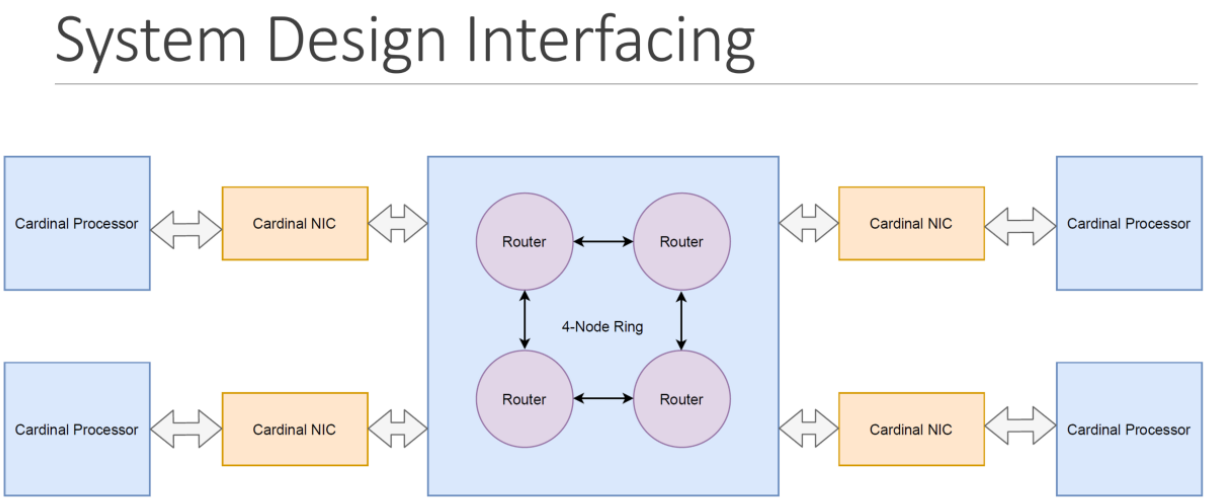
Ahmed Abuhjar
EE-577B Final Report

Cardinal NIC and Chip Multiprocessor

Top-Level Design:

I first broke the system into bunch of blocks from which I could then compose a router, an arbiter and other high-level blocks like the Ring topology NoC, the CPU, and the NIC (network interface component).

So the Cardinal Ring is composed of 4 routers and it's interfacing with the 4 CPUs using the Network Interface Components.

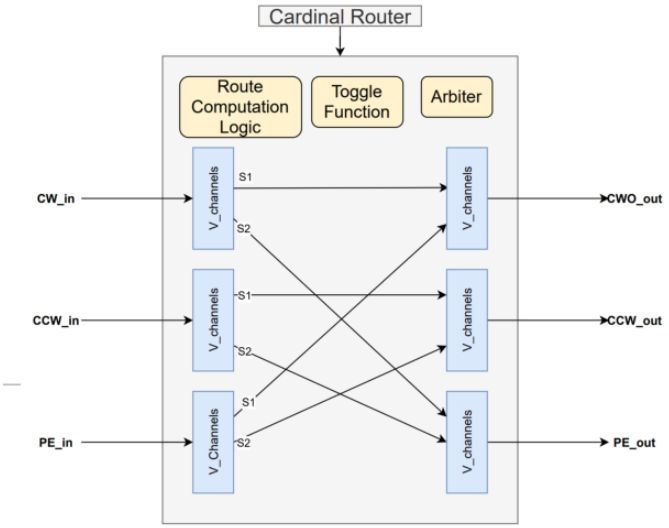


The design was done according to specifications. For making sure the proper working of NIC I tested it for various cases. The test cases covers testing for writing after checking status (from Router and Processor), reading from NIC Input/ Output Channels, simultaneous reading and writing operation of Input and Output Channel Buffers (From Router and Processor), Concurrent reading and writing from router.

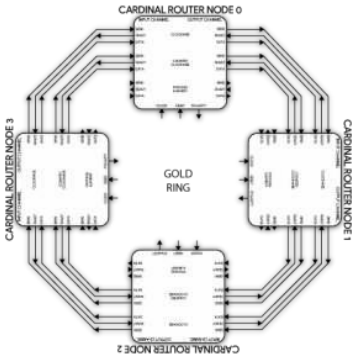
Router and Ring (cardinal_router.v and cardinal_ring.v):

As shown in the figure below, at a high level, the router contains the following components. It consists of input ports and output ports. For each port it will have two virtual channels. so It will have 6 buffers in the input channels and 6 buffers in the output channels for handling scenarios of contentions such as when you have multiple inputs sending to a single output. So the buffering allows these contention scenarios to be handled with the addition of toggle function to generate the polarity that I will go over shortly in this presentation. There's arbitration logic that basically allocates the buffering inside the router from multiple input buffers to one output buffer when contention occurs. There is also some logic to do route computations which I will also go over briefly in the next few slides. So basically in order to build such a router, I first implemented these building blocks from which I then composed the high-level block of the router. These building blocks include the buffers and virtual channels, the Route computation logic, toggle function for polarity, and the arbiter.

Ahmed Abuhjar
EE-577B Final Report



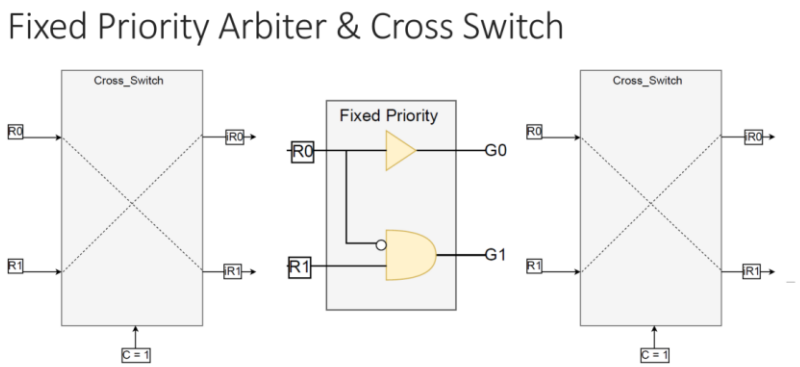
Ring Design:



Ring design was just instantiation of cardinal router 4 times to make a ring. I faced minor error in connection of ring while integrating with the processor. It was fixed by the diagram on pdf on phase3 where I was mapping each node. Testbench for ring was done using a Gather Test Bench approach (as required) and keeping in mind multiple packet injections to test deadlock.

Arbitration Logic:

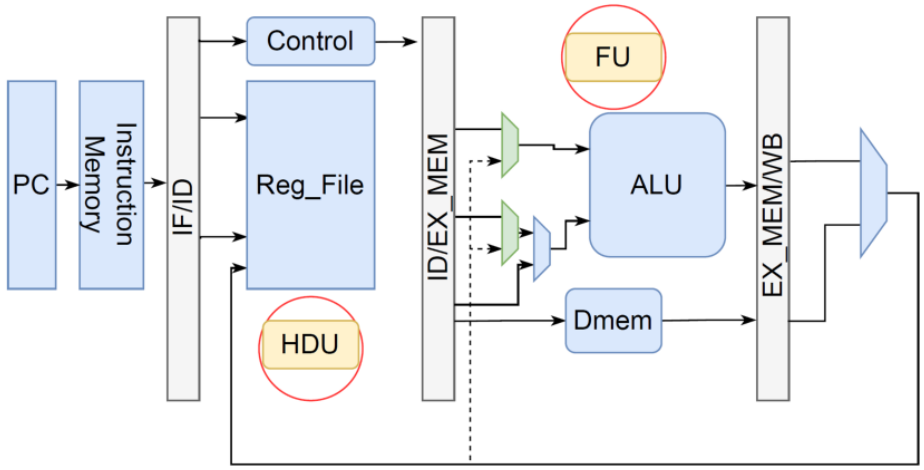
The idea here is that two input ports requesting access and I want basically an encoding of which one won or granted access. So if R0 is requesting, I get access to it and the other one doesn't get the access even though it might have requested the access. And so if the above one granted access, the below one cannot possibly win. This is the priority arbiter, But this has a fairness problem, and so the cross switch here will rotate the requests based on who got the access the last time the contention occurred.



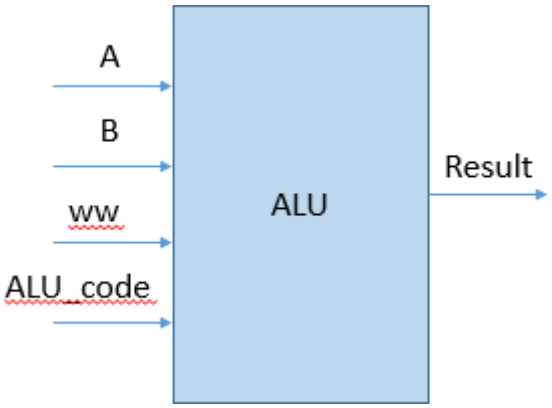
Ahmed Abuhjar
EE-577B Final Report

Processor (cardinal_processor.v):

In the CPU, the main components for each stage are the ALU, the register file, Decoder/Control unit and the two memories, one for the instructions and the other is data memory. Now the challenge in this design was that there are some hurdles that needed to be taken care of. Since it's a pipelined CPU, there could be some data dependencies among the instructions in different stages. RAW hazards could occur, meaning that instructions on the Decode stage could read from an obsolete value from the register file before their senior instructions to which they are dependent on write back into the register file. So to alleviate these data dependencies hazards, I have added the Forwarding Unit module for forwarding data as well as the Hazard Detection unit for stalling. The register file here was also designed to be internally forwarding register file to solve any data dependencies between instructions in the WB and ID stages, and I will go over the basic design of that shortly.
Register file



ALU (processor_alu.v)

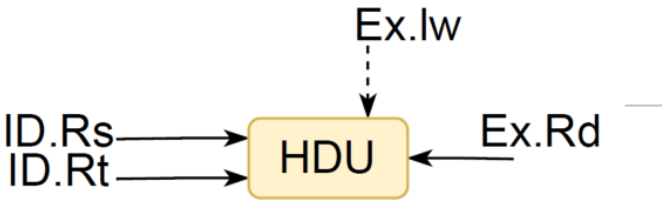


Given contents of source register, based on operation and word width I have to perform various operations. Here first I instantiated a sq_root module, and it worked properly when tested. It was done by referencing the code. But the function didn't work properly in the processor with other modules. So I used a Designware code. Testing was done mainly for different word-widths. Also division didn't work properly if B=0, then we added a condition to check that as well.

Ahmed Abuhjar
EE-577B Final Report

HDU Unit:

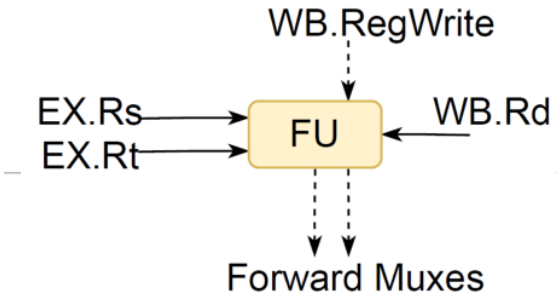
This is the Hazard detection unit used for stalling. One possible case to stall the stages is when there is a load word instruction in the EX_MEM stage and there is a dependent R-type instruction in the ID stage. In that case the HDU unit will stall both the IF and ID stage while allowing the load word instruction in the EX_MEM stage to propagate to the WB stage so that it can forward the data to its dependent R-Type instruction in the ID stage.



Hazard Detection Unit:
Case 1: Junior beq dependent on immediate senior load instruction Case 2: Junior r-type dependent on immediate senior load instruction

Forwarding Unit:

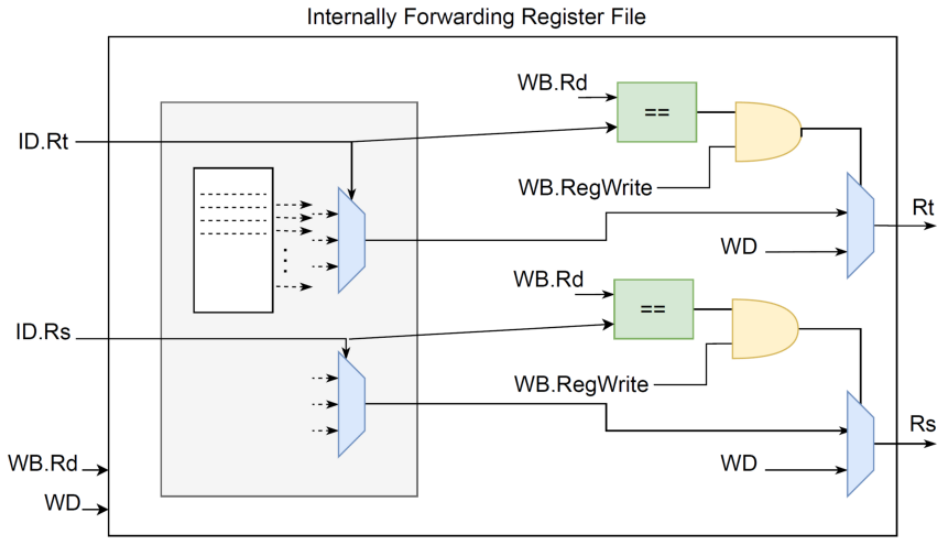
The forwarding unit is used to forward data from WB stage to EX_MEM stage to avoid RAW hazards. The forwarding unit is basically saying that whenever there's a Register Write instruction in the WB stage and there's a dependent instruction in the EX_MEM stage, then forward the data from WB to EX_MEM stage.

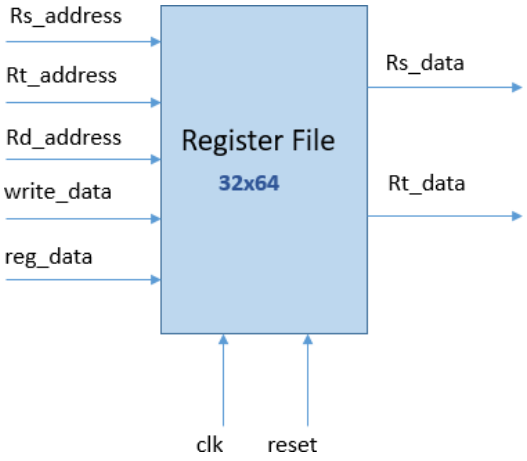


Forwarding and data dependency in the processor:
Forwarding from WB to ID (WB = S, ID = J2)
Forwarding from WB to EX/MEM (WB = S, EX/MEM = R-type i.e J1)
Forwarding from EX/MEM to ID (EX/MEM = R-type i.e S, ID = R- type/Bez/Bne i.e J1)

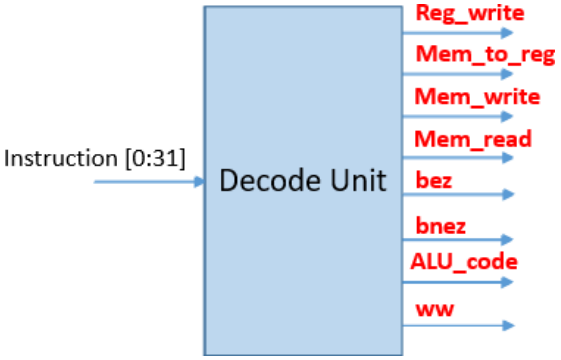
Internally Forwarding Register File:

Shown below is the internally forwarding register file. It basically functions as a forwarding unit from the WB stage to the ID stage whenever there's data dependency between these two stages. It will basically check if the instruction in the WB stage is a register writer instruction and if it's destination register is equal to one of the source registers of the instruction in the ID stage. And if that's the case then forward the data from WB directly to the source register.





Control Unit/ Decoder (processor_decoder.v)

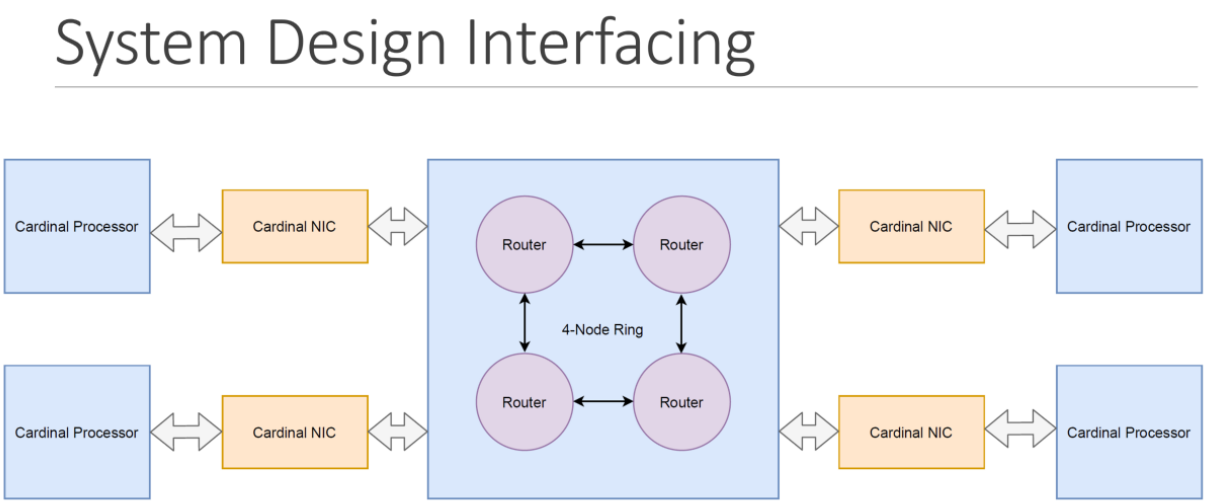


I broke the instructions according to rD, rA, rB, Opcode, ALU operation and word width. Based on the Opcode, I gave values to the control signals for the various instructions. I encountered a problem for store word and BNEZ/BEZ as rD is the source register. So depending on the Opcode, I gave rB as source register for store word and rA as source register for BEZ/BNEZ. I checked if the instructions were getting decoded properly or not and the corresponding control signals were going high for a particular instruction.

I tested the processor.v by using a testbench for a single processor for test cases 1- 39.The processor instantiated 4 times in the CPU.v file and it was tested using the TA's testbench for test cases 1 – 39. Integration of NIC, Ring and CPU

Ahmed Abuhjar
EE-577B Final Report

Testing for the system design:



Here I had to integrate all the modules together. I inserted NIC signals in the processor and gave the necessary signals between ID_EX/MEM and EX/MEM_WB stage as the NIC is contained in EX/MEM. Also I worked on the decoder to decode the m-type instruction for data memory or NIC. I instantiated the NIC 4 times in this module. I also have a ring which is instantiated once. The main problem over here was mapping of names of different instantiations. I got all the test cases working fine, except for imem0 where for memory location 5, 6 and 7 the value was going to zzzz. When I checked the waveforms I figured that the NIC was not getting the required values at all.

Synthesis:

For optimization while doing synthesis I used some switches.
To remove unwanted paths and cells I used following command:

1. set compile_delete_unloaded_sequential_cells false;

To optimize overall design I used following commands:

set compile_seqmap_propagate_constants false;
set compile_seqmap_propagate_constants_size_only true;
set hdlin_preserve_sequential true;
set_flatten true -effort medium -minimize multiple_output;

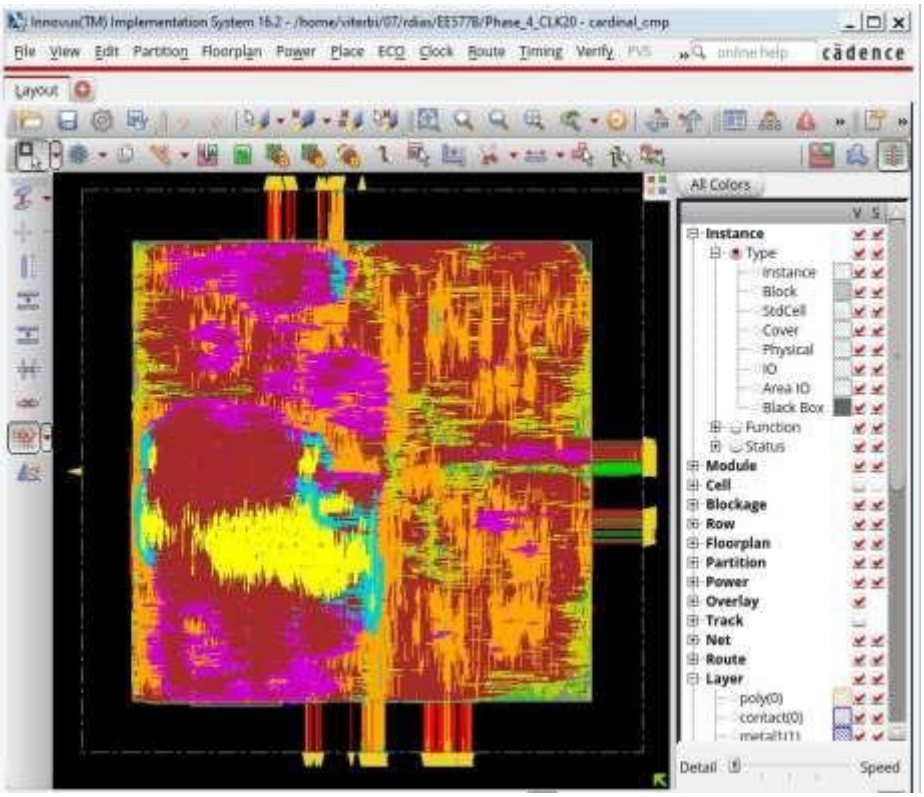
Initially I did a synthesis for clock period 100ns, then 50ns, then 30ns. I tried doing synthesis for 10ns, but got a negative slack of 7.8ns. Finally after trying many permutations and combinations, we got positive slack for 20ns. The critical path in the circuit was of MULT, DIV and SQ_ROOT for 32 and 64 bit. So if I would've pipelined the ALU for these instructions, the processor would've worked for 4ns clock period. The synthesis runtime was about 12 hours or more. This happened because the critical path in the processor was not pipelined.

Total cell area: 2310395.785494 μm2
Total clock period: 20 ns
Area × clock = 46207915.71 μm2 ns

```
Running Test # 0
diff --brief --ignore-all-space /home/viterbi/07/
diff --brief --ignore-all-space /home/viterbi/07/
diff --brief --ignore-all-space /home/viterbi/07/
Files /home/viterbi/07/ppawar/EE557/NO_Phase_4/tes
7,8c7,8
< Memory location #          6 : 4000000300000cc2
< Memory location #          7 : c000000000000ccc
---
> Memory location #          6 : c000000000000ccc
> Memory location #          7 : 4000000300000cc2
diff --brief --ignore-all-space /home/viterbi/07/
```

Ahmed Abuhjar
EE-577B Final Report

Place and Route (PnR)



While doing the PnR, I realized that depending on the dimensions of the core, the placement stage would take place, and depending on the I/O pad distance the routing stage would take place.During PnR, the dimensions of core were 1730.9 × 1730.9, but I gave dimensions of core as 1800 × 1800 with I/O pad spacing of 500 × 500. I generated the gds2 file as well.

```
#Total number of DRC violations = 0
#    number of violations = 0
#cpu time = 00:08:23, elapsed time = 00:08:24, memory = 5110.46 (MB), peak = 536
2.88 (MB)
#CELL_VIEW cardinal_cmp,init has no DRC violation.
#Total number of DRC violations = 0
#Post Route wire spread is done.
#Total wire length = 22403181 um.
#Total half perimeter of net bounding box = 19633198 um.
#Total wire length on LAYER metal1 = 329672 um.
#Total wire length on LAYER metal2 = 4664991 um.
#Total wire length on LAYER metal3 = 6424596 um.
#Total wire length on LAYER metal4 = 4230677 um.
#Total wire length on LAYER metal5 = 3851504 um.
#Total wire length on LAYER metal6 = 1716307 um.
#Total wire length on LAYER metal7 = 648951 um.
#Total wire length on LAYER metal8 = 250861 um.
#Total wire length on LAYER metal9 = 224635 um.
#Total wire length on LAYER metal10 = 60987 um.
#Total number of vias = 5394280
#Up-Via Summary (total 5394280):
#
#-----------------------
#  Metal 1      2656510
#  Metal 2      2002146
#  Metal 3       503839
#  Metal 4       133393
#  Metal 5        53926
#  Metal 6        25112
#  Metal 7        10580
#  Metal 8         7161
#  Metal 9         1613
#-----------------------
#               5394280
```

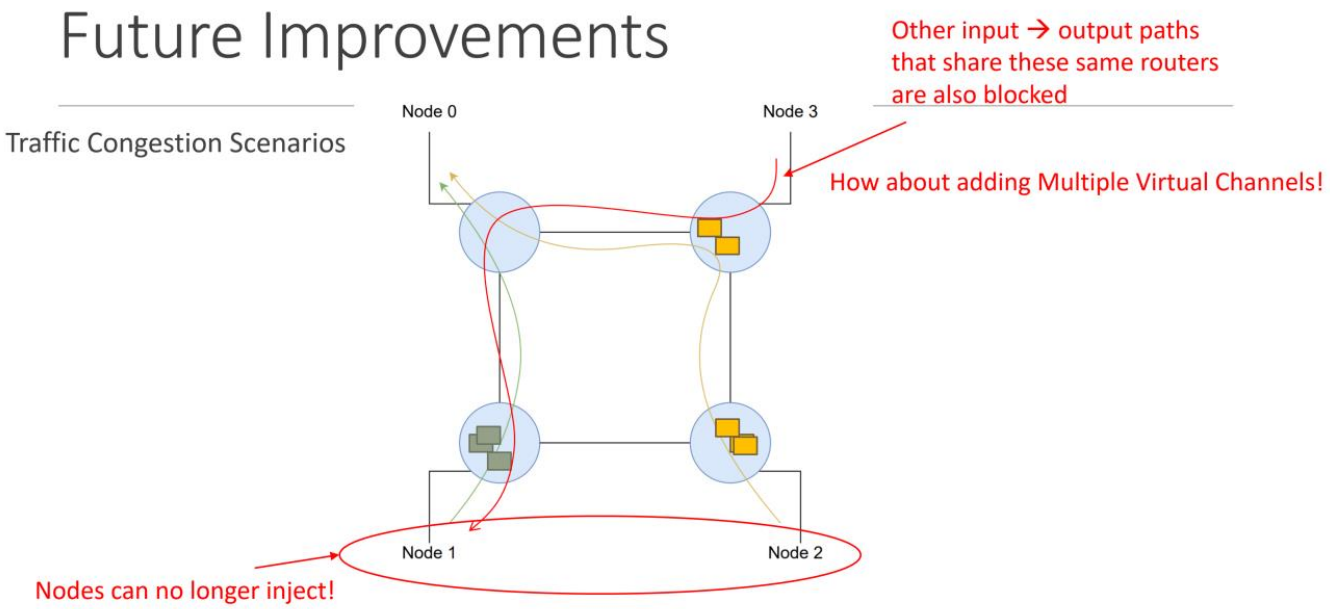
Ahmed Abuhjar
EE-577B Final Report

Future Improvements:

Problem statement:

Shown in the NoC below, a classic traffic congestion scenario is shown here where we have two different input ports. Here we have input 1 and input 2 sending data to output port 0. Under this scenario, the packets meet at the router that is connected to the node 0 where they both are asking for access to the output port. So this is a scenario of contention where only one packet can be granted access to the output port where the other must be buffered within the router.

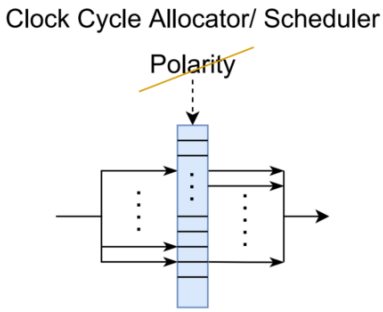
The buffers that are already available in the routers of the NoC might not be sufficient to handle contention. Now let's say this scenario happens continuously over a long burst of traffic. Eventually, the buffer at the routers of this path will become full causing contention to be back-pressured to downstream routers making its way back to the input ports. This will prevent the input port to inject any more packets to the NoC, and additionally because the input/output paths within the network on chip are shared, other paths could also be blocked. So here for example, the path between node 3 and node 1 also uses this router which is blocked because of the contention that happened from input port 1 and input port 2. So we see that this scenario not only can prevent these nodes from injecting but can also create contention at other ports as well. This is often called a Head of Line Blocking. So in order to alleviate this traffic congestion or the head of line blocking , we can introduce multiple virtual channels.

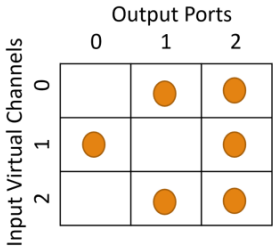


Solution for the problem:

As shown in the figure below, we can have many virtual channels instead of only two to alleviate congestion. We have 3 input ports and at each input port we can have N virtual channels within an input buffer. Due to the head of line blocking I want to pick from a number of virtual channels to go to. And the clock cycle allocator will then be more involved in this case. The allocator is just one step further of the arbiter design. I have many inputs requesting access to many outputs as opposed to the arbiter where we have many inputs requesting to one output port. Some relevant work was done to implement this in the past, I've drawn here some 2D matrices to illustrate the idea of this potential improvement.

The circles are inputs requesting access to these outputs. If for example we have four inputs requesting access to four outputs. And so the allocation or the arbitration is done in two stages. First thing it does is that for each input, I pick one virtual channel that's going to win. Once I've done that, we will have one virtual channel sending to different or same output port, which I can then arbitrate as well to determine which input wins.





Division of the work:

Team members: All the work was done by
Ahmed Abuhjar