Multicore CPU using a NoC based on packet switching

Talibov Serhan

May 2025

User manual

Contents

1 Thanks to

- Lezhnev E.V., Romanov A.Yu. - for the developing the idea and making this project possible;
- Grushevskii N.I. - for being a codeveloper of the NoC connection subsystem taking an equal part of this task;
- Nigmatullin N.R. - for the development of NoC converters;
- Zhelnio S. - for the development of schoolRISCV soft core.

2 Introduction

2.1 Network on a chip

Network on a chip (NoC) is a network-based communication between IP cores on an integrated circuit (IC), that uses packet switching to transmit data packets. This type of interconnect gives a great tradeoff between the chip area used and the interconnect bandwidth compared to other technologies such as point-to-point, bus, or ring-type connections when the number of connection points increases.

2.2 Goals of this project

The goal of this project is to create an HDL description of an example of a multicore CPU based on NoC technology. This CPU consists of several parts:

- CPU core - a schoolRISCV single-cycle soft core;
- NoC interconnect - a mesh 3x3 NoC for a 9-core CPU;
- packet converters - to convert CPU output into transmittable data packets (flits).

3 Repository directories

Directiry	Description
doc	
└ UserManual.pdf	user manual for this project
boards	**HDL files and scripts for programming FPGA boards**
└ toplevel.sv	a common toplevel module to be used in board-specific modules
cores	**HDL files for schoolRISCV soft core and supporting modules**
├ converters	HDL files for converters between memory controller (MC) packets and NoC packets
├ packet_collector.sv	converter from NoC to MC
└ splitter.sv	converter from MC to NoC
└ src	HDL files for the schoolRISCV soft core
├ cpu_with_ram.sv	module that connects CPU and RAM to the MC
├ ram.sv	a two-port RAM

├ sm_register.v	a DFF for an instruction counter
├ sm_rom.v	preloaded instructions
├ sr_cpu.v	a CPU module with a counter, decoder, register file, ALU, AGU and a control unit
├ sr_cpu.vh	‘define macros for RISCV opcodes and ALU/AGU oper codes
├ sr_mem_ctrl.sv	a memory controller (MC) connecting CPU to RAM through the NoC
└ sr_mem_ctrl.svh	‘define macros for MC instructions
cpu	**HDL files for the 9-core CPU on a NoC**
├ noc_with_cores.sv	connects 9 CPU cores to the mesh 3x3 NoC
└ uart.sv	hooks up the 9-core CPU to the UART to monitor RAM data at a given address
mesh_3x3	**HDL files for the 3x3 mesh NoC**
├ inc	‘define macros for NoC configuration
├ noc.svh	macros for general NoC parameters
├ noc_XY.svh	macros for topology-specific (mesh) parameters
├ queue.svh	macros for queue parameters
└ router.svh	macros for router parameters
├ noc	
└ noc.sv	module that connects 9 routers into a NoC
└ src	HDL files for router components
├ algorithm.sv	an XY algorithm for packet switching
├ arbiter.sv	a module that chooses a packet to be switched
├ queue.sv	FIFOs for collecting incoming packets
└ router.sv	a module that creates a router from its components
modelsim	
├ image_chunk_1..2.hex	RAM images that contain a picture
├ instr_node_0..8.hex	RAM images that contain RISCV codes for each core
├ modelsim_run.bat	a batch file that launches ModelSim using modelsim_script.tcl script
└ modelsim_script.tcl	a script, according to which the simulation is ran
tb	HDL files for testbenches
└ tb.sv	a testbench files that tests the CPU, dumping RAM contents at the end

4 General description

This project contains a complete HDL description of a 3x3 mesh NoC, that uses an XY routing algorithm for packet switching. Each NoC router has a number of bidirectional connections for its neighbors and a single one for a CPU core assigned to the router as shown in Figure 1:

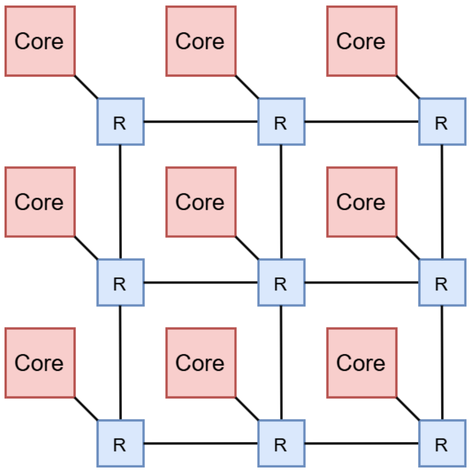

Figure 1: Simplified illustration of the NoC

If simulated or programmed into an FPGA without any modifications, this CPU will compute a simple convolution of a 30x60 image with a 3x3 kernel without any padding, so with any given 30x60 image the result will be a 28x58 image, where each of the pixels are computed with a formula (1), where *in* is an initial image, *k* is the convolution kernel and *out* is the result:

$$out(x-1, y-1) = \sum_{i=1}^{3}\sum_{j=1}^{3} k(i,j) * in(x-i, y-j) \tag{1}$$

Given that $k = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$, the resulting image will look like a blurred version of the original when divided by a factor of $\sum_{i=1}^{3}\sum_{j=1}^{3} k(i,j) = 10$ as shown in Figure 2:

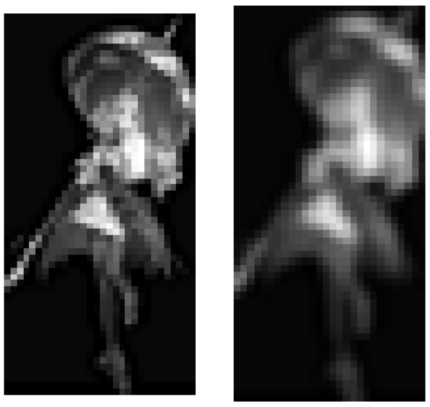

Figure 2: Initial image (left) and the result of convolution after being divided by 10 (right)

5 Detailed description

5.1 CPU structure

In this project a schoolRISCV single-cycle soft core [1] was used as a CPU. It is a tiny core that supports only a tiny subset of RV32I integer instructions: `add, or, srl, sltu, sub, addi, lui, beq, bne`. The structure of this core is presented in Figure 3:

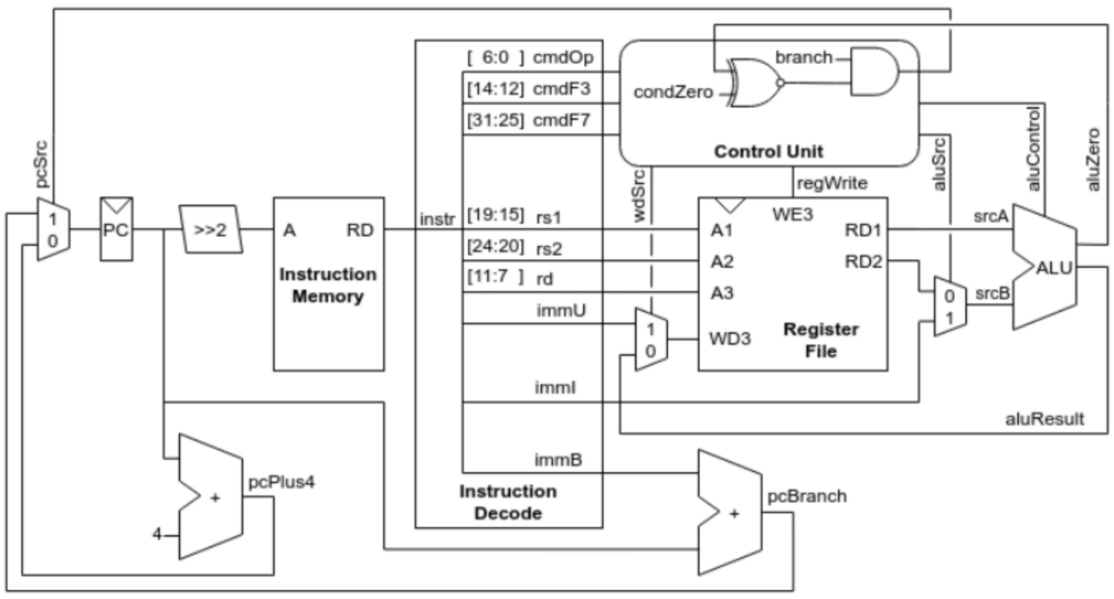

Figure 3: Original single-cycle schoolRISCV CPU core

This instruction set is not enough to compute integer convolution because it uses multiplication and addition, hence the CPU was modified: new opcodes and alu operations were added inside the `sr_cpu.vh` file, and ALU itself was modified to support integer multiplication.

Another modification was to add `sw` and `lw` instructions, because the idea is to read the image from one location in the RAM and write the result to another. To support these instructions, an AGU (address generation unit) was developed, which decodes `lw rd, imm(rs1) / sw rs2, imm(rs1)` instruction format and outputs data, address and instruction, which are read by the memory controller (MC). AGU is able to stop the instruction counter inside the CPU to pause it when it encounters an `lw/sw` instruction until further notice by the MC: after `lw` AGU waits until the MC signals that the data were written into the register file (RF), and after `sw` AGU waits until the MC signals that it took the instruction. A modified CPU structure is presented in Figure 4:

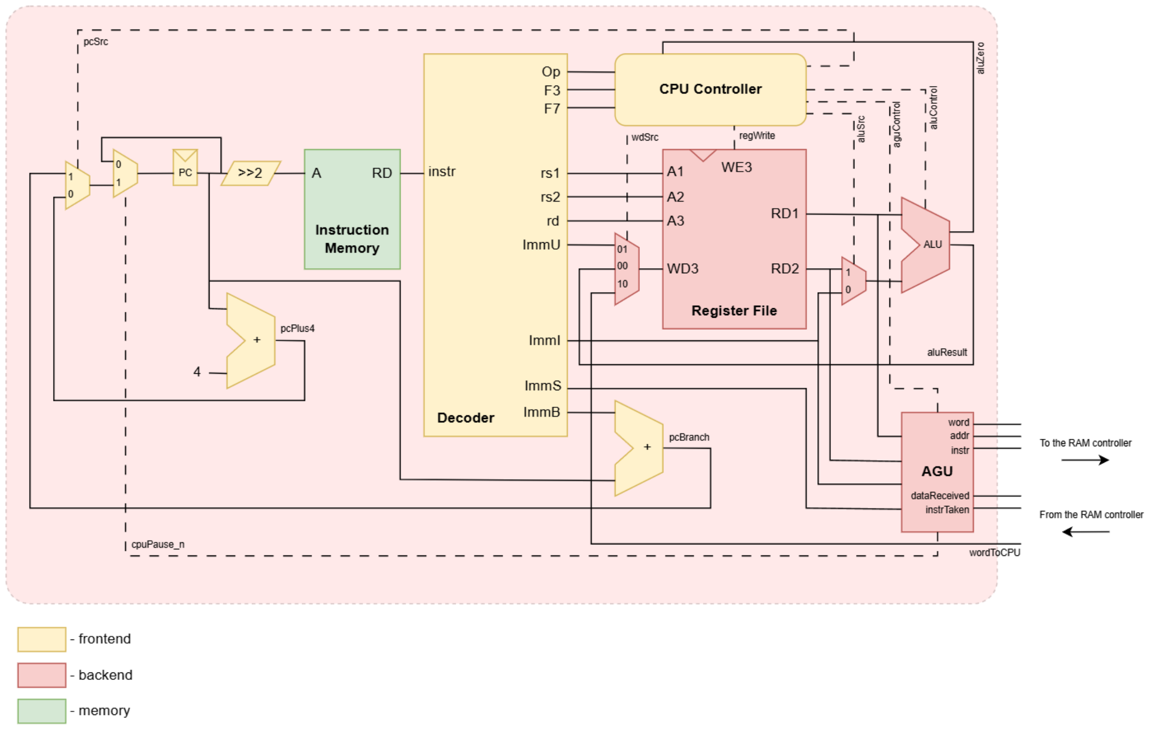

Figure 4: Modified single-cycle schoolRISCV CPU core

5.2 Convolution code

A 9-core CPU has a 1024-word RAM chunk assigned to each core totaling 9216 words for the entire CPU. The processor sees it as a continuous memory with logical addresses spanning from 0x0 to 0x23FF. The 30x60 image is located in 0x0-0x707 RAM space, the 3x3 kernel is located in 0x800-0x809 RAM space, and the resulting 28x58 image will be written to the 0xC00-0x1258 RAM space. The following listing contains a full RISCV assembly code that is preloaded into core 0, that computer the first 6 lines of the resulting image:

```
li a2, 1

li t0, 30
sub a3, t0, a2 ; a3 contains horizontal dimension of the resulting image

li a4, 7 ; a4 = image's last line to compute + 1

li t2, 1024
addi t2, t2, 1024

addi t3, t2, 1024 ; starting address for storing the image

li a0, 1 ; width loop counter
li a1, 1 ; height loop counter

lw s2, 0x0(t2)
lw s3, 0x1(t2)
lw s4, 0x2(t2)
lw s5, 0x3(t2)
lw s6, 0x4(t2)
lw s7, 0x5(t2)
lw s8, 0x6(t2)
lw s9, 0x7(t2)
lw s10, 0x8(t2) ; loading the kernel into registers

convolveY: ; iterate over lines
    convolveX: ; iterate over each pixel

        li t6, 0 ; set up the result register

        mul gp, t0, a1
        add gp, gp, a0 ; gp = image[a1, a0] address


        add t4, gp, zero
        sub t4, t4, t0
        sub t4, t4, a2 ; t4 = image[a1 - 1, a0 - 1] address


        lw t5, 0x0(t4)
        mul t5, t5, s2
        add t6, t6, t5 ; t6 = t6 + image[a1 - 1, a0 - 1] * kernel[0, 0]

        addi t4, t4, 1 ; t4 = image[a1 - 1, a0] address


        lw t5, 0x0(t4)
        mul t5, t5, s3
        add t6, t6, t5 ; t6 = t6 + image[a1 - 1, a0] * kernel[0, 1]

        addi t4, t4, 1 ; t4 = image[a1 - 1, a0 + 1] address


        lw t5, 0x0(t4)
        mul t5, t5, s4
        add t6, t6, t5 ; t6 = t6 + image[a1 - 1, a0 + 1] * kernel[0, 2]

        add t4, gp, zero
        sub t4, t4, a2 ; t4 = image[a1, a0 - 1] address


        lw t5, 0x0(t4)
        mul t5, t5, s5
        add t6, t6, t5 ; t6 = t6 + image[a1, a0 - 1] * kernel[1, 0]

        addi t4, t4, 1 ; t4 = image[a1, a0] address


        lw t5, 0x0(t4)
        mul t5, t5, s6
        add t6, t6, t5 ; t6 = t6 + image[a1, a0] * kernel[1, 0]

        addi t4, t4, 1 ; t4 = image[a1, a0 + 1] address


        lw t5, 0x0(t4)
        mul t5, t5, s7
        add t6, t6, t5 ; t6 = t6 + image[a1, a0 + 1] * kernel[1, 0]

        add t4, gp, zero
```

```
            add t4, t4, t0
            sub t4, t4, a2 ; t4 = image[a1 + 1, a0 - 1] address


            lw t5, 0x0(t4)
            mul t5, t5, s8
            add t6, t6, t5 ; t6 = t6 + image[a1 + 1, a0 - 1] * kernel[2, 0]

            addi t4, t4, 1 ; t4 = image[a1 + 1, a0] address


            lw t5, 0x0(t4)
            mul t5, t5, s9
            add t6, t6, t5 ; t6 = t6 + image[a1 + 1, a0] * kernel[2, 0]

            addi t4, t4, 1 ; t4 = image[a1 + 1, a0 + 1] address


            lw t5, 0x0(t4)
            mul t5, t5, s10
            add t6, t6, t5 ; t6 = t6 + image[a1 + 1, a0 + 1] * kernel[2, 0]

            sw t6, 0x0(t3) ; result[a1 - 1, a0 - 1] = t6
            addi t3, t3, 1 ; next result pixel

        addi a0, a0, 1 ; next image pixel
        bne a0, a3, convolveX
        addi a0, zero, 1

    addi a1, a1, 1 ; next image line
    bne a1, a4, convolveY
    addi a1, zero, 1

idle:
  beq zero, zero, idle ; stop
```

Each core computes lines that are contained in the `[a0 - 1, a1 - 2]` range, where `a0` and `a1` are the initial values of the respective registers. The `t3` register should be set up correspondingly to the initial `a0` value. Each core is set up in a way that helps avoid any overlaps or skips in the resulting image.

5.3 Memory controller

Each CPU core sees the RAM as a continuous unit in a logical address space, meanwhile physically it is subdivided into 9 pieces of 1024 words each addressed from 0x0. This means that an MC, that is able to translate from logical address to physical, send and receive messages from the NoC and interact with the CPU core, is needed. The memory controller was designed to perform two types of actions:

- Take the data and instruction from the CPU and send them out to the NoC;
- Take the full packet from the NoC, analyse it and perform necessary actions.

For a full packet coming from the NoC there are 3 types of actions:

- Load requested: return the packet with data to the requester;
- Load satisfied: receive data from the packet and write it into the register file (RF);
- Store data: writes received data into the received RAM address;

A simple schematic for the developed MC is presented in Figure 5:

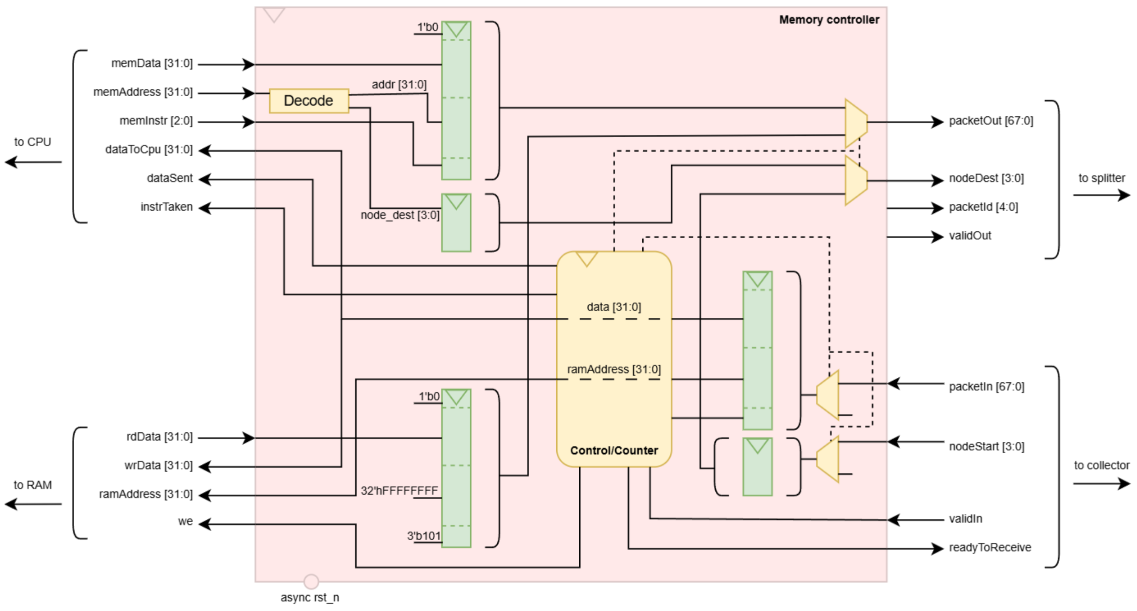

Figure 5: A simple schematic for the memory controller

In this diagram green bars represent flip-flops, all of the wires' are assigned to whatever they are connected to with regard to the connection direction. A brief description of the MC's working principle assuming that all of the inputs contain valid data:

- Cycle 0: data from the CPU is sent to the NoC and data from the NoC is received using corresponding flip-flops;
- Cycle 1: data from the NoC flip-flop is analyzed and necessary actions are performed:
  - Load requested: ramAddress is set to the received address, then the data is sent out to the NoC at cycle 2;
  - Load satisfied: the data is sent into the dataToCpu port to be written to the RF;
  - Store data: the data is written into the RAM to the received address.

5.4 Converters

An MC sends a 68-bit packet with the ID of a destination node and receives a 68-bit packet with the ID of a sender node. A NoC may transfer the data in a completely different format, so there was a need to design MC-to-NoC converters, more specifically a splitter, that divides an MC packet into transferrable data packets (flits), and a packet collector, that assembles NoC flits into a full packet, that is readable by the MC.

5.4.1 Splitter

A schematic of a splitter is presented in Figure 6:

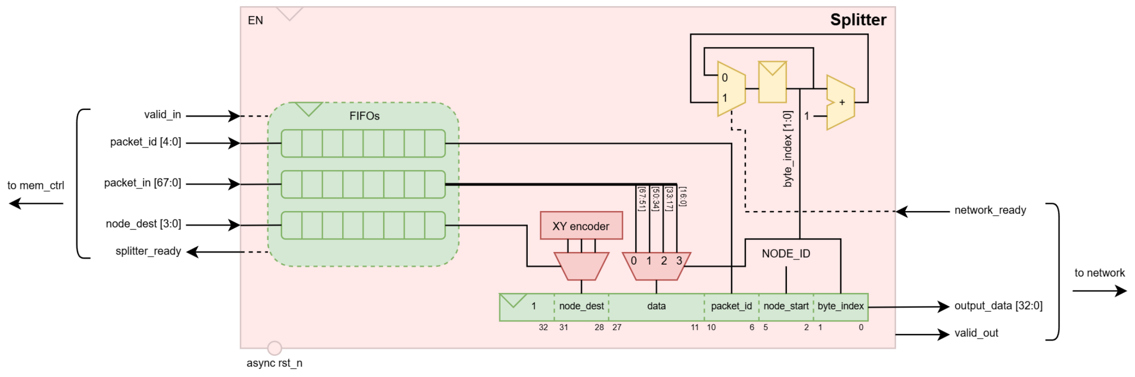

Figure 6: A diagram of a splitter

It consists of three FIFOs, that collect data packets, an ID of a destination node and a packet ID, that is generated by the MC. Each clock it creates a 33-bit flit, that contains a validity bit, a 17-bit payload and other information, that is used to route the flit and then assemble them into the full packet at the destination. A full data packet is split into 4 flits, and after the splitter finishes sending out pieces of a full packet it shifts all of the FIFOs.

5.4.2 Packet collector

A schematic of a packet collector is presented in Figure 7:

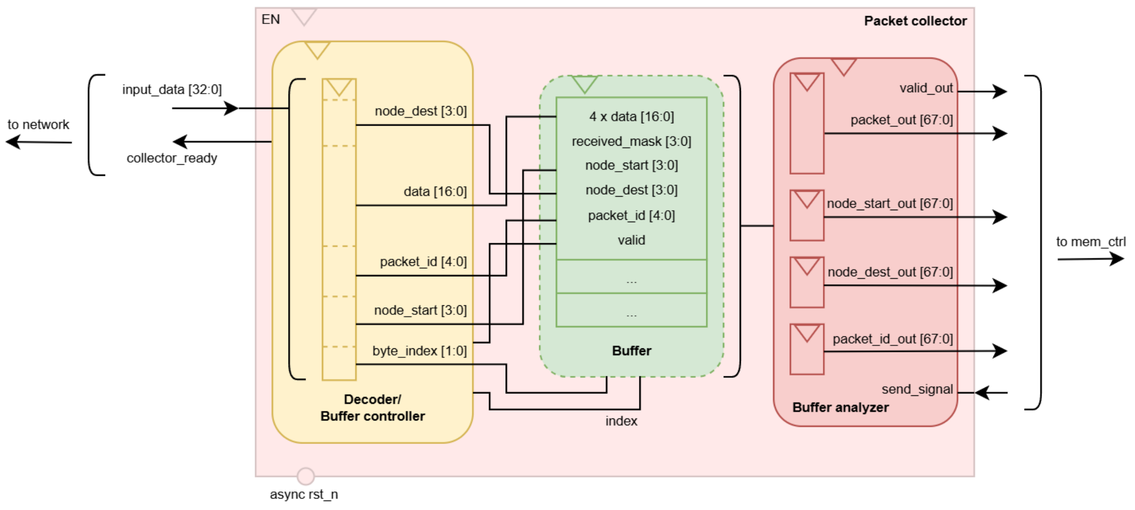

Figure 7: A diagram of a packet collector

It consists of a buffer, where the packets are assembled, a decoder, which writes into the buffer, depending on contents of the incoming packet, and an analyzer, which reads from a buffer each clock in search of a complete packet.

5.5 Network-on-chip

A Network-on-chip (NoC) allows to transfer data packets between completely different pairs of nodes simultaneously using a network of routers, that use packet switching to route traffic. This particular NoC is a 3x3 mesh network that uses an XY-algorithm to switch data packets. The developed router consists FIFOs for each input port, that collect incoming packets, a round-robin arbiter, that chooses a FIFO, from which the packet is taken out to be analyzed, and an algorithm, that analyzes the chosen packet and outputs it to the corresponding output. A packet contains XY-coordinates of a destination, which in conjunction with knowing its own coordinates is enough for the algorithm to determine the correct output. A diagram of a router is presented in Figure 8:

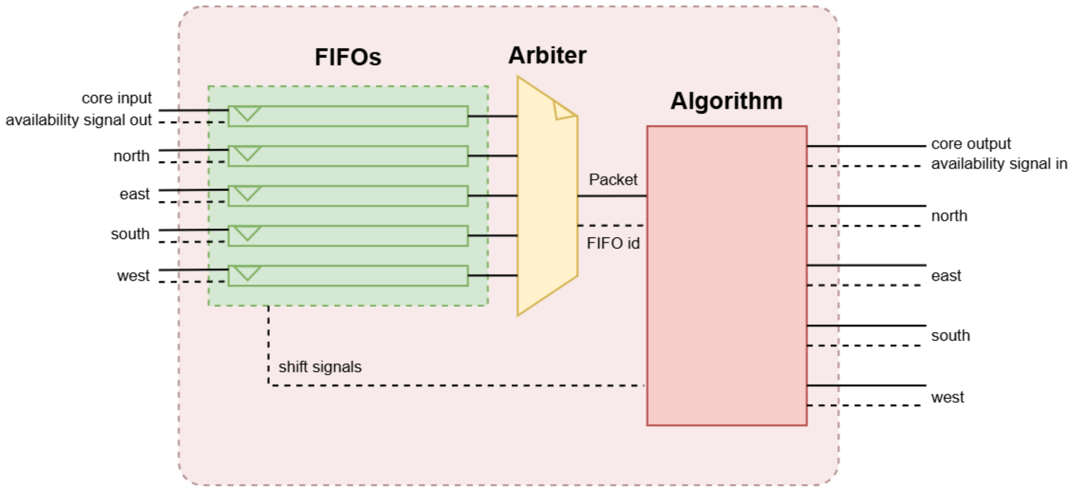

Figure 8: A diagram of a NoC router

Ports, which are named from north to west are used to connect routers together, the core input is connected to a splitter of the core assigned to the router and the core output is connected to a packet collector, that belongs to the same core. With all of the components assembled together, the whole network looks like the diagram, presented in Figure 9:

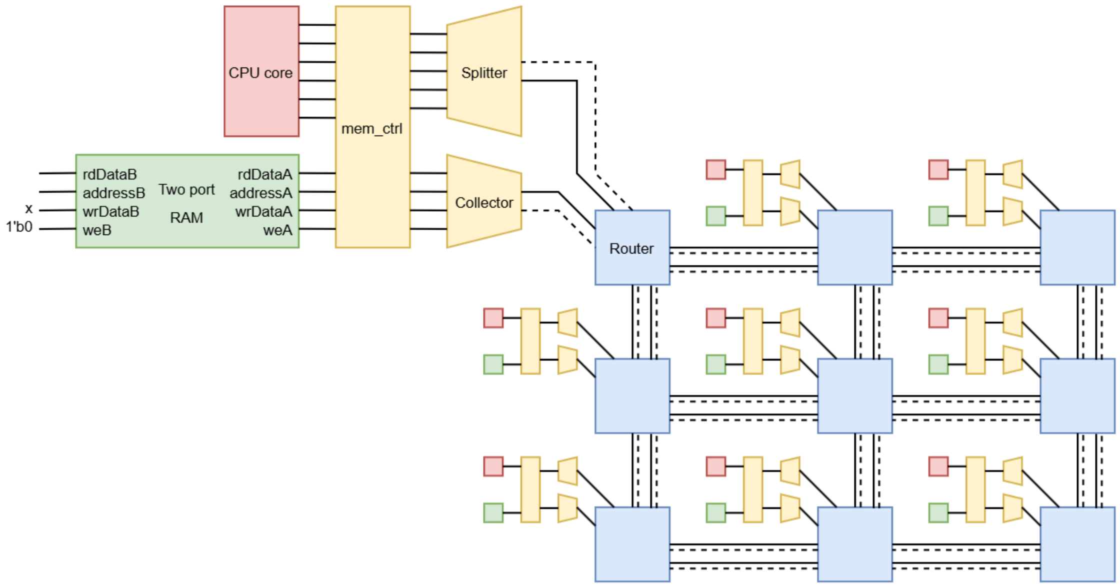

Figure 9: A diagram of a complete CPU

5.6 UART

UART is used to monitor RAM data at all times. Each of the nine RAM chunks is a 2-port RAM, where the first port is used in the CPU and the second one is read-only and connected to the UART logic. Separate uartTx and uartRx, that handle the RX and TX data streams, and the logic, that manages these modules, were developed. A diagram is presented in Figure 10:

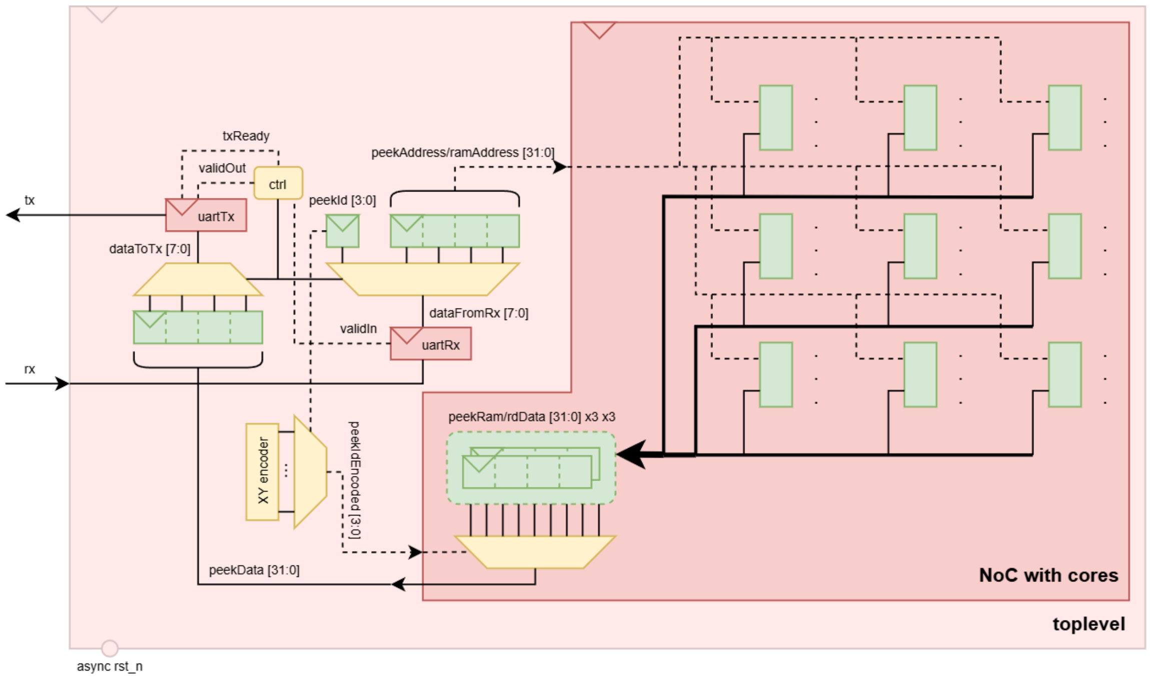

Figure 10: A diagram of UART logic

This particular UART controller is set to 115200 bauds per second, though it is easily configurable from the `toplevel.sv` file. The first step is to receive from the RX port 4 bytes, that encode an address (valid addresses are from 0x0 to 0x3FF) starting with the LSB first, then receive a single byte that encodes the ID of a node (valid IDs are from 0 to 8). Then the CPU returns RAM contents from that particular location, and since the word is 32 bit long, it is transferred through TX as 4 bytes starting with the LSB first too.

6 Usage and necessary software

// coming soon