NYCU IEE
mlchip070

Mlchip HW3 report

413510028 林柏佑

Code Structure

```
├── clockreset.cpp
├── clockreset.h
├── core.h
├── main.cpp
├── pe.cpp
├── pe.h
└── router.h
```

Code Overview

In this lab, I implemented a 4x4 mesh Network-on-Chip (NoC) architecture. The system contains 16 Cores, each with a dedicated Processing Element (PE) and a connected Router. The routers are arranged in a 2D mesh topology, and each router has five ports: UP, DOWN, LEFT, RIGHT, and a local CORE port. Instead of transmitting entire packets directly, data is broken into smaller flow control units called flits. This enables more efficient channel usage and buffer control during transmission.

File Descriptions

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.

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.

main.cpp: signal connect between layers & module create

- Creation of 16 Cores 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.cpp 、***pe.h***: TA provider's check_packet get_packet ad PE's init

These files implement the behavior of the PE, which is responsible for generating packets and checking received packets. It provides two key functions:

- ***get_packet()*** – Generates a packet to be sent into the network.
- ***check_packet(Packet*)*** – Verifies the correctness of the received packet.

The PE is kept simple as it is mainly used for validating network functionality, based on the version provided by the TA.

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

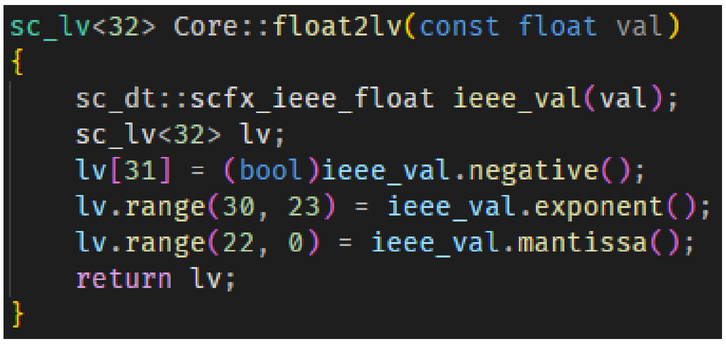
- ***Bits [33:32] = 10***: Header flag, indicates a header flit.
- ***Bits [33:32] = 01***: Tail flag, indicates a tail flit.
- ***Bits [31:0]: Payload*** – either a float number or address information.

This structure allows me to distinguish the flit type easily:

- A flit with bit 33 = 1 is the header flit, containing source and destination IDs.
- A flit with bit 32 = 1 (and bit 33 = 0) is the tail flit, marking the end of a packet.
- All others are body flits.

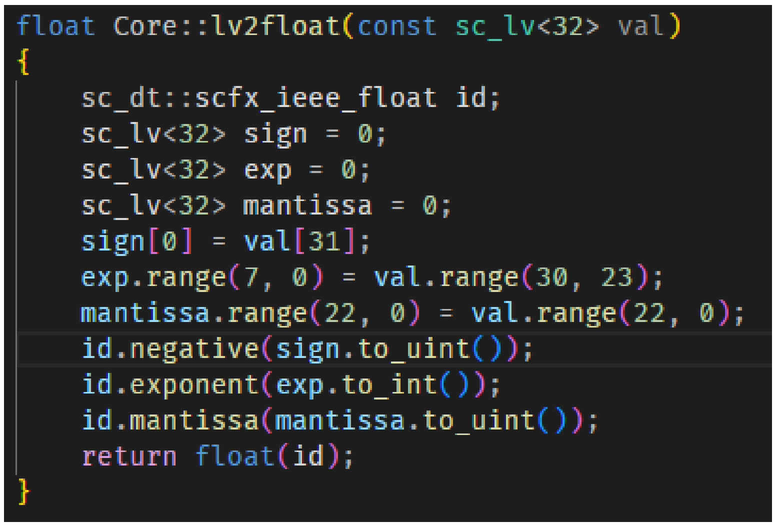
6. Float Conversion Functions

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

During the development of this 4x4 mesh NoC architecture, I encountered several key challenges related to synchronization, arbitration, and flow control between modules.

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.

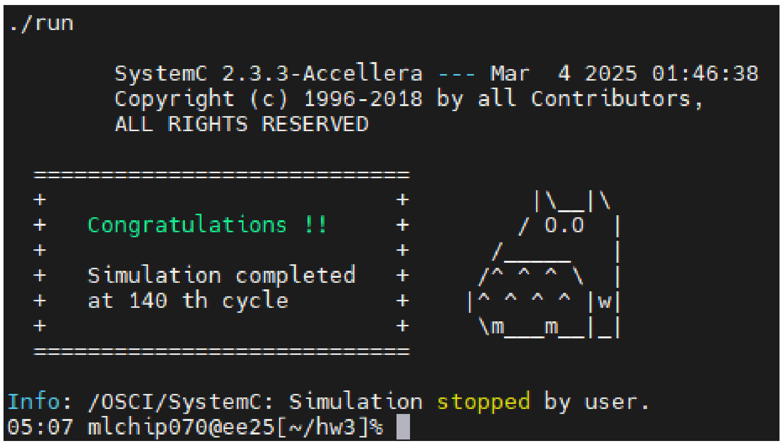
Observations and Insights

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 ([33:32]) was vital for correct packet reconstruction in the PE. Without this, packets could be corrupted or misrouted.

Result

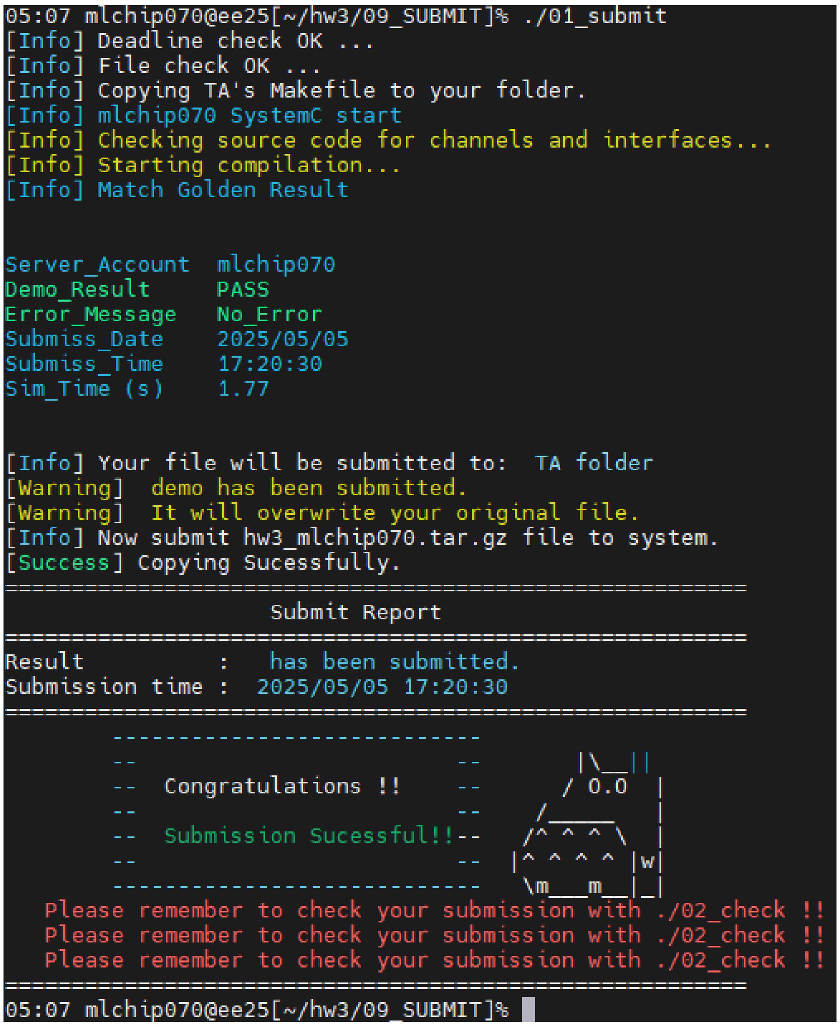


```
./run

        SystemC 2.3.3-Accellera --- Mar  4 2025 01:46:38
        Copyright (c) 1996-2018 by all Contributors,
        ALL RIGHTS RESERVED

  ============================
  +                          +          |\__|\
  +   Congratulations !!     +        / O.O  |
  +                          +       /_____  |
  +   Simulation completed   +      /^ ^ ^ \ |
  +   at 140 th cycle        +     |^ ^ ^ ^ |w|
  +                          +      \m___m__|_|
  ============================

Info: /OSCI/SystemC: Simulation stopped by user.
05:07 mlchip070@ee25[~/hw3]% 
```



```
05:07 mlchip070@ee25[~/hw3/09_SUBMIT]% ./01_submit
[Info] Deadline check OK ...
[Info] File check OK ...
[Info] Copying TA's Makefile to your folder.
[Info] mlchip070 SystemC start
[Info] Checking source code for channels and interfaces...
[Info] Starting compilation...
[Info] Match Golden Result


Server_Account  mlchip070
Demo_Result     PASS
Error_Message   No_Error
Submiss_Date    2025/05/05
Submiss_Time    17:20:30
Sim_Time (s)    1.77


[Info] Your file will be submitted to:  TA folder
[Warning]  demo has been submitted.
[Warning]  It will overwrite your original file.
[Info] Now submit hw3_mlchip070.tar.gz file to system.
[Success] Copying Sucessfully.
========================================================
                    Submit Report
========================================================
Result          :   has been submitted.
Submission time :  2025/05/05 17:20:30
========================================================
        --------------------------
        --                      --       |\__||
        --  Congratulations !!  --     / O.O  |
        --                      --    /_____  |
        --  Submission Sucessful!!--  /^ ^ ^ \ |
        --                      --   |^ ^ ^ ^ |w|
        --------------------------    \m___m__|_|
   Please remember to check your submission with ./02_check !!
   Please remember to check your submission with ./02_check !!
   Please remember to check your submission with ./02_check !!
========================================================
05:07 mlchip070@ee25[~/hw3/09_SUBMIT]% 
```