GPC_4T:
MAS

GPC_4T Micro-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 MAS*	*06 November 2021*
0.3	*Adi Levy*	*fixed and detailed flows*	*25 December 2021*

Figures

Tables

1 General Description

1.1 High Level Definition

The MAS describes the Micro-Level-Architecture of the **GPC_4T.**
This is the Logic Design implementation of GPC_4T Architectonic demands described on HAS.
This Section covers the micro architecture of the stages of the core, and the way the D_MEM_WRAP and I_MEM_WRAP interact with the core and the Fabric. The section will cover the logic design for the communication and the implementation of the GPC_4T.

1.2 Top-Level description

The GPC_4T module is one of two main components in the LOTR Tile (the other is the Ring Controller). The module contains 3 major building blocks : **Core**(Core_4t) , **Instruction memory wrapper**(I_MEM_WRAP) and **Data memory wrapper**(D_MEM_WRAP). The inputs and outputs coming directly from 2 **Ring controller buffers** : Core to Fabric buffer (AKA **C2F**) which handle requests coming from the core to the Fabric (For example, read request from non-local memory) and Fabric to Core buffer (AKA **F2C**) which handle request coming from the Fabric to the core.

The core module doesn't "aware" of the Ring Controllers and all the requests and responses going from the buffers to the I Mem and D Mem Wraps or vice versa. All the communication between Core and RC (Ring Controller) is done via the memory wrappers.

1.3 GPC_4T Block Diagram

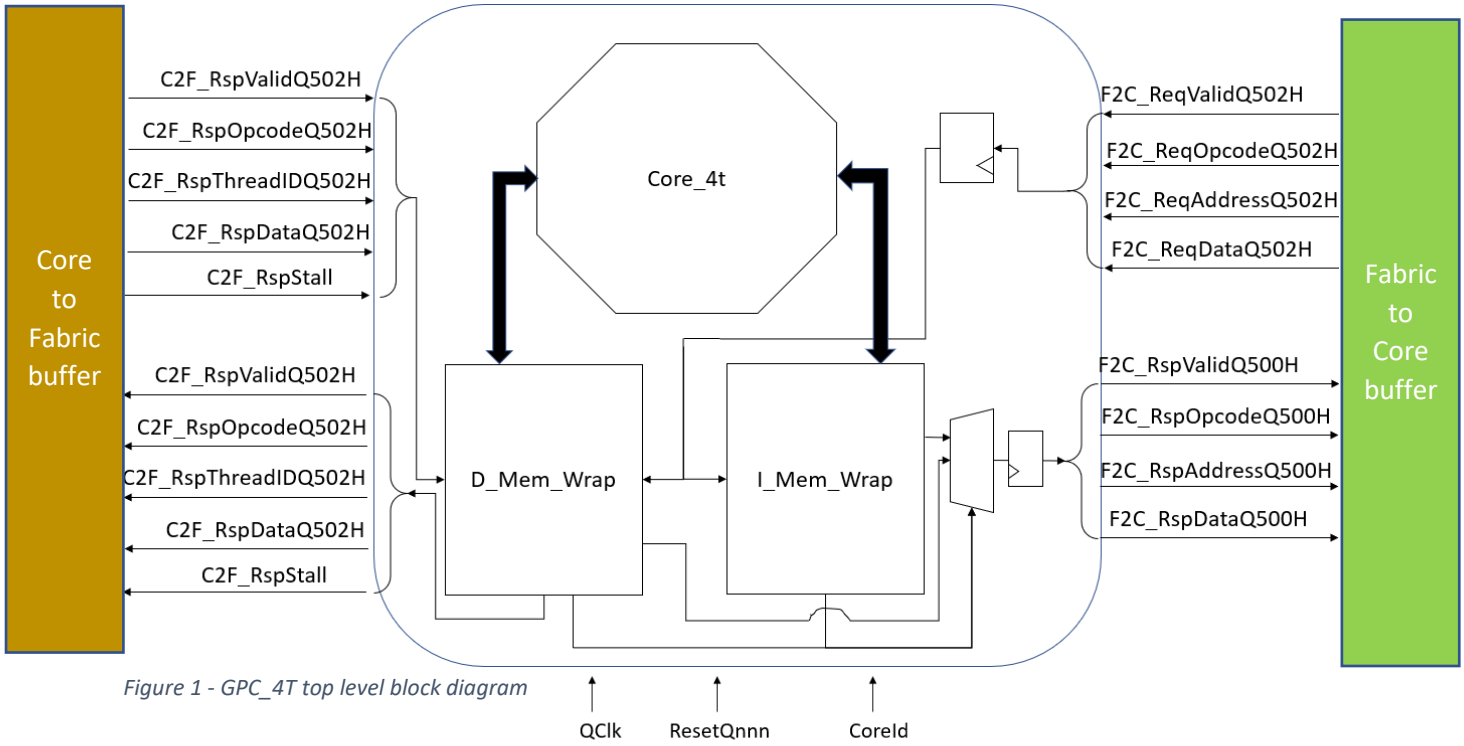


Figure 1 - GPC_4T top level block diagram

1.4 TOP LEVEL & BLOCKS INTERFACE

Default Parameter Values:

Name	Default	Description
INT_LEN	32	Integer Size - RV32I Spec "XLEN"
REGFILE_NUM	32X4	Number of registers in the register file - RV32I Spec is 32 for each thread
SIZE_I_MEM	2^12	Size of the instruction memory
LSB_I_MEM	0	Offset of the LSB in the I MEM
MSB_I_MEM	11	Offset of the MSB in the I MEM
I_MEM_OFFSET	0x00000000	Offset of the I mem
SIZE_D_MEM	2^12	Size of the date memory
LSB_D_MEM	0	Offset of the LSB in the D MEM
MSB_D_MEM	11	Offset of the MSB in the D MEM
D_MEM_OFFSET	0x00400000	Offset of the D mem
SIZE_SHRD_MEM	2^11	Size of shared memory
SIZE_MEM	2^13	Whole memory size
MSB_CR	11	Offset of the MSB in the CR MEM
LSB_REGION	22	LSB of the region bits on the address
MSB_REGION	23	MSB of the region bits on the address
I_MEM_REGION	00	I mem region bits
D_MEM_REGION	01	D mem region bits
CR_REGION	11	CR region bits
LSB_CORE_ID	24	LSB of the Core id bits on the address
MSB_CORE_ID	31	MSB of the Core id bits on the address
THREAD0_STACK_BASE	0x400200	Thread 0 stack pointer base offset
THREAD1_STACK_BASE	0x400400	Thread 1 stack pointer base offset
THREAD2_STACK_BASE	0x400600	Thread 2 stack pointer base offset
THREAD3_STACK_BASE	0x400800	Thread 3 stack pointer base offset

Table 1 – GPC_4T Parameters

General interface signals:

Name	Size	Direction	Description
QClk	1	Input	Q Clock – a single clock domain.
RstQnnnH	1	Input	Active High Reset
CoreID	8	input	Contains a unique Core Id number from the LOTR Tile

Table 2 - GPC_4T general interface

GPC_4T To Fabric Interface:

Name	Size	Direction	Description
Fabric to Core			
F2C_ReqValidQ502H	1	Input	The data entering the GPC_4T is a Req.
F2C_ReqOpcodeQ502H	1	Input	Opcode of the Req that is passed from the Fabric: RD or WR.
F2C_ReqAddressQ502H	32	Input	The address of the Req in the destination core.
F2C_ReqDataQ502H	32	Input	Data to be written in the destination core (when WR).
F2C_RspValidQ500H	1	output	The data leaving to the Fabric is a Rsp.
F2C_RspOpcodeQ500H	1	output	Opcode of the Rsp that is passed to the Fabric: RD or WR.
F2C_RspAddressQ500H	32	output	The address of the Rsp in the destination core.
F2C_RspDataQ500H	32	output	Data to be written in the destination core (when WR).
Core to Fabric			
C2F_RspValidQ502H	1	input	The data entering from the Fabric is a Rsp.
C2F_RspOpcodeQ502H	1	Input	Opcode of the entering Rsp that is entering from the Fabric: RD or WR.
C2F_RspThreadIDQ502H	2	Input	The destination thread of the entering Rsp.
C2F_RspDataQ502H	32	Input	Data that was requested by the original Req.
C2F_Stall	1	Input	When '1', Core does not send out Reqs
C2F_ReqValidQ500H	1	output	The data leaving to the Fabric is a Req.
C2F_ReqOpcodeQ500H	1	output	Opcode of the Req that is passed to the Fabric: RD or WR.
C2F_ReqThreadIDQ500H	2	output	The thread sending the Req.
C2F_ReqAddressQ500H	32	output	The address of the Req in the destination core.
C2F_ReqDataQ500H	32	output	Data to be written in the destination core (when WR).

Table 3 - GPC_4T top interface

2 DESIGN BUILDING BLOCKS

2.1 CORE & PIPE STAGES

The core (AKA Core_4t) is one of the main building blocks of GPC_4T. The cores task is to run and execute the program instructions. The gpc_4t core supports 4 hardware threads simultaneously, each has unique Pc (Program counter), unique register files (each with 32 registers capable holding 32bit integer value) and unique data memory region for its stack pointer. The core is in order pipeline architecture with 5 stages. The MAS covers the core stages along with the logic and interface with the other components of the GPC_4T. A better description of the core will be to describe each pipeline stage separately.

Core Interface:

Name	Size	Direction	Description
Instruction Output			
PcQ100H	32	output	PC that leaves to the I_MEM from Fetch stage
MemAdrsQ103H	32	output	Memory address that leaves to D_MEM from Memory Access stage
MemWrDataQ103H	32	output	Data that leaves to D_MEM from Memory Access stage
CtrlMemWrQ103H	1	output	WR indication bite that leaves to D_MEM from Memory Access stage
CtrlMemRdQ103H	1	output	RD indication bite that leaves to D_MEM from Memory Access stage
MemByteEnQ103H	4	output	Memory access indication according to Funct3 of the Opcode
ThreadQ103H	4	output	Thread sending the request
PcQ103H	32	output	Go to D Mem Wrap as CR Mem input
Instruction Input			
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset signal to stop Thread cycle
InstFetchQ101H	32	Input	Instruction received from I_MEM at Decode stage
MemRdDataQ104H	32	Input	Data requested at Memory Access stage received from D_MEM at Write Back stage
C2F_RspMatchQ104H	1	Input	Data requested at Memory Access stage from different GPC_4T memory received from D_MEM at Write Back stage.
T[i]RcAccess	1	Input	Comes from D MEM WRAP to freeze thread i PC if accessed non-local memory

Table 4 - Core interface

2.1.1 Detailed Core Block Diagram

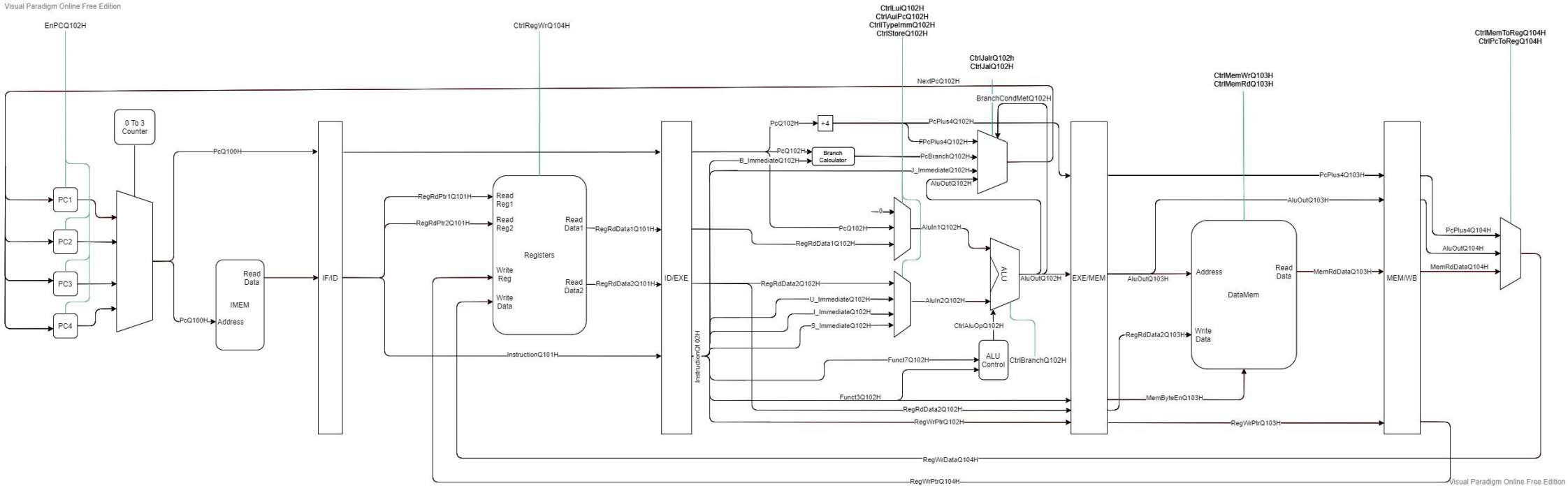


Figure 2 - Core Block diagram

2.1.2 Thread Switch and Scheduler

Before the pipe stages, or more abstract, above all the pipeline, is the logic that determine the threads scheduler. Each thread is decoded as a 4 bits number – 0001 , 0010 , 0100 , 1000 for thread 0, 1, 2, 3 respectively.

Five signals, one for each pipe stage, holding the thread number currently in this stage. For example, ThreadQ100H will hold 1000 means that thread 3 currently in stage Q100H which means that ThreadQ101H will hold 0100 and ThreadQ102H will hold 0010 ,etc.

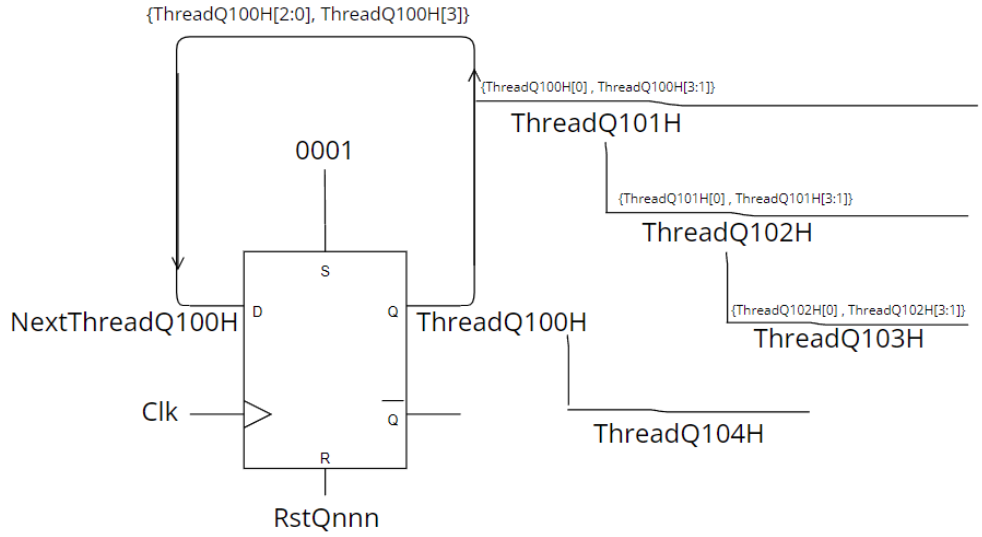


Figure 3 - scheduler design

The scheduler is implemented with a **shift register** flop with a "reset value" = 0001, input NextThreadQ100H and output ThreadQ100H.

At first ThreadQ100H will hold reset value 0001. with the implementation logic, NextThreadQ100H will get the 3 LSB of ThreadQ100H as its MSB and will hold the value "0010" which is the next thread that will run on stage Q100H in the next cycle.

With some bitwise shifting, ThreadQ101H, ThreadQ102H and ThreadQ103H will get the values 1000 , 0100 ,0010.

On the next cycle, ThreadQ100H will get the value 0010 and NextThreadQ100H with the shift register will get 0100 and so on.

```
logic [3:0] RstVal = 4'b0001;
assign NextThreadQ100H = {ThreadQ100H[2:0], ThreadQ100H[3]};
`LOTR_RST_VAL_MSFF(ThreadQ100H, NextThreadQ100H, QClk, RstQnnnH, RstVal)
assign ThreadQ101H = {ThreadQ100H[0] , ThreadQ100H[3:1]} ;
assign ThreadQ102H = {ThreadQ101H[0] , ThreadQ101H[3:1]} ;
assign ThreadQ103H = {ThreadQ102H[0] , ThreadQ102H[3:1]} ;
assign ThreadQ104H = ThreadQ100H;
```

Figure 4 - Schedulr code

	Q100H	Q101H	Q102H	Q103H	Q104H
cycle 0	0001				
cycle 1	0010	0001			
cycle 2	0100	0010	0001		
cycle 3	1000	0100	0010	0001	
cycle 4	0001	1000	0100	0010	0001
cycle 5	0010	0001	1000	0100	0010
cycle 6	0100	0010	0001	1000	0100

Table 5 - Thread switch signals example

2.1.3 Pipe Stages

2.1.3.1 Fetch

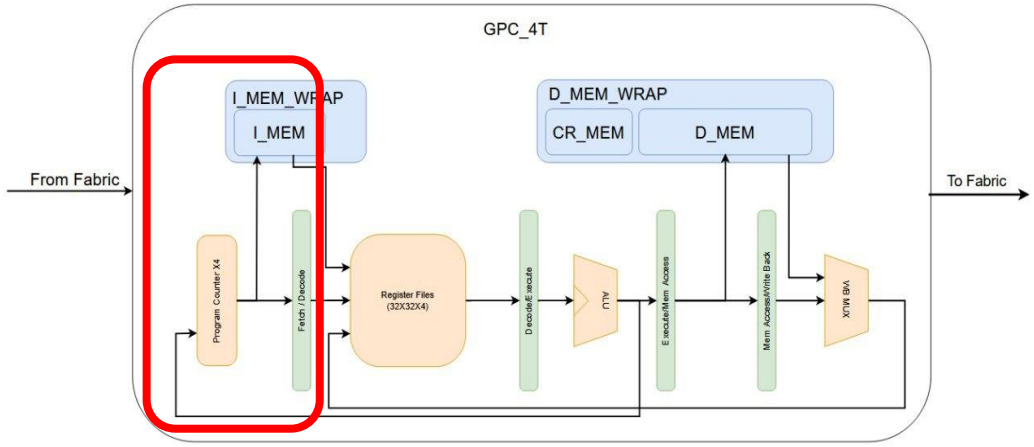


Figure 5 - Fetch on high level

Fetch stage first "job" is to maintain 4 separate program counters, one for each thread.

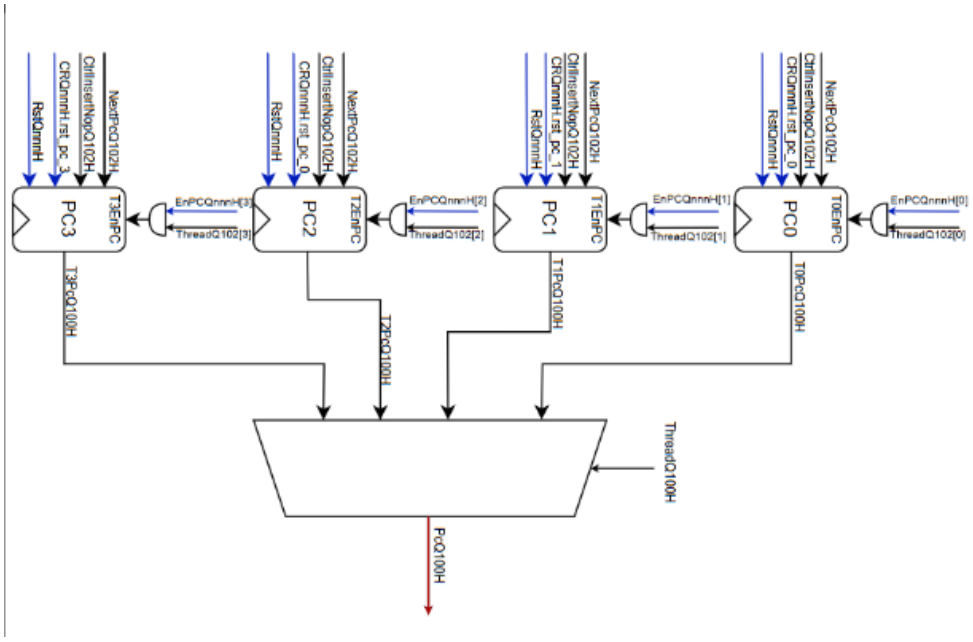


Figure 6 - The program counters

Each PC is fed by a signal "NextPcQ102H" will implies that the **next Pc for a thread is determined on stage Q102H.**
The "enable" for each PC is toggled by this logic:

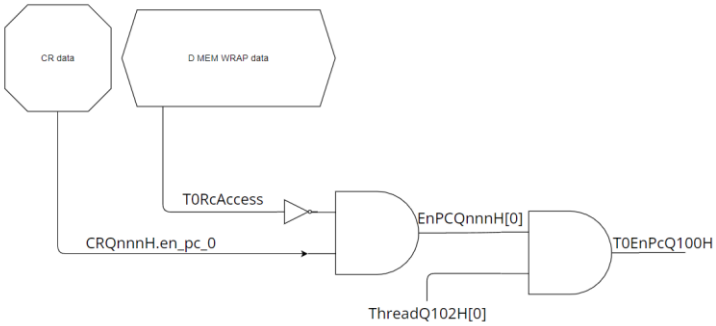


Figure 7 - Pc enable logic

Each Pc is sampling only if ThreadQ102H[i] is `1, impling that thread currnetly on Q102H is this thread – therfore the thread's PC will hold the same value for 3 cycles, until it reaches Q102H and the enable will toggle on and it will sample new next value.
all PCs are connected to MUX that will control the signal PcQ100H. The output of the MUX will be determine by the thread currently runs on stage Q100H.

The second importand role for this stage is to take the PcQ100H signal to the I Mem Wrap and to the instruction memory as the address of the instruction in the I_MEM.

in addition, Nop injection logic is defined in this stage:

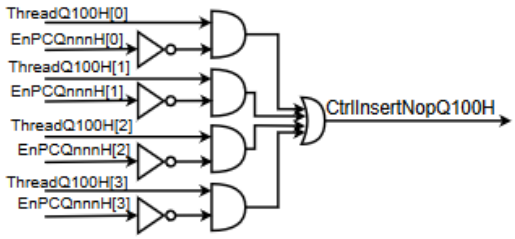


Figure 8 - Nop toggle logic

if current thread running on this stage is also disabled by CR Mem(software freeze) or D Mem wrap (hardware freeze by accessing to non-local memory), Nop control signal will toggled and will bubble until stage Q101H there the Decode stage will inject a "nop" instruction instead of a regular instruction. The signal will continue to Q102H and will connect to the PC as another enable signal.

2.1.3.2 Decode

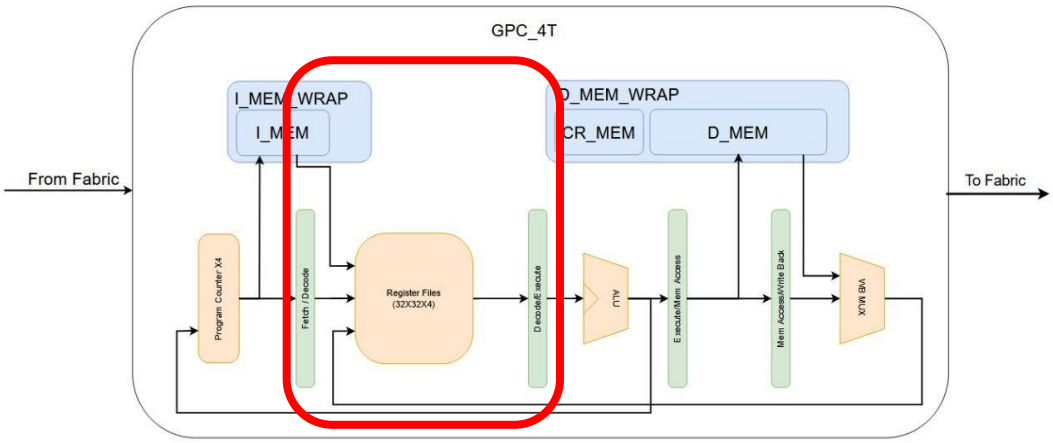


Figure 9 - Decode on high level

Decode stage is the pipe stage on which the 32bit encoded instruction arrived from the instruction memory is decoded. All relevant data from the instruction including registers numbers and OpCode control signals is extracts from the instruction fetched. For example, if 7 LSBits of the instruction, which are the OpCode, are 0000011, then it means this is load instruction and the signal **CtrlMemRdQ101H** is set to `1.

the stage logic design:

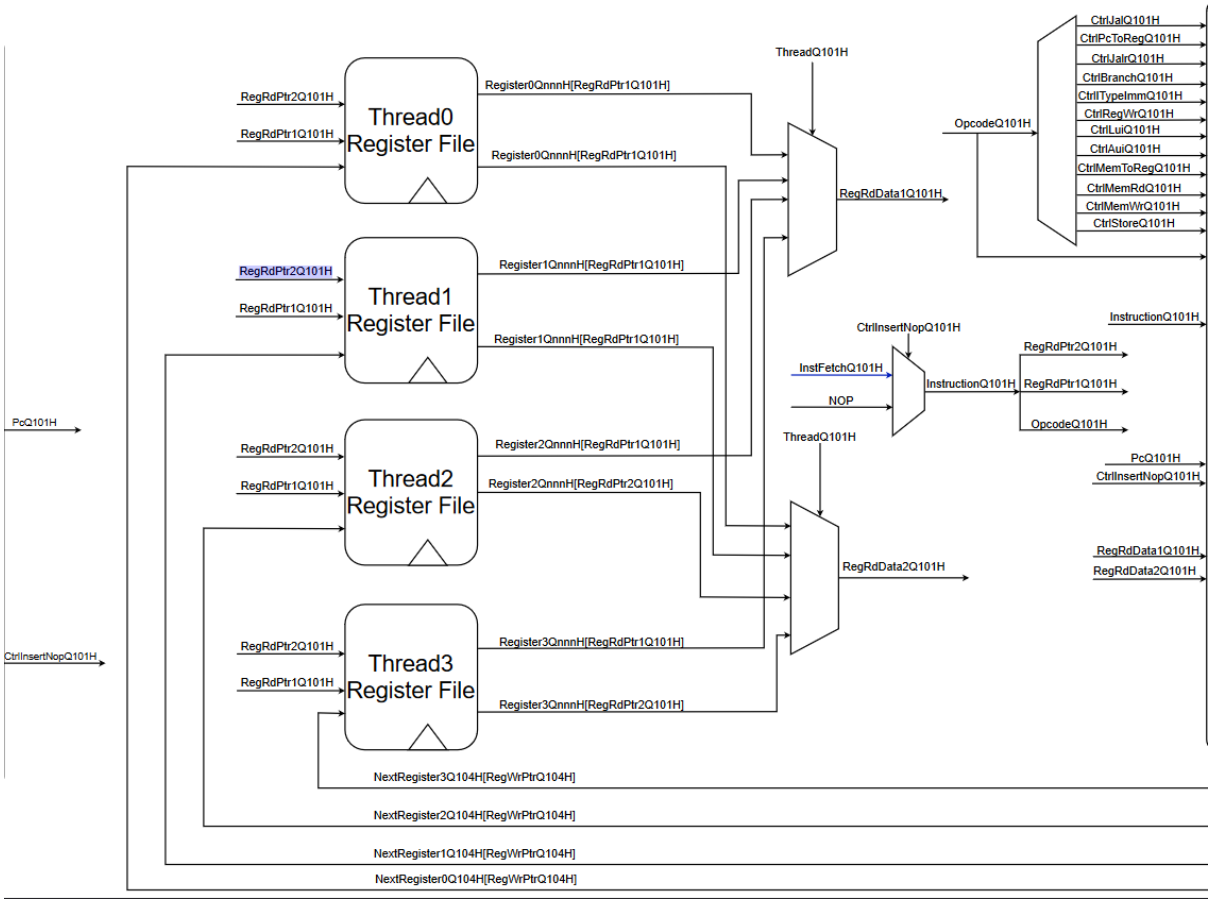


Figure 10 - Decode logic design

4 different register files for each thread, each with 32 registers that can save 32bit integer.
Data from designated registers(as in the instruction) is read simultaneously from all threads designated registers, and 2 Mux units determine which the output by the thread currently runs on Q101H.
output determined by the Mux units is the registers data that will be transferred to the next pipe stage as RegRdData1Q101H and RegRdData2Q101H.
The control signals are toggled by the data extracted from the instruction, with the OPCODE data and funct3 and funct7 data decode.
All the control signals are sampled from SignalQ101H to SignalQ102H to the next pipe stage.

2.1.3.3 Execute

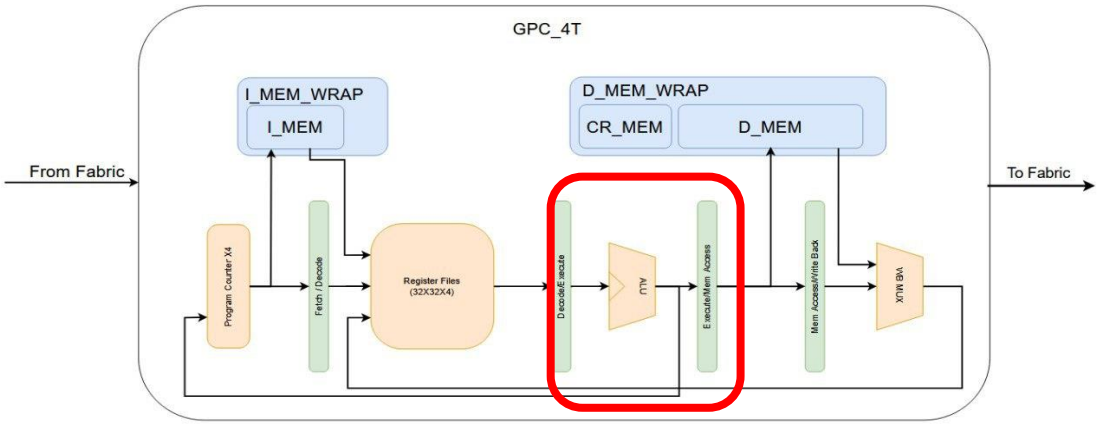


Figure 11 - Execute on high level

The Execution stage contains all the components that calculate values. In this stage all the arithmetic calculation are initiate, branch resolution is made and the next PC for current thread is determined. in addition the instruction data (now at InstructionQ102H) is passing another decode and broken down into the Immediates, funct3, and funct7.

The stage logic design:

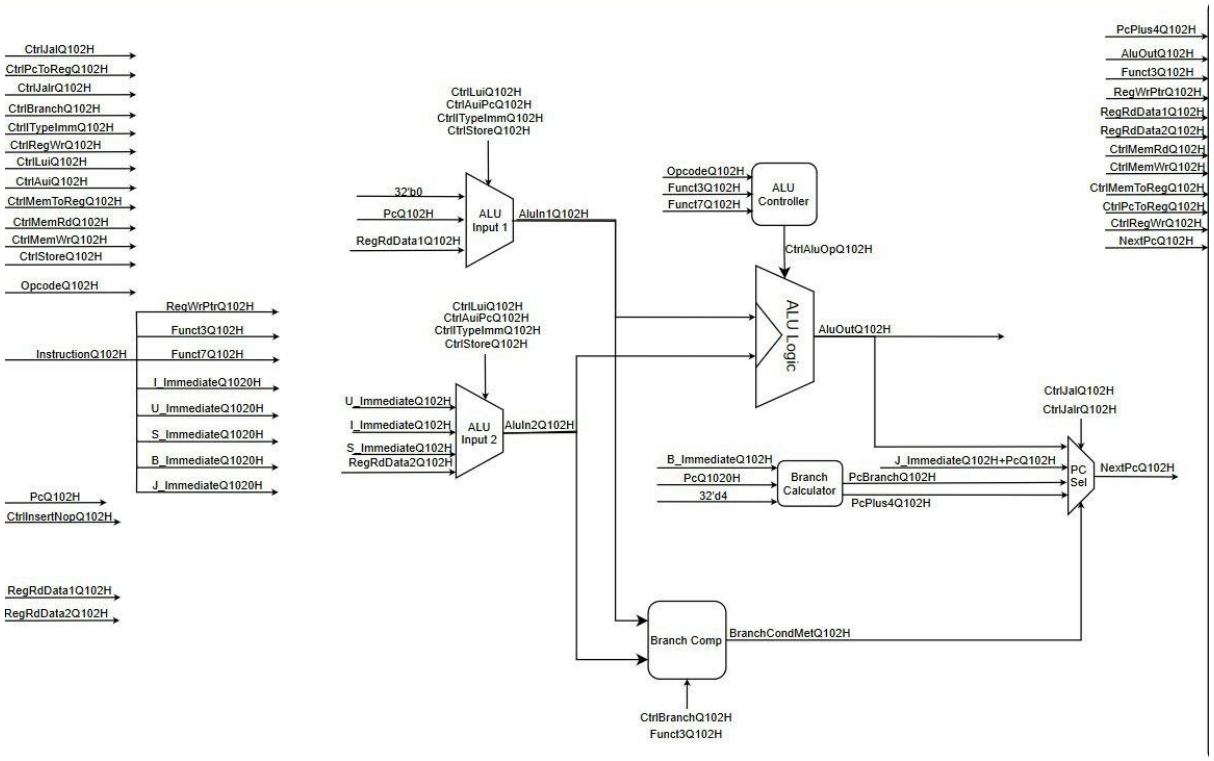


Figure 12 - execute logic design

Branch Calculator: calculate the 2 possible next PCs of the thread: PcPlus4Q102H=PcQ1020H+ 32'd4 and PcBranchQ102H= B_ImmediateQ102H+PcQ1020H. The new value for the next PC is chosen in the PS select component.

ALU Input 1 and ALU input 2 MUXs: selecting the correct inputs to the ALU using CtrlLuiQ102H, CtrlAuiPcQ102H, CtrlITypeImmQ102H and CtrlStoreQ102H as the select for both MUXs. ALU Input 1 selects between RegRdData1Q102H, PcQ102H and 32'b0 and ALU Input 2 selects between the Immediates (U,I,S) and RegRdData2Q102H. the outputs of the MUXs are AluIn1Q102H and AluIn2Q102H.

Branch Comparator: when a branch instruction is inserted, the inputs of the branch comparator (AluIn1Q102H and AluIn2Q102H) are the values to compare. Based on funct3 to decide which condition is to be set, BranchCondMetQ102H gets the correct value.

ALU Controller: based on funct3, funct7 and the opcode, will set a value to CtrlAluOpQ102H that will enter the ALU and decide the operation.

ALU Logic: the actual computational component. Will get AluIn1Q102H and AluIn2Q102H as inputs and according to CtrlAluOpQ102H will calculate the value for AluOutQ102H.

PC Sel: decides the next value for PC among the 4 options: PcBranchQ102H calculated by the Branch Calculator, PcPlus4Q102H also calculated by the Branch Calculator, AluOutQ102H calculated in the ALU and J_ImmediateQ102H+PcQ102H that is the address to jump if the instruction is a Non-

Conditional Branch. Using CtrlJalrQ102H, CtrlJalQ102H
and BranchCondMetQ102H as the select the chosen PC (NextPcQ102H) is passed to the four PCs in the Instruction Fetch stage and is updated using the indicated from Q102H (the Execute stage).

2.1.3.4 Memory

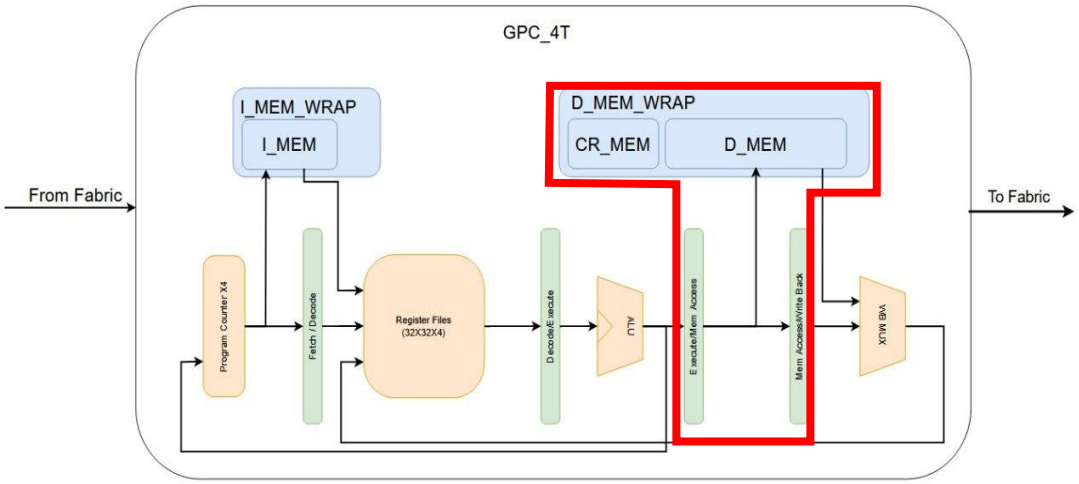


Figure 13 - Mem stage on high level

The memory stage doesn't contain much logic in the core. the main idea in this stage is the data memory transaction. The data memory is accessed to read from or write to. A **writable data** from the registers is assign on the signal "MemWrDataQ103H" and an **address** to read from in the memory is assign to signal "MemAdrsQ103H", both as the outputs to the data memory, and control signals implying read enable or write enable (or neither) is also assign to data mem. PcQ103H signal is also leave the core to the data memory wrapper in order to implement certain logic. The MUX output MemByteEnQ103H is selected by Funct3Q103H[1:0] from 4 possible inputs: 'b0001 (LB/SB), 'b0011(LH/SH), 'b1111(LW/SW) or 'b0000.

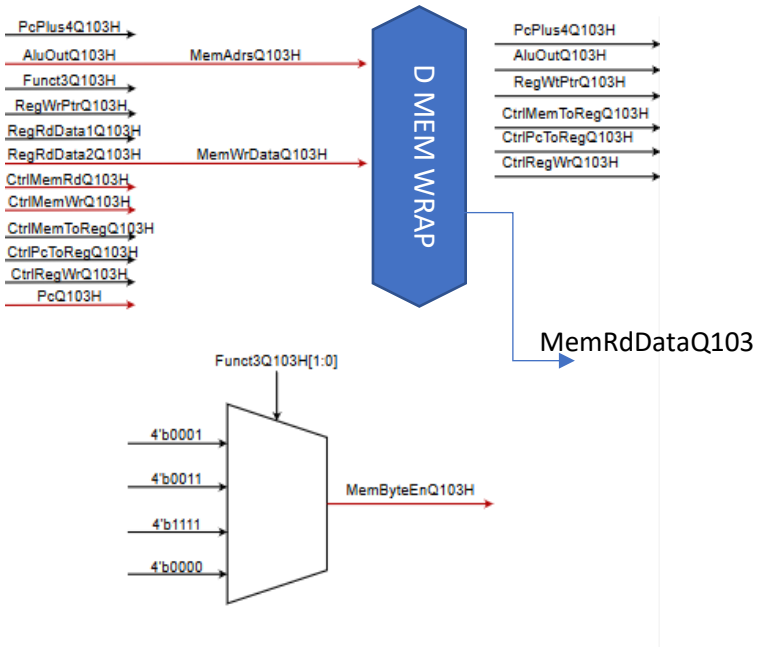


Figure 14 - Mem access logic design

2.1.3.5 Write Back

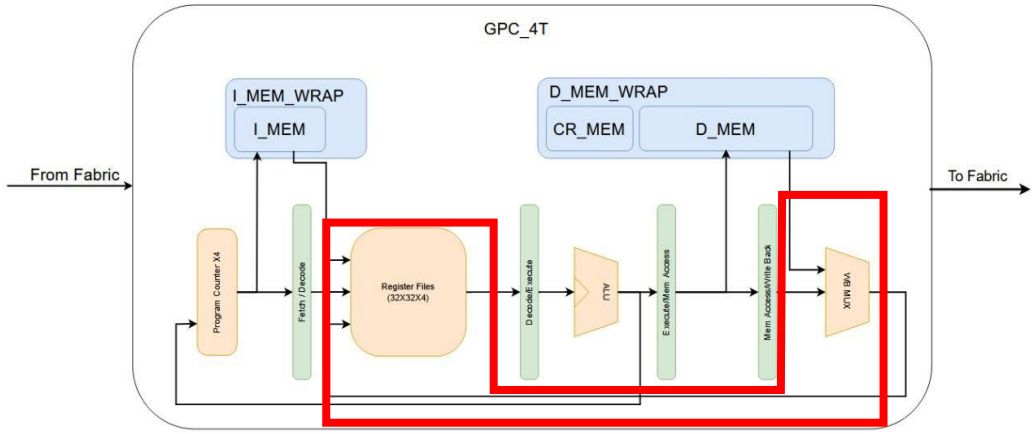


Figure 15 - Write-back on high level

The Write Back stage selects the correct data to be passed and written at the thread register file. Once the instruction was executed, the new might be written to the register file. The candidates for the data are PcPlus4Q104H (PC+4 for jump and link command), MemRdDataQ104H (data read from memory) or AluOutQ104H(ALU calculation result) . Using CtrlMemToRegQ104H, CtrlPcToRegQ104H and C2F_RspMatchQ104H the correct input is selected (RegWrDataQ104H) and using ThreadQ104H, RegWrDataQ104H is directed to the threads register file. The register files default behavior is to keep the old value and not update, unless CtrlRegWrQ104H is set to '1' and the register file will read the data in RegWrDataQ104H and update.

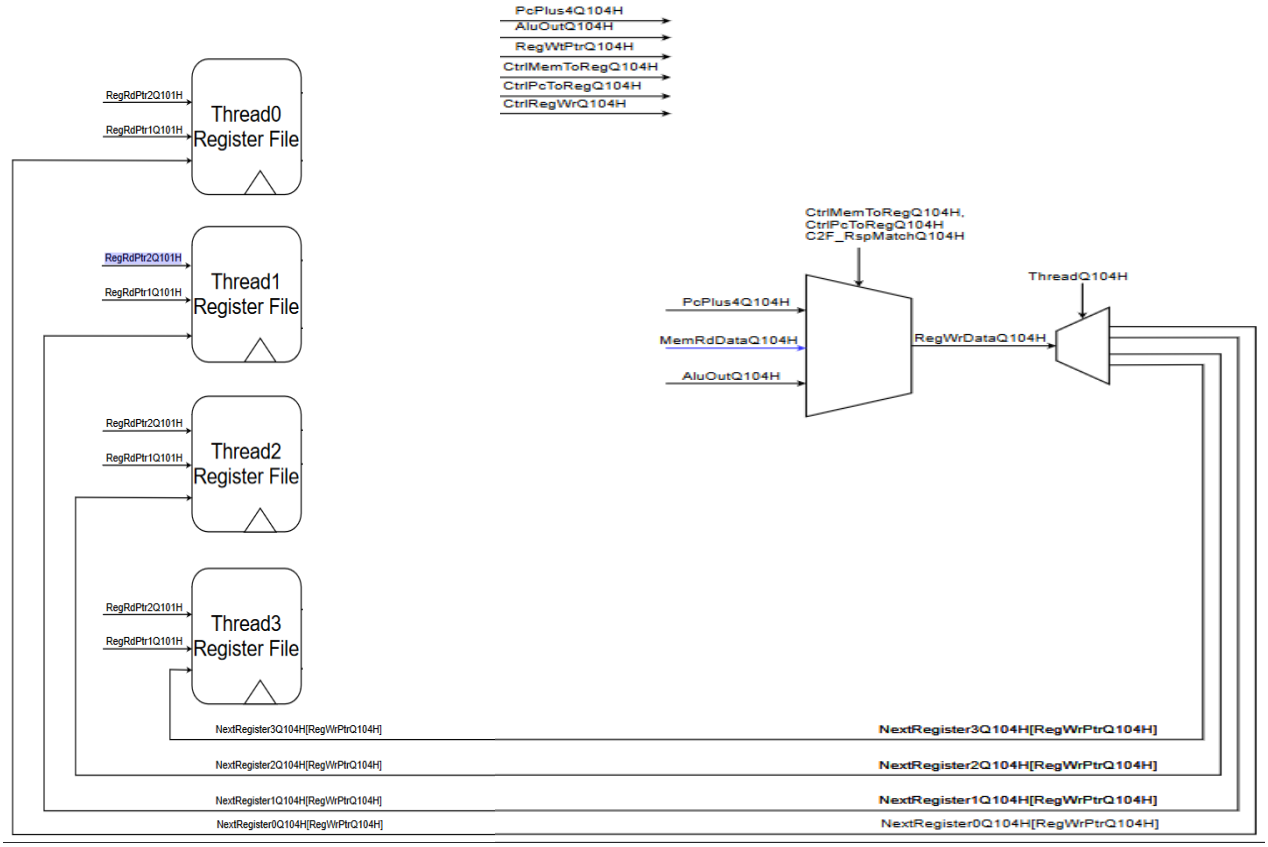


Figure 16 - WB logic design

2.2 I_MEM

The I_MEM is the memory module that contains the code of the program. The module is S-RAM based and can contain up to 4KB of instructions by default and defined in the addresses 0x00000000 – 0x00000FFF.

The I Mem is dual port – it contains separate inputs and outputs for both GPC_4t Core and the Ring Controller. In our design the core can only read instructions and the Rc can only write instructions. Its important to mention that in this design, we implemented the memory itself as a behavioral memory, which simulates the behavior of a physical memory. We used an array on the Verilog code and in the future synthesis plans, it will be replaced with FPGA supported design.

2 flops, one for each interface are sampling the memory instruction data at the input address for each interface (core with PC as address). The output of the Core interface flop is the instruction fetched. All the behavioral memory array also sampled in a flop.

Name	Size	Direction	Description
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset signal
Core Interface			
address_a	32	input	usually, the Pc of a thread from the core used as the address
data_a	32	input	data to be written by the core,usually `0. not in use
rden_a	1	input	read enable bit from the core, usually `1
wren_a	1	input	write enable from the core, usually `0 .not in use
q_a	32	output	data(instruction) sampled from i mem array to the core
RC interface			
address_b	32	input	address from Rc
data_b	32	input	data to be written by the Rc
rden_b	1	input	read enable bit from the Rc, usually `0. not in use
wren_b	1	input	write enable from the Rc
q_b	32	output	data(instruction) sampled from i mem array to the Rc

Table 6 - I mem interface

```
logic [7:0]  mem      [SIZE_I_MEM-1:0];
logic [7:0]  next_mem[SIZE_I_MEM-1:0];

always_comb begin : write_to_memory
    next_mem = mem;
    if(wren_a) begin
        next_mem[address_a+0]= data_a[7:0];
        next_mem[address_a+1]= data_a[15:8];
        next_mem[address_a+2]= data_a[23:16];
        next_mem[address_a+3]= data_a[31:24];
    end
    if(wren_b) begin
        next_mem[address_b+0]= data_b[7:0];
        next_mem[address_b+1]= data_b[15:8];
        next_mem[address_b+2]= data_b[23:16];
        next_mem[address_b+3]= data_b[31:24];
    end
end

// the memory flipflops
`LOTR_MSFF(mem, next_mem, clock)
```

Figure 17 - behavioral memorry array code

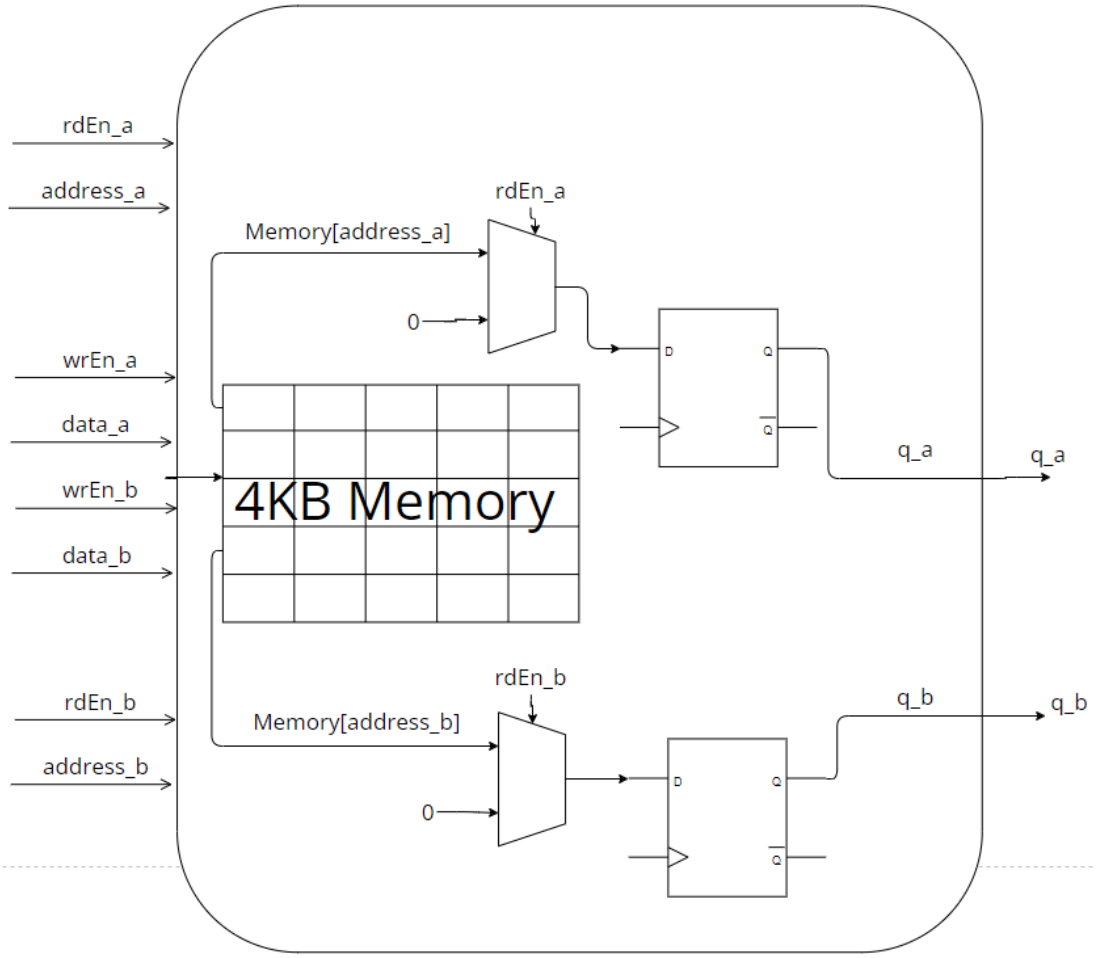


Figure 18 - I mem logic design

2.3 I_MEM _WRAP

The I mem wrap is the module wrapping the I mem module and is integrated to the core in the GPC_4T design. Using this module, the core can "talk" with the real instruction memory and this is the unit that connects with the core. This module simply instantiate the I Mem module, and also implements Fabric To Core logic for Ring controller interaction.

I_MEM_WRAP Interface:

Name	Size	Direction	Description
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset signal
Core interface			
PcQ100H	32	Input	PC from the core of the requested instruction
RdEnableQ100H	1	Input	RD enable bit for the I_MEM RD validation
InstFetchQ101H	32	output	Requested instruction
Ring interface			
F2C_RspIMemValidQ504H	1	output	Validation bit indicating a valid Rsp to a Fabric Req
F2C_I_MemRspDataQ504H	32	output	Rsp to a Fabric Req
F2C_ReqValidQ503H	1	Input	Validation for a Req from the Fabric
F2C_ReqOpcodeQ503H	1	Input	Opcode of a Req from the Fabric
F2C_ReqAddressQ503H	32	Input	Address of a Req from the Fabric
F2C_ReqDataQ503H	32	Input	Data of a Req from the Fabric

Table 7 - I Mem Wrap interface

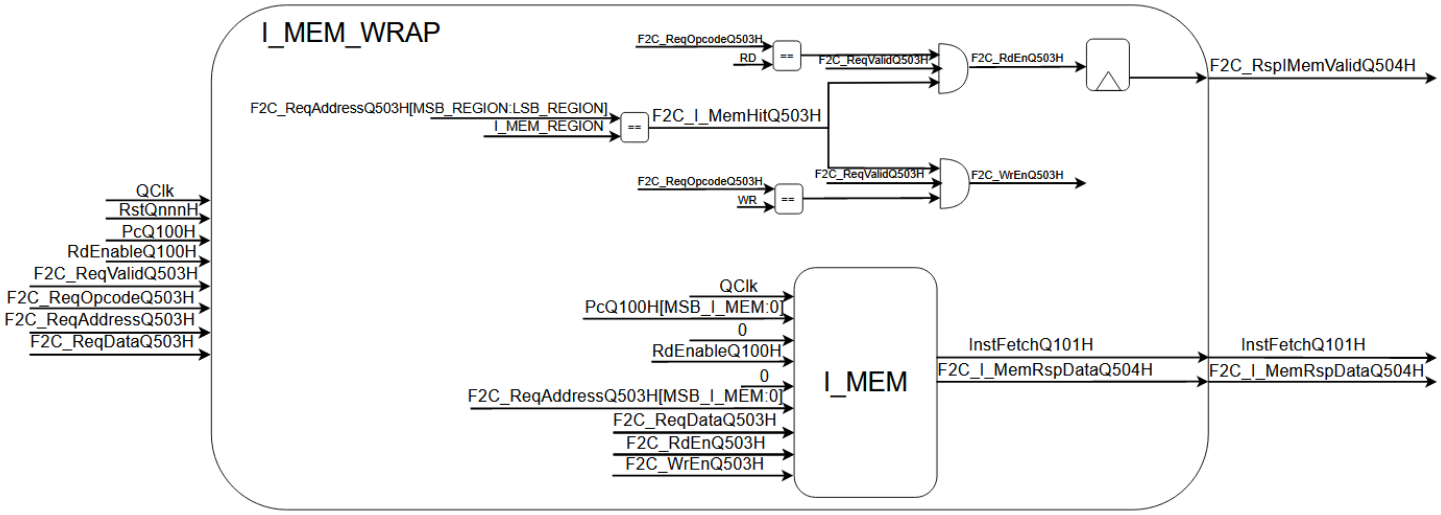


Figure 19 - I MEM WRAP design

Fabric to Core request flow:

Fabric Req: any communication between the I_MEM and the Fabric is done through using the I_MEM Fabric Interface logic. The logic checks if the Req is I_MEM Req by comparing F2C_ReqAddressQ503H[MSB_REGION:LSB_REGION] to I_MEM_REGION, checking the opcode F2C_ReqOpcodeQ503H (RD or WR) and checking for the validation bit of the Req F2C_ReqValidQ503H. once the Req is valid and the type is chosen, I_MEM receive F2C_ReqAddressQ503H[MSB_I_MEM:0], F2C_RdEnQ503H, F2C_WrEnQ503H (and F2C_ReqDataQ503H if WR). If the Req is WR, the data is written to the relevant address, else the Req is RD and a Rsp leaves I_MEM to the Fabric as a Rsp via F2C_I_MemRspDataQ504H and a valid bit for the Rsp F2C_RspIMemValidQ504H.

Core request flow:

Core Req: The request from the core is the typical instruction request. PcQ100H[MSB_I_MEM:0] is sent to I_MEM along with RdEnableQ100H from the Fetch Stage. The instruction of the thread leaves I_MEM as InstFetchQ101H to enter the core at the Decode Stage.

2.4 D_MEM

D_MEM is the memory module that contains the data of the program. The module is S-RAM based and can contain up to 4KB of data by default and defined in the addresses 0x00400000 – 0x00400FFF and its very similar to the I MEM.
The D Mem is dual port – it contains separate inputs and outputs for both GPC_4t Core and the Ring Controller. In our design the core can read and write data to the D Mem and the RC can do the same. Like on the I MEM, it's important to mention that in this design, we implemented the memory itself as a behavioral memory, which simulates the behavior of a physical memory. We used an array on the Verilog code and in the future synthesis plans, it will be replaced with FPGA supported design. 2 flops, one for each interface are sampling the memory instruction data at the input address for each interface (core with PC as address). The output of the Core interface flop is the instruction fetched. All the behavioral memory array also sampled in a flop.

On our Validation plan, before each program initiates, each thread determines its Stack Base Offset (From CR). From this offset each thread received 512B of data memory. Using this stack base, the D Mem is divided into 5 memory regions those default offsets:

- 0x00400200 - 0x00400000 Thread0 Region. 512B
- 0x00400400 - 0x00400200 Thread1 Region. 512B
- 0x00400600 - 0x00400400 Thread2 Region. 512B
- 0x00400800 - 0x00400600 Thread3 Region. 512B
- 0x00400FFF - 0x00400800 Shared Mem Region. 2KB

Each thread starts filling his stack from its offset to bottom. All thread can use the Shared Mem region in order to communicate each other. Those values are defaults and can be changed.

Name	Size	Direction	Description
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset signal
Core Interface			
address_a	32	input	address coming from core
data_a	32	input	data to be written by the core
rden_a	1	input	read enable bit from the core
wren_a	1	input	write enable from the core
byteena_a	4	input	used to distinct between LW/LH/LB and SW/SH/SB commands
q_a	32	output	data sampled from i mem array to the core as read data
RC interface			
address_b	32	input	address from Rc
data_b	32	input	data to be written by the Rc
rden_b	1	input	read enable bit from the Rc
wren_b	1	input	write enable from the Rc
byteena_b	4	input	used to distinct between LW/LH/LB and SW/SH/SB commands
q_b	32	output	data sampled from i mem array to the Rc as read data

Table 8 - D MEM interface

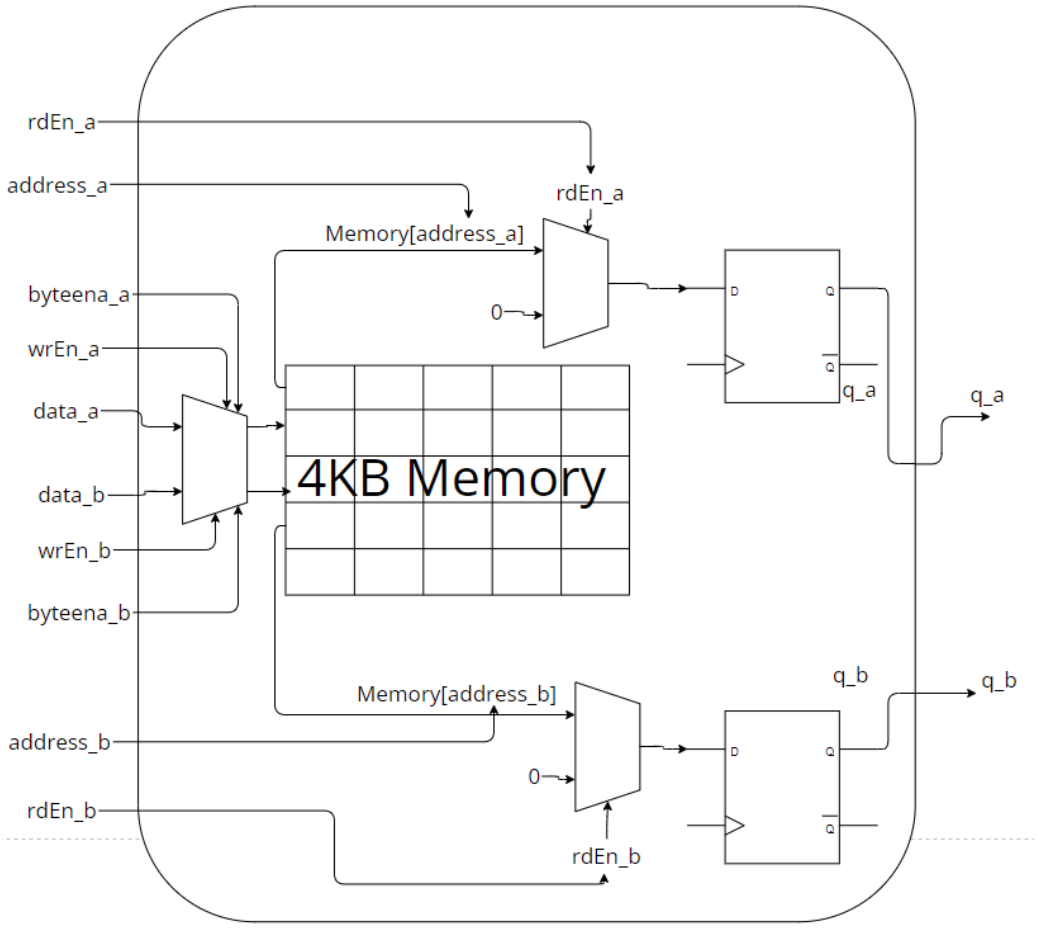


Figure 20 - D Mem logic design

2.5 CR_MEM

The CR (Control registers) Mem is a flip-flop-based memory module (unlike I mem and D mem which are S-RAM based). CR main purpose is to be the "controls" memory region. The module contains ports from the core and from the RC and contains 2 main sets of flops: Read Only and Read & Write flops. A logical "struct" contains a set of 1bit signals called Cr_En. Each of these signals is an "enable" signal for the many flops in the flops sets. For example Cr.En .pc_0 is the enable signal for the flop holds the Pc of thread0. When these signals turns to `0, the Pc0 flop holds the previous value.
For the enable signals going to R&W flops, they get their value depends on the program running (if software accessed to "reset_pc_1" offset, the " Cr.En .pc_1" will set to `1). For those who connects to the R.O flops, they get their value from hardware.

The Read Only flops contains 22 flops that samples important system data (thread i's PC, current running thread number, current core, i mem and d mem MSB and many more). these R.O flops get their value from the hardware (for example the flops holding the current PC of a thread fed by input signal PcQ103H and the enable bit of the flop shuts down if the thread is in Freeze). **These flops cannot get data from the software** by no chance. If program ask to write data to R.O CR offset, nothing will happen.

An important CRs to mention are those which contains the stack base offset. In our validation plan, using the program, each thread accessed those CRs at the beginning of the program and saved the value as the stack base offset.

The Read & Write flops contains 24 flops that samples data coming from the software. 16 of these flops holds readable system data like the R.O flops but it can also changed by the software (for example change the stack base offset of a thread). The 8 other flops are also called "Core_CR". These flops can get the values `0 or `1 by the write data from program and those values going directly to the core and affects the behavior of it. For example if " core_cr.rst_pc_0" signal will set to `1, then by going to the core it will cause the Pc of thread0 to reset and go back to 0 value.

It's important to mention that the **Ring gets priority on the Core** if both wants to write data to CR.

CR mem interface:

ame	Size	Direction	Description
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset signal
CoreIdStrap	8	input	Id of the local core
Core Interface			
CrAddressQ103H	32	input	CR offset address coming from core
ThreadQ103H	4	input	Thread Id of the thread currently on stage Q103H
PcQ103H	32	input	Pc of the thread currently on stage Q103H
CrWrDataQ103H	32	input	data to be written by the core to CR Read & Write Flops
CrRdEnQ103H	1	input	read enable bit from the core
CrWrEnQ103H	1	input	write enable from the core
CrRdDataQ104H	32	output	data read from the CR Flops assign to the core
core_cr	8	output	CR data reaching the core for Pc functionality
RC interface			
F2C_AddressQ503H	32	input	CR offset address coming from RC
F2C_WrDataQ503H	32	input	data to be written by the RC to CR Read & Write Flops
F2C_CrRdEnQ503H	1	input	read enable bit from the Rc
F2C_CrWrEnQ503H	1	input	write enable from the Rc
F2C_CrRspDataQ504H	32	output	data read from the CR Flops assign to the RC

Table 9 - Cr Mem interface

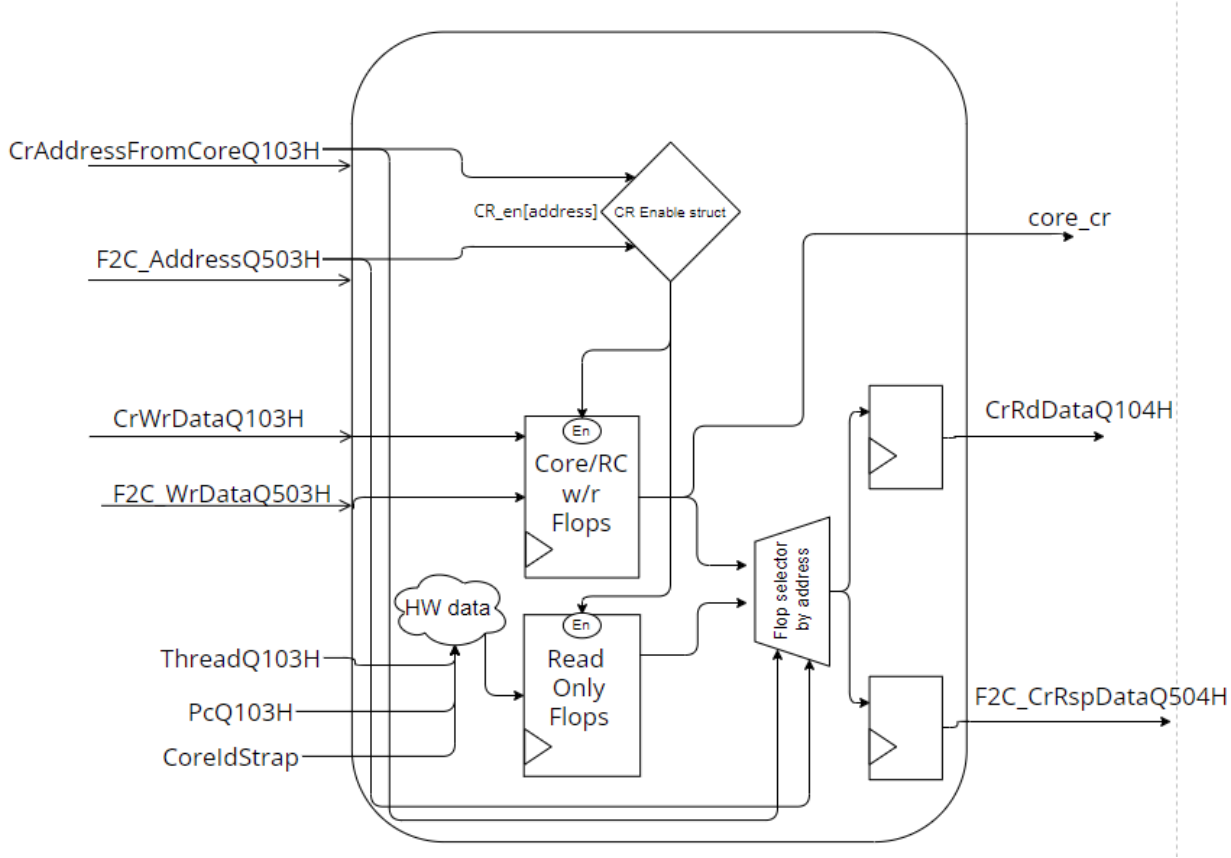


Figure 21 - Cr Mem logic design

2.6 D_MEM WRAP

The D MEM WRAP is the module wrapping and encapsulating the D Mem and the CR Mem modules. It's integrated to the core in the GPC_4T design and also to the Fabric. Using this module, the core can "talk" with the real data memory and the Control Registers memories, and this is the unit that connects with the core.

Like the I MEM WRAP, this module instantiate the D MEM and the CR MEM modules, and also implements Fabric To Core logic for Ring controller interaction.

Unlike the I MEM WRAP, the D MEM WRAP contains also a complex logic dedicate to handle Non-Local memory transactions. The core does not "know" about the ring and all the transactions are made by the D MEM Wrap.

D_MEM_WRAP Interface:

Name	Size	Direction	Description
QClk	1	input	Clock signal
RstQnnnH	1	Input	Reset for all Flip Flops
CoreIdStrap	8	Input	ID of the source core
Core Interface			
ThreadQ103H	4	Input	Number of requesting thread
PcQ103H	32	Input	PC of the request instruction
CRQnnnH	8	output	
AddressQ103H	32	Input	Request address
ByteEnQ103H	4	Input	Byte Enable according to Funct3 of the Opcode (LB/SB, LH/SH, LW/SW)
WrDataQ103H	32	Input	Data to be written
RdEnQ103H	1	Input	Identifying request as RD
WrEnQ103H	1	input	Identifying request as WR
MemRdDataQ104H	32	output	Data requested in Memory Access stage, retuned to Write Back stage
C2F_RspMatchQ104H	1	output	Signals to core about C2F response

T[i]RcAccess	1	output	signals to core if thread i accessed non local memory
Ring interface			
F2C_ReqValidQ503H	1	Input	Identifying valid Fabric Req
F2C_RspDMemValidQ504H	1	output	Identify as Rsp to the current GPC_4T
F2C_D_MemRspDataQ504H	32	output	Rsp data to the current GPC_4T
F2C_ReqOpcodeQ503H	1	Input	Fabric Req opcode
F2C_ReqAddressQ503H	32	Input	Fabric Req address
F2C_ReqDataQ503H	32	Input	Fabric Req data
C2F_RspValidQ502H	1	Input	Rsp to fabric validation bit
C2F_RspOpcodeQ502H	1	Input	Rsp to fabric opcode
C2F_RspThreadIDQ502H	2	Input	Destination thread
C2F_RspDataQ502H	32	Input	Rsp to fabric data
C2F_RspStall	1	Input	Currently NOT Implemented
C2F_ReqValidQ500H	1	output	Req validation bite from the current GPC_4T to a different GPC_4T in the ring
C2F_ReqOpcodeQ500H	1	output	Req opcode from the current GPC_4T to a different GPC_4T in the ring
C2F_ReqThreadIDQ500H	2	output	Req thread ID from the current GPC_4T to a different GPC_4T in the ring
C2F_ReqAddressQ500H	32	output	Req address from the current GPC_4T to a different GPC_4T in the ring
C2F_ReqDataQ500H	32	output	Req data from the current GPC_4T to a different GPC_4T in the ring (if opcode is Store)

Table 10 - D MEM WRAP interface

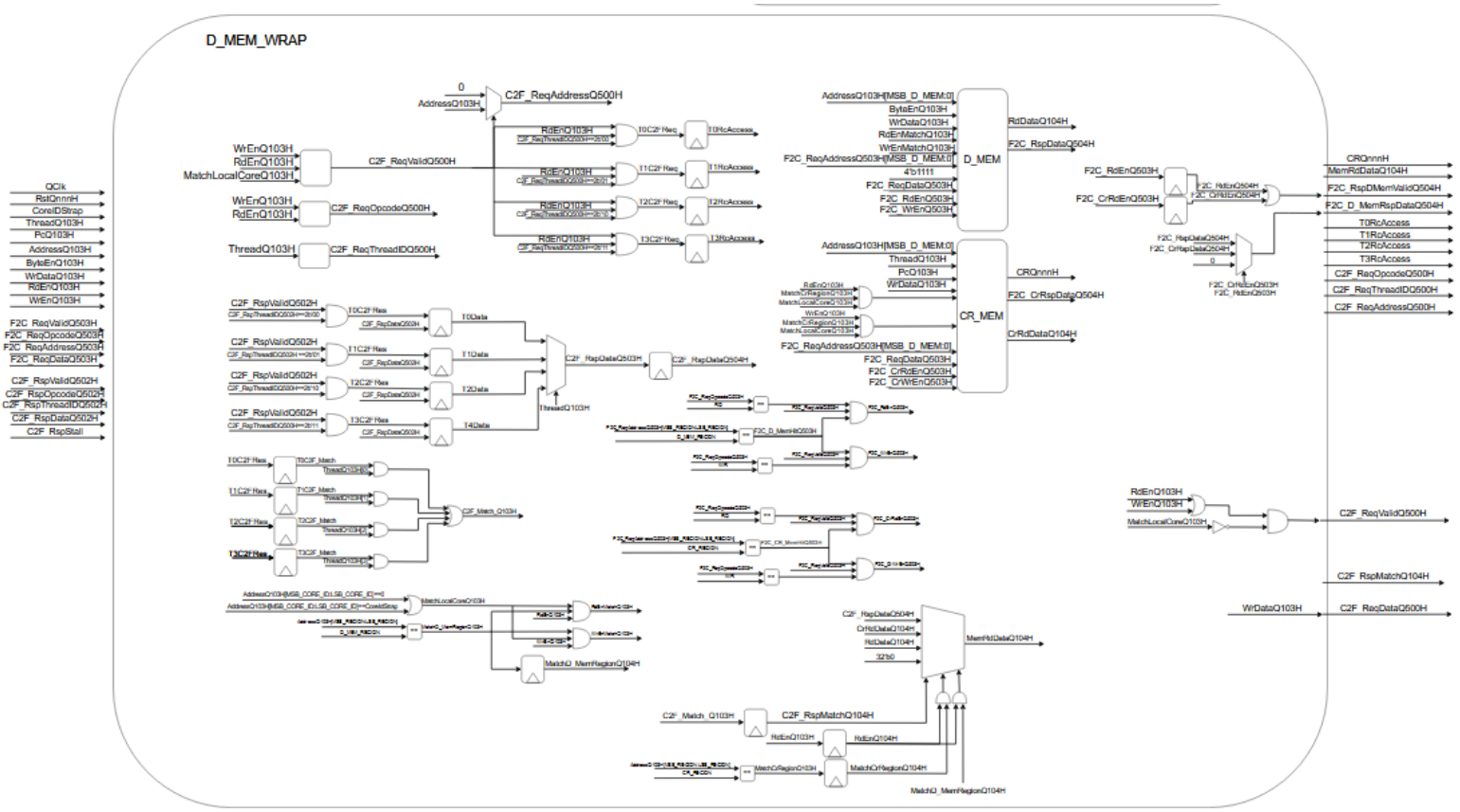
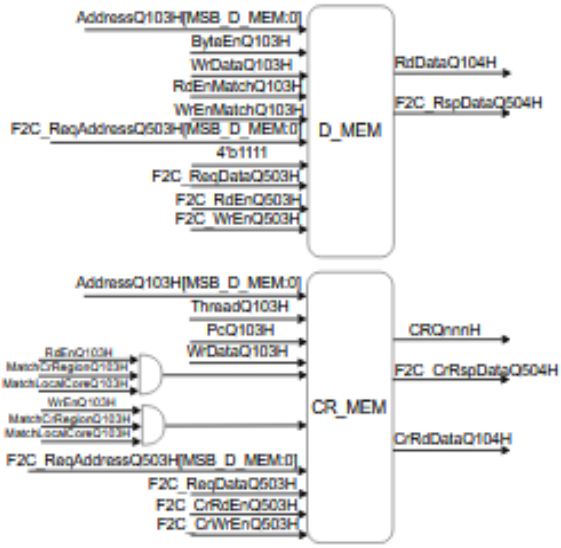


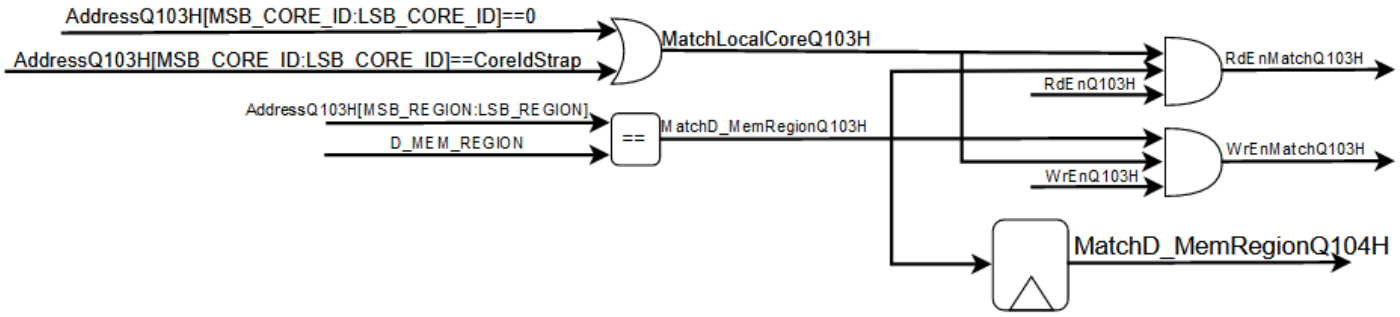
Figure 22 - D MEM WRAP logic design

as seen in the logic design figure, this module's logic is quite complicated. We'll try to simplify.

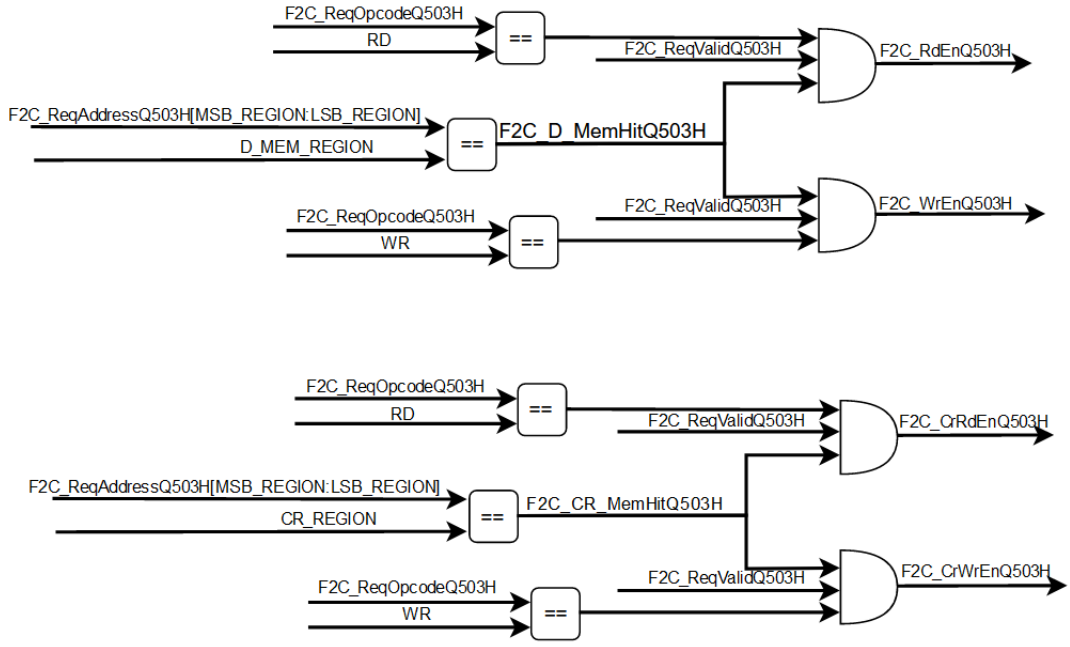
First There are the two instances of the D MEM and CR MEM modules:



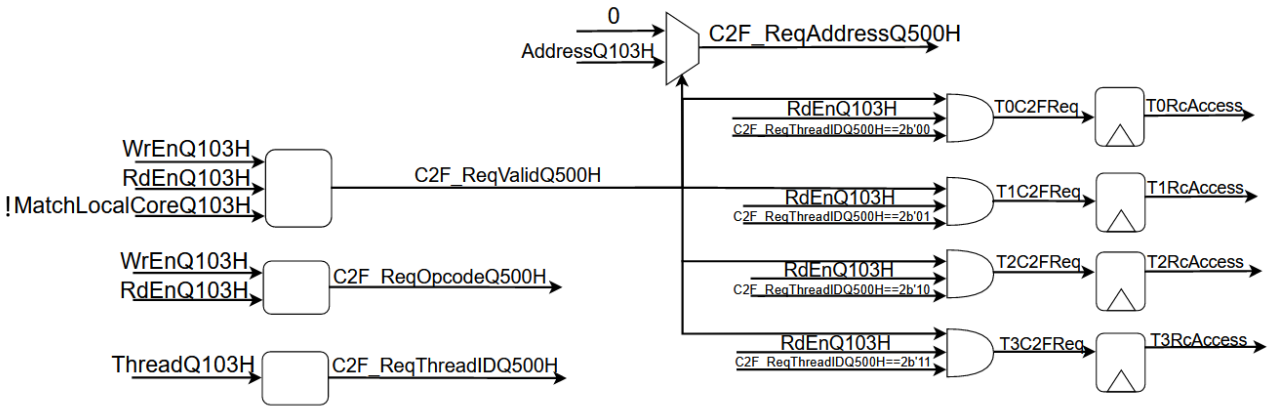
Each are fed by signals and data directly from the Core and the Fabric and by local logic signals. The Core's Read and write enable signals going to the instances, are toggled by the logic above. It must be a local core match and a valid D MEM address + read/write enable from the core.



The Fabric request signals (AKA F2C – Fabric to Core) coming also from the Fabric (By an F2C packet) and from local logic signals like the read and write signals for the Fabric:



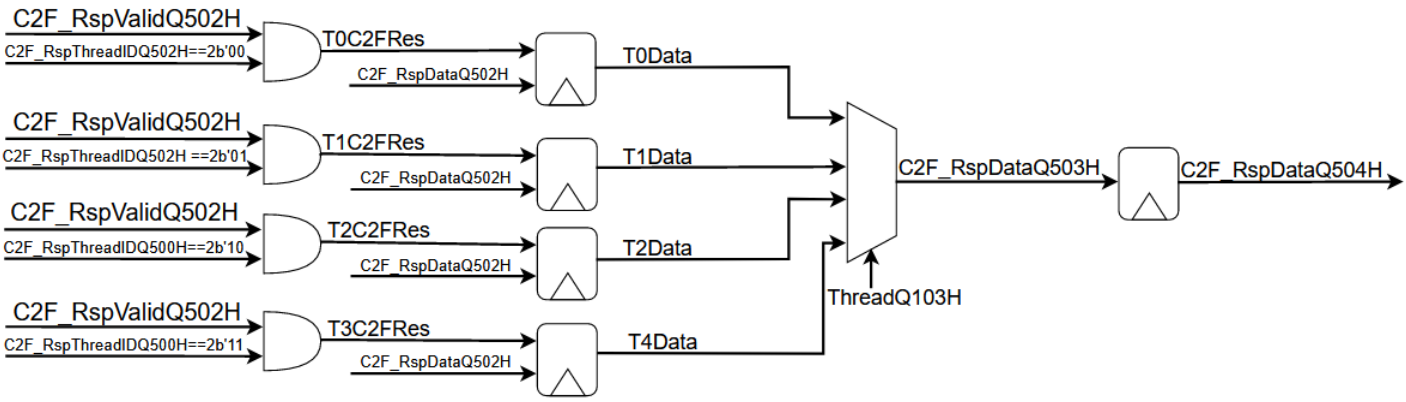
Now we'll describe the logic dedicated to handle non-local memory requests and responses.
The key signal is "MatchLocalCoreQ103H". If this signal set to `0, it means a request to non-local address came from the core (read or write) and a "C2F" packet (Core to Fabric) creates .



If it was a **write** request, the C2F request signals (C2F_ReqValidQ500H, C2F_ReqOpcodeQ500H, C2F_ReqThreadIDQ500H, C2F_ReqAddressQ500H, C2F_ReqDataQ500H) sent to the Fabric.

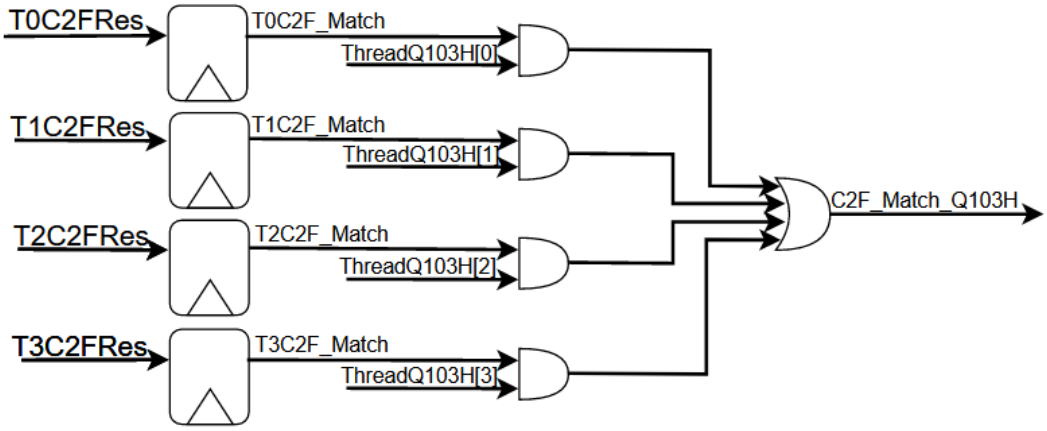
If **read** request, the complex logic comes to an action: 4 dedicated flops (one for each thread) holds the value `1 (depends on the thread) and send this signal to the Core, causing the PC of the thread to freeze as long as the signal is `1.
the C2F request signals are sent to the Fabric.

When the C2F response arrives from the Fabric (the signals C2F_RspValidQ502H, C2F_RspOpcodeQ502H, C2F_RspThreadIDQ502H, C2F_RspDataQ502H), 4 flops (one for each thread) samples the C2F data coming from Fabric. Each flop will sample only if the response Thread (the thread who sent the request) is it – the enable signal of each flop toggled. For example T0 Data flop will sample and hold the C2F data if the C2F response is Valid and the ID of the thread requested is 0. A Mux determine the C2F data based on the current running thread(ThreadQ103H).



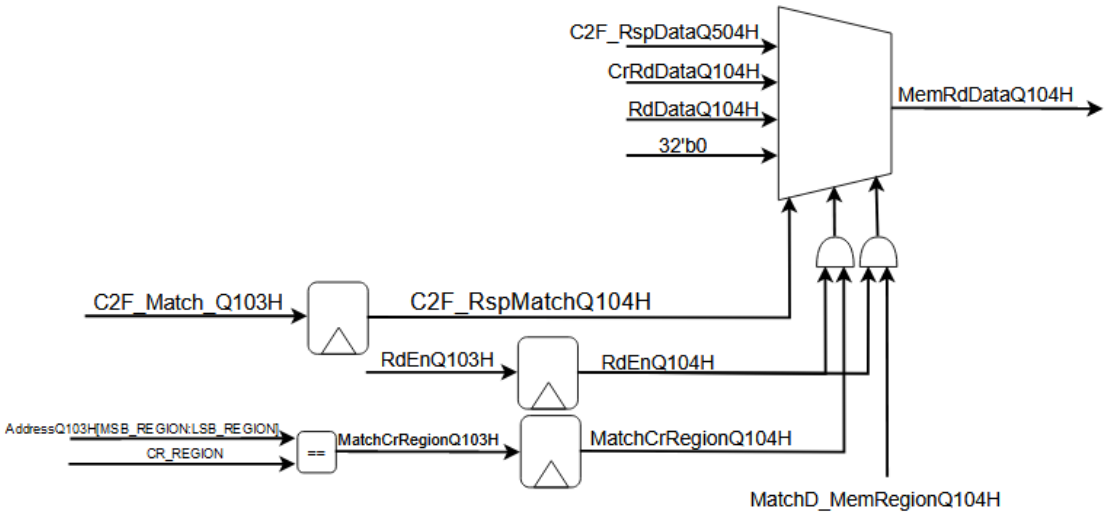
The response can come at any time. it means that if thread0 send request, the response can come when thread2 is currently running on Q103H. Therefore those 4 flops above will hold the data until the turn of the requested thread will come (and let's not forget that the thread is still on Pc Freeze).

For signaling and save the fact that the response arrived for the requested thread, and to release its Pc when its turn arrives, we used 4 dedicated flops holing the value `1 if response came.



Those flops will reset when the thread's turn arrives.
When the requested thread's turn finally arrives, and the "C2F_Match_Q103H" signal is `1 implying that the data from the fabric is ready and this is the threads turn, a **reset signal** will send to the flops freezing the thread's PC and it will be Unfrozen.

The signal "MemRdDataQ104H" which is the signal sent to the Core and holds the memory read data, will be determine by the following priority based on the signals: C2F_Match_Q104H, MatchCrRegionQ104H, MatchD_MemRegionQ104H.
if "C2F_Match_Q104H", implying this is a data from non-local core, then the data sent to Core will be C2F_RspData. If not – the Cr Data. and last the local memory data.



3 DESIGN FLOWS

3.1 THREAD FREEZE USING CRS

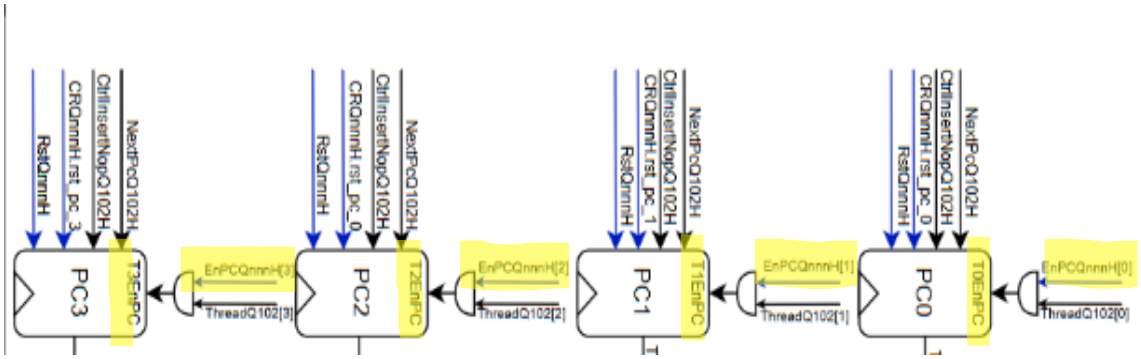
One of the most special uses of the CRs is the ability to "freeze" thread's PC from another thread (or from the thread itself) and force the thread not to progress in the running program.
This feature uses these CRs:

Offset	Name	Type	Structure
150	THREAD<i>_PC_EN	RW	[0]
154			
158			
15c			

Table 11 - CR offsets for thread freeze

Thread that wishes to freeze another thread's PC, will write '0' to the specific CR address.
for example to freeze thread 2, '0' will be written to (0x00C00158) address.

Before explaining the flow, notice the Core implementation of the PC's:




```
assign EnPCQnnnH[0] = CRQnnnH.en_pc_0 && (!(T0RcAccess));
assign EnPCQnnnH[1] = CRQnnnH.en_pc_1 && (!(T1RcAccess));
assign EnPCQnnnH[2] = CRQnnnH.en_pc_2 && (!(T2RcAccess));
assign EnPCQnnnH[3] = CRQnnnH.en_pc_3 && (!(T3RcAccess));

//  Enable bits for Thread's Pc - indicated from Q102H
assign T0EnPcQ100H = EnPCQnnnH[0] && ThreadQ102H[0];
assign T1EnPcQ100H = EnPCQnnnH[1] && ThreadQ102H[1];
assign T2EnPcQ100H = EnPCQnnnH[2] && ThreadQ102H[2];
assign T3EnPcQ100H = EnPCQnnnH[3] && ThreadQ102H[3];
```

Figure 23 - Pc enable code and logic

as seen in image, **T<X>EnPcQ100H** is the "Enable" for the PC flop. it affects if **Thread<x>** is currently running on Q102H and if **EnPCQnnnH[<x>]** is set to '1'. this signal effects by the **CRQnnnH** struct that enter the Core from the CR_MEM module. **If CRQnnnH.en_pc_X** is '0' then it means the Thread's PC will not progress (Enable for flip-flop is off). This is the signal that effects on whether the Thread's PC will freeze or not.

The Flow:

For simplicity let's say thread 0 wants to freeze thread's 2 PC.

1. Thread 0 Write's '0' to the address of the CR "CR_THREAD2_PC_EN" that is (0x00C00158).
2. On D_MEM_WRAP, "**MatchCrRegionQ103H**" turns to '1' due to accessing a CR address:

```
//=========================================
//    core interface
//=========================================
always_comb begin
    MatchLocalCoreQ103H    = (AddressQ103H[MSB_CORE_ID:LSB_CORE_ID] == 8'b0 || Addres
    MatchD_MemRegionQ103H = (AddressQ103H[MSB_REGION:LSB_REGION] == D_MEM_REGION);
    MatchCrRegionQ103H     = (AddressQ103H[MSB_REGION:LSB_REGION] == CR_REGION);

end
```

3. On CR_MEM, the address is decoded and a special struct called "cr_en" is assign in it designated field.

```
always_comb begin : cr_write_en
    cr_en = '0;
    WrEnQn03H[0]     = CrWrEnQ103H;
    WrEnQn03H[1]     = F2C_CrWrEnQ503H;
    for(int IO_NUM =0; IO_NUM<2; IO_NUM++) begin : core_wr_interface_and_ring_wr_interface
        if(WrEnQn03H[IO_NUM]) begin : decode_cr_write_en
            unique case (AddressQn03H[IO_NUM])
                CR_THREAD0_PC_RST              : cr_en[IO_NUM].rst_pc_0       = 1'b1 ;
                CR_THREAD1_PC_RST              : cr_en[IO_NUM].rst_pc_1       = 1'b1 ;
                CR_THREAD2_PC_RST              : cr_en[IO_NUM].rst_pc_2       = 1'b1 ;
                CR_THREAD3_PC_RST              : cr_en[IO_NUM].rst_pc_3       = 1'b1 ;
                CR_THREAD0_PC_EN               : cr_en[IO_NUM].en_pc_0        = 1'b1 ;
                CR_THREAD1_PC_EN               : cr_en[IO_NUM].en_pc_1        = 1'b1 ;
                CR_THREAD2_PC_EN               : cr_en[IO_NUM].en_pc_2        = 1'b1 ;
                CR_THREAD3_PC_EN               : cr_en[IO_NUM].en_pc_3        = 1'b1 ;
```

Figure 24 - Cr enabler code

this struct contains many signals, each is an "Enable" signal for a different flip-flop.

4. The "enable" for a flip-flop that updates the values of the "core_cr" struct that will go to the core , turns to '1' and the value updates with the value of the data that written, in this case is '0' (**WrDataQ103H[SelWrDataQ103H.en_pc_2][0] = 0**):

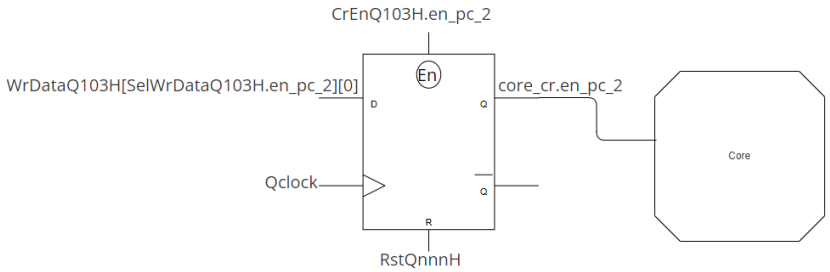


Figure 25 - freeze Pc Cr flop to core logic

5. The "core_cr" struct go to the core with "core_cr. en_pc_2=0" and as explained before flow, the **CRQnnnH.en_pc_2** will be '0' and **EnPCQnnnH[2]** will also be '0' due to it, so the Enable bit for the Thread 2 PC will be off and Thread 2 will be "freeze".

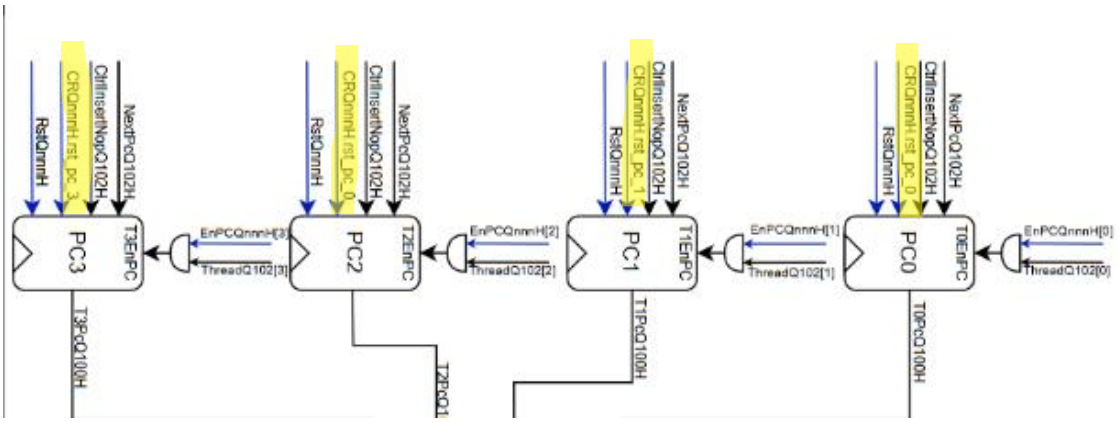
3.2 THREAD RST USING CRS

One other feature of the CRs is the ability to reset thread's PC back to 0. This feature uses these CRs:

Offset	Name	Type	Structure
140	THREAD<i>_PC_RST	RW	[0]
144			
148			
14c			

Table 12 - CR offsets for thread reset

The flow is quite similar to the PC freeze flow explained before.
Notice the implementation of the PC's flops in the core_4t module:



as seen in image, when the signal " **CRQnnnH.rst_pc_<X>**" is set to '1', the " T<X>PcQ100H" will be reset to 0.

The Flow:

For simplicity lets say thread 0 wants to reset thread's 2 PC.

1. Thread 0 Write's '1' to the address of the CR "CR_THREAD2_PC_RST" that is (0x00C00148).

2. On D_MEM_WRAP, "**MatchCrRegionQ103H**" turns to '1' due to accessing a CR address:

```
//=========================================
//    core interface
//=========================================
always_comb begin
    MatchLocalCoreQ103H   = (AddressQ103H[MSB_CORE_ID:LSB_CORE_ID] == 8'b0 || Addres
    MatchD_MemRegionQ103H = (AddressQ103H[MSB_REGION:LSB_REGION] == D_MEM_REGION);
    MatchCrRegionQ103H    = (AddressQ103H[MSB_REGION:LSB_REGION] == CR_REGION);

end
```

3. On CR_MEM, the address is decoded and the "cr_en" struct is assign in it designated field.

```
always_comb begin : cr_write_en
    cr_en = '0;
    WrEnQn03H[0]     = CrWrEnQ103H;
    WrEnQn03H[1]     = F2C_CrWrEnQ503H;
    for(int IO_NUM =0; IO_NUM<2; IO_NUM++) begin : core_wr_interface_and_ring_wr_interface
        if(WrEnQn03H[IO_NUM]) begin : decode_cr_write_en
            unique case (AddressQn03H[IO_NUM])
               CR_THREAD0_PC_RST             : cr_en[IO_NUM].rst_pc_0       = 1'b1 ;
               CR_THREAD1_PC_RST             : cr_en[IO_NUM].rst_pc_1       = 1'b1 ;
               CR_THREAD2_PC_RST             : cr_en[IO_NUM].rst_pc_2       = 1'b1 ;
               CR_THREAD3_PC_RST             : cr_en[IO_NUM].rst_pc_3       = 1'b1 ;
               CR_THREAD0_PC_EN              : cr_en[IO_NUM].en_pc_0        = 1'b1 ;
               CR_THREAD1_PC_EN              : cr_en[IO_NUM].en_pc_1        = 1'b1 ;
               CR_THREAD2_PC_EN              : cr_en[IO_NUM].en_pc_2        = 1'b1 ;
               CR_THREAD3_PC_EN              : cr_en[IO_NUM].en_pc_3        = 1'b1 ;
```

4. The "enable" for a flip-flop that updates the values of the "core_cr" struct that will go to the core , turns to '1' and the value updates with the value of the data that written, in this case is '1' (**WrDataQ103H[SelWrDataQ103H.rst_pc_2][0] = 1**):

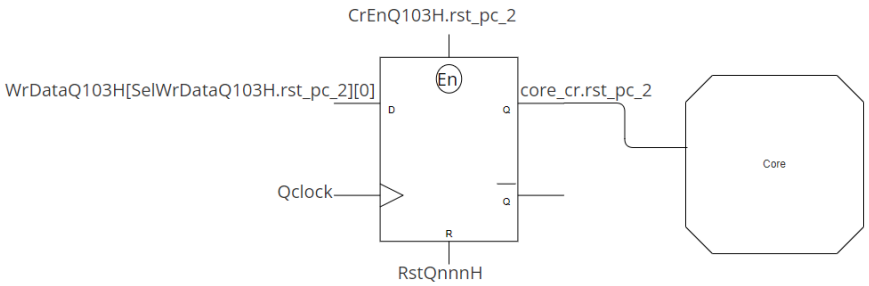


Figure 26 - reset Pc Cr flop to core logic

5. The "core_cr" struct go to the core with "core_cr. rst_pc_2=1" and as explained before flow, **CRQnnnH.rst_pc_2** will be '1' and " **T2PcQ100H**" will be reset back to 0

3.3 Multi Thread Support (Single 4 Thread Core)

In the design, there is a single instruction memory which the core reads instruction from. Therefore all 4 threads reading the same program.
The solution we found to run multithread applications is to use a dedicated CR register known as “CR_THREAD” which is located by the offset 0x4 on the CR address space (full address 0x00C00004). This flop is a read only flop contains the Id of the current thread running on stage Q103H (memory access stage).

- Each thread reads this CR’s value and then can use direct branch to jump a dedicated section of the program
- Each thread now initiates separate section of the code (for example different function)
- When a thread finished its section, it will perform a busy wait :
    - first signal the other threads it finished (by writing `1 to a sheared memory address or CR scratchpad for example)
    - Later waits for other threads signals.
- First thread to finish the busy wait, terminates the program.
  A detailed example in the Validation Plan.

3.4 WRITING TO NON-LOCAL MEMORY

As explained, data can be written to another Core's memory using the C2F buffer of the Ring Controller.

For simplicity let's say thread 1 wants to write data on the address 0x02000200

The Flow:

1. Thread 1 send write request with data to the address : (0x02000200) .
2. in the D_MEM_WRAP:
   2.1. a signal called " MatchLocalCoreQ103H" is set to '0' because of the 10 MSB digits (which are called “core ID”) in the address indicates this address is not in the local core.
   2.2. C2F packet is assembled:

packet signal	description	value
C2F_ReqValidQ500H	1 if MatchLocalCoreQ103H=0 and we have "write enable"	1
C2F_ReqOpcodeQ500H	type of C2F request	WR
C2F_ReqThreadIDQ500H	the thread requested the transaction	01
C2F_ReqAddressQ500H	address to be accessed	0x02000200
C2F_ReqDataQ500H	data to be written	[31:0] of data

Table 13 - C2F write request packet

3. The packet signals are sent to the Ring Controller which will allocate a “C2F” Buffer Entry. eventually The C2F Requests will exit our “Tile” towards the Fabric.
4. The packet arrives to the designated Tile (Core Id = 02) and will allocate a “F2C” Buffer Entry. Eventually the RC buffer will send the requests to the core. using the F2C_Req Interface to the D MEM WRAP and the data immediately written on this (Core 2) core memory.

It can be notice that for the requester, it’s not different from a local memory write request – fire and forget.

```
assign C2F_ReqValidQ500H    = (WrEnQ103H||RdEnQ103H) && !MatchLocalCoreQ103H;
assign C2F_ReqOpcodeQ500H   = WrEnQ103H ? WR :
                              RdEnQ103H ? RD : RD;
assign C2F_ReqThreadIDQ500H = (ThreadQ103H == 4'b0001) ? 2'b00 :
                              (ThreadQ103H == 4'b0010) ? 2'b01 :
                              (ThreadQ103H == 4'b0100) ? 2'b10 :
                                                         2'b11 ;
assign C2F_ReqAddressQ500H  = C2F_ReqValidQ500H ? AddressQ103H : 0;
assign C2F_ReqDataQ500H     = WrDataQ103H;
```

Figure 27 - C2F packet signals code

3.5 READING FROM NON-LOCAL MEMORY

As explained in HAS, data can be read from another Core's memory using the C2F buffer of the Ring Controller.
The special thing about this read request, is that when a thread from the local core sends the request, its PC is frozen until the respond with the read data arrives from the ring controller. This could take some clock cycles depends on the number of cores.

For simplicity let's say **thread 1** wants to read data on the address 0x02000200

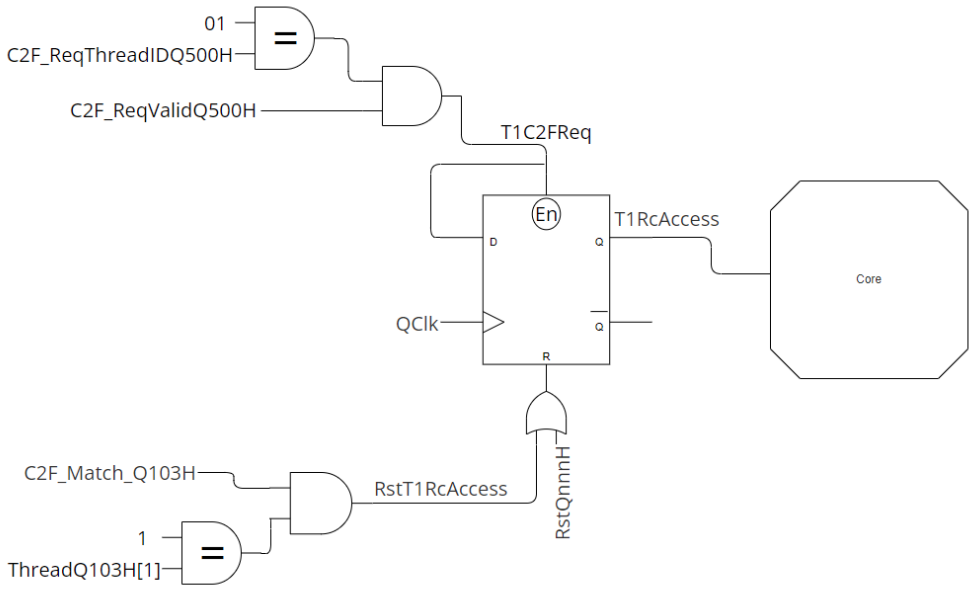
The Flow:

1. Thread 1 send read request to the address: (0x02000200)
2. in the D_MEM_WRAP:
   2.1. a signal called " MatchLocalCoreQ103H" is set to '0' because of the 10 MSB digits (Core ID) in the address indicates this address is not in the local core.
   2.2. C2F packet is assembled:

packet signal	description	value
C2F_ReqValidQ500H	1 if MatchLocalCoreQ103H=0 and we have "read enable"	1
C2F_ReqOpcodeQ500H	type of C2F request	RD
C2F_ReqThreadIDQ500H	the thread requested the transaction	01
C2F_ReqAddressQ500H	address to be read data from	0x02000200
C2F_ReqDataQ500H	data to be written	

Table 14 - C2F read request packet

   2.3. A signal called " T1C2FReq" will be set to 1
   2.4. A flip-flop with output signal "T1RcAccess" will be sampled with "1" as the " T1C2FReq" signal is used as the Enable bit for the flop.
   2.5. " T1RcAccess" is an output to the Core.



   2.6. in the later cycles, the signal T1C2FReq is set to '0' and the flip flop is locked with the value T1RcAccess = '1'.

2.7. The packet signals are sent to the Ring Controller which will allocate a “C2F” Buffer Entry. eventually The C2F Requests will exit our “Tile” towards the Fabric

3. In the core, the signal " T1RcAccess" shuts down the enable bit for Thread 1 and freeze it.

4. The packet goes thru the RC and arrives to the designated core (Core Id = 02) with the RC inputs to the D MEM WRAP and the data immediately read from this core memory to the signal "F2C_RspDataQ504H".

5. The packet data send to the Ring Controller and in the RC design it transformed to an C2F respond packet.
6. thru all this time the thread sends the read request is on freeze.
7. In D MEM WRAP:

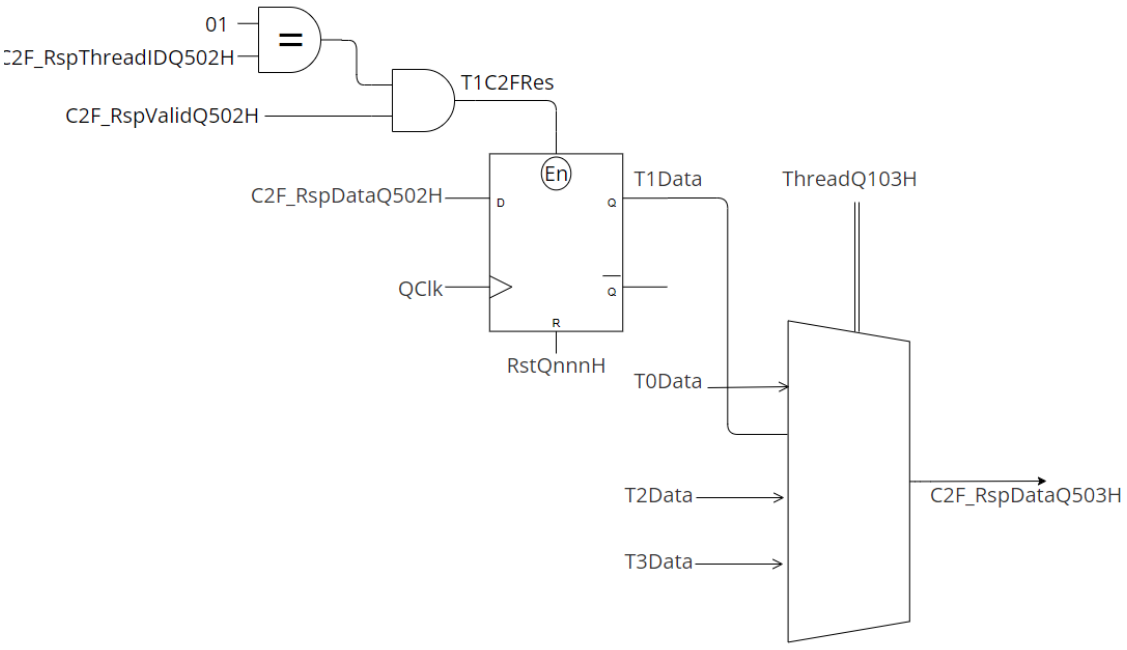
7.1. When C2F Response packet arrived (the signal C2F_RspValidQ502H='1') a C2F response packet assembled:

packet signal	description	value
C2F_RspValidQ502H	comes as an input from RC	1
C2F_RspThreadIDQ502H	the thread requested the original transaction	01
C2F_RspDataQ502H	data that read	some data

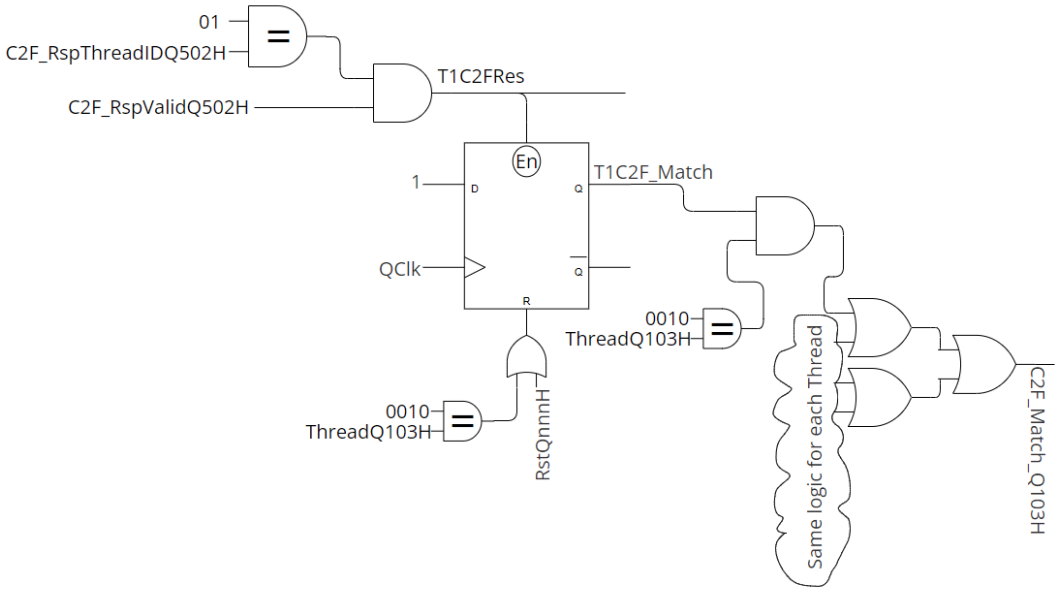
Table 15 - C2F response packet

7.2. A signal "T1C2FRes" set to '1' and that signal uses as the Enable bit for a flip-flop that samples the "C2F_RspDataQ502H" data on a signal called "T1Data". in the next cycle the T1C2FRes is set to '0' and "T1Data" is locked on the value that "C2F_RspDataQ502H" was when the packet arrived. this feature is used in order to save this data until the turn of the thread that sent the original read request comes.

7.3. When the thread's turn arrives, a signal named "C2F_RspDataQ503H" get the data from "T1Data" and will sampled to "C2F_RspDataQ504H".



7.4. another signal named " T1C2F_Match" is sampled to '1' by another flop with Enable bit as "T1C2FRes" like the previous flop.



7.5. Signal named "C2F_Match_Q103H" will set to '1' when "T1Data"='1' and the running thread is thread 1. When the thread turns arrived, this signal will indicate the D MEM WRAP output signal " MemRdDataQ104H" to take the value of "C2F_RspDataQ504H".

7.6. Signal named "RstT1RcAccess" will set to '1' when "C2F_Match_Q103H" is '1' and the current thread is thread 1. this signal will reset the flip flop with output signal "T1RcAccess" back to '0' and this will release thread 1 from Freeze and its PC will continue.

7.7. " MemRdDataQ104H" signal will be transferred to the core with the data read from another core.