Qiao Gao
CWID: A20282211
ECE 587 Hardware/Software Co-Design
Project 2
12/02/2013

1 Introduction

In this project, a Butterfly Fourier Algorithm is implemented in to 3*3 NoC structure.

The challenge of this project focuses on:

1. Extend the origin design into a 3*3 NoC network.
2. Map all 14 the processes into 9 PE in total.
3. Evaluate the performance of the newly built system.

To solve these problems, methods below are used correspondingly:

1. Modify the module declaration and change the connection in top.cpp. Add y axis function to router.cpp.

2. Merge some of the functions of processes into one PE. Let the functions be determined by the coordinate of current PE and the source location of current.

3. Add counters to all the output queue to judge whether there is a stuck or not and decide the fastest rate of this system depending on the evaluate result.

All implementation details will be served in part 3.

2 System description

2.1 System overview

In this section, the function of this system and how it is implemented will be provided. The main function of this system is finishing an 8-bit Fourier transformation with the butterfly structure. More details will be provided below.

2.1.1 Fast Fourier Transform

Fast Fourier Transform algorithm provides a fast Fourier algorithm that is easy for hardware implementation. The DFG of FFT is shown below:

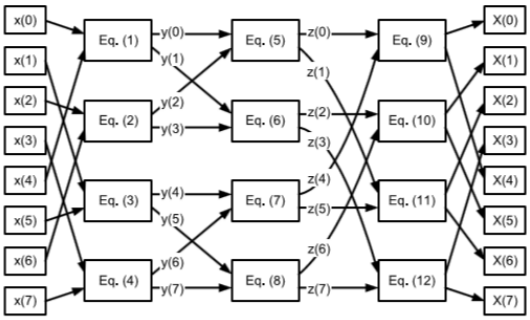

Figure1

In which the functions are:

$$
\begin{aligned}
y(0) &= x(0) + x(4), & y(1) &= x(0) - x(4); && (1)\\
y(2) &= x(2) + x(6), & y(3) &= x(2) - x(6); && (2)\\
y(4) &= x(1) + x(5), & y(5) &= x(1) - x(5); && (3)\\
y(6) &= x(3) + x(7), & y(7) &= x(3) - x(7); && (4)
\end{aligned}
$$

$$
\begin{aligned}
z(0) &= y(0) + y(2), & z(1) &= y(0) - y(2); && (5)\\
z(2) &= y(1) + y(3)w^2, & z(3) &= y(1) - y(3)w^2; && (6)\\
z(4) &= y(4) + y(6), & z(5) &= y(4) - y(6); && (7)\\
z(6) &= y(5) + y(7)w^2, & z(7) &= y(5) - y(7)w^2; && (8)
\end{aligned}
$$

$$
\begin{aligned}
X(0) &= z(0) + z(4), & X(4) &= z(0) - z(4); && (9)\\
X(1) &= z(2) + z(6)w, & X(5) &= z(2) - z(6)w; && (10)\\
X(2) &= z(1) + z(5)w^2, & X(6) &= z(1) - z(5)w^2; && (11)\\
X(3) &= z(3) + z(7)w^3, & X(7) &= z(3) - z(7)w^3; && (12)
\end{aligned}
$$

Figure 2

Including input and output process, there should be 14 processes in total: PI, PO, P1, P2, P3 P4,

P5, P6, P7, P8, P9, P10, P11 and P12. In this design, I build three types of PEs, which are: PE_IO, PE1 and PE2. The functions of them are described below:

PE_IO: Generate 8-bit complex number randomly in one cycle, print output when it is ready.

PE1: Bind the function of P5-P6.

PE2: Bind the function of P1-P4 or P9-P12 depending on where all operands come from.

2.1.2 Mapping strategy

Our task is to mapping the processes to the structure below:

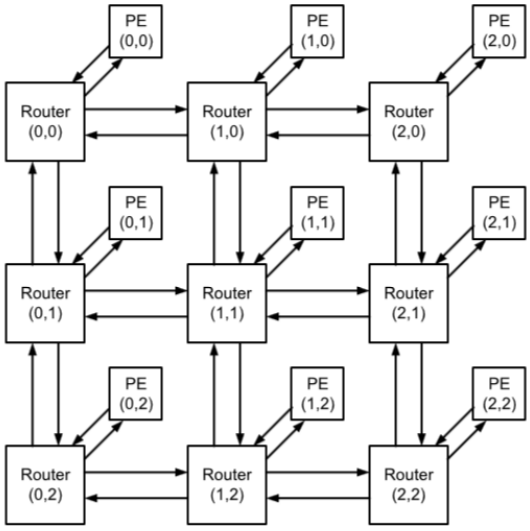

Figure 3

Then the figure below shows my strategy on this mapping:

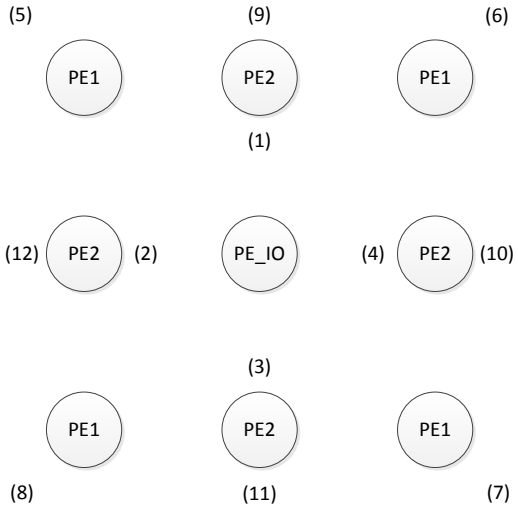

Figure 4

2.2 Module Implementation

2.2.1 Routers

Add y axis route function to the router.cpp, using the code below:

```
if (p.dest_y == y_) // to PE
{
    if (p.dest_x == x_) // to PE
     {
        out_queue_[PE].push_back(p);
     }
    else if (p.dest_x < x_) // left to WEST
     {
        out_queue_[WEST].push_back(p);
     }
    else // (p.dest_x > x_) right to EAST
    {
        out_queue_[EAST].push_back(p);
    }
}
else if (p.dest_y < y_) // left to NORTH
{
    out_queue_[NORTH].push_back(p);
}
else // (p.dest_x > x_) right to SOUTH
{
    out_queue_[SOUTH].push_back(p);
}
```

Figure 5

```
for (int j = 0; j < N; ++j)
{

    for (int i = 0; i < N; ++i)
    {
        char name[100];
        sprintf(name, "router%d%d", i,j);

        // create router
        routers[i][j] = new router(name);
        routers[i][j]->set_xy(i, j);
        routers[i][j]->clock(clock);


        // connect router to north routers
        if (j != 0)
        {
            routers[i][j]->port_out[router::NORTH](
                router_to_router_north[i][j-1]);
            routers[i][j]->port_in[router::NORTH](
                router_to_router_south[i][j-1]);
        }
        else // or make a loop
        {
            routers[i][j]->port_out[router::NORTH](
                terminal_loop_north[i]);
            routers[i][j]->port_in[router::NORTH](
                terminal_loop_north[i]);
        }

        if (j != N - 1) // connect router to south routers
        {
```

Figure 6

Then we can make 2D assign to our design. The declaration part in top.cpp is shown in Figure 6.

Keep doing this until all ports connections of all routers are settled. Moreover, modify the packet information with two complex double numbers to perform a complex number:

```
typedef double token_type;
struct packet
{
    int src_x, src_y;
    int dest_x, dest_y;
    token_type real;
    token_type image;

    packet(int sx = -1, int sy = -1, int dx = -1, int dy = -1,
        token_type t1 = token_type(),token_type t2 = token_type())
        : src_x(sx), src_y(sy), dest_x(dx), dest_y(dy), real(t1), image(t2)
    {
    }
```

Figure 7

2.2.2 PEs

As I have declared in 2.1. There are three kinds of PEs with the function:

PE_IO: Generate 8-bit complex number randomly in one cycle, print output when it is ready.

PE1: Bind the function of P5-P6.

PE2: Bind the function of P1-P4 or P9-P12 depending on where all operands come from.

The exact function specification is complicated, so it will be shortly described in part 3. The declaration in top.cpp can be:

```
void create_pes()
{
    pes[0][0] = new PE_1("P00");
    pes[0][0]->clock(clock);
    pes[0][0]->set_xy(0, 0);

    pes[1][0] = new PE_2("P10");
    pes[1][0]->clock(clock);
    pes[1][0]->set_xy(1, 0);

    pes[2][0] = new PE_1("P20");
    pes[2][0]->clock(clock);
    pes[2][0]->set_xy(2, 0);

    pes[0][1] = new PE_2("01");
    pes[0][1]->clock(clock);
    pes[0][1]->set_xy(0, 1);

    pes[1][1] = new PE_IO("PI/PO"
    pes[1][1]->clock(clock);
    pes[1][1]->set_xy(1, 1);

    pes[2][1] = new PE_2("P21");
    pes[2][1]->clock(clock);
    pes[2][1]->set_xy(2, 1);

    pes[0][2] = new PE_1("P02");
    pes[0][2]->clock(clock);
    pes[0][2]->set_xy(0, 2);

    pes[1][2] = new PE_2("P12");
    pes[1][2]->clock(clock);
    pes[1][2]->set_xy(1, 2);

    pes[2][2] = new PE_1("P22");
    pes[2][2]->clock(clock);
    pes[2][2]->set_xy(2, 2);
}
```

Figure7

This mapping is exactly the same with what is shown in Figure 4.

3 Implementation Details

3.1 PE implementation

Before introducing PEs, I have to claim some new things that I involved in this design. Since the routers and PEs can just read one packet from each port and PEs need more than one operands all the time, there must be an area of storage to store the valid inputs when the inputs are not sufficient. Then I declare an input queue for every PE to make the storage. For example:

```
class PE_IO : public PE_base
{
public:
    PE_IO(const char *name) : PE_base(name) {}

protected:
    std::list<packet> in_queue_;
    void execute();
```

Figure 8

Note that I claim two input queues for PE2 because it should handle two types of function separately.

```
class PE_2 : public PE_base
{
public:
    PE_2(const char *name) : PE_base(name) {}

protected:
    std::list<packet> in_queue_[2];
```

Figure 9

The input should be saved in this queue for every PE, so include the following code at the start of every PE:

```
void PE_IO::execute()
{
    if ((packet_in_.src_x != -1)
        && (packet_in_.src_y != -1))
        in_queue_.push_back(packet_in_);
```

Figure 10

3.1.1 PE_IO

In this PE, eight random complex numbers are generated at a certain possibility. Then they queue in the PE output queue one by one. The destinations of PE_IO are also declared here:

```
void PE_IO::fire_PI()
{
    int P_x[8] = { 1, 0, 1, 2, 1, 0, 1, 2 }, P_y[8] = { 0, 1, 2, 1, 0, 1, 2, 1};
    for (int i = 0; i < 8; ++i)
    {
        packet p(x_, y_, P_x[i], P_y[i], rand()%100, rand()%100);

    printf("PI: send %6.2f+(%6.2f)j to (%d,%d)\n",
        p.real, p.image, p.dest_x, p.dest_y);

    out_queue_.push_back(p);
    }
}
```

Figure 11

For output, just keep an eye on the size of input queue. Once it is eight, pop up everything inside.

```
assert(in_queue_.size()==8);
printf("OUTPUT:\n");//print the output
for (int i = 0; i < 8; ++i)
{
    printf("%6.2f+(%6.2f)j\n", in_queue_.front().real, in_queue_.front().image);
    in_queue_.pop_front();
}
```

Figure 12

3.1.2 PE1 and PE2: execution issue

For the execution of PE1 and PE2, I notice that there are only four kings of calculations:

(1) a+b, a-b

(2) a+bw, a-bw
(3) $a+bw^2$,$a-bw^2$
(4) $a+bw^3$,$a-bw^3$

I use a sign number to stand for these functions, which is “func”. Since all inputs are complex numbers and w=0.707-0.707j, finally I got the general function below:

```
switch (func)
{
case 0: output_real[0] = input_real[0] + input_real[1];
    output_image[0] = input_image[0] + input_image[1];
    output_real[1] = input_real[0] - input_real[1];
    output_image[1] = input_image[0] - input_image[1];
    break;
case 1: output_real[0] = input_real[0] + (input_real[1] + input_image[1])*0.707;
    output_image[0] = input_image[0] + (input_image[1] - input_real[1])*0.707;
    output_real[1] = input_real[0] - (input_real[1] + input_image[1])*0.707;
    output_image[1] = input_image[0] - (input_image[1] - input_real[1])*0.707;
    break;
case 2: output_real[0] = input_real[0] + input_image[1];
    output_image[0] = input_image[0] - input_real[1];
    output_real[1] = input_real[0] - input_image[1];
    output_image[1] = input_image[0] + input_real[1];
    break;
case 3: output_real[0] = input_real[0] + (input_image[1] - input_real[1])*0.707;
    output_image[0] = input_image[0] - (input_image[1] + input_real[1])*0.707;
    output_real[1] = input_real[0] - (input_image[1] - input_real[1])*0.707;
    output_image[1] = input_image[0] + (input_image[1] + input_real[1])*0.707;
    break;
default: break;
}
```

Figure 13

Then the next job is just to determine the function number at the beginning of each PE.

3.1.3 PE1 and PE2: operands issue

Operands are come from the input queue when the queue size is two. Specially, for PE2, which has two input queue, the operands come from the queue which has the size two first. As the operands come in one by one, these two queues cannot become size two at same time. For PE2, the input queue sizes also affect function and destinations define.

3.1.4 PE1 and PE2: destination mapping

The destination mapping is depended on the location of the PE, the source location of the operands and the input queue size. Here I provide the example for PE2. In PE2, the input queue sizes indicate the input source type. So the destination and function is determined by queue sizes first. Then the location of PE is taken into consideration. The example is shown below:

```
if (in_queue_[0].size() == 2)
{
    queue = 0;
    func = 0;
    if (x_ == 1)
    {
        if (y_ == 0)
        {
            PO1_x = 0; PO1_y = 0;
            PO2_x = 2; PO2_y = 0;
        }
        else
        {
            PO1_x = 2; PO1_y = 2;
            PO2_x = 0; PO2_y = 2;
        }
    }
    else if (x_ == 0)
    {
        PO1_x = 0; PO1_y = 0;
        PO2_x = 2; PO2_y = 0;
    }
    else
    {
        PO1_x = 2; PO1_y = 2;
        PO2_x = 0; PO2_y = 2;
    }
}
else
{
    queue = 1;
    PO1_x = 1; PO1_y = 1;
    PO2_x = 1; PO2_y = 1;
    if (x_ == 1)
    {
        if (y_ == 0)
        {
            func = 0;
        }
        else
        {
            func = 2;
        }
    }
    else if (x_ == 0)
    {
        func = 3;
    }
    else
    {
        func = 1;
    }
}
```

Figure 14

3.2 Evaluation method

The key point of the evaluation is to get the queue size conditions in order to determine the maximum throughput. Unlike what have been done in project 1, I think there is no need to observe

queue sizes of all units. Actually, I just keep an eye one the most critical queue, which is the output queue of PE_IO since it produces eight packets in the queue in just one cycle. This one is so critical that whenever there is a stuck in any place in this design, the problem will reflect on this queue automatically. So just let the PE_IO print its output queue size every cycle. The follow sentence is located at the end of PE_IO::execute.

```
printf("Critical Queue Size: %d\n", (unsigned)out_queue_.size());
```

To generate the average delay, we need a cycle count, finished number counter and a delay accumulation. So in top.cpp, I define:

```
extern int cycle, finished, total_delay;
```

And then initialize them in pe.cpp:

```
int cycle=0, finished=0, total_delay=0;
```

Moreover, I need a queue to store the start point in PE_IO. So in pe.h:

```
class PE_IO : public PE_base
{
public:
    PE_IO(const char *name) : PE_base(name) {}

protected:
    std::list<int> start;
    std::list<packet> in_queue_;
```

Finally modify PE_IO::execute:

```
void PE_IO::execute()
{
    if ((packet_in_.src_x != -1)
        && (packet_in_.src_y != -1))
        in_queue_.push_back(packet_in_);
    cycle++;
    int r = rand()%100;
    int delay;
    if (r < 12)
    {
        fire_PI();
        start.push_back(cycle);
    }

    if (in_queue_.size() == 8)
    {
        fire_PO();
        finished++;
        delay = cycle - start.front();
        total_delay = total_delay + delay;
        printf("Average Delay: %d Delay: %d Finish: %d\n",
            total_delay/finished, delay, finished);
        start.pop_front();
    }
    printf("Critical Queue Size: %d\n", (unsigned)out_queue_.size());
}
```

4 Result Analysis

Firstly, we have to determine the maximum throughput. Just let this system run 2000 cycles, which is a very big number of executions. If no queue accumulation at final cycle, then this system runs correctly. Keep modifying output possibility to find this throughput:

Final cycle status when p=10% and 80%:

```
cycle 1999 ================================
OUTPUT:
424.00+(452.00)j
 36.00+(-118.00)j
 21.00+(  1.00)j
135.00+(-19.00)j
-110.48+( 21.95)j
 46.48+( 12.05)j
 35.50+(-34.57)j
 -5.50+(-61.43)j
Average Delay: 38 Delay: 42 Finish: 204
Critical Queue Size: 4
```

```
cycle 1999 ================================
PE(1,0): receive  67.00+( 18.00)j from (1,1)
PE(1,0): receive   4.00+( 18.00)j from (1,1)
send  71.00+( 36.00)j to (0,0)
send  63.00+(  0.00)j to (2,0)
PI: send   9.00+( 36.00)j to (1,0)
PI: send  47.00+( 60.00)j to (0,1)
PI: send  46.00+( 97.00)j to (1,2)
PI: send  51.00+( 62.00)j to (2,1)
PI: send  57.00+( 18.00)j to (1,0)
PI: send  51.00+( 62.00)j to (0,1)
PI: send  43.00+( 64.00)j to (1,2)
PI: send  32.00+( 39.00)j to (2,1)
Critical Queue Size: 10961
```

Figure 15

Keep trying and find that at p=12% and 13%:

```
cycle 1999 ================================
PE(1,0): receive 198.00+(189.00)j from (0,0)
PE(1,0): receive  89.00+(251.00)j from (2,2)
send 287.00+(440.00)j to (1,1)
send 109.00+(-62.00)j to (1,1)
Critical Queue Size: 14
PE(2,1): receive  32.00+( 11.00)j from (1,1)
PE(2,1): receive   5.00+( 62.00)j from (1,1)
send  37.00+( 73.00)j to (2,2)
send  27.00+(-51.00)j to (0,2)
```

```
cycle 1999 ================================
PE(0,0): receive 123.00+( 85.00)j from (1,0)
PE(0,0): receive  75.00+(104.00)j from (0,1)
send 198.00+(189.00)j to (1,0)
send  48.00+(-19.00)j to (1,2)
Critical Queue Size: 165
PE(1,2): receive -30.00+(-58.00)j from (2,2)
PE(1,2): receive  81.00+(-33.00)j from (0,0)
send -63.00+(-139.00)j to (1,1)
send   3.00+( 23.00)j to (1,1)
```

Figure 15

The critical queue size changed a lot. So the maximum output possibility should be **12%**.

Secondly, to get the minimum latency, we can just keep an eye on the report in first finish cycle. When p=12%, I got:

```
cycle 39 ================================
OUTPUT:
427.00+(490.00)j
-29.00+(-70.00)j
 14.00+(-221.00)j
-38.00+( 69.00)j
 -9.62+(-140.91)j
-46.38+( 42.91)j
 26.91+( 35.78)j
 65.09+( 20.22)j
Average Delay: 28 Delay: 28 Finish: 1
```

Figure 16

So the minimum latency should be **28 cycles**.

Finally, get the average latency at the report of last finish cycle among all 2000 cycles. This value should be steadier when the cycle number goes up. When p=12%, I got:

```
cycle 1993 ================================
OUTPUT:
368.00+(425.00)j
 -2.00+(-47.00)j
152.00+( 85.00)j
170.00+(-133.00)j
 27.88+( 90.61)j
 32.12+( 69.39)j
-12.03+(-36.22)j
-45.97+( 88.22)j
Average Delay: 61 Delay: 35 Finish: 235
Critical Queue Size: 12
```

```
cycle 1985 ================================
OUTPUT:
416.00+(308.00)j
 -6.00+(-44.00)j
 13.00+(-37.00)j
-47.00+(-43.00)j
-89.06+(-72.85)j
 65.06+( 38.85)j
 24.05+(144.14)j
 33.95+(-188.14)j
Average Delay: 61 Delay: 28 Finish: 234
Critical Queue Size: 12
```

```
cycle 1962 ================================
OUTPUT:
476.00+(372.00)j
-87.00+(-17.00)j
-146.00+(  8.00)j
  5.00+(109.00)j
-57.23+(-12.74)j
 93.71+(-45.57)j
  9.23+(-45.26)j
 10.29+(-72.43)j
Average Delay: 61 Delay: 41 Finish: 233
Critical Queue Size: 19
```

It is clear that the average latency in this case should be **61 cycles**.