The Design of SIMD Controllers for a Polymorphous Multimedia Processor

Lin Pu[1] , Tao Li[1], Xueyuan Yi[2], and Jungang Han[2]

Abstract. The design and implementation of a SIMD controller for a polymorphous array processor is presented in the paper. The polymorphous array processor is intended for graphics and multimedia applications. To meet the needs for efficient data level parallel computation, we use the state machine approach to design the row, column, and cluster controllers. A row controller dispatches SIMD instructions to processing elements; a column controller performs data and program loading as well as remote data access. The polymorphous machine employs two computation modes, the SIMD mode and the MIMD mode. The controllers are mainly used for SIMD mode of computation. A dedicated hardware circuit design improves the ability to handle parallel data of the processor.

Keywords: polymorphous array processor • array machine • SIMD controller • data level parallel

1 Introduction

Graphic and multimedia processing is one of the hot technologies in the field of information processing. It has led to a wide range of application, such as graphics rendering, audio and video processing, computer vision, meteorological information, earth resources survey, etc. With the increasing computational demand of graphics and multimedia applications, it has become a tough work to improve the speed in the field of graphics and multimedia processing [1]. The technology of parallel processing is an effective approach to enhance the speed of massive graphics and multimedia processing. The inherent complexity in graphics and image processing, such as high data volume, high power dissipation and computational complexity, demands ever increasing computational capability.

[1] Lin Pu(✉), Tao Li
School of Electronic Engineering, Xi'An Univ of Posts and Telecom, Xi An, China 710061
e-mail: pulinup@126.com
[2] Xueyuan YI, Jungang HAN,
School of Computer Science, Xi'An Univ of Posts and Telecom, Xi An, China 710061

To cope with the computational demand, we developed a polymorphous machine PAAG [2, 3], that seamlessly integrates the Single Instruction Multiple Data (SIMD) mode and Multiple Instruction Multiple Data (MIMD) mode[4]. PAAG is our first attempt to face the computational challenges. The design of the SIMD [5, 6] controllers helps the polymorphous machine to achieve its computational efficiency. This paper is a detailed exposition of the structure of the SIMD controllers of PAAG.

2 The Polymorphous Array Processor

A PAAG array processor consists of many clusters of processing elements (PE). A cluster is a 2D array of processors. The clusters can be hierarchically organized. A cluster is shown in Fig.1.

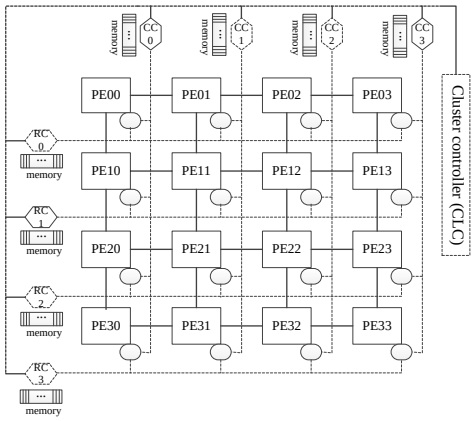

Fig. 1 A single cluster with 16 processing elements (PEs), a cluster controller, 4 row controllers, 4 column controllers, and memories.

A base cluster is typically a 4x4 array of 16 processing elements (PEs) as shown in Fig.1. PEs are connected by near neighbor connections. Each row of the array has a row controller (RC*i*) and each column has a column controller (CC*j*). These are used to configure the PEs and to issue SIMD instructions to PEs. Each cluster also has a controller (CLC) to sequence cluster-wide SIMD operations, and a cluster memory with cache to store cluster wide data and program.

A typical PAAG system has a configuration of a front-end processor, some F cluster with floating point capability, some S clusters with fixed point capability, a few special purpose hardware accelerators, an on-chip SRAM memory with cache, and an interconnect. PEs in a F cluster contains integer and floating point units and PEs in a S cluster contains only fixed point units.

A PE contains the ALU with an attached router, data memory, instruction memory, near neighbour connections with registers and SIMD interface controller.

3 The functionality of the Controllers

To allow execution of SIMD instructions at various levels in a cluster and to facilitate data access in shared memory, we employ a number of controllers, including a CLC, four RC*i* and four CC*j* as shown in Fig.1.

3.1 The functions of the controller modules

This subsection is divided into several parts according to the hardware modules and the function of each module is described as follows.

1. The function of the cluster controller module

This module is mainly used to control the processing unit in the entire cluster, including the coordination of the programs and the loading of data, as well as control the entire cluster to achieve SIMD computing.

The cluster interfaces the external world with its internal world. It is able to send programs and data to RC*i*, CC*j* and PEs. It is responsible for issuing SIMD instructions to row controllers which then forward to all PEs in the cluster.

2. The function of the row controller module

This module is mainly used to re-configure the 4 PEs of one row into SIMD mode for data level parallel computing [7].

The CLC broadcasts the instructions and data to the RC*i*. If the RC*i* detects the initial instruction, it will store the instructions to its own memory.

The CALLC instruction calls a SIMD program in a RC*i*. After receiving the CALLC instruction, the RC*i* will send out a SIMD request signal to the 4 PEs which are in a row. When the request signal comes to the PE, the PE will stop its fetch unit. Each PE will return a response signal to the RC*i* after the instructions finish execution in the pipeline.

After receiving the response signals from the 4 PEs, the RC*i* starts to send the SIMD instructions to the 4 PEs. If the RETC is detected by RC*i*, the RC*i* will stop sending subsequent instructions. A SIMD finishing signal will be sent to the 4 PEs in a row and a RETC packet will be sent to the sender of the CALLC instruction.

3. The function of the column controller

This module is mainly focused on controlling of shared memory and interconnected data channels in the cluster. If the initialization data is detected, the CC*j* will load them to the corresponding CC*j* or PE memory. When it comes the data-reading instruction, the CC*j* will send the data to CLC. If the data-reading instruction is detected by CC*j*, it will send the data to CLC. If the dynamic reading

request from the PE is detected, the CC*j* will read the reference data from the shared memory and send them to reference PE.

3.2 The Data Packet Format

Data packets can be transmitted among the PEs and the controllers. The format of the data packet is shown in Table 1**.** Packet contains a header and payload. The data package header is shown in Table 2**.**

Table 1. packet format

31:0	31:0 ·····	31:0
header	data ·····	data

Table 2. Header format

31	30	29:27	26:0
W/R	instr/data	size	address

Bit31 of the data packet header represents the read/write operation (1=write/0=read). Bit30 is the instruction/data flag (1=instruction/0=data). Bits [29:27] are the size of the payload. Bits [26:0] represent the target address.

We employ unified addressing among the controllers and the PEs. Bits [14:10] identifies the PE or RC*i*. When any bit of [26:15] equals to 0 and the [14] bit equals to 1 means that it is the operation of PE and the bits of [13:12] are the identifier of RC*i* as well as [11:10] are the identifier of PE. [9:0] bit is the memory address of RC*i*. When the condition, any bit of [26:14] is not equal to 0, is satisfied means that it is the operation of CC*j*.

CALLC and RETC are the specific instructions for SIMD, its data packet formats are shown in Table 3 and Table 4 respectively.

Table 3. CALLC packet format

31:29	28:25	24:22	21:10	9:0
100	size	tID	Addr	Ctrl
	mask	rID	rAddr	rPE

Table 4. RETC packet format

31:29	28:25	24:22	21:10	9:0
101	reserved	tID	Addr	PE

As shown in Table 3, the CALLC packet is consist of two 32-bit data. tID is the target thread ID, Addr indicates the target address, Ctrl is the target identifier of RCi, mask indicates the mask, rID represents the source thread ID, rPE represents

the source PE number, rAddr indicates the source address. When CALLC ends, a RETC packet is needed to the sender PE which sends CALLC instruction. And the second data packet should be packed in the form shown in Table 4, and hand down to the PE which send CALLC instruction.

4 The design and implement of cluster controller

The design of CCj is divided into two parts, as shown in Fig.4. One is initialization programs/data, the other is the dynamic reading program and data.

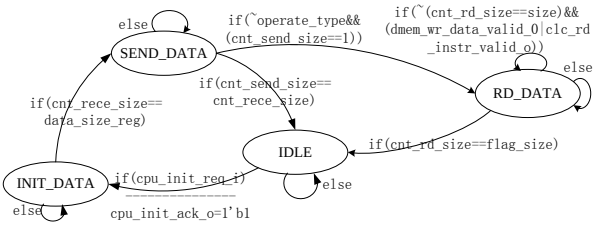

Fig. 4 The state machine of cluster controller

The descriptions of state transitions are as follows:

- *IDLE:* The idle state. After receiving the initialization request signal *cpu_init_req_i*, the state transfers to INIT_DATA state, and the acknowledge signal *cpu_init_ack_o* becomes high at the same time.
- *INIT_DATA:* The state of initialization instruction/data to cluster cache. The size of cluster cache is limited, the initialization of instructions/data needed to be divided to batches. When the initializing number of the instructions/data equal to the given size, the state transfers to SEND_DATA.
- *SEND_DATA:* The state of sending instructions/data. After the loading of instructions/data is accomplished, CLC starts to broadcast the instructions/data of cluster cache to RC*i* and CC*j*. If a write operation is detected, the state will transfer to the IDLE after the corresponding size instructions/data is transmitted. If a read operation is detected, the state will transfer to RD_DATA.
- *RD_DATA:* The state of reading data. According to the given size contained in the header, the state transfer to IDLE state after receiving the data from CCj.

5 The design and implementation of row controller

The design of RC*i* is divided into two parts, as shown in Fig.5. One of them is authorized to initialize SIMD instructions; the other is supposed to reconfigure the PEs, which is in the same row with the current RCi, into SIMD mode for data level parallel computing.

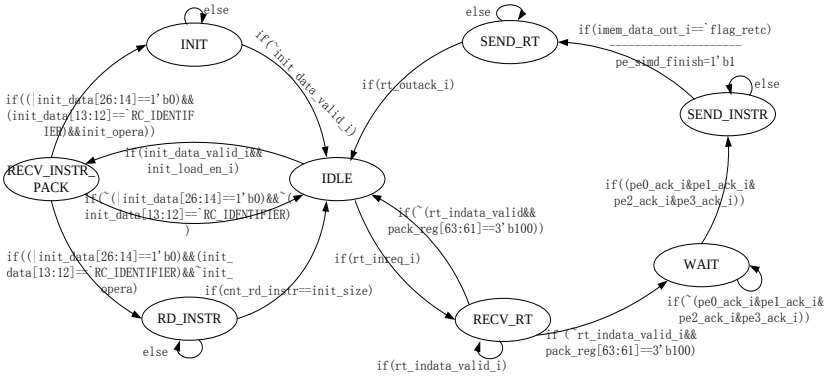

Fig. 5 The state machine of row controller

The descriptions of state transitions are as follows:

- *IDLE:* The idle state. RC*i* starts to analyze data packet as long as the initialization enable signal and the data valid signal are detected, the state will transfer to RECV_RT. If the request signal from router is detected, the state transfers to RECV_RT and a response signal is sent to router at the same time.
- *RECV_INSTR_PACK:* The state of receiving instruction packet. If the instruction is not for the current RC*i*, the state returns to IDLE. If *init_opera* becomes high, which means the writing operation, the state transfers to INIT.
- *INIT:* The state of initialization instructions. When the initialization valid signal *init_data_valid_i* is detected to be low, which means the initialization instructions has been accomplished, the state transfers to IDLE.
- *RECV_RT:* The state of receiving router packet. If *rt_indata_valid_i* is detected to be low, which means the data packet has been taken, and CALLC instruction is detected as well, a request signal is given to the PE where the RC locals to and the state transfer to WAIT. If it is not the CALLC instruction, it indicates an incorrect packet from router, the state turns to IDLE.
- *WAIT:* The state of waiting. After sending the request signal to 4 PEs where the RC*i* locals, all of the 4 PEs's fetch units stops. When the instructions in the pipeline are empty, a response signal will be returned. If all of the 4 response signals have arrived to RC*i*, the state turns to SEND_INSTR.
- *SEND_INSTR:* The state of sending instructions. It sends the initialization address and starting instructions decoded from the data packet to the corresponding 4 PEs to complete SIMD calculation mode. If the RETC instruction is detected, the sending of instruction will stop and a request signal to router will be transmitted, as well as a pe_simd_finish signal to the 4 PEs at the same time and the state turns to SEND_RT. During this time, the RETC will be sent to PE in a packet mode.
- *SEND_RT:* The state of sending router packet. The RC*i* will not send the data packet until a response signal sent by router is arrived. If it arrived, the state will turns to IDLE.

6 The design and implementation of column controller

The design of CC*j* is divided into 4 parts. First, the initialization loading of program/data. Second, CLC reads the result data from memory in CC*j*. Third, to write the result from PE to memory of CC*j*. Fourth, to read data dynamically from memory of CC*j*. The state machine is shown in Fig.6.

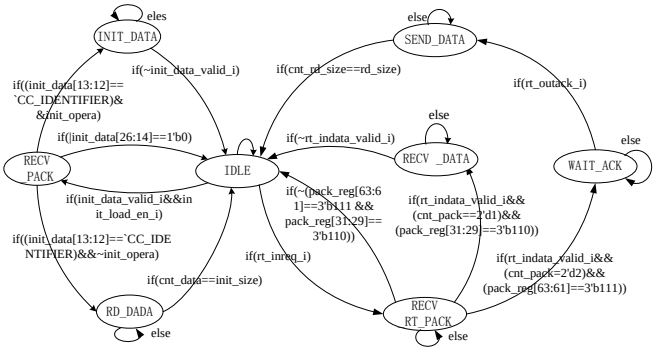

Fig. 6 The state machine of column controller

The descriptions of state transitions are as follows:

- *IDLE:* The idle state. CC*j* starts to analyze data packet when the *init_load_en_i* and *init_data_valid_i* are detected, and the state will transfer to RECV_PACK. When request signal *rt_inreq_i* from router is detected, the state transfers to RECV_RT_PACK.
- *RECV_PACK:* The state of receiving data packet. If the serial number in packet header not equals to the identifier of the current CC*j*, which means that it is not the data for current CC*j*, the state will transfers to IDLE. If the serial number equals to the identifier and it is write-operation, the state will transfer to INIT_DATA. If it is read-operation, the state transfers to RD_DATA.
- *INIT_DATA:* The state of initialization data. Broadcast the initialization data to corresponding column memory or the 4 PEs. It means that once the *init_data_valid_i* becomes low, the operation of initializing data will be accomplished and the state will return to IDLE.
- *RD_DATA:* The state of reading data. According to the initialization address and the size contained in the packet header, the CLC starts to read data from column memory. When the read-operation is completed, the state returns to IDLE.
- *RECV_RT_PACK:* The state of receiving packet from router. It starts to receive packet header from router when *rt_indata_valid_i* becomes high. There are two different kinds of data packet, one for reading data dynamically (MVF instruction), the other for writing data into column memory (MVT instruction). The error occurs if any of the two conditions is not detected and the state will return to IDLE. If the MVT is detected, the state turns to RECV_DATA. If the MVF is detected, then the state will turns to WAIT_ACK.

- *RECV_DATA:* The state of storing the result from PE to column memory. It indicates the end of writing data when *rt_indata_valid_i* becomes low. Since then, state returns to IDLE. Otherwise the current state will be kept to storing the result.
- *WAIT_ACK:* The state of waiting the response signal from router. The dynamic read data are transmitted through router. The router will not response until it is available after the request signal has been detected by router. And if it is detected, state will turn to SEND_DATA, otherwise the current state will be kept.
- *SEND_DATA:* The state of sending data. According to initialization address and size contained in data packet header, CC*j* starts to read data dynamically from column memory. When the requested data have been read, the state turns to IDLE.

7 Conclusions

This paper designs and implements a SIMD controller by the means of state machine. Function verification, DC synthesizes and FPGA validations are accomplished for the RTL. The results show that the SIMD controller works properly. The circuit is well scalable and it is in a good practical which could satisfy the requirement of polymorphism parallel processor.

Acknowledgements I would like to thank my mentor TaoLi for many discussion and his contributions to this work. I am also grateful to anonymous reviewers whose suggestions helped to improve the paper. This work is supported by China Natural Science Foundation (grant 61136002) and Shaanxi Provincial government (grant 2011k06-47).

References

1. LiQiang, Research and Design of the Key Technologies of a 32-bit Image Vector Processor[D], DaLian: Dalian university of technology, 2009.
2. TaoLi.A Polymorphic Array Architecture for Graphics and Image Processing [J]. IEEE, 2012: 242-249.
3. LiTao, A thin-core array architecture for graphics and image processing [J], academic journal of an Institute of Posts & Telecommunications, 2012, 17(3): 42-47
4. Flynn M. Some Computer Organizations and Their Effectivenes[J]. IEEE Trans. Computers, 1972, C-2: 948-949.
5. http://en.wikipedia.org/wiki/SIMD, Retrieved on July 25, 2012.
6. ShenXB. Evolution of MPP SoC architecture techniques[J], Sci China Ser FlnfSci. 2008, 51(6): 756-764.
7. Chas Boyd, "Data Parallel Computing", ACM QUEUE, March/April 2008, pp30-39.