Project 1 - RTL Design of NoC Router and creation of 3x3 mesh architecture

About NoC

Network On Chip is an interconnect designed to connect different cores in a multicore system on chip (SoC). Figure 1 shows a 2X2 mesh architecture that is 2 routers are present in one row, and there are two such rows. PE (Processing Elements) refers to the IP (intellectual property) in a multicore system, which can be CPU, GPU, memory or any other IP. A router port can be connected to another router or a processing element. In a mesh design, a typical 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, the northern ports of routers A and B are not connected to anything.

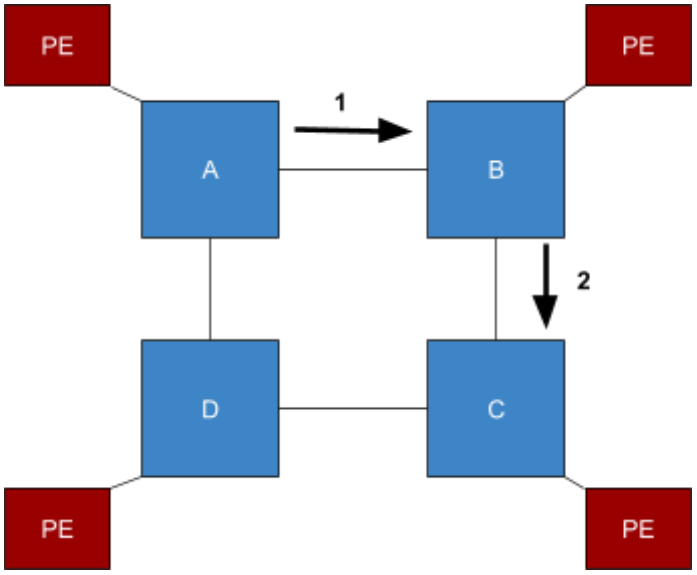


Fig1: A 2X2 mesh NoC Architecture with Processing Elements

Now, let us consider that router A has to send data to router C. Two possible paths include ABC and ADC. To remove ambiguity in the path, **routing algorithms** are used. XY routing is one such algorithm that prioritizes horizontal movement over vertical 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 the figure 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 unit of transfer in an NoC. However, the packets cannot be transferred in a single go. To aid transmission, the packet is broken down into flits. The flits can be typically transferred over a single go, and hence the packet is transmitted as a series of flits. Flits are of three types - header, data and tail. The header flit contains the control information (source, destination etc.) and is responsible for creating the path for the network. The data flits contain the data to be transferred, and the tail flits indicate the end of the packet.

The following figure shows a simple NoC router. The router has one input and one output port in each direction. The crossbar (XBAR) connects every input to every output of an NoC. The switch allocator configures the crossbar according to the header flit and routing algorithm. For instance, in the above figure, 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.

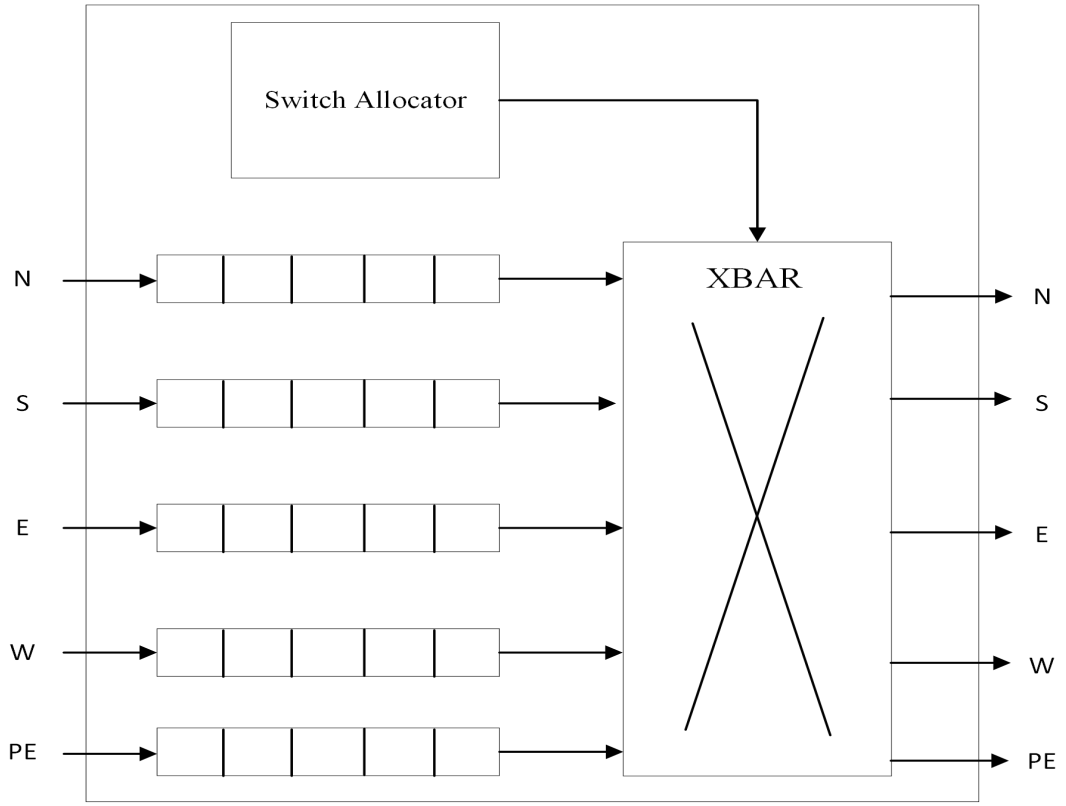
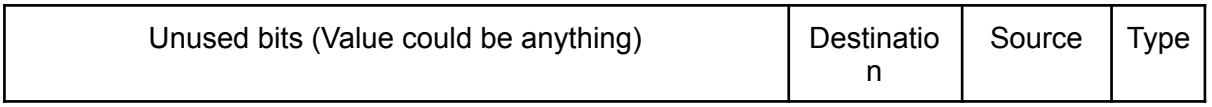


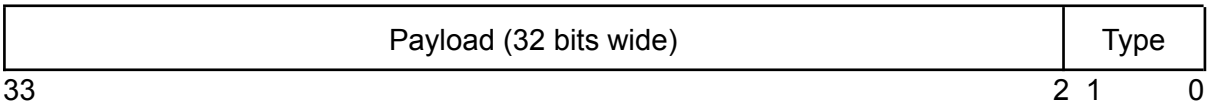
Fig 2: NoC Microarchitecture

Project Description:

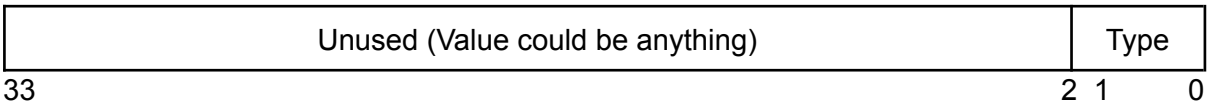
1. You have to do the project in any HDL (Verilog or VHDL) of your choice.
2. The packet consists of a header, payload and tail flit. The header contains control information and is 34 bits. The payload consists of 3 flits of 34 bits each and the tail of 34 bits. The flit size is 34 bits.
3. The router consists of 5-entry buffers for each input of an NoC. Each entry is 34 bits. A mesh configuration of 3x3 such routers needs to be designed with the protocol described below.
4. Your routers need to support XY routing. Moreover, for this project, you may assume that there will be no congestion issues.
5. The two LSBs of each flit describe the type of flit. For the head flit, the type is 00; for body flits, the type is 01; and for the tail flit, the type is 11.
6. The bits of the flits are as follows:
    a. Head Flit (Flit type = 00):



b. Body Flits:



c. Tail Flit:



NoC Communication Protocol:

The following table lists the signals used in the protocol.

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. A transfer takes place when both **VALID** and **READY** are asserted. Deasserted when tail flit is transferred.
READY	Slave	**READY** indicates that the slave can accept a transfer in the current cycle. Deasserted when the input buffer is full.
DATA[31:0]	Master	**TDATA** is the primary data passing across the interface

The protocol is given in the following: [EECS150: Interfaces: “FIFO” (a.k.a. Ready/Valid)]

Mid-term Deliverables:

1. Completed design of an NoC router with the described protocol
2. To test your design, add header flits to the buffers of an NoC. Depending on the packet, the switch allocator should configure the crossbar to allow packet movement. All possible packet movement should be considered by the testbench.

End-term Deliverable:

1. Completed 3x3 mesh design
2. Upload your final code to a GitHub repository.
3. To test your design, include at least three combinations of data movements from one router to another. The test scheme is given below:
    a. The local port of one router should be protocol compliant and be placed as a port to the design.
    b. Send 3 packets via test bench such that the three combinations of XY routing (along the X axis only, along the Y axis only and both) are tested.

Project 2- 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/

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 subset of instructions from the RV32I variant of RISCV ISA as mentioned below in the table.
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.

Instructions from RV32I (Understand semantics from the RISCV Manual)

AND	AND operation
OR	Bitwise OR operation
ADD	ADD
SUB	SUB
ADDI	Add Immediate
BEQ	Branch equal
LW	Load Word
SW	Store Word
SLL	Shift Logical Left
SRA	Shift Right Arithmetic

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.

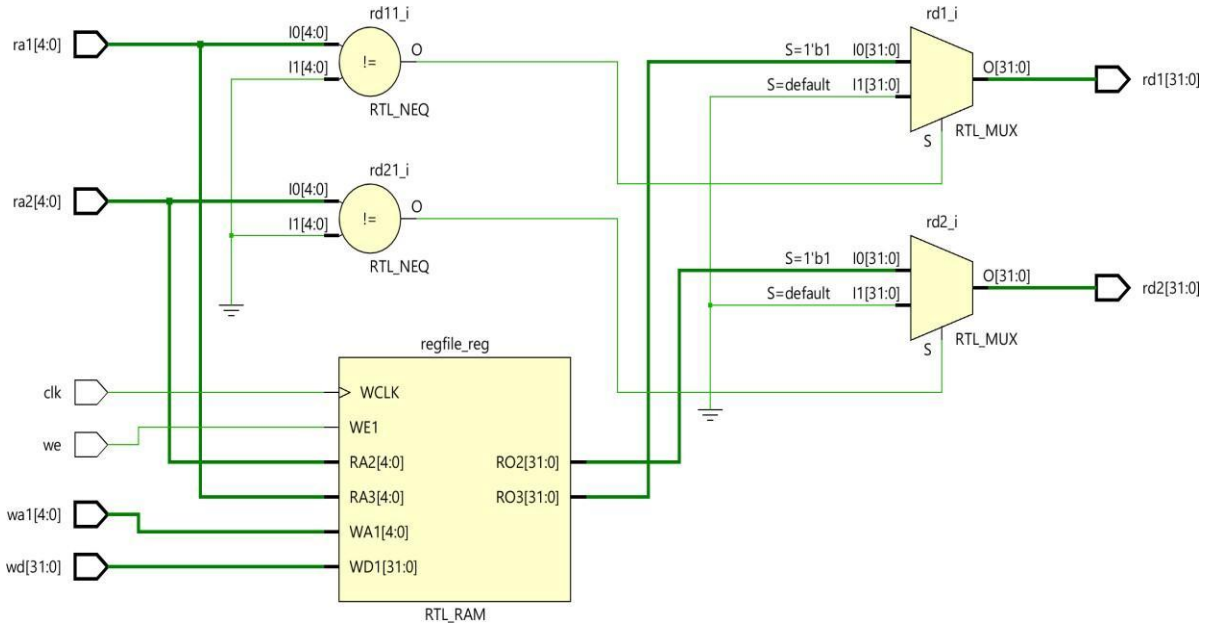


Figure 1

RISC-V REFERENCE CARD HAS BEEN GIVEN BELOW FOR YOUR REFERENCE

Free & Open RISC-V Reference Card

①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category / Name	Fmt	RV32I Base	+RV{64,128}
Loads Load Byte	I	LB rd,rs1,imm	
Load Halfword	I	LH rd,rs1,imm	
Load Word	I	LW rd,rs1,imm	L{D|Q} rd,rs1,imm
Load Byte Unsigned	I	LBU rd,rs1,imm	
Load Half Unsigned	I	LHU rd,rs1,imm	L{W|D}U rd,rs1,imm
Stores Store Byte	S	SB rs1,rs2,imm	
Store Halfword	S	SH rs1,rs2,imm	
Store Word	S	SW rs1,rs2,imm	S{D|Q} rs1,rs2,imm
Shifts Shift Left	R	SLL rd,rs1,rs2	SLL{W|D} rd,rs1,rs2
Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W|D} rd,rs1,shamt
Shift Right	R	SRL rd,rs1,rs2	SRL{W|D} rd,rs1,rs2
Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W|D} rd,rs1,shamt
Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W|D} rd,rs1,rs2
Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W|D} rd,rs1,shamt
Arithmetic ADD	R	ADD rd,rs1,rs2	ADD{W|D} rd,rs1,rs2
ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W|D} rd,rs1,imm
SUBtract	R	SUB rd,rs1,rs2	SUB{W|D} rd,rs1,rs2
Load Upper Imm	U	LUI rd,imm	
Add Upper Imm to PC	U	AUIPC rd,imm	
Logical XOR	R	XOR rd,rs1,rs2	
XOR Immediate	I	XORI rd,rs1,imm	
OR	R	OR rd,rs1,rs2	
OR Immediate	I	ORI rd,rs1,imm	
AND	R	AND rd,rs1,rs2	
AND Immediate	I	ANDI rd,rs1,imm	
Compare Set <	R	SLT rd,rs1,rs2	
Set < Immediate	I	SLTI rd,rs1,imm	
Set < Unsigned	R	SLTU rd,rs1,rs2	
Set < Imm Unsigned	I	SLTIU rd,rs1,imm	
Branches Branch =	SB	BEQ rs1,rs2,imm	
Branch ≠	SB	BNE rs1,rs2,imm	
Branch <	SB	BLT rs1,rs2,imm	
Branch ≥	SB	BGE rs1,rs2,imm	
Branch < Unsigned	SB	BLTU rs1,rs2,imm	
Branch ≥ Unsigned	SB	BGEU rs1,rs2,imm	
Jump & Link J&L	UJ	JAL rd,imm	
Jump & Link Register	UJ	JALR rd,rs1,imm	
Synch Synch thread	I	FENCE	
Synch Instr & Data	I	FENCE.I	
System System CALL	I	SCALL	
System BREAK	I	SBREAK	
Counters ReaD CYCLE	I	RDCYCLE rd	
ReaD CYCLE upper Half	I	RDCYCLEH rd	
ReaD TIME	I	RDTIME rd	

RV Privileged Instructions

Category / Name	RV mnemonic
CSR Access Atomic R/W	CSRRW rd,csr,rs1
Atomic Read & Set Bit	CSRRS rd,csr,rs1
Atomic Read & Clear Bit	CSRRC rd,csr,rs1
Atomic R/W Imm	CSRRWI rd,csr,imm
Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
Atomic Read & Clear Bit Imm	CSRRCI rd,csr,imm
Change Level Env. Call	ECALL
Environment Breakpoint	EBREAK
Environment Return	ERET
Trap Redirect to Superviso	MRTS
Redirect Trap to Hypervisor	MRTH
Hypervisor Trap to Supervisor	HRTS
Interrupt Wait for Interrupt	WFI
MMU Supervisor FENCE	SFENCE.VM rs1

Optional Compressed (16-bit) Instruction Extension: RVC

Category / Name	Fmt	RVC	RVI equivalent
Loads Load Word	CL	C.LW rd′,rs1′,imm	LW rd′,rs1′,imm*4
Load Word SP	CI	C.LWSP rd,imm	LW rd,sp,imm*4
Load Double	CL	C.LD rd′,rs1′,imm	LD rd′,rs1′,imm*8
Load Double SP	CI	C.LDSP rd,imm	LD rd,sp,imm*8
Load Quad	CL	C.LQ rd′,rs1′,imm	LQ rd′,rs1′,imm*16
Load Quad SP	CI	C.LQSP rd,imm	LQ rd,sp,imm*16
Stores Store Word	CS	C.SW rs1′,rs2′,imm	SW rs1′,rs2′,imm*4
Store Word SP	CSS	C.SWSP rs2,imm	SW rs2,sp,imm*4
Store Double	CS	C.SD rs1′,rs2′,imm	SD rs1′,rs2′,imm*8
Store Double SP	CSS	C.SDSP rs2,imm	SD rs2,sp,imm*8
Store Quad	CS	C.SQ rs1′,rs2′,imm	SQ rs1′,rs2′,imm*16
Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
ADD Word	CI	C.ADDW rd,rs1	ADDW rd,rd,imm
ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
ADD SP Imm * 4	CIW	C.ADDI4SPN rd′,imm	ADDI rd′,sp,imm*4
Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches Branch=0	CB	C.BEQZ rs1′,imm	BEQ rs1′,x0,imm
Branch≠0	CB	C.BNEZ rs1′,imm	BNE rs1′,x0,imm

ReaD TIME upper Half	I	RDTIMEH rd
ReaD INSTR RETired	I	RDINSTRET rd
ReaD INSTR upper Half		RDINSTRETH rd

Jump Jump Jump Register	CJ CR	C.J imm C.JR rd,rs1	JAL x0,imm JALR x0,rs1,0
Jump & Link J&L Jump & Link Register	CJ CR	C.JAL imm C.JALR rs1	JAL ra,imm JALR ra,rs1,0
System Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats
Instruction Formats

CR

	31	30	25	24	21	20	19	15	14	12	11	8	7	6	0
R	funct7			rs2			rs1		funct3		rd			opcode	
I	imm[11:0]						rs1		funct3		rd			opcode	
S	imm[11:5]			rs2			rs1		funct3		imm[4:0]			opcode	
SB	imm[12]	imm[10:5]		rs2			rs1		funct3		imm[4:1]	imm[11]		opcode	
U	imm[31:12]										rd			opcode	
UJ	imm[20]	imm[10:1]			imm[11]		imm[19:12]				rd			opcode	

16-bit (RVC)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2
C	funct4				rd/rs1					rs2				
I	funct3	imm			rd/rs1					imm				
CSS	funct3	imm								rs2				
CIW	funct3	imm										rd′		
CL	funct3	imm			rs1′					imm		rd′		
CS	funct3	imm			rs1′					imm		rs2′		
CB	funct3	offset			rs1′					offset				
	funct3	jump target												

Project 3 - Cycle accurate simulator for an NoC router and mesh

A NoC router is the basic building block of an NoC. It is responsible for routing the packets injected into the NoC by the components connected to the NoC (PEs). Please refer to the project description for Project 1 to get an idea of how an NoC works.

In this project, you are required to create an NoC simulator in a language of your choice. We can help you if you use python3, C, C++ or java. But if you use some other language, then we would not be able to provide you with support.

You need to simulate an NoC router and a 2x2 NoC mesh. As described above, each router needs to have 5 ports (north, south, east, west and local) to connect to its neighbours and to connect to its local processing element. Some ports of the routers may be left unconnected if there is no neighbour (for example, the north port of router B in Fig 1).

Your router must support XY routing as described in project 1. 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.

Your simulator needs to support reading a traffic file which describes which packets are inserted in the NoC at various clock cycles. The traffic file is a text file with four values mentioned in each line, separated by spaces. The first value describes the clock cycle at which a packet is inserted into the NoC. The second and third values represent the source and the destination for the packet, respectively. The fourth value is a 32*3 = 96-bit value which represents the payload of the packet.

You may decide the transfer latency for your router and the connection between the routers, but in each cycle, no more than a single flit must be transferred.

You do not have to simulate the NoC communication protocol in detail. You may implement the connection as an array in which the sender writes the data to the receiver’s array.

Your simulator must be cycle accurate, which means that your simulator must be able to simulate clock cycles. The data transfer and the state update for your routers must functionally happen at the positive clock edges.

Each router must have a crossbar, a switch allocator and I/O ports. You can find the details of these sub-components in the description of Project-1. These sub-components must be visible in the code.

Your simulator must generate a log file which indicates what flit is received at each cycle and at what router so that we can trace the packet flow. You must also indicate the contents of the flit received. Moreover, you must also indicate the insertion of packets into the NoC.

The input to the simulator would be the traffic file, the routing algorithm (XY or YX) and the simulator's output would be the log file.

You also need to write another program that reads this log file and generates the following two graphs:

1. A graph which plots the number of flits sent over a connection. A connection is defined as the 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 in the form of a bar graph.
2. 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.

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

1. A cycle-accurate NoC simulator which:
    a. Supports XY and YX routing
    b. Supports reading the traffic information for a text file
    c. Generates a log file which describes what is happening at each clock cycle.
2. A program which interprets the log file generated by the simulator and plots the following graphs:
    a. Number of flits transferred as a function of connections.
    b. Latency as a function of packets sent.
3. A README file which describes:
    a. The working and usage of your simulator.
    b. The usage of your graph plotter.
    c. The building instructions (if any).
    d. The usage instructions.

For the mid-evaluation of the project, you must have the following:

1. A working simulator which is able to generate the log file.
2. The log file should at least include the cycle count and the flits received in that cycle.
3. Your simulator must also support at least one of the routing algorithms
4. Your simulator must be able to read and interpret the traffic file.
5. Your simulator must also be able to inject packets as per the traffic file.

Project 4 - Cycle accurate simulator for 5-stage CPU

In this project, you will create a cycle-accurate simulator of a 5-stage RISC CPU defined in the above Project in any Programming language of your choice.
We can help you if you use python3, C, 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-2.

Implementation details

You can create the following modules/classes in the project

1) **CPU class** (Possible Inputs:- clock, data_read) (Possible outputs:- data_write,addr_read,addr_write)
   Possible inputs and outputs definitions
   Clock:- Global clock for the system
   Data_read:- is the data sent from the cache to CPU
   Data_write:- is the data sent out to the cache from CPU
   Addr_read:- the address from which CPU needs to read the data
   Addr_write:- the address on which CPU needs to write the data

   It contains the logic for the pipeline stages (taken from Project-2) and the register file that would be used by the pipeline stages.
   Pipeline stage and RegisterFile can also be modelled as a class.
   All the stages should closely follow the spec defined for Project-2

2) **Data Memory class** (Possible Inputs:- clock, addr,data_write) (Possible outputs:- data)
   Clock:- Global clock for the system
   Data_write:- is the data sent out to the cache from CPU
   Addr:- the address on which data is read or written to
   isRead:- is the data written to memory or sent to CPU

   It simulates the data memory element. The memory element returns the response to the given request from the CPU(read/write) after **x(delay)** clock cycles the request was submitted. **When memory is instantiated user should be able to set the value of delay**

3) **Instruction Memory class** (Possible Inputs:- clock, addr) (Possible outputs:- data)
   Clock:- Global clock for the system
   Addr:- the address from which instruction is read
   Data:- the instruction data which is sent to CPU
   It simulates the data memory element. The memory element returns the response to the given request from CPU(read) after **x(delay)** clock cycles the request was submitted. **When memory is instantiated, the user should be able to set the value of delay.**

Your simulator should take a text file as input that contains binary for the ISA stated in Project-2. 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.

   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.
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.