A Flexible High Speed Star Network Based on Peer to Peer Links on FPGA

Chao Wang¹, Junneng Zhang¹, Xuehai Zhou², Xiaojing Feng¹, Aili Wang²

¹ School of Computer Science
University of Science and Technology of China
Hefei, Anhui, China.
{saintwe, zjneng, bangyan}@mail.ustc.edu.cn

² Suzhou Institute of USTC
University of Science and Technology of China
Suzhou, Jiangsu, China
{xhzhou, wangal}@ustc.edu.cn

Abstract

Multi-Processor System on Chip (MPSoC) platform plays a vital role in parallel processor architecture design. However, it poses a great challenge to design a flexible high-speed network regarding as the growing number of processors. This paper proposes a star network based on peer to peer links on FPGA. The star network uses fast simplex links (FSL) for demonstration to connect scheduler and processing elements, including processors and hardware IP cores. Blocking and non-blocking applications interfaces are provided to users for programming. We built a prototype system on FPGA to evaluate the transfer time and hardware costs of the star network architectures. Experiment results shows the average transfer time for each word can be reduced to 7 cycles at least. Moreover, the star network architecture costs only 1.2% Flip Flops and 2.45% LUTs of the whole prototype MPSoC system.

Key words: MPSoC, Star Network, Network on Chip, Programming Interfaces, FPGA

1. Introduction

Multi-core technology has been widely applied in modern mainstream processor products. Since more and more processors are being integrated into single chip, MPSoC can provide increasingly speedups to various applications. However, interconnection and communication through processors is still regarded as a bottleneck in MPSoC architecture design [1]. Since traditional bus-based architectures are obviously non-scalable to fit the continuously increasing number of processor cores, new interconnects on chip need to be proposed. Networks on Chip (NoC), which provides better structure and modularity, has been an emerging paradigm to design scalable on-chip communication architectures. Although NoCs solve the interconnect scalability issues, their integration with the processing cores is still posing a serious challenge for system architects and system integrators in particular [2, 3]. As the system complexity grows, the problem of how to design a flexible and high speed NoC-based MPSoC platform is becoming more and more challenging. Until now the problem is still not been completely figured out.

In order to build flexible embedded MPSoC system, reconfiguration technologies have been demonstrated as an effective way, especially in application-specific systems. As embedded software applications are quite different from each other, run-time reconfiguration is becoming increasingly more important to adapt hardware to applications. For embedded devices, it is desirable to reuse the same hardware for different applications at run-time. Reconfiguration technology also allows the embedded hardware system to run new-designed applications after fabrication. Consequently, since hardware is reused, embedded system design time and space costs are largely decreased and TTM is significantly shortened. For example, Reconfiguration technologies in NoC designs can largely increase the flexibility of embedded system interconnect, benefiting the run-time adaptation of logics and data path interconnection. Therefore with the help of reconfigurable computing technologies, MPSoC can reconstruct hardware platform to fit applications.

In this paper we present a flexible high speed star interconnect architecture based on peer to peer links in FPGA. By utilizing FSL high speed bus channels and programming interfaces for demonstration, this approach can obtain high efficiency. Additionally, if computing resources are changed in the platform, star network can be easily reconfigured. Therefore this solution can achieve both flexibility and great performances.

The rest of the paper is organized as follows. We outline the related work in section 2. Section 3 discusses about the star network architecture and implementation. Experiment setup and data analysis is presented in section 4. Section 5 summarizes the paper.
2. Related Work
Since interconnections between processors have a great impact on performance, a variety of studies have already been proposed. The most commonly used interconnect architectures are as follows: on-chip bus, hierarchical bus, crossbar and mesh network, etc. Of these interconnections, bus-based architectures are still quite efficient in system on chip designs; cross-bar and mesh architectures can be configured to obtain high flexibility. For example, [4] and [5] use adaptive NOC interconnect to organize reconfigurable on chip architectures; [6] uses FLEXIO and SERDES high speed bus interfaces between CELL processor and the FPGA. However, bus based architecture is difficult to be modified after fabrication, which brings significant challenges to developers in flexible architecture design. In contrast, although cross-bar and mesh can get high flexibility, they take much more hardware resources. Therefore, new approaches need to be proposed to achieve both performance and flexibility.

Currently reconfigurable technologies are quite popular and common used in NOC researches [7]. [8] proposes run-time adaptive on-chip communication scheme, which provides an adaptive routing/path allocation algorithm to meet a required bandwidth. Unfortunately it only modifies buffer allocation and left communication channels unchanged. [9] introduces a hybrid communication reconfigurable network on chip for MPSoC, which uses a TDMA shared bus for cluster communication. [10] presents a reconfigurable multi-processor which is quite feasible for FPGAs, but the paper focuses on design flow and software tool chains. [11, 12] states a dynamically reconfigurable network for reconfigurable MPSoC, but the approach is only carried out in NS-2 simulator. FPGA based approaches [13] are also quite popular because of FPGA can be easily reconfigured. Of these studies, [14] refers FPGA design flow to provide MPSoC generation from high level descriptions. An approach of application mapping on FPGA is proposed in [15]. However, this work uses general processors and interconnections which cannot be adapted to applications.

In order to solve the above problems, this paper proposes flexible star network architecture on FPGA. By using star network architecture, the peer to peer links can be easily added or removed, so the architecture can both achieve high flexibility and great performances.

3. Architecture and Concepts
3.1. Star Network Architecture
In our proposed architecture, a scheduler and several processing elements organize a star network. Scheduler receives requests of tasks from users at first, and then maps and distributes the tasks to processing elements. There are two types of processing elements: embedded processors and hardware intellectual property (IP) cores.

(1) Embedded processors provide running environment to diverse software computing tasks. Each processor has a copy of task library which includes all the functions. Each function is called through individual application programming interfaces.

(2) Hardware IP cores: Every IP core can only deal with a specific task to achieve hardware acceleration. Generally MPSoC platform consists of a variety of heterogeneous hardware IP cores. In addition, the property of IP cores can be redesigned or reconfigured according to different application demands.

Figure 1 shows the star network architecture. Taking advantage of flexible and reconfigurable characteristic of FPGA, the architecture employs star network for on-chip connection between scheduler and processing elements. Each computing processor is connected to scheduler through a peer to peer link.

The communication data paths between different processing elements can be reconfigured. In this paper no direct paths among those processing elements are proposed. Scheduler distributes tasks and related data to different processing elements directly. Scheduler also provides application programming function interfaces to users. Every function is bound to a type of software functions or hardware modules. Scheduler runs an online scheduling algorithm to detect data dependences automatically. If the tasks are isolated from each other, they will be sent to different processing elements in parallel. However, if different tasks running on different processors require data from each other, these tasks must be issued serially. Scheduler will wait to send request until previous results are ready.

3.2. FSL based Organization
3.2.1. FSL Bus
Our proposed star network architecture is based on peer to peer bus channels. For demonstration we
choose Xilinx fast simplex link (FSL) [16] bus as the basic interconnection links. The message passing interfaces through FSL for data exchange between processors are used for data communication.

Figure 2 FSL Bus Signals[16]

Figure 2 shows the principle of the FSL bus system and the available signals. FSL bus is a one-way FIFO-based peer to peer communication bus, which is especially utilized for communication between modules in FPGA. FSL bus is driven by one master module and drives one slave module. Also, FSL contains a FIFO buffer for temporary data store when data comes serially. Xilinx EDK provides a set of macros for reading and writing interfaces through FSL link. FSL data path is directly connected to the registers of microblaze processor so data can directly transferred using microblaze instructions.

Since FSL is one-way channel, therefore each processing element is connected with scheduler through a pair of FSL links. Scheduler plays as master module when task is dispatched. After execution is finished, processing elements act as master modules to drive the bus.

Because all the interconnection channels use FSL as the data path links, every processing element is packaged in FSL manner to keep the consistent interfaces with FSL bus channel.

3.2.2. FSL Based Interfaces

In this section, we describe FSL manner packages. Since there are two kinds of computing resources including embedded processors and hardware IP cores, so we must implement software interfaces for processors and hardware interfaces for IP cores.

(1) Hardware Interfaces

Before each IP core is packaged into FSL manner, the behavior functionality and timing model for every IP core must be completely designed and fully testified (By using ISE or Modelsim design tools). After that, the whole hardware package and conversion process can be divided into three steps (the following description is in VHDL language for demonstration):

1. Declaration of components: every IP is regarded as a component in the packaged FSL slave module. A sample template of component declaration is shown in Figure 3. All the signals used inside the IP cores for execution are declared.

ARCHITECTURE behavior OF fsl_s_enc IS
COMPONENT fsl_s_enc
PORT
DATA_I : IN std_logic_vector(7 downto 0);
VALID_DATA_I : IN std_logic;
KEY_I : IN std_logic_vector(7 downto 0);
VALID_KEY_I : IN std_logic;
RESET_I : IN std_logic;
CLK_I : IN std_logic;
CE_I : IN std_logic;
KEY_READY_O : OUT std_logic;
VALID_o : OUT std_logic;
DATA_o : OUT std_logic_vector(7 downto 0)
);
END COMPONENT;

Figure 3 FSL Component Sample

2. Define FSL entity. The FSL module itself will be defined as an entity. All the ports are defined in consistent with FSL signals. When new data is coming from ports, values of ports are stored in local buffer. After all the data is ready, temporary data is assigned to the signals inside the component during execution. The timing is controlled by state machine.

3. Implement state machine. In order to ensure the original function can get correct results with the delay of FSL, input data are buffered before execution. Moreover, results must be collected before sent to the FSL bus. Therefore, a state machine must be designed.

The whole process is divided into four phases: idle, read, execution and write. In idle state, IP core keeps waiting for data from the FSL bus; Read phase is responsible for reading from the FIFO buffer in FSL links and store data into buffer; in execution phase IP core completes the execution, and then writes the results to output buffer; write phase is responsible for sending data from output buffer to FSL bus. Figure 5 shows the state machine transition diagram.

Figure 4 FSL Entity Sample

Figure 5 State Machine
(2) Software Interface

Compared to hardware, software interface is provided by function library running on microblaze processors.

Unlike to hardware IP cores, embedded processor can deal with diverse types of computing tasks. Therefore, which type of service is requested by user must be specified before execution. In our proposed architecture, each type of request is assigned with a specific ID. ID must be sent before input data and output results are transferred.

(3) Task Scheduling

After scheduler receives task sets from user, it divides the tasks into different sub tasks at first, and then decides which task runs on which processing element. After the choice is made, each sub task is transferred to certain processor or IP core through FSL.

Obviously, task scheduling and mapping algorithm running on scheduler has great impact on performance. In this paper, we designed a quite simple scheduling method for demonstration: if the hardware block is free, task will be transferred to hardware blocks. Or else, an embedded processor will be scheduled to run the application. If all the resources are busy, the task has to wait being scheduled.

The process is worked as follows:
1) On receiving the request, scheduler firstly decides the task in running on either processors or IP cores.
2) If task is scheduled to software, request ID is sent through FSL links to identify which type of functionality is requested, or else, this step is skipped.
3) Scheduler sends data to processing elements. Once all data are ready, execution will be started automatically.
4) After results are obtained, processor elements transmit the results back to scheduler. Similarly, before data is transferred by processor, request ID must be sent to indicate the slave module.
5) When results are received by scheduler, an interrupt is triggered to deal with the results. Interrupt handler will continue application execution after the results returned to the user.

3.2.3. Programming Interfaces

The architecture provides both blocking and non-blocking programming interfaces in different functions. Function name indicates the target request task functions and parameters are regarded as the input/output data for tasks. Totally there are two ways to invoke FSL access instructions: blocking or non-blocking. Both types of interfaces are provided by programming macros.

On one hand, for blocking interface, the requests run after current request must stall until results transferred back from the processing elements. In this situation, applications are executed serially. At one time, only one processing element can stay in busy state at most.

On the other hand, non-blocking interface provides an asynchronous mechanism to improve data parallelism. After scheduler sends request to processing elements, it can continue to execute the after programs. More requests can be continuously issued to different computing resources if there are available resources. After calculation is finished, the results will be returned the scheduler processor as an interrupt. Scheduler will break the current operation to handle the interrupt. After the value of output variable is updated, application will go back for execution.

Of these two kinds of interfaces, apparently non-blocking interface can make full use of different processors to improve the data parallelism, so we use non-blocking interfaces in our programming models.

4. Experiment

4.1. Experiment Setup

In order to demonstrate the star network, a prototype system is built on Digilent XUPV5 board with Xilinx XC5VLX110T FPGA. We use microblaze version 7.20.a (with the clock frequency 125MHz, local memory of 8KB) as processor. Xilinx ISE Design Suite 11.3 is used to set up the experiment.

Star network architecture consists of following components:
1. Scheduler is implemented in microblaze. The scheduling and mapping algorithm running on scheduler are implemented in software.
2. Computing processors are also constructed in microblaze. All the functions are designed into C library. Application programming interfaces for every function are provided to users.
3. Hardware processing elements are implemented in VHDL and packaged as standalone IP cores.

4.2. Transfer Time Results and Analysis

In order to measure the transfer time of FSL links, applications transferring data from 4 32-bit words to 128 words are designed. Since there are two types of data transfer of FSL links: direct transfer and loop transfer. We first measured the transfer time use direct transfer macros, and then used loops to transfer data through FSL links.

1) Direct transfer

In this case, data is transferred by invoking the macro for each word directly, no control loops are used. Figure 6 shows the direct transfer time through FSL links. As the number of transfer data grows from 4 to 128, the transfer time increases from 103 cycles to 969
cycles. Based on the results, we can also calculate the average transfer time for each word degrades from 25 cycles to 7 cycles. This is because there are still some glue functions such as time acquisition function needs several cycles (e.g. 74 cycles for get_current_time()), the influence of these functions can be ignored when data volume is big enough.

Figure 6 Direct Transfer Time through FSL

2) Loop Transfer
As there are no control loops included in direct transfer, so the delay time is quite low. Direct transfer can be used in low-volume data transmission. However, it is not feasible to call FSL transfer macro for each word when data size is big, because it will largely increase the program code size.

For transferring a group of input data, control loop must be used to control the access to each element. FSL transfer instruction is called only once in each iteration. In this case, lots of cycles will be spent in the control loop code.

Figure 7 Loop Transfer Time through FSL

Figure 7 shows the results of the loop transfer. Similar to the direct transfer, the average transfer time for each data decreases from 79 cycles to 52 cycles. The time is 3 to 7 times larger than direct transfer because of the control loops.

3) Comparison with application execution
In order to evaluate the influence of transfer delay, we ported AES encrypt and decrypt (32 words) software applications of EEMBC testbench and implemented AES IP cores. Applications are designed in loop transfer manner since it’s more convenient and widely used.

From the simulation and analysis of hardware IP core, standard AES core module concludes of four different phases: read key (16 cycles), key expansion (133 cycles), read data (16 cycles) and data encoding/decoding (99 cycles).

\[ T_{exec} = T_{read-key} + T_{key-expan} + T_{read-data} + T_{enc/dec} = 264 \text{ cycles} \]

We can get the execution of hardware takes 264 cycles. Moreover, since we packaged the AES core into FSL slave modules, which brings \( 16 \times 3 = 48 \) extra cycles for data buffering. So the total hardware time is:

\[ T_{hardware} = T_{exec} + T_{package} = 312 \text{ cycles} \]

In loop transfer situation, theoretical total time is calculated as follows. (32 words \( \times \) 52 cycles/word)

\[ T_{loop-transfer} = 52 \times 32 = 1664 \text{ cycles} \]

\[ T_{loop-total} = T_{loop-transfer} + T_{hardware} = 1976 \text{ cycles} \]

We demonstrate both of the hardware version in IP blocks and software version in microblaze processors.

Table 1. AES IP Core Execution Time

<table>
<thead>
<tr>
<th>Type</th>
<th>Hardware(cycles)</th>
<th>Software(cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES_ENC</td>
<td>1910</td>
<td>596259</td>
</tr>
<tr>
<td>AES_DEC</td>
<td>1922</td>
<td>595767</td>
</tr>
</tbody>
</table>

The transfer delay and execution time is shown in Table 1. Experiments demonstrate the accuracy of AES_ENC and AES_DEC experiment is 96.7% and 97.3%.

From the results, we can also get the time cost percent of transfer delay. On one hand, as software applications take extremely long time, the FSL delay only takes 0.3% of the total software running time. On the other hand, in hardware version, the total transfer time takes 84.2% of the total time. If direct transfer is used, the percent can be reduced to 41.8%.

\[ T_{direct-total} = T_{direct-transfer} + T_{hardware} = 536 \text{ cycles} \]

\[ P_{direct} = \frac{T_{direct-transfer}}{T_{direct-total}} \times 100\% = 41.8\% \]

Average transfer time only relates to the application interfaces and has nothing to do with the complexity of the application. If a more complex hardware IP is integrated, or task is scheduled in software, time cost percent of transfer delay is much smaller.

4.3 Hardware Implementation Overheads
In order to measure the hardware cost of FSL links, we evaluated the Flip Flops and Look up Table (LUT) used in FPGA for different FIFO depths.

Figure 8 shows as the depth of the FIFO increases, hardware resources for each FSL link grows from 7 Flip Flops and 44 LUTs to 23 Flip Flops and 172 LUTs. The lower bound of FIFO size is 16.

We also measured the hardware cost of a prototype MPSoC system using FSL links. The system includes
one scheduler processor, one computing processor and four heterogeneous hardware IP cores including Adder, IDCT, AES_ENC and AES_DEC in FPGA. 5 pairs of FSL links are utilized for star network. Hardware cost of flip flops and LUTs are shown in Table 2.

![Figure 8 Hardware Costs of FSL Links](image)

Table 2. Hardware Cost of MPSoC System

<table>
<thead>
<tr>
<th></th>
<th>Flip Flops</th>
<th>LUTs</th>
<th>BRAMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microblaze</td>
<td>2500</td>
<td>2148</td>
<td>0</td>
</tr>
<tr>
<td>BRAM Cache</td>
<td>12</td>
<td>26</td>
<td>48</td>
</tr>
<tr>
<td>Hardware IP</td>
<td>2253</td>
<td>14307</td>
<td>6</td>
</tr>
<tr>
<td>FSL Links</td>
<td>70</td>
<td>440</td>
<td>0</td>
</tr>
<tr>
<td>Peripherals</td>
<td>987</td>
<td>1032</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2 gives a statistics of hardware resources. Most Flip Flops and LUTs are used by microblaze processor and hardware IP cores, while BRAMs are utilized for caches. FSL links uses only 1.2% Flip Flops and 2.45% LUTs of the total system; which means this star network structure can save lots of on chip hardware resources on FPGA.

5. Conclusions

In this paper, we propose a flexible high speed star network in FPGA. This star network connects a scheduler and multiple processor and IP cores through peer to peer FSL links. All the processing elements and scheduler are packaged into FSL manners. In order to measure the transfer time and hardware costs of the star network, we built a prototype system on FPGA. Experiment results show the transfer time for each word can be reduced to 7 cycles. Compared to AES encrypt and decrypt functions, this transfer time only costs 0.3% of the running time of software applications. Furthermore, the hardware cost of the star network cost only 1.2% and 2.45% of the total MPSoC system hardware.

Acknowledgements

This work was supported by a grant from the National High Technology Research and Development Program of China (863 Program, No. 2008AA01Z101).

References