GPC_4T:
HAS

GPC_4T High-Level Architecture Specifications

Adi Levy (adi.levy@campus.technion.ac.il)

Saar Kadosh (skadosh@campus.technion.ac.il)

Contents

Revision History

Rev. No.	Who	Description	Rev. Date
0.1	*Saar Kadosh*	*Initial GPC_4T HAS*	*06 October 2021*
0.3	*Saar Kadosh*	*Add more data*	*30 November 2021*
0.4	*Saar Kadosh*	*implement ABD notes*	*15 December 2021*
0.5	*Adi Levy*	*design edits*	*25 December 2021*

Figures

Tables

1 General Description

1.1 High Level Definition

In this High-Level Architecture Specification Document (HAS) we will describe the specification of the GPC_4T IP – a RISCV Core with 4 HW Threads. The document will explain the macro design of the GPC_4T, the use and functionality of the different components of the GPC_4T and the features it supports to perform as a LOTR component.

This document is the architectonic document, and it is a macro view file of the project. To fully understand the micro level architecture, please read the MAS file as well.

1.2 General Background

1.2.1 RISCV & ISA Background

RISC-V is an open-source base-integer **instruction set architecture** (ISA) based on established reduced instruction set computer (RISC) principles. It is a classic RISC architecture rebuilt for modern times. At its heart is an array of 32 registers containing the processor's running state, the data is immediately operated on, and housekeeping information. RISC-V comes in 32-bit and 64-bit variants, with register size changing to match. The project began in 2010 at the University of California, Berkeley along with many volunteer contributors not affiliated with the university. It was originally designed to support computer architecture research and education but eventually in nowadays used for industry and many other uses.
The RISC-V eco system has all the SW needed to program, compile and creating RISC-V assembly & executable RISC-V machine code.
Unlike other ISAs, anyone can write compatible RISC-V Core without going through a bureaucracy of licenses and fees.
GPC_4T Supports the RV32I ISA that can be found in in this link - RVspec.

RV32I Base Instruction Set:						
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]	**rs1**		000	rd	1100111	JALR
imm[12|10:5]	**rs2**	**rs1**	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	**rs2**	**rs1**	001	imm[4:1|11]	1100011	**BNE**
imm[12|10:5]	**rs2**	**rs1**	100	imm[4:1|11]	1100011	**BLT**
imm[12|10:5]	**rs2**	**rs1**	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	**rs2**	**rs1**	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	**rs2**	**rs1**	111	imm[4:1|11]	1100011	BGEU
imm[11:0]	**rs1**		000	rd	0000011	LB
imm[11:0]	**rs1**		001	rd	0000011	LH
imm[11:0]	**rs1**		010	rd	0000011	LW
imm[11:0]	**rs1**		100	rd	0000011	LBU
imm[11:0]	**rs1**		101	rd	0000011	LHU
imm[11:5]	**rs2**	**rs1**	000	imm[4:0]	0100011	SB
imm[11:5]	**rs2**	**rs1**	001	imm[4:0]	0100011	SH
imm[11:5]	**rs2**	**rs1**	010	imm[4:0]	0100011	SW
imm[11:0]		**rs1**	000	rd	0010011	ADDI
imm[11:0]		**rs1**	010	rd	0010011	SLTI
imm[11:0]		**rs1**	011	rd	0010011	SLTIU
imm[11:0]		**rs1**	100	rd	0010011	XORI
imm[11:0]		**rs1**	110	rd	0010011	ORI
imm[11:0]		**rs1**	111	rd	0010011	ANDI
00000000	**shamt**	**rs1**	001	rd	0010011	SLLI
00000000	**shamt**	**rs1**	101	rd	0010011	SRLI
01000000	**shamt**	**rs1**	101	rd	0010011	SRAI
00000000	**rs2**	**rs1**	000	rd	0110011	ADD
01000000	**rs2**	**rs1**	000	rd	0110011	SUB
00000000	**rs2**	**rs1**	001	rd	0110011	SLL
00000000	**rs2**	**rs1**	010	rd	0110011	SLT
00000000	**rs2**	**rs1**	011	rd	0110011	SLTU
00000000	**rs2**	**rs1**	100	rd	0110011	XOR
00000000	**rs2**	**rs1**	101	rd	0110011	SRL
01000000	**rs2**	**rs1**	010	rd	0110011	SRA
00000000	**rs2**	**rs1**	110	rd	0110011	OR
00000000	**rs2**	**rs1**	111	rd	0110011	AND

Table 1 - RV32I ISA

1.2.2 CPU (HW Arch)

General-Purpose CPU is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program.
Principal components of a CPU include the arithmetic–logic unit (ALU) that performs arithmetic and logic operations, processor registers that supply operands to the ALU and store the results of ALU operations, and a control unit that orchestrates the fetching (from memory), decoding and execution of instructions by directing the coordinated operations of the ALU, registers and other components.

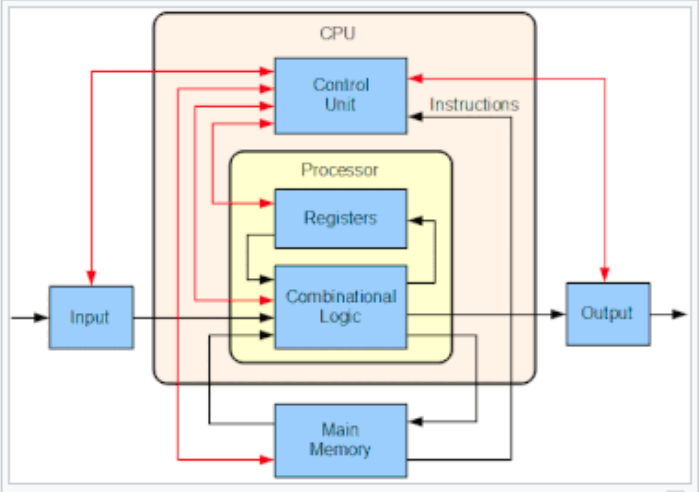

Figure 1- - Block diagram of a basic uniprocessor-CPU computer [image from Wikipedia]

1.2.3 Multi-Threading

Multithreading is the ability of a central processing unit (CPU) to provide multiple threads of execution concurrently. All the threads share many resources, but each Thread also has its own resources (like registers).
Thread has its own PC (Sequencer) + registers + stack. All threads (within a process) share same address space. The idea is to Improve throughput on multiple thread workloads and maximize utilization.
In GPC_4T we have 4 threads working in "Fine Grained Multithreading" method – each clock cycle the running thread changed in cyclic way.

Figure 2 - Fine Graine MT illustration. each thread different color

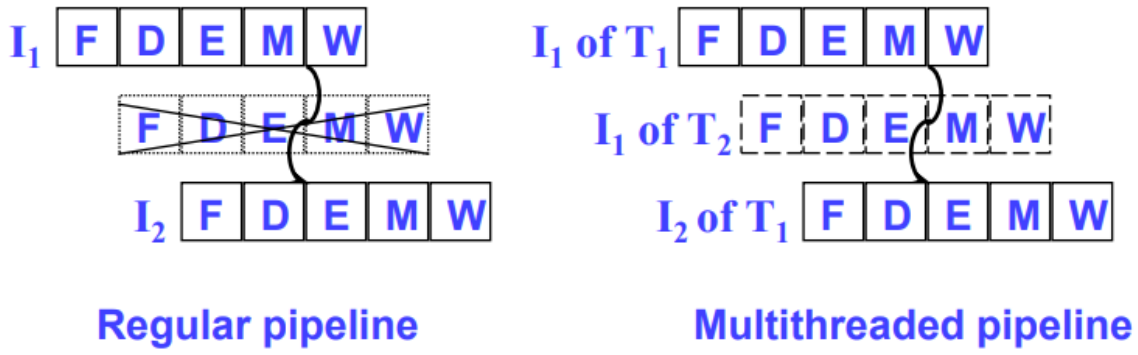

Figure 3- Data hazard elimination in Fine Grain Multithreading

1.2.4 GPC_4T within the LOTR project

The LOTR project - "Lord of The Ring" also known as "***THE FABRIC***", is the top-level project that will use the GPC_4T alongside with a Ring Controller to create a Multi-Core Ring Fabric architecture.
Ring Controller – also known as RC, is the arbitration logic for Core & Fabric transactions. The RC maintains the ordering coherency, starvation & Dead Lock prevention. Contains the C2F& F2C Buffers to arbitrate the current Fabric<->Core interface.
LOTR Tile – A single unit that will contain the GPC_4T Core , Memory, and an RC unit.
Designate to connect many other LOTR Tiles in a ring connection to create the Multi-Core Ring architecture.

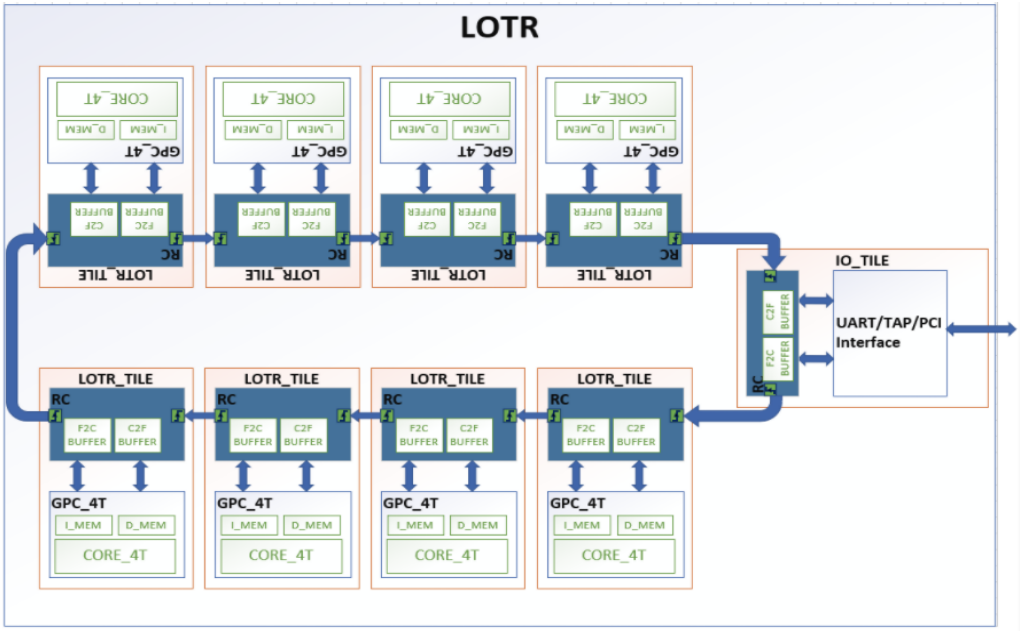

Figure 4 – LOTR Architecture

1.3 TOP LEVEL DESCRIPTION

GPC_4T, which all of this project converse to, is our own implementation of a General-purpose compute unit based on RISC-V32I ISA Core which supports 4 hardware threads, and memory modules.
The module is one of two main components in the **LOTR Tile** (the other is the Ring Controller).
Several of LOTR Tiles creates the LOTR Fabric, described above.
The GPC_4T transactions to the Fabric (For example reading a memory of a different GPC_4T unit in the ring Fabric) are **made through the Ring controller** unit located in the same Tile as the GPC_4T.

- GPC_4T includes 3 Main Building Blocks: Core, Instruction Memory wrapper and Data memory wrapper.
- The entire module and building blocks architecture based on synchronic pipeline architecture.
- The Pipeline is In-Order pipeline (instructions performs by their order in the program) .
- The pipeline is single scalar : only one instruction is executed per clock cycle, unlike "super-scalar" pipeline.
- Core: 5 staged pipeline RV32I core that support 4 threads by Hardware design.
- Instruction Memory wrapper : AKA I MEM WRAP:
    - Contains the Instruction Memory which the core reads the 32bit RISC-V32I instructions from
    - Logic design handle Fabric communication.
- Data Memory wrapper : AKA D MEM WRAP:
    - contain the Data Memory on which the core can use to initiate load and store commands
    - Contains Control Registers memory region (CR) which supposed to fill a major role and holds data directly effects on the behavior of the design.
    - Logic that handles Fabric communication.
- 2 types of Standard Interface with the Ring Controller:
    - F2C : Fabric to Core interface
    - C2F : Core to Fabric interface.
- Communication between Core and Ring made only through the memory wrappers
- Compatible with the RISCV ISA - RV32I
- Style of multithreading ("Fine Grained Multithreading”): every cycle a different thread enters the pipe and imports the next command from instruction memory
- Steady State Instructions Per Cycle (IPC) of 1. (1/4 for each thread).

1.4 GPC_4T BLOCK DIAGRAM

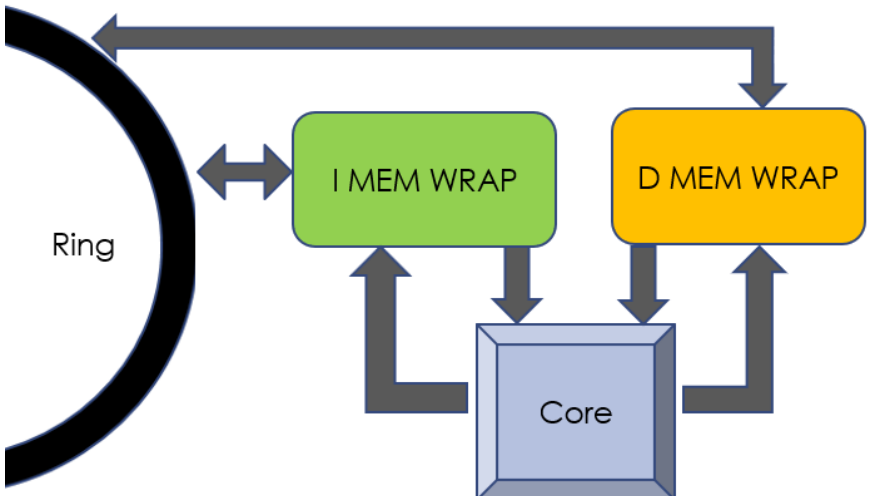

Figure 5 - Very High level GPC_4T block diagram

2 Design Building Blocks

2.1 Core & Pipe Stages

The core is the building block that runs and executes the program instructions. It is a RV32I based core, which means the core entire design supports only RISC-V32I instructions – 32bits encoded assembly instructions. The Core supports multi-thread of 4 threads based on Hardware.

The Core contains a **hardware scheduler** defines which thread is currently runs on each pipe stage.

GPC_4T core is a piped core with 5 stages:
First stage – Q100H:
Instruction Fetch. On this stage the core maintains 4 different program counters (PC) – one for each thread. The Next Pc calculated on stage Q102H. In this stage a read instruction memory transaction to the I mem Wrap performed with the Pc as the instruction address. Each thread on its turs sends the I mem Wrap its PC.
Second stage – Q101H:
Decode. The I mem response with the 32bit encoded instruction arrives to this stage. On this stage the instruction is decoded to extract information from the instruction such as the Opcode of the commands, the designated registers numbers, relevant address etc. depending on the instruction. This stage contains 4 register files (each register file is a 32X32), one for every thread. The relevant registers data is read in this stage
Third stage – Q102H:
Execute stage. Where the data calculations, comparisons, PC and branch calculations etc. occur. This stage contains the logic to do the calculations such as the ALU, ALU controllers, ALU inputs selection, branch resolution, and next PC select.
Fourth stage – Q103H:
Memory Access, where the core requests to read or write data from/to the data memory of the core or a different core in the ring.
Fifth stage – Q104H:
Write Back, where the relevant data (from previous calculation or from the memory access) is written to the registers. This stage selects the data between the possible Write Back options (Pc+4 calculation, Data returned from the memory, ALU output) to be returned to the register file and based on the thread, the relevant register file is chosen.

2.2 Core Block Diagram

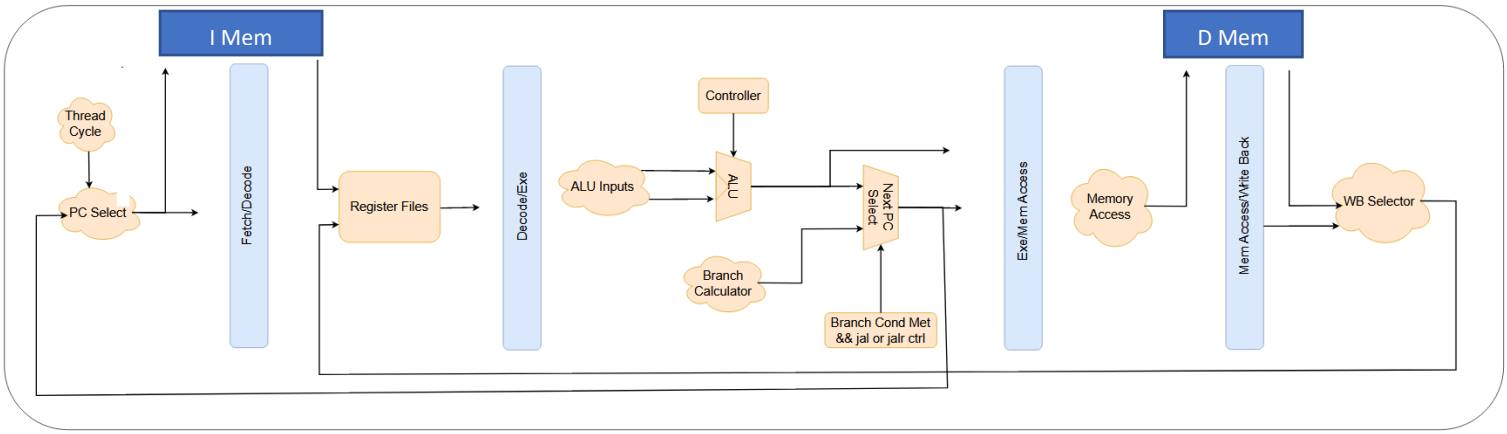

Figure 6 - Pipe Stages and the main blocks of every stage

2.3 MEMORY REGIONS

GPC_4T has total of 8KB of S-RAM memory in addition to Flip-Flop memories.
The entire memory it divided into 3 regions, which sometimes refer as memory modules:

1. I_MEM
2. D_MEM
3. CR_MEM.

The memory regions can be accessed from within the current GPC_4T core or from other cores in the ring for communication.

2.3.1 The Address Space

Each memory transaction has a 32-bit address.
The address may be a "Local" address or a "Peer" address.

Core ID	Region	Reserved	Memory Offset
Address[31:24]	Address[23:22]	Address[21:14]	Address[11:0]

Table 2 - address structure

Memory Offset:
Address[11:0] - 12 bit offset to support up to 4KB of D_MEM/I_MEM of each core. (8 KB = 4KB D_MEM + 4KB I_MEM) In case we want as many cores as possible we might want to use bits [10:0] 2KB of memory. (2KB Data + 2KB Instruction = 4KB)

Region:
Address[23:22] - 2 bit to determine the request destination.
2'b00 - I_MEM (instruction memory) - SRAM.
2'b01 - D_MEM (Data memory) - SRAM.
2'b11 - CR (Control Register) - Flip-Flop.

CORE ID:
Address[31:24] - 8 bit unique ID - Each core in the LOTR fabric has a unique ID.
We reserve the 8'b0000_0000 & 8'b1111_1111 ID's for special use.

Reserved: In our current design, we limit our memory and we won't use the MSB bits to map memory within the LOTR Fabric. These bits are for future purposes.

2.3.1.1 Address Aliasing

Using the reserved CORE_ID we can access the cores local memory with LOAD/STORE with 3 different addresses:

1. 8'b????_???? - LOAD/STORE Using the correct local core ID. (which is stored in the CR-Control Register).
2. 8'b0000_0000 - LOAD/STORE request with a '0 in the "core ID" - the request will always "match" and access the local memory - D_MEM or CR.
3. 8'b1111_1111 - STORE (not LOAD) request has a '1 "core ID" - the request will "match" and access the local memory +
   the request will be sent to the fabric and will match all the other cores on the Fabric.
   when the request returns back to the ring it originated from the request will be "killed".
   this allows to "broadcast" a write data to all the cores in the fabric that originated from a single request.

2.3.1.2 Core's Point Of View

Each core has continuous address space.
Example:

Region	from address	to address	Description
I_MEM	0x00**0**00000	0x0000_0FFF	4KB of Instruction Memory
D_MEM	0x00**4**00000	0x0040_0FFF	4KB of Data Memory
CR_MEM	0x00**C**00000		CR Space offset – flop based

A requests with core ID = 0 (address[31:24] == 8'b0) will allows the core to access the D_MEM & CR. If the CORE LOAD/STORE request comes with the *correct* CORE_ID (address[31:24] == 8'b<core_id>), the request must match and allows access to the local memory. Having the CORE ID = '1 (address[31:24] == 8'b1111_1111) on a STORE request will allow the core to access to the D_MEM & CR (this request will be sent to the fabric to access all the other cores local memory. CORE ID = '1 with a LOAD request will return only the Local memory and will NOT be send to fabric).

2.3.1.3 Fabric Point Of View

Each core has an ID which the RD/WR request must match to access.
When a Core sends a LOAD/STORE request that the ID does not match the local core ID (and the ID is not '0) the request will be sent to the Fabric.
In case of a LOAD - the thread will stall until we get the data response. incase of write - the thread will continue regularly.
due to the "forward progress" of the "STORE" (writes) in the fabric, different requests LOAD/STORE (Read/writes) will never "pass" the "STORE" and Execution order will be maintained
due to stalling a thread that sent a LOAD (Read), no other LOAD/STORE (reads/write) will be sent from the thread -> Execution order will be maintained.
A request will access a core on the fabric when the core ID matches (address[31:24] == 8'b<core_id>).
In case of a write matches '1 (address[31:24] == 8'b1111_1111), the write will match and access the Cores local memory and continue to the next core.
Once the request will find its originator the WR request will be terminated.

2.3.2 I_MEM

I_MEM is the memory region that contains the instructions of the program.
It is SRAM based and 4KB in size in default.
All the threads share the same program and the division of the program to each thread is done by Software.
The program is loaded to the I_MEM from fabric using F2C Write requests.
The Memory Size is control by parameter the parameter "SIZE_I_MEM" and its basic size is 4KB
The memory itself can be access in the same cycle both from RC (Fabric) and from Core – dual port.
Typically, the Fabric will "write" to the I_MEM to load the program, but it can also Read in some cases.
The core can only send a "READ" request to the I_MEM.

2.3.3 I_MEM_WRAP

I MEM WRAP is a wrapper that instantiates the I Mem module. It has interface to handle core read request and Ring write and read request by transferring it to the I mem instance.

2.3.4 D_MEM

The memory region that contains the data of the program.
The data is loaded and extracted from D_MEM or stored to D_MEM by the core of the GPC_4T and can also be accessed by the Fabric in the same cycle – Dual port. D_MEM contains data for each thread at a different address location and a shared region where data that is relevant to all threads can be written and read. Threads can also access different Threads D_MEM regions when correctly updating D_MEM address and Core ID.
D_MEM size is 4KB and is defined by the variable SIZE_D_MEM.
Each thread's default stack base offset defines 512B of separate data for each thread.
SHARED_MEM size is 2KB and is defined by the variable SIZE_SHRD_MEM.

2.3.5 CR_MEM

CR_MEM is the control registers region memory. It is a flip-flop based that contains essential registers for the core operations. Some CRs data has direct influence on the core behavior. Other CR flops holds "setting" data. CR data can be accessed through the software by requesting the specific offset of the specific CR in the CR Table. Some CRs are read only that sample data from the hardware. Others are read and write CRs. Some of the CRs signals effects on the core and can freeze or reset specific thread's PC. The CRs are also dual port and can be accessed by the core and by the ring in the same cycle.

2.3.6 D_MEM_WRAP

D_MEM_WRAP is a wrapper that encapsulates the D_MEM and the CR_MEM. It also contains the logic to handle non-local memory transactions and an interface to handle Fabric(RC) requests.
This module also contains many comparation to check the request address using the hardware and passing the transaction to the designated module.
At the end of the cycle, depends on the address (which indicated the request) the D_MEM_WRAP will determine which data source to transfer the Core or the Ring Controller : Local data memory, CR memory data or non-local memory data from the Fabric.
data will get the Req/Rsp from the Fabric and the Core, match them with the Core ID and based on those matches D_MEM_WRAP will pass Rsp with data or Req to the Fabric or sent the Core the data requested the relevant Memory Region. D_MEM_WRAP is the interface of the Core with the Fabric. Req from the Core that don't match the Core ID in D_MEM_WRAP will proceed to the Fabric.

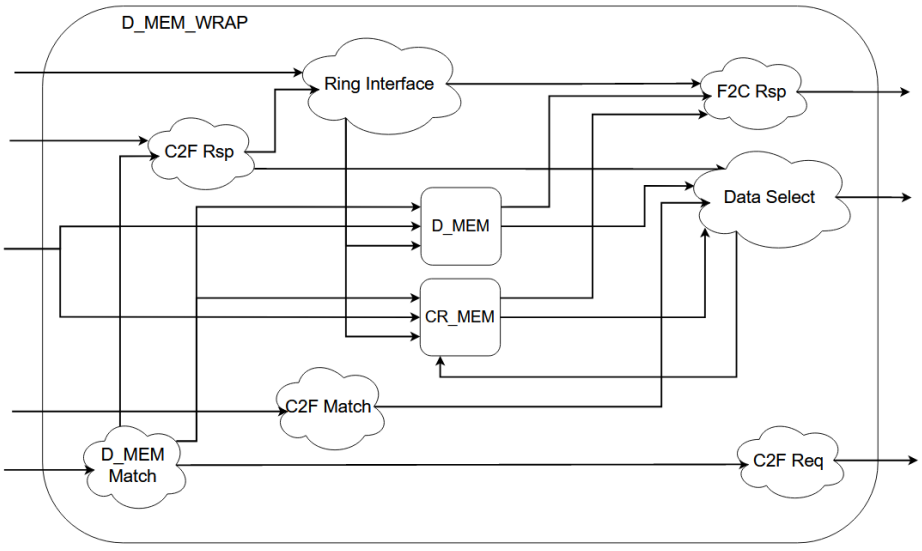

Figure 7 - D_MEM_WRAP in high level

3 SUPPORTED FEATURES

For the ring to operate as designed, each GPC_4T needs to support features that allows it to communicate between thread and with other cores.

- Reading from other cores
- Writing to other cores
- Freeze Thread from SW (CR_THREAD<id>_PC_EN)
- Stalling and signaling different threads

3.1 CR TABLE & FUNCTIONALITY

CR Registers:

Offset	Name	Type	Description	Structure	Comment
0	WHO_AM_I	RO	Current threads {core ID, thread ID)	[15:0]	
4	CR_THREAD_ID	RO	Current threads thread ID	[1:0]	The thread id value will be according to the thread doing the load(RD)
8	CR_CORE_ID	RO	Current threads core ID	[7:0]	
C	CR_STACK_BASE_OFFSET	RO	Threads Base offset SP `THREAD\<i\>_STACK_BASE_OFFSET`	[15:0]	
10	CR_TLS_BASE_OFFSET	RO	Threads Local Storage `THREAD\<i\>_TLS_BASE_OFFSET` (Heap)	[15:0]	chunk of data for thread-local runtime data (e.g. heap
14	CR_SHARED_BASE_OFFSET	RO	D_MEM shared Data region	[15:0]	
18	CR_I_MEM_MSB	RO	Number of bits required to Instruction Memory	[7:0] MEMORY_OFFSET_BITS	`MEMORY_OFFSET_BITS` indicates the number of bits in D_MEM/I_MEM offset
20	CR_D_MEM_MSB	RO	Number of bits required for Data Memory	[7:0] MEMORY_OFFSET_BITS	`MEMORY_OFFSET_BITS` indicates the number of bits in D_MEM/I_MEM offset
24	CR_SUPPORTED_ARCH	RO	What RV extensions are supported	[7:0]	
110 **114** **118** **11c**	CR_THREAD<id>_STATUS	RO	Freeze, Running, Reset, Exception, Stall (LoadFromFarMem)	[7:0]	0=Freeze, 1=Running, 2=Reset, 3=Freeze&Reset, 4=Freeze&Exception, 5=Running&Exception 6=Stall(LoadFromFarMem)
120 **124** **128** **12c**	CR_THREAD<id>_EXCEPTION_CODE	RO	see Exception code table	[31:0]	Each bit will be a Exception event - in case the exception event is non-distructive multiple Exception may be set
130 **134** **138** **13c**	CR_THREAD<id>_PC	RO	PC of thread `i`	[31:0]	
140 **144** **148**	CR_THREAD<id>_PC_RST	RW	When set the Thread PC will be reset to '0'	[0]	

14c					
150 154 158 15c	CR_THREAD<id>_PC_EN	RW	When set the Thread PC Can "run". HW Exception may set PC_EN=0	[0]	
160 164 168 16c	CR_THREAD<id>_DFD_REG_ID	RW	Number of register to read from thread `i`	[4:0]	Reading register value is only possible if Thread<i> is in some freeze State
170 174 178 17c	CR_THREAD<id>_DFD_REG_DATA	RO	Value of register `THREAD<i>_REG_ID`	[31:0]	If Thread<i> is not in some Freeze State the value is garbage
180 184 188 18c	CR_THREAD<id>_STACK_BASE_OFFSET	RW	The base stack pointer of thread i	[15:0]	
190 194 198 19c	THREAD<i>_TLS_BASE_OFFSET	RW	The base Stack pointer Offset for Thread `i`	`[15:0]`	
200 204 208 20c	CR_SCRATCHPAD<id>	RW	Scratchpad registers for software. Set to zero on reset	[31:0]	

Table 3 - CR table

3.1.1 Some key and important to mention CRs:

- CR_THREAD_ID – this CR will hold the current running thread's Id. it is used by multi thread applications.
- CR_STACK_BASE_OFFSET – this CR will hold the current running thread's stack base offset. In our validation plan, before each program is running, each thread takes this data and stores it in the sp (stack pointer) register.
- CR_THREAD<id>_PC_EN – These CRs has the ability to freeze a specific thread's PC by writing '0 to the desired offset using the software.
- CR_SCRATCHPAD<id> - used to communicate between threads.

More on those CRs and CR features in MAS document.

3.2 Non-Local Memory

3.2.1 Writing to Non-Local Memory

Local & Non-Local match are distinguish using the “core-ID” property which is encoded in the MSB of the Address - Bits [31:22].
To support Non-Local Memory Writes, the GPC_4T will need to send any write request to the fabric. The write is “posted” requests, which mean we are not waiting for a response.
Due to nature of our ring architecture we know that ordering is being preserved in the fabric.
This means we may send write requests Back 2 Back without concerning ourselves with coherency problems. Only in case the signal “C2F_stall” is high we will backpressure the Core Write requests.

3.2.2 Reading from Non-Local Memory

GPC_4T communication with other cores require the ability to extract data from different cores in the ring. GPC_4T supports that feature by being able to create reading requests from different cores by changing the destination core ID as needed so the request will leave the current GPC_4T and pass through the ring until reaching the core with the correct ID. That destination core will respond to the

sending core with the data as requested and the data initially requested from the non-local memory region can be processed in the current core.
Once a Thread sends a read request to a Non-Local Memory, the data might arrive after many cycles. During the cycles where the Thread that sent the Req is supposed to run with other instructions, no operation can happen because the data to continue the Thread program is not yet available. The Thread must wait until the response arrives. Therefore, the Thread can freeze until the requested data to continue the program arrives from the Fabric. The freezing is done by setting CR_THREAD<id>_PC_EN = ‘0’.

3.2.3 Stalling and Signaling Different Threads

One usage of GPC_4T is run programs using multiple Threads. To use multi-thread programing, the Threads need the ability to communicate, stall and signal different threads within the current core or different cores in the ring. The feature is applied using the CR registers and the ability of Thread to signal other Threads to continue after stalling (using CR_THREAD<id>_PC_EN = ‘1’ or CR_SCRATCHPAD<id>) and using Shared Space as shown in D_MEM.