General guideline: Please do not expect any response from the TAs 48 hours before any deadline(s) for the projects mentioned below. Please carefully plan the deliverables accordingly.

Project 1 – Designing a Cycle Accurate Simulator for Network-on-Chip (NoC) router and mesh with provision to study the impact of Process Variations

What is an NoC?

As Systems-on-Chips become more demanding, complex, and elaborate, interconnection via traditional bus-based networks became highly inefficient and slow, giving rise to the development of network-on-chip architectures. The basic and major advantage of NoCs over bus communication is that multiple processing elements (PEs) like CPU, GPU, ADC, Memory, or any other IP, can communicate with one another simultaneously (parallelly) with the help of **"Routers"** attached to every peripheral. As the name also suggests, Routers decide to direct the data to various network peripherals (connected to other routers). Every router is connected to at least two other routers and one PE. In a mesh design, a router has **five** ports - **four** for other routers (**North, South, East, and West**) and **one for PE**. Note that *not all ports of the router need to be connected*. For example, in the NoC shown below (Fig. 1), the northern ports of routers A and B are not connected to anything.

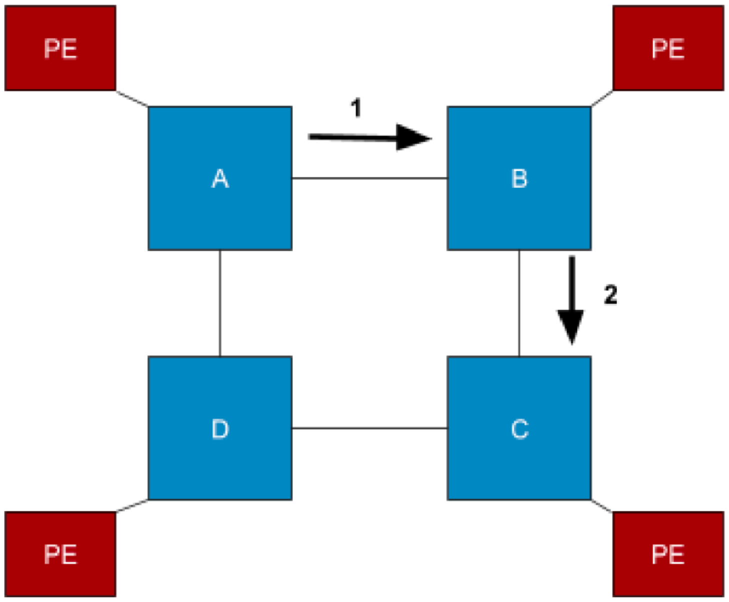


Fig. 1. Simple 2X2 mesh NoC Architecture with Processing Elements

Now, let us consider that router A has to send some data to router C. Two possible paths include ABC and ADC. To remove ambiguity in the path selection, **"routing algorithms"** are used. "**XY routing"** is one such algorithm that prioritizes horizontal movement **(X-axis)** over vertical **(Y-axis)** one. In this example, XY routing allows movement from A to B but not through A to D. So, the XY routing path in the above example is through ABC. This path is demonstrated in Fig. 1 using arrows. Since we are using XY routing, the packet first travels in the X direction (arrow number 1) and then travels in the Y direction (arrow number 2). Note that if no horizontal path exists, only vertical movement is done. For instance, vertical movement from A to D is done if the data transfer between A and D is needed.

A "**packet"** is a basic data transfer unit in an NoC. However, the packets cannot be transmitted from one node (**source**) to another (**destination**) in a single go. To aid transmission, the packet is broken down into **flits**. The flits can be transferred over a single go; hence, the packet is transmitted as a series of flits. **Flits are of three types - "header flit (HF), body flit (BF), and tail flit (TF)."** The **header** flit contains the control information (source, destination, etc.) and is responsible for creating the path for the network. The **body** flits contain the data to be transferred, and the **tail** flits indicate the end of the packet.

The following Fig. 2 shows a simple NoC router **microarchitecture**. The router has **one input** and **one output** port in **each direction (North, South, East, and West).** The **"crossbar (XBAR)"** connects every input to every output of a NoC (except in the same directions). The **"switch allocator (SA)"** configures the crossbar according to the header flit and routing algorithm. Whenever a packet (or flit) enters a router, it is initially stored in the "**input buffers**" corresponding to the respective "**input port**". For instance, in the above Fig. 1, router A's PE will inject the packet (or flit) at the local input port, which will stay in buffer till it is decoded as the type of flit, then switch allocator will configure the XBAR to connect local input port to east output port which will be received by router B's west input port followed by the buffer. The router B's switch allocator will configure the crossbar to connect the west input port to the south output port. Further details about NoC can be found in the attached book.

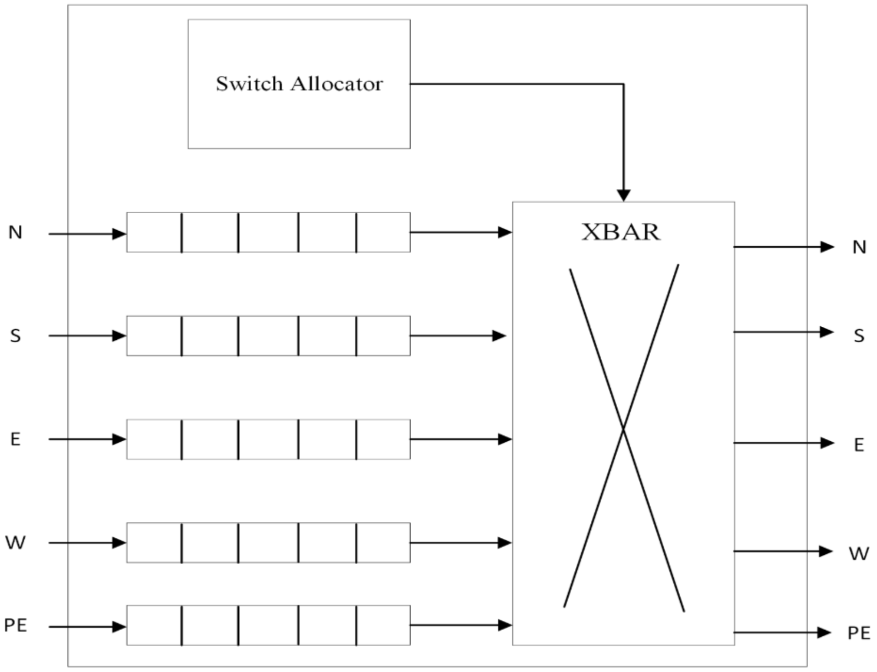


Fig. 2. Router Microarchitecture

Project Description:

1. In this project, you are required to create a NoC simulator in a language of your choice. Recommended language choices are (but are not limited to) Python3, C, or C++.
   Since this is not a programming-focused course, you should choose whatever language you are comfortable with, especially for debugging purposes.
2. You have to simulate a **NoC router** and a **3x3 NoC mesh containing nine routers**. As described above, each router needs to have five ports (north, south, east, west, and local) to connect to its neighbors and connect to its local PE. Some ports of the routers may be left unconnected if there is no neighbor (for example, the north port of router B in Fig. 1).
3. Your router must support XY routing as described above. Moreover, your router must also support YX routing, in which the packet travels first in the Y direction and then in the X direction. The type of routing to be used will be passed to your simulator via the command line arguments.
4. Your simulator needs to support reading a "**traffic file"** that describes which **packets (split into three types of flits: HF, BF,& TF)** are inserted in the NoC at which respective clock cycles. The **traffic file** is a **text file** with **four** values mentioned in each line, separated by spaces. The value in the **first** column describes the clock cycle where a packet/flit is inserted into the NoC. The **second** and **third** columns' values represent the **source** and the **destination** for the packet, respectively. The **fourth** value is a 32-bit value representing the flit which needs to be injected in NoC at the clock cycle (the first column).

   → The Bits of the flit-types of a Packet are defined as follows:

   a. **Head Flit (Flit Type = 00) :**

Destination Node ID															Source Node ID															Type	
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

b. **Body Flit (Flit Type = 01) :**

Payload (30 Bits Wide)																														Type	
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. **Tail Flit (Flit Type = 10) :**

Unused (any value)																														Type	
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

5. Your simulator must be **cycle accurate**, which means that your simulator must be able to simulate clock cycles. Your routers' data transfer and state update must functionally happen at the positive clock edges.
6. Each router must have a crossbar, a switch allocator, input buffers, and I/O ports. You can find the details of these sub-components in the aforementioned description. These sub-components must be visible in the code.
7. Every router in your NoC must accept the packets (split into flits) via the local input port (ideally connected to PE) at the specified clock cycle in the "traffic file".
8. The simulator must read the delay of individual router elements (SA, XBAR & buffers) via a **"delays file". Nominal** Delays to the routers' elements described in Fig. 1. must be defined in the delays file for all of the nine routers (3x3 NoC mesh) elements. The clock frequency has to be derived from these delays assuming that the router works in a pipelined fashion, i.e., first, the flit enters the buffers, then if it is a header-flit, it gets decoded by the SA unit. Then, the crossbar gets switched accordingly, and the flit passes through XBAR to the next node. These three operations in buffer, SA, and XBar must be assumed to be pipelined for a packet containing three flits: head, body, and tail. So, the final time period of the clock depends on the element of the router having maximum delay.
9. For this project, you may assume there will be no congestion issues, i.e., no flit will have to wait in the buffer for the previous flit to be handled.
10. You may implement the connection as an array in which the sender writes the data to the receiver's array.
11. You have to enable two types of Simulation Modes for simulation, analysis, and reporting: **Process-Variation-Agnostic (PVA) and Process-Variation-Supported (PVS):**
    a. <u>**PVA**</u>: This is the FIRST simulation mode. The chosen **nominal** delays of individual router element(s) (buffer, SA, and XBar) must remain the same across the complete NoC. The clock frequency is derived using the delay file's nominal delays. For example, if you choose buffer, SA, and Xbar delay to be x, y & z, respectively, then these, "x, y & z" values should be applied to all the routers in NoC.
    b. <u>**PVS**</u>: This is the SECOND simulation mode. Following a **Gaussian Normal Distribution,** the above delays should be **randomly** assigned to the different routers' elements in this simulation mode **by your simulator only**. However, the clock frequency decided during PVA will remain the same without any modification in this mode. **The mean** value of the delay distribution will be the **nominal** delays considered in <u>PVA,</u> and **sigma** (standard deviation) must be taken as **10%** of those **nominal** delay values. You must ensure that delays ranging from **"mean + 3*sigma"** and **"mean – 3*sigma"** must be spread according to the Gaussian Bell curve of standard normal distribution across the complete NoC Mesh. For example, the delay "x" of the buffer element must be varied from **"x + 3*sigma_x"** to **"x - 3*sigma_x"** where **"sigma_x = 10% of x",** across the complete NoC**.** The same applies to other element delays as well.
12. Your simulator must generate a **log** file that indicates <u>which flit is received at each cycle (cycle number should be displayed)</u>, <u>at what router</u>, and <u>at what element of that router</u> so that we can trace the flit flow at <u>**every clock cycle**</u>. Your log file must indicate the <u>contents of the flit received at every clock cycle</u>. The log file should indicate the <u>clock cycle(s) at which different flits were inserted in the NoC</u>.
13. Additionally, a separate **report** file must be generated from the simulator, which indicates the impact of process variation applied in <u>PVS</u> mode, i.e., if any node cannot receive a particular flit sent (in case the flit gets delayed than expected time in nominal simulation mode) or not or if any node receives the flit earlier than the expected delay time. So, in addition to clock cycle-wise tracking of packets, your report has to <u>indicate in terms of delay units,</u> <u>which flit took what exact delay units</u> to reach <u>which exact element of which</u>

exact router in the 3x3 NoC first for **PVA simulation mode** and then for **PVS simulation mode** for an apple-to-apple comparison.

14. The user (here, the evaluator TA) will give three inputs to the simulator,
    a. Traffic file
    b. Delays file
    c. Choice of routing algorithm XY or YX
15. The simulator is expected to generate two output files and two graphs:
    a. Log file (as explained in Point #12.)
    b. Report file (as explained in Point #13.)
    c. A graph that plots the number of flits sent over a connection. A connection is a link between two routers or between a router and its PE. So in your case, you will have eight links. Your graphs must show the number of flits sent over each of these eight links as a bar graph. Plot for both PVS and PVA modes.
    d. A graph showing the packet transfer latency for each packet. Each packet transferred via the NoC will take some clock cycles to go from the source to the destination. You must plot this latency as a function of packets sent. Plot for both, PVS and PVA modes.

Summary of Deliverables:

To summarize, you'll have the following deliverables for the final evaluation:

1. A cycle-accurate NoC simulator which:
    a. Performs PVA simulation, followed by PVS simulation (automated).
    b. Supports XY and YX routing.
    c. It supports reading the traffic and delays information from a text file.
    d. Generates a log file that describes what is happening at each clock cycle.
    e. Generates a report file that compares the PVS and PVA modes of simulation.
    f. Interprets the log file generated by the simulator and plots the following graphs:
        i. Number of flits transferred as a function of connections.
        ii. Latency as a function of packets sent.
2. A README file that describes:
    a. The working and usage of your simulator.
    b. The building instructions (if any).
    c. The usage instructions use the command line interface.
3. Upload the code to GitHub and regularly update it. Make the project-TA(s) as a collaborator.

- **Mid-Project Evaluation (MPE):**
  We have elaborated the mid-project deliverables for your convenience, which can be found only after reading the aforementioned project's details. Following are deliverables for MPE:
    1. Your simulator must read and interpret the **traffic file.**
    2. Your simulator must be able to read and interpret the **delays file.**
    3. Your simulator must also support *at least one* of the **routing algorithms.**
    4. Your simulator must also be able to inject packets as per the traffic file.
    5. A working simulator that can generate the **log file** for *at least* **PVA mode.**
    6. A working simulator that can generate the **report file** for *at least* **PVA mode.**
    7. The **log** file should include the cycle count and the flits received in that cycle.

  Even though you can use your own traffic and delays file, the evaluator will randomly change values to test your simulator, which should perform expectedly.
- **End-Project Evaluation (EPE):**
  Please ensure the following are completed by EPE:
    a. Whatever you could not implement in MPE has to be completed before EPE.
    b. All the modes, i.e., the PVS and PVA, must be implemented and run on the same input files – traffic and delays.
        i. The delays file is directly used for PVA mode, but delays have to be derived and independently modified in PVS mode without changing the original delays file.
        ii. Apple-to-apple performance comparison has to be clearly shown for PVS and PVA.

c. The two types of graphs must be generated for both PVA and PVS modes.

Bonus (maximum Bonus is capped at 10% of EPE):

Note: Bonus marks will not be awarded in case any one of the EPE/MPE deliverables are missing.

- 5% of EPE: Make the simulator support a modified traffic file with only three columns, first for the clock cycle , second for the source ID, and third for the packet (instead of a single flit, the packet has all three types of flits concatenated) inserted in NoC at that clock cycle.
- 10% of EPE: Simulated NoC should support congestion with modified traffic file (in the above point), i.e., multiple packets wanting to be inserted at the same clock cycle at the same source, and multiple packets at different input port buffers inside the NoC router demanding the access to the same output port. The NoC should be able to serve the waiting packets by storing them in buffers and stalling them wherever and whenever necessary. Any other solution will also be appreciated.

Project 2: Implementation of Memory Subsystem with Approximate Data Transfer

Project Summary: Whenever there is a cache miss in L1 data (L1D) cache, data is fetched from the next memory layer. However, as the next memory layer is not close to the core, it takes a significant amount of energy in order to bring data from next memory layer to the L1D cache. In this project, you will implement a memory subsystem consisting of an L1D cache and a main memory. Whenever there is a miss in L1D cache, an error tolerance information will be sent along with the data request that aims to reduce power consumption. This project will be implemented using the **HDL** language of your choice (Verilog, VHDL, SystemVerilog), though we will be able to help you only in Verilog.

Project Description:
In this project, we have a 2 KB, 4 way set associative L1D cache. Each cache line is of 32 bytes each and processor word size is of 4 bytes. Note that this size excludes the tag information present in each cache block. Just as in a typical cache, just passing the address should give a word size value. During the evaluation, a 32 bit address will be given and the output should be a word (4 byte) value.

Whenever there is a cache miss in L1D cache, the memory request is sent to a 8 KB main memory, with each location of 4 bytes each. Note that in case of cache miss, the cache line is fetched and not just the word missed. As each cache line is of 32 bytes and processor size is of 4 bytes, the transfer happens as a stream of eight 4-byte numbers. Make sure that the first data is multiple of 8, else wrong data may be passed as output. As the main memory may not be available, a simple protocol consisting of four signals- CLK, VALID, READY and DATA. A transfer takes place when both **VALID** and **READY** are asserted. The summary of each signal is given below:

Signal	Source	Description
CLK	Clock Source	Global clock signals. All signals are sampled at the rising edge of the clock.
VALID	Master	**VALID** indicates that the master is driving a valid transfer. For both load and store operation, asserted by L1D cache to initiate transfer. For load operation, once READY signal is received, it sends the address. Deasserted when the data is received. For store operation,once READY signal is received, it sends the address followed by the data. Deasserted when the data is sent.
READY	Slave	**READY** indicates that the slave can accept a transfer in the current cycle. For both load and store operation, asserted by the main memory in next clock cycle after receiving the VALID request. For load operation,it sends the data to the L1D cache after receiving the address. Deasserted by the main memory after last data is sent. For store operation, it stores the data in the buffer and then modifies the main memory. Deasserted by the main memory after last data is received.
DATA[31:0]	Master/Slave	**DATA** is the primary data passing across the interface. The L1D cache sends(recieves) the address(data) to (from) the main memory using this bus.

The data transfer in this case happens as a sequence of eight-four byte numbers. This causes a lot of area consumption. In order to reduce the data transfer, we compress the data before sending it across the protocol. In this project, you will implement the AxDeduplication compression scheme discussed in AxBA. Note that you just need to implement the AxDeduplication scheme and not the software aspects of AxBA.

Mid-Project Deliverables:

1. Implement the memory subsystem for load operation without compression.
2. During the evaluation, the address and the data present in the main memory and L1D cache will be provided. Highlight the protocol signals and explain the usage.

End-Project Deliverables:

1. Completed memory subsystem with load/store compression
2. Upload your final code to a GitHub repository.

Project 3-Memory Ordering Demonstration in Multicore Shared Environments and Implementation of Sequential consistency and Total Store Order

Project Description: Modern computer systems, particularly multicore chips or chip multiprocessors, incorporate hardware support for shared memory. In shared memory systems, each processor core has the capability to both read from and write to a single, shared address space. The concept of consistency, often referred to as the memory consistency model, plays a pivotal role in establishing the correctness of shared memory interactions. Consistency definitions lay down rules governing how memory operations, such as reads and writes, interact with the memory itself.

Consistency models serve as frameworks for determining the proper behavior of shared memory within a system, focusing exclusively on memory operations like reads and writes, independent of considerations about caches or coherence mechanisms. These models delineate the acceptable conduct of multithreaded programs running on shared memory. For a given multithreaded program operating with specific input data, the memory model outlines the potential values that dynamic loads can produce and the range of possible end states for the memory. Unlike the more straightforward scenario of single-threaded execution, the presence of multiple correct behaviors introduces complexity to the understanding of memory consistency models.

These memory consistency models, often referred to as memory models, perform the crucial role of defining how shared memory systems should function for both programmers and system implementors. By establishing criteria for correctness, these models provide programmers with expectations regarding the behavior of their code, and they offer implementors guidance on the necessary provisions to ensure proper system operation.

To see why shared memory behavior must be defined, consider the example execution of two cores depicted in Table 3.1 (assuming that the initial values of all variables are zero). Most programmers would expect that core C2's register r2 should get the value NEW. Nevertheless, r2 can be 0 in some of today's computer systems. Hardware can make r2 get the value 0 by reordering core C1's stores S1 and S2. Locally (i.e. if we look only at C1's execution and do not consider interactions with other threads), this reordering seems correct because S1 and S2 access different addresses. With the reordering of S1 and S2, the execution order may be S2, L1, L2, S1, as illustrated in Table 3.2.

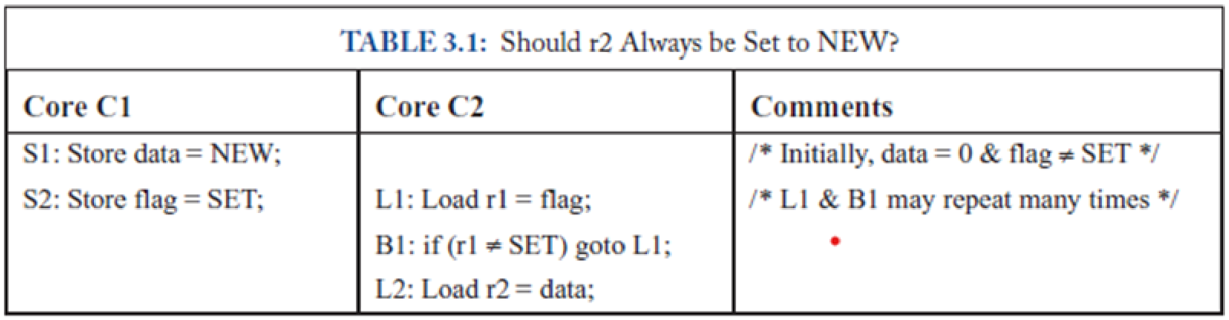


TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

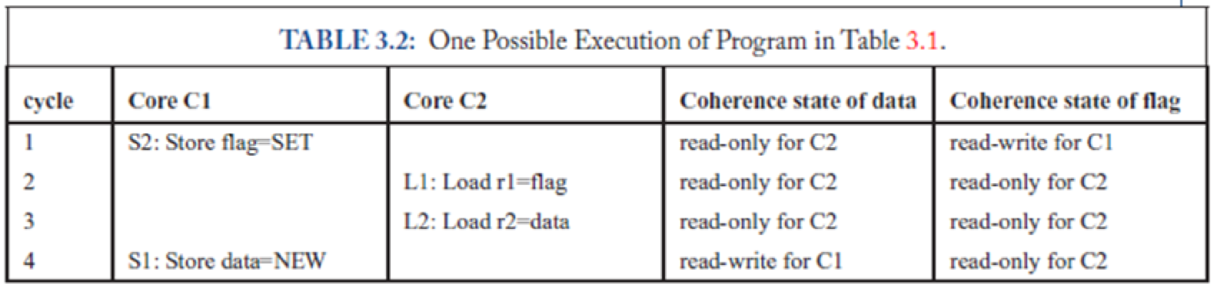


TABLE 3.2: One Possible Execution of Program in Table 3.1.

cycle	Core C1	Core C2	Coherence state of data	Coherence state of flag
1	S2: Store flag=SET		read-only for C2	read-write for C1
2		L1: Load r1=flag	read-only for C2	read-only for C2
3		L2: Load r2=data	read-only for C2	read-only for C2
4	S1: Store data=NEW		read-write for C1	read-only for C2

How a Core Might Reorder Memory Accesses:

Store-store reordering. Two stores may be reordered if a core has a non-FIFO write buffer that lets stores depart in a different order than the order in which they entered. This might occur if the first store misses in the cache while the second hits or if the second store can coalesce with an earlier store (i.e., before the first store). Note that these reordering are possible even if the core executes all instructions in program order. Reordering stores to different memory addresses has no effect on a single-threaded execution. However, in the multithreaded example of Table 3.1, reordering Core C1's stores allow Core C2 to see the flag as SET before it sees the store-to-data.

Load-load reordering. Modern dynamically scheduled cores may execute instructions out of program order. In the example of Table 3.1, Core C2 could execute loads L1 and L2 out of order. Considering only a single-threaded execution, this reordering seems safe because L1 and L2 are to different addresses. However, reordering Core C2's loads behave the same as reordering Core C1's stores; if the memory references execute in the order L2, S1, S2 and L1, then r2 is assigned 0. This scenario is even more plausible if the branch statement B1 is elided, so no control dependence separates L1 and L2.
Load-store and store-load reordering. Out-of-order cores may also reorder loads and stores (to different addresses) from the same thread. Reordering an earlier load with a later store (a load-store reordering) can cause many incorrect behaviors.
A memory consistency model, or, more simply, a memory model, is a specification of the allowed behavior of multithreaded programs executing with shared memory. For a multithreaded program executing with specific input data, it specifies what values dynamic loads may return and what the final state of memory is.

SEQUENTIAL CONSISTENCY (SC)

The most intuitive memory consistency model is sequential consistency (SC). Lamport first formalized sequential consistency. Lamport first called a single processor (core) sequential if "the result of an execution is the same as if the operations had been executed in the order specified by the program." He then called a multiprocessor sequentially consistent if "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program." This total order of operations is called memory order. In SC, memory order respects each core's program order, but other consistency models may permit memory orders that do not always respect the program orders.

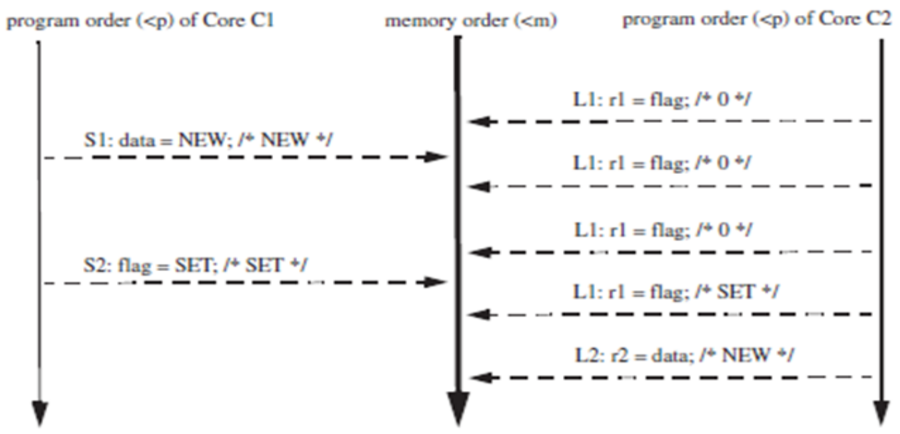


FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

Figure 3.1 depicts an execution of the example program from Table 3.1. The middle vertical downward arrow represents the memory order (<m) while each core's downward arrow represents its program order (<p). We denote memory order using the operator <m, so op1 <m op2 implies that op1 precedes op2 in memory order. Similarly, we use the operator <p to denote program order for a given core, so op1 <p op2 implies that op1 precedes op2 in that core's program order. Under SC, memory order respects each core's program order. "Respects" means that op1 <p op2 implies op1 <m op2. The values in comments (/* ... */) give the value loaded or stored. This execution terminates with r2 being NEW. More generally, all executions of Table 3.1's program terminate with r2 as NEW. This example illustrates the value of SC.

L(a) and S(a) represent a load and a store, respectively, to address a. Orders <p and <m define the program and global memory order, respectively. Program order <p is a per-core total order that captures the order in which each core logically (sequentially) executes memory operations. Global memory order <m is a total order on the memory operations of all cores.
An SC execution requires:

(1) All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or a≠b). There are four cases:
If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */
(2) Every load gets its value from the last store before it (in global memory order) to the same address: Value of L(a) = Value of MAX <m {S(a) | S(a) <m L(a)}, where MAX <m denotes "latest in memory order."

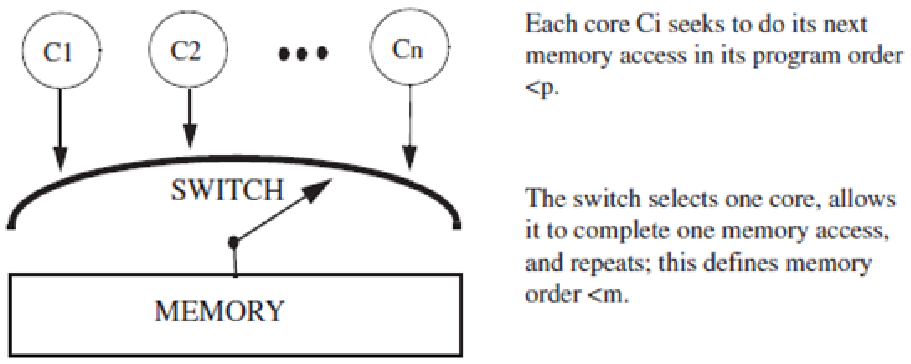

C1
C2
•••
Cn
Each core Ci seeks to do its next memory access in its program order <p.
SWITCH
The switch selects one core, allows it to complete one memory access, and repeats; this defines memory order <m.
MEMORY

FIGURE 3.2: A simple SC implementation using a memory switch

You can implement SC with a set of cores Ci, a single switch, and memory, as depicted in Figure 3.2. Assume that each core presents memory operations to the switch one at a time in its program order. Each core can use any optimizations that do not affect the order in which it presents memory operations to the switch. For example, a simple 5-stage in-order pipeline with branch prediction can be used.

Assume next that the switch picks one core, allows memory to satisfy the load or store fully, and repeats this process as long as requests exist. The switch may pick cores by any method (e.g.,random) that does not starve a core with a ready request. This implementation operationally implements SC by construction.

- Load → Load
- Load → Store
- Store → Store
- Store → Load /* Included for SC but omitted for TSO */

Total Store Order(TSO) includes the first three constraints but not the fourth. Detail definition of TSO can be found in [1] chapter 4.

OBJECTIVE:

1. Your project is to demonstrate the list of reordering mentioned above using a multithread operation (using 2 thread) in any programming language like (C , C++ etc) using a memory litmus test. You must design a separate memory litmus test for each reordering case.
2. You must draw the execution of the litmus tests.
3. Using proper synchronization primitive (e.g. lock, semaphores), you must demonstrate the implementation of sequential consistency and total store order(TSO)

Mid-Project Evaluation (MPE):

1. Demonstration of the reordering using memory litmus test:
   a. Store-store reordering.
   b. Load-load reordering
   c. Load-store and store load reordering.
2. Draw the execution of the litmus tests provide by you.

End-Project Evaluation (EPE):

Please ensure EPE completes the following:

1. Whatever you cannot implement in MPE has to be completed before EPE.
2. Implementation of naïve SC models to avoid reordering in the three litmus test.

3. Implementation of Total Store Order (TSO) memory model

In summary, this project aims to demonstrate memory ordering behaviors in a shared multicore environment practically. By designing memory litmus tests and implementing sequential consistency and TSO models, you will gain insight into memory consistency challenges and solutions. Your mid-project and end-project evaluations will assess the completeness and effectiveness of your demonstrations and implementations.

Further detail of memory consistency can be found in:

[1]A Primer on Memory Consistency and Cache Coherence, Second Edition

Project 4: Simulation of Directory-based Cache Coherence: Programming Project

About Coherency:

Coherency refers to the consistency of data shared between different cores or processing units in a multi-core processor. In a multi-core system, each core has its own local cache, which stores copies of data from the main memory. In a single program, different threads might run on separate cores to perform parallel tasks. These threads could be working on different aspects of the same problem and likely share the same data. When multiple cores must work with the same data, coherency mechanisms ensure that all cores see a consistent view of that data.

About Directories:

A Directory is just one entity that is commonly used in Network architectures to maintain data coherency between the cores. There are many ways of sizing and organizing the directory, and we consider a model where there is a directory entry for every block of the memory.

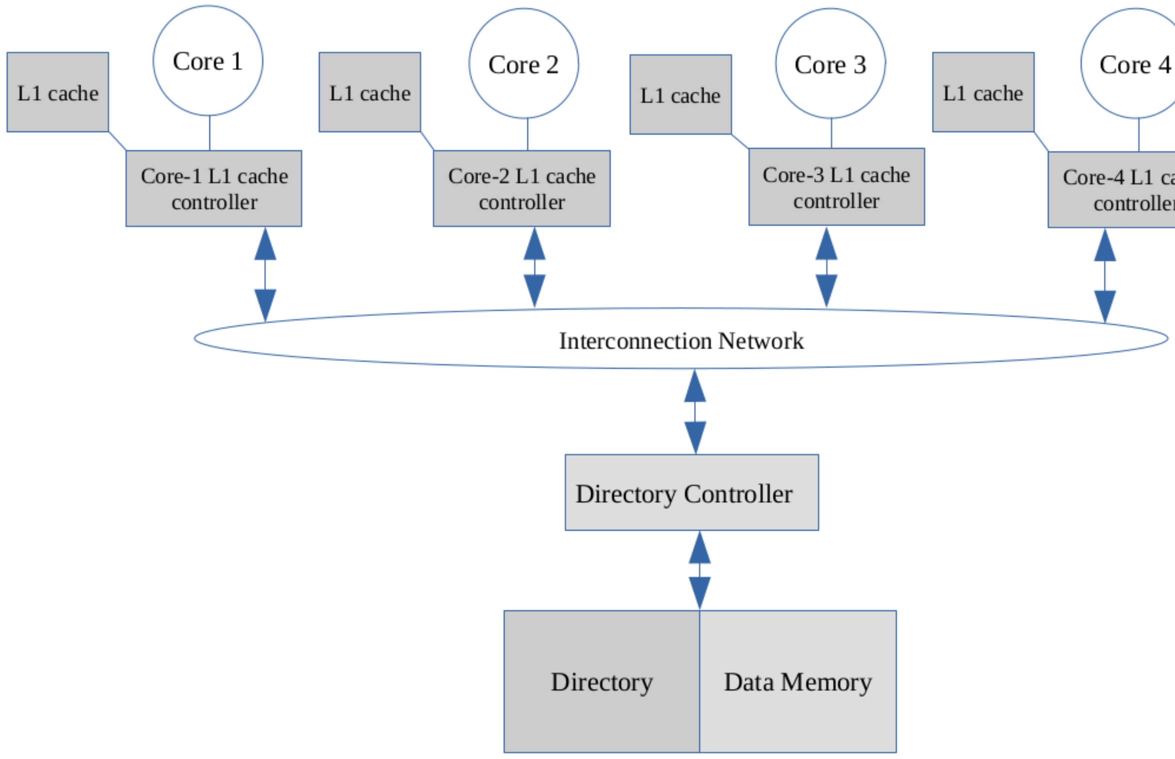


Fig. 1. The directory architecture to be Implemented

The directory will be maintained with the help of an entry for each memory location, as follows:

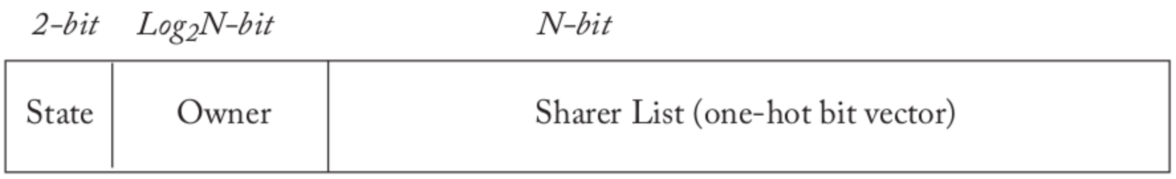


Fig. 2. Format for a directory entry (MSB to LSB going left to right)

Coherency Protocol:

A protocol is usually followed to effectively maintain data coherency. This is facilitated with the help of *transactions* between the cores and the directory over the interconnect. For the purpose of this project, you do not need to implement the interconnect network, but you can assume it is in place when making *requests, transactions, and responses* (explained later). The coherency protocol to be followed is *MOSI* (Note: subtle differences exist in the state implementations for MSI, MOSI, and other protocols). Each state in MOSI has been described below for clarity:

State symbol	State binary	State	Description
M	00	Modified	Data is up-to-date, can be read or written, and is exclusively owned by the core
S	01	Shared	An up-to-date read-only copy of data, shared by at least 2 cores (One of the cores can also be the owner)
I	10	Invalidated	A core having a copy of an address in state M invalidates the same address for other cores
O	11	Owned	Every cached directory entry has an owning core, which has the latest, read-only copy of the address data

Project Description:

Every core is identical and follows the same set of instructions to communicate with the cache controller to fetch data from the memory - into the cache. Following are the instructions, their description, and the reachable state(s) because of their behavior:

I-no	Instruction semantic	Instruction Description	Facilitates state
1	<Core> LS <address>	Fetches read-only address location from memory to L1 cache	O / S (only S for MSI protocol)
2	<Core> LM <address>	Fetches address location with read-write access - from memory to L1 cache	M / I
3	<Core> IN <address>	Invalidates address with the core	I
4	<Core> ADD [address] #immediate	Adds immediate value to the data stored in the memory location and overwrites the result into the same location	M / I

I.e., Each core can issue load requests to its cache controller; the cache controller will choose a block to evict when it needs to make room for another block.

The Cache and Memory Organization:

1. The main-memory contains 64 address locations, and each L1 cache has 4 address locations.
2. Initialize the memory array with zeros at all locations.
3. Cache line width = size of memory location = 1 byte (byte-addressable)
4. The L1 caches must be *2 way set-associative*, and the L1 controller follows the *LRU (Least-Recently Used) replacement policy*.
5. The L1 cache controller also follows the *write-through policy* in the same cycle the data in a particular address location is modified.

Requests, Transactions, and Responses:

Transactions between the cache controller and the directory controller are necessary to provide the cache read/read-write access to a particular block in one of the mentioned coherence states (M, O, S, or I).
The request for these transactions is made by the core and passed on to its cache controller, which initiates the appropriate transaction and gets a response from the destination. [2]
Acknowledgment from the destination as a response is omitted for the purpose of this project since every request does get fulfilled in this simulation.
Valid Transaction for each of the mentioned instructions:

Transaction	Instructions that initiate the transaction	Description
GetShared	LS	1. Checks whether address is owned and assigns ownership if address was unassigned. It provides read access to the core for a particular address. 2. In case of owned address - The latest copy is also provided by the owner as a *response* to a core's *request*
GetModified	LM, ADD	Provides read-write access and ownership of the address along with the latest copy from the owner - as a *response*
Put	IN, LM, ADD	The local copy is invalidated, and the data block is *evicted* from the cache to the main memory in the following cycle

Mid-Project Deliverables:

1) The simulator must be able to read and interpret the instructions file
2) Implement directory-based coherence for a dual-core system with MSI protocol (with the earlier mentioned cache and memory specifications, but memory should have 32 address locations, and the caches are *fully associative* following the *random replacement policy*)
3) The architecture should be represented accurately (as displayed in Fig1, but for a dual-core system) with all its entities and their functionality defined completely.
4) The interpreted instructions should generate the right requests, transactions, responses, and Directory updates.
5) Generate a log file with:
   (i) The cache memory dump after executing each instruction of the program
6) The final state of the directory entries and the corresponding memory dump should be printed after processing the entire program.
7) Make test cases to exhaustively test and prove the working of your architecture

End-Project Deliverables (in addition to points 3 - 7 of MPD) :

1) The architecture should be extended to a quad-core system with the MOSI protocol and should still be able to read and interpret the instructions file
2) The memory and cache specifications are mentioned on the previous page
3) Make the following Plots after executing the program:
   (i) Plot the average memory access latency and miss rates for each of the cores after processing the program. (assume miss penalty = 2 * hit time, time unit = clock cycle, and it takes 1 clock cycle to execute 1 instruction)
   (ii) Plot the number of directory updates with time (1 clock cycle to execute 1 instruction)
4) Generate another log file:
   (ii) Core Request, generated Transaction(s), and their Response(s) should be written in human-readable format - into a log file for each instruction of the program.
5) A README file should contain the following details:
   a) The working and usage of your simulator, along with the instruction semantics you followed.
   b) The building instructions (if any).
   c) The usage instructions use the command line interface.

Reference Material:

1. Chapters 6 (detailed definition of each state) and 8 (directory based implementation of the MSI/ MOSI protcol), A Primer on Memory Consistency and Cache Coherence
2. CMU Lecture on Directory-based coherence

Project 5 - RTL Design of 5-stage pipelined RISC-V processor in RTL

References:

1. https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html#ebreak
2. https://riscv.org/technical/specifications/
3. https://www.cs.unh.edu/~pjh/courses/cs520/15spr/riscv-rv32i-instructions.pdf

Plagiarism Policy

1. The project is to evaluate your own implementation of the project goals.
2. Your code WILL be crosschecked with any online codes/material as well as any submissions from other courses. Any reasonable similarity found will be subjected to the Academic Dishonesty Policy.

Project Description:

1. You have to do this project in any HDL(e.g. Verilog, VHDL, etc.) of your choice. In this project, you have to implement a processor based on the RV32I variant of RISCV ISA *(description at the end of this page).
2. The microarchitecture should be a 5-stage pipeline viz. Fetch, decode, execute, memory and writeback.
3. Single Clock should be used for the whole design.
4. You have to use separate memory for instruction and data memory.
5. Microarchitecture should also include forwarding/bypassing.
6. You also have to include a separate execution unit for NOC operations.
7. You have to include a 2-bit branch predictor as shown in the below fig.

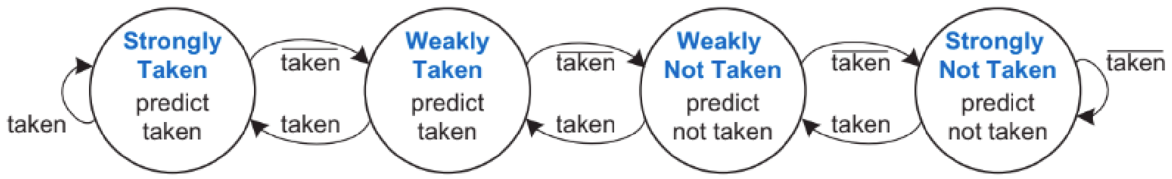


Figure 7.67 **Two-bit branch predictor state transition diagram**

*** The project should support the entire RISC-V instruction set listed here [Excluding instructions from FENCE to RDINSTRETH]:**
https://www.cs.unh.edu/~pjh/courses/cs520/15spr/riscv-rv32i-instructions.pdf

Peripheral Support

In addition to the above instructions; you also have to implement two special instructions that are not present in the standard RISC-V ISA. These instructions are:

1. LOADNOC:

Syntax: LOADNOC <rs2> <rs1> <imm>

This instruction will store the data in the register rs2 to special memory mapped registers whose address will be (rs1+imm). In theory, this instruction is very similar to the “Store” instruction of RISC-V. However, a regular store would write data to the Data Cache while this instruction will write the data to memory-mapped registers. Note that the registers in the register file are not the same as memory-mapped registers. Memory Mapped Registers are not present in the register file. In standard RISC-V, Memory Mapped Registers are written by the same load/store instructions that are used to write to the data cache. Whether to actually write to the data cache or to the memory mapped registers is determined by the address of the load-store. For the sake of simplicity, you need to assume that addresses in the range 0x0000 - 0x3fff will be loading/storing from the data cache and addresses from 0x4000 will be loading/storing from memory mapped registers. For the purpose of this project, there are only 4 memory mapped registers of 32 bits. Let us call these registers MMR0, MMR1, MMR2, MMR3, and the mapping is:

Register	MMR0	MMR1	MMR2	MMR3
Address	0x4000	0x4004	0x4008	0x400c

Eg. to store a 32-bit integer 0x33293745 into MMR2; you would first need to move this integer and 0x4000 into a Register File Registers(let’s say R1 and R2) respectively, then use the instruction LOADNOC R1, R2, #8.

1. SENDNOC:

Syntax: STORENOC

This instruction is similar to LOADNOC, but it will always write the integer 1 to the Memory Mapped Register with address 0x4010(MMR4). Note that this instruction does not take any argument. So you need to hardcode that when this instruction is issued, the memory mapped register with address 0x4010 will be written with the value 1.

For clarity you can refer to the memory map given below:

Region	Start Address	End Address
D-Cache	0x0000	0x3fff
MMR0	0x4000	0x4003
MMR1	0x4004	0x4007
MMR2	0x4008	0x400b
MMR3	0x400c	0x400f
MMR4	0x4010	0x4013

For simplicity, you can assume that any address beyond 0x4014 will never be accessed.

Testing Infrastructure

1. Write a test bench to verify the functionality of your implemented processor. The testbench should take the path of the binary as an input.
2. Dump the register file after executing the input binary.
3. Also, plot the waveforms of all the intermediate control and datapath signals on Vivado.
4. The dump of the test binary should match the actual simulation results.

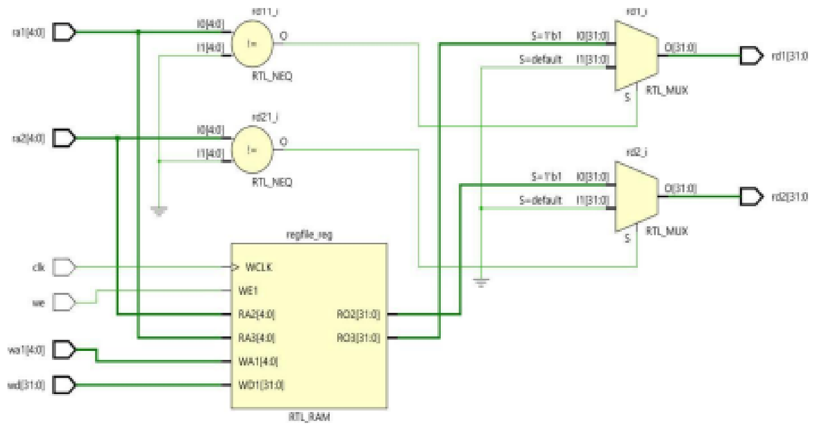
Mid-Evaluation Deliverables

1. Implement all the instructions mentioned in the table. You will NOT be evaluated based on forwarding/bypassing logic.
2. Create a test binary incorporating at least one instance of all the instructions you have implemented. You should provide an assembly code for your binary. If your assembly code is too simple(does not incorporate all the functionality your processor can offer), then marks will be deducted accordingly.
3. Do not hardcode your output of the binary. It must be generated by your processor implementation.
4. Working Testing infrastructure based on the requirements mentioned above.
5. Explain your project's working and include the RTL schematic (as shown in Figure 1) of all your implemented modules (Eg. ALU, register File, etc.) in a presentation.

Final Evaluation Deliverables

1. Implement all the functionalities as described in the project description.
2. Prepare a presentation describing the project, your implementation and results etc.
3. Upload your final code to a GitHub repository.

Example of RTL Schematic for Register File.



Project 6 - Cycle accurate simulator for 5-stage CPU

Project Description

In this project, you will create a cycle-accurate simulator of a 5-stage RISC CPU defined in Project 5 in any Programming language of your choice. We can help you if you use Python3, C++ or Java. But if you use some other language, we would not be able to provide you with support. The simulator should replicate the setup as stated in Project-5, with some additional features.

1. Implement an Assembler that converts RISC-V ISA to binary. The Assembler should implement the entire RISC-V instruction set listed here [Excluding instructions from FENCE to RDINSTRETH]: https://www.cs.unh.edu/~pjh/courses/cs520/15spr/riscv-rv32i-instructions.pdf
2. The further processing should be done from the binary obtained from the above assembler. [Assume there are no syntax errors and there will be a valid binary for each test case].
3. Implement a 2-way set associative cache which has 8 blocks which follows the LRU replacement policy. Each block can store 32 bits of information.

Your simulator should take a text file as input that contains the ISA stated in Project-5. And provide the following points:

1) Create a log file that contains, per clock state of CPU (state includes the value in the register file, instruction in each pipeline stage, is the CPU stalled) + memory state at the end of the program.
2) Log the cache state after each cycle. Also maintain data for cache hit cases. [Up To you whether you want it in the same file or different file.]

Using the log file and the input file now, you can generate the following graphs:

- Divide the provided instructions into groups of Register instructions or memory instructions and plot the number of register instructions and memory instructions in a given text file
- Plot the memory access pattern of the CPU (1 plot for all the instructions + 1 plot for data access pattern)
- Plot the number of data stalls in the program

In summary, you need to the following things for **final evaluation**

1) The **test file** should contain all the ISA instructions.
2) A **simulator** program that will take the test file and simulate the CPU to generate a log file with details about CPU, Memory state and stalls
3) A **graph plotter** that will use the test file and the generated log to create plots for memory access (instruction and data access), types of instructions and data stalls in the program
4) **README** to give any necessary details about how to use the **simulator** and **graph plotter**

Mid Sem Evaluation

1. Test binary that includes all the provided instructions in ISA. [Binary should be generated from your assembler code].
2. Working simulator that creates the log file
3. Simulator should be able to parse all different binary instructions in the ISA
4. Simulator should at least generate the partial CPU state (value of register file per clock cycle), cycle number and ending memory state.

Bonus: Implementation of Single Instruction Multiple data (SIMD) Instruction

About the instruction: SIMD instructions are used when we need to process multiple data with a single instruction. In this part, you need to implement the SIMD execution. You are free to define the instruction and the corresponding microarchitecture. It is recommended to keep the proposed SIMD instruction to be of the same size as of standard RV3 extension.

Working: When the SIMD instruction is decoded, the control is transferred to a dedicated hardware unit responsible for handling SIMD instruction. The core does XOR operation on a ‘n’ sized array with each element being 32 bit in size. While this operation is being executed, the processor is stalled with the SIMD instruction at execute phase.