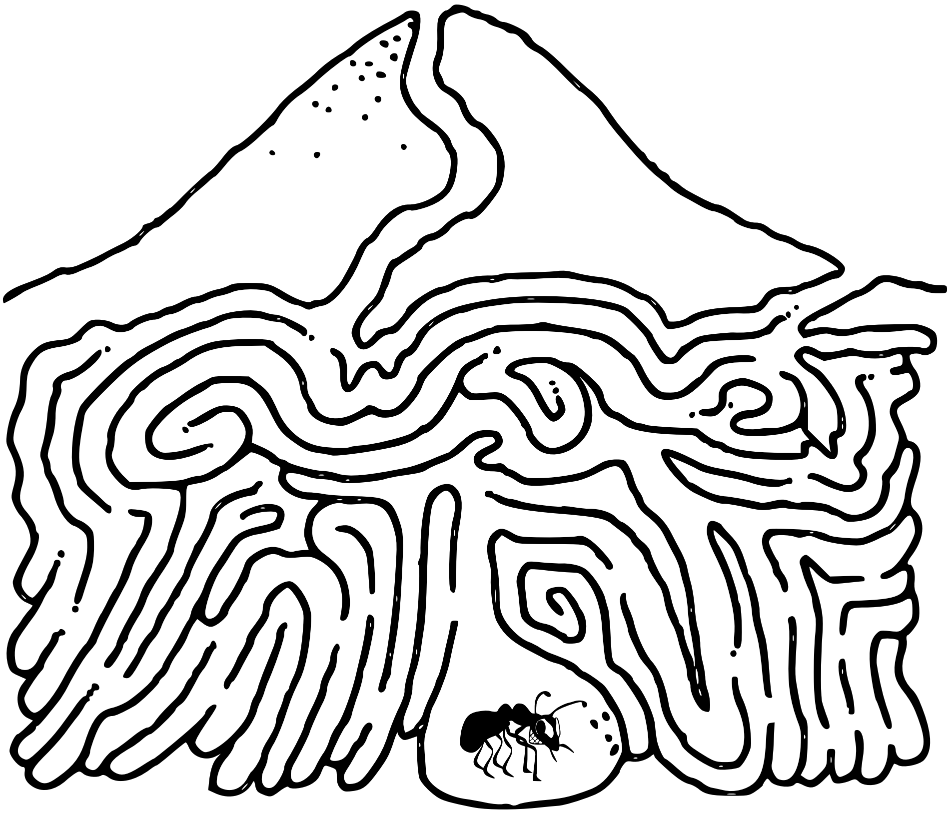


The 512-core Formic Hardware Prototype

Architecture Manual & Programmer's Model

FORTH-ICS / TR-430, June 2012

Spyros Lyberis
George Kalokerinos

Foundation for Research & Technology – Hellas (FORTH)
Institute of Computer Science (ICS)
Computer Architecture & VLSI Systems Laboratory (CARV)

Abstract

Manycore FPGA prototypes are difficult to build, but rewarding: they offer very fast running times and provide significant architectural insight during the hardware modeling process. At a time where hundreds of cores may become feasible in single-chip processors in the near future, research on parallel software and hardware can benefit greatly from manycore FPGA prototypes. Existing FPGA prototyping boards are limited for multi-board setups — they usually do not offer SRAM memory to model caches, have insufficient inter-board connectivity to scale the design, or they simply cost too much — we have designed the *Formic board,* a new FPGA board which specifically targets manycore architecture prototyping[1]. In this technical report, we describe how we develop a 64-board system that implements a 512-core, MicroBlaze-based, non-coherent hardware prototype with a full network-on-chip in a 3D-mesh topology.

Acknowledgements

The research leading to these results has received funding from the European Union 7^{th} Framework Programme [FP7/2007-2013], under the ENCORE (grant agreement n^{o} 248647) and TEXT (n^{o} 261580) Projects. We would like to thank Xilinx for the donation of 64 Spartan-6 FPGA devices.

[1] http://formic-board.com

Contents

Chapter 1

System Description

1.1 Block Diagrams

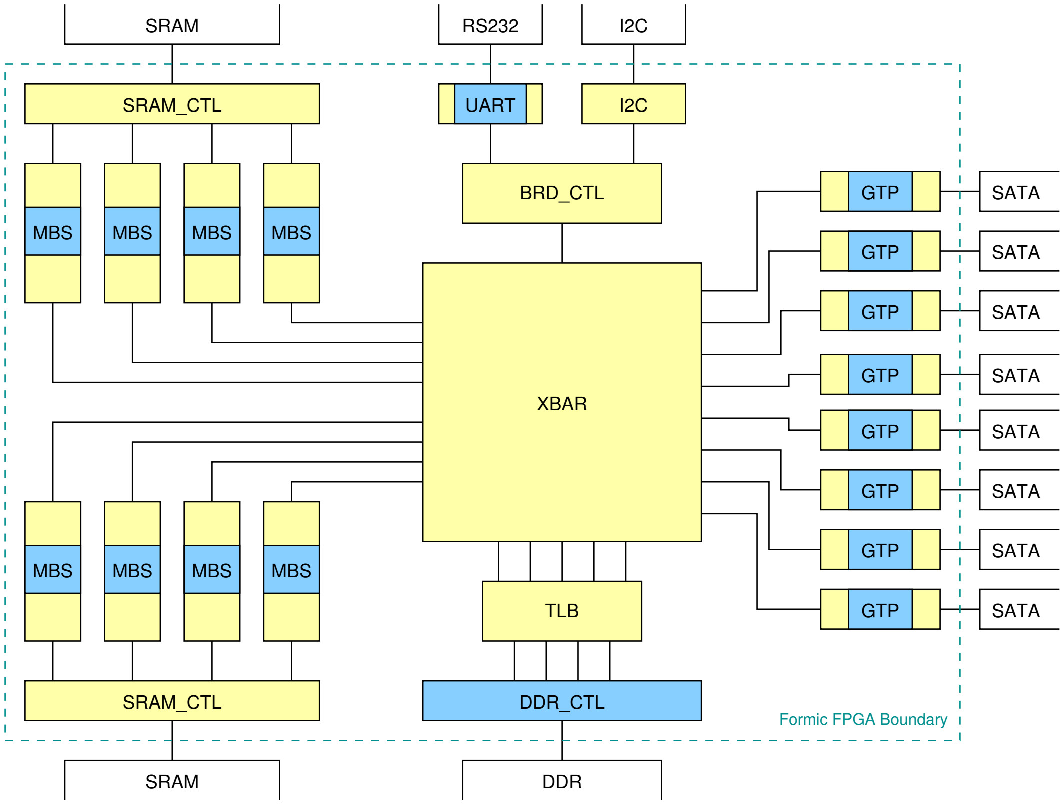


Figure 1.1: Formic top-level Block Diagram. Blue color indicates usage of Xilinx IP.

The 512-core hardware prototype is based on the the FORTH-ICS Formic board. Figure 1.1 presents the top-level diagram for the Formic boards FPGA. There are eight Microblaze Slice blocks (MBS, detailed in chapter 2, page 5) each contain a MicroBlaze processor, its private L1 and L2 cache hierarchy, counters, mailbox and network engines. Eight serial GTP links provide communication to other boards. A 5-port TLB provides virtual-to-physical translation and access to the board DDR2 DRAM and a board controller provides board configuration and access to the UART and I^{2}C ports. All the blocks are interconnected with a 22-port crossbar.

1.2 Formic FPGA Resources Budgeting

1.2.1 Spartan-6 LX150T facts

- 92,152 LUTs
- 184,304 flip-flops
- 268 BRAMs of 16 Kbits (without parity) or 18 Kbits (with parity).
- 8 GTP links
- 6 PLLs, 12 DCMs

1.2.2 Microblaze and Alternative Cores resources

Microblaze

Absolute maximum	6,900 LUTs
FPU, MMU, BTC, no L1	3,900 LUTs
FPU, no MMU, D+I L1 caches (8+16)	2,800 LUTs
FPU, no MMU, BTC, no L1	2,400 LUTs
FPU, no MMU, no L1, area opt (3-stage), trace/debug	**2,194 LUTs**
FPU, no MMU, no L1	2,100 LUTs
FPU, no MMU, no L1, area opt (3-stage)	1,800 LUTs
No FPU, no MMU, no L1, area opt (3-stage), no int div	1,000 LUTs
Absolute minimum	860 LUTs

Alternative Cores

LEON3: Minimum 3,000 LUTs (No FPU, no mul/div, no BP, no MMU, no caches) to average 4,000 LUTs (FPU, BP, int mul/div, no MMU, no caches). Open-source, VHDL, mature, many configuration options.

OpenRISC: Standard profile of 4,000 LUTs (no MMU, no FPU, no caches). Open-source, Verilog, not very mature, few configuration options.

1.2.3 FPGA Budget

Block or System module	LUTs	BRAMs	Instances	tot. LUTs	tot. BRAMs
CPU	2,194	0			
ART	224	0			
IL1	124	3			
DL1	212	5			
L2C	1,150	7			
MNI	1,316	1			
CTL	774	0			
CMX	385	3			
XBI	269	2			
MBS Total	6,698	21	8	53,584	168
SRAM_CTL	304	0	2	608	0
DDR_CTL	468	0	1	468	0
XBAR	10,824	0	1	10,824	0
GTP (quad block)	2,255	8	2	4,510	16
TLB	3,716	12	1	3,716	12
BRD_CTL	2,610	8	1	2,610	8
UART	179	1	1	179	1
Total				**76,499**	**205**

Table 1.1: FPGA Resource Budget

Chapter 2

Formic MBS Blocks Description

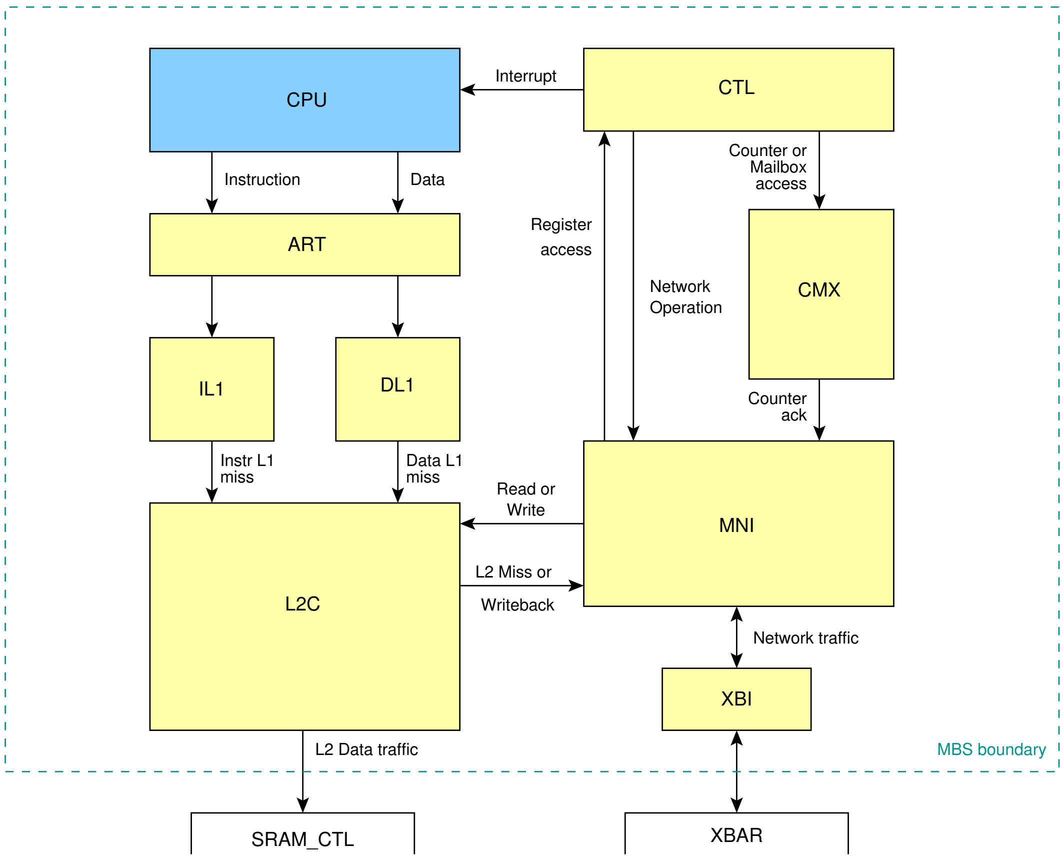


Figure 2.1: Microblaze Slice (MBS) Block Diagram. Blue color indicates usage of Xilinx IP.

2.1 Address Region Table block (ART)

2.1.1 Purpose

The ART block is responsible for the following actions:

- Intercepts CPU instruction and data accesses to the L1 caches
- Based on a table of address regions (kept by the CTL block), the access type (instruction or data) and whether the CPU is in privileged mode, it judges if the CPU accesses are legal or not
- Provides the cacheable (C) flag and the I/O access flag (I) to the L1 caches

2.1.2 Pin List

Pin Name	Width	Description
clk_cpu	1	CPU clock (10 MHz)
clk_mc	1	ART & Caches clock (40 MHz)
rst_cpu	1	Reset for clk_cpu
rst_mc	1	Reset for clk_mc
CPU Instruction Interface (clk_cpu)		
i_cpu_iadr	32	Instruction address
i_cpu_istrobe	1	Command strobe
i_cpu_ifetch	1	Read enable
o_cpu_idata	32	Response data for i_cpu_iadr
o_cpu_iready	1	Instruction access complete
CPU Data Interface (clk_cpu)		
i_cpu_dadr	32	Data address
i_cpu_dstrobe	1	Command strobe
i_cpu_drd	1	Read enable
i_cpu_dwr	1	Write enable
i_cpu_dben	4	Byte enables
i_cpu_dwdata	32	Data to be written
o_cpu_drdata	32	Response data
o_cpu_dready	1	Data access complete
Control block (CTL) Interface (clk_mc)		
i_ctl_entry0_base	12	Entry #0 base address (12 MSB)
i_ctl_entry0_u_flag	1	Entry #0 flag: U
i_ctl_entry1_base	12	Entry #1 base address (12 MSB)
i_ctl_entry1_end	12	Entry #1 end address (12 MSB)
i_ctl_entry1_flags	5	Entry #1 flags: C, R, X, U, P
i_ctl_entry1_valid	1	Entry #1 valid
i_ctl_entry2_base	12	Entry #2 base address (12 MSB)
i_ctl_entry2_end	12	Entry #2 end address (12 MSB)
i_ctl_entry2_flags	5	Entry #2 flags: C, R, X, U, P
i_ctl_entry2_valid	1	Entry #2 valid
i_ctl_entry3_base	12	Entry #3 base address (12 MSB)
i_ctl_entry3_end	12	Entry #3 end address (12 MSB)
i_ctl_entry3_flags	5	Entry #3 flags: C, R, X, U, P
i_ctl_entry3_valid	1	Entry #3 valid
i_ctl_entry4_base	12	Entry #4 base address (12 MSB)
i_ctl_entry4_end	12	Entry #4 end address (12 MSB)

Pin Name	Width	Description
i_ctl_entry4_flags	5	Entry #4 flags: C, R, X, U, P
i_ctl_entry4_valid	1	Entry #4 valid
i_ctl_privileged	1	CPU in privileged mode
o_ctl_perm_fault	1	Permission fault detected
o_ctl_miss_fault	1	ART address not found fault detected
o_ctl_tlb_fault	1	TLB address not found fault detected
i_ctl_fault_ack	1	Fault acknowledged
Instruction Level 1 Cache block (IL1) Interface (clk_mc)		
o_il1_adr	32	Instruction address
o_il1_valid	1	Access valid
o_il1_flag	1	Instruction flag: C
i_il1_data	32	Response data
i_il1_tlb_fault	1	IL1 encountered a TLB fault
i_il1_stall	1	When low, IL1 response is ready
Data Level 1 Cache block (DL1) Interface (clk_mc)		
o_dl1_adr	32	Data address
o_dl1_valid	1	Access valid
o_dl1_flags	2	Data flags: C, I
o_dl1_ben	4	Byte enables
o_dl1_wen	1	Data write enable
o_dl1_wdata	32	Data to be written
i_dl1_rdata	32	Response data
i_dl1_tlb_fault	1	DL1 encountered a TLB fault
i_dl1_stall	1	When low, DL1 response is ready

Table 2.1: ART Pin List

2.1.3 Functionality

ART receives from CTL a number of address regions, along with a set of flags for each region. All these signals are considered static and are used directly by ART.

Each region is defined by a start, low address (i_ctl_entryN_base) as well as a stop, high address (i_ctl_entryN_end). These boundaries are 1MB aligned and thus only the 12 MSB bits of the CPU virtual addresses are needed. A region entry is considered valid when its i_ctl_entryN_valid bit is set, otherwise it is ignored by the ART block.

The flags (i_ctl_entryN_flags) handled by ART for each region are the following:

C flag: Cacheable bit. When 0, region is non-cacheable and therefore will not be allocated in either L1 or L2 caches.

R flag: Read-only bit. When 1, region is read-only. Trying to write on it (through the data interface) will generate a Permission fault.

X flag: Execute bit. When 0, region is non-execute. Trying to read it through the instruction interface will generate a Permission fault.

U flag: User access bit. When 0, region is not accessible in user mode. Trying to read it while the CPU is in user mode (i_ctl_privileged = 0) will generate a Permission fault. *Exception:* only for the first Entry, which is the compulsory I/O region entry, access to the first register (CPU System Call Register, section 4.2.1, page 67) is granted for user mode even if Entry #0 is marked with U=0. Thus, if i_cpu_dadr[19:0] = 20'b0 and i_cpu_dadr[31:20] belongs to Entry #0, no Permission fault is generated.

P flag: Privileged access bit. When 0, region is not accessible in privileged mode. Trying to read it while the CPU is in privileged mode (i_ctl_privileged = 1) will generate a Permission fault.

In addition to these flags, entry #0 is considered to be an I/O region and entries #1 to #4 non-I/O regions. Entry #0 also has a bound equal to its base address (i.e. it's always 1MB in size) and has hardwired values of C=0, R=0, X=0 and P=1. Accesses to the I/O region will not result in memory access. Instead, the LSB of the address will be used to address a memory mapped register in CTL or a mailbox or counter in CMX.

ART listens to CPU instruction and data interfaces. For each of them, it tries to map the address the CPU is trying to access to the first region that is valid and includes this address. If no region includes this address, a Miss fault is generated. Under normal circumstances, the address hits in some region and the flags for the region are used to determine if the access is valid, as specified above; otherwise, a Permission fault is generated.

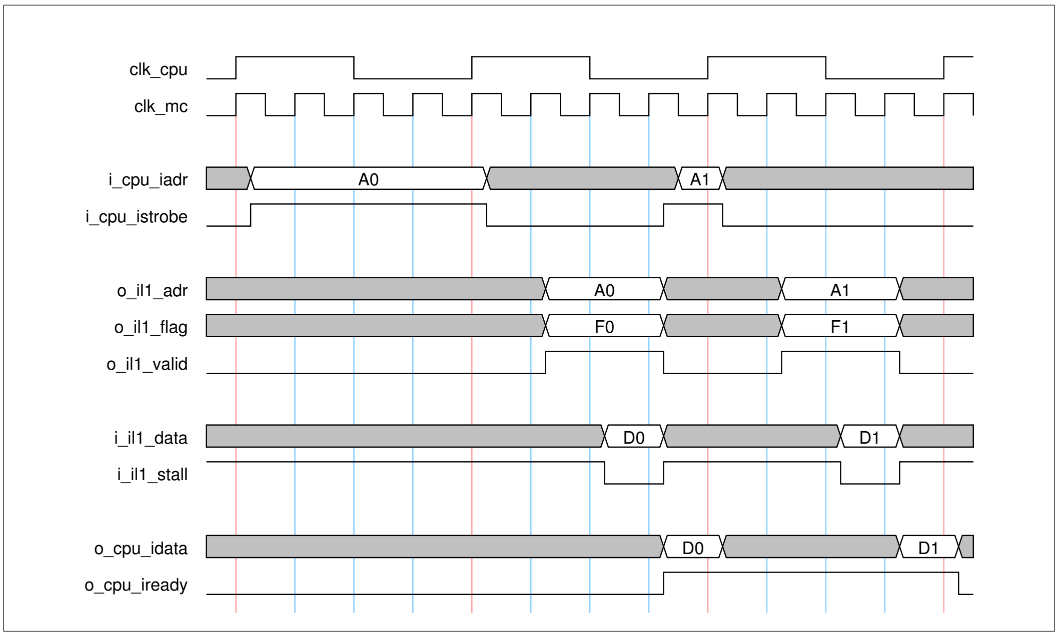


Figure 2.2: ART Interface to IL1: Two consecutive IL1 hits

If the address passes these tests, it is forwarded accordingly to either the IL1 block, if it came from the Instruction CPU interface or to the DL1 block, if it came from the Data CPU interface. ART must wait until IL1 or DL1 notify that the access is complete (i_?l1_stall goes to 0 for one clock cycle) before notifying the CPU that the access is complete (o_cpu_?ready). The reason is that the L1 caches can respond with a TLB fault (i_?l1_tlb_fault). In this case, ART must generate a TLB fault. If no fault is encountered, ART will finalize the access to the CPU, asserting the related ready signal and responding with the data in the case of a read access.

This process is shown in figure 2.2, where two CPU accesses (of address A0 and then A1) are forwarded to IL1, which responds after a single clk_mc cycle (note later that the L1 response is forwarded combinatorially back to ART, in order to fit a complete CPU-to-L1-hit into 4 clk_mc cycles, which equal 1 clk_cpu cycle). ART forwards the L1 response back to the CPU on the next clk_mc cycle, which means that the CPU will manage to see the response 1/4 of a clk_cpu cycle before the next edge. This is the reason that the ready signal is asserted for two clk_cpu positive edges continously and the CPU thinks that it can access the L1 caches in back-to-back clk_cpu cycles.

Faults and interrupts proccess

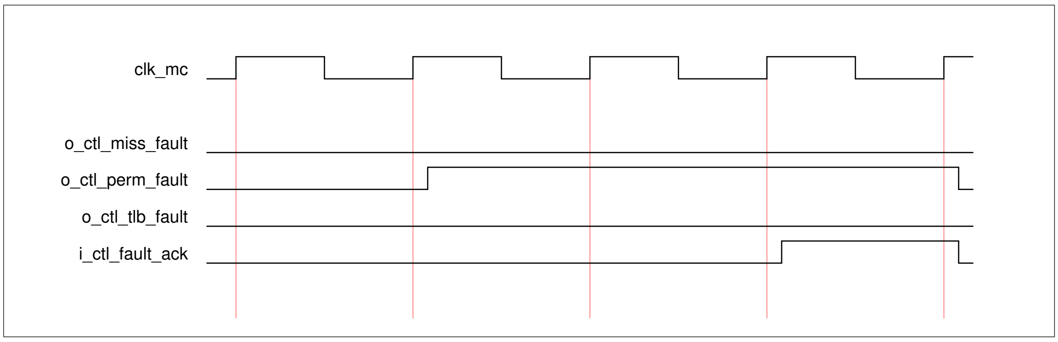


Figure 2.3: ART Interface to CTL: Permission fault occurs

In case any of the three faults is generated (Miss, Permission or TLB fault) ART must notify the CTL block, which will accordingly trigger an interrupt to the CPU. This is shown in figure 2.3.

In order for the CPU to be able to process an interrupt, it must not have any pending instruction or data accesses. Whenever a fault is given to the CTL block, ART must ensure the following:

- If any instruction was pending from the CPU, a NOP must be answered (instruction 32'h80000000). Any answer from IL1 must be ignored.
- If any data read was pending from the CPU, a zero word must be returned (data 32'h00000000). Any answer from DL1 must be ignored.
- If any data write was pending from the CPU, the o_cpu_dready signal must be asserted. Any answer from DL1 must be ignored.
- ART will resume its normal operation only when the the above conditions are fulfilled, as well as the CTL block has acknowledged the fault (i_ctl_fault_ack is set for one clk_mc cycle).

2.1.4 Features to be tested

ART-1: Regions that have addresses occuring in the testcase, but their valid bit is set to 0

ART-2: Miss fault

ART-3: Permission fault

ART-4: TLB fault

ART-5: System call entry with I/O region tested at P and not P cases

ART-6: Region priority access (overlapping region entries)

ART-7: Region reset values

2.2 Instruction L1 Cache block (IL1)

The IL1 block is a trimmed-down version of the Data L1 Cache block (DL1, section 2.3). Please refer to this block for avoidance of text duplication.

The differences between the two blocks are only the following:

- IL1 is 4-KB, two-way set associative (DL1 is 8-KB)
- IL1 does not implement any writes
- IL1 does not have an input for the I/O (I) bit

2.3 Data L1 Cache block (DL1)

2.3.1 Purpose

The DL1 block does the following:

- Provides an 8-KB, two-way set associative, write through, no write allocate level 1 cache for data accesses
- Forwards without storing any access that is marked non-cacheable (C=0) or I/O (I=1). Also, everything is forwarded when the block is disabled.
- Provides a serial clear operation, where all entries are invalidated in both ways
- Provides an invalidation interface to L2 cache, where a single entry is invalidated if it exists in the DL1 cache

2.3.2 Pin List

Pin Name	Width	Description
clk_mc	1	ART & Caches clock (40 MHz)
rst_mc	1	Reset for clk_mc
Control block (CTL) Interface		
i_ctl_en	1	Cache enabled
i_ctl_clear_req	1	Clear operation request
o_ctl_clear_ack	1	Clear operation completed
Address Region Table block (ART) Interface		
i_art_adr	32	Data address
i_art_flags	2	Data flags: C, I
i_art_ben	4	Byte enables
i_art_wen	1	Data write enable
i_art_wdata	32	Data to be written
i_art_valid	1	Access valid
o_art_rdata	32	Response data
o_art_tlb_fault	1	L2C encountered a TLB fault
o_art_stall	1	When low, response is ready
Level 2 Cache block (L2C) Request Interface		
o_l2c_adr	32	Data address
o_l2c_flags	2	Data flags: C, I
o_l2c_ben	4	Byte enables
o_l2c_wen	1	Data write enable
o_l2c_wdata	32	Data to be written
o_l2c_valid	1	Access valid
i_l2c_rdata_valid	1	Response data valid
i_l2c_rdata	32	Response data
i_l2c_tlb_fault	1	L2C encountered a TLB fault
i_l2c_stall	1	When low, L2C response is ready
Level 2 Cache block (L2C) Invalidation Interface		
i_l2c_inv_req	1	Invalidation request
i_l2c_inv_adr	32	Cache line address to be invalidated
o_l2c_inv_ack	1	Cache line invalidated

Table 2.2: DL1 Pin List

2.3.3 Memory Organization

- 2 ways. Each way has 1024 words of 32 bits = 128 cache lines of 32 bytes. Total size 2 x 128 x 32 = 8 KB.
- adress (32 bits) = tag (20 bits) + index (7 bits) + byte offset (5 bits)
- We have 2 x 128 tags of 22 bits (20 tag + 1 valid + 1 LRU). Both are stored in a single BRAM, 512 x 32. One port reads the tag for way #0, second port for way #1.
- Each data way is 2 BRAMs 512 x 32, oraganized as 1024 x 32.

2.3.4 Timing

In the case of normal, cacheable, non-I/O accesses (C = 1 and I = 0), the timing is as follows:

Read hit: Cycle 1: read tags and both data. Cycle 2: mux correct data back to ART combinatorially.

Read miss: Cycle 1: read tags. Cycle 2: read request from L2. Cycle N: get first word. Cycle N+7: get last word. Respond to ART on correct cycle from N to N+7. After N+7, get ready for next access.

Write hit: Cycle 1: read tags. Cycle 2: update tags (LRU bit), write data, write them through to L2 and respond OK to ART (don't wait for L2 completion, it's a hit).

Write miss: Cycle 1: read tags. Cycle 2: write request to L2. Cycle N: get the response (or TLB fault) from L2 and respond accordingly to ART.

In the case of non-cacheable or I/O accesses (C = 0 or I = 1), no hits are allowed. The misses are treated as follows:

Read miss: Cycle 1: read request from L2. Cycle N: get *single* word, respond to ART.

Write miss: Cycle 1: write request to L2. Cycle N: get the response (or TLB fault) from L2 and respond accordingly to ART.

2.3.5 Features to be tested

DL1-1: Way allocation and replacement policy

DL1-2: Cacheable bit: check that C=0 or I=1 expects single-word L2 answers (and not cache line size)

DL1-3: Enable bit: check that tags/data arrays are not written and that expected L2 answer does NOT depend on it (it must only depend on the C & I bits).

DL1-4: I/O bit, also verify that behaves as C = 0

DL1-5: L2 invalidation request (64 bytes $\rightarrow$ 32 bytes)

DL1-6: CTL clear request

2.4 Level 2 Cache (L2C)

2.4.1 Memory Organization

- 8 ways. Each way has 8192 words of 32 bits = 512 cache lines of 64 bytes. Total size 8 x 512 x 64 = 256 KB.
- address (32 bits) = tag (17 bits) + index (9 bits) + byte offset (6 bits)
- We have 8 x 512 tags of 25 bits (17 tag + 3 epoch + 1 valid + 1 dirty + 3 state). They are stored in 6 BRAMs (each 512 x 36), organized as a memory array of 512 x 216. Each tag occupies 3 BRAM "lanes" of 9 bits (8 data + 1 parity), so that 3 byte enables can be used to write it (25 of the 27 bits in the 3 lanes are used). Thus, the memory is actually organized as 512 x 8 x 27 = 512 x 216.
- An LRU policy is also supported, using 3 bits per tag. Due to floorplanning issues, and to avoid some multiplexers, these are implemented in a separate BRAM.
- The L2 data are stored in the off-FPGA SRAM through one of the ports of the SRAM_CTL block.

2.4.2 Pin List

Pin Name	Width	Description
clk_mc	1	ART & Caches clock (40 MHz)
rst_mc	1	Reset for clk_mc
Control Block (CTL) Interface		
i_ctl_en	1	Cache enabled
i_ctl_lru_mode	1	0 = Epoch-based replacement policy 1 = LRU-based replacement policy
i_ctl_clear_req	1	Clear operation request
i_ctl_flush_req	1	Flush operation request
o_ctl_maint_ack	1	Clear/flush operation(s) completed
i_ctl_epoch	3	Current task epoch number
i_ctl_min_cpu_ways	3	Minimum ways reserved for CPU per epoch
o_ctl_trace_ihit	1	Instruction-side hit event
o_ctl_trace_imiss	1	Instruction-side miss event
o_ctl_trace_dhit	1	Data-side hit event
o_ctl_trace_dmiss	1	Data-side miss event
SRAM Controller (SRAM_CTL) Interface		
o_sctl_req_adr	18	Word address
o_sctl_req_we	1	Write enable
o_sctl_req_wdata	32	Data to be written
o_sctl_req_be	4	Byte enables
o_sctl_req_valid	1	Request valid
i_sctl_resp_rdata	32	Response data read from SRAM
i_sctl_resp_valid	1	Response valid
Instruction L1 Cache (IL1) Interface		
i_il1_adr	32	Instr address (cline or word aligned)
i_il1_flag	1	Instr flag: C
i_il1_valid	1	Access valid
o_il1_rdata_valid	1	Response data valid
o_il1_rdata	32	Response data
o_il1_tlb_fault	1	L2C encountered a TLB fault
o_il1_stall	1	When low, L2C response is ready

Pin Name	Width	Description
o_il1_inv_req	1	Invalidation request
o_il1_inv_adr	32	Cache line address to be invalidated
i_il1_inv_ack	1	Cache line invalidated
Data L1 Cache (DL1) Interface		
i_dl1_adr	32	Data address (cline or word aligned)
i_dl1_flags	2	Data flags: C, I
i_dl1_ben	4	Byte enables
i_dl1_wen	1	Data write enable
i_dl1_wdata	32	Data to be written
i_dl1_valid	1	Access valid
o_dl1_rdata_valid	1	Response data valid
o_dl1_rdata	32	Response data
o_dl1_tlb_fault	1	L2C encountered a TLB fault
o_dl1_stall	1	When low, L2C response is ready
o_dl1_inv_req	1	Invalidation request
o_dl1_inv_adr	32	Cache line address to be invalidated
i_dl1_inv_ack	1	Cache line invalidated
Microblaze Network Interface (MNI): Writeback Interface		
i_mni_wb_space	1	Space for a new writeback exists;
		Valid can be asserted and data placed
		on dout at any time
o_mni_wb_valid	1	New writeback request
o_mni_wb_adr	32	Writeback address (cline aligned)
Microblaze Network Interface (MNI): Writeback Acknowledge Interface		
i_mni_wb_ack_valid	1	New writeback acknowledge request
i_mni_wb_ack_fault	1	When 1, memory error was encountered
i_mni_wb_ack_adr	32	Wback ack address (cline/word aligned)
o_mni_wb_ack_stall	1	Acknowledge accepted when 0
Microblaze Network Interface (MNI): Miss Interface		
o_mni_miss_valid	1	New miss request
o_mni_miss_adr	32	Miss address (cline or word aligned)
o_mni_miss_flags	2	Miss flags: C, I
o_mni_miss_wen	1	Miss write enable (for C=0 or I=1)
o_mni_miss_ben	4	Byte enables (for C=0 or I=1)
o_mni_miss_wdata	32	Miss write data (for C=0 or I=1)
i_mni_miss_stall	1	Miss accepted when 0
Microblaze Network Interface (MNI): Fill Interface		
i_mni_fill_valid	1	New fill request
i_mni_fill_fault	1	When 1, memory error was encountered
i_mni_fill_len	4	Fill length in 32b-words
i_mni_fill_adr	32	Fill address (cline or word aligned)
o_mni_fill_stall	1	Fill accepted when 0; place data on din
Microblaze Network Interface (MNI): Write Interface		
i_mni_write_valid	1	New write request
i_mni_write_adr	32	Write address (cline aligned)
i_mni_write_dirty	1	Dirty bit value of incoming line
o_mni_write_stall	1	Write accepted/not accepted when 0;
		If nack=0, place data on din
o_mni_write_nack	1	When 1, write is not accepted
Microblaze Network Interface (MNI): Read Interface		
i_mni_read_valid	1	New read request
i_mni_read_adr	32	Read address (cline aligned)
i_mni_read_ignore	1	Reset dirty bit value on read hit

Pin Name	**Width**	**Description**
o_mni_read_stall	1	Read accepted/not accepted when 0;
		If nack=0, data will be placed on dout
o_mni_read_nack	1	When 1, read is not accepted
Microblaze Network Interface (MNI): Common Data Busses		
i_mni_data	32	Common Data Input from MNI
o_mni_data	32	Common Data Output to MNI

Table 2.3: L2C Pin List

2.4.3 Tag Organization

The BRAM-stored tags are structured as follows:

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
	V	D	B	W	F	EPC	Cache line tag

Where:

V: If 1, cache line is valid, otherwise it doesn't exist.

D: If 1, cache line is dirty.

B: If 1, a Writeback is scheduled for the cache line but has not yet completed.

W: If 1, a Write is scheduled for the cache line but has not yet been written.

F: If 1, a Fill is scheduled for the cache line but has not yet been received.

EPC: Epoch number.

Tag: Bits [31:15] of the cache line address. When F=0 and W=0, the tag refers to the stable or new cache line. When F=1 or W=1 the tag refers to the old cache line that has been written back (B=0) or is scheduled to be written back (B=1).

The cache line can be used for Hits only when V=1, B=0, W=0 and F=0.

A 3-bit LRU per tag is also stored in a separate BRAM. When L2C operates in LRU mode, the 3 EPC bits are ignored and the 3 LRU bits are used instead for replacement policy decisions.

2.4.4 Functionality

Figure 2.4 shows the L2C internal block structure.

The Level 2 Cache answers to requests from multiple agents: the two Level 1 Caches (instruction and data), the four MNI-initiated interfaces(write, read, fill, writeback acknowledges) and the CTL-initiated maintenance operations. The arbitration and ordering of all these requests happens in the Tags subblock, which makes all the decisions regarding replacements and holds transient states in the related BRAM tag bits. Access to the L2C data happens through the SRAM subblock, which receives arbitration orders from the Tags subblock.

We will describe how each of the events that can happen are handled by the various subblocks in L2C.

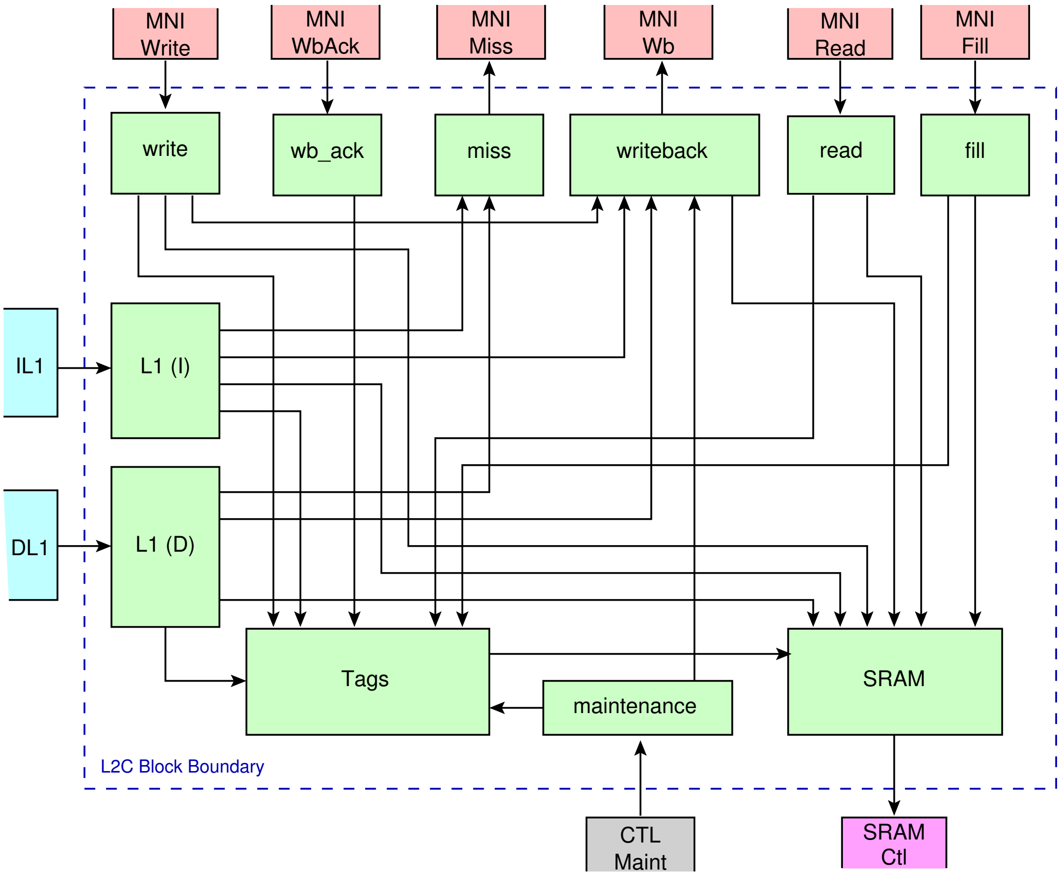


Figure 2.4: L2C block diagram

L1 Cache Accesses

Instruction or Data accesses are handled by two separate L1 subblock instances. There are a few cases:

Uncacheable or I/O accesses:
The L1 subblock requests a new miss to the Miss subblock. When the miss request is done, if it was a read request then the Fill subblock is monitored until a reply is returned. If it was a write to non-I/O space, then the WbAck subblock is monitored until the acknowledgment for the write has arrived. For I/O space writes, the L1 access terminates directly after the write is done (MNI does the write directly to the CTL registers).

Cacheable hits:
The L1 subblock requests a Tag access; if the Tags subblock replies it's a hit, then L1 waits for an SRAM subblock reply with the correct data (for read hits) or the SRAM permission to write data (for write hits).

Hits happen only on non-transient lines (B=W=F=0). Otherwise, the L1 subblock waits and retries the Tags request whenever a new WbAck, Fill or Write is completed — these subblocks broadcast their completions for these purpose.

Cacheable misses, no replacement:
The L1 subblock requests a Tag access; if the Tags subblock replies it's a clean miss, then L1 does a miss request to the Miss subblock and then waits for the Fill subblock to fetch the cache line. While Fill writes the cache line to the SRAM, L1 intercepts the correct 32-B half and forwards it to the L1 block. In the case of a write miss, for the

one clock cycle that refers to the exact word, the L1 subblock changes the word that goes to the SRAM on the fly (taking care of the byte enables as well, by combining Fill data with CPU-supplied data in a byte-by-byte basis).

The Tags set the F bit for the way that the Fill is expected to arrive into. As soon as the Fill arrives, the F bit is cleared, the new tag is written and the V bit is asserted.

Cacheable misses, with replacement:
In the above case, if Tags reply that it's a replacement, then before the Miss request the L1 subblock does a Writeback request for the old cache line. The rest of the procedure is as described above.

The Tags set both the B and F bits in this case. The Writeback subblock must send the line that is marked with the B bit to the DRAM over the network. As soon as the Writeback is done and an acknowledge has come (monitored by the WbAck subblock), the B bit is cleared for this cache line. If a Fill has arrived but still finds the old cache line (B set), it waits and retries the Tags access whenever new WbAcks arrive. When the B bit is clear, the Fill proceeds as above, puts the cache line in the SRAM, clears the F bit, sets the new tag and asserts the V bit.

MNI Write Accesses

A network-initiated Write access tries to push a new cache line into the L2C. These are handled by the Write subblock. The following can happen in such a case:

Refusal:
Write accesses the Tags and requests a new Write for the cache line address. When L2C operates in Epoch mode, it can happen that the network epoch has already exceeded its allowed cache lines (see the Cache Control register (page 76) for more details). In this case, a Nack is answered to MNI and the cache line is rejected. Note that in LRU mode a Write is always accepted.

Clean insertion:
If the Tags answer that the line is accepted into a clean position (no B flag is set), Write waits for permission from the SRAM subblock to write the cache line data into the selected cache way, which is the one that the Tags have asserted the W bit.

As soon as the data are placed, Write accesses Tags for a second time to notify that the data is in place, reset the W bit, assert the V bit and install the correct tag.

Following that, invalidations are sent both to the IL1 and the DL1 caches; if either of them has the cache line, they invalidate them. Note that they cannot have new data on the line, since the caches are write-trhough, and they are not allowed to introduce a race condition, because the programmer model forbids accessing through the CPU memory areas that are under DMA access.

Replacement and insertion:
If the Tags answer that the line is accepted but it must take the place of an old line which is dirty, then the Tags have set both the B and W bits. Write first does a request to the Writeback block to take care of evicting the old line; when the Writeback has been accepted by the Tags and arbitrated for an SRAM access, the Write access continues as above (the data will be arbitrated to be placed in SRAM after the writeback sends over the old data). However, when for the second Tags access from Write block to succeed, the B bit must have been cleared by the WbAck, otherwise Write waits and retries the access after new WbAck broadcasts.

MNI Read Accesses

A network-initiated Read access probes if a cache line is in L2C, and if so reads its data. This is handled by the Read subblock. The following can happen in such a case:

Not found:
Read accesses the Tags and requests a new Read for the cache line address. If the cache line is not present, a Nack is sent to the MNI and the access terminates.

If the tag of the cache line is found in L2C but is in a transient state (W or F or B is set), the access is retried after any WbAck, Fill or Write completion broadcast.

Found:
If the cache line is found, Read waits until the SRAM subblock starts reading the cache line from the SRAM. The data is sent over to the MNI.

Maintenance Operations

The CTL block may request L2C maintenance operations, such as Clear (invalidate all tags), Flush (writeback all dirty lines) or both Clear and Flush at the same time.

First, the Maintenance subblock notifies all other initiator subblocks (both L1 instances, Read and Write) that maintenance is under way and they should remain idle after they have completed their current accesses. When this has been achieved, maintenance begins serially for every set in the cache.

If a Flush has been ordered, for every dirty way a Writeback is requested and a WbAck is waited for. When all ways are not dirty, the next set is considered.

If a Clear has been ordered, all ways are marked invalid (V=0).

For combined flush/clears, first the Flush is done for the whole set and then the Clear.

2.5 MBS Network Interface (MNI)

2.5.1 Purpose

- Provides the single MBS access point to the network
- Links L2C to the network (Miss requests/replies and Read/Write coherent accesses)
- Provides access to the CTL registers through the network
- Sends CMX notifications over the network
- Provides a DMA engine that is coherent with the L2C block

2.5.2 Pin List

Pin Name	Width	Description
clk_ni	1	Network Interface clock (80 MHz)
clk_mc	1	ART & Caches clock (40 MHz)
rst_ni	1	Reset for clk_ni
rst_mc	1	Reset for clk_mc
Configuration (static)		
i_board_id	8	Board ID
i_node_id	4	Node ID
i_ctl_adr_base	12	CTL regisers base address (12 MSB)
CTL Register Access Interface (clk_ni)		
o_ctl_reg_adr	20	Register offset address (2B-aligned)
o_ctl_reg_valid	1	New register access valid
o_ctl_reg_wen	1	When 1, access is write
o_ctl_reg_from_cpu	1	When 1, access originates from CPU When 0, access originates from Network
o_ctl_reg_ben	2	Byte enables
o_ctl_reg_wdata	16	MNI access data to be written
o_ctl_reg_rlen	3	Read length in 16b words (0=1w, 7=8w)
i_ctl_reg_stall	1	When 1, no more requests can be handled
i_ctl_reg_resp_valid	1	Response valid
i_ctl_reg_resp_rdata	16	Response data read
i_ctl_reg_resp_block	1	Response incomplete, block CPU
i_ctl_reg_resp_unblock	1	Pending response is now carried out
CTL Operation Interface (clk_ni)		
i_ctl_op_valid	1	New operation valid
i_ctl_op_data	16	Operation data
o_ctl_op_stall	1	When 1, stop sending operations
o_ctl_cpu_fifo_ops	6	CPU operations available FIFO space
o_ctl_net_fifo_ops	6	Network operations available FIFO space
CTL Trace Interface (clk_ni)		
o_ctl_trace_op_local	1	New CPU DMA operation
o_ctl_trace_op_remote	1	New network DMA operation
o_ctl_trace_read_hit	1	Read from L2C succeeded
o_ctl_trace_read_miss	1	Read from L2C failed
o_ctl_trace_write_hit	1	Write to L2C succeeded
o_ctl_trace_write_miss	1	Write to L2C refused
o_ctl_trace_vc0_in	1	New VC #0 input packet
o_ctl_trace_vc0_out	1	New VC #0 output packet
o_ctl_trace_vc1_in	1	New VC #1 input packet

Pin Name	Width	Description
o_ctl_trace_vc1_out	1	New VC #1 output packet
o_ctl_trace_vc2_in	1	New VC #2 input packet
o_ctl_trace_vc2_out	1	New VC #2 output packet
CMX Interface (clk_ni)		
i_cmx_valid	1	Request for a new NI notification
i_cmx_data	16	Notification board/node/address
o_cmx_stall	1	When 1, no more data can be received
i_cmx_mbox_space	10	Available mailbox space (in 16-b words)
i_cmx_mslot_space	1	When 1, mailslot is empty
L2C Writeback Interface (clk_mc)		
o_l2c_wb_space	1	Space for a new writeback exists; Valid can be asserted and data placed on input data bus at any time
i_l2c_wb_valid	1	New writeback request
i_l2c_wb_adr	32	Writeback address (cline aligned)
L2C Writeback Acknowledge Interface (clk_mc)		
o_l2c_wb_ack_valid	1	New writeback acknowledge request
o_l2c_wb_ack_fault	1	When 1, memory error was encountered
o_l2c_wb_ack_adr	32	Wback ack address (cline/word aligned)
i_l2c_wb_ack_stall	1	Acknowledge accepted when 0
L2C Miss Interface (clk_mc)		
i_l2c_miss_valid	1	New miss request
i_l2c_miss_adr	32	Miss address (cline or word aligned)
i_l2c_miss_flags	2	Miss flags: C, I
i_l2c_miss_wen	1	Miss write enable (for C=0 or I=1)
i_l2c_miss_ben	4	Byte enables (for C=0 or I=1)
i_l2c_miss_wdata	32	Miss write data (for C=0 or I=1)
o_l2c_miss_stall	1	Miss accepted when 0
L2C Fill Interface (clk_mc)		
o_l2c_fill_valid	1	New fill request
o_l2c_fill_fault	1	When 1, memory error was encountered
o_l2c_fill_len	4	Fill length in 32b-words
o_l2c_fill_adr	32	Fill address (cline or word aligned)
i_l2c_fill_stall	1	Fill accepted when 0; place data on dout
L2C Write Interface (clk_mc)		
o_l2c_write_valid	1	New write request
o_l2c_write_adr	32	Write address (cline aligned)
o_l2c_write_dirty	1	Cache line dirty bit value
i_l2c_write_stall	1	Write accepted/not accepted when 0;
		If nack=0, place data on dout
i_l2c_write_nack	1	When 1, write is not accepted
L2C Read Interface (clk_mc)		
o_l2c_read_valid	1	New read request
o_l2c_read_adr	32	Read address (cline aligned)
o_l2c_read_ignore	1	When 1, reset cache line dirty bit to 0
i_l2c_read_stall	1	Read accepted/not accepted when 0;
		If nack=0, place data on din
i_l2c_read_nack	1	When 1, read is not accepted
L2C Common Data Busses (clk_mc)		
i_l2c_data	32	Common Data Input from L2C
o_l2c_data	32	Common Data Output to L2C
Network Output Interface (clk_ni)		
o_nout_enq	3	Enqueue (per VC bitmap)

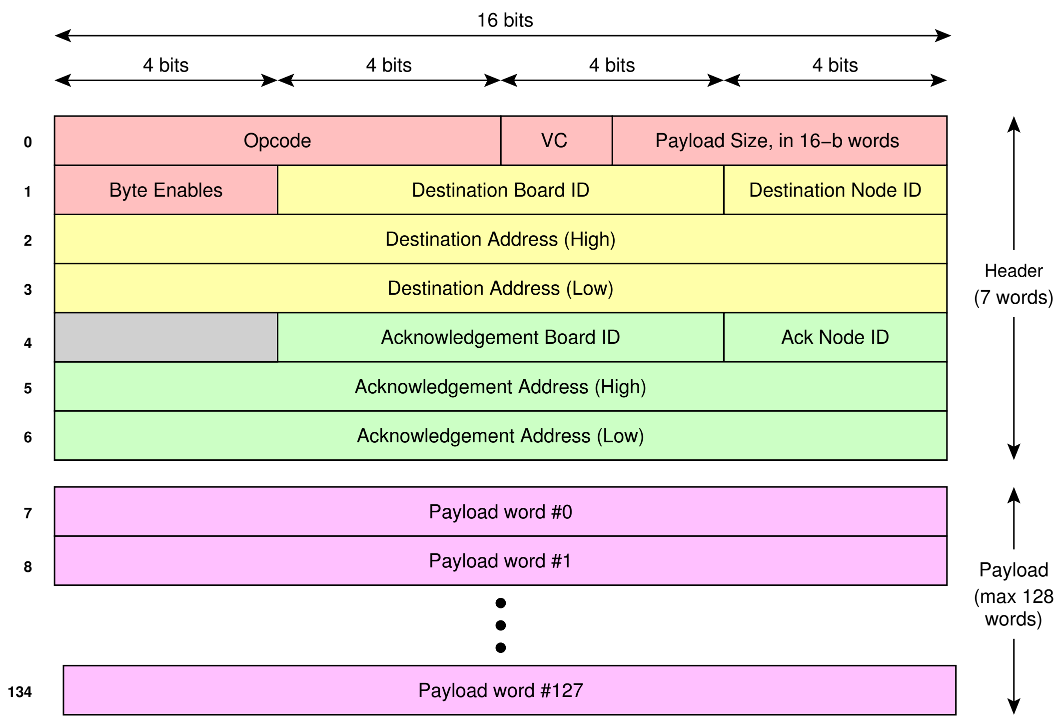


Figure 2.5: Network Packet (Write packet format, for intra-FPGA use)

Pin Name	Width	Description
o_nout_offset	8	Offset in packet (in 16-b words)
o_nout_eop	1	End of packet, send it to destination
o_nout_data	16	Data to be written to offset
i_nout_full	3	FIFO full (per VC bitmap)
i_nout_packets_vc0	3	Space left in VC #0 FIFO (in packets)
i_nout_packets_vc1	3	Space left in VC #1 FIFO (in packets)
i_nout_packets_vc2	3	Space left in VC #2 FIFO (in packets)
Network Input Interface (clk_ni)		
o_nin_deq	3	Dequeue (per VC bitmap)
o_nin_offset	8	Offset in packet (in 16-b words)
o_nin_eop	1	End of packet, proceed to next one
i_nin_data	16	Data read from offset
i_nin_empty	3	FIFO empty (per VC bitmap)
i_nin_packets_vc0	3	Packets arrived in VC #0 FIFO
i_nin_packets_vc1	3	Packets arrived in VC #1 FIFO
i_nin_packets_vc2	3	Packets arrived in VC #2 FIFO

Table 2.4: MNI Pin List

2.5.3 Network packet format

The network packet format for intra-FPGA use is shown in figures 2.5 (Write packet format – VCs #0 or #1) and 2.6 (Read packet format – VC #2). There are no CRC fields for header, payload or packet. Checksums for inter-FPGA use are added and removed on the fly, as specified in section 3.5 (page 54).

The Opcode field is 6 bits, and consists of the following fields:

Bit 0: 0 = Normal, 1 = Ignore dirty bit on source.

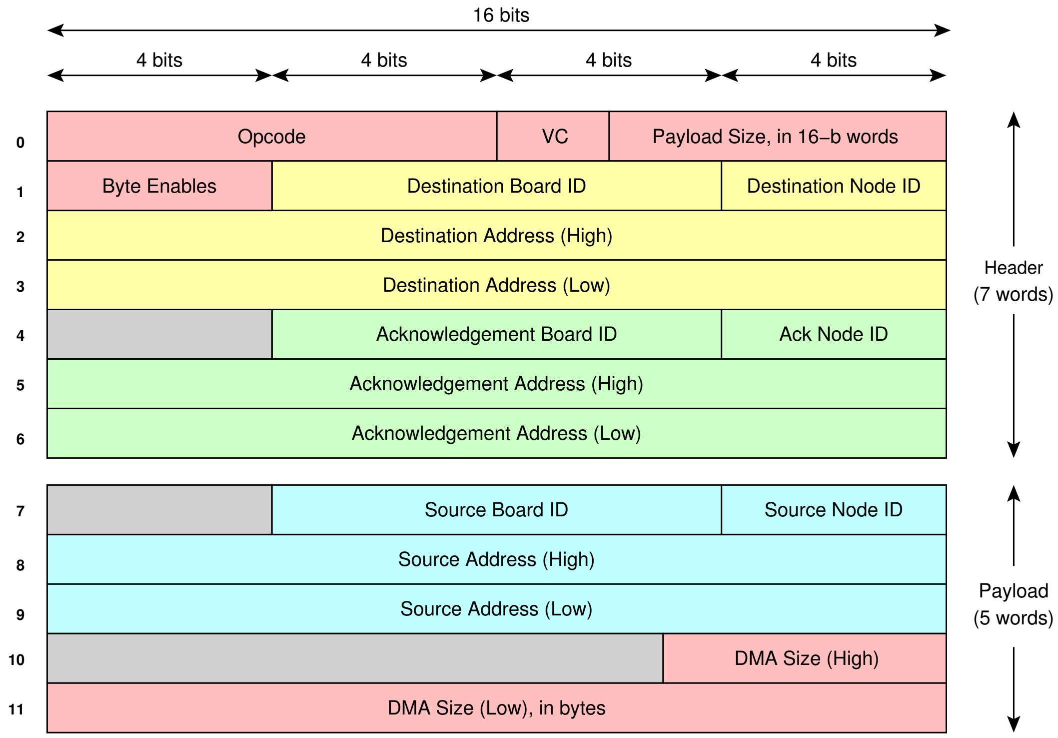


Figure 2.6: Network Packet (Read packet format, for intra-FPGA use)

Bit 1: 0 = Don't use Acknowledgment, 1 = Use Acknowledgment.

Bit 2: 0 = Miss/fill traffic, 1 = Generic traffic.

Bit 3: 0 = Normal, 1 = Write through on destination.

Bit 4: 0 = Normal, 1 = Clean on destination.

Bit 5: 0 = Normal, 1 = TLB error.

Bit 6: 0 = Write packet, 1 = Read packet.

Bit 7: Reserved for future use.

The Payload Size field contains the length of the packet payload in 16-b words. The payload has a maximum of 32 words. Write packets are considered the ones having opcode[6]=1, read packets are considered the ones having opcode[6]=0.

Acknowledge packets are always sent using VC #0, which is the highest priority VC. Write packets which will *not* cause any other write (but may cause an acknowledge) can be sent using VC #1, the medium priority VC. Read packets, which will definitely cause a Write packet response, must be sent using VC #2, the lowest priority VC. Write packets which can trigger another write must also be sent in VC #2, to avoid deadlocks.

As an example, the DMA engine sending data to another MBS (which will trigger an L2C Write event there) should use VC #2, because the L2C Write on the destination may trigger a Writeback to the DRAM which must complete before the Write. This writeback should use VC #1, and its acknowledge VC #0.

For Write packets, Acknowledgment Board ID, Node ID and Address will be used when Opcode[1]=1 to send a new acknowledgement VC #0 Write packet to the specified address. In its single-word payload, the written bytes will be specified.

For Read packets, the acknowledgment address will be transfered to the reponse VC #1 Write packet as it is, again if Opcode[1]=1; when the reponse packet arrives at its destination, an acknowledgement packet will be generated in VC #0 as specified above.

Write packets contain in their payload only the data to be written to the destination address. Read packets request to read from the Destination address a number of bytes equal to the DMA Size (fourth/fifth payload word). The reply VC #1 Write packet will be sent with Destination Board/Node/Address fields set to the ones specified in the VC #2 Read packet Source fields.

2.5.4 Block diagram

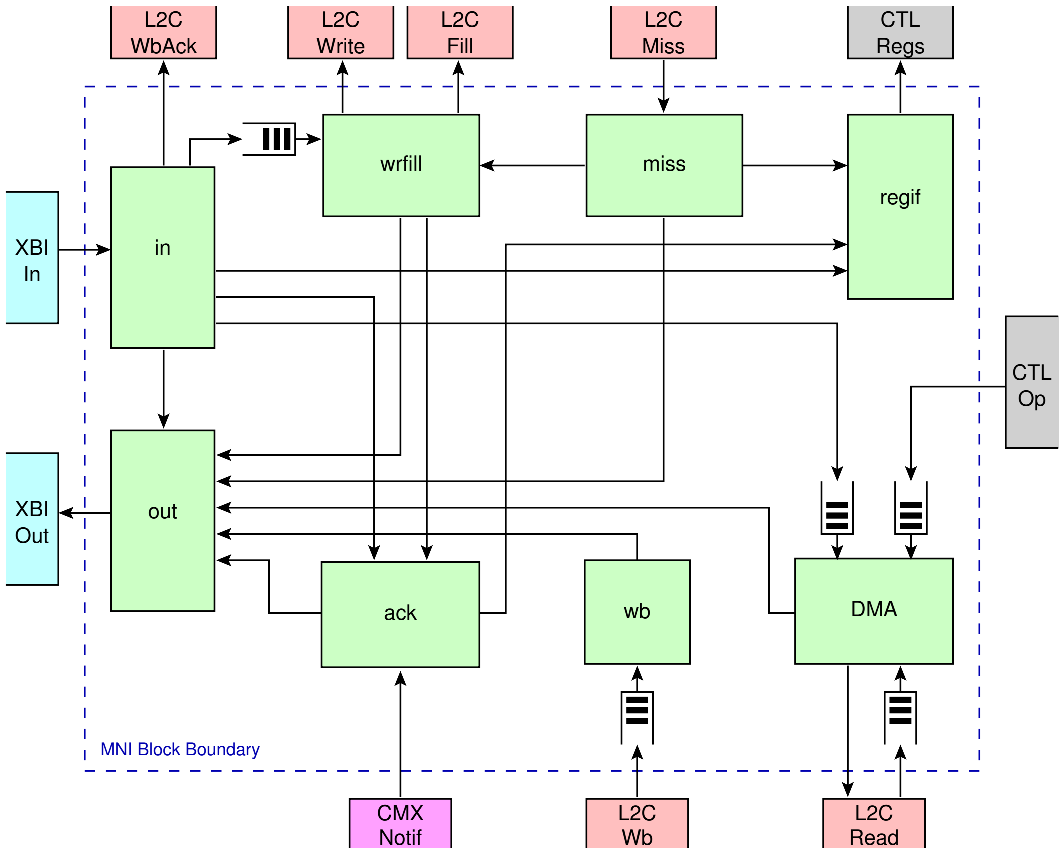


Figure 2.7: MNI block diagram

The MNI block diagram is shown in figure 2.7. It consists of 8 sub-blocks (in, out, wrfill, miss, regif, ack, wb and DMA) as well as the following FIFOs:

Wrfill FIFO: 64x16 bits, distributed memory. Holds an incoming cache line from the network to be replied as a Write or a Fill to the cache, along with its destination address and acknowledgement addresses. Data of this FIFO may be reused to write through the cache line to the DRAM.

Wb FIFO: 32x16 bits, distributed memory. Holds an incoming cache line from L2C that needs to be written back to the DRAM.

Read FIFO: 32x16 bits, distributed memory. Holds an incoming cache line from L2C that was requested from the DMA engine.

DMA CPU operation FIFO: 512x16 bits, half BRAM. Holds up to 32 DMA descriptors provided by the local CPU.

DMA Network operation FIFO: 512x16 bits, half BRAM. Holds up to 32 DMA descriptors provided by the network in the form of incoming read packets (VC #2).

2.5.5 Functionality

We discuss here an overview of the actions that MNI takes, according to the requests that arrive from the network or the L2C, CTL or CMX blocks.

Network-initiated functionality:

Incoming write or ack, destination is a register
: Write the payload to CTL register interface. If acknowledgement is needed, generate an acknowledgement packet (if it is remote) or write it to the CTL register interface (if it is local). Special case is a mailbox write access: if not enough mailbox/mailslot space exists (i_cmx_mbox_space or i_cmx_mslot_space) for the packet, the packet is dropped. If it has an acknowledgement address, generate a Nack (a VC #0 ack packet with payload value of 0).

Incoming ack, destination is not a register
: Inform L2C (WbAck interface) that an acknowledgment has arrived.

Incoming write, destination is not a register
: Push the payload (one word or one cache line) to L2C Write or Fill interface, depending on Opcode[2] bit. If it's an L2C Write, L2C may accept it or not; in either case, depending on Opcode[3] and Opcode[4] bits, the data may have to be sent to the DRAM. If so, generate a VC #1 packet to the DRAM.

If acknowledgment is needed, generate a VC #0 packet (or do a local register access) only if L2C was the final destination of the data; in the case DRAM write through is done, do not generate an acknowledgement but instead copy the acknowledgment address in the VC #1 packet, so that DRAM will generate the acknowledgment itself.

Incoming read, destination is a register
: Read the register value from CTL and generate a response VC #1 packet with this value.

Incoming read, destination is not a register
: Enqueue the read packet to the DMA engine network queue, if it is not full (see below for the DMA functionality). If the queue is full, drop the packet; if it has an acknowledge address, generate a Nack (a VC #0 ack packet with payload value of 0).

L2C-initiated functionality:

Miss Request, I bit set
: Perform a read or a write to the CTL register interface. The request may block; if so, L2C Miss interface is kept blocked until a CTL Unblock happens. If (and when) the request completes and it's a read, the response value must be answered back through the L2C Fill interface.

Miss Request, I bit not set
: Send a VC #2 packet to the DRAM that requests the needed word (if C bit not set) or cache line (C bit set). Set Opcode[2] of this packet to 0, so that the response is forwarded to the Fill interface.

Writeback Request
: Send a VC #1 packet to the DRAM that contains the cache line to be written back. This packet has an acknowledgement address set, so that the DRAM can send back a VC #0 packet which will be forwarded to the L2C WbAck interface.

CTL or Network-initiated DMA functionality:

Two DMA descriptor FIFOs are filled by the incoming network (VC #2 read packets) and the CTL Operation interface (CPU DMA/Message requests). A DMA Engine handles in a round-robin fashion both FIFOs, implementing the following cases:

Source board/node is remote, but not DRAM
A single VC #2 read packet is sent, with destination board/node/address set to the DMA source board/node/address. The remote node will execute the DMA.

Source board/node is local
The DMA is split into cache line segments. For each segment, a request is made through the L2C Read interface. If L2C responds with the cache line, a VC #2 write packet is sent. If L2C does not have the cache line, a VC #2 read packet is sent to the local DRAM for the cache line, with source board/node/address set to the final destination – i.e. the DRAM will forward the packet directly to its final destination. In both cases, if acknowledgement is needed, the related fields of the packet are filled.

Source board/node is (any) DRAM
"Any" DRAM here means that it may be the local board DRAM or a remote board DRAM. In both cases, the DMA is split into cache line segments and handled as specified above, without requesting anything from the L2C – a VC #2 read packet to the DRAM is generated per cache line. This is needed because the DRAM destination (i.e. the TLB block) does not have DMA engine capabilities and cannot split read requests into cache lines.

CMX-initiated functionality:

Counter notifications are handled like an acknowledgment: a VC #0 packet with payload equal to the notification value (0 or 1) is sent to the destination, if the acknowledgement address is remote. If it is local, a CTL register interface write access is made to write the notification value.

2.5.6 Subblock Description

mni_in subblock
The block does not dequeue a packet unless a conservatively estimated space (2 or more packets) exists in needed VCs. When it does, it decodes the incoming packet and as descibed before, it may do the following:

- Make an L2C WbAck access
- Push the payload to the WrFill FIFO, along with dst/ack addresses
- Do a register read access, followed by a VC #1 reply
- Do a register write access, possibly followed by a positive mni_ack access
- Enqueue a network DMA, if space exists in the FIFO
- Send a Nack (zero mni_ack access), if space does not exist.

mni_out subblock
The block simply arbitrates all other block accesses that request permission to send a packet, selects one of them in a round-robin fashion and prepares the packet according to the contents they specify.

mni_wrfill subblock
Depending on Opcode[2], an access to L2C Write or Fill is made. For Fills, the cache always accepts the data and further action is not required (no acknowledgment exists).

For Writes, o_l2c_write_dirty takes the value of (not Opcode[4]). The possible scenaria of L2C-MNI interfaction are the following:

- Cache accepts, Opcode[3] = 0: Send the line to cache. If Opcode[1] do an mni_ack access.
- Cache accepts, Opcode[3] = 1: Send the line to cache, then do an mni_out access to re-send the line to the DRAM. If Opcode[1], copy the ack fields to this packet.
- Cache refuses, Opcode[4] = 0: Do an mni_out access to send the line to the DRAM. If Opcode[1], copy the ack fields to this packet.
- Cache refuses, Opcode[4] = 1: Drop the cache line altogether. If Opcode[1], do an mni_ack access.

mni_miss subblock
For I/O accesses (I bit set), the block does a register read or write access to mni_regif. This may block, in which case mni_miss remains stalled and thus L2C (and thus the CPU) is blocked. When the access completes, if it's a read a mni_wrfill request is done to return the value through the L2C Fill interface.

For non-I/O accesses (I bit not set), an mni_out request is made for a single word (C bit not set) or a whole cache line (C bit set). The packet has Opcode[2]=0 (miss/fill traffic) and no acknowledgment address.

mni_regif subblock
The block does CTL register accesses, arbitrated among mni_miss (a CPU read/write), mni_in (a network read/write) or mni_ack (a local or remote generated acknowledgement write). Writes can be any length, reads are always two 16-bit words.

mni_ack subblock
Arbitrates among mni_in and mni_wrfill requests for acknowledgments. Checks if the acknowledgement is a local register; if it its, it forwards the request to mni_regif. If it's not, it generates a VC #0 packet by doing an mni_out access.

mni_wb subblock
Receives cache lines to be written back from the L2C Wb interface and does an mni_out access to create a VC #1 packet. Opcode[2] is set to 1 (generic traffic) and Opcode[1] is 1 (acknowledgment is needed, address set to the cache address).

mni_dma subblock
Does the actions described in the DMA functionality paragraph (section 2.5.5) above.

2.5.7 Features to be tested

Common cases are not covered here; we assume all basic functionality will be tested in regular scenaria.

MNI-1: Verify DMA operation with source of a remote DRAM

MNI-2: Verify all 4 L2C Write cases (cache accepts/refuses $\times$ Opcode[3 or 4])

MNI-3: Network and CPU FIFO full conditions. Verify Nack generation and CTL exception generation.

MNI-4: Verify behavior under congestion. Check that mni_in, mni_wb, mni_miss, mni_dma and mni_ack can stop independently.

2.6 Control block (CTL)

2.6.1 Purpose

The CTL block:

- Keeps all memory-mapped, architecturally visible registers that control the MBS block
- Implements an interrupt controller for the CPU
- Passes through Mailbox and Counters accesses to CMX
- Initiates Netork Operations to MNI

2.6.2 Pin List

Pin Name	Width	Description
clk_cpu	1	SRAM clock (10 MHz)
clk_mc	1	ART & Caches clock (40 MHz)
clk_ni	1	Network Interface clock (80 MHz)
rst_mc	1	Reset for clk_mc
rst_ni	1	Reset for clk_ni
i_core_id	3	Core ID number
CPU Interface (clk_cpu)		
o_cpu_interrupt	1	Level sensitive interrupt, active high
rst_cpu	1	Reset for clk_cpu
ART Interface (clk_mc)		
o_art_entry0_base	12	Entry #0 base address (12 MSB)
o_art_entry0_end	12	Entry #0 end address (12 MSB)
o_art_entry0_flags	6	Entry #0 flags: C, I, R, X, U, P
o_art_entry0_valid	1	Entry #0 valid
o_art_entry1_base	12	Entry #1 base address (12 MSB)
o_art_entry1_end	12	Entry #1 end address (12 MSB)
o_art_entry1_flags	6	Entry #1 flags: C, I, R, X, U, P
o_art_entry1_valid	1	Entry #1 valid
o_art_entry2_base	12	Entry #2 base address (12 MSB)
o_art_entry2_end	12	Entry #2 end address (12 MSB)
o_art_entry2_flags	6	Entry #2 flags: C, I, R, X, U, P
o_art_entry2_valid	1	Entry #2 valid
o_art_entry3_base	12	Entry #3 base address (12 MSB)
o_art_entry3_end	12	Entry #3 end address (12 MSB)
o_art_entry3_flags	6	Entry #3 flags: C, I, R, X, U, P
o_art_entry3_valid	1	Entry #3 valid
o_art_privileged	1	CPU in privileged mode
i_art_perm_fault	1	Permission fault detected
i_art_miss_fault	1	ART address not found fault detected
i_art_tlb_fault	1	TLB address not found fault detected
o_art_fault_ack	1	Fault acknowledged
IL1 Interface (clk_mc)		
o_il1_en	1	Cache enabled
o_il1_clear_req	1	Clear operation request
i_il1_clear_ack	1	Clear operation completed
DL1 Interface (clk_mc)		
o_dl1_en	1	Cache enabled

Pin Name	Width	Description
o_dl1_clear_req	1	Clear operation request
i_dl1_clear_ack	1	Clear operation completed
L2C Interface (clk_mc)		
o_l2c_en	1	Cache enabled
o_l2c_clear_req	1	Clear operation request
o_l2c_flush_req	1	Flush operation request
i_l2c_maint_ack	1	Clear/flush operation(s) completed
o_l2c_epoch	3	Current task epoch number
o_l2c_min_cpu_ways	3	Minimum ways reserved for CPU per epoch
MNI Register Access Interface (clk_ni)		
i_mni_reg_valid	1	New register access valid
i_mni_reg_from_cpu	1	When 1, access originates from CPU When 0, access originates from Network
i_mni_reg_adr	20	Register offset address (2B-aligned)
i_mni_reg_wen	1	When 1, access is write
i_mni_reg_ben	2	Byte enables
i_mni_reg_wdata	16	MNI access data to be written
i_mni_reg_rd_len	3	Read length in 16b words (0=1w, 7=8w)
o_mni_reg_stall	1	When 1, no more requests can be handled
o_mni_reg_resp_valid	1	Response valid
o_mni_reg_resp_rdata	16	Response data read
o_mni_reg_resp_block	1	Response incomplete, block initiator
o_mni_reg_resp_unblock	1	Pending response is now carried out
MNI Operation Interface (clk_ni)		
o_mni_op_valid	1	New operation valid
o_mni_op_data	16	Operation data
i_mni_op_stall	1	When 1, stop sending operations
i_mni_available_stat	7	How many more operations can be fired
i_mni_pending_stat	7	How many operations are pending
CMX Interface (clk_ni)		
o_cmx_valid	1	New counter/mailbox request valid
o_cmx_opcode	4	Operation opcode: 0000: Counter write 0001: Counter read 0010: Counter increment 1000: Mailbox write 1001: Mailbox read 1011: Mailbox depth read 1100: Mailslot write 1101: Mailslot read 1111: Mailslot depth read
o_cmx_rd_len	3	Read length in 16b words (0=1w, 7=8w) For mbox reads, this should be 1 (=2w)
o_cmx_cnt_adr	9	Counter adr (6 MSB) + offset (3 LSB)
o_cmx_wdata	16	Data to be written or incremented
i_cmx_stall	1	When 1, no more requests can be handled
i_cmx_resp_rdata	16	Response data read
i_cmx_resp_valid	1	Response valid
i_cmx_resp_block	1	Response incomplete, block reader
i_cmx_resp_unblock	1	Pending response is now carried out
i_cmx_resp_mbox_full	1	Mailbox/mailslot full, write ignored
i_cmx_int_mbox	1	Mailbox/mailslot just got non-empty
i_cmx_int_cnt	1	Counter triggerd interrupt
i_cmx_int_cnt_adr	6	Address of counter that triggered

Pin Name	Width	Description

Table 2.5: CTL Pin List

2.6.3 Functionality

Registers

CTL keeps all the architecturally visible registers that are accessible within a Microblaze Slice block. The length, addresses, bit fields and purpose of these registers will not be replicated here – please refer to section 4.2 on page 66 for a complete description.

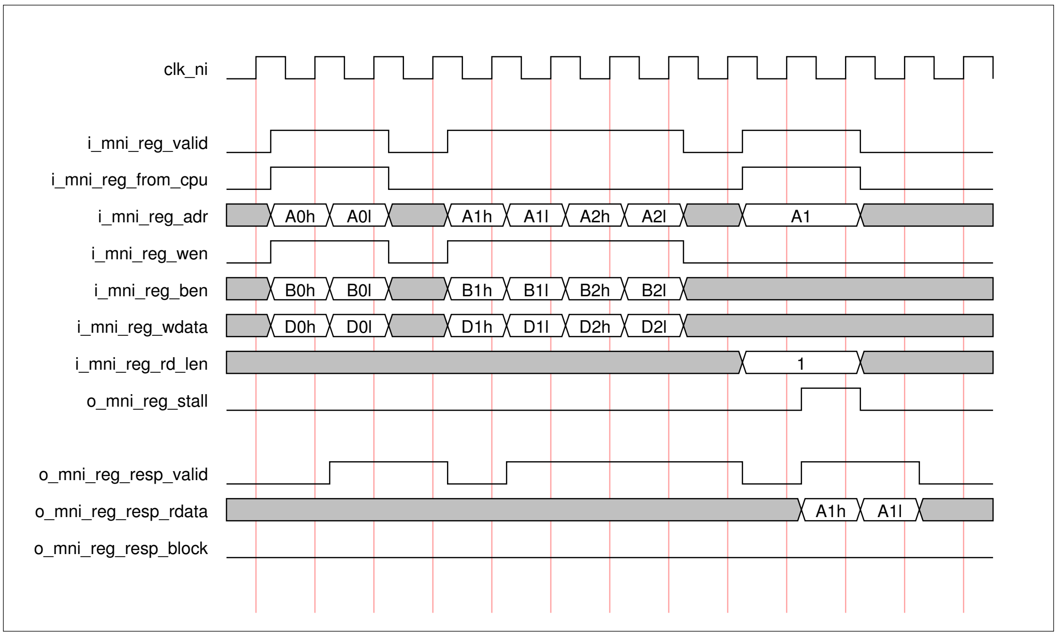


Figure 2.8: Register Access (Normal case, not Mailbox or Counter)

All the registers are accessed through the MNI-CTL register access interface. Figure 2.8 shows the basic functionality for the common case, which is an access to a "normal" or interrupt-related registers; this excludes Mailbox and Counters, which are handled by the CMX block and must be forwarded to it. In the figure, first the CPU writes one 32-bit register: this is translated to two 16-bit back-to-back write accesses. Then, the network writes in a single batch two registers (4 16-bit accesses), still back-to-back as they may originate from a single network packet. Note that all these addresses are 16-bit aligned, so they specify whether to write the bottom half or the top half of the register.

Finally, a read request comes for a 32-bit register. As a convention, the read address is the one of the lower half (32-bit aligned) but the response must always be high-first, low-second. Multiple read bursts are handled in the same manner.

Many registers (e.g. all ART registers) or bit fields (e.g. like L1 and L2 enable bits) are simply written and read through the MNI-CTL interface and their value is broadcasted statically to blocks. Other registers have side-effects when being written to or read from, or they need special treatment. These will be explained in the following sections.

Interrupts

For each Interrupt source specified in the CPU Status Register (section 4.2.1, page 69) there is a corresponding Clear bit. If the Interrupt is maskable, there is also a corresponding Mask bit.

Maskable interrupts are considered a valid source to trigger an actual Interrupt to the CPU only when their Mask bit is set to 1. Non-maskable interrupts can always trigger an actual Interrupt. Note that when the CPU Status Register is read it must show the status of all interrupt sources, regardless if they are masked or not.

Whenever any valid source for an interrupt exists, CTL asserts o_cpu_interrupt and keeps it high. Eventually the interrupt handler will procceed to clear the source of the interrupt by writing 1 to the corresponding Clear bit (writing 0 must be ignored, because the CPU does not have *bit* enables). This interrupt source must not be considered valid anymore. Note that if any other source of interrupts is still active, o_cpu_interrupt must remain high: this will force the interrupt handler to run again as soon as it exits until the next source is cleared.

The sources of the interrupts are considered the following conditions:

System Call: Activated when the CPU System Call Entry Register bit S is written with any value (section 4.2.1, page 67).

TLB Miss: Activated when ART asserts i_art_tlb_fault.

ART Miss: Activated when ART asserts i_art_miss_fault.

Permission Fault: Activated when ART asserts i_art_perm_fault.

Mailbox/mailslot Full: Activated when CMX responds with i_cmx_resp_mbox_full.

Network Exception: Activated when i_mni_op_stall is activated but a new Network Operation has arrived and needs to be sent to MNI.

Software Interrupt: Activated when the CPU Control Register I bit is written with 1 (section 4.2.1, page 68).

Mailbox/mailslot Interrupt: Activated when CMX asserts i_cmx_int_mbox.

Counter Interrupts: There are four Counters associated with Interrupts, specified in the Counter Interrupt register (section 4.2.7, page 105). When CMX asserts i_cmx_int_cnt, these four counter addresses must be checked towards i_cmx_int_cnt_adr. If anything hits, the corresponding of the four Counter Interrupts is activated.

Note that ART faults must be acknowledged and this interface is on another clock domain (clk_mc). Also, the CPU is on another clock domain (clk_cpu). An example timing diagram is shown in figure 2.9, where a Permission Fault is caused by ART and the CPU (at a much later time than shown in the figure) writes the Clear bit of the Permission Fault. The interrupt in this case will be deasserted at the next clk_cpu positive edge, assuming that no other interrupt causes were active.

Cache Maintenance Operations

Level 1 and Level 2 caches support a Clear operation which invalidates all entries. The Level 2 cache also supports a Flush operation which writes back the dirty lines. All these operations must be seen as synchronous to the CPU: the register accesses that write these fields and trigger these operations must not return until the operations are finished.

This behaviour is shown in figure 2.10, where a Data Level 1 Cache Clear is performed. Notice that all interface signals to the cache belong to the clk_mc clock domain. Initially, the CPU writes the D bit in the Cache Maintenance register (section 4.2.3, page 78). This write must not return to the CPU and for this reason o_mni_reg_resp_block is asserted as soon as access to the C bit is performed. The cache maintenance is communicated to the DL1 block

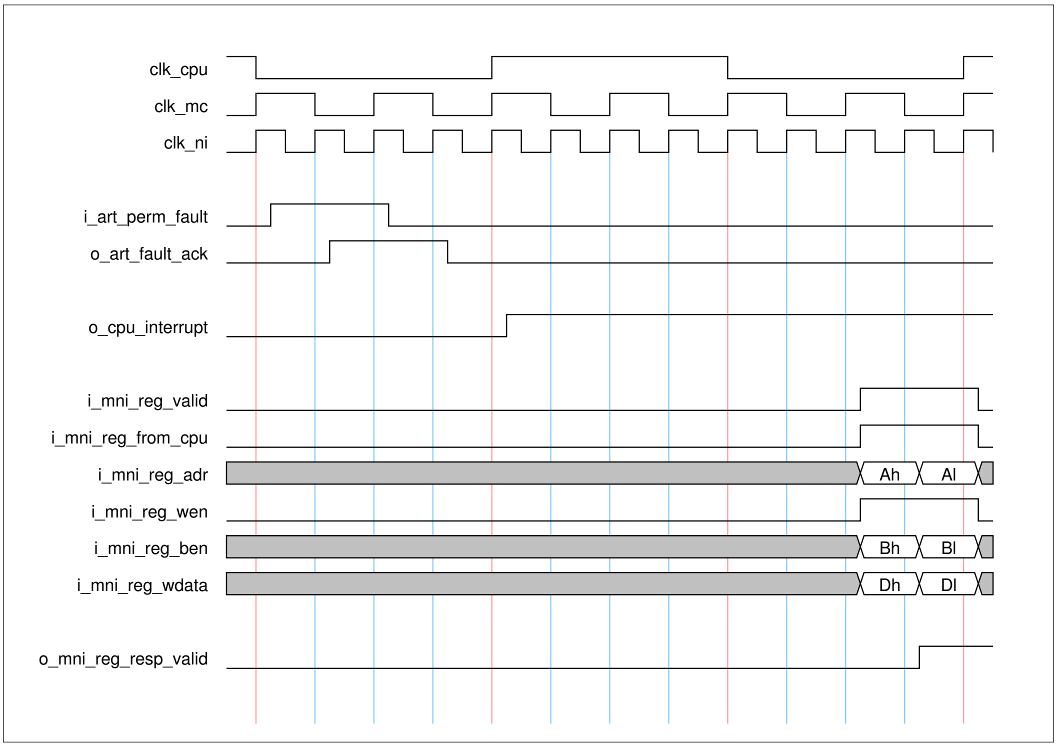


Figure 2.9: CTL Interrupt caused by ART

and at some point later in time DL1 responds that the operation has been completed. CTL must assert a response to unblock the CPU by asserting the o_mni_reg_resp_unblock signal as show in the figure.

Warning: although the CPU will be blocked for this period of time, the MNI is free to issue new register accesses at any time. CTL must make sure that the o_mni_reg_resp_valid will not get mixed with other requests by asserting the o_mni_reg_stall signal.

Processor Enable & Core ID

The read-only CPU Status register (section 4.2.1, page 69) also contains a CID field, standing for CPU Core ID. This must be served by the static i_core_id signal which differentiates the MBS instances from one another.

On a related topic, there is a CPU enable bit in the CPU Control register (section 4.2.1, page 68). When the CPU is disabled, its reset signal must be active. CTL is responsible to generate rst_cpu based on the E field of the CPU Control register.

Counter and Mailbox Access

If the register access address falls into the Counter or Mailbox address space, the following cases apply:

Counter Interrupt Register: This is a normal register kept inside CTL (section 4.2.7, page 105).

Counters #0 . . . #127 Registers: An access to a specific Counter is made, according to the register addresses specified in section 4.2.7, page 110. This must be forwarded to the CMX block. Bits [10:4] of i_mni_reg_adr specify which of the 128 Counters must be addressed. Figures 2.12 and 2.13 (page 35) show the protocol that must be followed.

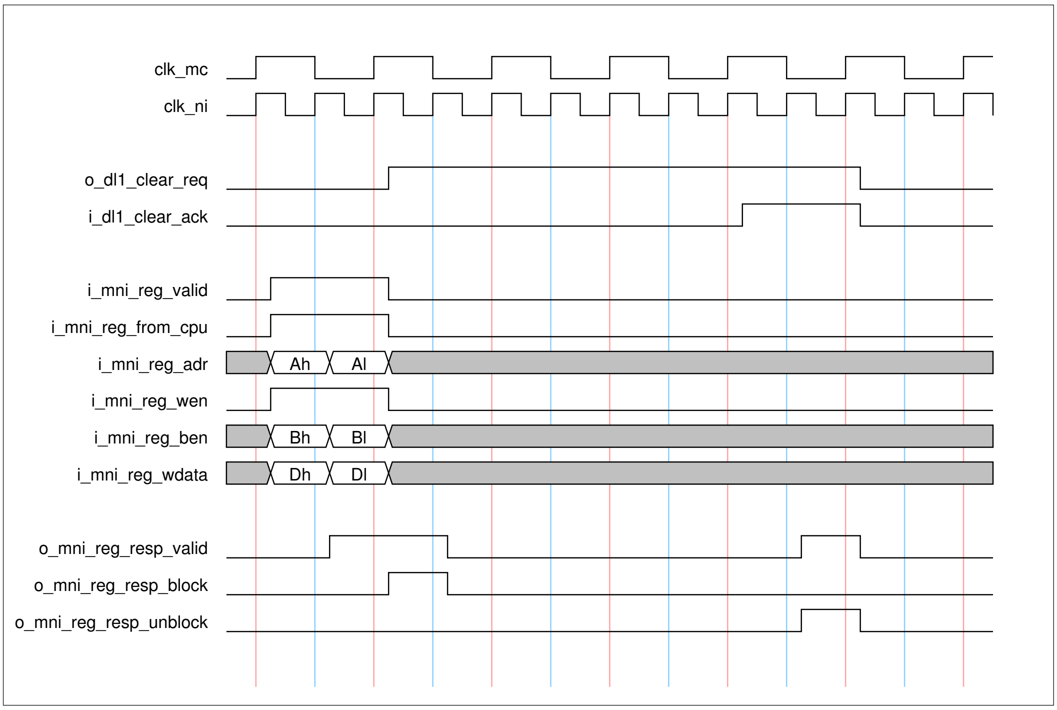


Figure 2.10: CPU orders a Data Level 1 Cache Clear operation

Mailbox Access Register: An access to the Mailbox is done (4.2.8, page 111). This must be forwarded to the CMX block. Figures 2.14 and 2.15 (page 36) show the protocol that must be followed. If a Mailbox read responds with a block, this block must be transfered back to the MNI block in the register access interface, in a similar manner as was shown for the cache maintenance operations.

Note that an even number of reads and writes are always expected from the register access interface.

MNI Operation Interface

CTL keeps a set of registers for the Network Interface. Their use is explained in section 4.2.6 (page 94). All of them are normal registers, except the MNI Status Register which read-only. Its fields, AVAILABLE and PENDING are mapped directly to the i_mni_available_stat and i_mni_pending_stat signals.

Furthermore, a write into the OP field in the MNI Opcode Register has a side-effect: it triggers a new Network Operation that must be sent through the MNI Operation Interface. Remember that software guarantees that space must always exist in this interface: if i_mni_op_stall is high, the new operation is discarded and a Network Exception is raised, as discussed previously in the Interrupts section.

Figure 2.11 shows the normal case where the CPU triggers a new operation. Notice that the register access is complete (o_mni_reg_resp_valid is asserted for the CPU access) but the o_mni_reg_stall is asserted: no more register accesses will be made until the MNI operation is finished, in order to simplify corner cases.

The MNI operation is always 16 clock cycles long. It dumps the eight 32-bit registers (section 4.2.6) in order. For each register, first the upper half and then the lower half is transmitted.

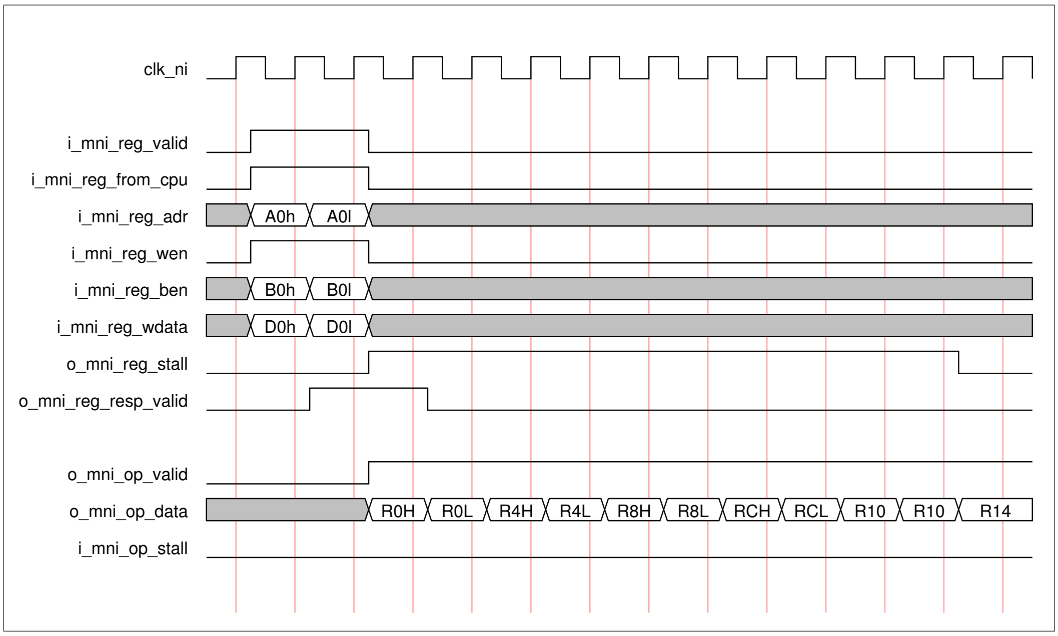


Figure 2.11: CPU triggers a new MNI operation

2.6.4 Features to be tested

Features to be verified	Test case
Masked interrupts and Status bits	
Multiple interrupts occuring together	
Cache blocking together with an MNI access arbitration while a CTL unblock must happen	
Blocking mailbox read; also check with MNI access arbitration while an unblock happens (back-to-back packets; one writing to the Mailbox which causes the unblock and the next one trying to access a register).	

Table 2.6: CTL Test Coverage Plan

2.7 Counters & Mailbox block (CMX)

2.7.1 Purpose

The CMX block:

- Holds 128 arhitecturally visible counters that can be used to track DMA progress
- Implements a 4-KB mailbox, which can be written from the network side and read from the CPU side
- Implements a separate, 1-word mailslot to facilitate remote register reads

2.7.2 Pin List

Pin Name	Width	Description
clk_ni	1	Network Interface clock (80 MHz)
rst_ni	1	Reset for clk_ni
CTL Interface		
i_cpu_interrupt	1	CPU under interrupt, abort blocking
i_ctl_valid	1	New counter/mailbox request valid
i_ctl_opcode	3	Operation opcode: 0000: Counter write 0001: Counter read 0010: Counter increment 1000: Mailbox write 1001: Mailbox read 1011: Mailbox depth read 1100: Mailslot write 1101: Mailslot read 1111: Mailslot depth read
i_ctl_rd_len	3	Read length in 16b words (0=1w, 7=8w) For mbox reads, this should be 1 (=2w)
i_ctl_cnt_adr	10	Counter adr (7 MSB) + offset (3 LSB)
i_ctl_wdata	16	Data to be written or incremented
o_ctl_block_aborted	1	When 1, blocking aborted
o_ctl_stall	1	When 1, no more requests can be handled
o_ctl_resp_rdata	16	Response data read
o_ctl_resp_valid	1	Response valid
o_ctl_resp_block	1	Response incomplete, block reader
o_ctl_resp_unblock	1	Pending response is now carried out
o_ctl_int_mbox	1	Mailbox/mailslot just got non-empty
o_ctl_int_cnt	1	Counter triggerd interrupt
o_ctl_int_cnt_adr	6	Address of counter that triggered
MNI Interface		
o_mni_valid	1	Request for a new NI notification
o_mni_data	16	Notification board/node/address
i_mni_stall	1	When 1, no more requests can be handled
o_mni_mbox_space	12	Mailbox occupancy
o_mni_mslot_space	1	Mailslot occupancy

Table 2.7: CMX Pin List

2.7.3 Functionality

The memories inside CMX are organized as follows:

- 2 BRAMs (4 KB) used for Mailbox and 1 BRAM (2 KB) for the Counters.
- Counters: current value (32b) + ack adr 0 (32b) + (ack board ID 0 (8b) + ack node ID 0 (8b)) + ack adr 1 (32b) + (ack board ID 1 (8b) + ack node ID 1 (8b)) = 8 x 16b words. We allocate 8 words per counter, thus 128 counters can be implemented.
- Mailbox: 2048 x 16b words are available. But, since CPU always accesses things in 32b alignment, we always use the mailbox doing multiples of 2 x 16b accesses.
- Mailslot: an independent, single 32b word second mailbox, useful for remote reads (e.g. for reading a remote register)

The Counters architectural functionality is explained in the programmer's model (page 106) in detail. Figures 2.12 and 2.13 show how the CTL block interfaces the CMX block in order to initialize, increment or read a counter value. The second figure also shows a CMX trigger event where the board/node notifications are sent over to the MNI block.

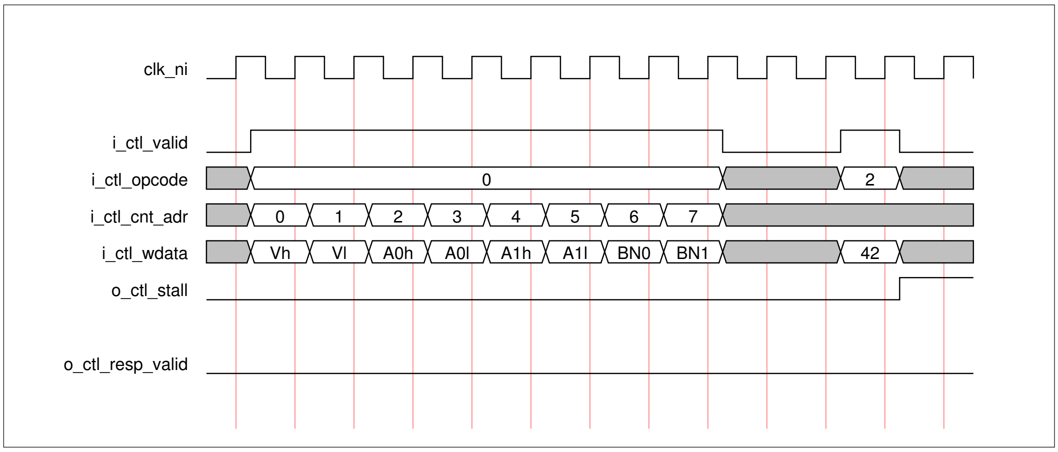


Figure 2.12: Counter Initialization and Increment

Upon a mailbox/mailslot write request from CTL, the incoming data is written in the related memory. If no space exists, no data will be allowed to come from the MNI block (o_mni_*_space signals specify how much memory is available for write operations).

A normal read and write operation from the CTL is shown in figure 2.14.

When the CPU tries to read from an empty mailbox or mailslot, this causes the CMX to reply with a "block" event, which will in turn cause the CTL block to stall the MNI access indefinitely and thus effectively block the CPU. When a new write happens through MNI and CTL, CMX must "unblock" and complete the pending access with the start of the newly written data, which will in turn cause the CTL block to complete the pending MNI access and will resume the CPU blocked data access. This is shown in figure 2.15.

A complication exists for the case the CPU receives an interrupt (i_cpu_interrupt). In this case, the blocked access must immediately unblock the CTL/MNI paths, using dummy data (e.g. value 0).

Whenever a counter triggers, apart from the MNI notification, possibly an interrupt must be generated from the CTL block. CMX always asserts o_ctl_int_cnt in this case to specify that a counter has triggered to CTL — which may, in turn, generate an interrupt if the programmer has unmasked it. Also, when the mailbox or the mailslot just becomes non-empty, this is a possible interrupt cause and o_ctl_int_mbox is asserted. Figure 2.16 depicts all this.

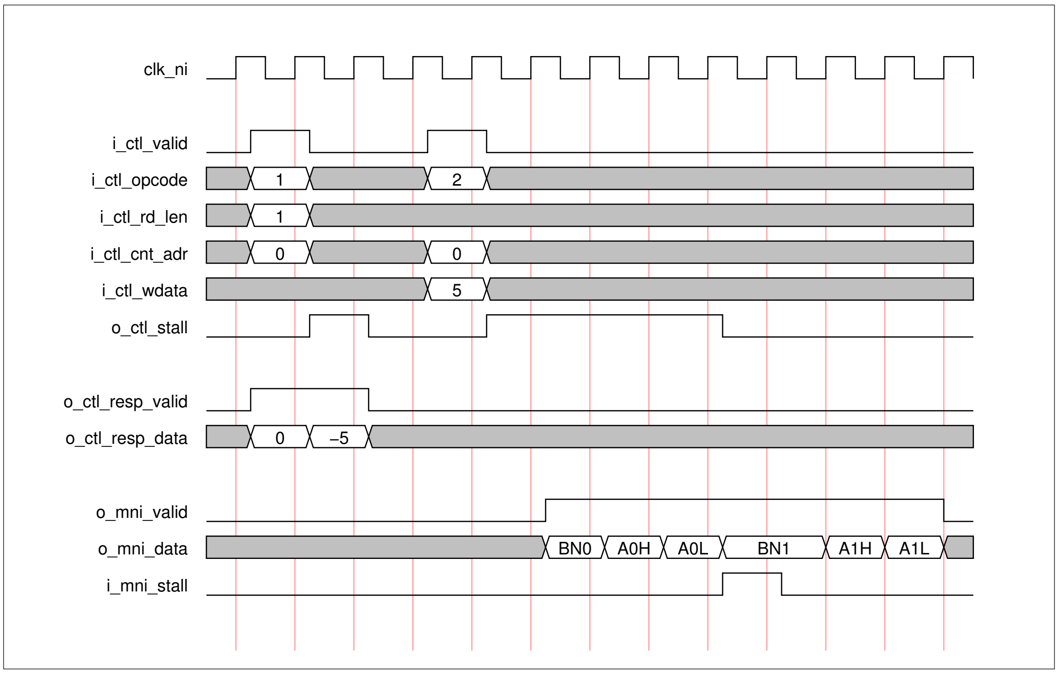


Figure 2.13: Counter Read, Increment and Trigger

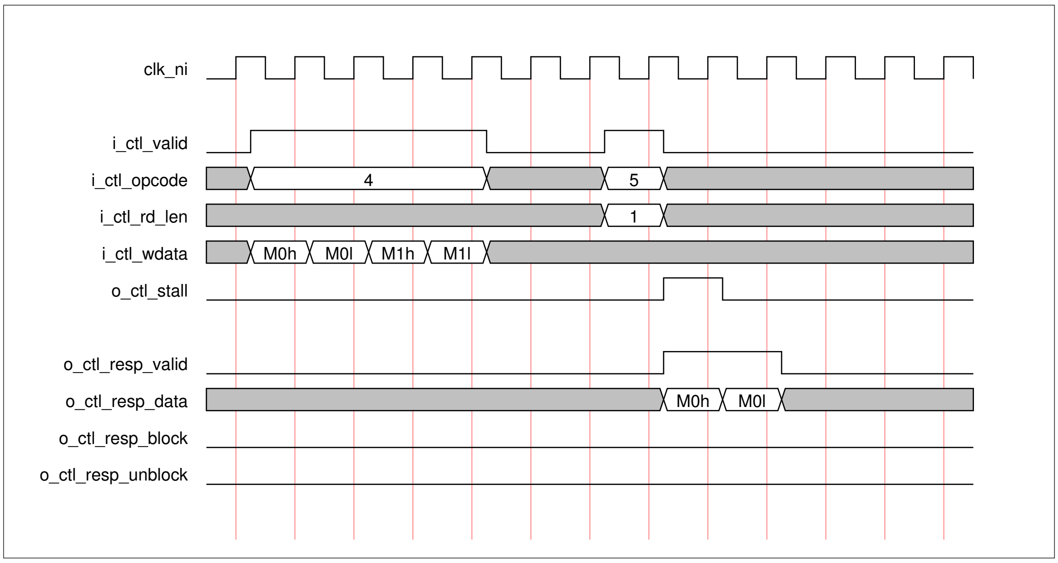


Figure 2.14: Mailbox Normal Write and Read

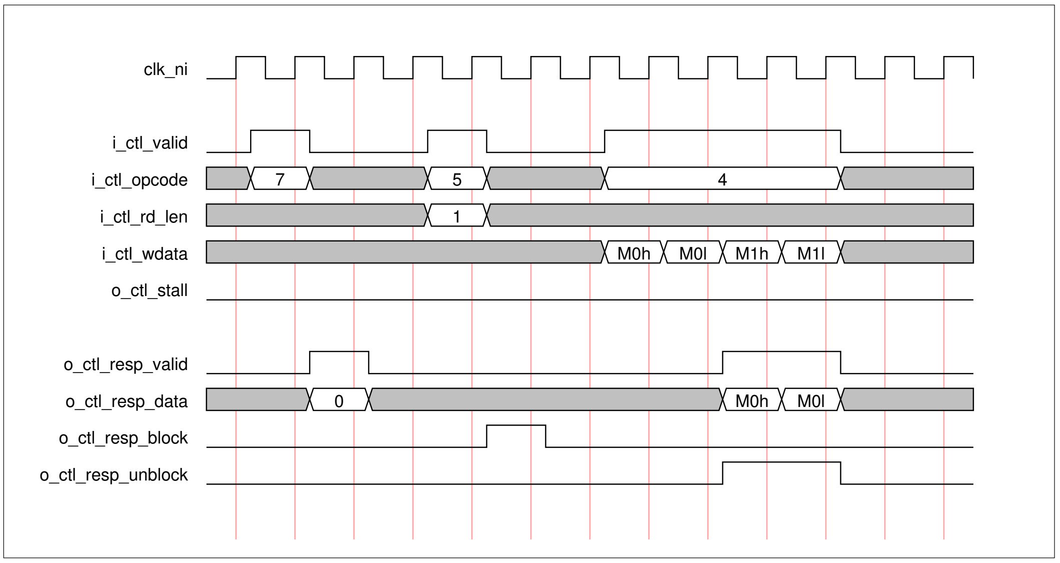


Figure 2.15: Mailbox Blocking Read and Unblock

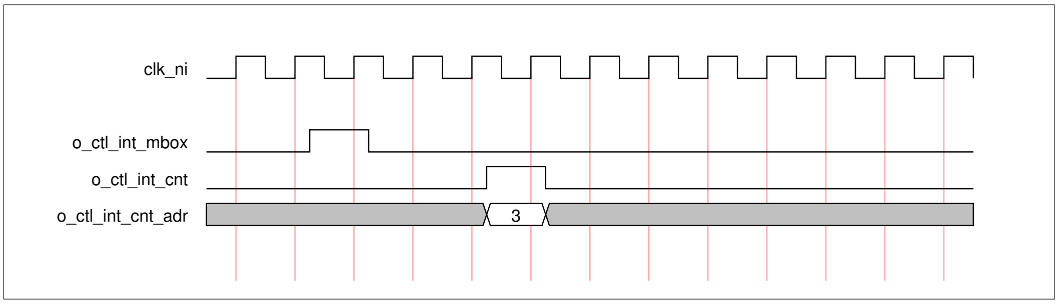


Figure 2.16: Mailbox and Counter Interrupt

Chapter 3

Formic Non-MBS Blocks Description

3.1 Crossbar Interface block (XBI)

3.1.1 Purpose

The XBI block:

- Provides 6 virtual FIFOs for communication with the Crossbar: 3 for each incoming VC and 3 for each outgoing VC.
- Utilizes only 2 BRAM blocks (one for incoming traffic and one for outgoing). Each BRAM is separated statically to hold 6 packets per VC.
- Allows each side to access any location inside the packet that is currently enqueued or dequeued.
- Provides a completely registered in/out interface to the Crossbar to facilitate FPGA routing.
- When data link is idle in the Crossbar input direction, provides an automatic peeking of the destination board/node IDs among all eligible VCs in order to minimize overhead to the input scheduling which is caused by the registered interfaces.

3.1.2 Pin List

Pin Name	Width	Description
User port (clk_usr in general, clk_ni when used inside the MBS block)		
clk_usr	1	User clock (MBS: clk_ni = 80 MHz)
rst_usr	1	Reset for user clock
i_usr_nout_enq	3	Enqueue (per VC bitmap)
i_usr_nout_offset	8	Offset in packet (in 16-b words)
i_usr_nout_eop	1	End of packet, send it to destination
i_usr_nout_data	8	Data to be written to offset
o_usr_nout_full	3	FIFO full (per VC bitmap)
o_usr_nout_packets_vc0	3	Space left in VC #0 FIFO (packets)
o_usr_nout_packets_vc1	3	Space left in VC #1 FIFO (packets)
o_usr_nout_packets_vc2	3	Space left in VC #2 FIFO (packets)

Pin Name	Width	Description
i_usr_nin_deq	3	Dequeue (per VC bitmap)
i_usr_nin_offset	8	Offset in packet (in 16-b words)
i_usr_nin_eop	1	End of packet, proceed to next one
o_usr_nin_data	8	Data read from offset
o_usr_nin_empty	3	FIFO empty (per VC bitmap)
o_usr_nin_packets_vc0	3	Packets arrived in VC #0 FIFO
o_usr_nin_packets_vc1	3	Packets arrived in VC #1 FIFO
o_usr_nin_packets_vc2	3	Packets arrived in VC #2 FIFO
Crossbar port, registered inputs & outputs (clk_xbar)		
clk_xbar	1	Crossbar clock, 160 MHz
rst_xbar	1	Reset for clk_xbar
i_xbar_out_enq	3	Enqueue (per VC bitmap)
i_xbar_out_offset	8	Offset in packet (in 16-b words)
i_xbar_out_eop	1	End of packet, send it to destination
i_xbar_out_data	8	Data to be written to offset
o_xbar_out_full	3	FIFO full (per VC bitmap)
o_xbar_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
o_xbar_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
o_xbar_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
i_xbar_in_deq	3	Dequeue (per VC bitmap)
i_xbar_in_offset	8	Offset in packet (in 16-b words)
i_xbar_in_eop	1	End of packet, proceed to next one
o_xbar_in_data	8	Data read from offset or packet destination peek when idle
o_xbar_in_empty	3	FIFO empty (per VC bitmap)

Table 3.1: XBI Pin List

3.1.3 Functionality

XBI contains two 1024x16 BRAM blocks, each one of them organized as seen in figure 3.1. Six packets per each of the three VCs are held. The maximum packet size is always assumed. For VC #0 (used for Acknowledges) this is set to 12 16-bit words, whereas for VCs #1 and #2 (used for Data requests and responses) this is set to 64 16-bit words. The choice of 12 and 64 was made to minimize the hardware area of the multiplication.[1] For each direction one such BRAM block is used, so the total buffering provided by XBI is 36 packets (18 per direction, or 6 per VC per direction).

The block is organized around the two BRAM blocks and six FIFO pointer blocks. Figure 3.2 shows a block diagram of how each BRAM block is connected to three FIFO pointer blocks. One pointer block synchronizes the enqueue and dequeue requests from each side for a single VC. The FIFOs recognize six positions and are used only when a packet is ready to depart (from the write side) or when it has been read and is not needed anymore (from the read side): each FIFO updates its pointers when the end-of-packet is given from the input. Until that time, any packet word can be accessed from the BRAM by giving the appropriate word offset.

Internally, a FIFO pointer block is synchronizing a write pointer and a read pointer (which can each take values 0-5) to be used as if the read and write clock domains are asynchronous, which can be true for some cases of the XBI block usage (such as when synchronizing the input of a GTP link to the Crossbar). The logic is gray-pointer based and its principle is shown in figure 3.3.

[1] The number 12 contains two 1s when written in binary and thus multiplying any quantity with them can be done with only two additions.

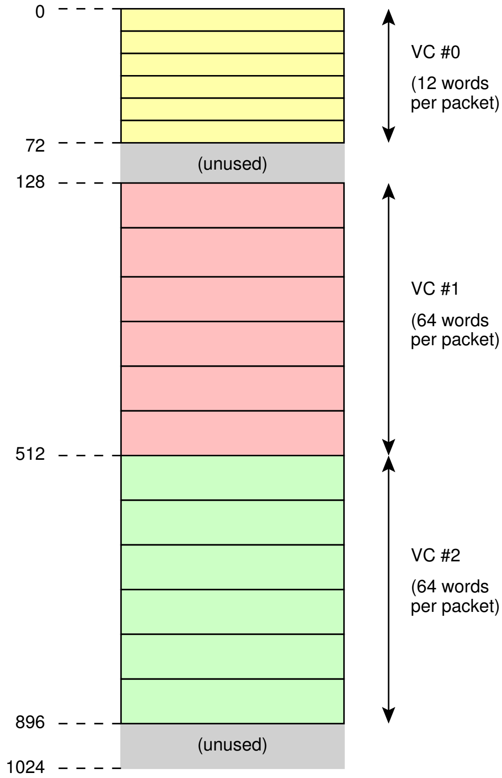


Figure 3.1: XBI BRAM memory organization

A special gray code encoding of six positions is used here. All possible values are encoded in three bits and the transitions are:

$000 \rightarrow 001 \rightarrow 011 \rightarrow 010 \rightarrow 110 \rightarrow 100$

which ensure that always a single bit changes from a position to the next one.

Binary pointer positions are converted to this gray encoding before being synchronized to be used in the other clock domain. As soon as the pointers are synchronized, they are converted back to binary in order to compute the write-read pointers difference. As 6 positions do not fit into 3 bits exactly, the subtraction might be adjusted by 2 when wrapping around in order to provide the correct packets and full/empty estimations.

The base address is calculated as the VC base address (0, 128 and 512 for VCs 0-2 respectively) plus the packet size (12, 64 and 64 respectively) times the current pointer value.

3.1.4 Interfaces

The User port and the Crossbar port have the same interface behaviour, with the following exceptions:

- The user port does not have registered inputs and the BRAM output is not registered. The crossbar port registers all inputs and outputs, as well as the BRAM output.
- The crossbar input port, when idle, will eagerly cycle through the non-empty VCs and push the second packet word out, which contains the destination board ID and destination node ID.

Figures 3.4 and 3.5 show examples of crossbar-side enqueueing and dequeueing. Note that 2 additional cycles of delay appear between the input signals and the output signals due to the extra registers. These 2 cycles are not present in the user-side interfaces.

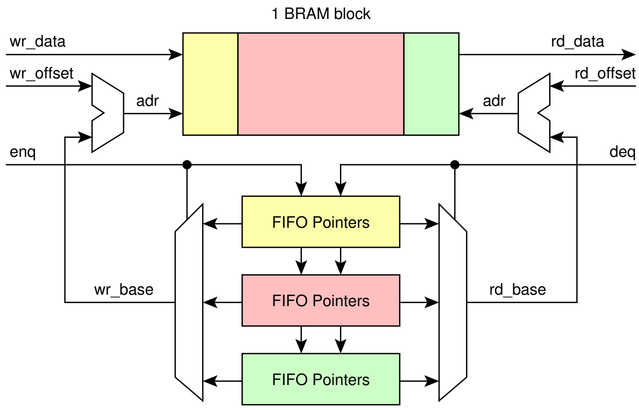


Figure 3.2: XBI block diagram for one of the two directions

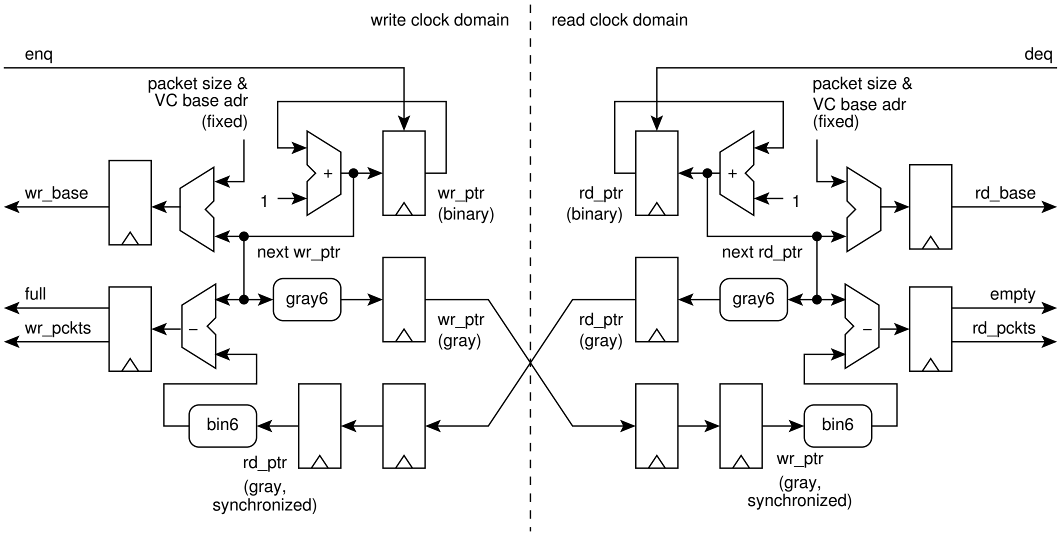


Figure 3.3: XBI FIFO pointer block internal structure

Enqueuing and dequeuing can happen in any offset order, as seen in these examples. When the respective enqueue/dequeue bit and the end-of-packet is asserted, the FIFO proceeds to the next packet and updates the packet count and full/empty flags accordingly.

Note that when the interface is idle in figure 3.5, as soon as a new packet arrives in VC #0 its destination (D0) is sent. As soon as a new packet arrives in VC #1, both destinations (D0 and D1) are cycled in round robin.

In this situation, the data word consists of the following fields: bits 14 to 12 are the one-hot encoding of the VC that is being sent, bits 11 to 4 are the desination board ID and bits 3 to 0 are the destination node ID.

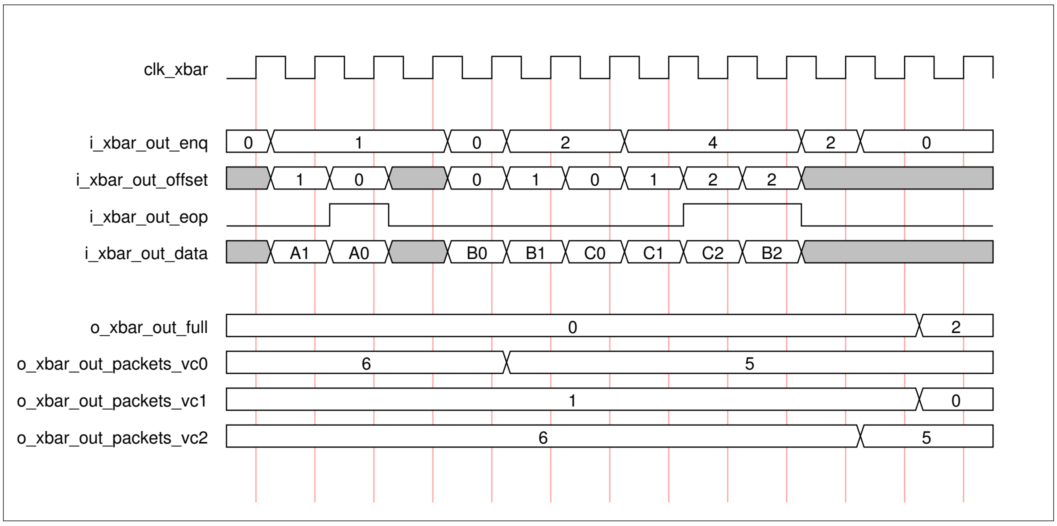


Figure 3.4: Crossbar side enqueueing

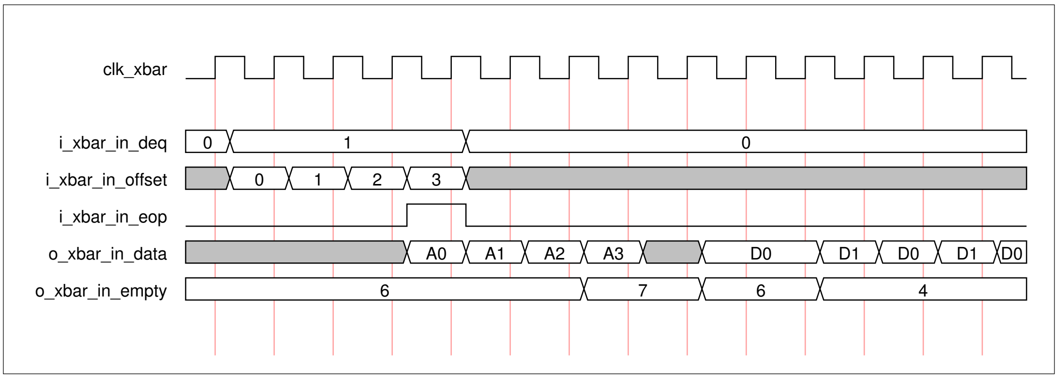


Figure 3.5: Crossbar side dequeueing

3.2 Crossbar (XBAR)

3.2.1 Purpose

- A full 22x22 crossbar switch, operating at 160 MHz using 16-bit wide links, supporting 3 VCs.
- Provides the intra-FPGA connectivity among all blocks, input/output ports and memory controllers
- The XBAR block is in itself unbuffered, but it is always interfaced by 22 respective XBI blocks, which buffer 6 packets per VC for each crossbar input and output port.

3.2.2 Pin List

Pin Name	Width	Description
clk_xbar	1	Crossbar clock, 160 MHz
rst_xbar	1	Reset for crossbar clock
i_board_id	8	Local Board ID
Port #00 XBI interface		
o_port00_out_enq	3	Enqueue (per VC bitmap)
o_port00_out_offset	8	Offset in packet (in 16-b words)
o_port00_out_eop	1	End of packet, proceed to next one
o_port00_out_data	8	Data to be written to offset
i_port00_out_full	3	FIFO full (per VC bitmap)
i_port00_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
i_port00_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
i_port00_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
o_port00_in_deq	3	Dequeue (per VC bitmap)
o_port00_in_offset	8	Offset in packet (in 16-b words)
o_port00_in_eop	1	End of packet, proceed to next one
i_port00_in_data	8	Data read from offset
i_port00_in_empty	3	FIFO empty (per VC bitmap)
Port #01 XBI interface		
...		...
...		...
...		...
Port #21 XBI interface		
o_port21_out_enq	3	Enqueue (per VC bitmap)
o_port21_out_offset	8	Offset in packet (in 16-b words)
o_port21_out_eop	1	End of packet, proceed to next one
o_port21_out_data	8	Data to be written to offset
i_port21_out_full	3	FIFO full (per VC bitmap)
i_port21_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
i_port21_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
i_port21_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
o_port21_in_deq	3	Dequeue (per VC bitmap)
o_port21_in_offset	8	Offset in packet (in 16-b words)
o_port21_in_eop	1	End of packet, proceed to next one
i_port21_in_data	8	Data read from offset
i_port21_in_empty	3	FIFO empty (per VC bitmap)

Table 3.2: XBAR Pin List

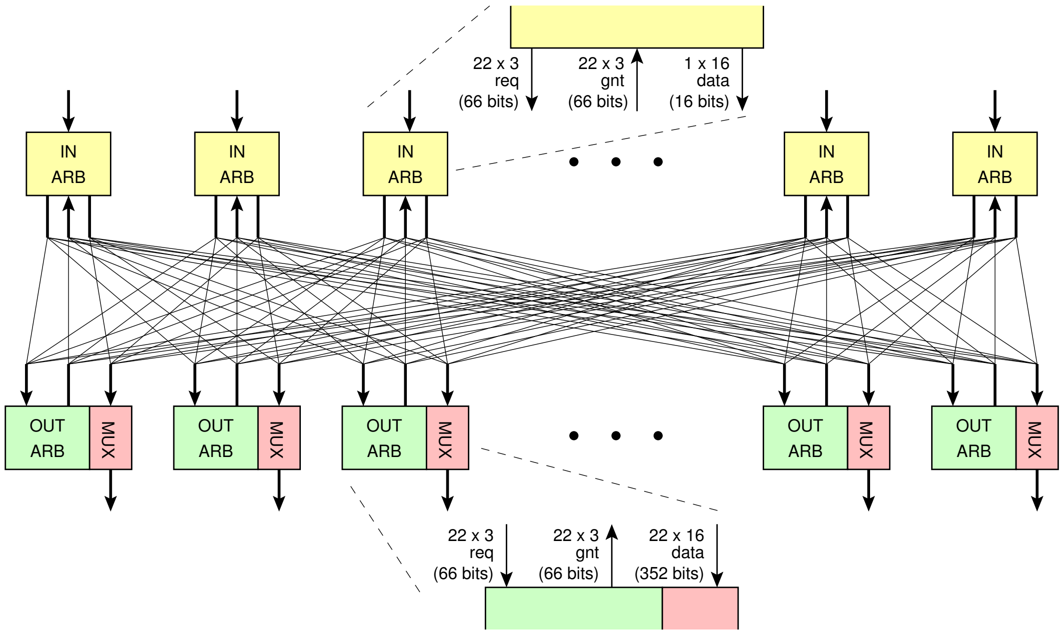


Figure 3.6: XBAR block diagram

3.2.3 Functionality

Figure 3.6 shows a block diagram of the 22-port full Crossbar. There are three blocks, each instantiated 22 times: an Input Arbiter, an Output Arbiter and a switching fabric Multiplexer. All blocks are heavily pipelined; all have registered inputs and outputs, which on one hand increase considerably the arbitration and communication latency but on the other hand make possible the repeatedly successful placing and routing on a Xilinx Spartan-6 at a speed of 160 MHz.

The crossbar implements a typical Request-Grant-Accept distributed scheduling algorithm. Each input arbiter peeks the destination board and node for each of the up to three non-empty VCs in its input port – the XBI interface provides this peeking in an efficient manner. When the input arbiter remains idle, it computes the destination port for each for the three packets and asserts a request to the respective output arbiter. Thus, up to three requests (one per VC) may be active by any input arbiter.

Each output arbiter receives 66 requests (22 input arbiters times 3 VCs). To make the timing at 160 MHz, the 66 requests are broken into 9 groups as follows:

Group	**Requests**
0	VC #0, Requests 00-07
1	VC #0, Requests 08-15
2	VC #0, Requests 16-21
3	VC #1, Requests 00-07
4	VC #1, Requests 08-15
5	VC #1, Requests 16-21
6	VC #2, Requests 00-07
7	VC #2, Requests 08-15
8	VC #2, Requests 16-21

The selected group enters a pipeline where a round-robin priority enforcer selects a request. When a winning request is found, a grant signal is asserted to the respective input arbiter.

The input arbiter waits for any of its (up to three) grant signals to be asserted. In order to economize on wiring, no dedicated Accept wires exist. Instead, the convention dictates that the input arbiter deasserts its (up to two) non-granted requests but keeps its granted request asserted for one more clock cycle. In the case of simultaneous grants, the input arbiter also keeps a round-robin priority enforcer in order to avoid VC starvation in mostly-empty crossbars.

The output arbiter keeps track of clock cycles elapsed since its grant. The computed latency of communication is 5 clock cycles: if on the 5^{th} clock cycle after a grant the input arbiter has deasserted its request, it means that the input arbiter has selected another output arbiter and that the local output arbiter must continue the search for a new request, starting from the exact previous point (same group out of the nine where it was before and at the same position on the priority enforcer out of the eight selected requests). Otherwise, if on the 5^{th} clock cycle the request is still selected, it means that the local output arbiter was selected.

At this point, the output multiplexer is set to receive the selected input arbiter while the input arbiter begins to send the new packet. There is an additional 3 clock cycle latency until the first word of the packet arrives at the output port, having passed through the 3-stage pipelined multiplexer.

In order to allow a high utilization on the crossbar outputs, 13 cycles before the current packet finishes exiting through the output port, the output arbiter starts searching for a new input arbiter request. The 13 cycles are computed to be the exact best-case latency to undergo a complete negotiation and start outputing the first word of the new packet back-to-back after the last word of the old packet.[2] This also means that small packets with total size less than 13 words cannot be served back-to-back.

3.2.4 Routing function

$\mathbf{X}_d \overset{?}{>} \mathbf{X}_s$	$\mathbf{Y}_d \overset{?}{>} \mathbf{Y}_s$	$\mathbf{Z}_d \overset{?}{>} \mathbf{Z}_s$	$\mathbf{W}_d \overset{?}{>} \mathbf{W}_s$	**Node**$_d$	**XBAR port**	**Physical block**
$>$	x	x	x	x	15	GTP x+1
$<$	x	x	x	x	12	GTP x-1
$=$	$>$	x	x	x	8	GTP y+1
$=$	$<$	x	x	x	14	GTP y-1
$=$	$=$	$>$	x	x	10	GTP z+1
$=$	$=$	$<$	x	x	11	GTP z-1
$=$	$=$	$=$	$\neq$	x	9	GTP w+1
$=$	$=$	$=$	$=$	0	0	MBS #0
$=$	$=$	$=$	$=$	1	1	MBS #1
$=$	$=$	$=$	$=$	2	2	MBS #2
$=$	$=$	$=$	$=$	3	3	MBS #3
$=$	$=$	$=$	$=$	4	4	MBS #4
$=$	$=$	$=$	$=$	5	5	MBS #5
$=$	$=$	$=$	$=$	6	6	MBS #6
$=$	$=$	$=$	$=$	7	7	MBS #7
$=$	$=$	$=$	$=$	12	16-20	Random TLB port
$=$	$=$	$=$	$=$	15	21	BCTL

Table 3.3: XBAR routing function

Table 3.3 shows how the input arbiters compute the crossbar destination port given the destination Board ID and Node ID.

The 6 least significant bits of the board IDs are used as follows: board_id[5:4] is the X dimension of the boards 3D mesh, board_id[3:2] is the Y dimension and board_id[1:0] is the Z dimension. This configuration allows 4 possible values of X, Y and Z (0-3), i.e. for 64

[2] This can be expected when the output port is contended and there are requests for it all over the 9 request groups. In the worst case, where a single request exists in the exactly previous group than the current one, 8 more cycles will be added to the latency.

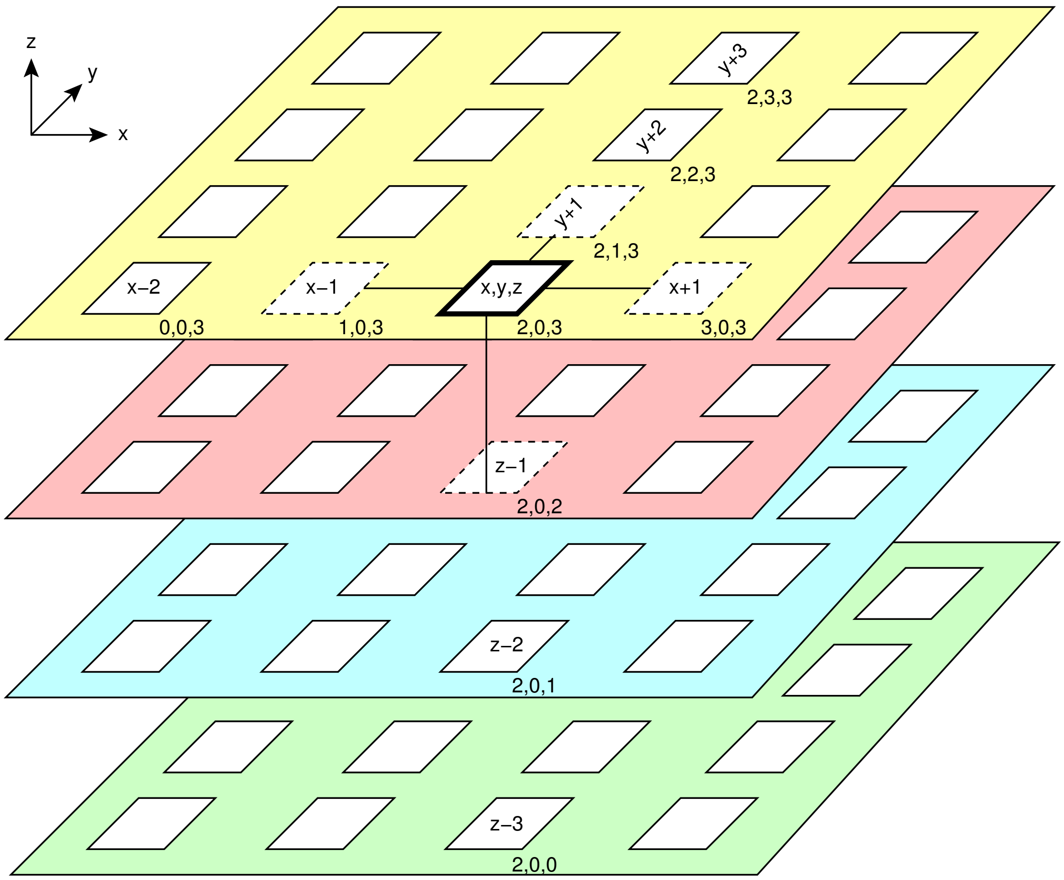


Figure 3.7: 64-board 3D-Mesh

(4×4×4) Formic boards. A seventh bit, board_id[6] is used as a "W" dimension, to allow for connecting a few Versatile boards to the Formic mesh. All Formic boards have W=0 and all Versatile boards have W=1.

The assumed topology of an up to 64-board configuration of Formic boards is depicted in figure 3.7. All boards are connected to their neighbours in the X, Y and Z axis using two GTP links for each dimension, unless they are in the periphery (X/Y/Z values of 0 or 3) where no connection is made[3]. In the figure, the board with values X=2, Y=0 and Z=3 is shown to be connected to its 2 X neighbors, its single Y neighbor and its single Z neighbor. These are accessible through a single hop. To access, for example, the board of 2,0,0 (labeled "z-3") three hops are required through boards 2,0,2 and 2,0,1 and their respective XBAR blocks.

The Versatile boards are connected to a few designated points in this 3D-mesh, with X/Y/Z settings equal to the Formic boards they connect to but with the W bit set to 1.

The routing function presented in table 3.3. In the table, X_d represents the X destination board ID, X_s the X source board ID — the same applies for Y, Z and W. An "x" denotes a don't care value. We avoid deadlock by routing first always the X hop of the board ID, followed by the Y hop, followed by the Z hop and lastly followed by the W hop. If no board-to-board hops remain, we then route internally to the respective crossbar port based on the node ID.

To balance the DDR2 DRAM traffic, five ports are dedicated for the TLB/DRAM interface. Each input arbiter selects a round-robin TLB port whenever the packet destination is node 12 (the TLB node). This selection happens independently from all other input arbiters, which in practice creates a relatively uniform memory traffic across the five TLB ports.

[3] We do not use links to create a 3D-torus, because this would require additional VCs to resolve circular dependencies deadlocks.

3.3 SRAM Controller block (SRAM_CTL)

3.3.1 Purpose

The SRAM_CTL block has the following responsibilities:

- Drives a single SRAM chip at 160 MHz and handles the tristate data bus
- Serves 4 interfaces to 4 respective MBS L2 caches, each clocked at 40 MHz

3.3.2 Pin List

Pin Name	Width	Description
clk_sram	1	SRAM clock (160 MHz)
clk_mc	1	ART & Caches clock (40 MHz)
rst_sram	1	Reset for clk_sram
rst_mc	1	Reset for clk_mc
SRAM Interface		
o_sram_adr	18	Word address
o_sram_bw_n	4	Byte enables (active low)
o_sram_we_n	1	Write enable (active low)
io_sram_dq	32	Bidirectional data bus
MBS #0 Interface		
i_mbs0_req_adr	18	Word address
i_mbs0_req_we	1	Write enable
i_mbs0_req_wdata	32	Data to be written
i_mbs0_req_be	4	Byte enables
i_mbs0_req_valid	1	Request valid
o_mbs0_resp_rdata	32	Response data read from SRAM
o_mbs0_resp_valid	1	Response valid
MBS #1 Interface		
i_mbs1_req_adr	18	Word address
i_mbs1_req_we	1	Write enable
i_mbs1_req_wdata	32	Data to be written
i_mbs1_req_be	4	Byte enables
i_mbs1_req_valid	1	Request valid
o_mbs1_resp_rdata	32	Response data read from SRAM
o_mbs1_resp_valid	1	Response valid
MBS #2 Interface		
i_mbs2_req_adr	18	Word address
i_mbs2_req_we	1	Write enable
i_mbs2_req_wdata	32	Data to be written
i_mbs2_req_be	4	Byte enables
i_mbs2_req_valid	1	Request valid
o_mbs2_resp_rdata	32	Response data read from SRAM
o_mbs2_resp_valid	1	Response valid
MBS #3 Interface		
i_mbs3_req_adr	18	Word address
i_mbs3_req_we	1	Write enable
i_mbs3_req_wdata	32	Data to be written
i_mbs3_req_be	4	Byte enables
i_mbs3_req_valid	1	Request valid
o_mbs3_resp_rdata	32	Response data read from SRAM

Pin Name	Width	Description
o_mbs3_resp_valid	1	Response valid

Table 3.4: SRAM_CTL Pin List

3.3.3 Functionality

The SRAM controller implements a standard ZBT 3-stage pipeline to drive the Cypress SRAM chips. For more information, please refer to the datasheet of the Cypress CY7C1354CV25 SRAM chip.

On the inside, the 4 MBS blocks are interfaced at the $\frac{1}{4}$ of the SRAM clock rate. The two clocks have aligned clock edges. Thus, a static round-robin scheduling can be applied so that each of the 4 MBS blocks can operate at full bandwidth independently of the other 3 blocks.

3.4 Translation Lookaside Buffer (TLB)

3.4.1 Purpose

The TLB block:

- Provides a full 4K-entries mapping of virtual to physical addresses, with a page size of 1MB
- Supports burst writes and reads of up to 64 bytes per packet
- Links the 5 crossbar ports of aggregate bandwidth 12.8 Gbps (5 ports x 16 bits x 160 MHz) to the 4 ports of the Xilinx DDR2 DRAM controller, maintaining the same bandwidth of 12.8 Gbps (4 ports x 32 bits x 100 MHz).

3.4.2 Pin List

Pin Name	Width	Description
clk_xbar	1	Crossbar clock (160 MHz)
clk_ddr	1	DDR clock (100 MHz)
rst_xbar	1	Reset for clk_xbar
rst_ddr	1	Reset for clk_ddr
Board control TLB maintenance interface (clk_ddr)		
i_bctl_tlb_enabled	1	When 0, TLB is in bypass mode
i_bctl_maint_cmd	1	Maintenance command active
i_bctl_virt_adr	12	Virtual address entry
i_bctl_phys_adr	7	Physical address translation
i_bctl_entry_valid	1	Address entry is valid when 1
o_bctl_drop	5	Packet dropped (per crossbar port)
Crossbar interface #0 (clk_xbar)		
i_xbar0_out_enq	3	Enqueue (per VC bitmap)
i_xbar0_out_offset	8	Offset in packet (in 16-b words)
i_xbar0_out_eop	1	End of packet, send it to destination
i_xbar0_out_data	16	Data to be written to offset
o_xbar0_out_full	3	FIFO full (per VC bitmap)
o_xbar0_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
o_xbar0_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
o_xbar0_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
i_xbar0_in_deq	3	Dequeue (per VC bitmap)
i_xbar0_in_offset	8	Offset in packet (in 16-b words)
i_xbar0_in_eop	1	End of packet, proceed to next one
o_xbar0_in_data	16	Data read from offset or packet destination peek when idle
o_xbar0_in_empty	3	FIFO empty (per VC bitmap)
Crossbar interfaces #1-#4 (clk_xbar)		
...		...
...		...
...		...
DDR controller port #0 (clk_ddr)		
o_ddr0_cmd_en	1	New command valid
o_ddr0_cmd_instr	3	Bit 2: refresh when 1
		Bit 1: auto precharge when 1
		Bit 0: write when 0, read when 1

Pin Name	Width	Description
o_ddr0_cmd_bl	6	Burst length (0=1 word)
o_ddr0_cmd_byte_addr	30	Byte address
i_ddr0_cmd_empty	1	Command FIFO empty
i_ddr0_cmd_full	1	Command FIFO full
o_ddr0_wr_en	1	Write data enable
o_ddr0_wr_data	32	Write data
o_ddr0_wr_mask	4	Byte enable mask (1=don't write)
i_ddr0_wr_almost_full	1	Write FIFO almost full
o_ddr0_rd_en	1	Read data enable
i_ddr0_rd_data	32	Read data
i_ddr0_rd_empty	1	Read FIFO empty
DDR controller port #1-#3 (clk_ddr)		
...		...
...		...
...		...

Table 3.5: TLB Pin List

3.4.3 Functionality

A block diagram of the TLB block is shown in figure 3.8. It consists of 5 XBI blocks (described in section 3.1, page 39), each of which is connected to a small network interface (TLB_NI in the figure). The network interface blocks translate the input packets into Xilinx DDR controller-compatible requests. A small 5x4 crossbar connects each NI block to an available DDR controller port. If a response is needed, the NI creates an output packet and sends it back to its XBI port.

All TLB_NI subbblocks use a shared address translation mechanism at the start of their access. This mechanism is built around a 4096x8 memory (2 BRAM blocks of 2048x8), which translates the virtual address page number – virtual address bits 31:20 – to a physical page number. Since we have only 128 MB of DRAM memory, i.e. 128 pages of 1 MB, the resulting physical page address is only 7 bits. The eighth bit is used for the page validity info. All possible 4,096 pages are fully contained into the TLB and thus TLB misses do not exist. The BRAM block is two-ported. One port is reserved for the shared reads and the second one is used by the Board Controller interface to install TLB entries.

The small 5x4 crossbar is scheduled using two round-robin priority enforcers. The 5 TLB_NI requests are handled by the first enforcer; the 4 DDR ports non-busy status are handled by the second enforcer. Whenever both enforcers have a valid output, a match is found: the TLB_NI port is paired with the DDR port, all switching fabric multiplexers are updated to establish a bidirectional connection between them, and the DDR port is marked as busy. When the TLB_NI finishes its operation, it notifies the crossbar scheduler which frees the DDR port.

The crossbar scheduler serves a single request per clk_ddr clock cycle, in the round-robin order among TLB_NI subblocks as mentioned above. The same schedule applies to the sharing of the address translation: the TLB_NI asserts its request signal along with a requested virtual address. When a grant is signalled by the crossbar scheduler it also controls the BRAM multiplexer to translated the virtual address to a physical one. The TLB_NI gets the translation always exactly 1 clock cycle after the grant signal.

TLB_NI Functionality

TLB_NI serves requests from the incoming XBI in either VC #1 (a write data packet that goes into the memory) or VC #2 (a read packet that will read from the memory). In the

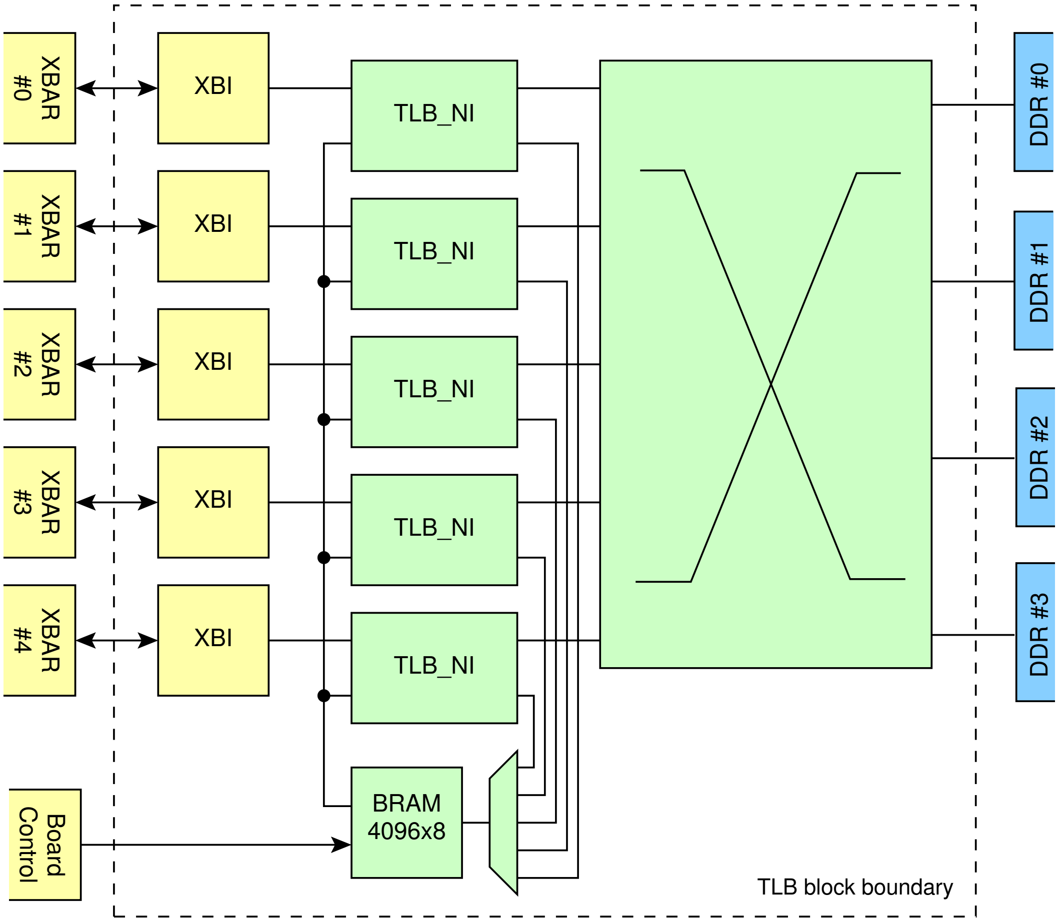


Figure 3.8: TLB block diagram

first case, a packet is served only when outgoing VC #0 has at least one packet of space left, because an acknowledge packet might be generated. In the second case, a packet is served only when outgoing VC #1 has at least on packet of space left, because a write packet must be generated. If all conditions are true, packets from incoming VCs #1 and #2 are served in a round-robin fashion.

The block detects some basic packet errors, like a wrong VC in the header or a wrong requested size. These packets are dropped and the Board Controller block is notified.

When the related header fields have been processed and the virtual address is retrieved, TLB_NI makes a request to the 5x4 crossbar scheduler. As soon as a grant is replied, the physical address translation is also present. In the case of an inexistent TLB page (i.e. when the page valid bit is 0), an empty reply packet is generated with the "TLB fault" bit in the header set to 1. In the case of a read packet, this reply is a write in the VC #1. In the case of a write packet, the reply is an acknowledge in the VC #0, but only if an acknowledge was requested by the original write packet; if not, the write is silently ignored (but still the Board Controller block is notified of this drop).

Write packets are handled by writing the payload to the DDR controller and then triggering a write command to it. Inversely, read packets first trigger a read command to the DDR controller and when the DDR replies, the outgoing packet payload is filled by reading the response data from it.

Special care is paid to write packets with acknowledges. In order to avoid races among the multiple TLB ports and the multiple DDR ports, an acknowledged write will return the acknowledge packet only when the command FIFO of the DDR port is completely *emptied*. This ensures that the DRAM has begun writing the payload at the moment that the acknowledge packet is being generated.

Additional latency

The DDR2 DRAM operates at a very fast rate (400 MHz DDR). Although the bandwidth is shared among four ports, it still makes a single DRAM access from an MBS CPU to be complete in a very short time (20-25 clk_cpu cycles), which is unrealistic. The TLB block implements a timestamp-based mechanism to introduce additional latency per DRAM operation, in order to create more realistic delays.

Specifically, a clk_cpu timer counts continuously. Whenever a new request enters any of the five ports, the timer value (called a "timestamp") is enqueued in a special 8x16 FIFO. Ten such FIFOs exist, one per port and possible input VC ($5 \times 2 = 10$).

When TLB_NI processes the requests, it will not consider any of the available ones to be eligible until the timestamp for this port and VC is above a predefined clock cycle overhead. When the clk_cpu timer reaches the correct time, the request will become eligible and will be dequeued from the small FIFO together with the request packet.

3.5 Gigabit Transceiver Port (GTP)

3.5.1 Purpose

The GTP block:

- Transmits/receives network packets by interfacing the on-FPGA crossbar with an off-FPGA Xilinx Gigabit Transceiver Port back-end
- Provides flow control between peers using credit-based messages
- Verifies the integrity of the off-FPGA links using a CRC-16 algorithm

To correspond directly to GTP back-end FPGA locations, the GTP block is organized in two quads for Formic Spartan-6 FPGAs (4 “top” GTP links and 4 “bottom”), and in a dual pair for Versatile Virtex-5 FPGAs (just 2 GTP links).

3.5.2 Pin List

Pin Name	Width	Description
clk_xbar	1	Crossbar clock (160 MHz)
clk_gtp	1	GTP clock (150 MHz)
rst_xbar	1	Reset for clk_xbar
rst_gtp	1	Reset for clk_gtp
rst_ni	1	Reset for clk_ni
rst_mc	1	Reset for clk_mc
Crossbar interface for port 0 (clk_xbar)		
i_xbar0_out_enq	3	Enqueue (per VC bitmap)
i_xbar0_out_offset	8	Offset in packet (in 16-b words)
i_xbar0_out_eop	1	End of packet, send it to destination
i_xbar0_out_data	16	Data to be written to offset
o_xbar0_out_full	3	FIFO full (per VC bitmap)
o_xbar0_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
o_xbar0_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
o_xbar0_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
i_xbar0_in_deq	3	Dequeue (per VC bitmap)
i_xbar0_in_offset	8	Offset in packet (in 16-b words)
i_xbar0_in_eop	1	End of packet, proceed to next one
o_xbar0_in_data	16	Data read from offset or packet destination peek when idle
o_xbar0_in_empty	3	FIFO empty (per VC bitmap)
Crossbar interface for ports 1, 2 & 3. (clk_xbar)		
...		...
...		...
...		...
GTP physical port 0 (asynchronous)		
i_gtp0_rx_p	1	Receive pin (positive)
i_gtp0_rx_n	1	Receive pin (negative)
o_gtp0_tx_p	1	Transmit pin (positive)
o_gtp0_tx_n	1	Transmit pin (negative)
GTP physical ports 1, 2 & 3. (asynchronous)		
...		...
...		...

Pin Name	Width	Description
...		...
Formic BCTL status (asynchronous)		
o_link_up	4	Link Up (1 bit per port)
o_link_error	4	Link physical error (1 bit per port)
o_credit_error	4	Credit packet error (1 bit per port)
o_crc_error	4	CRC decoding error (1 bit per port)

Table 3.6: GTP Pin List (quad version – encompasses 4 GTP links)

3.5.3 Functionality

The crossbar-part of the functionality is provided by instantiating one XBI block instance per GTP link.

Packets are output from the XBI queue onto the link whenever they are eligible. Each of the three VCs is eligible as soon as a full packet is available in the XBI queue and as long as the credit-based flow control has not asserted an Xoff for this VC (see below). When a VC is eligible, the packet is sent word-by-word (at 16 bits), while a CRC-16 polynomial is being computed. After the last packet word, the 16-bit CRC-16 checksum is appended.

The CRC-16 polynomial used is the IBM standard one:

$$x^{16} + x^{15} + x^2 + 1$$

Packets are input from the GTP link into the XBI queue by writing the incoming words one by one and computing the CRC-16 of the incoming data. At the last word, the computed CRC is compared to the incoming CRC from the link; if they match, the XBI packet is enqueued to the incoming VC; otherwise, the packet is discarded (no XBI eop is asserted and the offset begins again from 0 for the next packet) and a CRC error is asserted to the board controller block.

Whenever the crossbar dequeues a packet from the XBI, a credit is sent back over the link to the connected peer. Credits are single 16-b words that are transmitted as separate packets among the normal, data-carrying packets. The GTP layer keeps track of how many credits are available for packets at the other side of the link. Initially, a number of credits equal to the XBI capability (i.e. six packets per VC) minus a safety margin dictated by the link latency, the packet maximum length and the clock period (we use two packets safety) is given for each VC. When this number of transmitted packets is reached, a “transmit Off” (Xoff) event happens in the GTP link and no more packets can be received, until more credit packets are received from the peer which notify that the other side crossbar has begun to retrieve packets from the remote XBI block.

3.6 Formic Board Controller block (BCTL)

3.6.1 Purpose

The Formic BCTL block:

- Holds the board-wide architecturally visible registers
- Coordinates booting and software-initiated reset procedures
- Provides control and receives status over the non-MBS blocks
- Controls the board LEDs and does pulse-width modulation for them
- Connects the I^{2}C and UART blocks to the on-FPGA network

3.6.2 Pin List

Pin Name	Width	Description
clk_xbar	1	Crossbar clock (160 MHz)
clk_ddr	1	DDR clock (100 MHz)
clk_ni	1	Network interface clock (80 MHz)
clk_mc	1	Cache clock (40 MHz)
clk_cpu	1	CPU clock (10 MHz)
rst_xbar	1	Reset for clk_xbar
rst_ddr	1	Reset for clk_ddr
rst_ni	1	Reset for clk_ni
rst_mc	1	Reset for clk_mc
Static configuration		
i_board_id	8	Board ID dip switches
i_dip_sw	8	Remaining dip switches
Reset manager interface (asynchronous)		
i_ddr_boot_req	1	Request to boot, after a hard reset
o_ddr_boot_done	1	Set when the DRAM boot code is written
o_rst_soft	1	When 1, soft reset will be done
o_rst_hard	1	When 1, hard reset will be done
Board LEDs (asynchronous)		
o_led	12	Twelve LED on-off values
GTP interface (asynchronous)		
i_gtp_link_up	8	GTP Link Up status bits
i_gtp_link_error	8	GTP Link error status bits
i_gtp_credit_error	8	GTP Credit error status bits
i_gtp_crc_error	8	GTP CRC error status bits
TLB maintenance interface (clk_ddr)		
o_tlb_enabled	1	When 0, TLB is in bypass mode
o_tlb_maint_cmd	1	Maintenance command active
o_tlb_maint_wr_en	1	Maintenance command is write
o_tlb_virt_adr	12	Virtual address entry
o_tlb_phys_adr	7	Physical address translation (writes)
o_tlb_entry_valid	1	Address entry valid when 1 (writes)
i_tlb_phys_adr	7	Physical address translation (reads)
i_tlb_entry_valid	1	Address entry valid when 1 (reads)
i_tlb_drop	5	Packet dropped (per crossbar port)
DDR Controller interface (clk_ddr)		

Pin Name	Width	Description
i_ddr_p0_error	1	When 1, DDR port #0 got an error
i_ddr_p1_error	1	When 1, DDR port #1 got an error
i_ddr_p2_error	1	When 1, DDR port #2 got an error
i_ddr_p3_error	1	When 1, DDR port #3 got an error
I2C Slave interface (clk_cpu)		
i_i2c_miss_valid	1	Command valid
i_i2c_miss_adr	8	Register address
i_i2c_miss_flags	2	Miss flags (unused)
i_i2c_miss_wen	1	Command is write
i_i2c_miss_ben	4	Miss byte enables (set to 1)
i_i2c_miss_wdata	32	Data to be written
o_l2c_miss_stall	1	When 1, no more commands can be given
o_i2c_fill_valid	1	Response valid
o_i2c_fill_data	32	Response data
i_i2c_fill_stall	1	When 1, no more responses can be given
UART interface (clk_xbar and clk_ni)		
o_uart_enq	1	TX byte enqueue valid
o_uart_enq_data	8	TX byte to be enqueued
i_uart_tx_words	11	TX FIFO level in bytes
i_uart_tx_full	1	When 1, TX FIFO is full
o_uart_deq	1	RX byte dequeue valid
i_uart_deq_data	8	First RX byte in FIFO
i_uart_rx_words	11	RX FIFO level in bytes
i_uart_rx_empty	1	When 1, RX FIFO is empty
i_uart_byte_rcv	1	Set whenever a new byte is received
MBS UART & Timer interface (clk_cpu)		
o_mbs_uart_irq	8	Interrupt lines to MBS blocks
i_mbs_uart_irq_clear	8	Interrupt clear bits from MBS blocks
o_mbs_drift_fw	1	Global timer drift forward
o_mbs_drift_bw	1	Global timer drift backward
MBS Load status interface (clk_ni)		
i_mbs0_status	4	Core & network load from MBS #0
i_mbs1_status	4	Core & network load from MBS #1
i_mbs2_status	4	Core & network load from MBS #2
i_mbs3_status	4	Core & network load from MBS #3
i_mbs4_status	4	Core & network load from MBS #4
i_mbs5_status	4	Core & network load from MBS #5
i_mbs6_status	4	Core & network load from MBS #6
i_mbs7_status	4	Core & network load from MBS #7
MBS Trace interfaces (clk_ni)		
i_mbs0_trc_valid	1	MBS #0 trace valid
i_mbs0_trc_data	8	MBS #0 trace data
i_mbs1_trc_valid	1	MBS #1 trace valid
i_mbs1_trc_data	8	MBS #1 trace data
i_mbs2_trc_valid	1	MBS #2 trace valid
i_mbs2_trc_data	8	MBS #2 trace data
i_mbs3_trc_valid	1	MBS #3 trace valid
i_mbs3_trc_data	8	MBS #3 trace data
i_mbs4_trc_valid	1	MBS #4 trace valid
i_mbs4_trc_data	8	MBS #4 trace data
i_mbs5_trc_valid	1	MBS #5 trace valid
i_mbs5_trc_data	8	MBS #5 trace data
i_mbs6_trc_valid	1	MBS #6 trace valid

Pin Name	Width	Description
i_mbs6_trc_data	8	MBS #6 trace data
i_mbs7_trc_valid	1	MBS #7 trace valid
i_mbs7_trc_data	8	MBS #7 trace data
Crossbar interface (clk_xbar)		
i_xbar_out_enq	3	Enqueue (per VC bitmap)
i_xbar_out_offset	8	Offset in packet (in 16-b words)
i_xbar_out_eop	1	End of packet, send it to destination
i_xbar_out_data	16	Data to be written to offset
o_xbar_out_full	3	FIFO full (per VC bitmap)
o_xbar_out_packets_vc0	3	Space left in VC #0 FIFO (packets)
o_xbar_out_packets_vc1	3	Space left in VC #1 FIFO (packets)
o_xbar_out_packets_vc2	3	Space left in VC #2 FIFO (packets)
i_xbar_in_deq	3	Dequeue (per VC bitmap)
i_xbar_in_offset	8	Offset in packet (in 16-b words)
i_xbar_in_eop	1	End of packet, proceed to next one
o_xbar_in_data	16	Data read from offset or packet destination peek when idle
o_xbar_in_empty	3	FIFO empty (per VC bitmap)

Table 3.7: Formic BCTL Pin List

3.6.3 Functionality

The Formic board controller reuses an MNI block (section 2.5, page 19) and an XBI block (section 3.1, page 39) to provide network access to board peripherals and registers. Figure 3.9 shows which MNI interfaces are used to connect to which BCTL subblocks. The MNI read, write, trace, operation and CMX interfaces are not used.

In the case of a normal power up sequence or a board hard reset, the board reset manager will assert a boot request (i_ddr_boot_req). A boot manager inside the Formic board controller must then load the contents of 4 BRAMs (8 KB total) to the board DRAM, at address 0, by using the MNI writeback interface to write the ROM contents with a cache-line granularity. When all the acknowledges for these cache lines come back, the boot manager notifies the reset manager that booting is complete (o_ddr_boot_done is asserted). In the case of a soft reset, the reset manager does not ask for the boot procedure to begin.

4 BRAMs are used to implement an 8-KB boot ROM that contains the MicroBlaze program memory. Using appropriate Xilinx utilities[4], a bitstream can be patched to alter the ROM contents.

The TLB registers programming interface is detailed in section 4.3.2 (page 121). Whenever a TLB entry is written, a TLB maintenance command is sent to the TLB block (o_tlb_maint_cmd) marked as a write (o_tlb_maint_wr_en asserted), with the new virtual and physical address translation as well as the valid bit value. A register read issues a read command to the TLB block (o_tlb_maint_wr_en deasserted) which responds with the appropriate physical address and valid bit that are returned to the reader.

To implement the LED blinking and half-tone brightness (section 4.3.1, page 116), pulse-width modulation using the clk_cpu must be provided for each of the twelve LEDs independently. After some experimentation, a 24-bit clk_cpu counter is used as follows to approximate the LED brightness values:

75% bright: cnt[16]
50% bright: cnt[16] & cnt[15] & cnt[14]

[4] The `data2mem` utility can patch a bitstream. See section 4.4 on page 132 for more details.

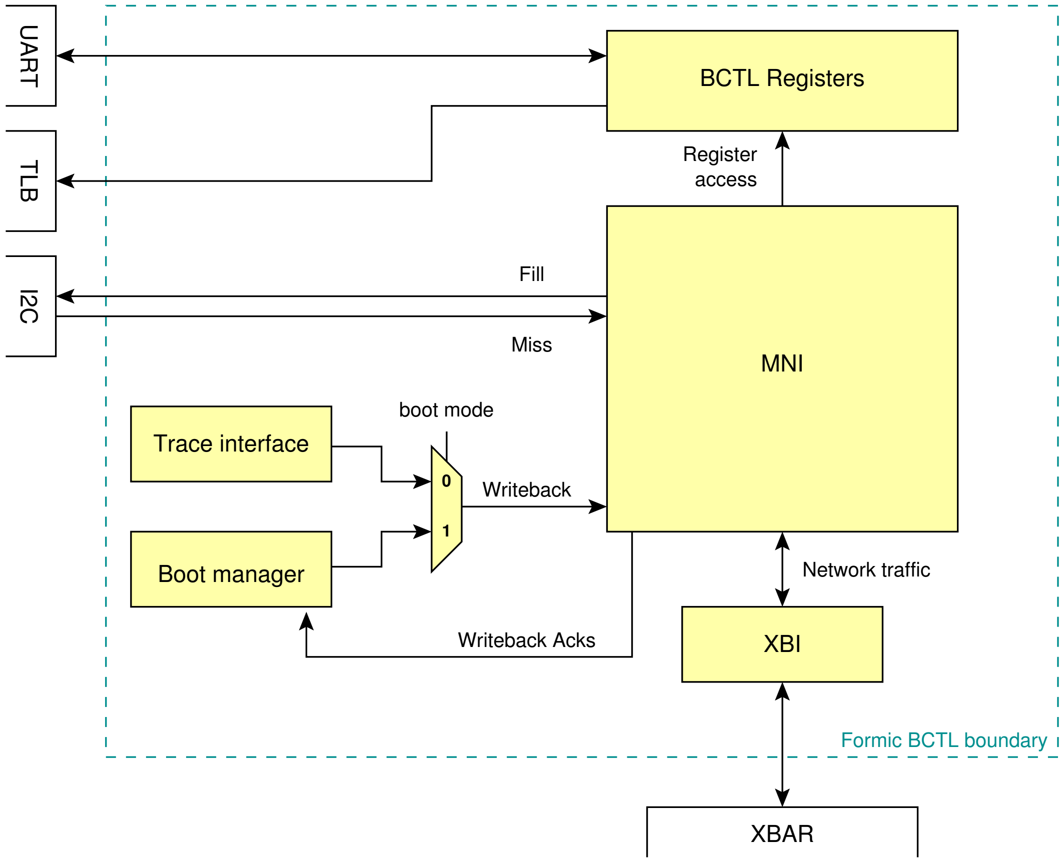


Figure 3.9: Formic BCTL block diagram

Blinking: cnt[23] & cnt[16]

The UART register interface (section 4.3.4, page 125) support a single-byte read/write, or an up to 63 bytes bulk write. This is directly translated to a simple read/write to the UART block, by asserting the o_uart_enq or o_uart_deq for a single or a consecutive number of clock cycles. UART enqueueing/dequeueing happens at 16-bit quantities under clk_ni, as a normal register access. However, to support an 8-bit interface to the UART block, writes are upconverted to a clk_xbar 8-bit bus, whereas reads remain at clk_ni since they refer to a single byte.

An interrupt line to one of the 8 MBS blocks can be programmed to be asserted when the UART receive FIFO becomes non-empty. In this case, BCTL asserts one of the 8 lines (one of the o_mbs_uart_irq bits) and keeps it high until the MBS block asserts the respective i_mbs_uart_irq_clear bit.

BCTL also implements a 32-bit global timer, which counts at the clk_cpu rate. The value of this counter represents the board-level time and should be: *(i)* visible locally (and read-only) inside all 8 MBS blocks, and *(ii)* synchronized among all boards through the network or through the I^{2}C bus. To fulfill the second objective, the BCTL timer register (page 124) is writable from the network and the I^{2}C bus. Upon a write, the difference of the timer and the written value is computed and the timer drifts forward or backward with a value of 1 per clock cycle, until the difference is reconciliated — if the difference is N, then after N clock cycles the timer will have drifted to the correct time. In order not to have eight 32-bit busses to expose the synchronized global timer to each MBS block, we only send two bits, one whenever a forward drift is done (o_mbs_drift_fw) and one whenever a backward drift is done (o_mbs_drift_bw). Each MBS block keeps a local read-only version of the timer, which drifts backward or forward based on these two bits that BCTL generates.

Finaly, a trace interface implements the functionality of the trace registers (section 4.3.5, page 129). When tracing is enabled board-wide, then whenever a trace-enabled MBS block sends a memory-related trace, the data is packed temporarily in BCTL and tagged with an identifier of the MBS that caused this trace. Whenever a full cache line of traces is accumulated, the entry is written in DRAM using the MNI writeback interface.

Small 64x8 FIFOs, one per MBS block, are used to collect the tracing information from the MBS blocks. Each such trace is 8 clock cycles long, transmitted over an 8-bit link at clk_ni speed to minimize wiring cost. The trace consists of two 32-bit words, one for the trace instruction address and one for the trace data address.

A single 512x32 FIFO is used to collect traces from the 8 small FIFOs and submit them, cache line by cache line, to the MNI writeback interface. On the case of too many simultaneous, back-to-back traces from multiple MBS blocks and/or a slow network to the DRAM, it can happen that some of the traces may be lost. A special "drop" event is recorded in this case to the DRAM to reflect this case. Specifically, the final DRAM format for the combined traces is as follows:

Adr	Drop bit, 27 zero bits, 3-bit MBS ID of trace event 0
Adr + 4	Instruction address of trace event 0
Adr + 8	Data address of trace event 0
Adr + 12	Drop bit, 27 zero bits, 3-bit MBS ID of trace event 1
Adr + 16	Instruction address of trace event 1
Adr + 20	Data address of trace event 1
Adr + 24	Drop bit, 27 zero bits, 3-bit MBS ID of trace event 2
...	...

The "Drop bit" is set to 1 if the small 64x8 FIFO received at least one trace that could not fit in it and was forced to drop it.

3.7 I2C block (I2C)

3.7.1 Purpose

The 2C block implements the 2-wire I^2C tristate protocol that enables slave access to the board controller registers. On the inside, it mimics the MNI-L2C Miss/Fill interface, so it can hookup to a standard MNI block which is inside the Board controller.

3.7.2 Pin List

Pin Name	Width	Description
Static configuration		
i_board_id	7	Board ID of the slave interface
MNI Miss/Fill interface (clk_mc)		
clk_mc	1	Cache clock, 40 MHz
rst_mc	1	Reset for clk_mc
o_i2c_miss_valid	1	New register access valid
o_i2c_miss_adr	8	Register address
o_i2c_miss_flags	2	Miss flags (unused)
o_i2c_miss_wen	1	When 1, access is write
o_i2c_miss_ben	4	Byte enables
o_i2c_miss_wdata	32	Write data to register
i_l2c_miss_stall	1	When 0, miss access is complete
i_i2c_fill_valid	1	Reply data for a read access is valid
i_i2c_fill_data	32	Read data from register
o_i2c_fill_stall	1	When 0, fill access is complete
I^2C bus		
i_scl	1	Input clock
io_sda	1	Input/output data (tristated)

Table 3.8: I2C Pin List

3.7.3 Functionality

The I2C block implements a typical 8-bit I^2C slave interface that can read or write a single 32-bit register. Please refer to the I^2C standard for the details of the bus protocol. The 32-bit read/write is supported by four bursts of 8 bits, prepended by the register address.

An external I^2C master (M) should use the following format to write a register to a Formic board I^2C slave (S):

Bit position	Contents	Direction
0	Start condition	$M \rightarrow S$
1-7	Board ID	$M \rightarrow S$
8	0 (indicates write access)	$M \rightarrow S$
9	Ack	$S \rightarrow M$
8-15	Register address	$M \rightarrow S$
16	Ack	$S \rightarrow M$
17-24	Write data [31:24]	$M \rightarrow S$
25	Ack	$S \rightarrow M$
26-33	Write data [23:16]	$M \rightarrow S$
34	Ack	$S \rightarrow M$
35-42	Write data [15:8]	$M \rightarrow S$

Bit position	Contents	Direction
43	Ack	S $\rightarrow$ M
44-51	Write data [7:0]	M $\rightarrow$ S
52	Ack	S $\rightarrow$ M
53	Stop condition	M $\rightarrow$ S

Table 3.9: Register write from external master

To read a register, the format is modified as follows:

Bit position	Contents	Direction
0	Start condition	M $\rightarrow$ S
1-7	Board ID	M $\rightarrow$ S
8	1 (indicates read access)	M $\rightarrow$ S
9	Ack	S $\rightarrow$ M
8-15	Register address	M $\rightarrow$ S
16	Ack	S $\rightarrow$ M
17-24	Read data [31:24]	S $\rightarrow$ M
25	Ack	M $\rightarrow$ S
26-33	Read data [23:16]	S $\rightarrow$ M
34	Ack	M $\rightarrow$ S
35-42	Read data [15:8]	S $\rightarrow$ M
43	Ack	M $\rightarrow$ S
44-51	Read data [7:0]	S $\rightarrow$ M
52	Ack	M $\rightarrow$ S
53	Stop condition	M $\rightarrow$ S

Table 3.10: Register read from external master

The special board ID value of 0x7F is implemented as a broadcast address, so that an external master can write a register in bulk to all boards at once. This is useful to support soft/hard resets simultaneously for all boards.

3.8 UART block (UART)

3.8.1 Purpose

The UART block implements a basic RS-232 serial interface at 38,400 bps (8 data bits, 1 stop bit, no parity, no flow control) to drive the board serial port.

3.8.2 Pin List

Pin Name	Width	Description
clk_cpu	1	CPU clock (10 MHz)
clk_ni	1	Network interface clock (80 MHz)
clk_xbar	1	Crossbar clock (160 MHz)
rst_ni	1	Reset for clk_ni
rst_xbar	1	Reset for clk_xbar
UART Interface (clk_xbar and clk_ni)		
i_uart_enq	1	TX byte enqueue valid
i_uart_enq_data	8	TX byte to be enqueued
o_uart_tx_words	11	TX FIFO level in bytes
o_uart_tx_full	1	When 1, TX FIFO is full
i_uart_deq	1	RX byte dequeue valid
o_uart_deq_data	8	First RX byte in FIFO
o_uart_rx_words	11	RX FIFO level in bytes
o_uart_rx_empty	1	When 1, RX FIFO is empty
o_uart_byte_rcv	1	Set whenever a new byte is received
RS-232 Interface (asynchronous)		
i_RX	1	Receive bit
o_TX	1	Transmit bit

Table 3.11: UART Pin List

3.8.3 Functionality

We mainly use the Xilinx UART Lite block, configured at 38,400 bps, 8 data bits and no parity. This performs all the conversion from a byte-level interface to the asynchronous RS-232 protocol.

Externally to the UART Lite IP, we use two 1024x8 FIFOs, one for the transmit path and one for the receive path. The reason is that UART Lite has two very small buffers inside it, that are not adequate to support bulk writes and/or multiple incoming bytes.

Writes are done with clk_xbar to the transmit FIFO and reads are done with clk_ni from the receive FIFO. The UART Lite and the other ends of the two FIFOs are clocked with clk_cpu.

Chapter 4

Programmer's Model

4.1 Formic Memory Map

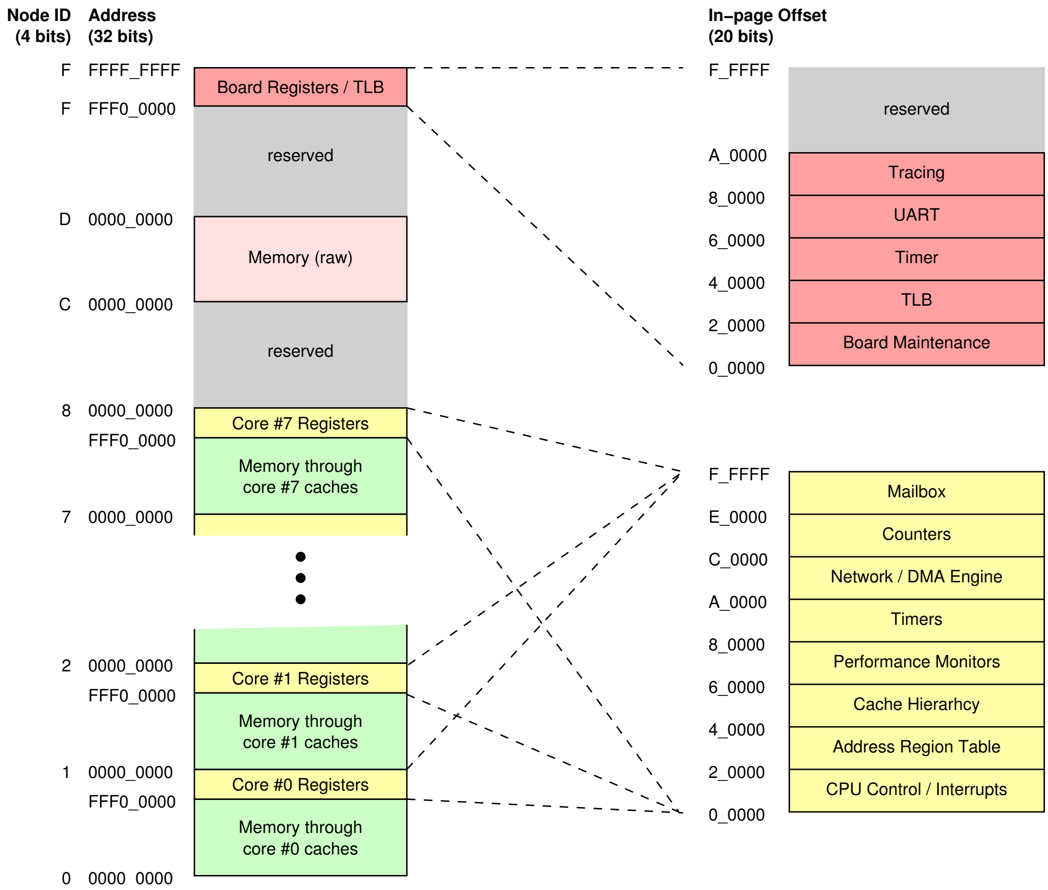


Figure 4.1: Formic Board Memory Map

The Formic board memory map is shown in figure 4.1. Note that the 1 MB window each core has from the address 0xFFF0_0000 – 0xFFFF_FFFF is the default configuration loaded at reset. The window is programmable through the core Address Region Table at region entry #0. The board-wide registers 1 MB window (node ID 0xF, addresses 0xFFF0_0000 – 0xFFFF_FFFF) is fixed and cannot be reprogrammed.

Each CPU core can directly access its own Register window through store and load instructions at the programmed window addresses. In order to access the Board Register window or another core's Register window (e.g. to enqueue something in its Mailbox), it must perform a Network operation by accessing its own Network Interface.

The Board Register window is also (mostly) accessible through the I^2C slave interface, at special 8-bit addresses noted beside each accessible register.

4.2 Formic MBS Registers Description

In the following pages, all Formic per-core registers are explained in detail. These registers are directly accessible through load/store instructions by each core.

A note on Access types for register fields:

RW	Read-write: reads return the last write
RZ	Read-zeroed: read only register, but any write initializes it to 0
R	Read-only: writes are ignored
W	Write-only: reads return unspecified value
(unused)	Writes are ignored, reads return unspecified value

4.2.1 CPU Control & Interrupts Registers

CPU System Call Entry Register

31–9	8	7–1	0
	Y		S

Address offset 0x0_0000

Reset value Unspecified

Access type W

Fields

Y *Yield Processor.* Writing 1 to this bit will cause the CPU to block until an interrupt or exception occurs. Not all interrupts will cause the processor to wake up; they must have been unmasked in the CPU Interrupt register (page 70).

When the processor wakes up from such an interrupt or non-maskable exception, the A bit in the CPU Control register (page 68) will be set to 1; interrupt handlers can use this value to decrement the interrupt return address and cause the program to block again, if needed.

Note that this register can be written in non-privileged mode, even if the whole register address region is privileged.

S *System Call Entry.* Writing 1 to this bit will cause a System Call exception.

Note that this register can be written in non-privileged mode, even if the whole register address region is privileged.

CPU Control Register

31–26	25	24	23–17	16	15–11	10	9	8	7–1	0
	T	M		I		A	H	P		E

Address offset 0x0_0004

Reset value T=0, M=0, I undefined, A=0, H=0, P=1, E=special

Access type All RW, except I (W)

Fields

T *CPU Data Access Tracing Enable.* When set to 1, the Data Access tracing is enabled. For details see page 129.

M *Performance Monitoring Enable.* When set to 1, the performance monitors (section 4.2.4, page 79) are enabled. Note that the programmer must clear the monitors he's interested in before enabling the M bit here.

I *Software Interrupt.* Writing 1 to this bit causes a Software Interrupt.

A *Instruction Aborted.* A is set to 1 automatically by the hardware when an exception or interrupt occurs, while the processor was blocked because of a Yield (page 67) or a blocking Mailbox read (page 111). It means that the current instruction did not complete. Interrupt handlers can use this value to decrement the interrupt return address and cause the program to retry the instruction, if needed, possibly causing a new block. Interrupt handlers are expected to set this bit to 0 before they exit, if they plan to use this value; otherwise, a stale value of 1 might be present on a new interrupt.

H *Handler privileged mode.* H is set to 1 automatically by the hardware when an exception, interrupt or system call occurs. It overrides the P bit value and causes the CPU to run in privileged mode.

The interrupt handler is expected to clear this bit, in order to let the privileged mode return to its programmed P value. This must be close to the last actions the interrupt handler does, because H can be reset to 1 by a new or remaining interrupts. To avoid races, the H clear should happen *after* having cleared the served interrupts in the CPU Interrupt register and after having searched the CPU Status register for any remaining interrupts. For more details, see section 4.5 (page 133).

P *Privileged mode.* When 1, CPU operates in privileged mode and when 0 in user mode. Note that when H is 1, CPU always operates in privileged mode regardless of the P value.

E *Processor Enable.* Enables (E=1) or disables (E=0) this CPU. The reset value of the E bit is 1 for single-core configurations. For multi-core configurations, core ID #0 has a reset value of E=1 and all other cores have a reset value of E=0.

CPU Status Register

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

BID	MI		NMI		CID

Address offset 0x0_0008

Reset value MI=0, NMI=0, CID and BID hardcoded to core/board ID

Access type R

Fields

BID *Board ID.* Board identifier, taken from board dip switches (8 least significant bits).

MI *Maskable Interrupts Status.* Indicates whether one of the maskable interrupts is set. These bits will be set regardless of whether they are masked and did not actually cause an interrupt.

Bit 23: *UART RX Interrupt.*	UART has received new byte(s).
Bit 22: *Timer Interrupt.*	The Private Timer reached 0.
Bit 21: *Counter Interrupt #3.*	A Counter triggered (4^{th} source).
Bit 20: *Counter Interrupt #2.*	A Counter triggered (3^{rd} source).
Bit 19: *Counter Interrupt #1.*	A Counter triggered (2^{nd} source).
Bit 18: *Counter Interrupt #0.*	A Counter triggered (1^{st} source).
Bit 17: *Mailbox Interrupt.*	The mailbox or the mailslot just became non-empty.
Bit 16: *Software Interrupt.*	A software-initiated interrupt occured.

NMI *Non-maskable Interrupts Status.* Indicates whether one of the non-maskable interrupts (actually exceptions or faults) is set.

Bit 12: *NI FIFO Exception.*	The local network operation FIFO was full while the local CPU tried to enqueue a new DMA or Message operation.
Bit 11: *Permission Fault.*	The CPU tried to access a forbidden memory location and ART rejected it.
Bit 10: *ART Miss Fault.*	The CPU tried to access a location which was unmapped in the ART.
Bit 09: *TLB Miss Fault.*	The CPU tried to access a location which did not have a valid TLB entry.
Bit 08: *System Call.*	The System Call Entry register was written.

CID *Core ID.* Unique identifier for each cpu on board, hardcoded.

CPU Interrupt Register

31–24	23–16	15–13	12–8	7–0
	MCLR		NMCLR	MASK

Address offset 0x0_000C

Reset value MCLR/NMCLR undefined, MASK = 8'b11111111

Access type MCLR/NMCLR W, MASK RW

Fields

MCLR *Maskable Interrupts Clear.* Setting one of these bits to 1, clears the respecting interrupt as it appears in the same bit position in the CPU Status register on page 69. Writing 0 to any bit is ignored.

NMCLR *Non-maskable Interrupts Clear.* Setting one of these bits to 1, clears the respecting non-maskable interrupt as it appears in the same bit position in the CPU Status register on page. Writing 0 to any bit is ignored. 69.

MASK *Interrupt Mask.* Setting these bits to 1 will cause the related maskable interrupt not to occur. The CPU may still find out about the cause by reading the CPU Status register on page 69.

The bit position order for the interrupts is the same as in the MCLR field – just decrement the number 16 from it. For example, the Counter Interrupt #2 (bit 20 in MCLR as well as MI in the CPU Status register) is found on bit 4 in the MASK field.

4.2.2 Address Region Table Registers

ART Entry #0 Register (Compulsory I/O region)

31	30	29	28	27	26	25	24	23–12	11–0
0	1	0	0	U	1		1	BOUND=BASE	BASE

Address offset 0x2_0000

Reset value BASE=0xFFF, U=0

Access type RW (BASE, U), R (C, I, R, X, P, V, BOUND)

Fields

This is a special case of an ART Entry register. Since an I/O region must be present at all times (otherwise the CPU could not have access to anything, not even these ART registers), it is always valid and supplies an I/O region. At reset it maps the I/O page to the addresses 0xFFF0_0000 – 0xFFFF_FFFF.

Please see the ART Entry #1 Register on the next page for a field description. Only the BASE and U fields are writable, every other write is ignored.

The fields that have stuck values are: C=0, R=0, X=0, P=1 and BOUND=BASE. These are read-only.

ART Entry #1 Register

31	30	29	28	27	26	25	24	23–12	11–0
C	0	R	X	U	P		V	BOUND	BASE

Address offset 0x2_0004

Reset value BASE=0x000, BOUND=0xFFE, V=1, C=0, R=0, X=1, U=0, P=1

Access type RW, except I (R)

Fields

C *Cacheable bit.* When 1, region is cacheable in L1 and L2 caches.

I *I/O bit.* When 1, region is not mapped to real memory; accesses to it are redirected onto the Register area and the 20 LSBs of any address will be translated to offsets for Register accesses.

Only the ART Entry #0 has I=1. All other entries are forced to have I=0.

R *Read-only bit.* When 1, region is read only. Trying to write to it will generate a Permission fault.

X *Execute bit.* When 0, region is non-executable. Trying to fetch instructions from it will generate a Permission fault.

U *User-level bit.* When 0, region cannot be accessed in user (non-privileged) mode. If the CPU is in user mode (CPU Control P bit = 0) and tries to access such a region, a Permission fault will occur.

P *Privileged bit.* When 0, region cannot be accessed in privileged mode. If the CPU is in privileged mode (CPU Control P bit = 1) and tries to access such a region, a Permission fault will occur.

V *Valid bit.* When 1, this region is valid and its Base and Bound addresses will be used to match addresses from the CPU. Note that enough valid regions should exist to cover all the expected CPU instruction and data accesses, otherwise an ART Miss fault will occur.

BOUND *Bound address.* The 12 MSBs of the Bound address. See below.

BASE *Base address.* The 12 MSBs of the Base address. A CPU generated address hits in an ART region if:

$$\text{BOUND} \leq \text{CPU\_adr}[31:20] \leq \text{BASE}$$

Note that ART entries are prioritized: entry #0 is looked up first, entry #1 second and so on. This allows regions being enclosed into larger regions.

ART Entry #2 Register

31	30	29	28	27	26	25	24	23–12	11–0
C	0	R	X	U	P		V	BOUND	BASE

Address offset 0x2_0008

Reset value All fields to 0

Access type RW, except I (R)

Fields

Please see the ART Entry #1 register for a field description on page 72.

ART Entry #3 Register

31	30	29	28	27	26	25	24	23–12	11–0
C	0	R	X	U	P		V	BOUND	BASE

Address offset 0x2_000C

Reset value All fields to 0

Access type RW, except I (R)

Fields

Please see the ART Entry #1 register for a field description on page 72.

ART Entry #4 Register

31	30	29	28	27	26	25	24	23–12	11–0
C	0	R	X	U	P		V	BOUND	BASE

Address offset 0x2_0010

Reset value All fields to 0

Access type RW, except I (R)

Fields

Please see the ART Entry #1 register for a field description on page 72.

4.2.3 Cache Hierarchy Registers

Cache Control Register

31–27	26–24	23–19	18–16	15–5	4	3–1	0
	CMW		EPC		L		E

Address offset 0x4_0000

Reset value CMW=4, EPC=0, L=0, E=0

Access type RW

Fields

CMW *CPU Minimum Ways.* When the cache is in Epoch mode (L=1), CMW defines the minimum ways that the CPU is guaranteed to have for the current Epoch in the Level 2 cache. CMW=0 means 1 way is guaranteed and CMW=7 means 8 ways are guaranteed.

This guarantee works for cache lines that are conflicting between the CPU and the different-epoch part of MNI (see EPC field below). When conflicts among Epochs happen, they are resolved by the following rules (assume that all Epoch arithmetic is modulo 8):

1. When CPU accesses a cache line (read/write hit or miss which is fetched by MNI), the cache line is marked with Epoch number EPC; EPC is assumed to be the "current" Epoch.
2. When MNI pulls in new cache lines while doing a DMA operation, the cache lines are marked with Epoch number EPC+1; EPC+1 is assumed to be the "next" Epoch.
3. When a L2 miss or a MNI DMA cache line conflicts with other lines in the set which have an Epoch between EPC+2 and EPC+7, the highest one of the latter is replaced (the one closer to EPC+7, which is assumed to be the "oldest" Epoch).
4. When all the lines in the set are either marked EPC or EPC+1 – i.e. only current and next Epoch lines are in this set – then the CPU can allocate up to CMW ways for the current Epoch and will leave the rest for the MNI next Epoch. Respectively, MNI cannot allocate more than 8-CMW ways in this case.

EPC *Current Epoch.* When the cache is in Epoch mode (L=1), EPC specifies the current Epoch number that is used to tag the Level 2 cache lines. System software is expected to increment this by one each time a new task, context or other working set indicator is changed.

CPU requests and updates cache lines that belong to the current Epoch. MNI may fetch cache lines either in the current Epoch – serving misses and evictions – but may also fetch cache lines in the next Epoch – doing Remote or Local DMAs to fetch data from other cores and caches.

L *LRU Mode.* When L=1, the Level 2 cache works in strict LRU mode and the values for CMW and EPC do not matter. When L=0, the Level 2 cache works in Epoch mode and the values of CMW and EPC work as described above.

E *Enable bit.* Enables (E=1) or disables (E=0) the Instruction Level 1, the Data Level 1 and the unified Level 2 cache. When the cache hierarchy is disabled, all CPU instruction and data accesses are handled like a L1/L2 miss and are requested from the DRAM; however, in this case the miss fetches only a single 32-bit word.

Note that at reset the internal structure of all cache tags is unspecified. Cache clear maintenance operations must be performed *before* the caches are enabled, otherwise the results are unpredictable.

Cache Maintenance 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
														F	C								D								I

Address offset 0x4_0004

Reset value undefined

Access type W

Fields

F *Level 2 Flush bit.* When set to 1, a cache flush maintenance operation starts in the unified Level 2 cache. All dirty cache lines will be flushed to DDR memory and they will remain valid unless the Level 2 Clear bit is also set (see below). The CPU instruction will not return until the maintenance is finished, which varies dramatically depending on how many cache lines were dirty and what is the upper bound of a DDR flush operation with traffic in the system. The F bit can be set simultaneously with other bits; see the I bit for details.

C *Level 2 Clear bit.* When set to 1, a cache clear maintenance operation starts in the unified Level 2 cache. All valid cache lines will be invalidated. Dirty lines flushing depends on the Level 2 Flush bit, specified above. The CPU instruction will not return until the maintenance is finished, which for the 256-KB 8-way unified Level 2 cache will happen in the order of 128 processor clock cycles if the F bit is not also set; otherwise the latency can be much higher. The C bit can be set simultaneously with other bits; see the I bit for details.

D *Data Level 1 Clear bit.* When set to 1, a cache clear maintenance operation starts in the Data Level 1 cache. The CPU instruction will not return until the maintenance is finished, which for the 8-KB 2-way Data Level 1 cache will happen in the order of 32 processor clock cycles. The D bit can be set simultaneously with other bits; see the I bit for details.

I *Instruction Level 1 Clear bit.* When set to 1, a cache clear maintenance operation starts in the Instruction Level 1 cache. The CPU instruction will not return until the maintenance is finished, which for the 4-KB 2-way Instruction Level 1 cache will happen in the order of 16 processor clock cycles.

Note that the I, D, C and F bits be set simultaneously; the CPU instruction will return when all maintenance operations are finished, i.e. when the longest of all requested operations has been completed.

4.2.4 Performance Monitoring Registers

Instruction Level 1 Cache Monitor

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
IL1_MISSES																IL1_HITS															

Address offset 0x6_0000

Reset value Undefined

Access type RZ

Fields

IL1_MISSES *Instruction Level 1 Cache Misses.* Counts the number of misses in the Instruction Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

IL1_HITS *Instruction Level 1 Cache Hits.* Counts the number of hits in the Instruction Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Data Level 1 Cache Monitor

31 … 16	15 … 0
DL1_MISSES	DL1_HITS

Address offset 0x6_0004

Reset value Undefined

Access type RZ

Fields

DL1_MISSES *Data Level 1 Cache Misses.* Counts the number of misses in the Data Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

DL1_HITS *Data Level 1 Cache Hits.* Counts the number of hits in the Data Level 1 cache. Note that write hits still cause a Level 2 access, because the Data Level 1 cache is write-through.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Level 2 Instruction Monitor

31 … 16	15 … 0
IL2_MISSES	IL2_HITS

Address offset 0x6_0008

Reset value Undefined

Access type RZ

Fields

IL2_MISSES *Level 2 Instruction Misses.* Counts the number of misses in the unified Level 2 cache that are caused by the Instruction Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

IL2_HITS *Level 2 Instruction Hits.* Counts the number of hits in the unified Level 2 cache that are served to the Instruction Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Level 2 Data Monitor

31–16	15–0
DL2_MISSES	DL2_HITS

Address offset 0x6_000C

Reset value Undefined

Access type RZ

Fields

DL2_MISSES *Level 2 Data Misses.* Counts the number of misses in the unified Level 2 cache that are caused by the Data Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

DL2_HITS *Level 2 Data Hits.* Counts the number of hits in the unified Level 2 cache that are served to the Data Level 1 cache. Note that this number also includes all the traffic caused by the write-through nature of the Data Level 1 cache.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Registers Access Monitor

31–16	15–0
NET_REGS	CPU_REGS

Address offset 0x6_0010

Reset value Undefined

Access type RZ

Fields

NET_REGS *Network Register Accesses.* Counts the number of accesses made to any Core register of this MBS block, which originated from the Network.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

CPU_REGS *CPU Register Accesses.* Counts the number of accesses made to any Core register of this MBS block, which originated from the local CPU.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Operations Monitor

31–16	15–0
REMOTE_OPS	LOCAL_OPS

Address offset 0x6_0014

Reset value Undefined

Access type RZ

Fields

REMOTE_OPS *Remote Network Operations.* Counts the number of DMAs that were received from the network side and have been (including these that will be) serviced by the local DMA engine.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

LOCAL_OPS *Local Network Operations.* Counts the total number of Messages and DMAs that were requested by the local CPU. These include remote DMA read operations, which are simply forwarded to a remote DMA engine to be serviced.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Level 2 Reads Monitor

31–16	15–0
READ_MISSES	READ_HITS

Address offset 0x6_0018

Reset value Undefined

Access type RZ

Fields

READ_MISSES *Network Level 2 Read Misses.* Counts the number of cache lines that the DMA engine requested from the Level 2 cache, but the cache did not contain them. The DMA engine asked these cache lines to be forwarded from the local DRAM to their final destination.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

READ_HITS *Network Level 2 Read Hits.* Counts the number of cache lines that the DMA engine requested from the Level 2 cache and the cache contained them. The DMA engine was able to send these cache lines directly to their final destination.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Level 2 Writes Monitor

31–16	15–0
WRITE_FAILURES	WRITE_SUCCESSES

Address offset 0x6_001C

Reset value Undefined

Access type RZ

Fields

WRITE_FAILURES *Network Level 2 Write Failures.* Counts the number of cache lines that came from the network and tried to be written in the Level 2 cache, but the cache did not have sufficient space for them. These cache lines might or might not have been written through to the local DRAM, depending on the C bit setting of the MNI Opcode register (section 4.2.6, page 94) that ordered the corresponding DMAs.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

WRITE_SUCCESSES *Network Level 2 Write Successes.* Counts the number of cache lines that came from the network and were successfully written in the Level 2 cache. These cache lines might have also been written through to the local DRAM, depending on the W bit setting of the MNI Opcode register (section 4.2.6, page 94) that ordered the corresponding DMAs.

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Packets VC #0 Monitor

31 … 16	15 … 0
VC0_OUT	VC0_IN

Address offset 0x6_0020

Reset value Undefined

Access type RZ

Fields

VC0_OUT *Network VC #0 Output Packets.* Counts the total number of packets that were sent to the network Virtual Circuit #0 (Acknowledge VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

VC0_IN *Network VC #0 Input Packets.* Counts the total number of packets that were received from the network Virtual Circuit #0 (Acknowledge VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Packets VC #1 Monitor

31–16	15–0
VC1_OUT	VC1_IN

Address offset 0x6_0024

Reset value Undefined

Access type RZ

Fields

VC1_OUT *Network VC #0 Output Packets.* Counts the total number of packets that were sent to the network Virtual Circuit #1 (Data VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

VC1_IN *Network VC #1 Input Packets.* Counts the total number of packets that were received from the network Virtual Circuit #1 (Data VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

Network Packets VC #2 Monitor

31–16	15–0
VC2_OUT	VC2_IN

Address offset 0x6_0028

Reset value Undefined

Access type RZ

Fields

VC2_OUT *Network VC #2 Output Packets.* Counts the total number of packets that were sent to the network Virtual Circuit #2 (Request VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

VC2_IN *Network VC #2 Input Packets.* Counts the total number of packets that were received from the network Virtual Circuit #2 (Request VC).

When the counter reaches 0xFFFF, it stops counting and stays at this value until it is zeroed by a write.

CPU Energy Monitor

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
CPU_ENERGY																															

Address offset 0x6_002C

Reset value Undefined

Access type RZ

Fields

CPU_ENERGY *CPU Energy Estimation.* Estimates the energy consumption of the CPU, i.e. how many "energy units" have been consumed from the last time the counter was zeroed. The energy units are based on the completed CPU instructions, according to the following model:

Loads/Stores:	Add 1 to the counter.
Arithmetic/Bitwise/Branches:	Add 2 to the counter.
Integer Mul/Div, all Floating point:	Add 4 to the counter.
Misses/Blocked clock cycles:	Do not add to the counter.

When the counter reaches 0xFFFF_FFFF, it wraps around.

CPU Power Monitor

31–16	15–0
	CPU_POWER

Address offset 0x6_0030

Reset value Undefined

Access type RZ

Fields

CPU_POWER *CPU Power Estimation.* Estimates the transient power consumption of the CPU, based on the same model as the CPU Energy Monitor (see previous page). Every 256 CPU clock cycles, the monitor is automatically updated with the sum of the energy units accumulated over this period.

4.2.5 Timers

Global Timer

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
GLOBAL_TMR																															

Address offset 0x8_0000

Reset value N/A

Access type R

Fields

GLOBAL_TMR *Global Timer.* This is a 32-bit, read-only, incrementing counter, which is synchronized to the board-level global timer (page 124). It counts in CPU clock ticks and wraps around when it reaches 0xFFFF_FFFF.

Note that due to the global timer semantics, if timers among multiple boards are synchronized, the timer value may briefly stall for some clock cycles or count at twice the pace. For details, please see page 124.

Private Timer

31	30	29–0
E	I	PRIVATE_TMR

Address offset 0x8_0004

Reset value E=0, I=0, PRIVATE_TMR undefined

Access type RW

Fields

E *Timer Enable.* When 1, enables the timer to count down from the given value.

I *Interrupt Enable.* When 1, enables the assertion of the Timer interrupt when the counter reaches the value 0.

PRIVATE_TMR *Private Timer value.* When written, selects the initial value to start counting. The counter will count down from this value when it is enabled, in CPU clock ticks. When the value reaches 0 it stays there until it is rewritten, possibly triggering a Timer interrupt.

4.2.6 Network & DMA Engine Registers

MNI Opcode Register

31–28	27–20	19–16	15–6	5	4	3	2	1–0
	DST_BRD	DST_ND		I	W	C	A	OP

Address offset 0xA_0000

Reset value Undefined

Access type RW, except OP (W)

Fields

DST_BRD *DMA or Message Destination Board ID.* For Message and DMA operations, contains the Destination Board ID.

DST_ND *DMA or Message Destination Node ID.* For Message and DMA operations, contains the Destination Node ID.

I *Ignore dirty bit on Source.* For DMA operations originating from the local Board and Node, if a cache line is found and read, it informs the source cache to ignore its dirty bit status. The cache line will not be written back to memory, even if it was dirty. However, the cache line remains in the source cache – if the processor writes it again, it will become dirty again.

W *Write-through on Destination.* For all operations that are destined to cacheable nodes, all related cache lines are forced to be also written to main memory even when they can fit into the caches (and would not be normally be written to main memory). The cache lines Dirty bit is set to 0.

C *Force Clean on Destination.* For all operations that are destined to cacheable nodes, all related cache lines that manage to fit into caches are forced to be considered clean (their dirty bit is forced to 0, instead of 1) although they are not written to main memory. Also, if the cache lines do not manage to fit into the caches, they will *not* be written through to main memory.

A *Use Acknowledgments.* For all operations, use the Acknowledgment registers to specify an acknowledgement packet.

OP *Network Operation Opcode.* When this field is written, a new network operation is triggered. Specifically, the following opcodes are supported:

Opcode	**Operation**
'b00	Message, 1 word (MSG0)
'b01	Message, 2 words (MSG0 and MSG1)
'b10	DMA
'b11	*reserved*

A "Message" operation sends one or two 32-bit words, written by the software on MSG0 (first word) and MSG1 (second, if applicable) registers to the destination defined by the tuple DST_BRD, DST_ND and DST_ADR registers. If an acknowledge-ensuing opcode was used, an acknowledgement packet will be sent after the message arrives to the address specified by the ACK_BRD, ACK_ND and ACK_ADR tuple. The acknowledgment will carry a payload consisting of the word "4" (when 1 word was trasnfered) or "8" (when 2 words were transfered).

A "DMA" operation transfers SIZE bytes from the location defined by the SRC_BRD, SRC_ND and SRC_ADR tuple to the location defined by DST_BRD, DST_ND and DST_ADR tuple. The DMA will be split to cache-line sized packets (64 bytes). If an acknowledge-ensuing opcode was used, acknowledgement packets will be sent after each such packet arrives to its destination. The acknowledgment address is specified by the ACK_BRD, ACK_ND and ACK_ADR tuple. The payload of each packet will contain the number of bytes that were transferred. The following three main courses of action are taken, depending on the SRC_BRD and SRC_ND combination:

Source Board/Node	**Action**
Board=local, Node=local	Split into cache-line packets; ask local cache for each one. If cache hits, send the cache line to destination, otherwise send a read packet to local DRAM which will answer directly to destination.
Board=any, Node=0xC	Split into cache-line segments; send a read packet to local or remote DRAM for each segment. DRAM will answer directly to destination.
Other combination	Send a single DMA descriptor packet to the source; it will enqueue it to its Network Operation FIFO and serve it from there. The source FIFO must have adequate space; if not, a Nack will be received.

All addresses and sizes must be 4-bytes aligned when they refer to other registers in the system, a Mailbox, or raw (node ID 'hC) DDR memory locations. This is already enforced by the registers, which hardwire the last two bits of all addresses (as well as the DMA size) to 0. If the addresses refer to cacheable memory locations, they must additionally be cache-line aligned, i.e. aligned on 64-byte (last 6 bits 0) boundaries. An additional constraint is that read operations from registers must be exactly 4 bytes in size – no read bursts are supported. Write bursts are supported normally. All these constraints are shown in the following table:

Source	Destination	Alignment	Size
Cache	(any)	64 B	Multiple of 64 B
(any)	Cache	64 B	Multiple of 64 B
DDR Memory	(any)	4 B	Multiple of 4 B
(any)	DDR Memory	4 B	Multiple of 4 B
Registers	(any)	4 B	Exactly 4 B
(any)	Registers	4 B	Multiple of 4 B
(any)	Mailbox	4 B	Multiple of 4 B

Note that a space of at least 1 must exist in the local network operation FIFO (page 104), otherwise a Network FIFO Full Exception will occur.

Warning: The DMA operations on cacheable locations *cannot* be subject to be written or read by the CPU during the DMA. If the CPU interferes with a memory location that is bound to be accessed by a DMA operation, the behavior is *unspecified*. Also note that for any DMA to work as expected on cacheable locations, the appropriate memory regions must have been marked as Cacheable in the related ARTs. Also, the involved caches should be Enabled.

MNI DMA Size Register

31 … 21	20 … 2	1	0
	SIZE	0	0

Address offset 0xA_0004

Reset value Undefined

Access type RW (bits 1 and 0 are stuck at 0)

Fields

SIZE *DMA Size.* For DMA operations, contains the transfer size in bytes. All DMAs are at least word-aligned (4 bytes boundary) – see the Opcode register on page 94 for more details. For this reason, SIZE contains bits 20 . . . 2 of the intended transfer size in bytes; bits 1 and 0 are always stuck at 0.

The maximum transfer size is 1 MB. Greater values are truncated by the hardware to an unspecified length.

MNI Destination Address Register

31 ... 2	1	0
DST_ADR	0	0

Address offset 0xA_0008

Reset value Undefined

Access type RW (bits 1 and 0 are stuck at 0)

Fields

DST_ADR *DMA or Message Destination Address.* For Message and DMA operations, contains the initial Destination Address.

All message and DMA addresses are at least word-aligned (4 bytes boundary) – see the Opcode register on page 94 for more details. For this reason, DST_ADR contains bits 31 . . . 2 of the byte destination address; bits 1 and 0 are always stuck at 0.

MNI Board/Node Register

31–28	27–20	19–16	15–12	11–4	3–0
	SRC_BRD	SRC_ND		ACK_BRD	ACK_ND

Address offset 0xA_000C

Reset value Undefined

Access type RW

Fields

SRC_BRD *DMA Source Board ID.* For DMA operations, contains the Source Board ID.

SRC_ND *DMA Source Node ID.* For DMA operations, contains the Source Node ID.

ACK_BRD *DMA or Message Acknowledgement Board ID.* For DMA or Message operations, contains the Acknowledgement Board ID.

ACK_ND *DMA or Message Acknowledgement Node ID.* For DMA or Message operations, contains the Acknowledgement Node ID.

MNI Acknowledgement Address Register

31 … 2	1	0
ACK_ADR	0	0

Address offset 0xA_0010

Reset value Undefined

Access type RW (bits 1 and 0 are stuck at 0)

Fields

ACK_ADR *DMA or Message Acknowledgement Address.* For Message and DMA operations, contains the Acknowledgement Address. This may or may not be used, depending on the network operation opcode.

All message and DMA addresses are at least word-aligned (4 bytes boundary) – see the Opcode register on page 94 for more details. For this reason, ACK_ADR contains bits 31 . . . 2 of the byte acknowledgment address; bits 1 and 0 are always stuck at 0.

MNI Source Address Register

<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="30">SRC_ADR</td><td>0</td><td>0</td></tr>
</table>

Address offset 0xA_0014

Reset value Undefined

Access type RW (bits 1 and 0 are stuck at 0)

Fields

SRC_ADR *DMA Source Address.* For DMA operations, contains the initial Source Address.

All message and DMA addresses are at least word-aligned (4 bytes boundary) – see the Opcode register on page 94 for more details. For this reason, SRC_ADR contains bits 31 . . . 2 of the byte source address; bits 1 and 0 are always stuck at 0.

MNI Message Word 0 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
MSG0																															

Address offset 0xA_0018

Reset value Undefined

Access type RW

Fields

MSG0 *Message Word 0.* For Message operations, contains the first word to be sent.

MNI Message Word 1 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
MSG1

Address offset 0xA_001C

Reset value Undefined

Access type RW

Fields

MSG1 *Message Word 1.* For two-word Message operations, contains the second word to be sent.

MNI Status Register

31–22	21–16	15–6	5–0
	NET_OPS		CPU_OPS

Address offset 0xA_0020

Reset value NET_OPS=32, CPU_OPS=32

Access type R

Fields

NET_OPS *Available Network Operations.* Contains the number of Network Operations that can be remotely programmed without overflowing the MNI remote operations buffer. Such operations are the result of a remote DMA engine programmed with a Source Board/Node equaling the local node; the whole DMA is sent to the local node and results in this buffer.

It is the software's responsibility not to overflow the remote operations buffer. In such a case, the incoming packet will be dropped. If an acknowledgment address was used, a Nack will be sent to this address.

CPU_OPS *Available Local Operations.* Contains the number of DMA Operations that can be programmed by the local CPU without overflowing the MNI local operations buffer. Any message or DMA results in an enqueue in the local operations buffer.

It is the software's responsibility not to overflow the local operations buffer, otherwise a Network FIFO Full Exception will be triggered and the new operation will be disregarded.

4.2.7 Counter Registers

Counter Interrupt Register

31–30	29–24	23–22	21–16	15–14	13–8	7–6	5–0
	INT3		INT2		INT1		INT0

Address offset 0xC_0000

Reset value All fields to 0

Access type RW

Fields

INT3 *Counter Interrupt #3.* Selects the Counter for Counter Interrupt #3. Value N implies that Counter N will cause the interrupt #3 to be asserted whenever Counter N triggers, irrespective of any Network notifications that it may also trigger.

Note that always one of the 64 Counters will be associated with this interrupt. If this is not desired, software can mask interrupt #3 by setting the appropriate bit to 0 in the MASK field of the CPU Interrupt Register on page 70.

INT2 *Counter Interrupt #2.* Selects the counter for Counter Interrupt #2, as INT3 above.

INT1 *Counter Interrupt #1.* Selects the counter for Counter Interrupt #1, as INT3 above.

INT0 *Counter Interrupt #0.* Selects the counter for Counter Interrupt #0, as INT3 above.

Counter #0 Value Register

<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">VAL</td></tr>
</table>

Address offset 0xC_1000

Reset value Undefined

Access type RW

Fields

VAL *Counter Value.* Writing this register from the CPU sets the Counter to the given value. Reading it from the CPU simply gives back the current Counter value and does not interfere with it.

When the counter is written to from the Network side (e.g. when used as an Acknowledgement address), the existing data value will be incremented by the amount specified in the network packet. If the increment causes the counter to reach zero, the Counter triggers. When this happens, up to two notification packets will be sent to the addresses specified in the following registers. The notification packet(s) contain the value "1" in this case. An interrupt may also be triggered, depending on the setting of the Counter Interrupt Register.

When the counter receives a Negative Acknowledgment (Nack) packet from the network. This may happen, for instance, when a DMA is started from a remote node, but its remote network operations FIFO is already full. In this case, the counter also triggers (as above), sending up to two notifications and/or causing an interrupt. The notification packet(s) contain the value "0" in this case. When using interrupts, software can simply read back the actual counter value to find out if the counter trigger event was caused by an actual completion (the value will be zero) or a Nack (the value will not be zero).

The normal usage scenario for DMA completion is to initialize the counter from the CPU at a negative value, equal to the amount of data bytes expected for the DMA size. Notifications can be programmed to go to other counters (local or remote), which can gather notifications from multiple DMAs and trigger when all of them are done. Note that in the case of Negative Acknowledgements, this counter "chaining" still works, because the top-level counter will also trigger when it receives a negative notification packet of value 0.

Counter #0 Notification Address #0 Register

31 ... 0
NTF_ADR0

Address offset 0xC_1004

Reset value Undefined

Access type RW

Fields

NTF_ADR0 *Notification Address #0.* Sets the first Notification address for Counter #0.

Counter #0 Notification Address #1 Register

31 … 0
NTF_ADR1

Address offset 0xC_1008

Reset value Undefined

Access type RW

Fields

NTF_ADR1 *Notification Address #1.* Sets the second Notification address for Counter #0.

Counter #0 Notification Board/Node 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
	V1	NTF_BRD1	NTF_ND1		V0	NTF_BRD0	NTF_ND0

Address offset 0xC_100C

Reset value Undefined

Access type RW

Fields

V1 *Notification Valid #1.* When 1, the second Notification Address/Board is considered valid and will be used when the counter triggers (its value reaches zero or a Nack is received).

NTF_BRD1 *Notification Board ID #1.* Sets the second Notification Board ID for Counter #0.

NTF_ND1 *Notification Node ID #1.* Sets the second Notification Node ID for Counter #0.

V0 *Notification Valid #0.* When 1, the first Notification Address/Board is considered valid and will be used when the counter triggers (its value reaches zero or a Nack is received).

NTF_BRD0 *Notification Board ID #0.* Sets the first Notification Board ID for Counter #0.

NTF_ND0 *Notification Node ID #0.* Sets the first Notification Node ID for Counter #0.

Counters #1 . . . #127 Registers

Four registers per each of the 128 Counters exist, similar to the four presented for Counter #0. Specifically, all Counter registers can be found in the following addresses:

Address	**Register**
0xC_0000	Counter Interrupt Register
0xC_1000	Counter #0 Value Register
0xC_1004	Counter #0 Notification Address #0 Register
0xC_1008	Counter #0 Notification Address #1 Register
0xC_100C	Counter #0 Notification Board/Node Register
0xC_1010	Counter #1 Value Register
0xC_1014	Counter #1 Notification Address #0 Register
0xC_1018	Counter #1 Notification Address #1 Register
0xC_101C	Counter #1 Notification Board/Node Register
0xC_1020	Counter #2 Value Register
0xC_1024	Counter #2 Notification Address #0 Register
0xC_1028	Counter #2 Notification Address #1 Register
0xC_102C	Counter #2 Notification Board/Node Register
...	...
0xC_17F0	Counter #127 Value Register
0xC_17F4	Counter #127 Notification Address #0 Register
0xC_17F8	Counter #127 Notification Address #1 Register
0xC_17FC	Counter #127 Notification Board/Node Register

4.2.8 Mailbox Registers

Mailbox Access 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
ACCESS																															

Address offset 0xE_0000

Reset value N/A

Access type Special – see below

Fields

ACCESS *Mailbox Access Register.* This is the data access interface to the Mailbox.

Writing to this register is supported only for the Network interface and results in enqueueing to the Mailbox. Enqueues are done through this single register. Multiple enqueues can be done by addressing a multi-word Message or DMA to this register; this is handled correctly and does not result in overwriting other registers (such as the Mailbox Status Register, which is next in addressing). However, enqueues from the Network should not overflow the Mailbox: this will cause data to be lost, either silently (if the network packet does not have an acknowledgment address) or by triggering a Nack (value 0 is sent to the acknowledgement address).

Reading from this register is supported only for the CPU and results in dequeueing from the Mailbox. If not enough data are present in the Mailbox to cover a CPU read, the CPU is blocked until data arrives from the Network.

Mailbox Status Register

31–29	28–18	17	16	15–13	12–2	1	0
	WR_MARK	0	0		RD_MARK	0	0

Address offset 0xE_0004

Reset value WR_MARK=4096, RD_MARK=0

Access type R

Fields

WR_MARK *Write-side Watermark.* Number of bytes available in the Mailbox for writing. Note that bits 17 and 16 are stuck at 0, because only 4-byte aligned transfers are allowed in the Network. This field should be used to avoid overflowing the Mailboxes.

RD_MARK *Read-side Watermark.* Number of bytes available in the Mailbox for reading. Note that bits 1 and 0 are stuck at 0, because only 4-byte aligned transfers are allowed in the Network. This field may be used if blocking behavior is not required.

Mailslot Access Register

<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">ACCESS</td></tr>
</table>

Address offset 0xE_0008

Reset value N/A

Access type Special – see below

Fields

ACCESS *Mailslot Access Register.* This is the data access interface to the Mailslot.

This register behaves exactly like the main Mailbox access register (page 111), but refers to a separate mailbox that can hold only a single 32b word.

Mailslot Status Register

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3	2	1	0
	F	0	0

Address offset 0xE_000C

Reset value 0

Access type R

Fields

F *Full*. Number of bytes available in the Mailslot for reading. Note that bits 1 and 0 are stuck at 0, because only 4-byte aligned transfers are allowed in the Network. This field may be used if blocking behavior is not required.

4.3 Formic Board Registers Description

In the following pages, all Formic board-wide registers are explained in detail. These registers cannot be directly accessed through load/store instructions; instead, each core needs to do a network operation (message or DMA) to destination node 0xF.

Moreover, most of these registers can also be accessed through the I^2C slave interface at the special 8-bit addresses noted at each such accessible register.

A note on Access types for register fields:

RW	Read-write: reads return the last write
RZ	Read-zeroed: read only register, but any write initializes it to 0
R	Read-only: writes are ignored
W	Write-only: reads return unspecified value
(unused)	Writes are ignored, reads return unspecified value

4.3.1 Board Maintenance Registers

Board 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
L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	L0		LM		S	H

Address offset 0x0_0000 [I^{2}C address: 0x00]

Reset value L11-L8=0; L7-L0=2; LM=2; S & H bits undefined

Access type RW

Fields

L11 – L0 *LED Control.* Writing these fields controls the twelve Formic board LEDs. The supported values are 2 bits for each LED, with the following meaning:

- **2'b00:** Switched off
- **2'b01:** Switched on, 50% bright
- **2'b10:** Switched on, 75% bright
- **2'b11:** Blinking

LM *Led Mode.* Controls LED behavior as follows:

- **2'b00:** All LEDs fully controlled by L11 - L0 bits.
- **2'b01:** *Reserved*
- **2'b10:** LEDs 11 - 8 fully controlled by L11 - L8 bits. LEDs 7 - 0 show GTP LINK_UP bits (see page 118). The LINK_UP values are ANDed with the L7 - L0 bits to produce the final LED behavior.
- **2'b11:** LEDs 11 - 8 fully controlled by L11 - L8 bits. LEDs 7 - 0 show GTP LINK_ERROR bits (see page 118). The LINK_ERROR values are ANDed with the L7 - L0 bits to produce the final LED behavior.

S *Soft Reset.* Writing 1 to this bit will cause the Formic board controller to perform a soft reset. This resets all logic blocks (CPUs, caches, network interfaces, counters, per-core and board registers) but does not redo DDR2 DRAM calibration nor GTP links initialization. Also, the boot ROMs are not written to the DRAM.

H *Hard Reset.* Writing 1 to this bit will cause the Formic board controller to perform a full hardware reset. This completely restarts everything on the FPGA of the hardware board, including re-initializing all clock frequencies, re-calibrating the DDR2 DRAM and re-writing the boot ROMs contents on the DRAM.

Board Status Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																				DIP				BID							

Address offset 0x0_0004 [I^{2}C address: 0x01]

Reset value Depends

Access type R

Fields

DIP *DIP Switches.* The value of the 4 leftmost DIP switches of the Formic board.

BID *Board ID.* The Board ID: set to the value of the 8 rightmost DIP switches of the Formic board.

Board Link Status Register

31–24	23–16	15–8	7–0
CRC_ERROR	CREDIT_ERROR	LINK_ERROR	LINK_UP

Address offset 0x0_0008 [I^2C address: 0x02]

Reset value All fields to 0

Access type All fields RZ except LINK_UP (R)

Fields

CRC_ERROR *CRC Error bitmap.* For each link (SATA7 being the leftmost), the bitmap contains a 1 if one or more CRC Errors were detected upon packet reception. When the register is written, all bits return to 0 until further error(s) are found.

CREDIT_ERROR *Credit Error bitmap.* For each link (SATA7 being the leftmost), the bitmap contains a 1 if one or more packets were dropped due to the receiving FIFO being full. This indicates a problem with the credits transmission and reception. When the register is written, all bits return to 0 until further error(s) are found.

LINK_ERROR *Link Error bitmap.* For each link (SATA7 being the leftmost), the bitmap contains a 1 if one or more times in the past the low-level, physical GTP interface indicated an error. When the register is written, all bits return to 0 until further error(s) are found.

LINK_UP *Link Up bitmap.* For each link (SATA7 being the leftmost), the bitmap contains a 1 if the link is currently up and running.

Board Cores Status Register

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
C7	C6	C5	C4	C3	C2	C1	C0

Address offset 0x0_000C [I^{2}C address: 0x03]

Reset value Undefined

Access type R

Fields

C7 *Core 7 Status.* For MBS Core #7, the two leftmost bits contain an estimation of the CPU load (00: idle, 11: fully loaded) and the two rightmost bits an estimation of its network interface load (00: idle, 11: fully loaded).

C6 *Core 6 Status.* Same as C7, but for MBS Core #6.

... ...

C0 *Core 0 Status.* Same as C0, but for MBS Core #0.

Board DRAM Status Register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
																												ERROR			

Address offset 0x0_0010 [I²C address: 0x04]

Reset value 0

Access type RZ

Fields

ERROR *DDR2 DRAM Controller Error bitmap.* For each DDR2 DRAM Controller port (#3 being the leftmost), the bitmap contains a 1 if one or more times in the past the DRAM Controller indicated an error. When the register is written, all bits return to 0 until further error(s) are found.

4.3.2 TLB Registers

TLB 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
	E

Address offset 0x2_0000 [I^{2}C address: 0x20]

Reset value E=0

Access type RW

Fields

E *Enable TLB.* When the TLB is enabled (E=1), every access to the DDR DRAM memory on board will pass through the translation table. Be sure to install all needed page mappings first before enabling the TLB, otherwise the behavior is undefined.

Physical addresses are 27 bits (128 MB DRAM size) and virtual addresses are 32 bits. Translation happens on a 1-MB page size and the TLB fits all possible 4096 entries; TLB misses do not occur. The translation is based on the following rule:

$$PA[26:0] = \{TLB(VA[31:20]), VA[19:0]\}$$

When the TLB is disabled (E=0), the TLB entries are not used. Instead, the lowest bits of the virtual page number are used directly as follows:

$$PA[26:0] = VA[26:0]$$

TLB Entry #0 Register

31–9	8	7	6–0
	V		PA

Address offset 0x2_1000 [Inaccessible through I^2C]

Reset value Undefined

Access type RW

Fields

V *Entry Valid.* Writing to this register installs the TLB mapping from Virtual page number 0 to Physical page PA. When V=1, the entry is marked "valid", which means that accesses to this 1-MB page will happen normally. When V=0, the entry is marked "invalid", which means that accesses to this page will fail and a TLB Miss Fault exception (page 69) will be raised.

PA *Physical Address.* Indicates the physical address of the Virtual page 0 that is being installed because of writing to this register.

One Entry register exists for each of the 4,096 TLB entries – see next page for a summary of address offsets.

TLB registers summary

One TLB Entry register for each one of the 4,096 possible TLB entries (12 highest bits of the virtual address – 1-MB page size) exists. The table below summarizes the addresses that must be used to access all the TLB registers.

Address	**Register**
0x2_0000	TLB Control Register
0x2_1000	TLB Entry #0 Register (Virtual addresses 0x0000_0000 - 0x000F_FFFF)
0x2_1004	TLB Entry #1 Register (Virtual addresses 0x0010_0000 - 0x001F_FFFF)
0x2_1008	TLB Entry #2 Register (Virtual addresses 0x0020_0000 - 0x002F_FFFF)
...	...
0x2_4FFC	TLB Entry #4095 Register (Virtual addresses 0xFFF0_0000 - 0xFFFF_FFFF)

4.3.3 Timer Registers

Board Global Timer

<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">GLOBAL_TMR</td></tr>
</table>

Address offset 0x4_0000 [I^{2}C address: 0x40]

Reset value 0

Access type RW, but writes are special (see below)

Fields

GLOBAL_TMR *Global Timer.* An incrementing counter counting in CPU clock ticks. Wraps around when it reaches 0xFFFF_FFFF.

The counter starts counting at 0 when the board is reset. When written, the counter internally computes the difference between the old and the new value and begins to drift until it compensates for this difference. Forward drifting (when new value is greater than the old value) is handled by adding 2 to the current value per CPU clock tick instead of the normal 1. Backward drifting (new value is less than old value) is handled by not adding anything per CPU clock tick.

This implies that a difference of N will take exactly N CPU clock cycles to be compensated for. It further implies that the timer cannot simply skip to any arbitrary value; its purpose is to provide a globally synchronous time view among different boards, either using an I^{2}C broadcast operation or through the employment of some distributed timing synchronization protocol. Note that if the board boot times are significantly different (e.g. due to DDR2 calibration or GTP links), a broadcasted soft reset (page 116) might be needed after all boards have booted to ensure counters are kept consistent.

The counter value is also present as a read-only copy inside each MBS core register (see page 92), in order to facilitate low-latency counter reads.

4.3.4 UART Registers

UART Control Register

31–11	10–8	7–1	0
	CID		I

Address offset 0x6_0000 [I^2C address: 0x60]

Reset value I=0, CID=0

Access type RW

Fields

CID *Interrupt Core ID.* If interrupts are enabled, selects which core will receive them. The selected core must still have the UART RX interrupt unmasked in its CPU Interrupt Register (page 70).

I *Interrupt Enable.* When 1, an interrupt will be generated towards the MBS block selected in the CID field whenever a new byte arrives in the UART receive fifo. The interrupt will be cleared when the selected core clears it in its own CPU Interrupt Register (page 70).

UART Status Register

31–27	26–16	15–11	10–0
	TX_FIFO_LEVEL		RX_FIFO_LEVEL

Address offset 0x6_0004 [I^{2}C address: 0x61]

Reset value TX_FIFO_LEVEL=1024, RX_FIFO_LEVEL=0

Access type R

Fields

TX_FIFO_LEVEL *Transmit FIFO Free Space.* Indicates the number of bytes that can be written to the UART Transmit FIFO without overflowing it. Writing more bytes than the level indicated here will result in ignoring the ones that do not fit.

RX_FIFO_LEVEL *Receive FIFO Used Space.* Indicates the number of bytes that have been received by the UART and need to be read by software. If the level reaches 1024, it means that newly arrived bytes from the UART will be lost.

UART Byte Access Register

31–8	7–0
	BYTE

Address offset 0x6_0008 [I^{2}C address: 0x62]

Reset value Undefined

Access type RW

Fields

BYTE *Read/Write Byte.* When reading, one byte will be returned from the UART Receive FIFO, assuming it is not empty. When writing, one byte will be pushed to the UART Transmit FIFO, assuming it is not full.

Reading from an empty FIFO will return gargabe; writing to a full FIFO is ignored. See previous page for determinig current FIFO levels.

UART Bulk Write Register

31–24	23–16	15–8	7–6	5–0
...	BYTE1	BYTE0		LENGTH

Address offset 0x6_0100 – 0x6_013C [Inaccessible through I^{2}C]

Reset value Undefined

Access type W

Fields

BYTE0 – BYTE63 *Bulk Write Data.* Performing a DMA to the register start address 0x6_0100, allows an atomic write of up to 63 bytes to the UART Transmit FIFO. This makes possible to enqueue short text lines from any core to the UART, without worrying about garbling the output. Fields BYTE0 to BYTE63 (spawning register addresses 0x6_0100 to 0x6_013C) contain the actual bytes to be enqueued to the FIFO.

LENGTH *Bulk Write Length.* Specifies how many bytes are going to be enqueued. The DMA size should be a whole cache line (64 bytes), but only LENGTH bytes of them will be actually enqueued to the FIFO.

4.3.5 Trace Registers

Trace Base Address Register

31–6	5	4	3	2	1	0
TRC_BASE	0	0	0	0	0	0

Address offset 0x8_0000 [I^{2}C address: 0x80]

Reset value 0

Access type RW

Fields

TRC_BASE *Trace Base Address.* Specifies the base address for the MBS Data Access tracing. The address must be 64-B aligned. Data traces will begin at this address. Data for an MBS block will only be included if the T bit in its CPU Control register (page 68) is enabled.

For changes in TRC_BASE to take effect, disable the trace collection altogether (setting the Enable bit in TRC_CONTROL to 0) and reenable it.

Trace Bound Address Register

31–6	5	4	3	2	1	0
TRC_BOUND	0	0	0	0	0	0

Address offset 0x8_0004 [I^2C address: 0x81]

Reset value 0

Access type RW

Fields

TRC_BOUND *Trace Bound Address.* Specifies the limit address for the MBS Data Access tracing. The address must be 64-B aligned. When enough data traces are gathered to reach this address, data tracing will stop.

For changes in TRC_BOUND to take effect, disable the trace collection altogether (setting the Enable bit in TRC_CONTROL to 0) and reenable it.

Trace 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
	E

Address offset 0x8_0008 [I^{2}C address: 0x82]

Reset value 0

Access type RW

Fields

E *Trace Enable.* Setting this bit from 0 to 1 (re)starts trace collection from TRC_BASE to TRC_BOUND address range. Setting the bit to 0 stops the tracing.

4.4 Formic Boot Procedure

All processors on a Formic board are disabled, except the first one (the one with Node ID = 0). When needed, the first core can wake up the others by sending a network message to their CPU Control registers and setting their E bit to 1.

All caches are disabled and dirty. Cache Clear operations must be issued before they can be activated.

There are two known ways to download code to the board DRAM:

1. *Through the JTAG, using the `XMD` tool.*

   After a program has already been loaded and run by the boot ROM, `XMD` can connect to an active processor through the board JTAG interface and download an ELF file directly using the processor's data interface, which passes normally through ART, DL1, L2C, MNI, XBI, XBAR and TLB paths. If all these blocks are in an appropriate mode to support this, the ELF will be successfully written to the DRAM. Note that if L2C is active, the ELF contents will just be cached there and will not reach the DRAM. Also note that XMD cannot connect to a blocked MicroBlaze, such as if it waits for a mailbox access or an interrupt. Finally, do not hard reset the board after this (e.g. by pressing the board button), because this will cause the boot ROM to overwrite the new ELF file.

   In short, to make this method work, it is recommended to have a boot ROM that has a minimal program which keeps caches disabled and the MircoBlaze busy-wait on a loop. Use the `XMD` commands below to stop the processor, download the ELF file "test.elf" and restart the processor using a soft reset:

   ```
   connect mb mdm -cable type xilinx_platformusb frequency 750000
                  -debugdevice deviceNr 1 cpunr 1

   stop

   dow test.elf

   run
   ```

2. *Patch the bitstream using the `data2mem` utility.*

   To patch an existing bitstream "in.bit", change the 8-KB boot ROM contents to that of the file "test.mem" and create a new bitstream "out.bit" with the new contents, use the `data2mem` utility as follows:

   ```
   data2mem -bm rom.bmm -bd test.mem -bt in.bit -o b out.bit
   ```

   where "rom.bmm" is a Xilinx `BMM` file that specifies which 4 BRAM blocks hold the boot memory contents. An example file is given here, supposing the 4 BRAM blocks are locked into RAMB16 sites X2Y16, X2Y18, X2Y20 and X2Y22:

   ```
   ADDRESS_SPACE boot_mem COMBINED [0x00000000:0x00001FFF]

     ADDRESS_RANGE RAMB16
       BUS_BLOCK
         i0_formic_bctl/i0_boot_mem/Mram_mem0 [31:0] PLACED = X2Y16;
       END_BUS_BLOCK;
     END_ADDRESS_RANGE;
   ```

```
    ADDRESS_RANGE RAMB16
      BUS_BLOCK
        i0_formic_bctl/i0_boot_mem/Mram_mem1 [31:0] PLACED = X2Y18;
      END_BUS_BLOCK;
    END_ADDRESS_RANGE;

    ADDRESS_RANGE RAMB16
      BUS_BLOCK
        i0_formic_bctl/i0_boot_mem/Mram_mem2 [31:0] PLACED = X2Y20;
      END_BUS_BLOCK;
    END_ADDRESS_RANGE;

    ADDRESS_RANGE RAMB16
      BUS_BLOCK
        i0_formic_bctl/i0_boot_mem/Mram_mem3 [31:0] PLACED = X2Y22;
      END_BUS_BLOCK;
    END_ADDRESS_RANGE;

  END_ADDRESS_SPACE;
```

The ROM contents of file “test.mem” in this case is a single address line (@0000) followed by a raw hexadecimal dump of the all ELF sections, dumped from address 0 and gap-filled as necessary. The beginning of such a file should look like this:

```
@0000
 B0000000
 B8080050
 B0000000
 B80800A0
 B0000000
 B80800A8
```

4.5 Formic MBS Interrupt handlers

Microblaze interrupts should be enabled (IE bit set to 1 in the MSR register) for the CTL interrupt handler to be able to work. Furthermore, the return from an interrupt should be done as follows:

- Clear the interrupt in the CPU Interrupt Register
- (possibly search for other interrupts at this point, by reading the CPU Status register)
- Clear the H bit from the CPU Control Register, if privileged/user mode separation is needed
- Clear the A bit from the CPU Control Register, if yield/blocked instructions are needed to be restarted (see below)
- Execute RTID instruction

By default, the return from the interrupt handler assumes that the interrupted instruction has been completed. The A bit in the CPU Control register specifies that a yield operation or a mailbox blocking read was aborted. If the interrupt handler needs to re-issue the aborted instruction, it must change the return address (stored in the special Microblaze register r14) to point to the correct instruction.

Normally, decrementing r14 by 4 is sufficient to point to the previous instruction. Exceptional cases are:

- If the aborted instruction was at the delay slot of a branch instruction, r14 will contain the branch target
- If the aborted instruction had above it an "imm" instruction, r14 must be corrected to point to this instruction; otherwise, instruction offsets will be wrong. Note that in this case, the "imm" instruction must be placed exactly in the previous instruction – no delay slots or other in-between instructions are valid in the Mircoblaze architecture.

It is suggested that yield and mailbox read operations are performed as assembly macros which do not present the above problems. Another (partial) solution is to make the interrupt handler inspect the instructions of the program and correct r14 accordingly.

Chapter 5

Test Plan

5.1 Testbench "formic_selftest"

This testbench does not contain any vital Formic block. It contains clock and reset logic, SRAM, DRAM and GTP controllers as well as self-validating test pattern generators (which use LFSRs to generate pseudo-random patterns which can be reused to verify written/sent data).

Testcase 01

Title:	**FPGA self-test simulation**
Description:	Depending on DIP switches values, read and write the SRAMs, the DRAM, and send packets over the GTP links.
Features tested:	

5.2 Testbench "formic_m1"

The testbench instantiates a single Formic board, which inside has a single MBS block glued to the TLB directly (without a crossbar).

Testcase 10

Title:	**Memory read, write, copy, no caches**
Description:	Caches remain disabled. Write 4 words to 0x1000 - 0x100C. Read them back and copy them to 0x1100 - 0x110C.
Features tested:	DL1-2

Testcase 11

Title:	**Memory read/write/copy, caches enabled**
Description:	Perform cache clear operations for IL1, DL1 and L2C. Enable the caches. Install ART entry #1 for whole memory (0x000 - 0xFFE) to be cacheable, privileged and executable. Write 16 words to 0x1000 - 0x103C. Read the 16 words back and write them to 0x1100 - 0x110C. Flush the L2C, so the cache line is written to DRAM.
Features tested:	DL1-4, DL1-6

Testcase 12

Title: **Software Interrupt test**

Description: Keep caches disabled.

Enable Microblaze interrupts and unmask software interrupt.

Write 3 words to 0x1000 - 0x1008.

Cause a software interrupt. Jump to interrupt handler, save the CPU_STATUS register onto 0x1100 + pos, where pos is kept at 0x1200 and incrementing each time interrupt handler is reached.

Write 3 more words to 0x100C - 0x1014 and cause another software interrupt.

Write 3 more words to 0x1018 - 0x1020.

Features tested:

Testcase 13

Title: **ART miss & permissions faults**

Description: Keep caches disabled. Dump ART entries reset values.

Reinstall only valid ART entry to cover addresses below 1MB. Make an access to address ¿1MB and receive an ART Miss fault. Take the interrupt, dump status and PC.

Install second valid ART entry for read-only region 1MB-2MB. Make a write access to such an address and receive an ART Permission fault. Take the interrupt, dump status and PC¿

Reinstall I/O page to be user accessible. Reinstall second ART entry to be writable, but only from privileged mode. Make a write access and expect to be allowed. Enter user mode, make the same access and expect an ART Permission fault. Take the interrupt (clearing the H bit) and return to user mode.

Make some normal accesses while still in user mode.

Features tested:

Testcase 14

Title: **MNI registers and simple DMA**

Description: Enable caches, reinstall ART read/write default page.

Initialize 2 cache lines starting at 0x1000.

Do a DMA from node 0, adr 0x1000 $\rightarrow$ node 0xC, adr 0x1200.

Do a DMA from node 0xC, adr 0x1200 $\rightarrow$ node 0, adr 0x1400.

Write all unused MNI registers (message and ack adr) and dump all MNI registers to 0x1100.

Do not flush the caches. The result should be that lines at 0x1000 and 0x1400 should only be found in the L2 dump, while the lines at 0x1200 should only be found in the DDR dump.

Features tested:

Testcase 15

Title: **Performance test**

Description: Enable caches, reinstall ART read/write default page.

Enable local timer to count down, dump both its value and the global timer value.

Clear all performance counters. Enable them as well as the trace interface.

Make lots of nops (to test instruction L1 hits), data reads (data L1 hits), data writes (no allocate in L1), data touches followed by writes (allocated L1 write throughs), and some for-loop small kernel for more randomness.

Dump all performance registers, as well as the timers. Verify visually that they are more or less consistent.

Flush the L2C, so all cache lines are written back to DRAM.

Features tested:

Testcase 16

Title:	**Blocking under interrupts, private timer**
Description:	Enable caches, reinstall ART read/write default page. Install a private timer interrupt after 100 cycles and yield using the CPU syscall register. The interrupt should wake us up, dump status and count which interrupt it is in a global variable. Reinstall timer interrupt and block on mailbox this time. The timer should again wake us up, aborting the mailbox read and setting the A bit in the CPU status register. Verify and dump all status. Especially while inside 2nd interrupt (noted by the global variable), make the program return to the aborted instruction by decrementing r14. Reinstall timer while in 2nd interrupt. Program should return to mailbox access and block again. 3rd timer interrupt should wake us up; dump all status and return normally this time (do not touch r14). Program should return after the mailbox access (effectively aborting the instruction); flush the L2C, so all cache lines are written back to DRAM and quit.
Features tested:	

Testcase 17

Title:	**Cache conflicts, L2 writebacks**
Description:	Enable caches, reinstall ART read/write default page. Fill 12 cache lines, spaced every 32 KB (adr 0x8000, 0x10000, 0x18000, ...) which all happen to conflict in the same L2 way. For each line, touch its two halves first by doing a read, so it’s allocated in L1 cache. Flush the L2C. L2 should have filled all 8 ways and caused 4 writebacks. Verify that all 12 lines are in the DRAM, while only 8 remain in the cache (random replacement policy).
Features tested:	

Testcase 18

Title:	**Byte enables**
Description:	Test uncacheable byte write accesses, by writing all 4 byte alignments at adress 0x1000 followed by a loop write of 16 consecutive bytes. Test uncacheable byte reads by reading all 0x1000 bytes and writing them to 0x1100. Enable caches, reinstall ART read/write default page. Test the same cacheable byte patterns, but writing at 0x1200 and then copying them back to 0x1300. Write different byte patterns in the already partially written 0x1000 (on different byte positions), testing that L1/L2 allocate the partially written lines while overwriting only the correct new byte positions. Copy some of the 0x1100 loop bytes using all possible halfword alignments (16-bit accesses) to 0x1400. Do a matrix-transpose double loop, which copies columns of 0x1200 positions to 0x1500 in a transposed format, using all 16 possible byte alignment combinations. Flush the L2C, so all cache lines are written back to DRAM.
Features tested:	

5.3 Testbench “formic_m8”

This testbench instantiates a single Formic board, which inside contains 8 MBS blocks interconnected using the crossbar. The TLB is also connected to 5 crossbar ports.

Testcase 20

Title:	**2 MBS Mailbox test**
Description:	Read core id; if core #0, wake up #1 by sending a message to its CPU control register. Both cores enable caches, reinstall ART read/write default page. Core #0: block on mailbox. Core #1: Initialize cache line 0x1200, send a 1-word message to #0, block on mailbox. Core #0: get word from mailbox, send 2 words reply to #1, block on mailbox. Core #1: get 2 words from mailbox, do a DMA from 1/0x1200 $\rightarrow$ 0/Mailbox. Block on mailbox. Core #0: get 16 words from mailbox, send 1 word message reply and block on mailbox. Core #1: get 1 word from mailbox, send 1 word reply, flush L2C and yield. Core #0: get 1 word from mailbox, flush L2C and quit.
Features tested:	

Testcase 21

Title: **2 MBS DMA/Acks test**

Description: Read core id; if core #0, wake up #1 by sending a message to its CPU control register.

Core #0: fill cache line 0x1240 in DDR. Enable caches and then fill 0x1200 in L2 and 0x1280 in L2. Block on barrier 0.[1]

Core #1: Enable caches and block on barrier 0. When unblocked, initialize counter #0 to receive 3 cache lines (192 bytes) and notify node 0/Mailbox. Do a DMA 0/1200 → 1/1300, acknowledges at 1/counter0. Block on mailbox.

Core #0: when unblocked from barrier 0, immediately block again on mailbox. We will awaken by the 1/counter0 trigger which will send a notification to our mailbox. Send a message to #1, flush L2C and block on mailbox.

Core #1: get 1 word from mailbox, flush L2C, send 1 word reply and yield.

Core #0: get 1 word from mailbox and quit.

Features tested:

[1] We are going to use "Barrier" here in the sense of "send 1 word message to other cores's mailbox; block on mailbox". Barrier N identifies the same number for both cores.

Testcase 22

Title: **DMA I/W/C flavours, Counter interrupt**

Description: Read core id; if core #0, wake up #1 by sending a message to its CPU control register. Both cores enable caches, reinstall ART read/write default page.

Core #1: set counter 9 to receive 6 cache lines (384 bytes), giving an interrupt. Yield.

Core #0: initialize 3 dual cache lines (start at 0x1000, 0x1100 and 0x1200, each 128 bytes). Program 3 DMAs, each 2 cache lines (128 bytes):

- 0/0x1000 → 1/0x2000, ack 1/counter9, with W bit set (write through on destination). The two lines should arrive at #1 L2, set to dirty, and written through to DRAM.
- 0/0x1100 → 1/0x2100, ack 1/counter9, with C bit set (clean on destination). The two lines should arrive at #1 L2, set clean, and not written to DRAM.
- 0/0x1200 → 1/0x2200, ack 1/counter9, with I bit set (ignore dirty on source). The two lines should be made clean at #0 L2, arrive at #1 and set dirty, and not written to DRAM.

Block on barrier 0.

Core #1: we should be wakened by the counter 9 interrupt. Record status and go to barrier 0.

Core #0: do two more DMAs:

- C/0x2000 → 0/0x1300, ack 0/mailbox. C/0x2000 should contain the written-through lines from the W bit DMA.
- 1/0x2100 → 0/0x1400, ack 0/mailbox.

Block on our mailbox, waiting for 4 acknowledgement words (1 per cache line). Go to barrier 1.

Core #1: Flush L2C and go to barrier 1. When reached, yield.

Core #0: Flush L2C and quit. Verify that 0/0x1200 is not dumped to DRAM: it must have been made clean by the I bit DMA. The same for 1/0x2100 (cleaned by the C bit DMA).

Features tested:

Testcase 23

Title: **DMA net Nack, local overflow exception**

Description: Read core id; if core #0, wake up #1 by sending a message to its CPU control register. Both cores enable caches, reinstall ART read/write default page.

Core #0: go to barrier 0.

Core #1: go to barrier 0.

Core #0: read #1 DMA fifo status by doing a DMA which returns to our mailbox. Prepare counter 3 to receive something very big (¿ 2 MB) and give an interrupt.

Do a first DMA 1/0x10000 $\rightarrow$ C/0x10000, size 1 MB, ack 0/counter3. This will keep core #1 DMA engine busy for a long period (greater than the end of this test).

While this DMA runs, start 32 more DMAs, size 64 bytes with the same arguments. The last one will not fit in the DMA fifo. It must send back a Nack, which will cause our counter to trigger.

Take the interrupt, read counter value and return to program.

Do a (local) DMA 0/0x10000 $\rightarrow$ C/0x10000, size 1 MB, no ack. This will keep our own DMA engine busy for a long period (greater than the end of this test).

While this DMA runs, start 32 more DMAs, size 64 bytes with the same arguments. Before starting each one, dump our own DMA fifo status level to DRAM. The last DMA must not fit in our own DMA fifo and must cause a DMA fifo full exception.

Take the interrupt, dump status, return to program, flush L2C and quit.

Features tested:

Testcase 24

Title: **Mailbox overflow, Nack, depth, interrupt**

Description: Read core id; if core #0, wake up #1 by sending a message to its CPU control register. Both cores enable caches, reinstall ART read/write default page.

Core #0: initialize cache line 0x1100 and go to barrier 0.

Core #1: prepare counter 3 to receive 1200 bytes and give an interrupt. Go to barrier 0.

Core #0: read #1 mailbox fifo level by doing a DMA that returns to our own mailbox. Send a 1-word mesage to mailbox #1, acknowledged to 1/counter3. Send a 2-word message, same ack. Start 16 DMAs 0/0x1100 $\rightarrow$ 1/mailbox, ack 1/counter3. Go to barrier 1.

Core #1: leave barrier 0 and immediately yield. We will be wakened by our own counter 3 which will trigger (giving an interrupt) because our mailbox cannot receive the 16^{th} cache line that #0 tries to send – the 1200 programmed bytes cannot be reached, the mailbox holds up to 1024 bytes.

Take the interrupt and record cause and counter value.

Read mailbox depth and dump all the mailbox contents up to this depth. We should dequeue 1 + 2 + (16 x 15) words: the 16^{th} cache line must have been discarded altogether. Clear and unmask mailbox interrupt and reach barrier 1, not with a blocking mailbox read but with yielding.

Core #0: send 2 more words to core #1 mailbox, flush L2C and reach barrier 2.

Core #1: leave barrier 1 through Mailbox interrupt. Record cause and return to program.

Dequeue the 2 new words, flush L2C, reach barrier 2 and then yield.

Core #0: quit the simulation.

Features tested:

Testcase 25

Title:	**8 MBS, L2C epochs, counter broadcast tree**
Description:	Core #0 wakes up the 7 other cores and then fills 8 ways of a cache set, after setting the current epoch to 1. Other cores initialize one cache line. All cores to a counter-based tree barrier, where in 3 levels counters notify two children counters of other cores. A reverse tree is used to broadcast the barrier completion. Core #0 waits for the final barrier counter using an interrupt and yielding the CPU. It changes the epoch number to 2. Cores #1-7 send their one cache line to #0. They also initialize a new cache line. Second barrier. Cores #1-7 send the new cache line to #0. Core #0 records how many lines were accepted/rejected for epoch 1. Third barrier. Core #0 records how many lines were accepted/rejected for epoch 2.
Features tested:	

Testcase 26

Title:	**UART test**
Description:	Sends some bulk writes and some byte writes to the UART. In the m8 testbench, the UART RX is looped back from the UART TX, so the test sets up to wait for a UART interrupt, the CPU yields and is awaken by the interrupt. The first byte is read and the interrupt is cleared.
Features tested:	

Testcase 27

Title: **8 MBS mem copy, Xbar contention/fairness**

Description: 7 MBS cores are woken up by core #0.

All 8 cores issue a 16-KB DMA from their cache to the DRAM (node 0xC) and they await for the DMA completion using a counter.

The testcase requires visual inspection to see if there are starvation/fairness-related issues in the 8 MBS access to the network.

Features tested:

Testcase 28

Title:	**TLB translation test, caches off-on-off**
Description:	Write and read back some TLB entries to verify accessibility. Create a linear page table for the first 16 pages in memory, and bulk-write it using a DMA to the TLB. Enable the TLB translation. Write a word in each of the 16 pages. Flush L2 and then turn off the caches so we can observe translation changes of existing pages. Change some TLB entries in memory so that they point to different pages; update the TLB with a DMA. Read back the 16 words in the 16 pages and verify the ones with the changed translation are remapped correctly. Turn of the TLB translation. Read back the 16 words in the 16 pages and verify that all of them refer to the original mapping.
Features tested:	

Testcase 29

Title:	**8 MBS cache overwrites**
Description:	7 MBS cores are woken up by core #0. A data-flow processing chain is set up in a loop. Core #0 initializes 32 data chunks of 4 cache lines. Each core receives a buffer from the previous core, processes it to contain the core's signature, and forwards it to the next core using a cache-to-cache DMA. Triple buffering is used by each core, and mailbox events for software "credits" (means there are buffers to accept new chunks) and "notifications" (means that an existing chunk has been sent to the next core) are used for flow control. Core #7 dumps all the chunks in memory.
Features tested:	Overwrite of same buffer space

Testcase 31

Title:	**L2C Write on dirty line under blocked Fill**
Description:	Core #0 wakes up #1, initializes a buffer and sends it to #1. It then blocks on its mailbox, but with a tuned number of `nop` instructions so that the mailbox access coincides with an instruction L2 miss, which is blocked because the data mailbox access is blocked. Core #1 receives the buffer, changes it and sends the modified one back to #0. The buffer address is chosen to happen in the same set as the blocked L2 instruction miss of core #0. Core #0 unblocks and completes the program.
Features tested:	L2 blocked miss correct way selection

Testcase 32

Title:	**Pull DMAs with cache replacements**
Description:	7 MBS cores are woken up by core #0. Each core initializes 32 cache lines, all of which are in the same L2C set. Barrier (mailbox-based). Each core issues 32 * 8 single-line DMAs to pull all the cache lines from all other cores to its own cache, in some semi-random ordering. Each cache line is checksummed. After all cores are done, they must all have all the lines and the same checksum.
Features tested:	

5.4 Testbench “formic_m8g8”

This testbench instantiates a single Formic board, which inside contains 8 MBS blocks, 8 GTP blocks and the full 22x22 crossbar with 5 TLB ports and 1 board controller port.

Testcase 30

Title:	**formic_m8g8 reset test**
Description:	Visual-only test to verify GTP links reset procedure, soft and hard resets.
Features tested:	

5.5 Testbench "2x_formic_m8g8"

This testbench instantiates two Formic board, each of them with 8 MBS blocks, 8 GTP blocks and the full 22x22 crossbar with 5 TLB ports and 1 board controller port. The boards are linked with connectors SATA4 and SATA7, i.e. along the X axis.

Testcase 40

Title:	**2 Formic boards communication test**
Description:	A 1-cache line DMA is ping-ponged between the two boards to verify the board connectivity.
Features tested:	

Appendix A: Formic MBS Registers Summary

CPU Control & Interrupts Registers

CPU System Call Entry Register [0x0_0000]

Bits	Field
31–9	
8	Y
7–1	
0	S

CPU Control Register [0x0_0004]

Bits	Field
31–26	
25	T
24	M
23–17	
16	I
15–11	
10	A
9	H
8	P
7–1	
0	E

CPU Status Register [0x0_0008]

Bits	Field
31–24	BID
23–16	MI
15–13	
12–8	NMI
7–3	
2–0	CID

CPU Interrupt Register [0x0_000C]

Bits	Field
31–24	
23–16	MCLR
15–13	
12–8	NMCLR
7–0	MASK

Address Region Table Registers

ART Entry #0 Register (Compulsory I/O region) [0x2_0000]

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
0	1	0	0	U	1		1	BOUND=BASE	BASE

ART Entry #1 Register [0x2_0004]

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
C	0	R	X	U	P		V	BOUND	BASE

ART Entry #2 Register [0x2_0008]

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
C	0	R	X	U	P		V	BOUND	BASE

ART Entry #3 Register [0x2_000C]

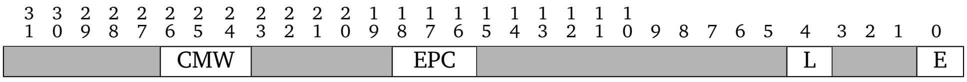
31	30	29	28	27	26	25	24	23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
C	0	R	X	U	P		V	BOUND	BASE

ART Entry #4 Register [0x2_0010]

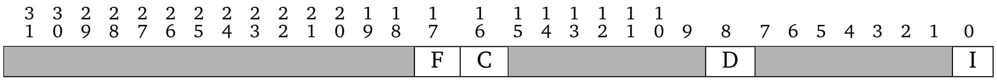
31	30	29	28	27	26	25	24	23 22 21 20 19 18 17 16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
C	0	R	X	U	P		V	BOUND	BASE

Cache Hierarchy Registers

Cache Control Register [0x4_0000]



Cache Maintenance Register [0x4_0004]



Performance Monitoring Registers

Instruction Level 1 Cache Monitor [0x6_0000]

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
IL1_MISSES	IL1_HITS

Data Level 1 Cache Monitor [0x6_0004]

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
DL1_MISSES	DL1_HITS

Level 2 Instruction Monitor [0x6_0008]

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
IL2_MISSES	IL2_HITS

Level 2 Data Monitor [0x6_000C]

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
DL2_MISSES	DL2_HITS

Registers Access Monitor [0x6_0010]

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
NET_REGS	CPU_REGS

Network Operations Monitor [0x6_0014]

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
REMOTE_OPS	LOCAL_OPS

Network Level 2 Reads Monitor [0x6_0018]

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
READ_MISSES	READ_HITS

Network Level 2 Writes Monitor [0x6_001C]

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
WRITE_FAILURES	WRITE_SUCCESSES

Network Packets VC #0 Monitor [0x6_0020]

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
VC0_OUT																VC0_IN															

Network Packets VC #1 Monitor [0x6_0024]

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
VC1_OUT																VC1_IN															

Network Packets VC #2 Monitor [0x6_0028]

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
VC2_OUT																VC2_IN															

CPU Energy Monitor [0x6_002C]

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
CPU_ENERGY																															

CPU Power Monitor [0x6_0030]

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
																CPU_POWER															

Timers

Global Timer [0x8_0000]

<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">GLOBAL_TMR</td></tr>
</table>

Private Timer [0x8_0004]

<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>E</td><td>I</td><td colspan="30">PRIVATE_TMR</td></tr>
</table>

Network & DMA Engine Registers

MNI Opcode Register [0xA_0000]

Bits	31–28	27–20	19–16	15–6	5	4	3	2	1–0
Field		DST_BRD	DST_ND		I	W	C	A	OP

MNI DMA Size Register [0xA_0004]

Bits	31–21	20–2	1	0
Field		SIZE	0	0

MNI Destination Address Register [0xA_0008]

Bits	31–2	1	0
Field	DST_ADR	0	0

MNI Board/Node Register [0xA_000C]

Bits	31–28	27–20	19–16	15–12	11–4	3–0
Field		SRC_BRD	SRC_ND		ACK_BRD	ACK_ND

MNI Acknowledgement Address Register [0xA_0010]

Bits	31–2	1	0
Field	ACK_ADR	0	0

MNI Source Address Register [0xA_0014]

Bits	31–2	1	0
Field	SRC_ADR	0	0

MNI Message Word 0 Register [0xA_0018]

Bits	31–0
Field	MSG0

MNI Message Word 1 Register [0xA_001C]

Bits	31–0
Field	MSG1

MNI Status Register [0xA_0020]

<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="10"></td><td colspan="6">NET_OPS</td><td colspan="10"></td><td colspan="6">CPU_OPS</td></tr>
</table>

Counter Registers

Counter Interrupt Register [0xC_0000]

31–30	29–24	23–22	21–16	15–14	13–8	7–6	5–0
	INT3		INT2		INT1		INT0

Counter #0 Value Register [0xC_1000]

31–0
VAL

Counter #0 Notification Address #0 Register [0xC_1004]

31–0
NTF_ADR0

Counter #0 Notification Address #1 Register [0xC_1008]

31–0
NTF_ADR1

Counter #0 Notification Board/Node Register [0xC_100C]

31–29	28	27–20	19–16	15–13	12	11–4	3–0
	V1	NTF_BRD1	NTF_ND1		V0	NTF_BRD0	NTF_ND0

Mailbox Registers

Mailbox Access Register [0xE_0000]

31–0
ACCESS

Mailbox Status Register [0xE_0004]

31–29	28–18	17	16	15–13	12–2	1	0
	WR_MARK	0	0		RD_MARK	0	0

Mailslot Access Register [0xE_0008]

31–0
ACCESS

Mailslot Status Register [0xE_000C]

31–3	2	1	0
	F	0	0

Appendix B: Formic Board Registers Summary

Board Maintenance Registers

Board Control Register [0x0_0000] [I^{2}C: 0x00]

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
L11	L10	L9	L8	L7	L6	L5	L4	L3	L2	L1	L0		LM		S	H

Board Status Register [0x0_0004] [I^{2}C: 0x01]

31–12	11–8	7–0
	DIP	BID

Board Link Status Register [0x0_0008] [I^{2}C: 0x02]

31–24	23–16	15–8	7–0
CRC_ERROR	CREDIT_ERROR	LINK_ERROR	LINK_UP

Board Cores Status Register [0x0_000C] [I^{2}C: 0x03]

31–28	27–24	23–20	19–16	15–12	11–8	7–4	3–0
C7	C6	C5	C4	C3	C2	C1	C0

Board DRAM Status Register [0x0_0010] [I^{2}C: 0x04]

31–4	3–0
	ERROR

TLB Registers

TLB Control Register [0x2_0000] [I^{2}C: 0x20]

31–1	0
(reserved)	E

TLB Entry #0 Register [0x2_1000] [I^{2}C: –]

31–9	8	7	6–0
(reserved)	V	(reserved)	PA

Timer Registers

Board Global Timer [0x4_0000] [I^2C: 0x40]

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
GLOBAL_TMR																															

UART Registers

UART Control Register [0x6_0000] [I^2C: 0x60]

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
																					CID										I

UART Status Register [0x6_0004] [I^2C: 0x61]

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
					TX_FIFO_LEVEL																RX_FIFO_LEVEL										

UART Byte Access Register [0x6_0008] [I^2C: 0x62]

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
																								BYTE							

UART Bulk Write Register [0x6_0100 – 0x6_013C] [I^2C: –]

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
...								BYTE1								BYTE0										LENGTH					

Trace Registers

Trace Base Address Register [0x8_0000] [I^2C: 0x80]

31–6	5	4	3	2	1	0
TRC_BASE	0	0	0	0	0	0

Trace Bound Address Register [0x8_0004] [I^2C: 0x81]

31–6	5	4	3	2	1	0
TRC_BOUND	0	0	0	0	0	0

Trace Control Register [0x8_0008] [I^2C: 0x82]

31–1	0
	E