO pads and enables periphery components. A list of used GPIO pads can be found in section 22.4. The remaining GPIO pads are inputs by default, but can be reconfigured later on.

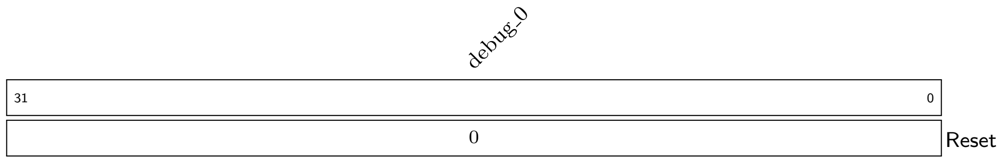
21.2 Register File Interface

The CNoC register file interface converts NoC packets e.g. write requests to the APB bus. All attached APB registers are accessible by the global memory address offset 0xf10000000. Only local register addresses (without offset) are denoted in the following.

debug register

Addresses 0x00000000 to 0x0000000c are debugging registers. These can be used to test access via external interfaces e.g. JTAG, I2C or SPI.

Register 21.1: DEBUG_0 (0x00000000)

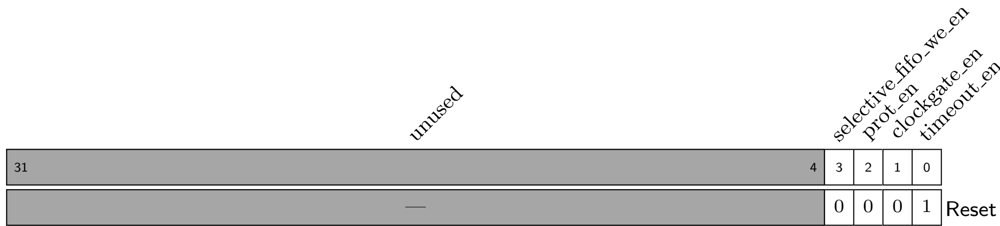


debug_0 (RW) debug register #0

register file control

The NoC-APB bridge and NoC gateway can be configured via register 0x00000010 .

Register 21.2: REGFILE_CTRL (0x00000010)



selective_fifo_we_en	(RW🔒)	enable selective write enable for NoC gateway
prot_en	(RW🔒)	enable protected mode
clockgate_en	(RW🔒)	enable clock gating for register file
timeout_en	(RW🔒)	enable timeout for the case regfile is not reacting

21.3 GPIO MUX

The GPIO multiplexer allows to map internal interfaces e.g. UART to a GPIO pad. Figure 29 shows the basic structure. Each individual pad has up to 5 different functions.

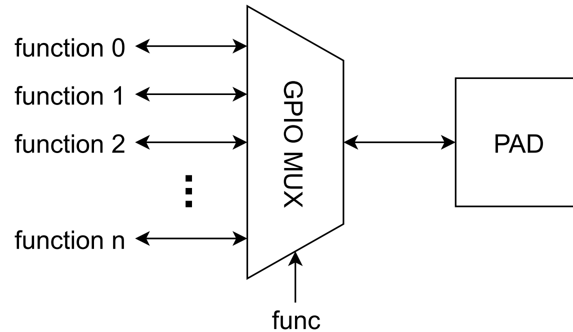


Figure 29: GPIO function multiplexer

pad	func0	func1	func2	func3	func4	note
0	RF	I2CS IRQ	SPIM INTR	ARM IRQ0	DBG0	bootstrap
1	RF	SPIS0 IRQ	SPIS1 INTR	ARM IRQ1	DBG1	bootstrap
2	RF	SPIM NSS	SPIS1 NSS	SDIO CMD	UART1 TX	
3	RF	SPIM SCLK	SPIS1 SCLK	SDIO CLK	UART1 RX	
4	RF	SPIM IO0	SPIS1 IO0	SDIO D0	UART1 RTR	
5	RF	SPIM IO1	SPIS1 IO1	SDIO D1	UART1 CTS	
6	RF	SPIM IO2	SPIS1 IO2	SDIO D2	I2CM SCL	
7	RF	SPIM IO3	SPIS1 IO3	SDIO D3	I2CM SDA	
8	RF	I2CS SCL[1]	CUART TX	PWM0		
9	RF	I2CS SDA[1]	CUART RX	PWM1		
10	RF	SPIF NSS[2]	SPIS0 NSS[3]	PWM2_0		
11	RF	SPIF SCLK[2]	SPIS0 SCLK[3]	PWM2_1		
12	RF	SPIF IO0[2]	SPIS0 IO0[3]	PWM2_2		
13	RF	SPIF IO1[2]	SPIS0 IO1[3]	PWM2_3		
14	RF	SPIF IO2	SPIS0 IO2	PWM2_4		
15	RF	SPIF IO3	SPIS0 IO3	JARM RST		
16	RF	JTAG TMS[0,2,3]	UART0 CTS	JARM TMS		
17	RF	JTAG TCK[0,2,3]	UART0 RTR	JARM TCK		
18	RF	JTAG TDO[0,2,3]	UART0 TX	JARM TDO		
19	RF	JTAG TDI[0,2,3]	UART0 RX	JARM TDI		

[0] GPIO[1:0] = 0; pad active in JTAG mode
[1] GPIO[1:0] = 1; pad active in I2C slave mode
[2] GPIO[1:0] = 2; pad active in SPI flash boot mode
[3] GPIO[1:0] = 3; pad active in SPI slave mode

GPIO pad configuration

The periphery core contains 20 GPIO pins with selectable functionality. Starting from address 0x000000c0 to 0x0000010c the GPIO function can be selected by changing the func register field.

Register 21.3: GPIO_0_CFG (0x000000c0)

	unused	pd	pu	ste	odn	odp	co	sr	ds	bias	unused	func
Bits	31–28	27	26	25–24	23	22	21	20	19–18	17–16	15–3	2–0
Reset	—	0	0	1	0	0	0	1	3	0	—	0

pd	(RW)	pull up
pu	(RW)	pull down
ste	(RW)	Schmitt trigger threshold
odn	(RW)	open drain for nMOS
odp	(RW)	open drain for pMOS
co	(RW)	current output
sr	(RW)	slew rate control
ds	(RW)	drive strength: 0=4mA, 1=8mA, 2=12mA, 3=16mA
bias	(RW)	bias source select
func	(RW)	gpio mux function

GPIO register file control

By selecting GPIO function 0, each GPIO can be controlled via register file. To read from GPIO pins the input enable must be set via register 0x00000114 . The GPIO read value can be obtained from register 0x0000011c . The GPIO output value can be set by enabling the desired output pin at register 0x00000110 and setting up the value of register 0x0000011c .

Register 21.4: GPIO_IE (0x00000114)

	unused	gpio_19	gpio_18	gpio_17	gpio_16	gpio_15	gpio_14	gpio_13	gpio_12	gpio_11	gpio_10	gpio_9	gpio_8	gpio_7	gpio_6	gpio_5	gpio_4	gpio_3	gpio_2	gpio_1	gpio_0
Bits	31–20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	—	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

Register 21.5: GPIO_DI (0x0000011c)

	unused	gpio_19	gpio_18	gpio_17	gpio_16	gpio_15	gpio_14	gpio_13	gpio_12	gpio_11	gpio_10	gpio_9	gpio_8	gpio_7	gpio_6	gpio_5	gpio_4	gpio_3	gpio_2	gpio_1	gpio_0
Bits	31–20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Register 21.6: GPIO_OE (0x00000110)

	unused	gpio_19	gpio_18	gpio_17	gpio_16	gpio_15	gpio_14	gpio_13	gpio_12	gpio_11	gpio_10	gpio_9	gpio_8	gpio_7	gpio_6	gpio_5	gpio_4	gpio_3	gpio_2	gpio_1	gpio_0
Bits	31–20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

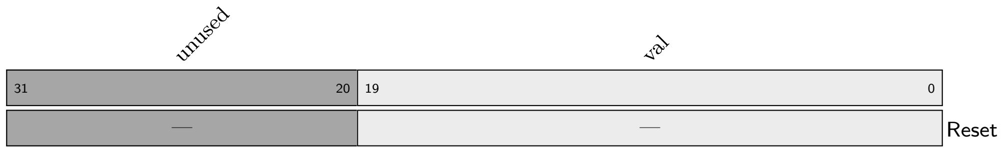
Register 21.7: GPIO_DO (0x00000118)

	unused	gpio_19	gpio_18	gpio_17	gpio_16	gpio_15	gpio_14	gpio_13	gpio_12	gpio_11	gpio_10	gpio_9	gpio_8	gpio_7	gpio_6	gpio_5	gpio_4	gpio_3	gpio_2	gpio_1	gpio_0
Bits	31–20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reset	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

GPIO reset value

During reset state all GPIO pins are configured as inputs. After the reset is released the GPIO state is captured and stored in register 0x00000304 .

Register 21.8: STARTUP_RESET_RD (0x00000304)



val (R) reset GPIO read value

21.4 Clock Configuration

CLK pad configuration

Register 21.9: CLKRST_PADS (0x000000a0)

Bits	31–10	9–8	7–2	1–0
Field	unused	resetn_ste	unused	clk_ste
Reset	—	1	—	1

resetn_ste	(RW🔒)	schmitt trigger control of reset_n pad
clk_ste	(RW🔒)	schmitt trigger control of clk_ref pad

Register 21.10: CLK_CONF (0x00000200)

Bits	31–12	11–9	8–2	1	0
Field	unused	periphery_divider	systick_divider	core_systick_en	core_ref_clk_en
Reset	—	1	50	0	0

periphery_divider	(RW🔒)	periphery core clock divider value
systick_divider	(RW🔒)	systick clock divider value n (div by 2n)
core_systick_en	(RW🔒)	enable systick clock for core area
core_ref_clk_en	(RW🔒)	enable reference clock for core area

Register 21.11: CLK_CONF_SPI (0x00000204)

Bits	31–5	4–3	2	1	0
Field	unused	spis_pll_div_n_2x	spis_pll_div_bypass	spis_clk_en	spis_sel
Reset	—	1	1	1	0

spis_pll_div_n_2x	(RW)	SPI slave pll clock divider n value (div by 2n)
spis_pll_div_bypass	(RW)	SPI slave pll clock divider bypass
spis_clk_en	(RW)	enable SPI slave clock
spis_sel	(RW)	SPI slave clock sel, 0: reg_clk 1: pll_core_clk

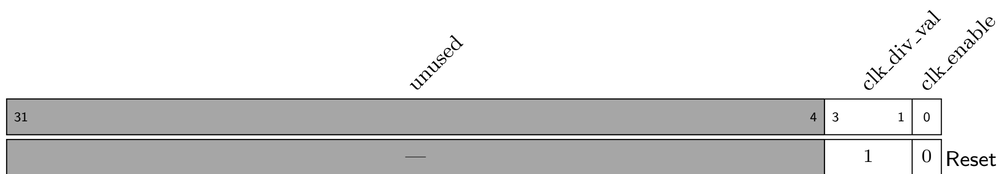
21.5 Periphery Arm Cortex-M4

The periphery ARM Cortex M4 core is an PE (see section 5) with reduced functionality. For area (and power) reduction the accelerator cores and floating point unit were removed.

1. ARM Cortex M4 core
2. 128kByte ECC SRAM
3. comms unit
4. max. 100MHz clock frequency
5. internal and external interrupt sources

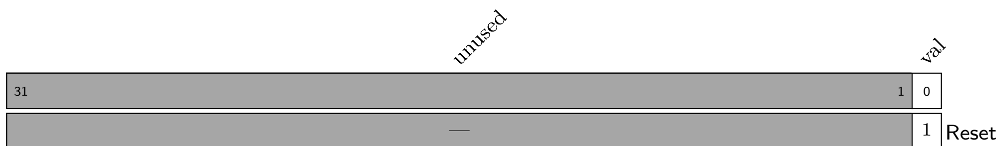
CLK and reset configuration

Register 21.12: ARM_CLK_CONF (0x000001bc)



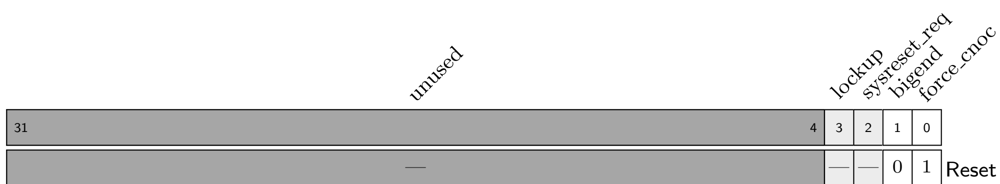
clk_div_val	(RW🔒)	core clock divider value
clk_enable	(RW🔒)	periphery ARM M4 core

Register 21.13: ARM_SRESET (0x000001c0)



val	(RW🔒)	periphery ARM M4 secondary reset

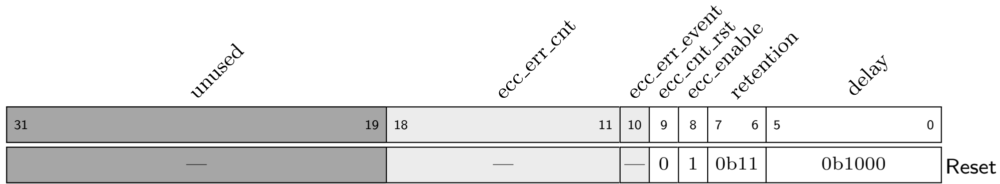
Register 21.14: ARM_CONF (0x000001c4)



lockup	(R🔒)	debug lockup
sysreset_req	(R🔒)	debug sysreset_req
bigend	(RW🔒)	debug bigend
force_cnoc	(RW🔒)	force CNoC routing

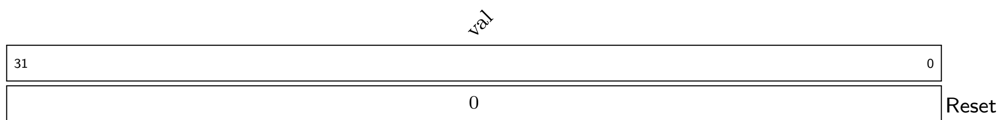
memory configuration

Register 21.15: ARM_SRAM (0x000001c8)



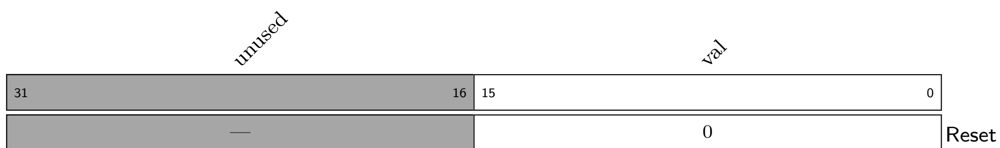
ecc_err_cnt	(R🔒)	ECC error counter value
ecc_err_event	(R🔒)	SRAM ECC error occured
ecc_cnt_rst	(RW🔒)	reset counter value
ecc_enable	(RW🔒)	enable SRAM ECC
retention	(RW🔒)	SRAM retention setting
delay	(RW🔒)	SRAM delay setting

Register 21.16: MBIST_CMD_SEQ0 (0x000001cc)



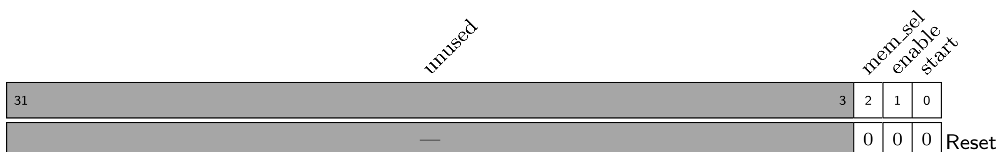
val (RW🔒) bist command sequence lower 32bit

Register 21.17: MBIST_CMD_SEQ1 (0x000001d0)



val (RW🔒) bist command sequence upper 16bit

Register 21.18: MBIST_CTRL (0x000001d4)



mem_sel	(RW🔒)	memory wrap selection
enable	(RW🔒)	enable mbist
start	(RW🔒)	start mbist

Register 21.19: MBIST_STATUS (0x000001d8)

pc (R🔒) captured program counter
res (R🔒) bist test result 0: PASS ; 1: FAIL
busy (R🔒) mbist test running

Register 21.20: MBIST_ADDR (0x000001dc)

val (R🔒) failed addr

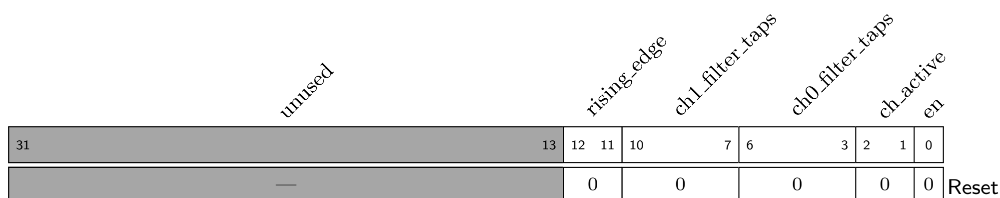
GPIO pads

The periphery ARM Cortex M4 core has 4 dedicated ARM JTAG pads and 2 interrupt input pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
AJTAG TMS	input	ARM JTAG select
AJTAG TCK	input	ARM JTAG clock input
AJTAG TDO	output	ARM JTAG serial data out
AJTAG TDI	input	ARM JTAG serial data in
ARM IRQ0	input	external ARM interrupt 0
ARM IRQ1	input	external ARM interrupt 1

external interrupt configuration

Register 21.21: ARM_GPI_IRQ (0x000001e0)



rising_edge (RW🔒) trigger IRQ on rising edge, falling edge otherwise
ch1_filter_taps (RW🔒) channel 1: number of active filter taps
ch0_filter_taps (RW🔒) channel 0: number of active filter taps
ch_active (RW🔒) enable channel
en (RW🔒) enable GPIO IRQ core

Register 21.22: ARM_PERIPHERY_IRQ (0x000001e4)

unused	noc_spi	uart1	uart0	rf	
31 19	18 13	12 8	7 3	2 0	
—	0	0	0	0	Reset

noc_spi (RW🔒) NoC SPI IRQ mask
uart1 (RW🔒) UART1 IRQ mask
uart0 (RW🔒) UART0 IRQ mask
rf (RW🔒) register triggered interrupt

Register 21.23: ARM_FT_IRQ_MASK_01 (0x000001e8)

unused	irq_mask1	irq_mask0	
31 20	19 10	9 0	
—	0	0	Reset

irq_mask1 (RW🔒) feed through IRQ mask bit 1
irq_mask0 (RW🔒) feed through IRQ mask bit 0

Register 21.24: ARM_FT_IRQ_MASK_23 (0x000001ec)

unused	irq_mask3	irq_mask2	
31 20	19 10	9 0	
—	0	0	Reset

irq_mask3 (RW🔒) feed through IRQ mask bit 3
irq_mask2 (RW🔒) feed through IRQ mask bit 2

21.6 JTAG

The JTAG IEEE 1149.1 system on the SpiNNaker chip provides access to the ARM Cortex M4F processors for software debug purposes and scan access to the SpiNNaker pins for PCB testing purposes.

GPIO pads

The JTAG interface uses 4 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
JTAG TMS	input	JTAG select
JTAG TCK	input	JTAG clock input
JTAG TDO	output	JTAG serial data out
JTAG TDI	input	JTAG serial data in

TO BE DONE

21.7 SPI

The Serial Peripheral Interface (SPI) is a common serial off-chip bus to interface e.g. sensors or memory ICs. In general the SPI bus is set up in a single master - (multi)slave configuration where the master always initiates the communication. SPI uses four signal wires in minimal configuration:

1. SCLK - serial clock, driven by master
2. NSS - low active slave select (also refereed as S# or CS#), driven by master
3. MOSI - master out - slave in data signal, driven by master
4. MISO - master in - slave out data signal, driven by slave

The picture (30) shows parts of a SPI communication:

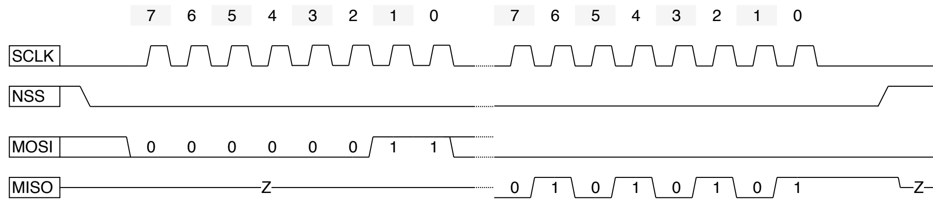


Figure 30: single SPI communication

To start a communication sequence, the SPI pulls down NSS. After this the SPI clock is toggling and the bus master transfers data on MOSI line. In the given example the clock line is default zero and data lines are assigned on a falling clock edge while the receiving bus node is sampling on rising clock edge (SPI mode 0). Typically SPI communication is byte oriented and the MSB is transferred first. When commutation is done, the master stops clocking and assigns slave select high. The example shows a classical SPI communication: the master drives MOSI and the slave drives MISO. This configuration allows a full-duplex communication, however data transfer usually just takes place on one signal line. In applications a higher data transfer rate is required (e.g. memories) the SPI protocol was slightly modified: in dual SPI operation MISO and MOSI lines can be bidirectional and are driven by the same bus instance. This allows a nearly doubled data transfer rate while maintaining four signal wires. For even higher data rates the quad SPI (4 data lines) is supported by many modern SPI memory devices.

21.7.1 NoC SPI

The SpiNNaker2 chip includes one QSPI slave, master and a dedicated SPI flash master controller (see figure 31).

NoC SPI enable

Register 21.25: NOC_SPI_EN (0x00000120)

	unused	spi_mux_enable	spis_enable	spim_enable	su_ctrl_en	noc_spi_ovr	master_mode	en
Bit	31–7	6	5	4	3	2	1	0
Reset	—	0	0	0	0	0	0	0

spi_mux_enable	(RW🔒)	enable SPI mux
spis_enable	(RW🔒)	spi slave enable
spim_enable	(RW🔒)	spi master enable
su_ctrl_en	(RW🔒)	startup core enable
noc_spi_ovr	(RW🔒)	override master and enable signal -> use rf val
master_mode	(RW🔒)	core in master mode (flash startup)
en	(RW🔒)	start startup routine

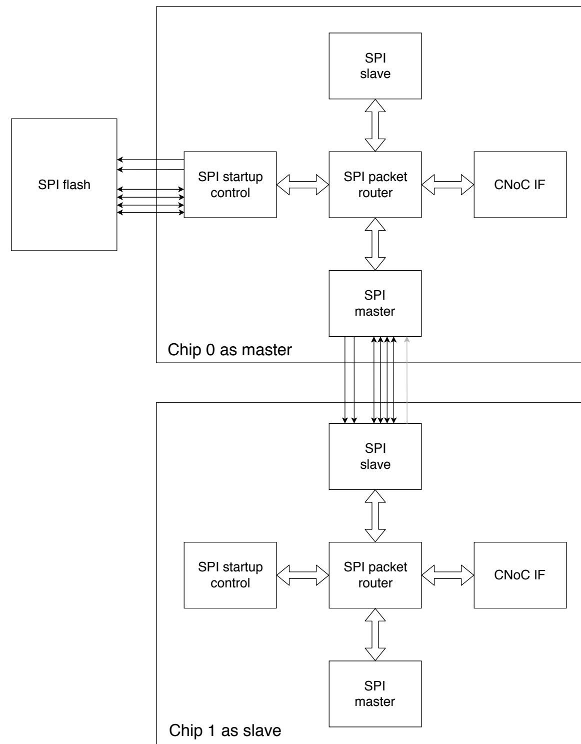


Figure 31: SPI multi chip setup

chip ID selection

Register 21.26: SPI_CHIP_ID (0x00000158)

	unused	chip_id_set	ctrl_src_id
Bits	31 – 7	6	5 – 0
Reset	—	0	0

chip_id_set	(RW🔒)	chip ID is set set, if not packets will go to NoCIF
ctrl_src_id	(RW🔒)	source chip ID of packet

multiplexer configuration

Register 21.27: SPI_MUX_CONF (0x00000168)

	unused	link_qpe_dest_mc	mux_master_mode
Bits	31 – 5	4 – 1	0
Reset	—	0	0

link_qpe_dest_mc (RW🔒) multicast enable
mux_master_mode (RW🔒) 0-slave mode, 1-master mode

Register 21.28: SPI_NOC_CONF (0x00000174)

31 … 6	5 … 2	1 … 0	
—	0	0	Reset

tx_mc_mask (RW🔒) multicast enable
tx_global_segment (RW🔒) mem segment

clock configuration

Register 21.29: SPI_CLK_DIV (0x0000013c)

31 … 15	14 … 10	9 … 5	4 … 0	
—	10	1	10	Reset

mst (RW🔒) spi master, clk div value
su_b (RW🔒) spi flash setup b, clk div value
su_a (RW🔒) spi flash setup a, clk div value

21.7.2 SPI slave

SpiNNaker2 includes a SPI slave interface with following features:

1. single SPI command; single, dual and quad SPI data operation
2. dual/quad SPI operation for command + data
3. supported frequency: up to 20 MHz
4. SPI mode 1: clock default low, data assigned at rising edge, sampled at falling edge
5. external interrupt, signalling available data
6. continuous write operation with auto address increment

GPIO pads

The SPI slave can use up-to 7 GPIO pads. To map the desired function to GPIO pad please see section 22.4. IO2 and IO3 are only necessary for quad SPI mode. These pads can be used for other purposes if not needed. The INTR pad is signalling that a NoC packet is available at RX buffer side. It is not mandatory to use this pad. The status command can be used instead.

name	direction	function
SPIS0 SCLK	input	SPI slave clock input
SPIS0 NSS	input	SPI slave low-active slave select
SPIS0 IO0	inout	SPI slave data 0 (MOSI)
SPIS0 IO1	inout	SPI slave data 1 (MISO)
SPIS0 IO2	inout	SPI slave data 2
SPIS0 IO3	inout	SPI slave data 3
SPIS0 INTR	output	SPI slave interrupt(packet available), low active, open drain

SPI slave commands

The SPI slave supports various commands which are described below. In general a slave sequence consists of a command byte, address and data bytes. To describe the used number of signal lines for the sequence a CAD style notation is used. E.g. a 1-4-4 sequence means single SPI for command, quad SPI for address and data. To receive a status from SPI slave a dedicated command can be used consisting of SPI command byte and one status byte. The status byte contains following information: status[7:4]=0x5, status[3]=0x0, status[2]=rx_rising_egde, status[1]=tx_falling_egde;

Command	Code	Sequence	Function
CMD_S_STATUS	0xC0	1-0-1	get 8bit slave status; SSPI
CMD_D_STATUS	0xC1	1-0-2	get 8bit slave status; DSPI
CMD_D_STATUS	0xC2	1-0-4	get 8bit slave status; QSPI
CMD_S_READ_REQ	0xA4	1-1-0	read request to address; SSPI
CMD_D_READ_REQ	0xA5	1-2-0	read request to address; DSPI
CMD_Q_READ_REQ	0xA6	1-0-4	read request to address; QSPI
CMD_S_READ	0x04	1-0-1	read 32bit data from buffer; SSPI
CMD_D_READ	0x05	1-0-2	read 32bit data from buffer; DSPI
CMD_Q_READ	0x06	1-0-4	read 32bit data from buffer; QSPI
CMD_S_WRITE	0xA8	1-1-1	write 32bit data to address; SSPI
CMD_D_WRITE	0xA9	1-2-2	write 32bit data to address; DSPI
CMD_Q_WRITE	0xAA	1-4-4	write 32bit data to address; DSPI

The following commands are used for packet based multi chip communication. To support a chain of SpiNNaker2 chips two packet specific bytes were added consisting of the 8bit SPI command, 4bit packet type (P_type), 6bit target, 6bit source chipID, 32bit address and 32bit data.

SPI_CMD	P_type	trgtID	srcID	addr	data

Command	Code	Sequence	Function
CMD_P_READ_REQ_S	0xB8	1-1-1	read request to address; SSPI
CMD_P_READ_REQ_D	0xB9	1-2-2	read request to address; DSPI
CMD_P_READ_REQ_Q	0xBA	1-4-4	read request to address; QSPI
CMD_P_READ_S	0xC4	1-1-1	read 10 bytes from buffer; SSPI
CMD_P_READ_D	0xC5	1-2-2	read 10 bytes from buffer; DSPI
CMD_P_READ_Q	0xC6	1-4-4	read 10 bytes from buffer; QSPI
CMD_P_WRITE_S	0xC8	1-1-1	write 10 bytes to buffer; SSPI
CMD_P_WRITE_D	0xC9	1-2-2	write 10 bytes to buffer; DSPI
CMD_P_WRITE_Q	0xCA	1-1-4	write 10 bytes to buffer; QSPI

Register 21.30: SPI_SLV_CONF (0x0000016c)

Field	unused	crc_restart	crc_enable	dummy_en	tx_out_reg_stages	tx_edge_dly	clk_falling_inv	clk_rising_inv	io_smpl_delay	clk_falling_dly	clk_rising_dly	rx_edge_rising	tx_edge_falling	mode
Bits	31–19	18	17	16	15–14	13–12	11	10	9–8	7–6	5–4	3	2	1–0
Reset	—	0	1	0	0	0	0	0	0	0	0	0	0	0

crc_restart	(RW🔒)	restart CRC
crc_enable	(RW🔒)	enable CRC byte checking
dummy_en	(RW🔒)	enable dummy byte between RX-TX transition
tx_out_reg_stages	(RW🔒)	registered TX data
tx_edge_dly	(RW🔒)	delay tx clock edge
clk_falling_inv	(RW🔒)	use rising edge
clk_rising_inv	(RW🔒)	use falling edge
io_smpl_delay	(RW🔒)	use delayed io samples
clk_falling_dly	(RW🔒)	>0: use delayed rising edge
clk_rising_dly	(RW🔒)	>0: use delayed falling edge
rx_edge_rising	(RW🔒)	read data on rising edge
tx_edge_falling	(RW🔒)	write data on falling edge
mode	(RW🔒)	SPI mode 0:SSPI; 1:DSPI; 2:QPSI

21.7.3 SPI master

The SPI master interface uses ut to 6 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
SPIM SCLK	output	SPI master clock
SPIM NSS	output	SPI master low-active slave select
SPIM IO0	inout	SPI master data 0 (MOSI)
SPIM IO1	inout	SPI master data 1 (MISO)
SPIM IO2	inout	SPI master data 2, add pullup resistor
SPIM IO3	inout	SPI master data 3, add pullup resistor
SPIM INTR	input	SPI master interrupt

configuration

Register 21.31: SPI_MST_CONF (0x0000015c)

Bits	31–11	10	9	8	7	6	5	4	3–2	1–0
Field	unused	rf_spim_tx_fifo_full	rf_fe_rx_fifo_empty	rf_fe_cycle_stop	rf_fe_ready	rf_fe_en	rf_bypass	continous_mode	cmd_mode	std_cmd
Reset	—	—	—	—	—	0	0	0	0	0

rf_spim_tx_fifo_full	(R🔒)	frontend tx buffer full
rf_fe_rx_fifo_empty	(R🔒)	frontend rx buffer empty
rf_fe_cycle_stop	(R🔒)	processing done
rf_fe_ready	(R🔒)	ready to process
rf_fe_en	(RW🔒)	enable frontend : processing FIFO entries
rf_bypass	(RW🔒)	control frontend via regfile
continous_mode	(RW🔒)	enable continous mode
cmd_mode	(RW🔒)	0-SSPI, 1-DSPI, 2-QSPI
std_cmd	(RW🔒)	0-SSPI, 1-DSPI, 2-QSPI

Register 21.32: MSPI_FE_CONFIG (0x00000138)

Bits	31–21	20	19–12	11–4	3	2	1	0
Field	unused	tx_fifo_flush	o_val_default	oe_default	loopback	cpol	tx_rising	lsb_first
Reset	—	0	0	0	0	0	1	0

tx_fifo_flush	(RW🔒)	clear TX fifo
o_val_default	(RW🔒)	default output value
oe_default	(RW🔒)	default output enable
loopback	(RW🔒)	loopback enable for test
cpol	(RW🔒)	clock polarity
tx_rising	(RW🔒)	rising edge is transmitting edge
lsb_first	(RW🔒)	send LSB first

receive buffer

Register 21.33: SPI_MST_RX_BUFFER (0x00000160)

Bits	31–8	7–0
Field	unused	data
Reset	—	—

data (R🔒) frontend rx buffer data

transmit buffer

Register 21.34: SPI_MST_TX_BUFFER (0x00000164)

Bits	31–18	17–0
Field	unused	data
Reset	—	0

data (RW🔒) frontend tx buffer data

21.7.4 SPI flash start-up controller

For autonomous low-level boot-up SpiNNaker2 contains a QSPI flash controller. The SPI start-up controller fetches commands e.g. NoC writes from the flash device in order to start chip components such as PLL. To start SpiNNaker2 from SPI flash the bootstrap GPIO configuration must be set properly during reset. When SPI start-up controller starts it first performs a wait operation to give the flash device time to start-up properly. A typical start-up time for flash devices is 300us until its ready for read operation. After the wait sequence is done, the controller will start reading the SPI flash from address 0x0000. It will use the 0x03 flash read command which is supported by most modern SPI flash devices. To allow a robust start-up the SPI clock frequency is set to 5MHz and standard SPI mode is used. Depending on the PCB design and used flash device the bus frequency can be increased to 50MHz and quad SPI mode can be enabled.

GPIO pads

The SPI flash controller can use up-to 6 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
SPIF SCLK	output	SPI flash controller clock
SPIF NSS	output	SPI flash controller low-active slave select, push/pull, add external pullup resistor
SPIF IO0	inout	SPI flash controller data 0 (MOSI)
SPIF IO1	inout	SPI flash controller data 1 (MISO)
SPIF IO2	inout	SPI flash controller data 2, add pullup resistor
SPIF IO3	inout	SPI flash controller data 3, add pullup resistor

SPI flash controller commands

The SPI start-up controller expects a data sequence organized in 9byte parts. A single command consists of 1byte command, 4byte address and 4byte data:

CMD	addr	data

The following commands are supported:

Command	Code	address	data	function
CMD_W_NOC	0xAA	32bit addr	32bit wdata	memory mapped NoC write
CMD_W_NOC_WAIT_SHORT	0x50	32bit addr	32bit wdata	write with short delay
CMD_W_NOC_WAIT_LONG	0x5F	32bit addr	32bit wdata	write with long delay
CMD_MEM_CPY	0xEE	32bit addr	number of bytes	copy n following bytes to address
CMD_REG_WR	0x11	32bit addr	32bit wdata	write internal register
CMD_R_NOC_WAIT_SHORT	0xB0	32bit addr	32bit expected data	read request, wait short until expected value is read
CMD_R_NOC_WAIT_LONG	0xBF	32bit addr	32bit expected data	read request, wait long until expected value is read
CMD_RF_BYPASS	0x99	unused	24bit wait count	start FIFO stored sequence
CMD_RF_BYPASS_SETUP	0x98	unused	24bit wait count	start FIFO stored sequence, toggle setop A¡-¿B
CMD_FLASH_ADDR	0x22	unused	32bit address	change SPI flash address
CMD_FLASH_SECTOR	0x33	offset	19bit address	change SPI flash address
CMD_WAIT	0xDD	unused	24bit wait count	wait
CMD_CRC	0xCC	unused	32bit CRC checksum	CRC checksum packet
CMD_STOP	0xFF	unused	unused	stop controller

configuration

Register 21.35: STARTUP_CONF (0x00000124)

unused	su_fe_tx_fifo_full	su_fe_rx_fifo_empty	su_fe_core_stop	su_ctrl_su_wait_done	su_ctrl_crc_error	su_ctrl_cmd_error	su_ctrl_busy	su_ctrl_done	su_ctrl_bypass_fe_en	su_ctrl_bypass_mode	su_ctrl_crc_en	su_ctrl_setup_sel	su_ctrl_dma_wr	su_ctrl_mode	
31 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
—	—	—	—	—	—	—	—	—	0	0	1	0	0	1	Reset

su_fe_tx_fifo_full	(R🔒)	frontend tx buffer full
su_fe_rx_fifo_empty	(R🔒)	frontend rx buffer empty
su_fe_core_stop	(R🔒)	frontend cycle done
su_ctrl_su_wait_done	(R🔒)	done waiting for powering up of SPI flash
su_ctrl_crc_error	(R🔒)	crc error detected
su_ctrl_cmd_error	(R🔒)	command error detected
su_ctrl_busy	(R🔒)	cycle ongoing
su_ctrl_done	(R🔒)	cycle done
su_ctrl_bypass_fe_en	(RW🔒)	enable FE
su_ctrl_bypass_mode	(RW🔒)	bypass RF FIFO to FE
su_ctrl_crc_en	(RW🔒)	enable crc checking during runtime
su_ctrl_setup_sel	(RW🔒)	use setup A(0), or B(1)
su_ctrl_dma_wr	(RW🔒)	DMA copy mode: write to flash
su_ctrl_mode	(RW🔒)	startup mode(1), copy mode (0)

Register 21.36: STARTUP_FLASH_ADDR (0x00000128)

val	
31 0	
—	Reset

val (R🔒) startup control flash address

Register 21.37: FLASH_FE_CONFIG (0x00000134)

unused	tx_fifo_flush	o_val_default	oe_default	loopback	cpol	tx_rising	lsb_first	
31 21	20	19 12	11 4	3	2	1	0	
—	0	0	0	0	0	0	0	Reset

tx_fifo_flush	(RW🔒)	clear TX fifo
o_val_default	(RW🔒)	default output value
oe_default	(RW🔒)	default output enable
loopback	(RW🔒)	loopback enable for test
cpol	(RW🔒)	clock polarity
tx_rising	(RW🔒)	rising edge is transmitting edge
lsb_first	(RW🔒)	send LSB first

setting A

Register 21.38: STARTUP_SPIM_A (0x0000012c)

	unused	data_master_cmd	dummy_num_bytes	dummy_master_cmd	dummy_en	addr_num_bytes	addr_master_cmd	cmd_byte	cmd_master_cmd	cmd_en
Bits	31–28	27–25	24–21	20–18	17	16–15	14–12	11–4	3–1	0
Reset	—	0	0	0	0	3	0	0x3	0	1

data_master_cmd	(RW🔒)	setup a, front-end command for data cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
dummy_num_bytes	(RW🔒)	setup a, number of dummy cycle bytes
dummy_master_cmd	(RW🔒)	setup a, frontend dummy cycle cmd
dummy_en	(RW🔒)	setup a, enable dummy cylce
addr_num_bytes	(RW🔒)	setup a, number of address cylce bytes
addr_master_cmd	(RW🔒)	setup a, front-end command for address cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
cmd_byte	(RW🔒)	setup a, cmd cycle byte for slave, 0x03-SPI, 0x3B DSPI, 0x6B-QSPI
cmd_master_cmd	(RW🔒)	setup a, front-end command for cmd cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
cmd_en	(RW🔒)	setup a, cmd cycle enable

setting B

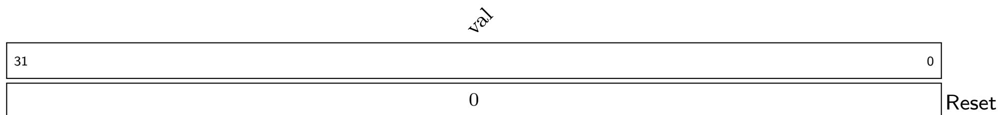
Register 21.39: STARTUP_SPIM_B (0x00000130)

	unused	data_master_cmd	dummy_num_bytes	dummy_master_cmd	dummy_en	addr_num_bytes	addr_master_cmd	cmd_byte	cmd_master_cmd	cmd_en
Bits	31–28	27–25	24–21	20–18	17	16–15	14–12	11–4	3–1	0
Reset	—	0	0	0	0	3	0	0x3	0	1

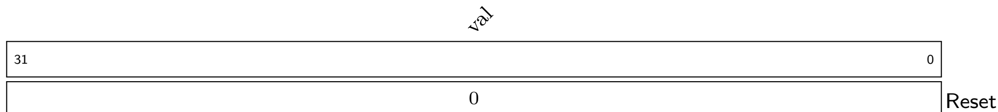
data_master_cmd	(RW🔒)	setup b, front-end command for data cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
dummy_num_bytes	(RW🔒)	setup b, number of dummy cycle bytes
dummy_master_cmd	(RW🔒)	setup b, frontend dummy cycle cmd
dummy_en	(RW🔒)	setup b, enable dummy cylce
addr_num_bytes	(RW🔒)	setup b, number of address cylce bytes
addr_master_cmd	(RW🔒)	setup b, front-end command for address cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
cmd_byte	(RW🔒)	setup b, cmd cycle byte for slave, 0x03-SPI, 0x3B DSPI, 0x6B-QSPI
cmd_master_cmd	(RW🔒)	setup b, front-end command for cmd cylce, 0x0-SPI, 0x1 DSPI, 0x2-QSPI
cmd_en	(RW🔒)	setup b, cmd cycle enable

register read mask

Register 21.40: STARTUP_NOC_MASK (0x0000014c)

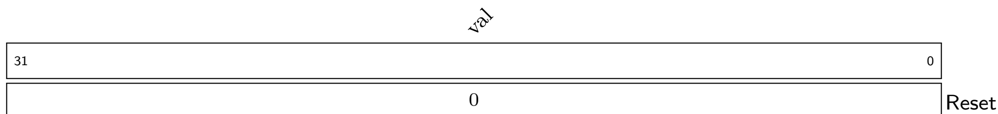


Register 21.41: STARTUP_NOC_MASK (0x0000014c)



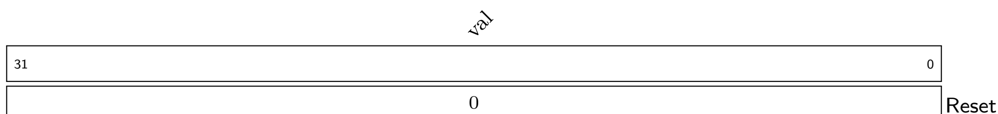
memory copy configuration

Register 21.42: STARTUP_MC_NOC_ADDR (0x00000140)



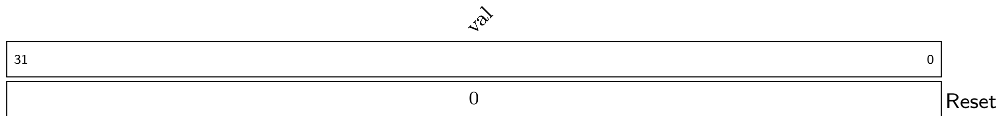
val (RW🔒) mem copy mmap start addr

Register 21.43: STARTUP_MC_SPI_ADDR (0x00000144)



val (RW🔒) mem copy spi start addr

Register 21.44: STARTUP_MC_BYTES (0x00000148)



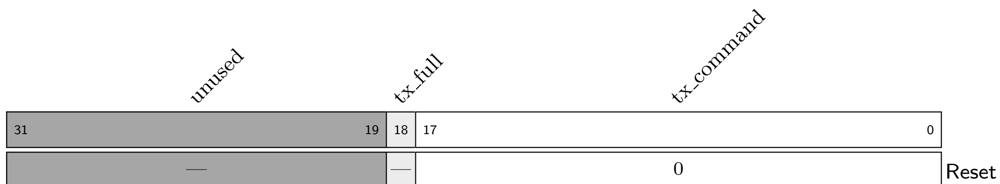
val (RW🔒) mem copy number of bytes to be transfered

SPI master access

Register 21.45: SPI_SU_RX_BUFFER (0x00000150)

rx_av (R🔒) SPI RX buffer available
rx_empty (R🔒) SPI RX buffer empty
rx_byte (R🔒) SPI master RX buffer

Register 21.46: SPI_SU_TX_BUFFER (0x00000154)



tx_full (R🔒) SPI TX buffer full
tx_command (RW🔒) SPI master TX buffer

21.7.5 Spike SPI slave

GPIO pads

The SPI spike slave can use up-to 7 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
SPIS1 SCLK	input	SPI spike slave clock input
SPIS1 NSS	input	SPI spike slave low-active slave select
SPIS1 IO0	inout	SPI spike slave data 0 (MOSI)
SPIS1 IO1	inout	SPI spike slave data 1 (MISO)
SPIS1 IO2	inout	SPI spike slave data 2
SPIS1 IO3	inout	SPI spike slave data 3
SPIS1 INTR	output	SPI spike slave interrupt(packet available), low active, open drain

configuration

Register 21.47: SPIS_CONFIG (0x00000194)

	unused	err_mmap	err_rx_pkt	err_cmd	err_nss	packet_rx_ready	interface_test	cmd_status_en	dummy_en	tx_out_reg_stages	tx_edge_dly	clk_falling_inv	clk_rising_inv	io_smpl_delay	clk_falling_dly	clk_rising_dly	rx_edge_rising	tx_edge_falling	mode	packet_mmap_segment	enable
Bits	31 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
Reset	—	—	—	—	—	—	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

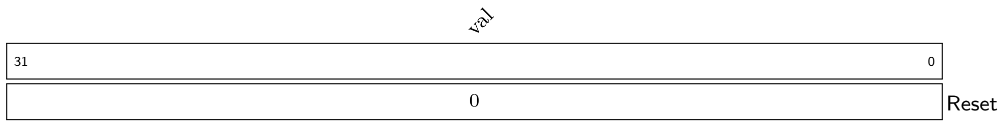
err_mmap	(R)	mmap access error
err_rx_pkt	(R)	no RX packet error
err_cmd	(R)	unknown command error
err_nss	(R)	NSS release error
packet_rx_ready	(R)	NoC packet received, ready to send out via SPI
interface_test	(RW)	test SPI bus by sending CMD_x_WR_SPK_0 / CMD_x_RD_BUF_D32, prevents NoC packet sending
cmd_status_en	(RW)	enable status output in CMD cycle, only in SSPI
dummy_en	(RW)	enable dummy byte between RX-TX transition
tx_out_reg_stages	(RW)	registered TX data
tx_edge_dly	(RW)	delay tx clock edge
clk_falling_inv	(RW)	use rising edge
clk_rising_inv	(RW)	use falling edge
io_smpl_delay	(RW)	use delayed io samples
clk_falling_dly	(RW)	>0: use delayed rising edge
clk_rising_dly	(RW)	>0: use delayed falling edge
rx_edge_rising	(RW)	read data on rising edge
tx_edge_falling	(RW)	write data on falling edge
mode	(RW)	SPI mode 0:SSPI; 1:DSPI; 2:QPSI
packet_mmap_segment	(RW)	memory mapped segment
enable	(RW)	enable SPI slave

Register 21.48: SPIS_SPIKE_CONFIG (0x00000198)

	unused	spike_ptr	control_byte	link	noc_c	noc_pe	noc_y	noc_x
Bits	31 27	26 23	22 15	14 12	11	10 6	5 3	2 0
Reset	—	4	0	0b111	0	8	2	2

spike_ptr	(RW)	packet byte pointer, to skip transfer
control_byte	(RW)	SpiNNaker packet control byte
link	(RW)	link number
noc_c	(RW)	use CNoC
noc_pe	(RW)	NoC PE address of SpiNNaker router
noc_y	(RW)	NoC Y coordinate of SpiNNaker router
noc_x	(RW)	NoC X coordinate of SpiNNaker router

Register 21.49: SPIS_SPIKE_ADDR (0x0000019c)



val (RW) address field default value, for reduced SpiNNaker packet transfer length

21.8 I2C

21.8.1 I2C slave

1. I2C bus frequency (up to 3MHz)
2. full memory access
3. selectable 7bit slave address (default is 0x1A)
4. broadcast address 0x00
5. selectable sampling frequency
6. configurable SDA and SCL digital input glitch filter

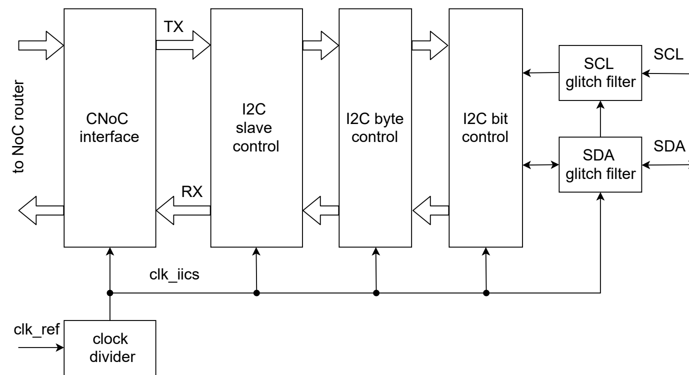


Figure 32: I2C slave structure

GPIO pads

The I2C slave can use up-to 3 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
IICS SCL	inout	I2C slave clock, open drain
IICS SDA	inout	I2C slave data, open drain
IICS INTR	output	I2C slave interrupt(packet available), low active, open drain

I2C slave configuration

The I2C slave core can be enabled by setting bit 0 of register 0x00000190 . By default the I2C slave address is 0x1A. The bus address can be changed by setting the slv_addr_rf field and enabling slv_addr_rf_en. To change the I2C sampling clock from 100MHz to lower values the clk_div can be used. Note that the SCL and SDA glitch filter length has to be changed accordingly.

Register 21.50: IICS_CONFIG (0x00000190)

	unused	sda_filter_taps	scl_filter_taps	sda_filter_bypass	scl_filter_bypass	clk_div	slv_addr_rf_en	slv_addr_rf	mmap_seg	priv_acc	en
Bits	31–24	23–21	20–18	17	16	15–12	11	10–4	3–2	1	0
Reset	—	7	7	0	0	1	1	0x1A	0	1	0

I2C slave command set

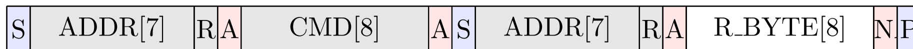
Command	Code	function
CMD_STATUS	0x00	read slave status
CMD_RD_REQ	0x1E	read request to 32bit mmap address
CMD_RD_BUF_D	0x2D	read 32bit data from buffer
CMD_RD_BUF_AD	0x33	read 32bit address + 32bit data from buffer
CMD_RD_BUF_HAD	0x48	read 32bit header + 32bit address + 32bit data from buffer
CMD_WR_NOC	0x55	write 32bit mmap address + 32bit data
CMD_WR_NOC_192	0x66	write raw packet 32bit header + 32bit addr + 128bit data
CMD_RD_BUF_192	0x78	read raw packet 32bit header + 32bit address + 128bit data from buffer

Note that the transfer byte order is from low to high: eg. command CMD_RD_REQ (read request to 32bit mmap address), command byte is followed by address byte 0, 1 and so on.

read commands

To execute a read command the I2C byte sequence is as followed:

1. start condition, slave address, write condition
2. read command (CMD_STATUS, CMD_RD_X)
3. repeated start, slave address, read condition
4. read n-1 bytes
5. read byte n with stop condition



write commands

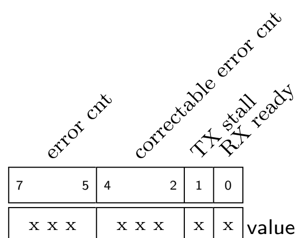
To execute a write command the byte I2C sequence is as followed:

1. start condition, slave address, write condition
2. write command (CMD_WR_NOC_X)
3. write n-1 bytes
4. write byte n with stop condition

slave status

Slave information can be read out with CMD_STATUS command:

Read request



21.8.2 I2C master

1. 16byte TX and RX FIFO buffer
2. selectable bus frequency (up to 1MHz)
3. interrupt generation
4. multi-master operation
5. option to share pads with I2C slave

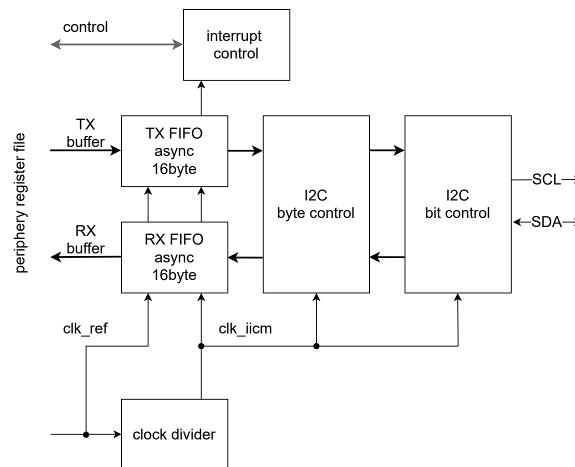


Figure 33: I2C master structure

GPIO pads

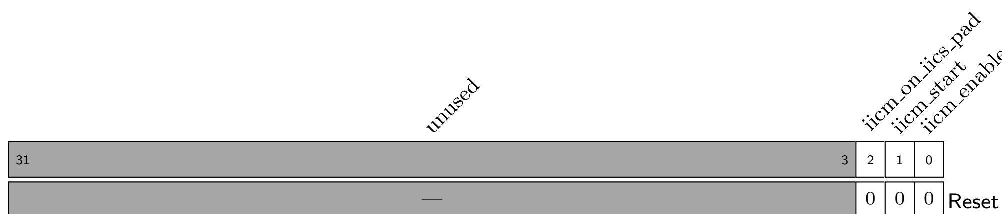
The I2C master can uses 2 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
IICM SCL	inout	I2C slave clock, open drain
IICM SDA	inout	I2C master data, open drain

I2C master enable

To enable the I2C master core bit 0 of register 0x00000080 is set to 1. In this state the TX buffer is accepting bytes by writing to address 0x00000090 . The transfer starts if bit 1 (iicm_start) is set additionally. The I2C master signals can be mapped onto the I2C slave pads by enabling bit 2 (iicm_on_iics_pad). In this mode the I2C master and slave can be operated at the same time.

Register 21.51: IICM_ENABLE (0x00000080)



iicm_on_iics_pad	(RW)	combine iic master and slave on slave pads
iicm_start	(RW)	start iic master core operation
iicm_enable	(RW)	enable iic master core

clock configuration

Clock configuration is done via register 0x00000084 and must be set before the I2C master is enabled. The master core logic is clocked by a clock derived from iicm_clkdiv value. SCL frequency is derived from iicm_prescaler (and iicm_clkdiv):

bus frequency	prescaler	clkdiv
100kHz	3	39
400kHz	3	9
1MHz	3	3

Register 21.52: IICM_CLK_CONF (0x00000084)

	unused	iicm_prescaler	iicm_clkdiv
Bits	31–24	23–8	7–0
Reset	—	3	10

iicm_prescaler	(RW)	iic master SCL prescaler
iicm_clkdiv	(RW)	iic master core clock divider

interrupt configuration

The I2C master is capable of generating an interrupt signal which is connected to periphery ARM core and to the interrupt bus. There are 6 types of interrupts which can be enabled by setting the interrupt mask value. The interrupt signal is the result of an or operation of all active interrupts.

Register 21.53: IICM_ITR_CONF (0x00000088)

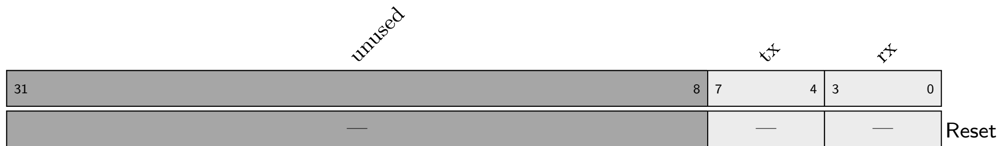
	unused	iicm_intr_rx_empty	iicm_intr_tx_full	iicm_intr_ai_err	iicm_intr_rx_cnt_tr	iicm_intr_tx_cnt_tr	iicm_intr_tx_empty	iicm_intr_mask	iicm_intr_rst	iicm_intr_tx_cnt	iicm_intr_rx_cnt
Bits	31–21	20	19	18	17	16	15	14–9	8	7–4	3–0
Reset	—	—	—	—	—	—	—	0	0	0	0

iicm_intr_rx_empty	(R)	triggered interrupt: RX buffer empty
iicm_intr_tx_full	(R)	triggered interrupt: TX buffer full
iicm_intr_ai_err	(R)	triggered interrupt: frontend error
iicm_intr_rx_cnt_tr	(R)	triggered interrupt: RX count
iicm_intr_tx_cnt_tr	(R)	triggered interrupt: TX count
iicm_intr_tx_empty	(R)	triggered interrupt: TX buffer empty
iicm_intr_mask	(RW)	enable mask of interrupts
iicm_intr_rst	(RW)	reset interrupt
iicm_intr_tx_cnt	(RW)	expected tx count to trigger interrupt
iicm_intr_rx_cnt	(RW)	expected rx count to trigger interrupt

byte counter

During operation the TX and RX buffer status can be tracked via register 0x0000008c . The buffers are cleared if the I2C core is disabled by register 0x00000080 .

Register 21.54: IICM_BYTE_CNT (0x0000008c)

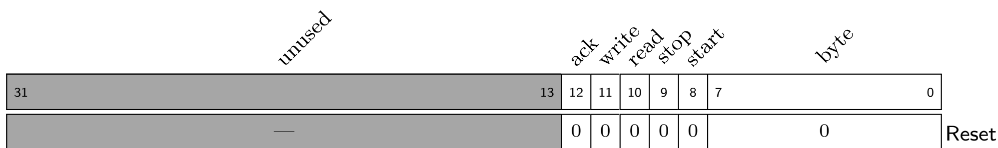


tx	(R)	number of bytes in TX buffer
rx	(R)	number of bytes in RX buffer

TX buffer

To define a I2C bus transfer a byte sequence is pushed to TX buffer via register 0x00000090 .

Register 21.55: IICM_TX_BUFFER (0x00000090)



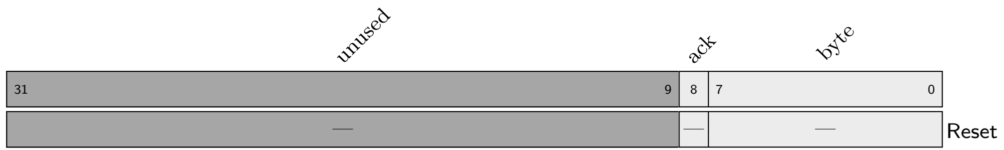
ack	(RW)	iic ack setting
write	(RW)	iic write byte
read	(RW)	iic read byte
stop	(RW)	iic stop condition
start	(RW)	iic start condition
byte	(RW)	iic data byte

A write to TX buffer register will push the data to the TX buffer FIFO, hence it should be written at once. The I2C byte control module is reading out the TX buffer and is executing the defined byte sequence. **start:** To start a I2C sequence the following fields should be set: start=1, to set the I2C start condition; write=1, master is writing a byte to bus; byte=0xXX: write byte e.g. slave address; read=1 (optional), write RX buffer to get ACK bit of slave, can be read out from RX buffer; other fields zero **write:** To write bytes to I2C slave the settings are: write=1; byte=0xXX; read=1 (optional): to get ACK information of slave; other fields zero. **read:** To read bytes from I2C slave the register settings are: read=1; ack=1; others zero. **stop:** To finish a I2C transfer, settings are: stop=1; other settings as write or read transfer.

RX buffer

The I2C master is storing receive data in the RX buffer which can be read via register address 0x00000094 . The number of stored byte can be read from byte counter register 0x00000080 . Note: data will be only stored in RX buffer if this was defined in register `IICM_TX_BUFFER` by setting read to one.

Register 21.56: IICM_RX_BUFFER (0x00000094)



ack	(R)	iic ack
byte	(R)	iic data byte

21.9 PWM

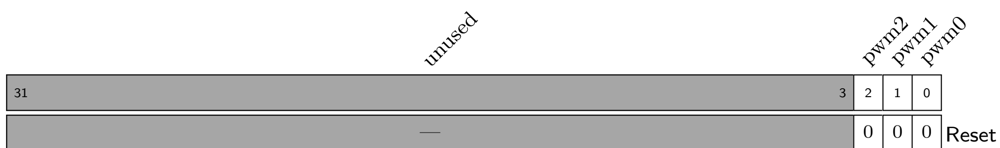
GPIO pads

The PWM cores can use up-to 7 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
PWM0	output	PWM 0 output
PWM1	output	PWM 1 output
PWM2_0-4	output	PWM 2 outputs 0 to 4

PWM enable

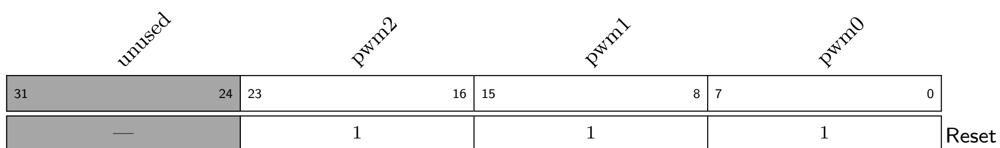
Register 21.57: PWM_ENABLE (0x00000050)



pwm2 (RW) enable PWM n channel core / PWM 2
pwm1 (RW) enable PWM core 1
pwm0 (RW) enable PWM core 0

PWM clock divider

Register 21.58: PWM_CLK_DIV (0x00000054)

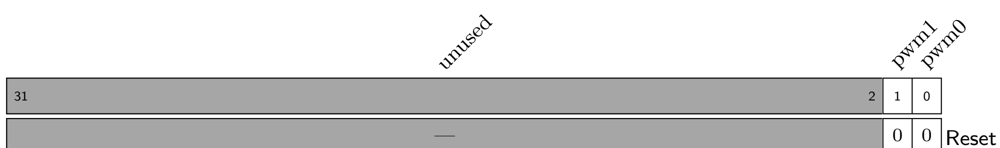


pwm2 (RW) clock divider PWM n channel core
pwm1 (RW) clock divider of PWM core 1
pwm0 (RW) clock divider of PWM core 0

21.9.1 PWM0 and PWM1

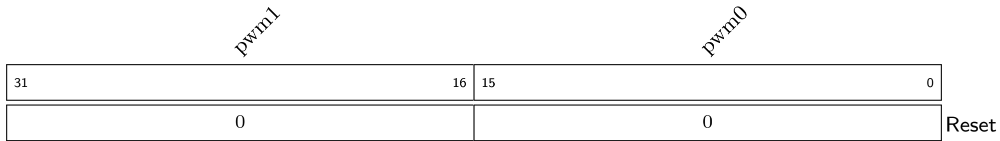
PWM mode configuration

Register 21.59: PWM_MODE (0x00000058)



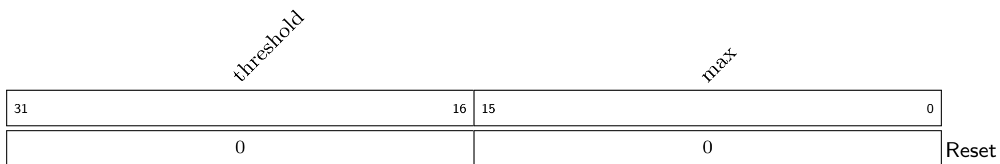
pwm1 (RW) increment mode PWM core 1
pwm0 (RW) increment mode PWM core 0

Register 21.60: PWM_STEP_VAL (0x0000005c)



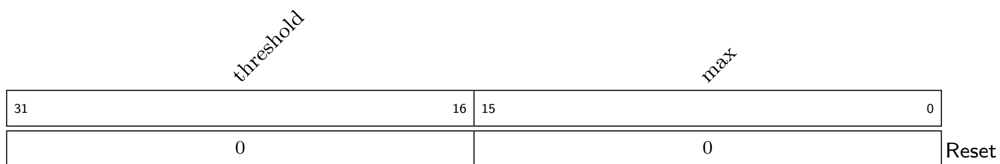
pwm1 (RW) step size of PWM core 1
pwm0 (RW) step size of PWM core 0

Register 21.61: PWM0_CNT (0x00000060)



threshold (RW) max cnt val of PWM core 0
max (RW) max cnt val of PWM core 0

Register 21.62: PWM1_CNT (0x00000064)



threshold (RW) max cnt val of PWM core 1
max (RW) max cnt val of PWM core 1

21.9.2 PWM2

The PWM2 core uses one 16bit counter with configurable clock frequency. Five comparator channels use the counter value and compare against the threshold register value (figure 34). If the threshold value was reached, the output will be switched to one. The output will be reset after the counter reaches zero (see figure 35). The mask value can be used to mask away the upper 8bit of the counter value. This allows to reduce the PWM resolution from 16 down to 8bit and therefore increasing PWM frequency. Additional each output channel can be inverted individually. Channel configuration registers are located at 0x00000068 to 0x00000078 .

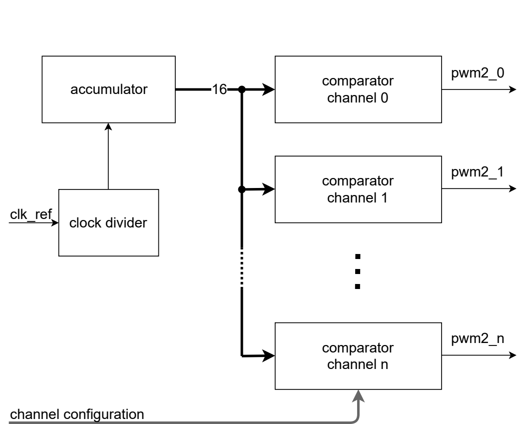


Figure 34: PWM 2 core structure

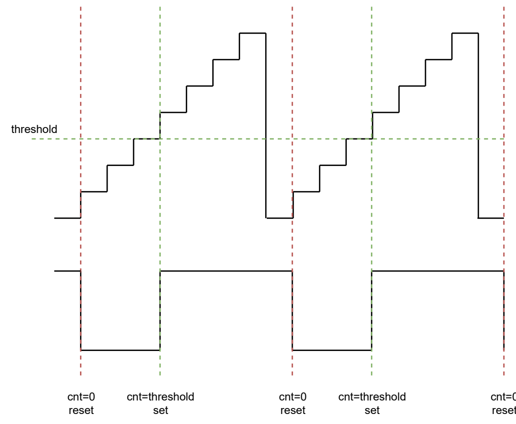


Figure 35: PWM 2 signal generation

Register 21.63: PWM2_CHANNEL_0 (0x00000068)

	unused	invert	mask	treshold
Bits	31 – 25	24	23 – 16	15 – 0
Reset	—	0	0	128

invert	(RW)	invert output of channel 0
mask	(RW)	cnt mask value to change frequency of channel 0
treshold	(RW)	threshold value to switch output of channel 0

21.9.3 GPIO debug output

21.10 UART

21.10.1 CUART

GPIO pads

The configuration UART core use 2 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
CUART TX	output	configuration UART transmit pad
CUART RX	input	configuration UART receive pad

configuration

Register 21.64: CUART_CONFIG (0x00000178)

unused	trigger	val	en	
31–26	25	24–1	0	
—	0	868	0	Reset

trigger (RW) trigger to update config.
val (RW) cuart clk divider default is 115.200kBaud
en (RW) enable cuart

status

Register 21.65: CUART_STATUS (0x0000017c)

unused	clkdiv	active	
31–25	24–1	0	
—	—	—	Reset

clkdiv (R) cuart clk divider read-out
active (R) enable is running

21.10.2 Printf UART

The printf UART core use up to 4 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
UART TX	output	printf UART transmit pad
UART RX	input	printf UART receive pad
UART CTS	input	printf UART clear to send
UART RTR	output	printf UART ready to receive

21.11 SDC Interface

GPIO pads

The configuration SD controller core use 6 GPIO pads. To map the desired function to GPIO pad please see section 22.4.

name	direction	function
SDIO CLK	output	SD bus clock
SDIO CMD	inout	command bus
SDIO D0	inout	data bus bit 0
SDIO D1	inout	data bus bit 1
SDIO D2	inout	data bus bit 2
SDIO D3	inout	data bus bit 3

Register 21.66: SDC_CONFIG (0x00000210)

	unused	clock_disable_en	rd_init_req	rd_req_size	global_segment	sdc_clock_en	clock_divider_en
Bits	31–10	9	8–7	6–4	3–2	1	0
Reset	—	0	0	2	0	0	0

clock_disable_en (RW🔒) disable SD clock if DMA fails to fetch data fast enough
rd_init_req (RW🔒) number of additional noc read requests at cycle start
rd_req_size (RW🔒) noc read request size, valid: 2=32bit; 3=64bit; 4=128bit
global_segment (RW🔒) global mmap segement select
sdc_clock_en (RW🔒) sd clock enable
clock_divider_en (RW🔒) sd clock divider enable

Register 21.67: SDC_ARGUMENT_REG (0x00000214)

	CMDA
Bits	31–0
Reset	0

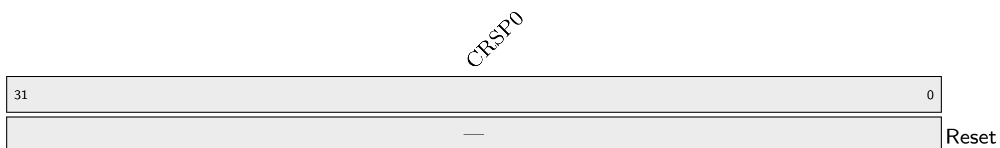
CMDA (RW🔒) CMDA command argument; command data, when writing to this register the transmission starts

Register 21.68: SDC_CMD_SETTING (0x00000218)

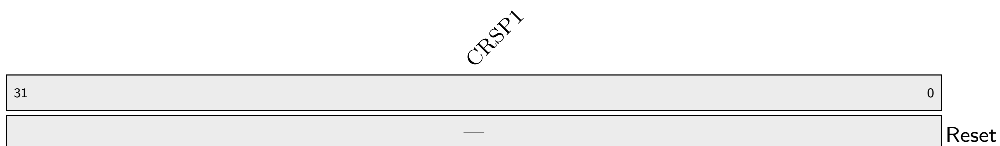
	unused	CMDI	unused	CWD	CICE	CIRC	CBSY	RTS
Bits	31–14	13–8	7	6–5	4	3	2	1–0
Reset	—	0	—	0	0	0	0	0

CMDI	(RW🔒)	CMDI - command Index; Index of the next command
CWD	(RW🔒)	CWD - data transfer specification. 0x0 - no data transfer; 0x1 - triggers read data transaction after command transaction; 0x2 - triggers write data transaction after command transaction
CICE	(RW🔒)	CICE - command index check; 0 : Do not perform index check on response CMD; 1 : Perform index check on response CMD
CIRC	(RW🔒)	CIRC - command CRC check; 0 : Do not perform CRC check on response CMD; 1 : Perform CRC check on response CMD
CBSY	(RW🔒)	CBSY - check for busy signal after command transaction (if busy signal will be asserted after command transaction, core will wait for as long as busy signal remains)
RTS	(RW🔒)	RTS - response check config. 0x0 - don't wait for response; 0x1 - wait for short response (48-bits); 0x2 - wait for long response (136-bits)

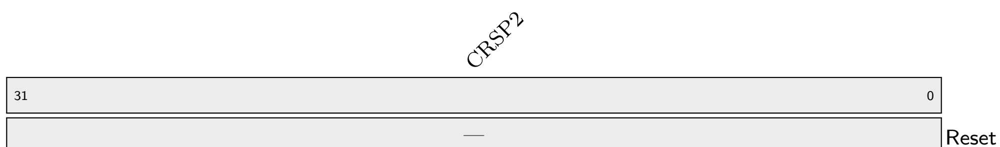
Register 21.69: SDC_RESPONSE_0 (0x0000021c)



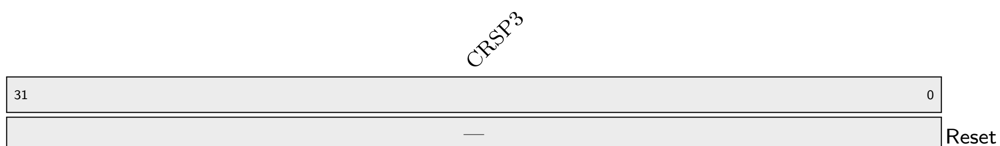
Register 21.70: SDC_RESPONSE_1 (0x00000220)



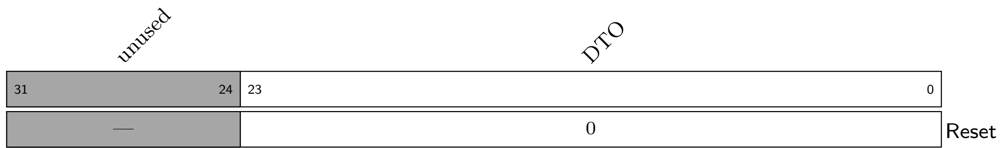
Register 21.71: SDC_RESPONSE_2 (0x00000224)



Register 21.72: SDC_RESPONSE_3 (0x00000228)

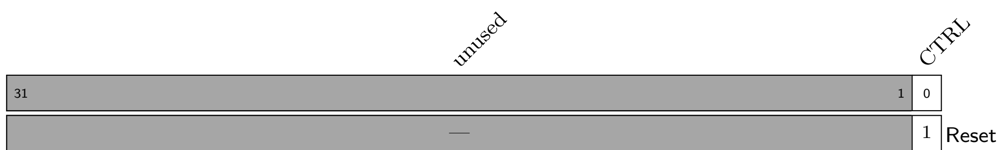


Register 21.73: SDC_DATA_TIMEOUT (0x0000022c)



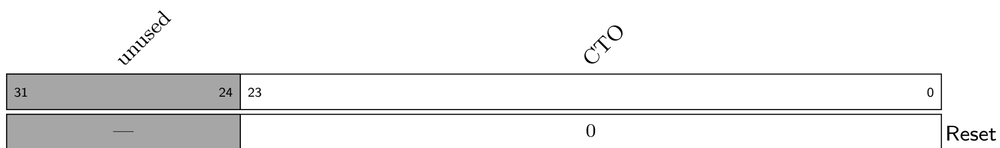
DTO (RW🔒) DTO - data transfer timeout register, 0=disabled

Register 21.74: SDC_CONTROL_SETTING (0x00000230)



CTRL (RW🔒) CTRL -control register, SD bus width, 0=1bit mode, 1=4bit mode

Register 21.75: SDC_CMD_TIMEOUT (0x00000234)

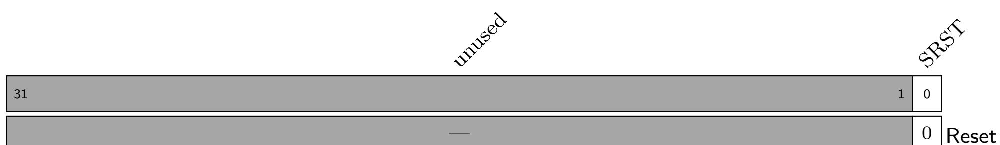


CTO (RW🔒) CTO - command transfer timeout register, 0=disabled

Register 21.76: SDC_CLOCK_DIVIDER (0x00000238)

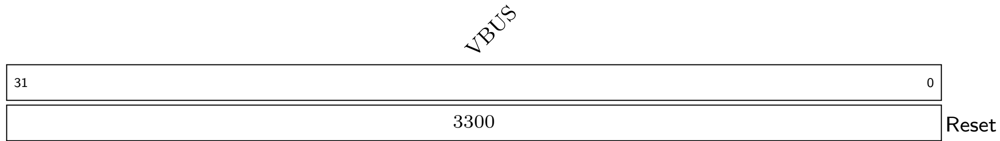
CLKD (RW🔒) CLKD - clock divider, SD controller clock

Register 21.77: SDC_SOFTWARE_RESET (0x0000023c)



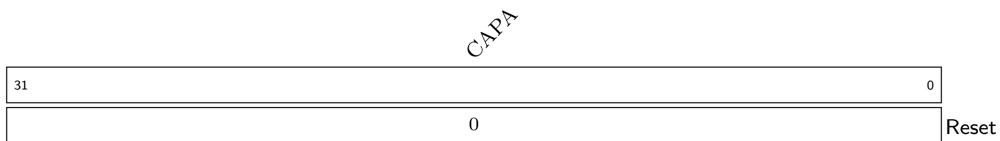
SRST (RW🔒) SRST - software reset; 0:1: Reset the hardware

Register 21.78: SDC_VOLTAGE (0x00000240)



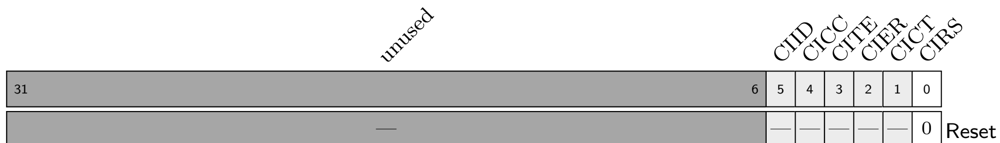
VBUS (RW🔒) VBUS - bus voltage information register, 3.3V

Register 21.79: SDC_CAPABILIES (0x00000244)



CAPA (RW🔒) CAPA - capability register

Register 21.80: SDC_CMD_INT_STATUS (0x00000248)

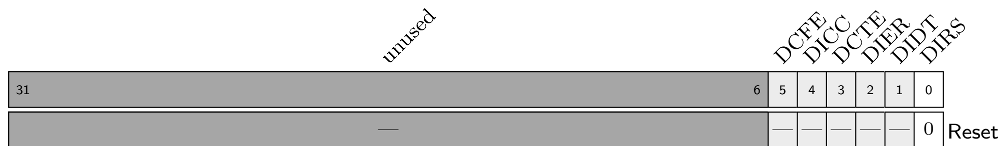


CIID (R🔒) CIID - interrupt: index error
CICC (R🔒) CICC - interrupt: CRC error
CITE (R🔒) CITE - interrupt: timeout error
CIER (R🔒) CIER - interrupt: error
CICT (R🔒) CICT - interrupt: command transaction successful completed
CIRS (RW🔒) CIRS - interrupt: reset register, resets triggered resets

Register 21.81: SDC_CMD_INT_ENABLE (0x0000024c)

EN_CIID (RW🔒) EN_CIID - enable: index error interrupt
EN_CICC (RW🔒) EN_CICC - enable: CRC error interrupt
EN_CITE (RW🔒) EN_CITE - enable: timeout error interrupt
EN_CIER (RW🔒) EN_CIER - enable: error interrupt
EN_CICT (RW🔒) EN_CICT - enable: command transaction successful completed interrupt

Register 21.82: SDC_DATA_INT_STATUS (0x00000250)



DCFE (R🔒) DCFE - interrupt: FIFO error
DICC (R🔒) DICC - interrupt: CRC error
DCTE (R🔒) DCTE - interrupt: timeout error
DIER (R🔒) DIER - interrupt: error
DIDT (R🔒) DIDT - interrupt: data transaction successful completed
DIRS (RW🔒) DIRS - interrupt: reset register, resets triggered resets

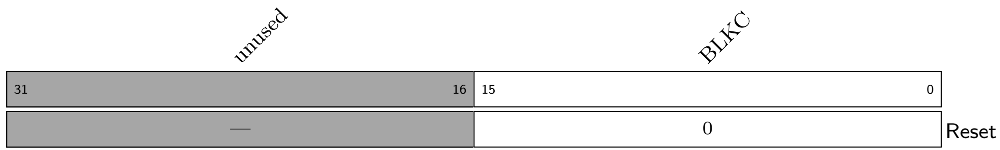
Register 21.83: SDC_DATA_INT_ENABLE (0x00000254)

EN_DCFE (RW🔒) EN_DCFE - enable: FIFO error interrupt
EN_DICC (RW🔒) EN_DICC - enable: CRC error interrupt
EN_DCTE (RW🔒) EN_DCTE - enable: timeout error interrupt
EN_DIER (RW🔒) EN_DIER - enable: error interrupt
EN_DIDT (RW🔒) EN_DIDT - enable: data transaction successful completed interrupt

Register 21.84: SDC_BLOCK_SIZE (0x00000258)

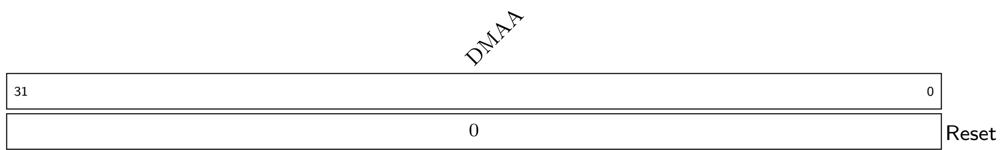
BLKS (RW🔒) BLKS - block size, max. counter value for block byte counter

Register 21.85: SDC_BLOCK_COUNT (0x0000025c)



BLKC (RW🔒) BLKC - block count, max. counter value for block number counter

Register 21.86: SDC_DMA_ADDR_REG (0x00000260)



DMAA (RW🔒) DMAA - DMA NoC mmap address

22 Host Interface

The SpiNNaker system connects to a host machine via a high-speed serial link. Each SpiNNaker chip includes a host link, although only a few of the chips are expected to use this interface.

In essence the module provides the following features:

1. 16x dynamic ports and one static port (1800) for incoming NOC data
2. 16x UDP port dependent routing and 16x dynamic ports for incoming Raw data
3. 16x Modid dependent udp routing for outgoing data
4. 16x C2C channel dependent udp routing for outgoing data
5. Spinnaker packet type (mc, nn, c2c, grw) dependent udp routing for outgoing data
6. Optional UDT layer and/or frame id protocol as measure to counteract possible packet loss
7. data alingment of incoming raw data (descend order, 32b/128b input, 32b order of 128b input, addr inc disable)
8. 5 different data modes for outgoing data (default and for each outgoing routing port)
9. error & status check registers

22.1 UDP Routing

UDP routing is divided into incoming (ic) and outgoing (og) packets. Both directions have different features and routing modes.

22.1.1 Incoming Packets

Incoming Packets are divided into NoC packet format (see attachement F) and raw data format. Both options provide a set of 16 dynamically set ports each (0x00000074 - 0x00000090 for noc and 0x00000094 - 0x000000b0 for raw). The module identifies the incoming data format by the port which is used for the incoming data. If the data is formatted as one or multiple NoC packets there are 3 possible burst formats one can utilize the UDP data frame headers listed in table 22.1.1. Figure 36 shows the three possibilities for data transfer with the NoC data type. For the incoming noc format with only one address field each new addr for the resulting noc packet is incremented by the size of the payload. It is further possible to disable the address increment by setting ic_addr_inc_disable.

Label	Magic	Description
UDP_MAGIC_SEND_A	0x722acce7	data send in always contain noc header and addr for each 128bit payload
UDP_MAGIC_SEND_B	0x722acce8	data sent in contains only one noc header and addr for full udp datagramm
UDP_MAGIC_SEND_C	0x722acce9	data sent in always contain add for each 128bit payload but only one noc header

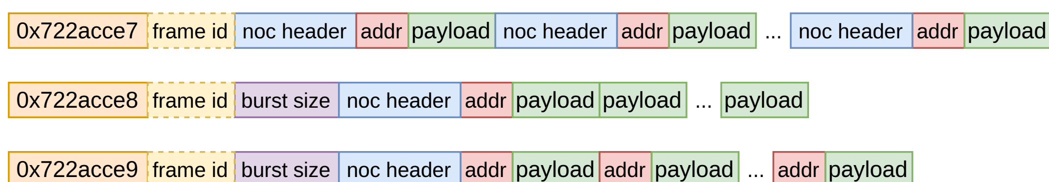


Figure 36: Incoming Packet Format for NoC Packets. Each field is 32bit big, except payload which can be 32bit, 64bit or 128bit. Please note that "frame id" is optional and disabled by default (see section 22.1.4). Burst size is the number of payload fields. If UDT is enabled this data field gets wrapped by a UDT header (see **??**).

For the incoming raw format each dynamic port has a 32b global address regfile entry paired with it. These global addresses determine to where the resulting noc packets are routed to. Per default the addresses of the generated noc packets are incremented for each resulting noc packet. Furthermore one datagramm frame can be translated into different noc packets by control bits. Those can be mixed. Figure 37a shows the default interpretation. Figure 37b-e shows the effect of the transformation to noc packets if specific control bits are flipped.

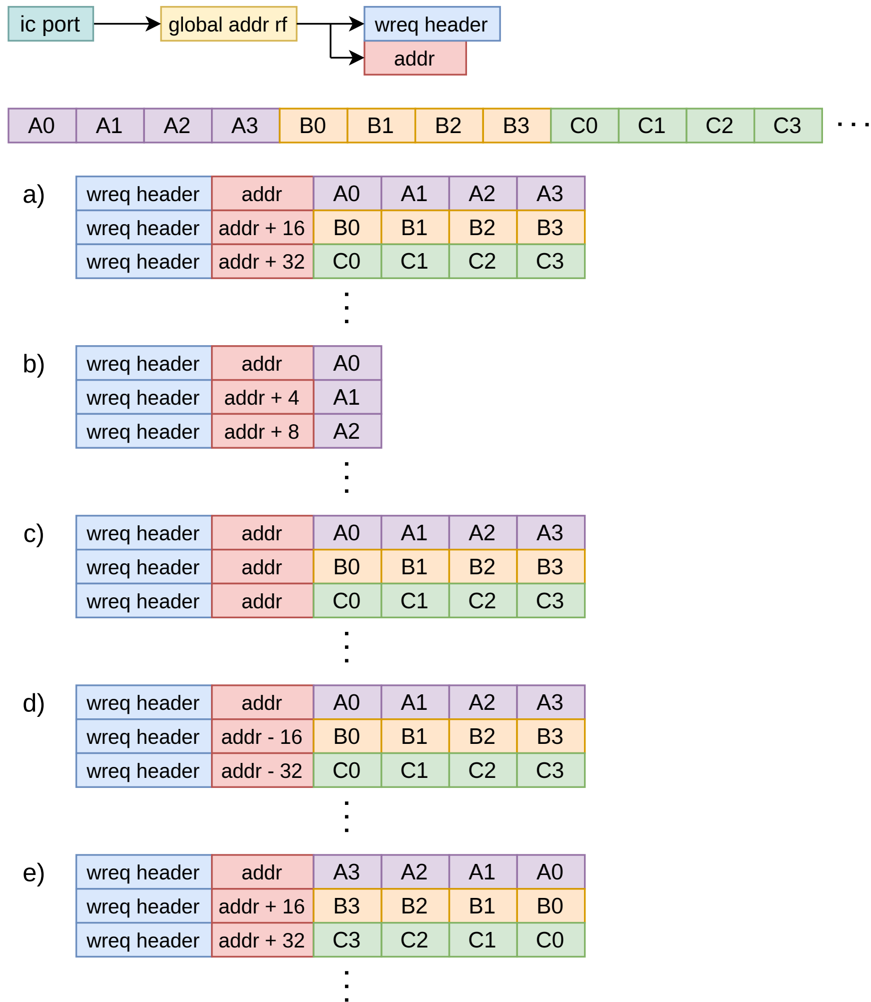


Figure 37: Options of noc packet conversion. The address and the write request noc header field is generated by the udp port selected register field which contains a global s2 address starting at 0x000000b4 . The noc packet interpretation can be changed by the following options. Each field is 32bit wide (except "ic port"). a) Using the default configuration. b) Enable ic_raw_32b_mode. c) Disable ic_raw_save_addr. d) Enable ic_raw_descend. e) Enable ic_raw_order32b.

22.1.2 Outgoing Packets

The outgoing direction provides different UDP datagram interpretations which can be controlled with og_data_mode. As it is with the incoming direction the user can choose between different burst modes and raw data formats which are described in figure 38. For the burst mode the user can set the maximum UDP datagram size with og_mtu and if the UDP datagram should be send out earlier if a noc packet arrives with a different noc header than the previous burst data noc packets with og_burst_break. The udp_noc module waits a time specified in (unoc_ctrl1) before sending out a burst UDP datagram. This timeout can also be disabled. The UDP source/destination port is either selected from a programmable register file entry or by reusing the ports from the last incoming UDP datagram. The send out is triggered either automatically or by setting og_auto_mode and og_trigger accordingly. Additionally a sendout can be triggered by magic packets described in table **??**. Furthermore the user can choose to use a LUT table for ip addresses and destination ports which are either dependent on the modid of noc packets or dependent on the SpiNNaker2 noc packet type. The routing priority is type routing ¿ modid routing ¿ default routing. For the c2c type the user can specify a different routing for different c2c channels. Each routing table entry has its own data_mode entry.

Description	Magic
UDP_MAGIC_RECV	0x2e2acce7
UDP_MAGIC_RECV_BURST	0x2e2acce9
UDP_MAGIC_RECV_RAW	0x2e2accea
UDP_MAGIC_RECV_RAW_BURST	0x2e2acceb

22.1.3 Packet Counters

For each incoming and outgoing packets there is a counter regfile entry. The outgoing direction differs by counting type specific routing (each type has its own counter, see **??** to **??**), modid specific routing (one counter, see **??**) and a counter for everything else (see **??**). The incoming direction has a counter for each entry of the 16 noc port LUT table (see **??** to **??**) and for each eantry of the 16 raw port LUT table (see **??** to **??**). Each entry can be reset by the host by sending a specific magic for each entry which are shown in the next table. x values must be replaced by the index of the entry e.g. resetting the third entry for incoming raw packets with 0x6e2a0003.

Description	Magic
UDP_MAGIC_RESET_CNT_ALL	0x4e2acce6
UDP_MAGIC_RESET_CNT_IC_ALL	0x4e2acce7
UDP_MAGIC_RESET_CNT_OG_ALL	0x4e2acce8
UDP_MAGIC_RESET_CNT_IC_NOC	0x5e2axxxx
UDP_MAGIC_RESET_CNT_IC_RAW	0x6e2axxxx
UDP_MAGIC_RESET_CNT_OG_DEF	0x7e2acce5
UDP_MAGIC_RESET_CNT_OG_NN	0x7e2acce6
UDP_MAGIC_RESET_CNT_OG_MC	0x7e2acce7
UDP_MAGIC_RESET_CNT_OG_GRW	0x7e2acce8
UDP_MAGIC_RESET_CNT_OG_C2C	0x7e2acce9
UDP_MAGIC_RESET_CNT_OG_C2C_L	0x8e2axxxx
UDP_MAGIC_RESET_CNT_OG_MODID	0x9e2axxxx

22.1.4 Frame ID Protocol

On top of UDT the host interface also provides a simple packet count protocol. If activated for either incoming or outgoing packets an additional frame_id field is added for non-raw UDP packets. For incoming packets this incrementing field is compared to the packet counters and sends out an error signal as UDP datagram.

Description	Magic
UDP_MAGIC_STATUS	0x2e2acce8
UDP_MAGIC_FRAME_ID_ERROR	0xae2acce8

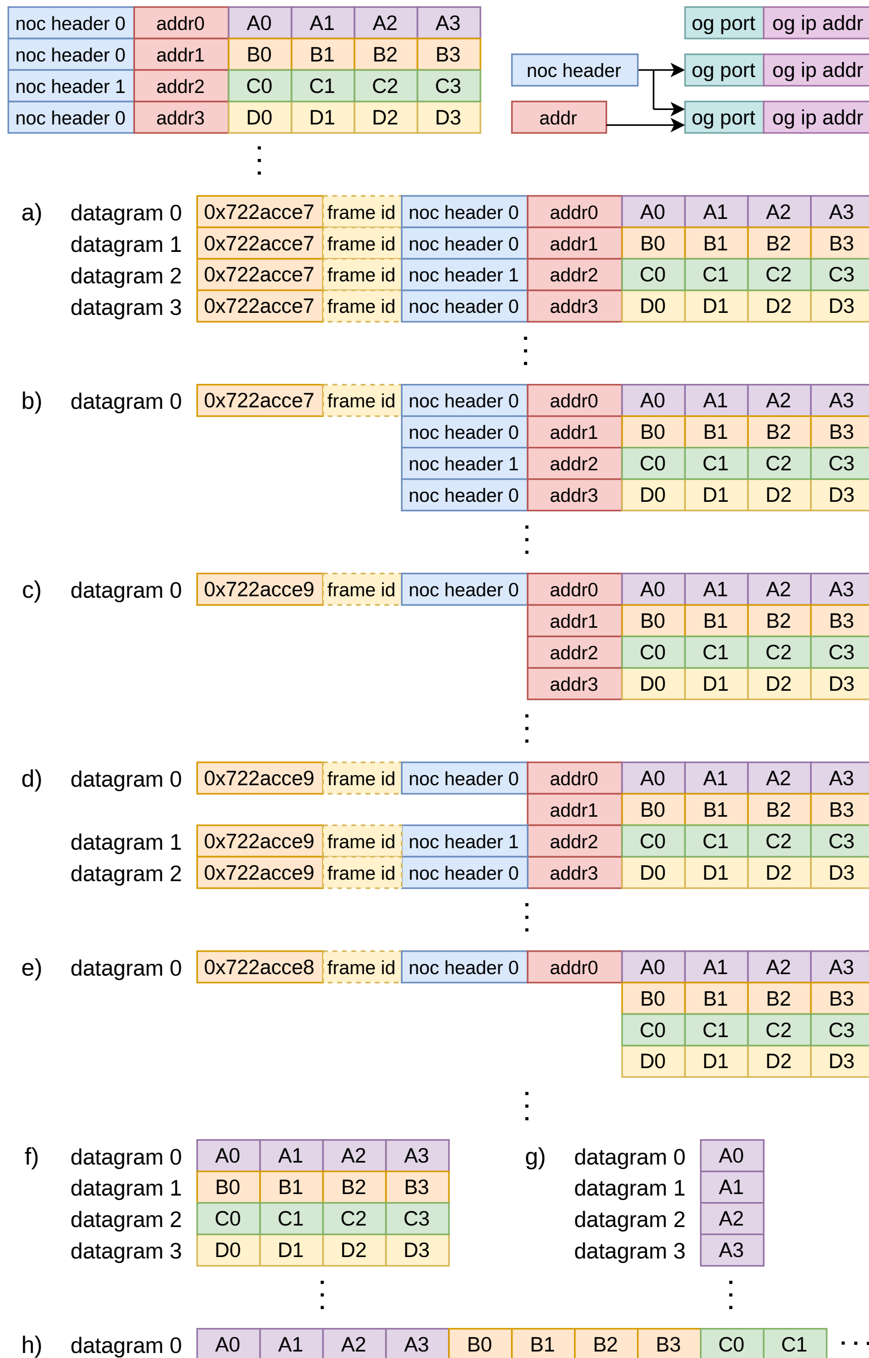


Figure 38: Options of UDP datagram conversion for an incoming stream of noc packets from chip side. Each field is 32bit wide (except "og port"). The destination port and destination ip address are selected by either a default regfile entry (**??**) or picked out by the LUT for modid or type routing which corresponds to the modid field or Spinnaker packet type (with channel field if c2c) respectively. Frame id is optional and disabled per default. a) data_mode=0. b) data_mode=1. c) data_mode=2. d) data_mode=2 + og_burst_break enabled. e) data_mode=3. f) data_mode=4. g) data_mode=5. h) data_mode=6.

22.2 UDT

Although UDP provides a high-speed data transmission between host and chips, the unguaranteed reliability prevents some data completeness demanding applications. To address this, a UDP-based Data Transfer Protocol (UDT) module is integrated into the spinnaker host interface. The UDT module has the following features:

1. High throughput designed for 1 Gbps.
2. Transparent to UDP packets according to port number.
3. Apply both control packets and data packets.
4. Allow loss packet resending.
5. Apply congestion control to avoid transmission jam.

22.2.1 UDT Packet Type

UDT provides data packet and 6 kinds of control packets (see Fig.39). Each data packet is assigned with a unique sequence number raising from 1 up to 0x7fffffff. The entire UDP frame is encapsulated as payload of UDT data packet. Control packets carry only scheduling information. Handshake packets (HS) are exchanged twice between two sides at the beginning to establish the UDT communication channel. Keep-alive packets (KA) is used to refresh timers of the peer side to avoid the channel close induced by time out. Acknowledgement (ACK) feeds back the maximum received data sequence number along with other information of the receiver, such as round-trip time (RTT), standard variance of RTT (RTTVar), free buffer size, packet arrival speed and estimated link capacity. As the response of an ACK packet, an acknowledgement of the acknowledgement (ACK2) confirms with the same ACK number. The sequence number of detected loss packets are compressed and transmitted by the negative acknowledgement (NAK). To close the channel, UDT informs the peer entity with the shut down (SD) packet. The individual packet and its main payloads are listed in Table 12.

32 bits

0	Packet Sequence Number	
FF	O	Message Number
Time Stamp		
Destination Socket ID		
Data		

(a) Data Packet

32 bits

1	Type	Reserved
Additional Info		
Time Stamp		
Destination Socket ID		
Control Information Field		

(b) Control Packet

Figure 39: UDT packet structure

Table 12: UDT packet overview

Type	Head	Payload
DATA	< 0x80000000	timestamp, data
HS	0x80000000	timestamp, socket ID, initial seq number, flow window size, cookie, destination IP
KA	0x80010000	timestamp
ACK	0x80020000	timestamp, LASN, RTT, RTTVar, available buffer size, arrival speed, link capacity
NAK	0x80030000	timestamp, compressed loss seq number
SD	0x80050000	timestamp
ACK2	0x80060000	timestamp, ACK number

22.2.2 Channel Set up and Shut down

The connection mode of UDT in spinnaker2 is in server-client paradigm. The host is the client to launch accesses, whereas spinnaker2 is the server for responding. Therefore in the idle state, the UDT in spinnaker2 is always listening to the listen socket and only responds the handshake request. After the first handshake response is received, a second handshake will be sent by the client to verify the cookie. If the authentication is passed with the second handshake response, the UDT data channel is open between the client and server.

Three approaches are feasible to shut down the established channel,. First, users explicitly set the shut down register in register file. Second, UDT received a shut down packet from the peer side. Third, the expiration timer times out as there is no sending or receiving actions for a long time (30 seconds). After the shut down, UDT returns to the idle state to wait for the next handshake packet.

22.2.3 Architecture

As a UDP-based protocol, UDT parses packets before UDP frames are converted to NoC packets, therefore the UDT module is connected with the UDP Ethernet interface and the UDP-NoC converter via *udp_ic/og* and *udt_ic/og*, respectively. The top level view of UDT is illustrated in Fig. 40.

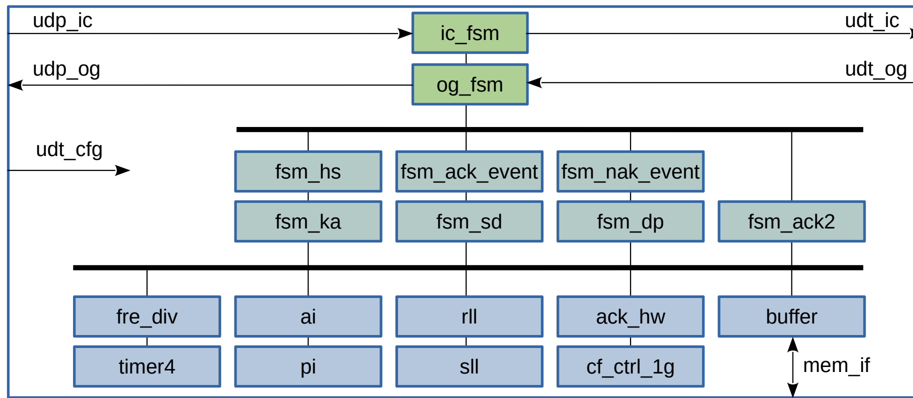


Figure 40: Top view of UDT module

UDT is designed as a configurable module addressing multifarious application scenarios. Users set UDT configuration registers in *udt_noc_regfile* to shape the behaviours of UDT, such as buffer size, congestion control, etc. More configuration details can be referred to section 22.2.4. Those set values are fed into UDT via *udt_cfg* signals.

There are 18 sub-modules abstracted into 3 levels. The 2 finite state machines (FSMs) in top level take over all packets in-flight from 2 directions. The 7 dedicated FSMs in the middle layer load information from units in bottom level to pack up an outgoing packet once corresponding event is triggered. The 9 functional units in the bottom layer calculate the information required by control packets, maintain and update values regularly. Their functionalities are described as follows.

fre_div The frequency divider divides the reference frequency of 100 MHz into 1 MHz and generates system time stamps with the unit of microsecond, which are carried in every UDT packets.

timer4 A cluster of 4 timers counting down to trigger data sending (SND), acknowledgement (ACK), negative acknowledgement (NAK) and expiration (EXP) events as initiatives. When condition is not met, the trigger signal is ignored. The minimal and maximal expiration time are 27 s, 30 s, respectively.

ai The arrival interval estimator stores the latest 16 data packet time interval and takes the median value as the current arrival interval. A look-up table is used to convert the interval to data arrival speed, which is inserted into the ACK packet.

pi The data packet pair estimator investigates the arrival interval of 2 adjacent data packets with sequence number $16n$ and $16n + 1$, also known as a packet pair to estimate the link capacity. Analogously, a median value filtering and look-up table based calculation is used to obtain the packet pair speed, which is loaded into the ACK packet as well.

rll The receiver's loss list stores the sequence number of all detected loss packets at receiving from discontinuous received sequence number. The stored items will be removed once new ACK packets confirm them.

sll The sender's loss list stores the sequence number of all detected loss packets at sending in a compressed format, either from NAK packets or the all unacknowledged sent packets at EXP event. As long as sender's loss list is not empty, all loss packets are resent prior to any new data packets due to the higher priority.

ack_hw The acknowledgement history window stores the sending time of ACK and receiving time of ACK2 packets to calculate the round-trip time (RTT) and its variance (RTTVar), which are used to dynamically regulate the control packets sending period.

cf_ctrl_1g The congestion control targeting 1 Gbps adjusts the congestion window size and data sending period through DAIMD algorithm. A brake signal from configuration register slows down the speed in advance to avoid the happening of congestion.

buffer The buffer only manages data packets, which aims to not only transfer data to the destination module, but also preserves a snapshot in the local TX/RX memory. Sending buffering is required for resending request in case packet loss is detected at the receiver, whereas receiving buffering is useful to maintain the packet order during the wait for the missing packet, so that packets will be parsed by next modules as desired. A 192 kB SRAM based buffer memory is connected with this module via *mem_if*. A detailed buffer memory can be referred to section 22.2.5.

22.2.4 UDT Configuration

UDT configuration registers are used to regulate UDT to fit the communication requirement. Their names, bit width and reset values are listed in Table 13. The buffer memory accommodates in total 128 data packets. As a balanced setting, TX and RX buffer have half each as default, but for any specific biased case, e.g. TX is dominating, unbalanced buffer size is also supported. However, due to the limitation of port numbers of memory, an alignment of 16 packets should be respected. The possible combinations of buffer size are listed in Table 14.

In terms of power saving, UDT is clock gated and disabled in default. To activate this module, *udt_clock_en* and *udt_enable* should be set. Conversely, *udt_shutd* is applied to forcibly close the UDT connection, which can be manipulated by the program running in PE.

Table 13: UDT configuration registers

Name	Reset	Description
udt_rx_size	7'd64	RX buffer size [packets]
udt_tx_size	7'd64	TX buffer size [packets]
udt_shutd	1'd0	UDT shut down
udt_clock_en	1'd0	UDT clock open
udt_enable	1'd0	UDT enable
udt_socket	32'd2021	UDT server socket
udt_brake	16'd15	UDT brake threshold
udt_snd_init	16'd63	UDT initial send period
udt_syn	16'd10000	UDT minimal synchronization period
udt_cwnd_init	16'd16	UDT initial congestion window size [packets]

The initial data sending speed of UDT is defined by *udt_snd_init*. The speed adapts dynamically according to the link situation during the transmission. It aims to achieve 1 Gbps if no congestion occurs. By approaching the peak value, users are allowed to define a brake threshold (*udt_brake*), after

Table 14: Supported configuration of the buffer size

rx_size [packets]	tx_size [packets]	total [packets]	comment
16	112	128	TX biased
32	96	128	TX biased
48	80	128	TX biased
64	**64**	**128**	**balanced**
80	48	128	RX biased
96	32	128	RX biased
112	16	128	RX biased

which the speed increases quasi-linearly instead of exponentially. This brake concept is not contained in the original UDT protocol but an empirical result. The speed and the corresponding configure values have a constant speed-value product of 12000 Mbps. Hence the default brake threshold is 800 Mbps, whereas the initial sending speed is around 200 Mbps.

Analogously, the sending period of control packets (ACK, NAK) is also adapted dynamically based on the estimation of RTT. However, a lower limitation of this synchronization period is coded by *udt_syn* of 10 ms with value 10000, in order to save bandwidth for data transmission as much as possible. To assist the rate control, UDT applied flow control as well with a sliding congestion window to constrict the maximum on-flight data packets, which is defined as the window size and whose initial value is set as *udt_cwnd_init* of 16.

22.2.5 Buffer Memory

The buffer memory, in total of 192 kB and instantiated with 24 SRAM macros (2048×38 bits, 6 of 38 for ECC), each with 8 kB, is allocated as RX and TX buffer. As a UDP frame is limited by the maximum size of 1500 bytes, each UDT buffer segment is assigned with 1.5 kB for each UDT data packet, namely 384 words, as shown in Fig.41. The first word is stored with the data sequence number, followed by the data word length of this segment and the data payload. Due to the port number limitation, the RX/TX buffers are required to be separated in different SRAM banks. Hence, the RX/TX buffer size is regulated every 16 segments (16×1.5 kB = 24 kB = 3×8 kB) via register file, referring to Table 14.

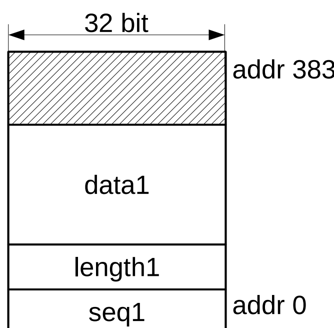


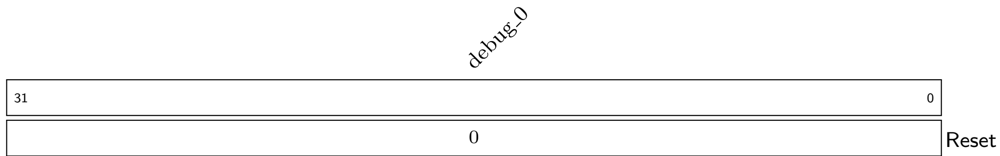
Figure 41: One segment of UDT buffer memory

22.3 Register summary

Debug Regs and Regfile Ctrl

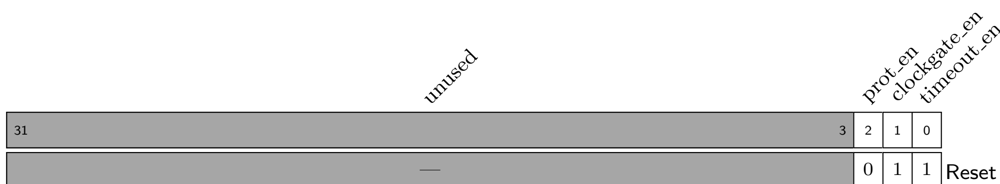
Addresses 0x00000000 to 0x0000000c are debugging registers. These can be used to test access via the ethernet interface.

Register 22.1: DEBUG_0 (0x00000000)



debug_0 (RW) debug register #0

Register 22.2: REGFILE_CTRL (0x00000010)

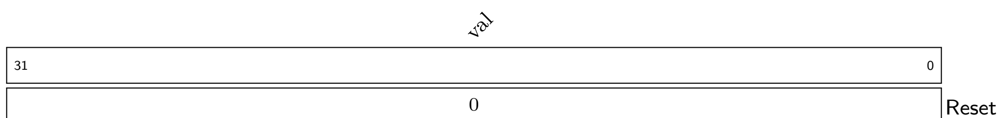


prot_en (RW) enable protected mode
clockgate_en (RW) enable clock gating for register file
timeout_en (RW) enable timeout for the case regfile is not reacting

UDT Buffer SRAM Control

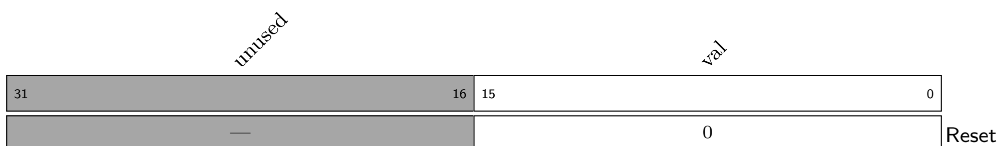
The UDT utilizes a SRAM buffer which can be controlled by accessing the regfile adressess 0x00000020 to 0x00000040 .

Register 22.3: MBIST_CMD_SEQ0 (0x00000020)



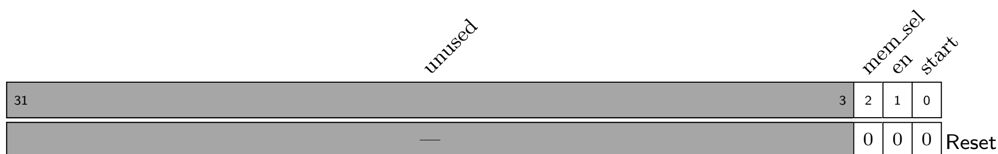
val (RW) bist command sequence lower 32bit

Register 22.4: MBIST_CMD_SEQ1 (0x00000024)



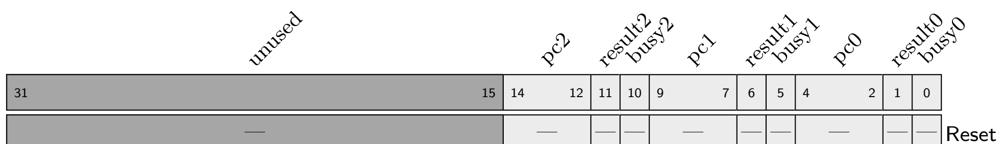
val (RW) bist command sequence upper 16bit

Register 22.5: MBIST_CTRL (0x00000028)



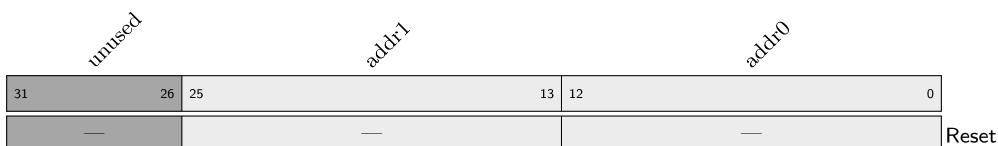
mem_sel	(RW)	memory wrap selection
en	(RW)	enable bist of udt buffer
start	(RW)	start mbist

Register 22.6: MBIST_STATUS (0x0000002c)



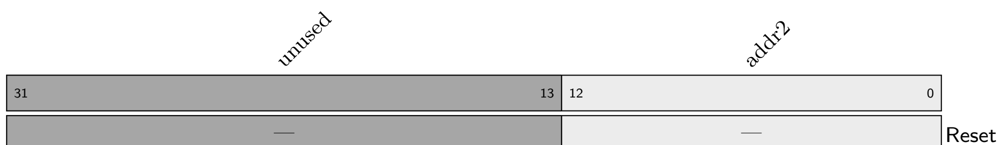
pc2	(R)	captured program counter
result2	(R)	bist test result 0: PASS 1: FAIL
busy2	(R)	bist test is running
pc1	(R)	captured program counter
result1	(R)	bist test result 0: PASS 1: FAIL
busy1	(R)	bist test is running
pc0	(R)	captured program counter
result0	(R)	bist test result 0: PASS 1: FAIL
busy0	(R)	bist test is running

Register 22.7: MBIST_ADDR0 (0x00000030)



addr1	(R)	failed addr
addr0	(R)	failed addr

Register 22.8: MBIST_ADDR1 (0x00000034)



addr2 (R) failed addr

Register 22.9: SRAM_CTRL (0x00000040)

31–12	11–10	9–4	3	2	1	0	
unused	retention	delay	ecc_err2	ecc_err1	ecc_err0	ecc_en	
—	0	0b1000	—	—	—	1	Reset

retention	(RW)	SRAM retention setting for power management state off
delay	(RW)	SRAM delay setting
ecc_err2	(R)	detected SRAM ECC error
ecc_err1	(R)	detected SRAM ECC error
ecc_err0	(R)	detected SRAM ECC error
ecc_en	(RW)	enable SRAM ECC

UDT Layer General Control

The UDT can be controlled with the regfile addressess 0x00000050 to 0x0000005c . For a more in depth description please refer to section 22.2.4.

Register 22.10: UDT_CFG0 (0x00000050)

31–17	16–10	9–3	2	1	0	
unused	udt_rx_size	udt_tx_size	udt_shutd	udt_clock_en	udt_enable	
—	64	64	0	0	0	Reset

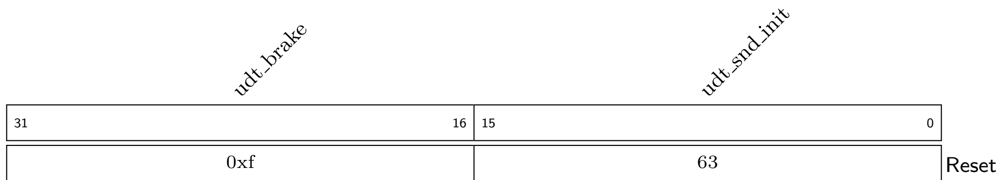
udt_rx_size	(RW)	RX buffer size (packets [0,127]), tx+rx <=128
udt_tx_size	(RW)	TX buffer size (packets [0,127]), tx+rx <=128
udt_shutd	(RW)	Active shut down UDT connection
udt_clock_en	(RW)	Enable clock for udt module
udt_enable	(RW)	Enable more stable udt transfer

Register 22.11: UDT_CFG1 (0x00000054)

31–0	
udt_socket	
2021	Reset

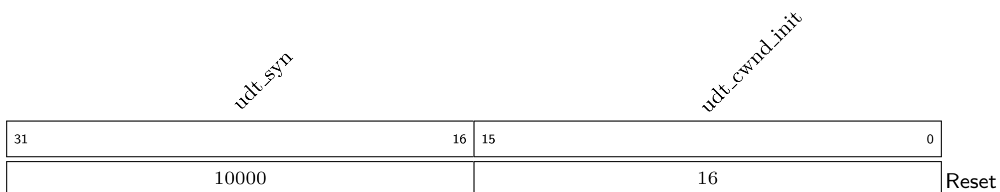
udt_socket (RW) udt server socket

Register 22.12: UDT_CFG2 (0x00000058)



udt_brake	(RW)	udt brake mode (800 mbps)
udt_snd_init	(RW)	udt init send period (200 mbps)

Register 22.13: UDT_CFG3 (0x0000005c)



udt_syn	(RW)	udt min sync period (10 ms) of ACK, NAK
udt_cwnd_init	(RW)	udt init congestion sending window (packets)

UDP2NOC General Control

The DNoC router is enabled by default. To disable the module bit zero of register 0x00000060 (udp_enable) must be set to zero. Register entry **??** controls how UDP datagrams are translated into NoC packets or vice versa. Register entry **??** controls if or how long the Hostif should wait for a NoC packet before sending out a udp frame. The LSB of entry **??** can be set to send out exactly one udp frame.

Register 22.14: UNOC_CTRL0 (0x00000060)

Bits	31–18	17	16–14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Field	og_mtu	og_burst_break	og_data_mode	og_auto_mode	og_def_src_port	og_def_dst_port_addr	og_modid_routing	og_type_routing	og_raw_order32b	og_frame_id_prot_en	ic_raw_32b_mode	ic_raw_save_addr	ic_raw_descend	ic_raw_order32b	ic_addr_inc_disable	ic_frame_id_prot_en	udp_enable
Reset	350	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1

og_mtu (RW) maximum transfer size for outgoing packets 32b

og_burst_break (RW) while waiting for noc packets during burst sendout the udp frame is send out not only if mtu reached but also if noc packet arrives with different noc header

og_data_mode (RW) 0- single noc packet / 1- burst with each noc header + addr / 2- burst with only one noc header + each addr per udp frame / 3- burst with only one noc header + addr per udp frame / 4- raw single 128b / 5- raw single 32b / 6- raw burst

og_auto_mode (RW) 0- outgoing packets sent out only after activated from outside package or og_trigger rf / 1- outgoing packets always sent out as burst

og_def_src_port (RW) 0- use last incoming destination ip address as source / 1- ip from outgoing LUT table (default)

og_def_dst_port_addr (RW) 0- use last incoming source ip address/port as destination / 1- port and addr from outgoing LUT table (default) -> 3rd priority

og_modid_routing (RW) 1- use ip addrress and port from outgoing LUT table (modid dependent) -> 2nd priority

og_type_routing (RW) 1- use ip addrress and port from outgoing LUT table (type dependent) -> 1st priority

og_raw_order32b (RW) 0- outgoing 128b payload are ordered 3/2/1/0 / 1- ordered 0/1/2/3

og_frame_id_prot_en (RW) enable protocol to send out warning if udp frame id sequence interrupted -> adds 32b header for frame id after magic (does not work in raw mode)

ic_raw_32b_mode (RW) 0- bundle 128b payload noc packets inside spinn2 and write 128b mem aligned / 1- 32b payload (unperformant but can avoid data fragmentation between udp packets)

ic_raw_save_addr (RW) 0- overwrite last raw udp packet in same addr / 1- increase/decrease ic_addr regfile between udp packet

ic_raw_descend (RW) 0- destination address increases per incoming raw packets / 1- addr decreases

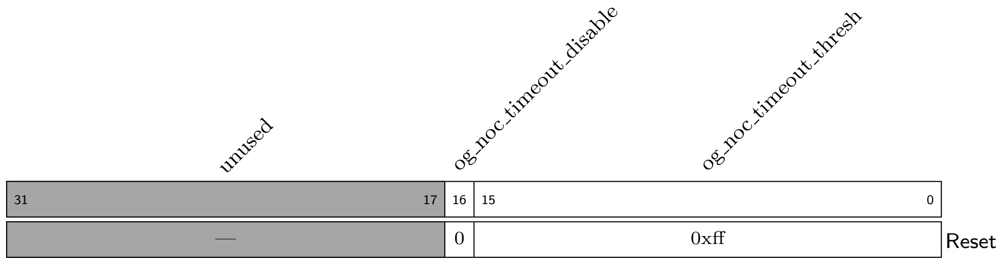
ic_raw_order32b (RW) 0- incoming 128b payload are ordered 3/2/1/0 / 1- ordered 0/1/2/3

ic_addr_inc_disable (RW) disable address increment/decrement for incoming noc & raw packets

ic_frame_id_prot_en (RW) enable protocol to send out warning if udp frame id sequence interrupted -> adds 32b header for frame id after magic (does not work in raw mode)

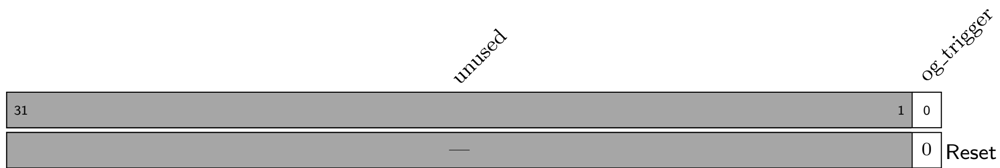
udp_enable (RW) enable udp_noc

Register 22.15: UNOC_CTRL1 (0x00000064)



og_noc_timeout_disable	(RW)	disable timer -> outgoing data waits for a noc packet
og_noc_timeout_thresh	(RW)	set the timer how long outgoing data should wait for a noc packet

Register 22.16: UNOC_OG_TRIGGER (0x00000068)

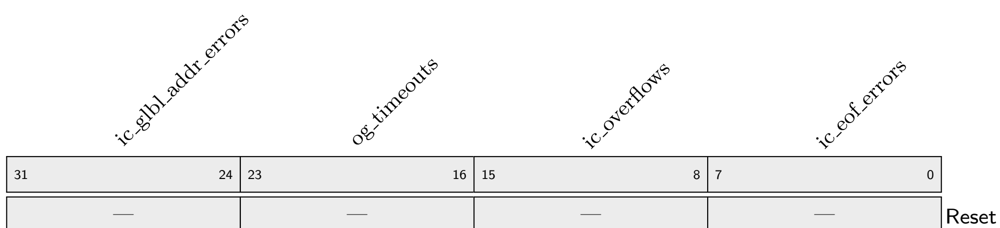


og_trigger	(RW)	1- trigger to send out noc packet/raw data/burst, only works if auto_mode == 0

UDP2NOC Status and Error

Register 22.17: UNOC_NOC_STATUS (0x0000006c)

Register 22.18: UNOC_ERRORS (0x00000070)

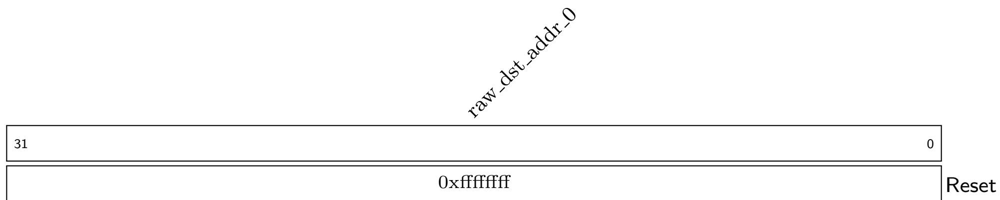


UDP2NOC Dynamic Incoming Ports

There are 16 ports for each noc and raw format. They are each 16bit in 8 regfile entries. NoC format ports are between address 0x00000074 and 0x00000090 . Raw format ports are between 0x00000074 and 0x00000094 .

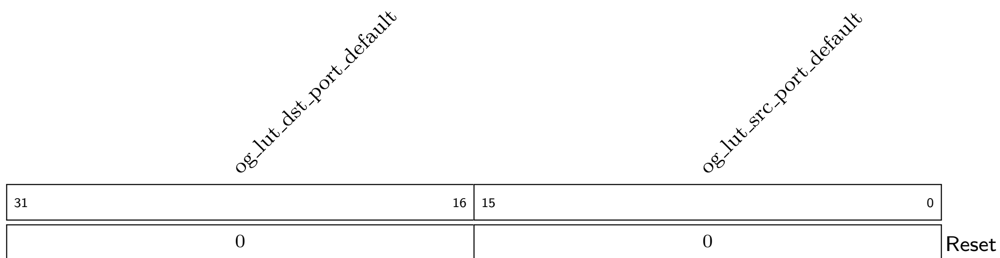
UDP2NOC Dynamic Incoming Raw Data Routing

Register 22.19: UNOC_IC_RAW_DST_ADDR_0 (0x000000b4)

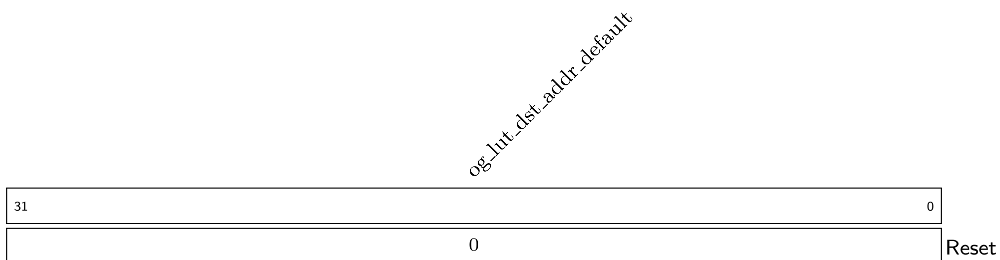


UDP2NOC Dynamic Outgoing Default Routing

Register 22.20: UNOC_OG_LUT_CFG_DEFAULT_PORTS (0x000000f4)

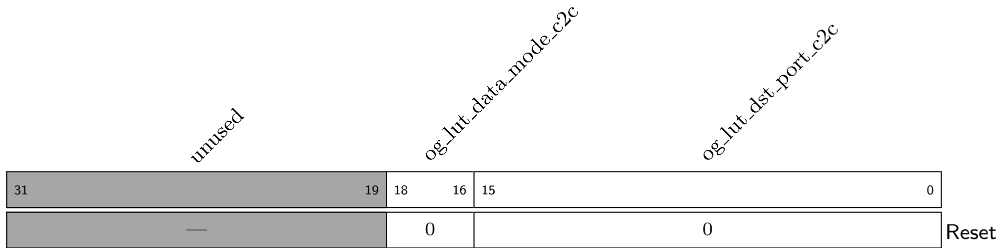


Register 22.21: UNOC_OG_LUT_CFG_DEFAULT_ADDR (0x000000f8)

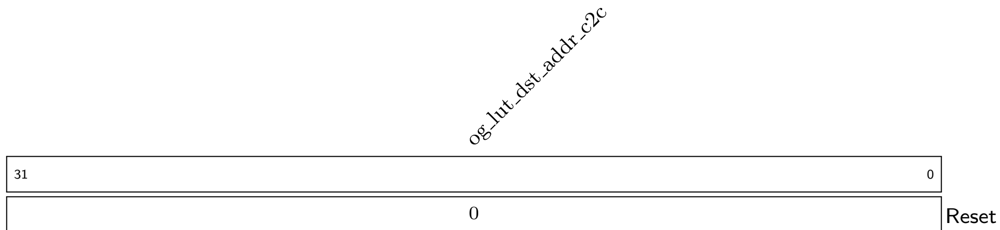


UDP2NOC Dynamic Outgoing Type Routing

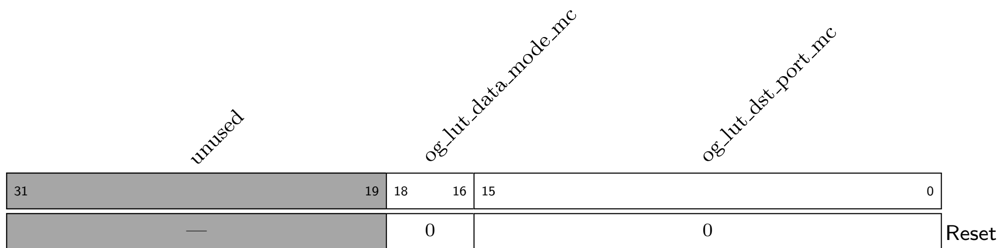
Register 22.22: UNOC_OG_LUT_C2C_CFG (0x000000fc)



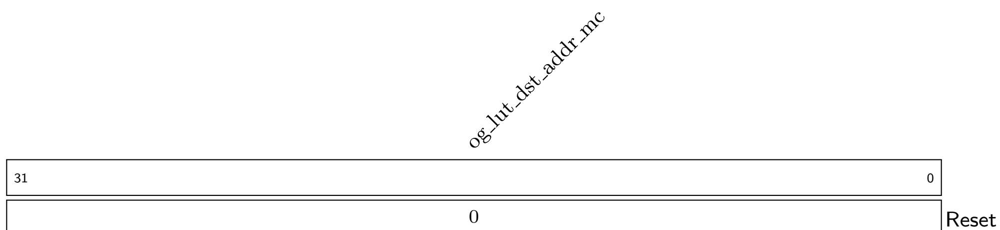
Register 22.23: UNOC_OG_LUT_C2C_ADDR (0x00000100)



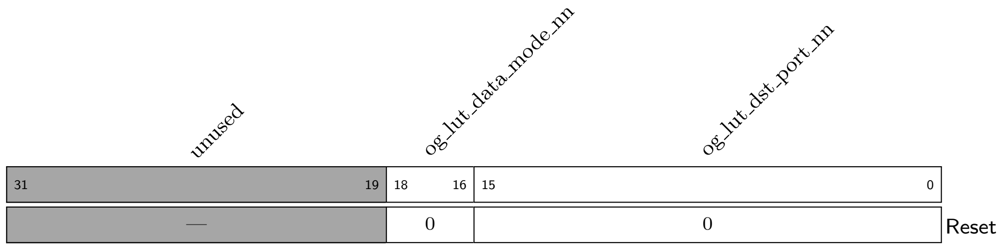
Register 22.24: UNOC_OG_LUT_MC_CFG (0x00000104)



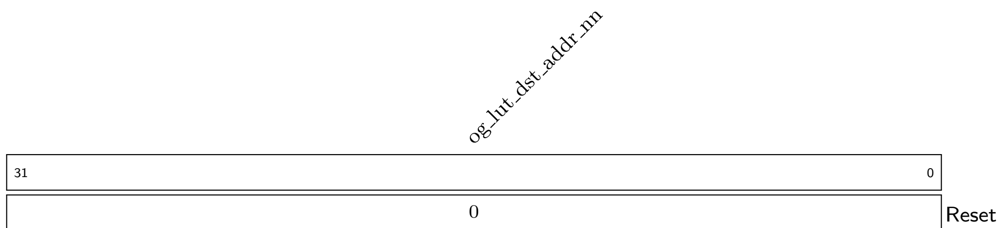
Register 22.25: UNOC_OG_LUT_MC_ADDR (0x00000108)



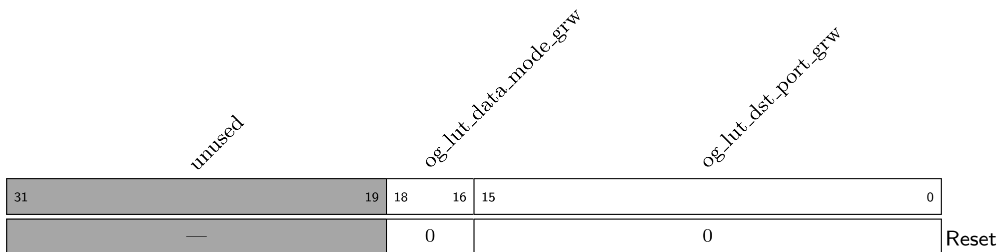
Register 22.26: UNOC_OG_LUT_NN_CFG (0x0000010c)



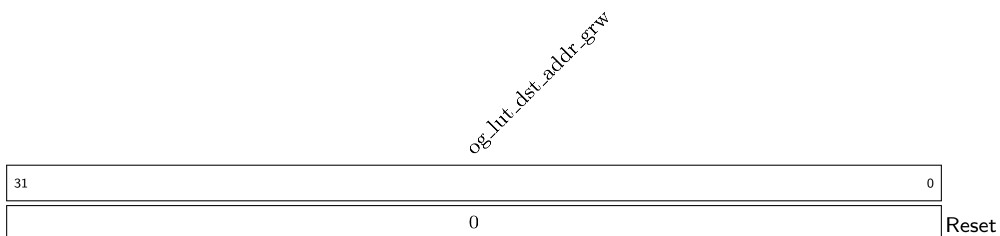
Register 22.27: UNOC_OG_LUT_NN_ADDR (0x00000110)



Register 22.28: UNOC_OG_LUT_GRW_CFG (0x00000114)



Register 22.29: UNOC_OG_LUT_GRW_ADDR (0x00000118)



Register 22.30: UNOC_OG_LUT_C2C_CH_0 (0x0000011c)

unused	og_lut_c2c_data_mode	og_lut_c2c_dst_port	og_lut_c2c_ch_id	
31 27	26 24	23 8	7 0	
—	0	20000	0	Reset

Register 22.31: UNOC_OG_LUT_C2C_CH_ADDR_0 (0x00000120)

og_lut_dst_addr	
31 0	
3232235650	Reset

UDP2NOC Dynamic Outgoing Modid Routing

Register 22.32: UNOC_OG_LUT_MODID_0 (0x0000019c)

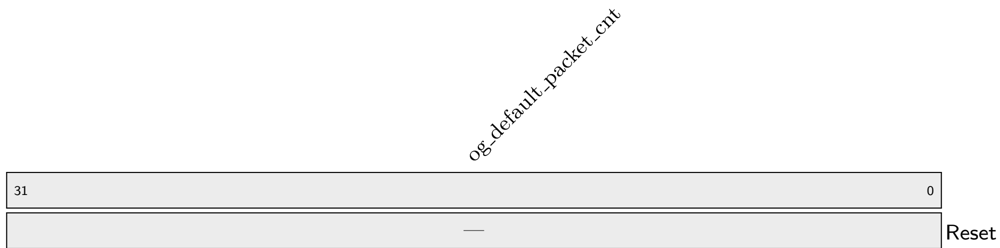
unused	og_lut_data_mode	og_lut_dst_port	og_lut_src_modid	
31 27	26 24	23 8	7 0	
—	0	30000	0	Reset

Register 22.33: UNOC_OG_LUT_MODID_ADDR_0 (0x000001a0)

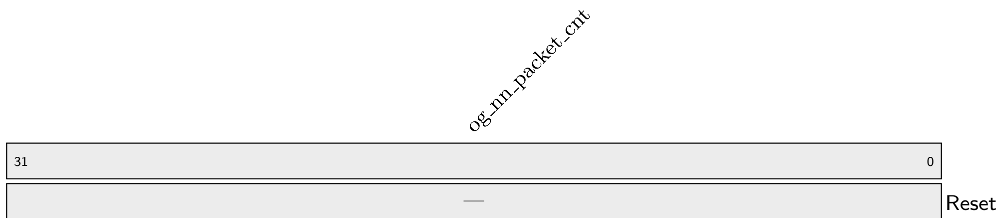
og_lut_dst_addr	
31 0	
3232235750	Reset

UDP2NOC Packet Counter

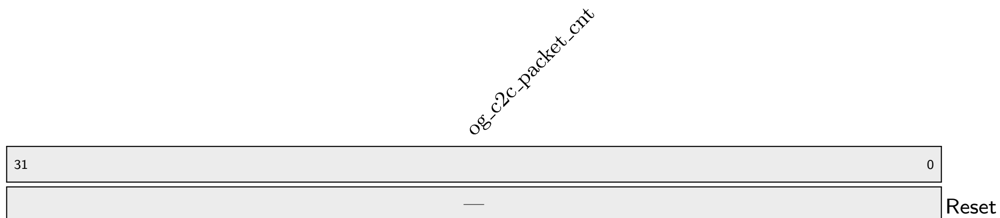
Register 22.34: UNOC_OG_DEFAULT_PACKET_CNT_16 (0x0000021c)



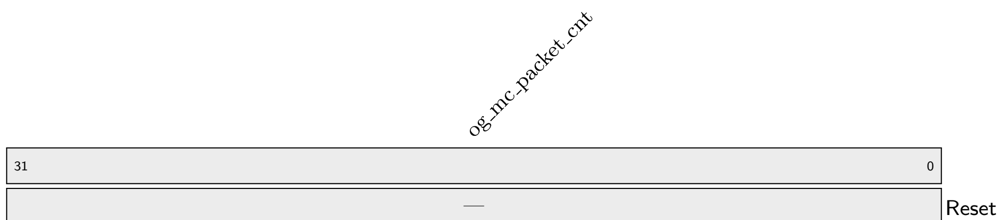
Register 22.35: UNOC_OG_NN_PACKET_CNT_16 (0x00000220)



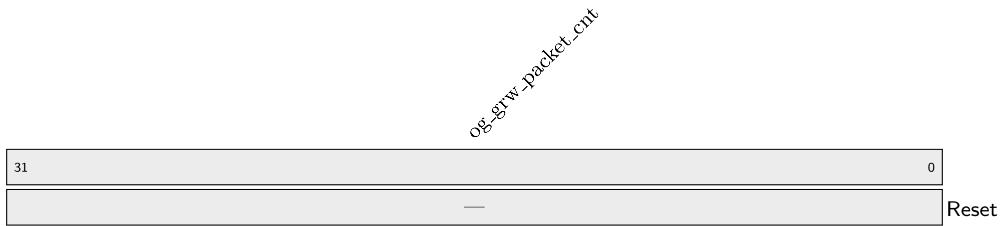
Register 22.36: UNOC_OG_C2C_PACKET_CNT_16 (0x00000224)



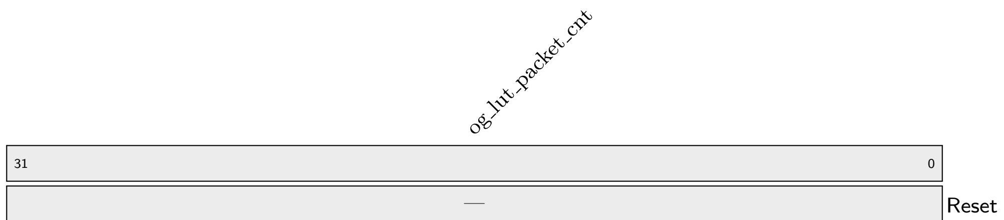
Register 22.37: UNOC_OG_MC_PACKET_CNT_16 (0x00000228)



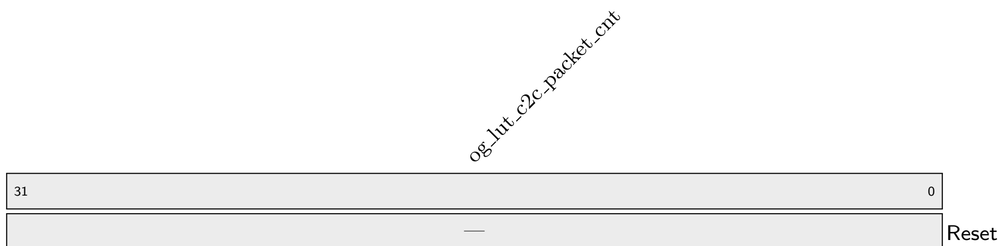
Register 22.38: UNOC_OG_GRW_PACKET_CNT_16 (0x0000022c)



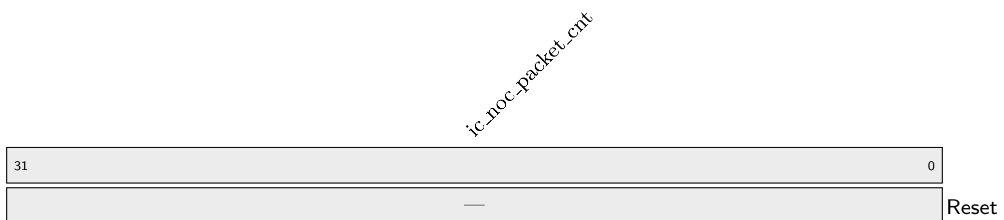
Register 22.39: UNOC_OG_LUT_PACKET_CNT_0 (0x00000230)



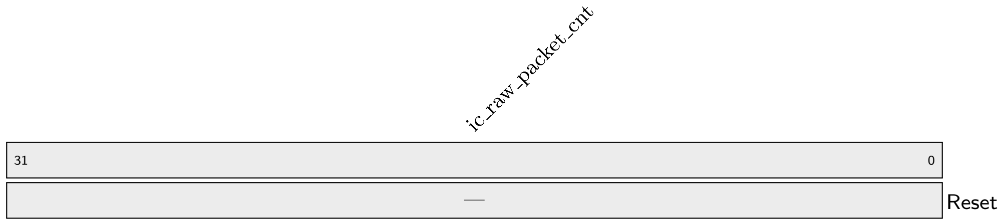
Register 22.40: UNOC_OG_PACKET_CNT_0 (0x000002b0)



Register 22.41: UNOC_IC_NOC_PACKET_CNT_0 (0x000002f0)



Register 22.42: UNOC_IC_RAW_PACKET_CNT_0 (0x00000330)



22.4 GPIO MUX

The GPIO multiplexer allows to map internal interfaces e.g. UART to a GPIO pads of HostIF. Figure 29 shows the basic structure. Each individual pad has up to 5 different functions.

pad	func0	func1	func2	func3
20	RF	CUART1_TX	PWM1_0	SPIS1_NSS
21	RF	CUART1_RX	PWM1_1	SPIS1_SCLK
22	RF	I2CM1_SDA	PWM1_2_CH0	SPIS1_IO0
23	RF	I2CM1_SCL	PWM1_2_CH1	SPIS1_IO1
24	RF	unused	PWM1_2_CH2	SPIS1_IO2
25	RF	SPIS1_IRQ0_N	PWM1_2_CH3	SPIS1_IO3
26	RF	SPIS1_IRQ1_N	FBIST_O	unused

pad	func4	func5	func6
20	UART1_0_TX	MDC	SPIM1_NSS
21	UART1_0_RX	MDIO__DATA	SPIM1_SCLK
22	unused	UART1_1_TX	SPIM1_IO0
23	unused	UART1_1_RX	SPIM1_IO1
24	UART1_0_RTR	UART1_1_RTR	SPIM1_IO2
25	UART1_0_CTS	UART1_1_CTS	SPIM1_IO3
26	unused	RMII_IRPT	unused

22.5 Register summary

22.6 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

23 System Controller

The System Controller incorporates a number of functions for system start-up, fault-tolerance testing (invoking, detecting and resetting faults), general performance monitoring, etc.

23.1 Features

1. 'Arbiter' read-sensitive register bit to determine Monitor Processor ID at start-up.
2. 32 test-and-set registers for general software use, e.g. to enforce mutually exclusive access to critical data structures.
3. individual interrupt, reset and enable controls and 'processor OK' status bits for each processor.
4. sundry parallel IO and test and control registers.
5. PLL and clock management registers.

TO BE DONE: which of the above do we need?

23.2 Register summary

23.3 Fault-tolerance

Fault insertion

1. TO BE DONE

Fault detection

1. TO BE DONE

Fault isolation

1. TO BE DONE

Reconfiguration

1. TO BE DONE

24 Watchdog timer

The watchdog timer is an ARM PrimeCell component (ARM part SP805, documented in ARM DDI 0270B) that is responsible for applying a system reset when a failure condition is detected. Normally, the Monitor Processor will be responsible for resetting the watchdog periodically to indicate that all is well. If the Monitor Processor should crash, or fail to reset the watchdog during a pre-determined period of time, the watchdog will trigger.

24.1 Features

1. generates an interrupt request after a programmable time period;
2. causes a chip-level reset if the Monitor Processor does not respond to an interrupt request within a subsequent time period of the same length.

24.2 Register summary

Base address: 0xe3000000 (buffered write), 0xf3000000 (unbuffered write).

User registers

The following registers allow normal user programming of the Watchdog timer:

Name	Offset	R/W	Function
r0: WdogLoad	0x00	R/W	Count load register
r1: WdogValue	0x04	R	Current count value
r2: WdogControl	0x08	R/W	Control register
r3: WdogIntClr	0x0C	W	Interrupt clear register
r4: WdogRIS	0x10	R	Raw interrupt status register
r5: WdogMIS	0x14	R	Masked interrupt status register
r6: WdogLock	0xC00	R/W	Lock register

Test and ID registers

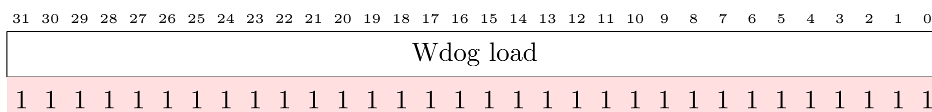
In addition, there are test and ID registers that will not normally be of interest to the programmer:

Name	Offset	R/W	Function
WdogITCR	0xF00	R/W	Watchdog integration test control register
WdogITOP	0xF04	W	Watchdog integration test output set register
WdogPeriphID0-3	0xFE0-C	R	Watchdog peripheral ID byte registers
WdogPCID0-3	0xFF0-C	R	Watchdog Prime Cell ID byte registers

See AMBA Design Kit Technical Reference Manual ARM DDI 0243A, February 2003, for further details of the test and ID registers.

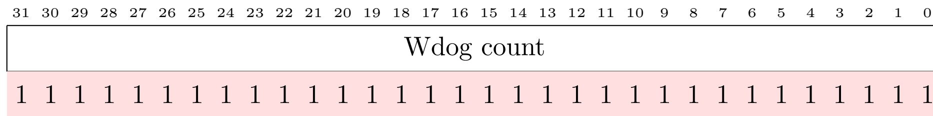
24.3 Register details

r0: Load



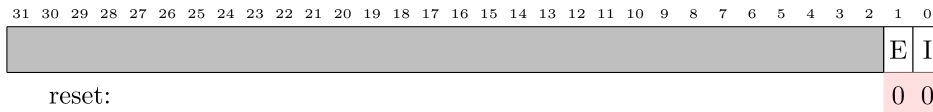
This read-write register contains the value the from which the counter is to decrement. When this register is written to, the count immediately restarts from the new value. The minimum value is 1.

r1: Count



This read-only register contains the current value of the decrementing counter. The first time the counter decrements to zero the Watchdog raises an interrupt. If the interrupt is still active the second time the counter decrements to zero the reset output is activated.

r2: Control

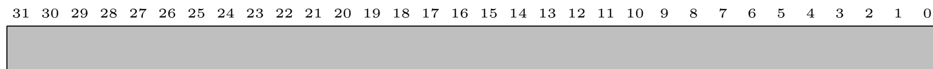


The functions of these fields are described in the table below:

Name	Offset	R/W	Function
E	1	R/W	Enable the Watchdog reset output (1)
I	0	R/W	Enable Watchdog counter and interrupt (1)

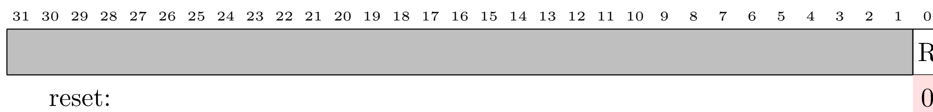
Once the Watchdog has been initialised both enables should be set to '1' for normal watchdog operation.

r3: Interrupt clear



A write of any value to this register clears the watchdog interrupt and reloads the counter from r1.

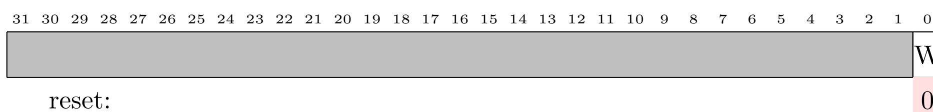
r4: Raw interrupt status



The function of this field is described in the table below:

Name	Offset	R/W	Function
R	0	R	Raw (unmasked) watchdog interrupt

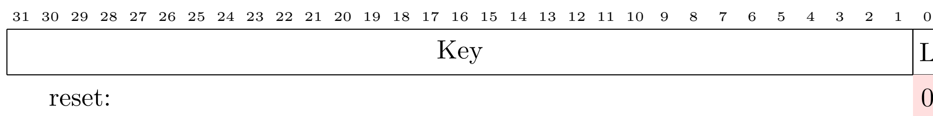
r5: Masked interrupt status



The function of this field is described in the table below:

Name	Offset	R/W	Function
W	0	R	Watchdog interrupt output

r6: Lock



The functions of these fields are described in the table below:

Name	Offset	R/W	Function
Key	31:0	W	Write 0x1ACCE551 to enable writes
L	0	R	Write access enabled (0) or disabled (1)

A read from this register returns only the bottom bit, indicating whether writes to other registers are enabled (0) or disabled (1). A write of 0x1ACCE551 enables write access to the other registers; a write of any other value disables write access to the other registers. Note that the 'Key' field is 32 bits and includes bit 0.

The lock function is available to ensure that the watchdog will not be reset by errant programs.

25 Power Management Architecture

As shown in Fig. 42, allows for switching between two VDD rails (PL1 and PL2 in Fig. ??) during operation. To enable ulra-low voltage operation, the PE domain is adaptively body biased. Adaptive body biasing (ABB) is a technique for FDSOI technologies [?] for the compensation of device performance variations caused by process, voltage and temperature (PVT) variations by means of the adaptive control of the back-gate voltages [?]. Considering energy per operation metric, there exists a minimum energy point (MEP) at nominal 0.50V operation. At higher voltages, more energy per operation is spent due to higher switching energy. At lower voltages more energy per operation is accumulated due to leakage power over the longer clock period. As result, the target implementation point has been chose at 0.50V nominal, at 150MHz sign-off frequency. However, this does not denote a significant performance scaling compared to the first generation SpiNNaker processor [?]. Therefore, the DVFS technique from [?, ?] is applied here, with two performance levels (PLs). PL1 is the MEP operating point of (0.50V, 150MHz) and PL2 is defined as the higher performance level at (0.80V, 300MHz).

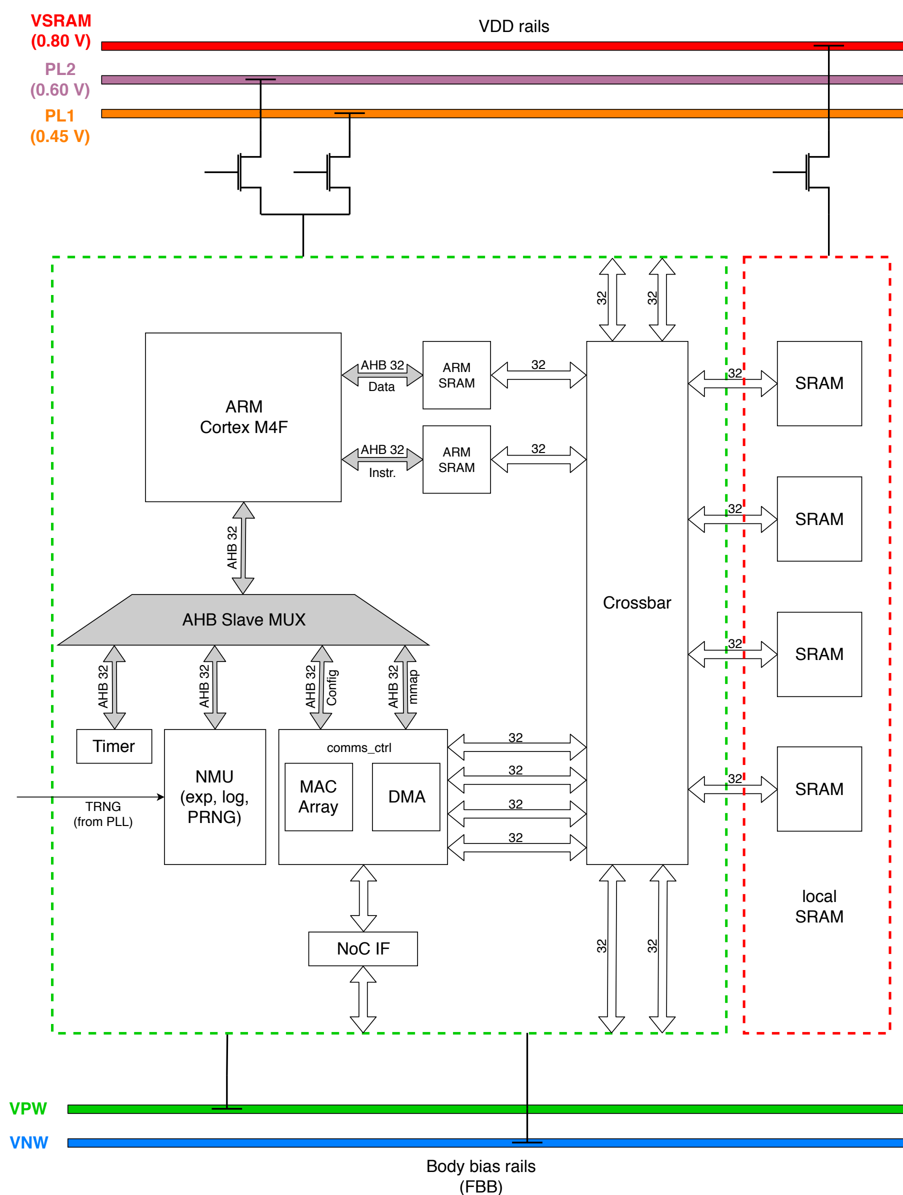


Figure 42: PE power management architecture

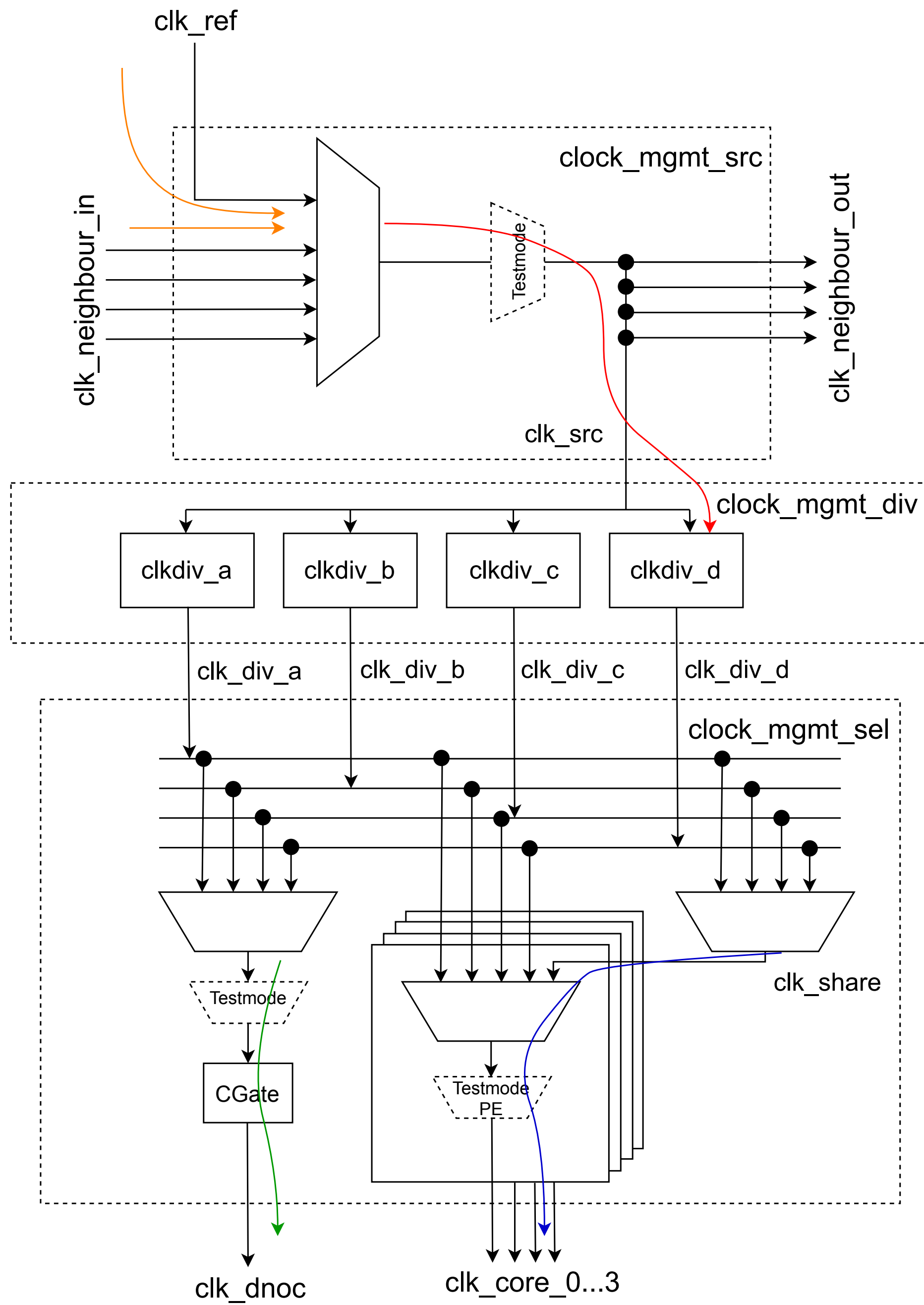


Figure 43: QPE Clocking

A Packaging

B Input and Output signals

TO BE DONE

C Electrical Specification

C.1 Operating Temperature

name	unit	min	typ	max	comment
T	C	0		85	

C.2 Power Supply

name	unit	min	typ	max	comment
VDDPE	V	0.475	0.500	0.525	core supply for PEs
VDD	V	0.760	0.800	0.840	core supply
VDDIO	V	1.620	1.800	1.980	IO supply
VDDSER	V	0.475	0.500	0.525	C2C Link supply
VDDQ	V		1.100		LPDDR4 interface supply
VDDPLL	V	0.760	0.800	0.840	PLL analog supply

C.3 Current consumption

supply domain power	unit	min	typ	max	comment
P(VDDPE)	W			0.654	core supply for PEs
P(VDD)	W			4.963	core supply
P(VDDIO)	W			0.130	IO supply
P(VDDSER)	W			0.180	C2C Link supply
P(VDDQ)	W			0.140	LPDDR4 interface supply
P(VDDPLL)	W			0.050	PLL analog supply

D Application Note

D.1 External Components

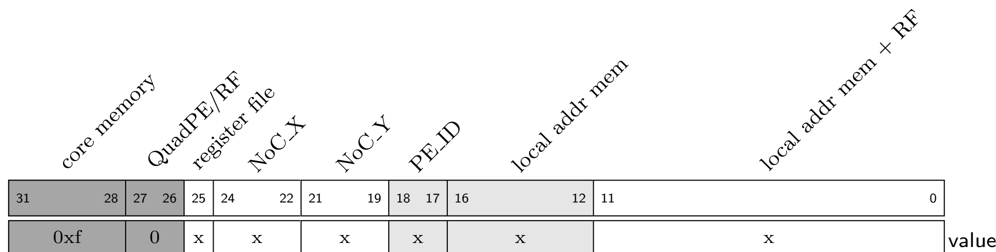
D.2 PCB Integration Guideline

E SpiNNaker2 Address Map and Register Summary

E.1 Core memories

Main configuration register and QuadPE register address space is located from s 0xf000 0000 to 0xf3ffffff. The particular memory address can be determined from the module NoC X and Y coordinate (see Figure 2):

Register E.1: REGISTER MEMORY MAP

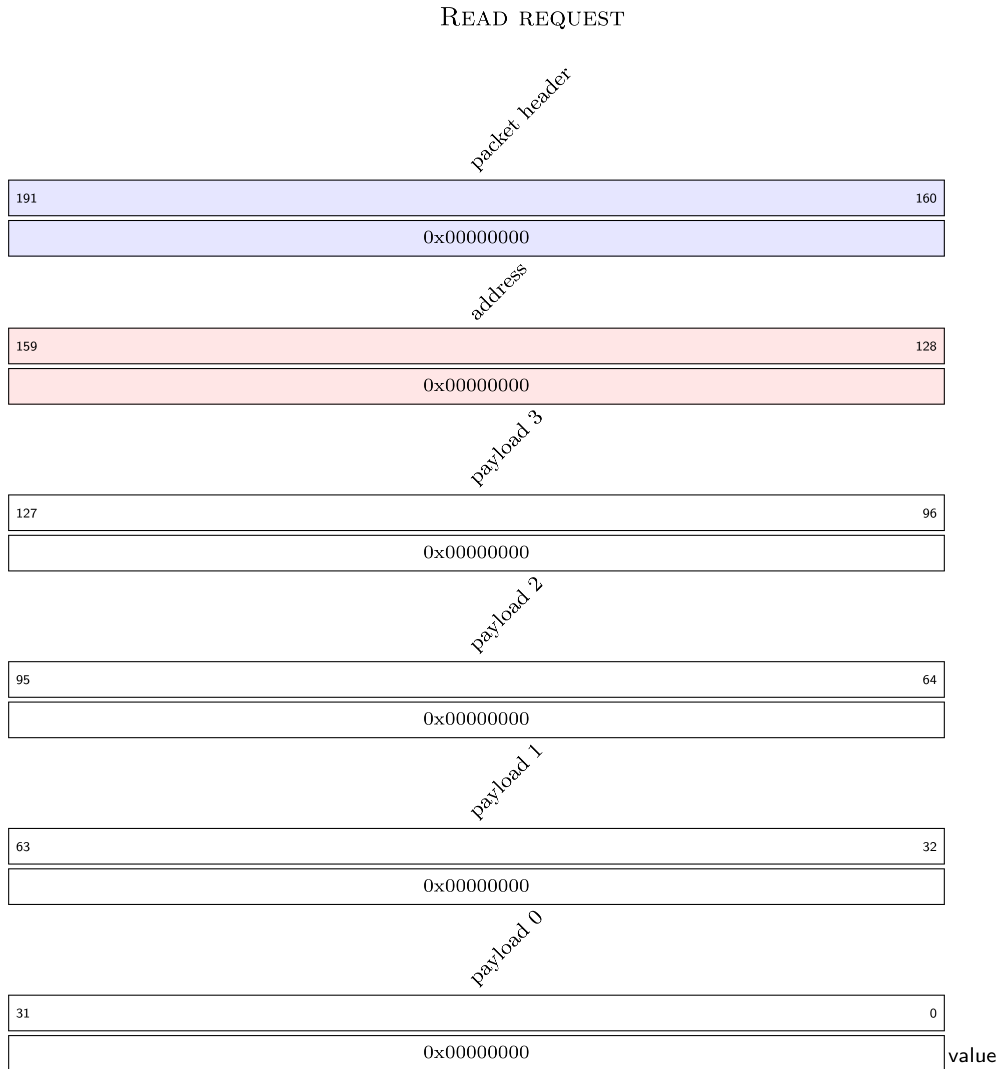


Accessing register files is done by setting address bit 25 to one and NoC_X + NoC_Y of the desired module. The local register file address is determined by the lowest 12 bit of the address. Bits 12 to 18 are ignored. The QuadPE memory can be accessed similarly by setting bit 20 to zero and X+Y value. The PE (0 to 3) is set with bits 17+18. The local memory address is set with the lowest 17 bits.

NoC_X	NoC_Y	Ref	Function
0	0	[21]	Periphery components
1-2	0	-	unused
1-7	1-7	[4]	Quad PE
4	4	[18]	SpiNNaker Router ID0
4	5	[18]	SpiNNaker Router ID1
0	1	[??]	LPDDR4 IF0 low
0	2	[??]	LPDDR4 IF0 high
0	5	[??]	LPDDR4 IF1 low
0	6	[??]	LPDDR4 IF1 high
5	7	[22]	HostIF right
6	7	[22]	HostIF left
3	0	[20]	Spike SerDesH 0 left
4	0	[20]	Spike SerDesH 0 right
5	0	[20]	Spike SerDesH 1 left
6	0	[20]	Spike SerDesH 1 right
2	7	[20]	Spike SerDesH 2 left
1	7	[20]	Spike SerDesH 2 right
4	7	[20]	Spike SerDesH 3 left
3	7	[20]	Spike SerDesH 3 right
0	4	[20]	Spike SerDesV 0 left
0	3	[20]	Spike SerDesV 0 right
7	3	[20]	Spike SerDesV 1 left
7	4	[20]	Spike SerDesV 1 right
x	y	-	TODO

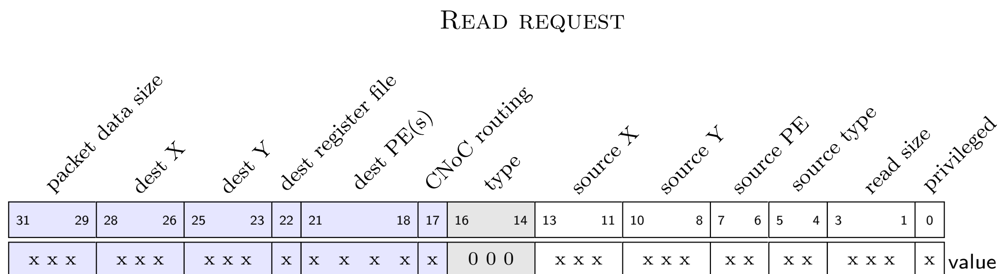
F NoC packet formats and header definitions

F.1 NoC packet format



F.2 NoC header definitions

The NoC packet header formats are summarised as follows:



Read response

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	source X	source Y	source PE	source type	unused	bus error
bits	31 29	28 26	25 23	22	21 18	17	16 14	13 11	10 8	7 6	5 4	3 1	0
value	x x x	x x x	x x x	x	x x x x	x	0 0 1	x x x	x x x	x x	x x		x

Write request

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	source X	source Y	source PE	source type	unused	buffered write	privileged access
bits	31 29	28 26	25 23	22	21 18	17	16 14	13 11	10 8	7 6	5 4	3 2	1	0
value	x x x	x x x	x x x	x	x x x x	x	0 1 0	x x x	x x x	x x	x x		x	x

Write response

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	source X	source Y	source PE	source type	unused	bus error
bits	31 29	28 26	25 23	22	21 18	17	16 14	13 11	10 8	7 6	5 4	3 1	0
value	x x x	x x x	x x x	x	x x x x	x	0 1 1	x x x	x x x	x x	x x		x

Control and exception command

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	source X	source Y	source PE	source type	unused	buffered write	privileged access
bits	31 29	28 26	25 23	22	21 18	17	16 14	13 11	10 8	7 6	5 4	3 2	1	0
value	x x x	x x x	x x x	x	x x x x	x	1 0 0	x x x	x x x	x x	x x		x	x

Protocol message

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	source X	source Y	source PE	source type	unused	privileged access
bits	31 29	28 26	25 23	22	21 18	17	16 14	13 11	10 8	7 6	5 4	3 1	0
value	x x x	x x x	x x x	x	x x x x	x	1 0 1	x x x	x x x	x x	x x		x

SpiNNaker: Multicast (MC)

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	unused	SpiNNaker packet type	software defined	time stamp	spinnaker packet size
bits	31–29	28–26	25–23	22	21–18	17	16–14	13–11	7–6	5–4	3–2	1–0
value	x x x	x x x	x x x	x	x x x x	x	1 1 1		00	x x	x x	x x

SpiNNaker: Core to Core (C2C)

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	unused	SpiNNaker packet type	software defined	time stamp	spinnaker packet size
bits	31–29	28–26	25–23	22	21–18	17	16–14	13–11	7–6	5–4	3–2	1–0
value	x x x	x x x	x x x	x	x x x x	x	1 1 1		01	x x	x x	x x

SpiNNaker: Nearest Neighbour (NN)

	packet data size	dest X	dest Y	dest register file	dest PE(s)	CNoC routing	type	unused	SpiNNaker packet type	?	route	spinnaker packet size
bits	31–29	28–26	25–23	22	21–18	17	16–14	13–11	7–6	5	4–2	1–0
value	x x x	x x x	x x x	x	x x x x	x	1 1 1		10	?	x x x	x x

Name	Function
packet data size	Payload size
Dest X	Notional NoC X position of destination QPE
Dest Y	Notional NoC Y position of destination QPE
Dest register	destination is register file within QPE
Dest PE	One-hot encoded destination PE(s) within QPE
CNoC routing	Use CNoC (=1) instread of DNoC (=0)
Source X	Notional NoC X position of source QPE
Source Y	Notional NoC Y position of source QPE
Source PE	Source PE identifier within QPE (0 to 3)
Source Type	Source unit type within PE (i.e. subsystem ID)
Read Size	Requested data size
Bus Error	Bus Error response
Buffered Write	buffer write operation (no response packet)
Privileged	Privileged / Supervisor bus cycle
Link	Off-chip link code
Software Defined	Software defined field
Time Stamp	time stamp of SpiNNaker packet
Route	NOT SURE!
SpiNNaker Size	Payload size in SpiNNaker packet (duplicate info.)

Packet Data Size is used primarily to determine how much of the potential packet is actually delivered.

Data Size]	Payload	'Size' equivalent
0 0 0	0 (no payload)	0 0
0 0 1	1 byte	0 1
0 1 0	2 bytes	0 1
0 1 1	4 bytes	0 1
1 0 0	8 bytes	1 0
1 0 1	16 bytes	1 1
1 1 0	—	—
1 1 1	—	—

Read Size is used primarily to determine how many and how 'full' the returned data packets will be.

Read Size	Returned bytes	Returned 'Pkt Sz'
0 0 0	1	0 0 1
0 0 1	2	0 1 0
0 1 0	4	0 1 1
0 1 1	8	1 0 0
1 0 0	16	1 0 1
1 0 1	32	1 0 1
1 1 0	64	1 0 1
1 1 1	128	1 0 1

Dest Register If 'R' is set the packet is addressed to the NoC registers at the upper QPE level.

Dest PE Is a multicast code. The numbered bits indicate which PE(s) the packet isdelivered to, a '1' indicating tht PE is included. From any one to all PEs may be indicated.

3	2	1	0
0	1	2	3

Source Type Identifies a source/sink within a PE.

Source Type	Source
0 0	M4 bus (bridge)
0 1	Tx/Rx (software)
1 0	DMA
1 1	ML-accelerator

route Something to do with NN packets.

F.3 NoC packet formats

Fig. 45 illustrates the SpiNNaker2 NoC packet formats.

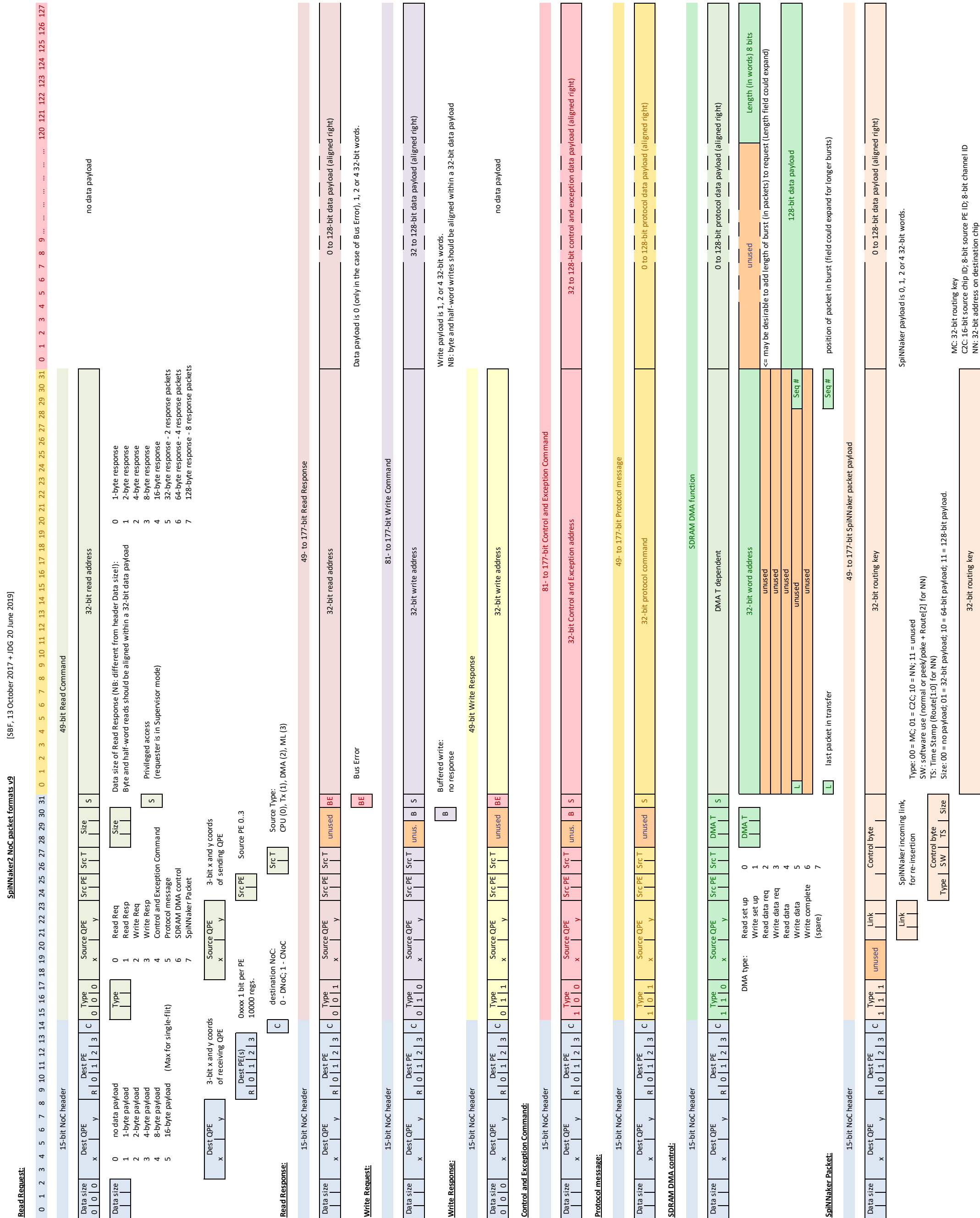


Figure 44: SpiNNaker2 NoC packet format

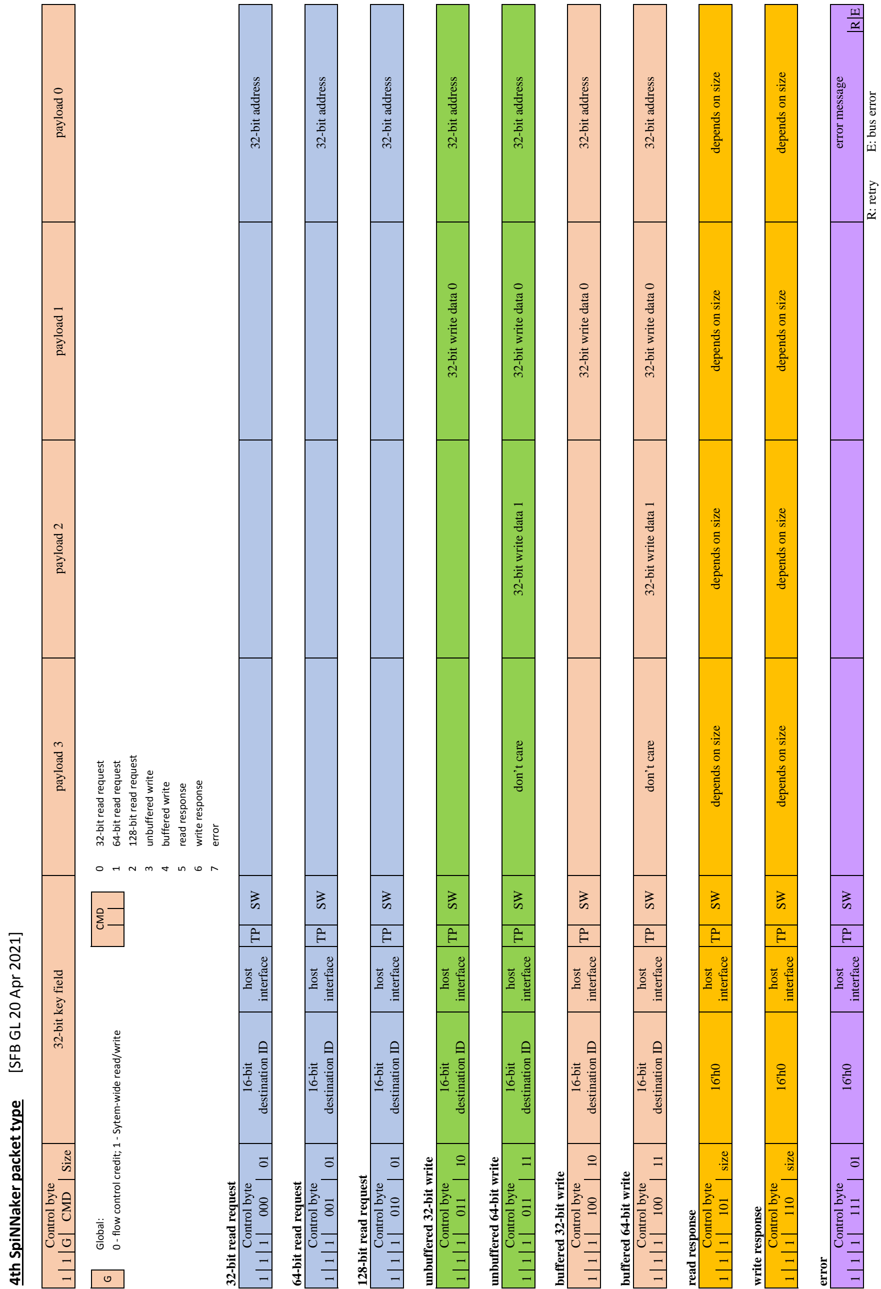


Figure 45: SpiNNaker2 NoC packet format