Mlchip Final Project report

413510028 林柏佑

Code Structure

```
├── FP
│   ├── clockreset.cpp
│   ├── clockreset.h
│   ├── controller.h
│   ├── core.h
│   ├── data
│   │   ├── dog.txt
│   │   ├── cat.txt
│   │   ├── conv1_bias.txt
│   │   ├── conv1_weight.txt
│   │   ├──    .
│   │   ├──    .
│   │   ├──    .
│   │   └── imagenet_classes.txt
│   ├── main.cpp
│   ├── Makefile
│   ├── pe.h
│   ├── ROM.cpp
│   ├── ROM.h
│   └── router.h
└── FP_Optimized
    ├── clockreset.cpp
    ├── clockreset.h
    ├── controller.h
    ├── core.h
    ├── data
    │   ├── dog.txt
    │   ├── cat.txt
    │   ├── conv1_bias.txt
    │   ├── conv1_weight.txt
    │   ├──    .
    │   ├──    .
    │   ├──    .
    │   └── imagenet_classes.txt
    ├── main.cpp
    ├── Makefile
    ├── pe.h
    ├── ROM.cpp
    ├── ROM.h
    └── router.n
```

Code Overview & Optimization

In the Final Project, the overall architecture remains the same as in HW4. However, we were required to implement the AXI4 protocol along with its optimized design. For the AXI4 interface, I adopted a simplified approach by focusing solely on the read transaction signals: ***ARADDR***, ***ARVALID***, ***ARREADY***, ***RDATA***, ***RVALID***, and ***RREADY***. The remaining signals were omitted, as they had minimal impact on the functionality of

our design.

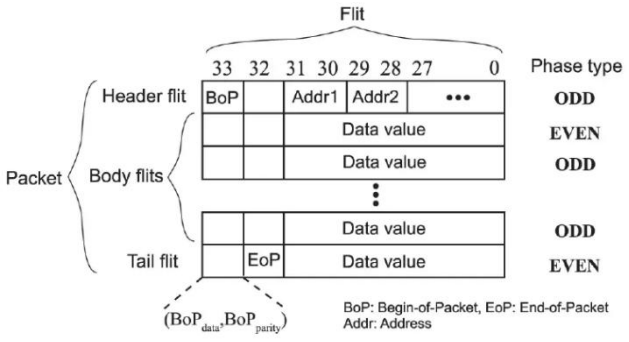
Read Address Channel:

Signal	Source	Description
ARADDR	Master (Controller)	Specifies the read target address. In this design, it is encoded as (layer_id << 1 | layer_id_type)
ARVALID	Master (Controller)	Asserted when the controller initiates a valid read request. It remains high until the slave acknowledges by asserting ARREADY.
ARREADY	Slave (ROM)	Asserted by the ROM when it is ready to accept the address and begin the read transaction.

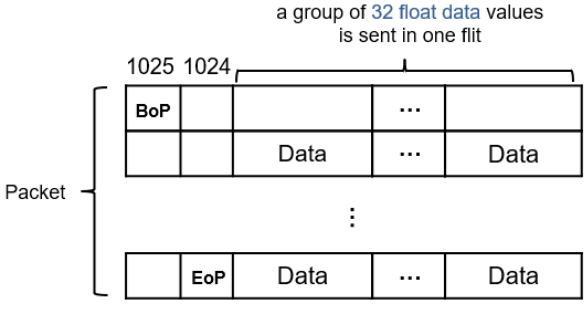
Read Data Channel:

Signal	Source	Description
RDATA[32]	Slave (ROM)	A 32-element array output from ROM. Each read cycle returns 32 floating-point values in parallel, significantly increasing memory bandwidth.
RVALID	Slave (ROM)	Asserted when RDATA is valid and contains 32 ready-to-read values. This signal is held high until the master acknowledges with RREADY.
RREADY	Master (Controller)	Indicates that the controller is ready to receive the data. Data is only transferred when both RVALID and RREADY are high.

As for the optimized design, the ***RDATA*** signal has been enhanced to support an array structure, allowing it to *transmit 32 data values simultaneously* in a single clock cycle. This significantly improves data throughput compared to the original design, which could only send one data value at a time. Correspondingly, the flit structure in the data transmission protocol has also been optimized: each flit is now capable of carrying *32 data entries*, effectively aligning with the updated RDATA width. This optimization reduces the number of required flits per packet, minimizes transmission overhead, and leads to more efficient use of the NoC bandwidth.

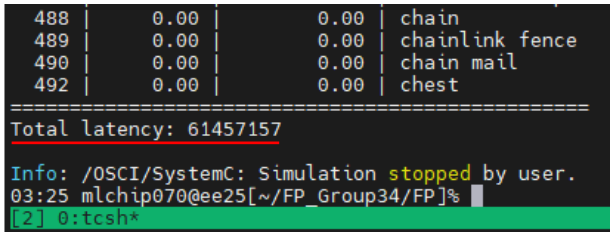


Before

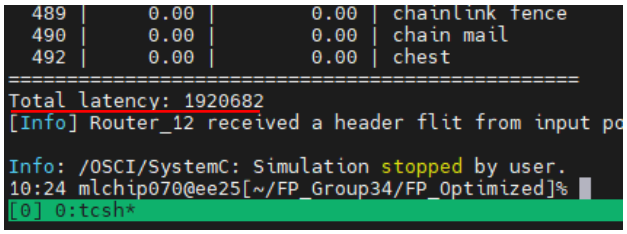


After (each flit carrying 32 data entries)

As a result of these enhancements, the system's total latency was dramatically reduced from 61,457,157 cycles to just 1,920,682 cycles—an improvement of nearly *32×*. This significant acceleration highlights the effectiveness of the parallel data transfer mechanism and the redesigned flit packaging strategy, both of which contributed to the substantial performance gain with minimal architectural overhead.



Before



After

One of the key advantages of this optimization is that it requires minimal changes to the overall design. The packet structure remains largely the same; the only major difference is that the flit size has been increased to accommodate more data. This means we can achieve significant performance improvements without needing to overhaul the existing data packaging or transmission mechanisms.

File Descriptions

core.h: Core Module: Handling PE <-> Router Communication

This module manages the interaction between the local Processing Element (PE) and the router. It is responsible for:

- Fetching packets from the PE, converting them into flits (including header/body/tail format), and pushing them into a transmission queue.
- Receiving flits from the router, reconstructing packets from them, and passing them to the PE.

It contains utility functions ***float2lv*** and ***lv2float*** to handle float-to-bit conversion and vice versa. The handshake logic uses ***req_tx***, ***ack_tx***, ***req_rx***, and ***ack_rx*** to ensure reliable communication.

controller.h: Specialized Core for ROM Data Dispatching and Inference Coordination

Acts as a specialized version of the core module, responsible for initializing the inference pipeline. Uses the same NoC handshake protocol (***req***/***ack***) as a standard core to transmit the packets via routers. It will read input feature maps and layer

configurations from the ROM and determines the correct destination PE (Processing Element) based on the CNN operation type (e.g., Conv, MaxPool, FC, Softmax).
For each processing step, constructs control packets that include:

- Source and destination PE IDs
- Operation ID (op_id) indicating the computation type
- Necessary data payloads split into header/body/tail flits

After completing all layer operations, collects the final results from the destination PE and prints the top-100 predictions in a fixed, formatted output.

router.h: Router Module with XY Routing

This file implements the core logic of the router module in a 4x4 mesh NoC. The router supports 5 ports: LEFT, RIGHT, UP, DOWN, and CORE. The routing algorithm used is ***XY routing***, which prioritizes horizontal movement (X-axis) before vertical (Y-axis). Each input port is associated with a small FIFO buffer (depth = 3) to store incoming flits. The router also handles output port arbitration, ensuring only one input can occupy each output port at a time. Tail flits signal the end of a packet and release the output port.

main.cpp: signal connect between layers & module create

- Creation of 15 Cores, 1 controller, 1 rom, and 16 Routers.
- Interconnection between routers to form a 2D mesh.
- Connection between each Core and its local Router.
- Setup of handshake signals (***req***, ***ack***, ***flit***) across all ports.

This file essentially acts as the system's backbone, coordinating the initialization and interconnect logic.

pe.h: PE Module – Layer-wise CNN Computation Unit

This file define the behavior of each Processing Element (PE), which is responsible for executing specific CNN sub-operations, such as Convolution, MaxPooling, Fully Connected (FC), ReLU, and Softmax.

- Each PE receives control packets from the controller via the router, which include metadata such as operation type and routing info.
- Upon receiving a packet, the PE:
    - Loads corresponding weights, biases, and input features.

- Performs the specified operation (Conv, MP, FC, Softmax) based on the op_id encoded in the packet.
- Stores the computation result into its internal buffer or sends it to the next PE via the router.

- Maintains internal state registers to track routing targets for multi-stage inference.
- Provides two primary functions:
  - ***get_packet()*** – Used to generate and dispatch a new packet into the NoC.
  - ***check_packet(Packet*)*** – Parses and verifies an incoming packet and executes the appropriate CNN operation.

This modular design allows each PE to work independently while collectively executing a full forward pass through the CNN, layer by layer.

clockreset.cpp 、***clockreset.h***: Clock and Reset Signal Generation

Analysis

---

1. Router and Network Interface (NI) Design

Each router I designed has five input/output ports corresponding to the four directions and the local core. Every input port is connected to a FIFO queue with a depth of 3. To prevent output contention, only one input is allowed to occupy a given output port at a time, controlled by a signal output_busy[i].

The Network Interface (NI) handles the communication between the Core and the Router:

- When the PE generates a packet, the NI converts it into multiple flits and pushes them into a transmission queue.
- When flits are received from the router, the NI reassembles them into a complete packet and passes it to the PE.

This modular structure separates the concerns of computation and communication, making the overall design more manageable.

2. Routing Algorithm

I used a simplified XY routing algorithm. The basic strategy is:

- If the destination is in the same row, the router moves vertically (UP or DOWN)

based on the column difference.

- If not in the same row, it prioritizes horizontal movement (LEFT or RIGHT).
- If the destination ID matches the router's own ID, the flit is forwarded to the CORE port.

This approach is straightforward and avoids unnecessary detours, although it might not handle certain congestion patterns efficiently.

3. Buffer Depth and Observations

Initially, I used a buffer depth of 2 for each input port but later increased it to 3 and even experimented with 100. Surprisingly, regardless of the buffer depth, the official Demo pattern always took 140 cycles to complete. This observation suggests that buffer size was not the limiting factor in performance under this pattern. To further improve throughput, I believe introducing virtual channels would be necessary to avoid output port blocking.

4. Use of Virtual Channels

This design ***does not include virtual channels***. Each input port has a single queue, and arbitration is done over a single output path. While this makes the implementation simpler, it can result in head-of-line blocking where one blocked input prevents others from sending even if the path is available.

5. Flit Format and Type Detection

- ***Bits [1025:1024] = 10***: Header flag, indicates a header flit.
- ***Bits [1025: 1024] = 01***: Tail flag, indicates a tail flit.
- ***Bits [1023:0]: Payload*** – either a float number or address information.

This structure allows me to distinguish the flit type easily:

- A flit with bit 1025 = 1 is the header flit, containing source and destination IDs.
- A flit with bit 1024 = 1 (and bit 1025 = 0) is the tail flit, marking the end of a packet.
- All others are body flits.

6. Float Conversion Functions

float2lv(const float val):

```
sc_lv<32> Core::float2lv(const float val)
{
    sc_dt::scfx_ieee_float ieee_val(val);
    sc_lv<32> lv;
    lv[31] = (bool)ieee_val.negative();
    lv.range(30, 23) = ieee_val.exponent();
    lv.range(22, 0) = ieee_val.mantissa();
    return lv;
}
```

lv2float(const sc_lv<32> val):

```
float Core::lv2float(const sc_lv<32> val)
{
    sc_dt::scfx_ieee_float id;
    sc_lv<32> sign = 0;
    sc_lv<32> exp = 0;
    sc_lv<32> mantissa = 0;
    sign[0] = val[31];
    exp.range(7, 0) = val.range(30, 23);
    mantissa.range(22, 0) = val.range(22, 0);
    id.negative(sign.to_uint());
    id.exponent(exp.to_int());
    id.mantissa(mantissa.to_uint());
    return float(id);
}
```

Challenges I encounter

Simulation Time Bottleneck

One of the major challenges encountered during development was the long simulation time. Since each PE must individually load large amounts of weights, biases, and image data from ROM, the startup phase becomes extremely time-consuming. This issue is further exacerbated when the design is tested on large-scale CNN models involving multiple layers and thousands of parameters per layer.

To mitigate this, I modularized the initialization process and separated the parameter loading phase from the actual inference. This allowed me to verify the correctness of weight and bias transmission independently, without running the full inference flow. However, even with this optimization, the full-system simulation—especially when involving packet transmission across the NoC—still incurred significant time overhead. Reducing simulation time in future work may involve:

- Using smaller test cases with minimal data sizes during development.
- Introducing fast-forward or warm-start mechanisms.

- Selectively disabling parts of the network unrelated to the current test

Handshake Protocol

Getting the handshake protocol (***req***/***ack*** signaling) correct was one of the most critical aspects of the design. If the ***req*** signal is asserted too early or the ***ack*** is not raised in time, the data flit could either be lost or cause unnecessary stalling. I had to carefully coordinate the timing of these signals to ensure proper synchronization between modules. This was especially important when transferring flits between the Core and Router, where a mismatch could easily lead to errors in packet delivery.

Output Port Arbitration

Each router output port must be controlled by only one input port at a time to preserve flit ordering and avoid conflicts. To enforce this, I implemented an arbitration mechanism using ***output_busy*** and ***occupy_by_input***, which ensures that only one input can access an output until its entire packet has been transmitted. Once a tail flit is detected, the router releases the port, making it available for the next input. This design guarantees atomicity of packet transmission while avoiding partial delivery.

Deadlock and Congestion

One of the most important observations was the risk of ***deadlock*** in the absence of ***virtual channels*** (VCs). Without VCs, if a router's output path is blocked and all its input FIFOs fill up, the system can reach a deadlock state where no progress is made. This highlighted the need for VC-based designs in more advanced NoC implementations, especially under heavy or bursty traffic.

Transmitting multiple packets causing tail misalignment

When managing the ***tx_queue***—which writes data into ***flit_tx*** and pops elements when both ***ack_tx*** and ***req_tx*** are asserted—an important consideration during multi-packet transmission was that a *new header flit could only be issued once the* ***tx_queue*** *was completely empty*. This guarantees that the previous packet, including its tail flit, has been fully transmitted. Injecting a new header flit prematurely could *overwrite ongoing transmissions or violate routing assumptions*, potentially causing dropped packets or protocol errors.

To address this, a lightweight flow control mechanism was implemented: *transmission of a new packet is paused until **tx_queue** becomes empty*, indicating the prior packet is fully delivered. This ensures protocol correctness while avoiding the need for a complex state machine.

Observations and Insights

Several key insights were gained during the simulation and testing of the NoC system:

Small FIFO Depth Reveals Congestion: I initially experimented with increasing the FIFO size from 2 to 100. However, the Demo pattern consistently took around 140 cycles regardless of FIFO size. This indicated that buffer capacity was not the main bottleneck and congestion was more structural. The small depth (3) helped surface these issues and forced the design to handle realistic flow control conditions.

XY Routing is Simple but Imbalanced: The simplified XY routing strategy (prioritizing horizontal movement before vertical) was easy to implement, but it can lead to unbalanced traffic and bottlenecks in certain traffic patterns. This suggests that more adaptive routing algorithms might be necessary in future designs.

Flit Decoding is Critical: Accurately distinguishing between header, body, and tail flits based on the highest two bits of each flit ([1025:1024]) was vital for correct packet reconstruction in the PE. Without this, packets could be corrupted or misrouted.

Result

w/o optimized

Cat:

```
Top 100 classes:
Total latency: 61457157
================================================
  idx |      val | possibility | class name
------------------------------------------------
  285 |    20.21 |       96.38 | Egyptian cat
  281 |    16.14 |        1.65 | tabby
  282 |    15.73 |        1.10 | tiger cat
  287 |    14.79 |        0.43 | lynx
  728 |    14.41 |        0.29 | plastic bag
  330 |    12.73 |        0.05 | wood rabbit
```

FP/result_cat.log

Dog:

```
Top 100 classes:
Total latency: 61457157
================================================
  idx |      val | possibility | class name
------------------------------------------------
  207 |    16.59 |       38.63 | golden retriever
  175 |    15.57 |       13.86 | otterhound
  220 |    15.36 |       11.26 | Sussex spaniel
  163 |    15.00 |        7.86 | bloodhound
  219 |    14.59 |        5.22 | cocker spaniel
  168 |    14.39 |        4.28 | redbone
```

FP/result_dog.log

w/ optimized

Cat:

```
Top 100 classes:
Total latency: 1920682
================================================
  idx |      val | possibility | class name
------------------------------------------------
  285 |    20.21 |       96.38 | Egyptian cat
  281 |    16.14 |        1.65 | tabby
  282 |    15.73 |        1.10 | tiger cat
  287 |    14.79 |        0.43 | lynx
  728 |    14.41 |        0.29 | plastic bag
  330 |    12.73 |        0.05 | wood rabbit
```

FP_Optimized/result_cat.log

Dog:

```
Top 100 classes:
Total latency: 1920682
================================================
  idx |      val | possibility | class name
------------------------------------------------
  207 |    16.59 |       38.63 | golden retriever
  175 |    15.57 |       13.86 | otterhound
  220 |    15.36 |       11.26 | Sussex spaniel
  163 |    15.00 |        7.86 | bloodhound
  219 |    14.59 |        5.22 | cocker spaniel
  168 |    14.39 |        4.28 | redbone
```

FP_Optimized/result_dog.log