Waseda University Doctoral Dissertation

SIMD Based Multicore Processor for Image and Video Processing

HE, Xun

Graduate School of Information, Production and Systems

Waseda University

February 2012

Abstract

Continuous improvements in image and video processing require high computational power to deal with the increasing complexity of algorithms and higher definition video. Meanwhile, development in VLSI technology allows the integration of more cores on a single chip for achieving higher performance. There are three levels of parallelism in applications: task level, data level and instruction level. Typically, multicore architecture can obtain high task level parallelism, SIMD can achieve high data level parallelism, and VLIW architecture can achieve high instruction level parallelism. However, hardware costs and performance gains of the three methods are quite different for video processing. Multicore can achieve almost double performance gains with double hardware costs for a video encoder. SIMD can achieve double data level parallelism with less than double gates. VLIW needs more than double gates for double instruction level parallelism. Based on these technologies, we design and fabricate a 32 cores processor, and also evaluate its performance. The results show that this processor can achieve very high performance for our target applications.

Chapter 1 [Introduction] presents a background introduction of parallel architectures and challenges in multicore processor.

Chapter 2 [SIMD based Core Architecture] presents the SIMD based core architecture. For maximizing on-chip parallelism, multicore and 128 bits SIMD architecture is applied. SIMD instruction set is optimized for multimedia applications. The core is consisted by one small 32 bits RISC core, and SIMD pipelines. RISC is based on open source project: OR1200. SIMD is divided into two parts for executing two instruc-

tions at the same time. Comparing with CELL processor's SIMD, the proposed work can reduce by 29% cycle count for video applications.

Chapter 3 [Application Specified Cache Coherence Protocol] proposes a manually controlled invalid cache coherence protocol (MCI). In multicore system, the cache coherence problem becomes more and more important with core number. In conventional snooping based protocol, coherence transaction broadcasts to all cache monitors. M. Ekman's Experiments in WDDD2002 show that more than 70% percentage of all snooping operations misses in other caches for conventional MOESI protocol. This means that most of the snooping induced tag-lookups just waste the power. In the proposed MCI protocol, memory space can be dynamically or statically defined as shared or private space. The shared spaces can be shared by all cores, clusters or several cores. A new programming model is proposed for defining the sharing patterns. A snooping unit is added for each core, which keeps the sharing configurations and sends out invalid messages automatically. For MCI, the cost for coherence is mainly determined by data sharing method, not the number of cores. Large scale SMP processor can also achieve good performance in MCI. In our experiments, MCI is compared with Jetty which is proposed for reduced snooping operations by A. Moshovos in HPCA2001. For a 32-core processor, snooping operations of MOESI protocol costs more than 50% of L1 cache's power. Jetty can reduce about 24.6% snooping operations for MOESI. Snooping operations in the proposed MCI protocol is about 42.7% less than MOSEI with Jetty.

Chapter 4 [Communication Network] describes a Core Interconnection Bus (CIB) for connecting eight cores. In this processor, there are four clusters, and each cluster is consisted by 8 cores and one shared cache. Data sharing within cluster is supported by the sharing cache. However, it's inefficient to use the shared cache for data

communication between SIMD cores, as it needs a write back and read operation to L2.To enhance the data communication ability for SIMD core, CIB is designed for achieving very low latency data transfer between cores. Vector cores directly exchange data through CIB. In CIB, every flit has routing header, and they share the links in time-division multiplexing method. Data transmissions are divided into independent flits. Each flit can route in CIB independently. CIB can broadcast one flit to all cores, which is not supported in channel based NoC. Comparing with S. R. Vangal's NoC design in JSSC08, CIB doesn't store data flits, which can save a lot of buffer resources. Thus CIB can achieve more than 10 GB/s BW for small size vector transfers, which is more than 2.5 times better than S. R. Vangal's work. When the injection rate is less than 8%, CIB's latency is less than 4 cycles in average.

Chapter 5 [Chip implementation and Performance Evaluation] shows the chip implementation and performance evaluation of the proposed 32 cores processor. Applying the proposed technology together, a 32 cores processor has been fabricated and verified in SMIC 65 nm CMOS. This chip is consisted by 32 cores (286KGates per core), 256KB L2 cache, two PLL, and one 64-bit DDR PHY, and the die is about 25 mm2.This processor can achieve a maximum speed of 750 MHz at 1.2 V core power. The whole chip can achieve a peak performance of 375 GMACs, or 750 GOPS of 8-bit data operations. It can achieve 1.9 times higher GMACs performance than D. N. Truong's 167 RISC cores chip in JSSC09. For SAD (Sum of Absolute Difference) and Matrix Multiply kernels, the proposed work's cycle count is reduced by 37% than TI C6415 DSP.

Chapter 6 [Extended Processor with Hardware Accelerator] presents a sharing hardware accelerator for extending the proposed SIMD processor. This multicore platform is designed for high performance multimedia applications by maximizing on-chip data level and task level parallelism. However, there are limitations for high parallel

system: sequential functions with less parallelism become the bottleneck. Hardware accelerators are added for resolving these problems. As the usage rate of accelerators is very low, it's unnecessary to add the same accelerator for every core of multicore processor. Sharing resources and reducing the cost becomes a hot topic recently for multicore processor. The low latency CIB network makes it possible for sharing hardware accelerators within a cluster. Based on our previous works on 4Kx2K@60fps H.264 Decoder, an intra decoder is added in the multicore platform as a shared hardware accelerator. It can satisfy eight channels parallel HD decoding at 31 MHz.

Chapter 7 [Conclusion] summaries the proposals. Based on this design, the proposed 32-core SIMD processor can be applied for a lot of multimedia applications, such as video decoding or image processing. The dual-issue SIMD cores can guarantee very high performance for processing 8-bit pixels. Together with hardware accelerators, the application fields can be wider.

Contents

List of Figures

List of Tables

1 Introduction

1.1 High Performance Solutions for Multimedia Applications

During the last decade, multimedia technology has witnessed its rapid development in the world. For example, digital HDTV is widely used in our daily life, such as Blue-ray disk and TV broadcasting. Continuous improvements in image and video processing require high computational power to deal with the increasing complexity of algorithms and higher definition video. Meanwhile, Continuous improvements in VLSI technology allow the integration of more transistors on a single chip. There are two popular solutions for achieving higher performance: Single-instruction, multiple-data (SIMD) and multicore processor. Both of them can explore parallelism at different levels, and then achieve high performance for multimedia applications. According to the requirements of our target applications, we use dual-issue SIMD core architecture and design a 32-core SIMD-based Vector Processor (SVP).

1.1.1 SIMD and Massively-Parallel Processors

SIMD is a class of parallel computers, which exploits multiple data streams to perform operations which may be naturally parallelized. It consists of multiple processing elements that perform the same operation on multiple data simultaneously [1]. Thus, such machines exploit data level parallelism. The first use of SIMD instructions was in vector supercomputers of the early 1970s such as the CDC Star-100 [2], which could process a vector of data by one by one instruction. In mainstream CPU for PC, SIMD becomes the basic extension for real-time graphics or games. The first wide-

ly-used desktop SIMD was with Intel's MMX ISA for x86 architecture in 1996 [3], followed by SSE/SSE2 [4]. IBM and Motorola also developed AltiVec [5] ISA for POWER architecture. A modern desktop CPU is often a multicore processor, where each core can execute SIMD instructions.

As multimedia applications have extensive data-level parallelism, there are a lot of embedded processors are designed with SIMD [6-9]. Massively-parallel single-instruction-multiple-data (MP-SIMD) machines [10] [11] are widely used to exploit as much data-level parallelism (DLP) as possible. Xetal-II [10] is also a SIMD processor with 320 processing elements (PE), and each PE contains one 16bit MAC. It delivers a peak performance of 107 GOPS or 27 GMACS on 16-bit data while dissipating 600 mW. They can achieve high DLP with very good power efficiency. However, they are insufficient in instruction-level-parallelism (ILP) and task-level parallelism (TLP). The Stream processor [11] contains two cores for main application threads, and a DPU for vector processing. In the DPU, sixteen SIMD lanes are combined to deliver performance of 512 8-bit GOPS or 128 16-bit GMACs with a power efficiency of 12.2 GMACS/W. However, sixteen lanes can only execute one task, and the TLP is limited.

A very long instruction word (VLIW) [12] vector coprocessor (VCP) is proposed for the computation requirements of image processing [13]. In VCP, the coprocessor includes three execution pipelines with cascaded SIMD ALUs to exploit the instruction-level parallelism. 3-way VLIW architecture is used for exploiting high instruction-level parallelism. 128-bit SIMD architecture is used frequently to exploit high data-level parallelism. VCP is designed to be a coprocessor for image processing of video CODECs, thus the width of SIMD ALUs is limited to that of macroblocks of CODECs.

1.1.2 Embedded Multicore Processors for Multimedia Applications

Multi-core processor is a single computing component with two or more independent processors (called "cores"), which can read and execute instructions independently. Today, multicore design is widely used in high performance processors. Even our video game consoles are also shifting to this direction. The Xbox 360 uses 3-way chip multiprocessor (CMP), and PlayStation 3 uses the CELL processor, which have one PowerPC Element and 8 SIMD Processing Elements [14]. Therefore, it is very important to understand the benefits of multi-processor systems.

Multicore processors can deliver high task level parallelism (TLP) at low power consumption, so they can offer higher computational power and design flexibility than single core processors. As embedded systems always have very limited budget for power and hardware cost, multicore architectures are very suitable for them. The TILE processor [15] is a 64-core SoC targeting at the high-performance demands of a wide range of embedded applications such as network and multimedia applications. Each core is a 3-wide VLIW machine with a 64-bit instruction word and connected through a scalable 2D mesh network. It attains 384 GOPS and 144 billion instructions per second (GIPS) with 10.8W. Another processor applies 167 fine-grain cores to build a many-core computational platform for DSP, embedded, and multimedia applications [16]. At a supply voltage of 1.3 V, the chip achieves 196.8 GIPS (16bit) or 393.6 GOPS (16-bit), while dissipating 10.2 W. Each core contains 16-bit ALU, multiplier, and a 40-bit accumulator, which just costs 0.17 mm2 in 65 nm. This fine-grain many-core processor is excellent in TLP.

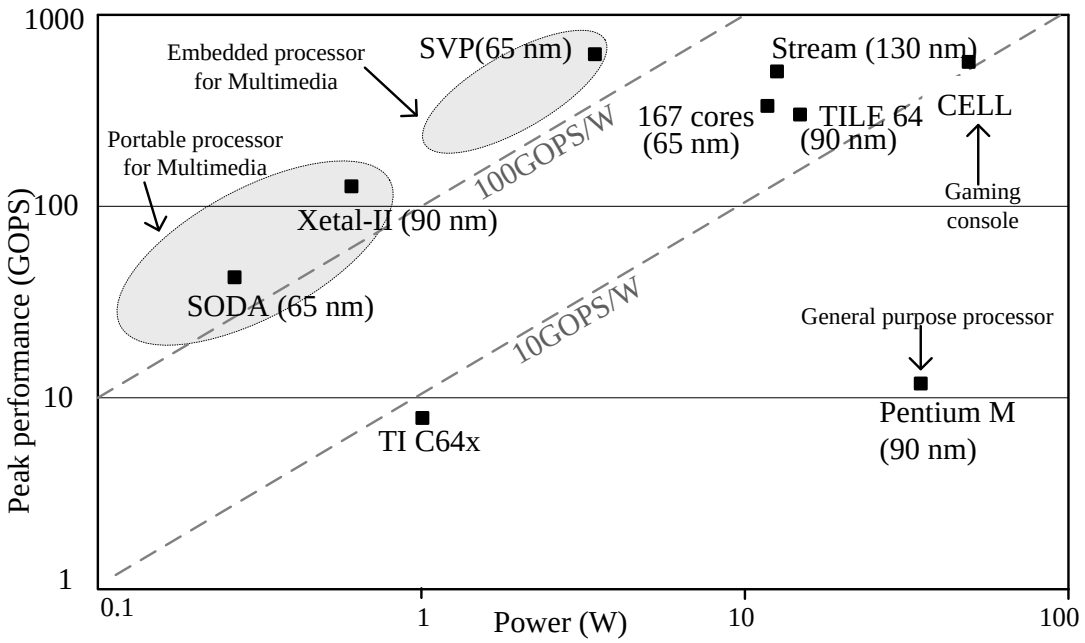


Figure 1-1 Power efficiency comparison of MP-SIMD and CMP processors.

Figure 1-1 shows the comparison of throughput and power. Compared with MP-SIMD, multicore architecture requires more instructions and power for vector processing. MP-SIMD can achieve better power efficiency, but poor GIPS performance. GOPS and GIPS are just a common indicator of the performance, and the improvement in performance gain depends on the software algorithms and implementations. For multicore processor, gains are limited by the fraction of the task level parallelism. This effect is described by Amdahl's law. In the best case, it may get a speedup factor near the number of cores. However, most applications are not accelerated so much which depends on applications. For MP-SIMD designs, gains are limited by the fraction of data level parallelism of applications. In our target multimedia applications, data level parallelism is higher than other general applications. SIMD based architecture is very suitable for this applications. For general applications, task level parallelism is more common.

To meet future multimedia system's requirement, a SIMD based Vector Processor (SVP) is presented in this thesis. Both SIMD and multicore architecture are used for

achieving higher performance. The target throughput and power budget (Embedded Processor for Multimedia) is also shown in Figure 1-1. Our target power efficiency is higher than 100GOPS/W, and power budget is about 3 Watt for a high performance embedded processor.

1.2 Challenges in Multicore Processors

1.2.1 Parallel Programming

In multicore processors, the cores are always connected for data exchange and often share an on-chip cache memory. The challenge of programming multi-core processors is real, but it is not a technical challenge. Since the 1970s, the problems of how to program multi-core processors and how to write programs for them have been resolved [17]. Many languages and systems implement dataflow programming for multicore processor, such as SISAL [18], and other early dataflow languages. MapReduce [19] is one of the most popular new tools for dataflow based programming.

Open Multi-Processing (OpenMP) [20] is an Application Programming Interface (API) that supports shared memory multiprocessing programming on most popular operating systems, including Linux, and Microsoft Windows systems. OpenMP is a user-model parallel programming tools, which is widely adopted. It can offer a flexible interface for developing parallel program [9]. It defines a lot of compiler directives, library routines, and environment variables for run-time behavior [21][22][23]. For example, the section of code which runs in parallel is marked with a preprocessor directive (such as “#pragma omp parallel”). Then the compiler knows which thread will be works in parallel and then each thread has an "id" attached to it. The thread id is an integer

value. For master thread, the id is "0". After the execution of the parallelized code, the threads join back into the master thread, which controls the whole task.

1.2.2 Cache Coherence Protocols

The caches in current distributed shared-memory multiprocessors improve performance by decreasing the bandwidth requirements of both the local memory and the global interconnect. However, the local caching of data introduces the cache coherence problem. Figure 1-2 shows an example of the cache coherence problem. Memory initially contains the value 0 for location A, and core 0 and 1 both read location A into their caches. Next, core 1 writes location A in its cache with the value 1, then core 0's cache still contains the old value 0 for location A. Read operation of location A by core 0 will get the old value 0. It isn't what we expected for a shared memory system. The expected behavior is to return the most up-to-date copy. In snooping based cache, all of the caches can get the write operations to L2 cache and then update or invalid themselves. For shared L2 cache by all cores, there aren't duplicated copies in L2 cache. But if there are private L2 cache as Figure 1-2, then the cache coherence problem also exists in L2 cache. Private L2 cache units also have to resolve the coherence problem.

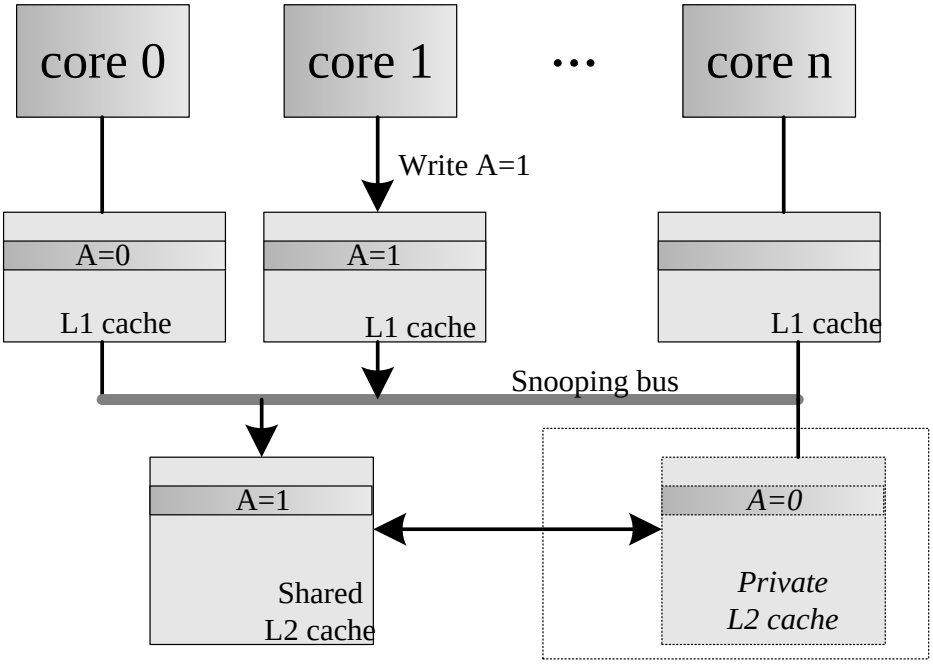


Figure 1-2 The cache coherence problem of a snooping based system

Cache coherence protocol is designed for resolving this problem. It can ensure that requests for a certain data always return the most recent value. Cache coherence protocol is a scheme for coordinating access to shared memory. Current-generation multiprocessors solve the cache coherence problem in hardware by supporting a cache coherence protocol. There are two main kinds of cache coherence protocols: snoopy protocols [24-27] and directory-based protocols [28-30]. Snoopy protocols always use a broadcast medium in the machine and only apply in small-scale multiprocessors. In these systems, each cache units “snoop” on the bus and look for transactions which will affect it. When a snooping unit sees a read request on the bus, it checks to see if it has the recent copy of the data. When a snooping unit sees a write on the bus, it will take that line out of its cache if it has that data. These snoopy bus-based systems are easy to build, and it’s suitable for small scale system and bus based network. However, as the number of processors increases, the single shared bus becomes a bandwidth bottleneck. The snoopy protocol’s broadcast mechanism causes a linear increase in the number of broadcasting operations. Snooping-based protocols may be not scalable, as all requests

must be broadcast to all processors. All processors should monitor all requests on the shared interconnect. Shared interconnect utilization can be high, leading to very long wait times.

To resolve the BW and scalability problem of snooping, the distributed shared memory (DSM) architecture with directory-based cache coherence protocol is proposed [31]. In a DSM processor, each node contains a core, a distributed memory, and a node controller. The controller is used for managing the data communication within and between cores. And the directories are stored in the shared cache. In directory based protocols, directory maintains coherence state. A simple directory structure is shown in Figure 1-3, which has one directory entry per block of memory. Each line of the cache has a corresponding entry directory, and each entry of the cache contains one bit for each core. The core number determines the width of directory. In addition, some state bits indicate whether the block is private, shared in multiple caches, or held exclusively by one cache. Then directory know which caches need to be invalidated when a location is written. The directory also indicates whether the copy of the block is up to date or which cache holds the most recent copy. Directory-based cache coherence protocols work by checking the directory on each cache miss, and then taking the appropriate operation based on the type of request and its state in the directory. For a cache miss, the cache units send a request to communication network for detecting which cores have this data. When one core writes to a cache line, the responding data in directory is checked. And then the core sends out invalidation or update messages to the cores which have this data according directory's information.

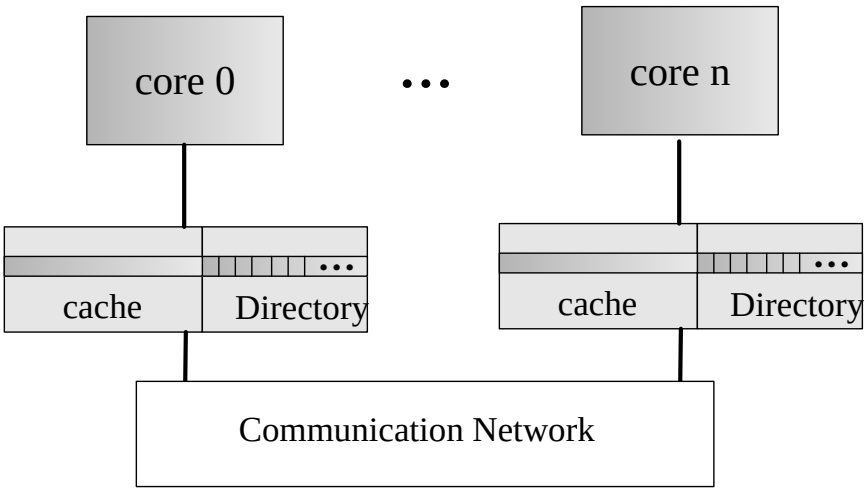


Figure 1-3 A directory based cache coherent multiprocessor

Thus directory is used to track state of each cache line, which is always stored in the cache. For each cache line, the directory needs to track, which cache has the latest copy of the line if the line is held exclusively or which caches have copies of the line. Maintain of directory is quite complicated, and the hardware cost of directory will increase linear with core number. For large scale multicore processor, both design complexity and hardware cost are very high. For example, a 64 cores processor needs more than 64 bits for each cache line.

Martin et al. [25] shows that snooping can outperform directories on a medium size system (<16 cores), at the cost of additional bandwidth. For a 64 cores system [26], one core needs more than 6 GB/s endpoint BW to achieve the best performance. It means that the whole system needs to provide about 384GB/s BW for coherence. If there is enough BW, snooping based protocols can achieve 1.6 times better performance than directory. However, if the endpoint BW is less than 2GB/s, directory can achieve more than two times better performance than snooping.

1.2.3 Inter-core Communication Challenge

As the number of cores increases significantly, the communication solutions also need to change drastically in order to satisfy the inter-core communication requirements. In conditional bus architecture, there are two kinds of architectures: shared bus based [32-35] and crossbar based interconnection [36][37]. Shared bus based systems provides a shared connection for various peripherals and cores. There is a central arbitrary, which is used for judging which one can use the bus. When one shared bus can't satisfy the BW requirement, multiple shared buses are added to increase bandwidth, decrease signal latencies [33]. When the core number increases, hierarchical bus architecture is used [34]. In shared memory multicore systems, a high bandwidth connection is required between the cores and the cache banks, which needs to allow multiple core ports to launch operations to the L2 subsystem in the same cycle. Crossbar based network is designed for resolving such kinds of problem. The crossbar system always consists of crossbar links and controlling logic modules. Crossbar consists of links from each core to all of the banks (cache's banks), and links from every bank to the cores.

A lot of research studies have demonstrated the feasibility and advantages of Network-on-Chip (NoC) over traditional bus-based architectures [37-39]. Networking theory and routing methods are applied in NoC and achieve notable improvements over shared bus and crossbar interconnections. It's layered-stack approach to the design of the on-chip inter-core communications methodology.

In a SoC system, NoC is used as a public information transportation system for cores and specialized IP blocks. It consists by point-to-point data links and routers. And then data packages can be delivered from any source point to any destination point over several links. Routers are in charge of data routing, both source and destination points can't control routing. It also needs extra routing information in each package, which is

determined by the size of network. NoC is similar to a modern telecommunications network, using digital bit-packet switching over multiplexed links.

NoC is consisted by topology, routing, and channel control [40]. Topology is the arrangement of nodes and channels into a system. Routing method defines how a packet chooses a path in these nodes. Channel control deals with the allocation of channel and buffer resources to a packet. NoC networks are composed of two types of resources: channels and buffers, and buffers are always associated with channels one by one. Once the channel and corresponding buffer is allocated for packet P, no other packets can use the associated channels until they are released. If packet P is blocked by some resource conflicts while holding the buffers, the allocated channels may be idle and waiting. However, the other packets can't use them.

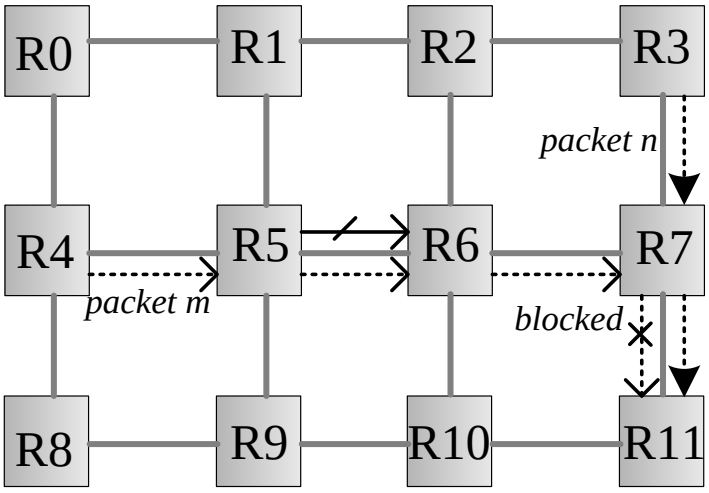


Figure 1-4 Buffer conflicts in a NoC

Figure 1-4 shows such kinds of resource conflicts. In the figure, there are 12 routers and each router has two links for each direction. Router R3 is sending data to R11, and the link between R7 and R11 is used. Then R4 sends a packet to R11, and allocated the links and buffers from R4 to R7. However, the link and buffer of R7 to R12 is occupied, and then R4's packet is blocked at R7. At this time, if R5 can't allocate a buffer for

sending data to R6, as R4 has not released the links of R5 to R6. Even that link is idle, the buffer is full of data from R4, and the other packets can't use it.

To resolve this problem, virtual channel is proposed. Virtual channels decouple resource allocation by providing multiple buffers for each channel in the network [41]. If a blocked packet holds a buffer associated with a link (channel), another buffer can be used for allowing other packets to pass this path. Figure 1-5 shows a 4x3 NoC with two virtual channels. When the buffers of R4 to R7 are allocated by R4 and the links are idle, the other units can use the idle links with the buffers of another channel.

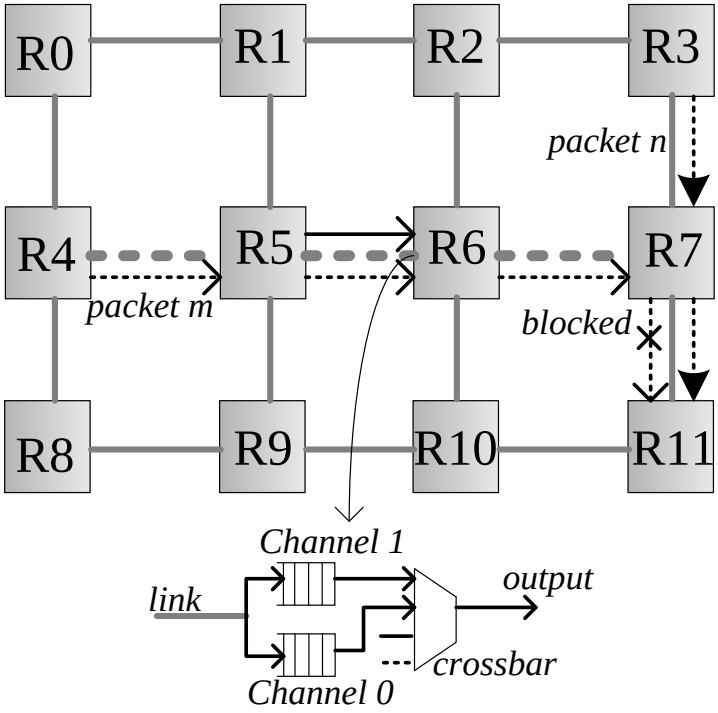


Figure 1-5 Virtual channel in NoC

If the links are idle and the buffer is allocated, virtual channel can improve the usage rate of the links and resolve the deadlock problem in NoC [42]. However, it can't resolve the block problem caused by data link resource.

In NoC, all data in one transmission are packaged as one packet. Packet is always divided into fixed width flits. To build a connection for sending a packet, several extra

flits are needed for sending destination memory address, routing information. Figure 1-6 shows a basic packet and the data format of each flit in [43]. One short data transmission needs three flits at least: the first one acquires a channel between source and destination, the second one sends the memory address and the others send the data. The data flits doesn't include information of destination, thus flits in one packet can't be spaced out. Six bits is used for controlling the channel. As the data packets don't have any information of destination, the flits of one packet can't be broken into pieces. The packet flow in Figure 1-6 shows that flits in one packet must follow one by one, and then size of packets is flexible.

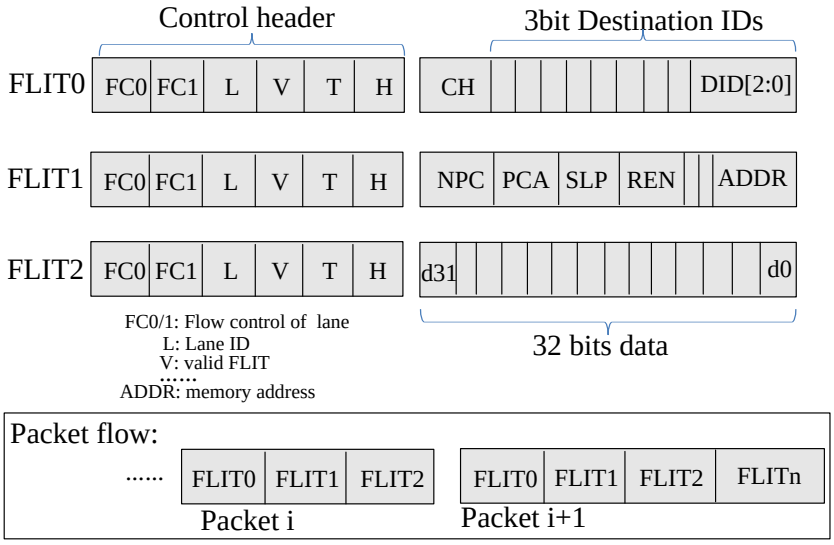


Figure 1-6 Data format of each flit and packet flow in NoC.

A lot of research predicts that packet switched networks will replace buses in multicore and many-core systems [39]. However, most of products are still using shared bus and crossbar network [36], as the latency of NoC is quite high than crossbar. As wire delay in high speed clock systems will affect the timing budget, leading to wires that cannot reach across the chip, several pipelines are used in crossbar network.

1.3 Open Source RISC: OR1200

RISC processors are widely used both in mainstream and embedded computing due to their high performance and low power computation. A lot of open source RISC projects are started for improving RISC's architecture. OpenRISC 1000 is one of open source RISC projects, which aims to develop a series of general purpose RISC architectures. OpenRISC 1000 is architecture for a family of free, open source RISC processor cores. The architectural description is for the OpenRISC 1000, describing a family of 32 and 64-bit processors with optional floating point and vector processing support [44]. OpenRISC 1000 allows for a spectrum of chip and system implementations at a variety of price/performance points for a range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, scalability and versatility.

OR1200 is an implementation of OpenRISC 1000 processor family. OR1200 is a 32-bit scalar RISC with Harvard micro-architecture, 5 stage integer pipelines, virtual memory support (MMU) and basic DSP extension. In default configuration, it has one 1-way direct-mapped 8KB data cache and 1-way direct-mapped 8KB instruction cache. And the cache line is 16-byte for both of them. MMUs are also implemented, which are constructed of 64-entry hash based data TLB and 64-entry hash based instruction TLB. Supplemental facilities include high resolution tick timer, programmable interrupt controller and power management support. The default OR1200 configuration is about 40k ASIC gates. When implemented in a 0.18 um process, it can provide over 150 dhrystone 2.1 MIPS at 150MHz and 150 DSP MAC 32x32 operations, at least 20% more than any other competitor in this class (typical corner 250MHz).

1.4 Organization of the Thesis

The rest of this thesis is organized as follows.

Overview of the whole system is presented in Chapter 2. Some kernels from our target applications are profiled at first. SIMD and VLIW architectures are compared. According to the results, dual-issue vector core architecture is proposed. A new vector memory is designed for storing vectors, which supports word address unaligned access and vertical vector access. The SIMD instruction set is also presented in this chapter. Cycle level performance evaluation and optimization process of this ISA is also shown.

Chapter 3 proposes an application specified cache coherence protocol to achieve a power-efficient embedded SMP processor. It's a snooping based protocol and resolves the high BW and energy cost problems for large scale SMP. Software needs to define the memory sharing spaces and cores. The broadcasting nature of snooping based protocols can be limited and the useless snooping operations can be avoided.

Chapter 4 presents a hierarchical communication network based on ring and mesh topologies, which can achieve a high BW and low latency network. Comparing with conventional NoC, every data flit has a 13 bit header for denoting the destination address. The routers don't have buffers for incoming flits in network. The links works in a time-division multiplexing (TDM) way. There isn't deadlock problem as conventional NoC.

Chapter 5 presents the design flow and test environments of this processor. The whole chip is fabricated by SMIC 65 nm technology, which can achieve a peak performance of 375 GMACs, or 750 GOPS of 8-bit data operations. Performance evaluation and comparison on some applications are presented in this chapter.

Chapter 6 presents a shared hardware accelerator on this multicore platform. As SIMD based ISA is poor for serial scalar functions, hardware accelerator is very im-

portant complementarities for this processor. However, it's a waste to add accelerator for every core as the usage rate is very low. A shared intra decoder is designed based our previous design. It can achieve about four times speedup as an accelerator with less hardware cost.

Chapter 7 concludes the contributions of this thesis.

2 Dual-issue Vector Core Architecture

Multimedia applications always have a huge of 8-bit pixels, and the computation complexity is very high, especially for high definition applications. General processors such as ARM or X86 are mainly designed for 32-bit tasks, and they also added some 8-bit extension instructions for multimedia applications. However, it's not enough for the requirements, and most of systems have to apply hardware accelerators for video codec. These kinds of solutions lose the programmable flexibility, and other applications such as face tracking or ray tracing can't use the accelerators.

In order to design a high performance platform for multimedia processing, we profile two typical multimedia applications on different architectures, and analyses the parallelisms of them. We compare the massively parallel single core processor [13], SIMD based 8 cores processor and 128-bit VLIW single core processor [14]. We found that the multicore architecture can achieve almost linear performance gains with hardware costs. The performance of massively parallel or VLIW single core processors is limited by application's data level or instruction level parallelism. Performance gains of them is poor than multicore architecture. A dual-issue core based on SIMD is proposed in this chapter.

In section 2.1, some target applications are introduced, and then performance profiling of them is presents. VLIW and SIMD are compared for our target applications.

Section 2.2 presents the overview of the whole chip, including core architecture, memory and 2D-DMA engine.

In section 2.3, the instruction set of this vector core is presented. The ISA of OR1200 is modified for parallel issuing. RISC and SIMD pipelines share the instruction

fetch and decode stages. SIMD ISA is optimized for 8 bits data processing with a lot of 8-bit MAC instructions.

Section 2.4 presents a vector memory, which can support word unaligned access and vertical vector load/store.

Section 2.5 presents cycle level performance evaluations for some video kernels.

Section 2.6 gives the summary of this chapter.

2.1 Background

2.1.1 Target Applications

Multimedia technologies make our daily life more interesting. Video technologies can record our past, and digital image technologies can make the world more colorful. As the HD video system (1080p) has already become popular in our daily life, even mobile devices need to support HD applications. And even higher definition such as the 4Kx2K format, which delivers about four times data throughput of HD, has been targeted by next-generation video systems. 3D TV and multiview applications [45-46], which are also becoming popular, also have several times of data processing complexity. In current systems, video decoding is the most common multimedia applications. However, general purpose processors such as X86 or ARM, are not capable for HD video real-time processing, and they have to integrate hardware video codec. In this work, we try to implement programmable platform for HD or other high complexity multimedia applications. The instruction set is optimized for 8-bit based pixel processing. Thus applications based on 8-bit pixels can get good performance with this architecture.

There are a lot of embedded systems which don’t have a complex OS such as Linux. For the embedded multimedia system, they always deal with single task, and real

time requirement is more critical. To reduce the hardware cost, this processor doesn't support MMU or multi-thread.

2.1.2 Performance Profile for Different Architectures

As frequency is limited by technology and power, maximizing on-chip parallelism becomes the way for achieving higher performance. According to Amdahl's law [47], the speedup of a program using parallel computing is limited by the time needed for the sequential fraction of the program. Thus we must know well about the target application's feature of parallelism, and then choose the responding architectures.

There are three levels of parallelism in applications: data level, instruction level and task level. In multimedia applications, one frame has a huge of pixels, and the data level parallelism is always very high. For example, video applications always bases on 16x16 data blocks, and 16 pixels are often processed by the same operations. SIMD is known as an efficient architecture for data level parallelism. Instruction level parallelism (ILP) widely exists in applications. VLIW architecture can achieve high instruction level parallelism. Multicore architecture can obtain high task level parallelism. In our previous work on CELL processor [55], we found that it can achieve almost linear performance gains with the core number for a video encoder. It means a double resource can achieve almost double performance.

Profile results of two video encoders on different DSP platform are shown in Table 2-1. The two DSP processors just have one 32-bit ALU. Sum of absolute differences (SAD) calculation and DCT/IDCT become dominant in video encoding. In order to compare the efficiency of the SIMD and VLIW architecture for our target applications, we select SAD and DCT/IDCT as our benchmarks and evaluate the cycle level performance of TI's single core 8-way VLIW DSP (C6415) and 128-bit SIMD processor (CELL). Table 2-2 shows the performance in cycle. In SAD, one pixel just need one

subtract and one add operations. CELL's SIMD is about two times better than the 8-way VLIW based DSP for SAD. In DCT/IDCT, one pixel needs more arithmetic than SAD, and ILP is higher than SAD. CELL's SIMD can still achieve more than 10% better performance. However, CELL is poor than C6415 DSP when the address of data is unaligned for SAD. CELL's local memory only supports 16 bytes aligned access. It needs about 4 cycles for loading one unaligned vector. On the other hand, CELL's SIMD just needs to fetch and decode one instruction for one 128 bits vector. 8-way VLIW needs to fetch and decode eight instructions per cycle. The power efficiency of SIMD is also better than VLIW for video applications.

Table 2-1 Profile results of two video encoders on DSPs

MPEG 4 encoder on TI C55 DSP [48]		MPEG 2 encoder on Blackfin DSP [48]	
Kernels in H. MPEG 4	Percentage	Kernels in MPEG 2	Percentage
Motion estimation (SAD)	35%	Motion estimation (SAD)	38.86%
DCT	10%	DCT	8.05%
IDCT	20%	IDCT	7.11%
Pixel interpolation	16%	Run-length Encoding	9.57%
other	20%	Format Conversion	8.22%

Table 2-2 Performance comparisons of VLIW and SIMD

Functions	IDCT 8x8	DCT 8x8	SAD 16x16	SAD 8x8	MAC/cycle
CELL[14]	126	102	33(78*)	17(40*)	8x16-bit MAC
C6415[48]	154	116	67	31	4x16-bit MAC

*: memory address is unaligned. CELL needs more than three instructions for one unaligned load.

2.2 Architecture Overview

2.2.1 System Overview

For embedded processor, low power requirement is not as critical as portable processor. But if power consumption exceeds 5 W, we still have to be rather careful about the thermal problem. Thus our power budget is about 2-4 W for this high performance embedded processor. Considering flexibility and power efficiency requirements, SMP based multicore architecture is utilized to achieve scalable performance for computationally-demanding image and video applications, such as FTV and UHD video. However, coherence operation increases rapidly with core number, which causes high power consumption and heavy traffic in communication network. To resolve this problem in power-efficient embedded SMP processor, an application specified cache coherence protocol is proposed, called manual control invalid (MCI) protocol. Based on this coherence protocol, a SMP processor with a hierarchical network is designed for high-performance multimedia applications as in Figure 2-1.

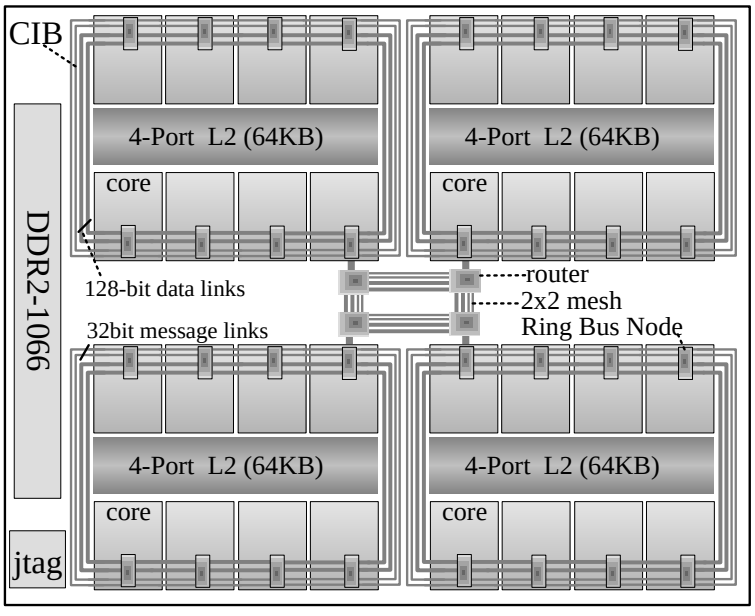


Figure 2-1 Block diagram of the proposed 32-core processor

It contains 32 dual-issue vector cores, 64-bit DDR2 1066Mb/s memory and 1 MB on chip memory. Each vector core can achieve a maximal frequency as 750 MHz. Du-

al-issue vector cores contain two 128-bit SIMD pipelines and one 32-bit RISC pipeline. The RISC core is based on OR1200. It has an 8 Kbytes 2-way associative data cache, an 8 Kbytes direct mapped instruction cache. In order to support issue two 32 bits instructions per cycle, instruction cache is designed as 64 bits wide cache, supporting word address access. The MMU, 32x32bit MAC and WISHBONE compliant interfaces is removed for saving gate count. Two SIMD instructions or one SIMD instruction with one RISC instruction can be issued in one cycle. As 8/16-bit wide data is widely used in multimedia applications, this SIMD instructions set focuses on 8/16-bit SIMD operations, including 8-bit MAC instructions. To achieve higher frequency, 32-bit MAC SIMD operations are not supported (but 16x32-bit MAC is supported). Generally, the bottlenecks of SIMD system are the data movement and rearranging operations, thus a flexible Vector Memory (VM) is designed with vertical vector access ability.

Then eight vector cores make up a cluster, and the whole chip consists of four clusters. Eight cores are connected by Core Interconnection Bus (CIB) in cluster, and then clusters are connected by 2x2 mesh network. For 128 cores, 4x4 mesh network is enough. Thus this hierarchical network can offer good flexibility for large-scale SMP. In this network, data transmission is divided into 16-byte packages, and several bits are added as package header. The whole chip can achieve about 192 GB/s inter-core BW (within cluster), and 18 GB/s inter cluster BW. Bandwidth management units (BMU) are used to allocate BW at heavy traffic case. DDR II controller (MIU) is also optimized to improve the bandwidth efficiency by eliminating most of the extra overhead as in [49].

2.2.2 Vector Core Architecture

In video applications, high parallel functions can easily achieve more than eight times speedup in SIMD based processors, but the whole application can't achieve such an excellent speedup [50]. There are a lot of functions for scalar data in applications, which can't be optimized by SIMD instructions. It's a waste to use SIMD register and pipeline to process scalar data. Thus a simple 32-bit RISC pipeline is closely coupled with SIMD pipelines in SVP. To reduce hardware cost, RISC pipeline doesn't support MMU and 32x32bit MAC. It is designed to release vector pipeline from scalar operations, which only costs 23 K gates (default: 40K).

In [51], average ILP of MPEG2 encoder and decoder are measured by Trimaran compiler. They are less than 2 ILP on average. Thus dual-issue core architecture is applied in this processor, including two SIMD instructions or one RISC instruction with SIMD instruction. SIMD based vector core consists of 32-bit scalar pipeline (SP), 128-bit SIMD based vector pipeline (VP), mixed pipeline (MP), 2D-DMA, Snoopy Unit, 16KB L1 cache and 8KB Vector Memory (VM) as in Figure 2-2. VP supports a lot of common 8/16/32 bits SIMD instructions, except 32x32 bits MAC. SP is the small RISC pipeline without MAC. Vector instructions which needs scalar operand (such as Vector load/store, extract/insert), is executed by MP. As parallelization of MP and VP requires a 5-port register file (RF), Vector RF is divided into two 3-ports RFs as in Figure 2-2. Compared with unified 5-port RF, it can reduce about 38% power and 31% area. VP can only access VM by load and store instructions. VP has a special instruction for exchange some small data with other cores. 2D-DMA DMA engine is in charge of moving data between VM and main memory. Both source and destination can have a different index for x or y directions.

For most multimedia applications, 8-bit data is used for denoting one pixel. 8-bit SIMD instructions (including 8-bit MAC) can achieve about 2 times better performance than 16-bit SIMD instructions for some functions. However, 8-bit SIMD instructions

require writing back 256-bit vector to vector register file (V_RF). 256-bit V_RF costs too much, thus most of processors [11] [16] don't support 8-bit MAC instructions and use 16-bit MAC instead. In this architecture, the temporal results of 8-bit SIMD operations are stored in 256-bit "acc" registers and only saturated 128-bit result can be written to V_RF. Then 8-bit MAC instructions are supported with small hardware cost.

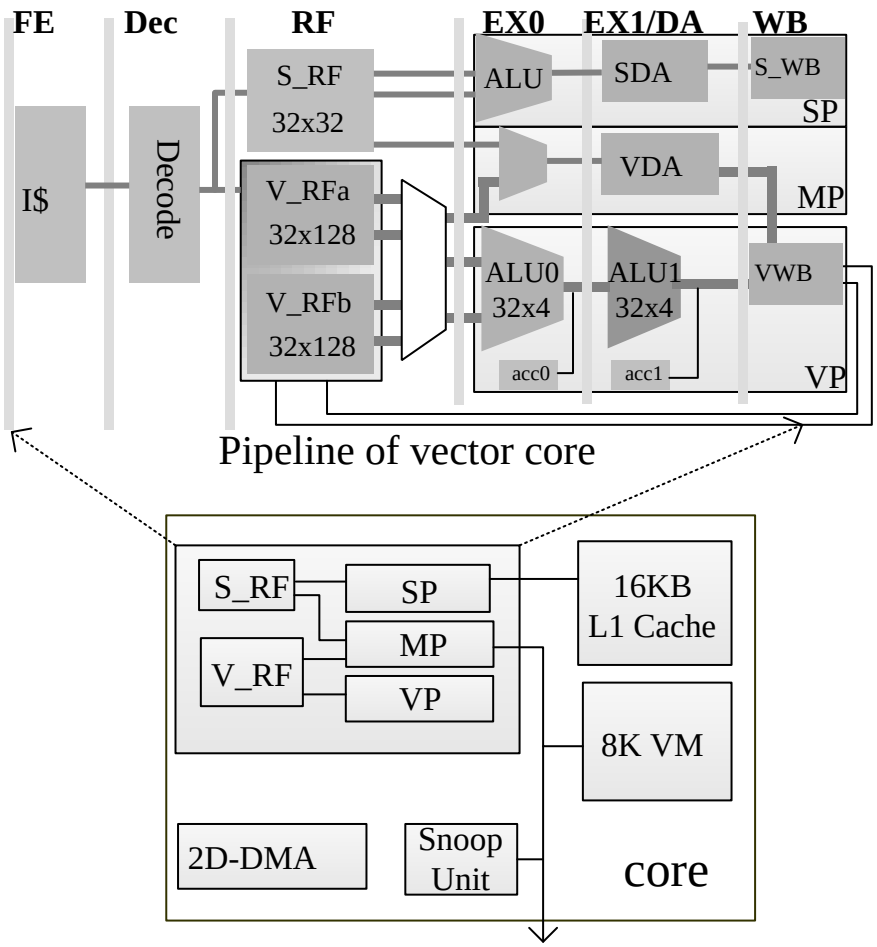


Figure 2-2 Memory configuration in a four tasks example.

2.2.3 Memory System

For single core chips, L2 cache's miss rate is the most important things. However, bandwidth of L2 cache becomes more important for large-scale SMP chips. In this 32 cores processor, DMA units are in charge of moving data between VM and L2 cache. Memory access delay becomes less important for DMA access. Another problem is that MCI protocol is just designed for resolving L1 cache's coherence problem. Adding one more coherence protocol for L2 cache will increase a lot of hardware cost and complexity. Thus L2 cache in this design doesn't keep duplicated copies of the same cache line. It means conventional private L2 cache isn't suitable for this chip.

High BW unified L2 cache designs such as [36] seems to be satisfied. However, its crossbar and L2 controller with directories cost a lot of hardware resource. It's not suitable for our low power embedded processor. New low power L2 cache architecture is proposed to reduce power consumption as in Figure 2-3. The L2 cache is divided into four units: L2A, L2B, L2C and L2D. Each unit is a private L2 cache for one cluster. L2 cache line (64 bytes) is four times of L1 cache line. To achieve high BW, L2 cache units are also divided into four data banks ("Ba"~"Bd"), which can be accessed independently. To save die area and power, only two-way associative architecture is used for L2 cache. L2 cache's tag unit consists of two dual-port SRAM units and can offer two access ports. A duplicated tag unit ("Tb") which contains almost the same data with "Ta" is also applied to double the BW. Figure 2-3 shows that the tag units "Ta" and "Tb" can produce four different addresses for four banks. Then four different data paths "data/B/C/D" can be accessed. Thus, one L2 cache unit can finish four data requests within one cycle. L2 cache controller assigns the incoming data access, and arbitrates access conflictions.

The disadvantage of private L2 cache is that data sharing between different clusters becomes inefficient and costs more BW. The MCI protocol can't guarantee data coherence for private L2 caches. For resolving the coherence problem and improving

the data sharing between L2 caches, special data access paths between different L2 cache units are built for sharing common data. Four L2 units don't keep duplicated cache lines. For example, if cache miss occurs at L2A, these paths can be used to check other L2 cache units one by one as Figure 2-3. In this process, the other L2 cache units provide data for one private L2 cache like L3 cache. Case "A/B" shows two kinds of data access pipeline and their cycle counts. "Mux" denotes the crossbar stage, and "L3" denotes checking other L2 cache units. In this design, it costs 3 cycles to check one L2 cache's tag unit. If cache miss occurs in all L2 units as case "A", the whole process costs about 22~28 cycles. In case "B", cache hits at the second L2 cache, and it just costs 10 cycles. The hit data won't be copied into other L2 cache unit, it's directly sent back to cores. However, most of cache misses on private L2 cache can't be found in other L2 units. It costs too much resource to send requests to other L2 cache units every time. In MCI protocol, memory space is divided as shared and private space. Thus, L2 cache units can also use memory shared information to determine whether to send out data request or not. Case "C" shows that data request is directly sent to MIU for private data, which can reduce a lot of redundant data requests. The proposed architecture can achieve 192 GB/s BW for the 32 cores. The performance of memory access time is evaluated in Chapter 5.

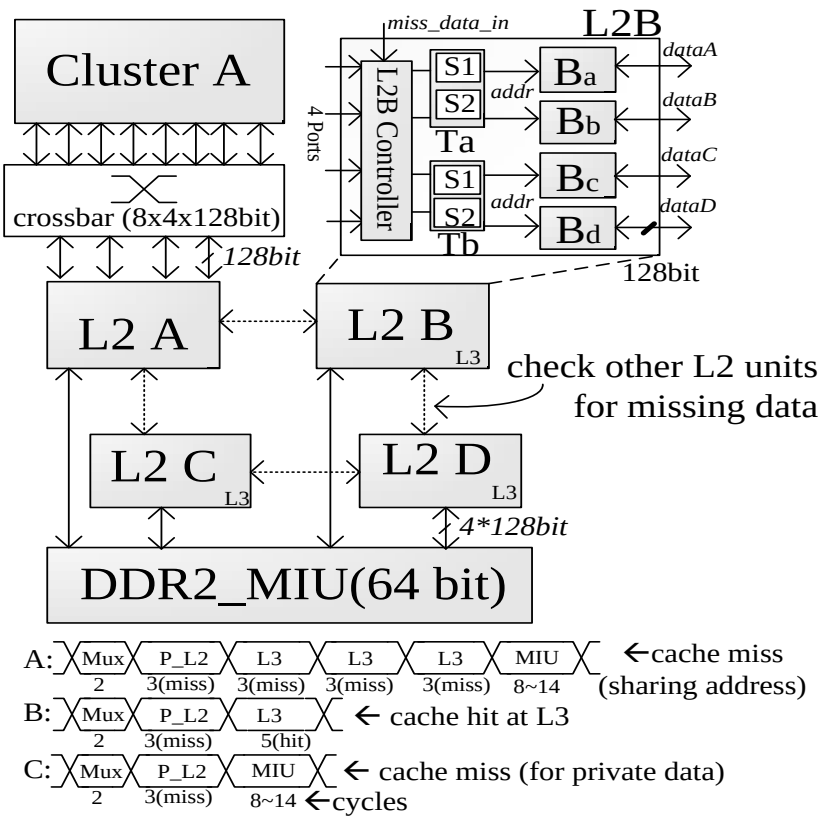
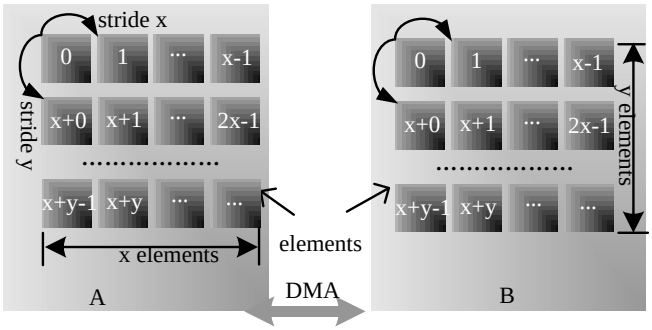


Figure 2-3 L2 cache architecture and cache access pipeline

2.2.4 2D-DMA

DMA engine handles all of communications for VM, including data transfer between main and VM, and data transfer with different VMs. To improve performance of small size DMA, DMA unit utilizes the L2 cache for access external memory. In this processor, cores are connected by CIB. Thus DMA between VMs utilizes CIB. DMA engine supports four channels and is controlled by RISC core. Each channel has two DMA-configuration registers (DCR0 and DCR1 as in Figure 2-4. DIR flag in DCR0 defines the transmission direction: from “A” to “B” or “B” to “A” as Figure 2-4. Stride x is index for each data elements per line and Stride y is index for each line’s start address. 2D DMA consists by several data elements transfers. Address of each elements is determined by Stride x and Stride y. 2-demenional DMA (2D DMA) is supported by using stride x and y. Source and destination can be a different

2-demenional pattern. For example, this 2D DMA supports transfer a 4x4 array to a 5x4 array by using different stride y for source and destination. This is very useful for vertical access in vector memory.



DCR0 (64 bits): DMA configuration register 0

SXA	SXB	SYA	SYB	EN	RDY	ERR	DIR	TC	size
7	15	31	47	48	49	50	51	56	

SXA/SXB: stride x of A/B, 8 bits
SYA/SYB: stride y of A/B, 16 bits
size: data size of elements, 7 bits
DIR: direction of DMA, from A to B or B to A
TC: target core, when B is VM.

DCR1 (64 bits): DMA configuration register 1

addr_A (16 bits)	addr_B (32 bits)	x_A	y_A	x_B	y_B

x_A/x_B: number of element in X (4 bits: max 16 elements)
y_A/y_B: number of element in Y (4 bits: max 16 elements)
addr_A: VM's address 16 bits
addr_B: VM/external memory's address, 16 bits/32 bits

Special Instruction for configuration:
SetDMA(Ra, Rb, channel, DCR0/1, En_flag);

Figure 2-4 Configuration of 2D DMA transmission

A lot of information should be sent to DMA engine, which uses a lot of cycles for small DMA transfers. To reduce the control information of DMA, data size of elements is limited to 7 bits, which means the maximal size of one element is 128 words (minimal: one word). One 64 bits data path is built between RISC and DMA. RISC has one special instruction to configure DMA's channels. It still needs dozens of cycles to combine all of the information into two 32bit registers and then uses "SetDMA" intrinsic instruction to set DMA channel. But if only source address or destination address should be changed, the configuration process needs only two cycles is enough for sending one DMA command.

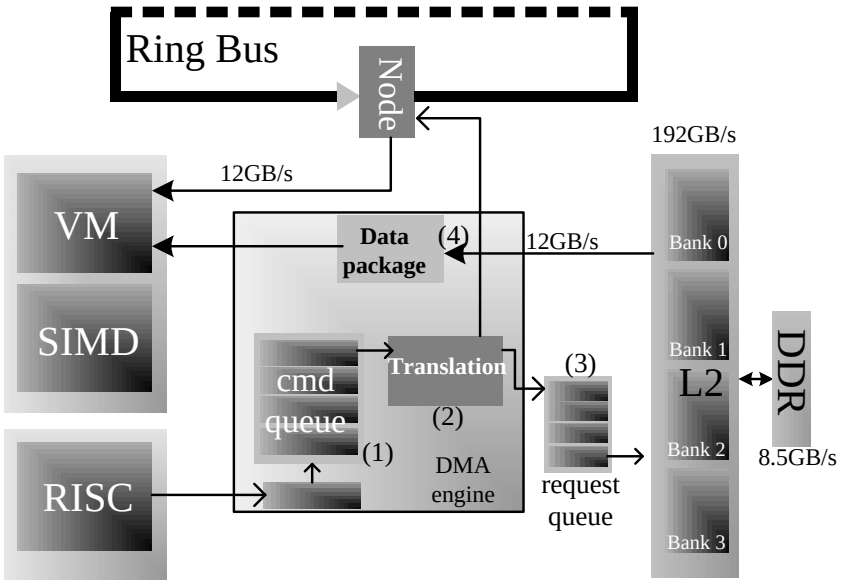


Figure 2-5 Basic flow of a DMA transfer

Figure 2-5 illustrates the basic flow of a DMA transfer between external memory and VM. The process consists of the following steps: First, RISC sets the En_Flag bit in DCR0, and then DCR0/1 are sent to "cmd queue". Second, DMA engine takes one command from queue and translate it to data request. One DMA command may response to a lot of basic data elements. In CELL's DMA design, every basic data element needs to fetch one configuration data from its local memory (costs 10 cycles). And all address of DMA must be translated from virtual to physical address before sending requests to EIB (costs about 30 cycles) [3]. This design just needs about two cycles for these processes. Third, data request is sent to another queue between DMA engine and L2 cache. L2 cache selects requests from all of the request queues and checks its missing-hit status. When cache miss occurs, data request for a whole cache line is sent to external memory. Then data is sent back to DMA unit for packaging into required format according to configurations. On the other hand, if DIR bit in DCR0 sets as inter VM DMA, data request is sent to node of ring bus.

2.3 SIMD Instruction Set Architecture

In this dual-issue core architecture, SIMD instruction just needs to do the arithmetic calculation. RISC is in charge of controlling peripherals and dispatch instructions for SIMD and mixed pipelines. Thus SIMD instructions and RISC instructions are mixed in instruction cache. To make the decode and dispatch process becomes simple, 32 bits RISC instruction set of OR1200 is reduced to 31 bits by reducing the 16 bits immediate to 15 bits as below:

RISC instruction: **l.addi** (add with a 15 bits immediate)

- syntax: l.addi $rD,$rA,$lo15
- format:

bits	31--26	25--21	20--16	15--1	0
l.addi	opcode	Register 1	Register 2	uimm15	RISC
value	0x27	rD	rA	lo15	0

SIMD instruction: **S.addi16** (16-bit vector add with 14 bits immediate)

- syntax: S.addi16 $VD,$VA,$lo14
- format:

bits	31--28	27--22	21--16	15--2	1	0
S.addi16	opcode	register 1	register 2	uimm14	Parallel	SIMD
value	0x3	VD	VA	lo14	1	1

The SIMD instructions have one more bits for denoting parallel status. When the parallel flag is “1”, then this instruction is executed at the same time with next instruction. This design supports two instructions (not in the same pipeline) working in parallel. Fetch unit reads and align instruction according the last bit and parallel flag of SIMD instructions. Decode unit utilize the last bit for dispatching instructions.

This dual-issue vector cores mainly support seven kinds of instructions, which are executed by VP or MP. SIMD instructions in different pipelines can be executed at the same time. Arithmetic instructions are mainly executed in VP, and the other instructions which just need one vector operand are executed in MP. Table 2-3 shows the data types and cycle delay of them. ALU is fully pipelined to accept sixteen 8x8 bit multiply-add (MAC), eight 16x16-bit MAC, or four 16x32-bit multiply operation. For add/subtract operation, 8 bit, 16 bit and 32 bit vector are all supported.

Table 2-3 Kernel Instruction Set of SIMD

Operations	Supported data types and description	Unit	Cycle
Add, Subtract	16x8b, 8x16b, 4x32b, saturating	VP	1,2(32bits)
Multiply+Add	16x8b,8x16b,saturating,accumulation	VP	1(8bits),2
Shift	arithmetic/logical,16x8b,8x16b, 4x32b	MP	1
Logic operation	And, Or, Xor, Extract/insert	VP	1
Package	4x4 transposes (8bit), pack, unpack	MP/VP	1
Load/store	word unaligned, with offset per word	MP	3
Communication	128 bits+32 bits,128 bits, 32bits,to CIB	MP	1

To reduce hardware cost of vector register file (RF), 8x8-bit MAC operations just can write back a 128 bits saturated result back to register file. 8x8-bit MAC instructions perform integer multiply on each 8-bit elements from source 1 register (Va) with the corresponding 8-bit word element in the source 2 register (Vb), and accumulate the 16-bit product with the corresponding 16-bit element in the accumulator (Acc). The operation occurs in parallel for all sixteen elements that are found in Va and Vb. The final result is written to the accumulator (Acc). The saturated results are optionally written back to register file.

MP pipeline has a port for sending data to CIB, which can be used for access the other core's VM or exchange data in RF. Package instructions have a special byte based 4x4 matrix transpose instruction, which is designed for vertical memory access.

Figure 2-6 presents details about a 32 bits unit of ALU0 and ALU1 for add, subtract and MAC operations. A0~3 is the four bytes of operand A, and B0~3 is the four bytes of operand B. C0~3 is the saturated results for 8-bit MAC. C0_16 and C1_16 is the 16x16-bit results. A vector core has four 32 bits unit, including sixteen 8-bit adders/multiplier, sixteen 16-bit adders, eight 16-bit multipliers, eight 32 bit adders.

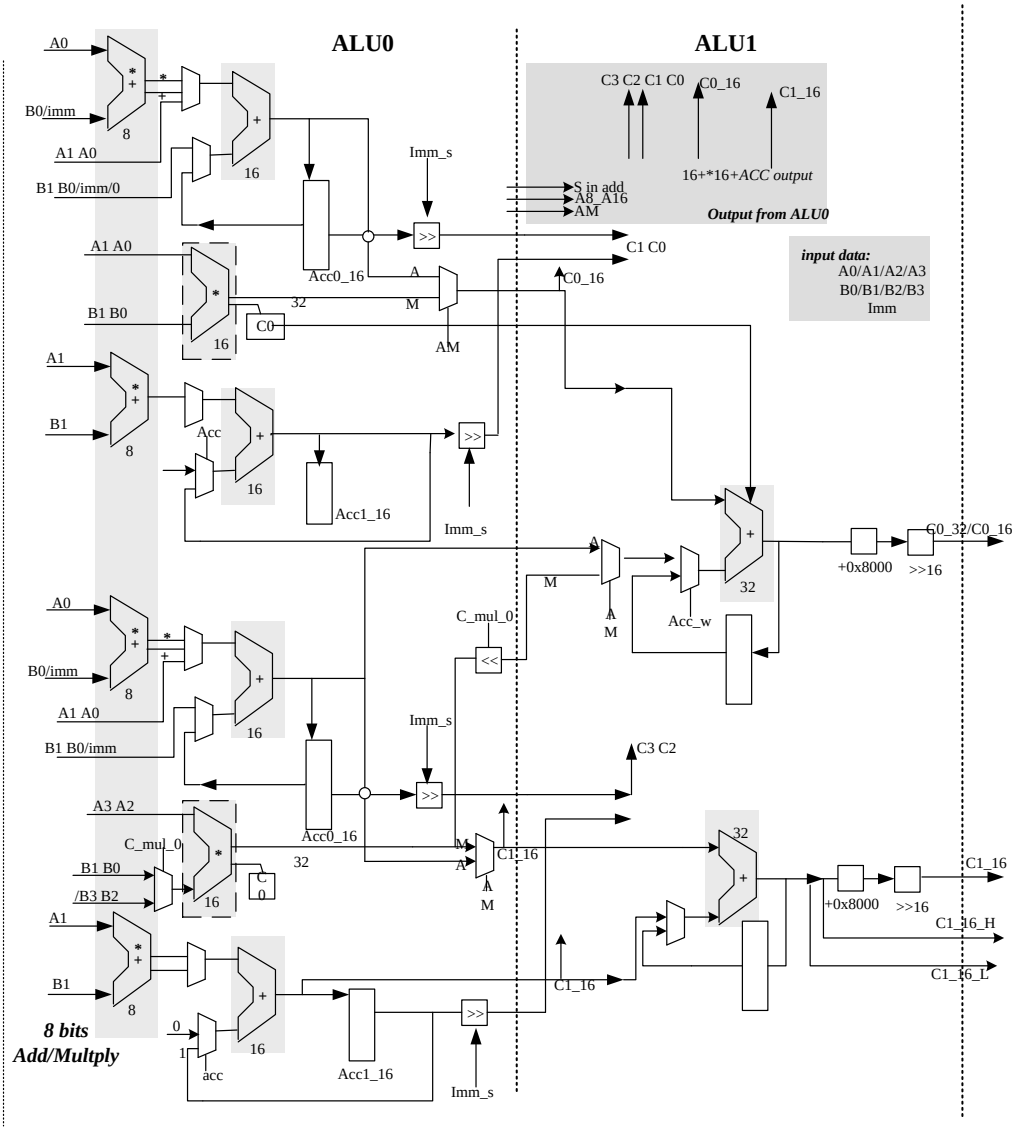


Figure 2-6 The 32 bits arithmetic unit of ALU0 and ALU1

2.4 Vector Memory Architecture

Vertical vector and unaligned data access widely exist in video or image processing. Most of previous designs such as CELL's SPU just support aligned 128 bits data access for local memory. To improve the data access ability for vertical vector, one four banks VM is designed for supporting vertical and unaligned vector types. It costs too much hardware source for dividing the VM into sixteen banks. In this design, VM is divided into four addressable banks, which also enable unaligned access (word address) as in Figure 2-7. Vector data is firstly fetched from external memory by DMA unit. Figure 2-7 shows an example of vertical vector access.

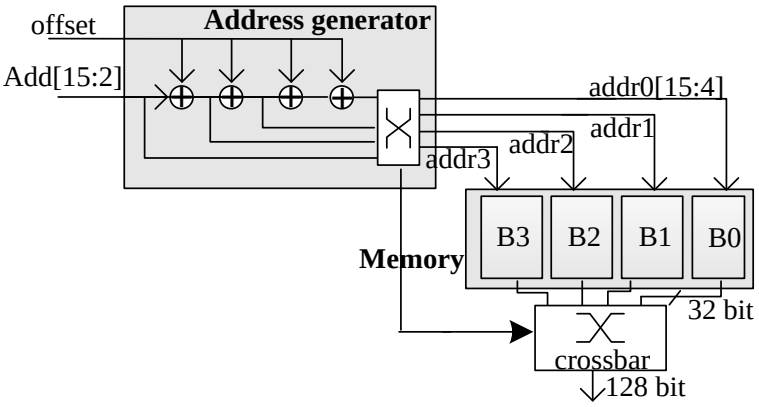


Figure 2-7 The architecture of vector memory with a address generator

Vertical Load/store instructions include register numbers, address of VM (16 bits) and Stride value (8 bits). Stride value is used to denote data array's width. To enable vertical access, each element of vertical vector must be putted into different banks in VM. For this design, array (a,b) must be stored as (4n+1,b) in VM ($4n-4 < a \leq 4n$), and then it can be accessed by Vertical Load/store instructions (Stride value: 4n+1).

For example, a 4x4 data array (*(0,0)~(3,3)*) is stored in VM as 5x4 in Figure 2-8. 2D-DMA unit supports such kind of data movement from external memory to VM. The fifth column is useless data, which denotes as (x,x). Then address generation unit (AG)

can generate four bank addresses for B0~4. To access the first column, “offset” equals to 5, and then AG outputs 0/1/2/3 for “addr0/1/2/3”, which are 16-byte addresses. And then data (0,0)~(3,0) is selected. For the second column, address is “1”, and AG outputs 0/1/2/4 (16-byte address) for “addr0/1/2/3”. However, the output data from four banks is in an order as (3, 1), (0, 1), (1, 1), (2, 1). The crossbar unit reorders them as (0, 1), (1, 1), (2, 1), (3, 1). For unaligned vector access, “offset” always equals to 1. For example, when address is “1” and stride value is “0”, AG outputs 1/0/0/0 for “addr0/1/2/3”. This architecture just can access 32 bit vertical vector.

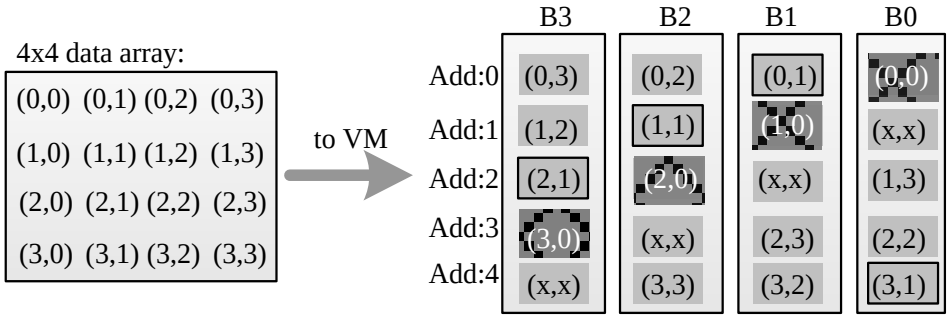


Figure 2-8 A vertical memory load operation for a 4x4 array.

2.5 Cycle Level Performance for Video Kernels

To achieve high performance for multimedia applications, the proposed vector core’s SIMD ISA is optimized for processing 8 bits pixels with a lot of 8 bits MAC or ADD instructions. The vector core is divided into two SIMD pipelines and one RISC pipeline for better instruction level parallelism. For saving costs, there are no duplicated units in two SIMD pipelines. The 64 registers are also divided into two groups for achieving more access ports with less cost.

To verify thee efficiency of the proposed SIMD ISA and core architecture, cycle level evaluations are performed for five kernel functions of H.264 codec: horizontal

6-tap filter, vertical 6-tap filter in ½ pixel interpolation, 16x16 SAD calculation, vertical deblock, and horizontal deblock. They are the most frequently used functions in a H.264 decoder. Figure 2-9 shows the calculation process of vertical 6-tap filter. The row of "h" is calculated by six horizontal vectors (A, C, G, M, R, T). With 8-bit MAC instructions, it just needs six load instructions, six 8-bit MAC instructions and one shift instruction for calculating 16 pixels. For the SIMD ISA without 8-bit MAC such as CELL, it needs 12 unpack instruction for changing the six 8-bit vectors to sixteen 16-bit vectors. And then it needs ten 16-bit MAC instructions for calculating. Thus the proposed 8-bit MAC can save more than two times of cycles in this case.

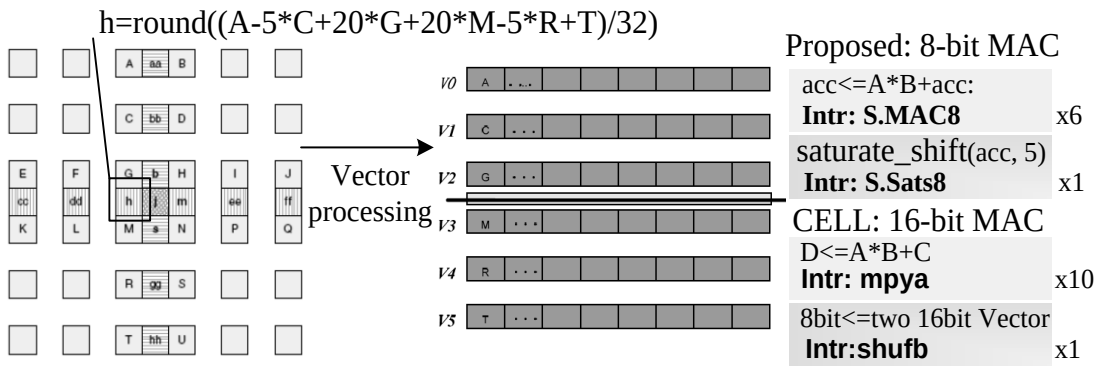


Figure 2-9 The calculation process of 6-tap filtering by SIMD instructions

In vertical deblocking process, vertical vectors are calculated. Figure 2-10 shows how to load vertical vectors in to registers. Firstly the pixels must be loaded into VM as Section 2.4. Then vertical load instruction can load a 4x4 array into one register. In Figure 2-x, a 4x16 array is loaded into four registers (R0~3) by four instructions. Next, four 4x4 matrix transpose instructions are applied for transposing the 4x4 array in vectors. Eight package instructions are needed for packing the pixels of the same column together and produce four vertical vectors such as V0/V1. For CELL, it needs about 44 instructions for loading four vertical vectors. This design needs sixteen instructions for loading four vertical vectors, which is about 2.75 times faster.

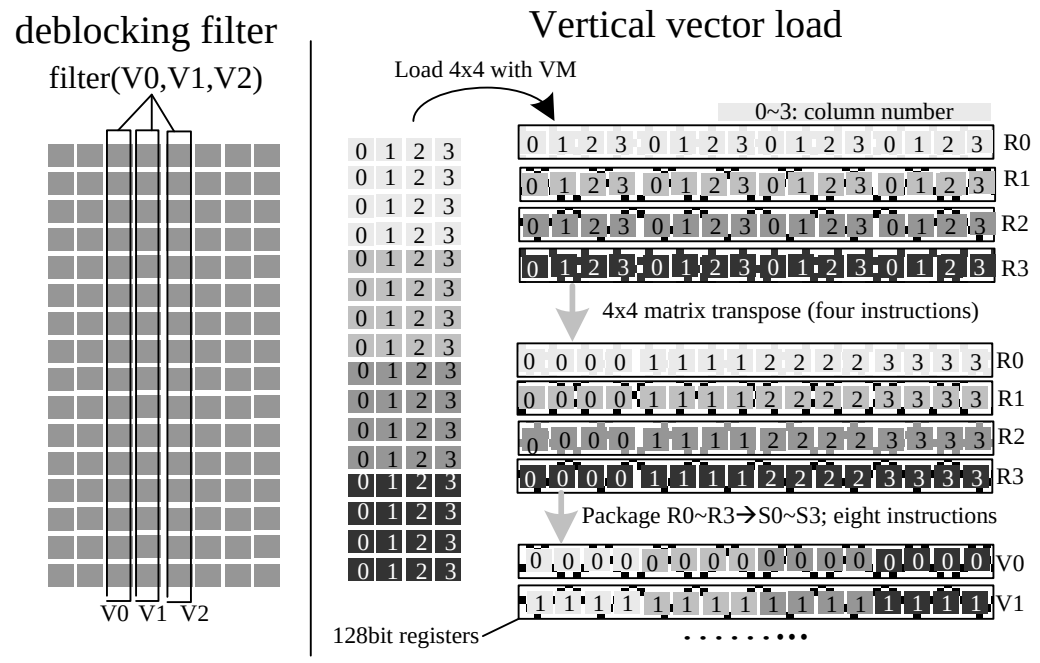


Figure 2-10 loading vertical vectors with VM and transpose instructions

All of the features and its benefits are shown in Table 2-4. Before optimization with the three features of this design (a/b/c), this design is slower than CELL. After optimized with the features one by one, it can achieve about 2.3 times speedup in average. As CELL's SIMD core doesn't support 8-bits MAC or non-alignment access, the proposed work can reduce about 29% cycles in average than CELL. However, this design doesn't support 32-bit MAC, and target applications are limited.

Table 2-4 Vector Core's features and performance (cycles/MB)

Features of Vector Core	(a).Dual-issue, parallelize data access and scalar instructions (b). 8b MAC with saturation, 8b add/sub with accumlation as:acc<=A*B+acc, acc<=A0 +/- B0+acc; C<=Sat(acc) optional. (c).Vertical vector access, 4x4 matrix transpose instruction				
Functions in H.264	without (a,b,c)	with (a)	with (a,b)	with (a,b,c)	CELL
Horizontal 6-tap filter	495	447	225	225	320
Vertical 6-tap filter	269	231	117	117	221
16x16 SAD	73	58	38	38	53
vertical deblock 16x16 (Luma)	1027	983	857	331	488
horizontal deblock 16x16 (Luma)	504	466	229	229	232

2.6 Chapter Summary

To achieve high performance for multimedia applications, the proposed vector core's SIMD ISA is optimized for processing 8 bits pixels with a lot of 8 bits MAC or ADD instructions. The vector core is divided into two SIMD pipelines and one RISC pipeline for better instruction level parallelism. Results in [51] shows that the average ILP of target applications is about two, then this architecture supports issuing two instructions per cycle. The cost of high ILP is very high, and the performance gain/cost is poorer than multicore or SIMD architectures. For saving costs, there are no duplicated units in two SIMD pipelines. The 64 registers are also divided into two groups for achieving more access ports with less cost. Cycle level performance evaluations prove that this vector is very efficient for video applications.

3 Application Specified Cache Coherence Protocol

In conventional snoop cache coherence, coherence transaction broadcasts to all cache monitors. When a processor issues a request to its cache, the other cache controllers has to examine the state of its own cache and takes suitable action, which may generate access operation to tag or data unit of cache. In snoop-based systems, all coherence transactions are broadcasted and therefore seen by all processors in the system. As a result, snoop protocols are generally limited to small-scale systems. Results in [52] indicate that more than 70% percentage of all snoop broadcasts miss in other caches. This means that most of the snoop induced tag-lookups just waste energy. Thus an application specified cache coherence protocol is proposed in this chapter, which can reduce about 67.8% L1 cache energy for 32 cores SMP processor. In this protocol, memory should be configured as private or shared spaces. It just sends out invalid messages when writing operation occurs on sharing spaces. Programmer should be aware of memory sharing methods, and manually control the protocol. Thus it's called manually controlled invalid protocol (MCI).

In section 3.1, the broadcasting nature of conventional snooping protocol is analyzed. The coherence operations increase rapidly with the core number.

In section 3.2, the MCI protocol is proposed in details.

Section 3.3 shows the programming method with MCI.

In section 3.4, a snooping unit is designed for MCI.

Section 3.5 shows performance evaluation of MCI.

And section 3.6 gives the summary of this chapter.

3.1 Wasted Energy in Previous Snooping Protocols

In small-scale shared memory multicore processor, snooping based protocols are very popular. In such kinds of processors, lower cache hierarchy levels (such as L2) are always shared by all cores. In a typical MOESI based consistency model [53], cache miss on shared cache generates a cache miss request to main memory, and then a state checking message will be sent to other core. If other cores hold this data, then it will be marked as shared ("S") in private cache, otherwise it will be marked as exclusive ("E"). When a core modify a shared cache line, one more snooping message should be broadcasting to other cores, then the owner of the new data is changed from "S" to owend ("O"), and the other cores should change from "S" to invalid ("I"). Two snooping message are broadcasted in this case. For a 32 cores system, it means that the other 31 cores need to check their private cache two times for one data modification. If there are only two cores have that data, then 60 cache tag checking operations are wasted. It also means that snooping operations occupy more than 31% of L1 cache with 1% miss rate in L1 cache (L1 is private, and L2 is shared).

Simulation results show that most of snooping operations missed in all the others cores as Table 3-1 for an eight cores SMP processor [52]. One snoop operation results a tag checking operation in one cache. If it doesn't hit in the cache, then this tag lookup is useless and doesn't leads to any response. "0 hit" means the other cores don't have that data, and the snooping operations are wasted. "7 hit" means the other cores don't have that data, and the snooping operations are wasted. We define a waste rate for snooping operations as (3.1). Table 3-1 shows that more than 85% snooping operations in private cache is wasted.

Waste rate =hit operations / total snooping operations (3.1)

Table 3-1 Snoop hit distribution in other cache units [52]

Benchmark	0 hit	1 hit	2 hit	3 hit	4 hit	5 hit	6 hit	7 hit	wasted
FFT	99.4%	0.2%	0.1%	0.1%	0.1%	0.1%	0.1%	0%	99.8%
Raytrace	79.1%	11%	3.7%	2.4%	1.8%	1.2%	0.6%	0.2%	94.2%
MPEG	96.4%	1.7%	0.3%	0.2%	0.3%	0.4%	0.5%	0.1%	98.6%
Barnes	53.3%	17%	12%	8.6%	5.2%	2.5%	0.8%	0.1%	85.9%
Radix	99%	0.7%	0.2%	0.1%	0%	0%	0%	0%	99.8%
Average	85.4%	6.2%	3.2%	2.3%	1.5%	0.8%	0.4%	0.8%	95.6%

3.2 The Proposed MCI Protocol

There are a lot of techniques for reduce snoop-induced power in multiprocessors, such as serial snooping [24] and Jetty [27]. Serial snooping scheme is based on assumption that if a miss occurs in one cache, it is possible to find the block in another cache without having to check all the other caches. Serial snooping only works for snoops which are induced by a read miss. Jetty [27] is a prediction structure for filtering out useless snoop accesses. Before doing a tag lookup, the Jetty is first checked. Jetty can reduce about 16%~56% snooping used energy for an 8 cores processor [52]. Its performance depends on applications and size of prediction structure. For example Jetty's performance is very low for Radix application. On the other hand, the prediction structure costs a lot of extra power and hardware resource.

To resolve the bottleneck of snoopy-based protocol and reduce the hardware cost, we propose the MCI protocol for large scale processors. MCI doesn't have "Shared" state, and it also allows that blocks don't have any state. MCI just has "M" and "I" state. In the proposed MCI protocol, memory space can be dynamically or statically defined

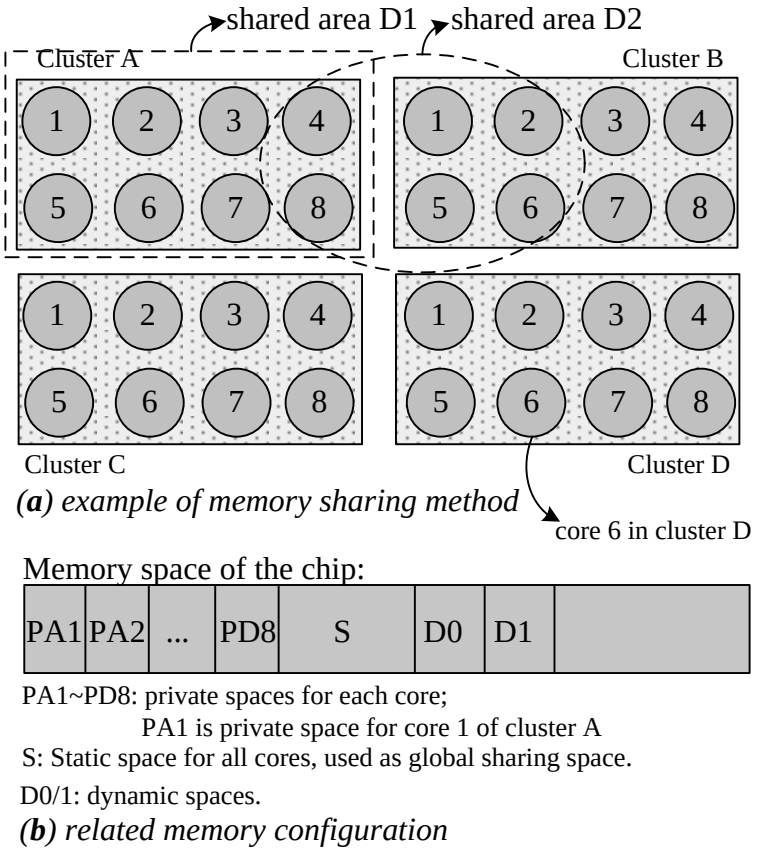
as shared or private space. The data blocks in sharing space are treated as shared data in MCI. If data blocks in sharing space are modified, then the new data block is marked as "M", the other data blocks are changed to "I". If a read request arrives at a block in "I", then cache miss occurs and the request will be sent to lower level cache.

The shared spaces can be shared by all cores, clusters or several cores. Only write operation in shared address will produce an invalid message to the sharing cores, and the status becomes to "M". In the proposed architecture, L2 cache units don't keep duplicated cache lines, and duplicated cache lines only occur in private L1 caches in sharing spaces. An extra snooping unit is necessary for managing the sharing configurations and sending out invalid message. When a core receives an invalid message, it checks its L1 cache's tag units and marks the old data as "I".

Sharing configuration includes three types of information: address ranges (64-bit), shared clusters (4 bits) and cores (8 bits), and share types. Address range includes start and end address, which means that it just supports blocks memory. Shared clusters and cores are used to denote which cores are sharing this space. All of 32 cores can share data between each other. There is no limitation for data sharing or parallel working. For example, "shared cluster: 0001, cores: 00111111" means that cores 0~5 in cluster A share this space. On the other hand, "shared cluster: 1111, cores: 11111111" means this space is shared by all cores. One sharing space can have more than one sharing configuration. Share types include dynamical and static sharing. Static sharing means that the sharing cores and address range can't be changed in run time. Dynamical sharing means the shared spaced or sharing cores can be changed in run time. Statically shared spaces are always active, and dynamically spaces can be dynamically set as inactive for saving power.

Figure 3-1 shows an example of memory sharing and configurations. Figure 3-1 (a) shows that core 1~8 in cluster A have a dynamic sharing space D1, and core 4/8 in clus-

ter A with core 1/2/5/6 in cluster B share dynamic sharing space D2. Eight cores in cluster a work on the same job, and then they can need share data with each other by D1. Six cores in cluster A/B work on the same job, and then they can need share data with each other by D2. In this example, we also define 32 private spaces PA1~PD8 for each core. Private space can be used for storing private data such as stack or local variables. A global static shared space S is also defined in Figure 3-1 (b), which can be used for sharing data with all cores.



(a) example of memory sharing method

(b) related memory configuration

Figure 3-1 Example of memory configurations and the memory map with MCI

Snooping unit (SU) is a critical unit for MCI. Each core must have one SU for supporting MCI. SU keeps the memory sharing configurations and snoops on the coherence messages. When write requests occur at shared space, SU checks every write request and sends invalid message to sharing cores. For read requests, there are no

snooping operations. Programmer must know well about the memory sharing patterns in applications, and configure the memory map correctly.

3.3 Programming with MCI

MCI protocol applies modified-invalid (MI) consistency model, which is a weak consistency model. If the tasks need sequential consistency, they need to add some synchronization operations in program. An API for parallel programming is also proposed, which can be used for sharing spaces definition, synchronization and task assignment.

3.3.1 Synchronization in MCI

As L1 cache applies write through policy, a filter cache (FC) is used for combining small write operations. When dirty cache lines in FC are displaced (LRU replacement policy), they are written through to L2 cache. It means that write operation may become out of order. And the delay between write instruction and writing through to L2 is also uncertain. Figure 3-2 (a) shows the pipeline of three writes operations at shared spaces. There are three write operations (WR_0~2) in a shared space. But the write through operations (WT_0~2) occur out of order. According to MCI, the snooping unit may send out invalid messages (MCI0~2) follow the write through operations. Before the invalid message arrives at the other cores, the read operation will get the old data. And this period is also uncertain. If two cores write to the same address at the same cycle, the result becomes uncertain.

Therefore, data sharing between cores needs to add some synchronization operations. Figure 3-2 (b) shows the pipeline of two write operations with synchronization (Syn). FC flushes the dirty data back to L2 cache immediately after the “Syn” instruction, and then snooping unit sends the invalid message. After FC is empty, a synchroni-

zation message (Syn_0) is sent to the other sharing cores. When other core gets that message, the new data can be read from L2 cache. It's inefficient to send synchronize every write operation at sharing spaces. It's better to combine the write operation together, and reduces synchronization process.

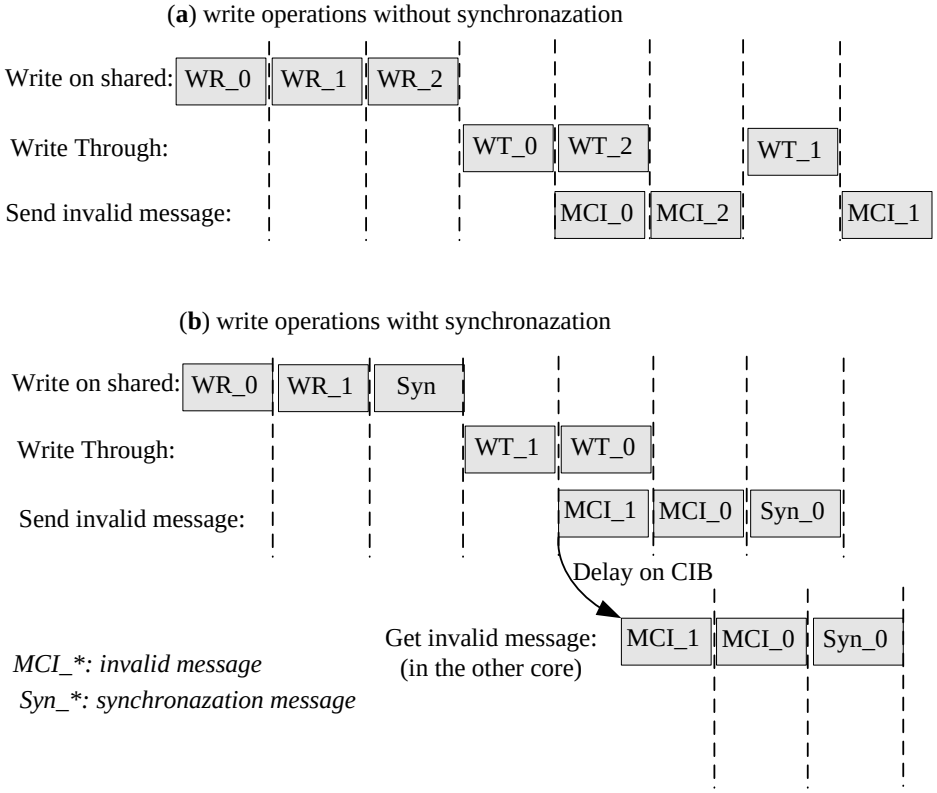


Figure 3-2 The data flow of writes operations with/without synchronization

3.3.2 Parallel Programming Model

OpenMP provides a relaxed-consistency parallel programming model, based on shared-memory architecture [22]. Each thread has access to another type of memory that must not be accessed by other threads, called thread private memory. There are two kinds of access to variables used in the associated structured block: shared and private. In MCI, the whole memory map can be configured as shared or private spaces, and shared spaces also have more attributes than OpenMP. Then we propose a new pro-

gramming model with an additional memory configuration process, and Figure 3-3 shows an example of four parallel tasks.

Two additional steps are needed for memory sharing configurations in MCI. Firstly, memory map should be defined according to MCI. Directive "#Program Share" is used to define the sharing memory patterns, including address range, a set of shared cores and shared type. Shared space "Exchange" is defined as a statically shared space by cores 0~3 in cluster A. Three dynamically shared spaces "CA*" are defined to be shared by two cores. Second, the defined memory spaces are assigned to tasks. Directive "#Program Parallel (C0)" is used to assign task "FuctionA" to core 0, and then two shared spaced "CA0_1" and "Exchange" are assigned to core 0. "Exchange" is used for sharing data with all of cores which are working together. Dynamically shared spaces "CA*" are used to share data between two cores. Undefined space is private.

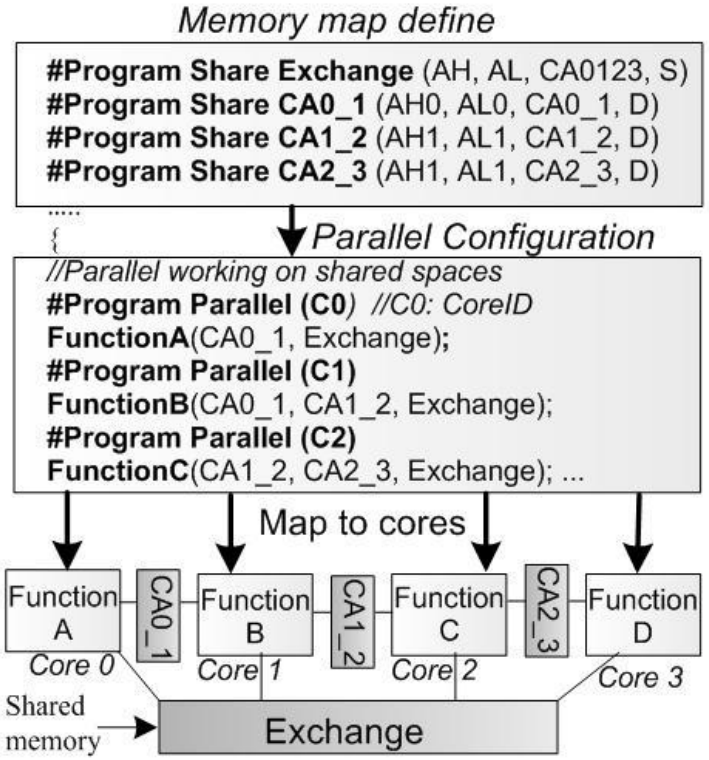


Figure 3-3 Memory configuration in a four tasks example.

3.4 Hardware Supports for MCI

3.4.1 Snooping Unit

In our chip, each core has 16 sets of sharing space configurations (CF0~15) and 4 active configurations as in Figure 3-4. "AH*" and "AL*" defines the address range. The chip has four L2 caches and L2 cache can be accessed by the other clusters, providing a shared L3 cache. When L2 cache miss occurs, L2 cache doesn't copy the hit cache line from other L2 caches. Only L1 cache may have duplicated data and the MCI protocol is applied to reduce coherence message. Compared with conventional snoop or directory protocol, no extra hardware resources are used to keep the cache line's share status, saving a lot of hardware resource for large scale SMP. The overhead of MCI is in software development.

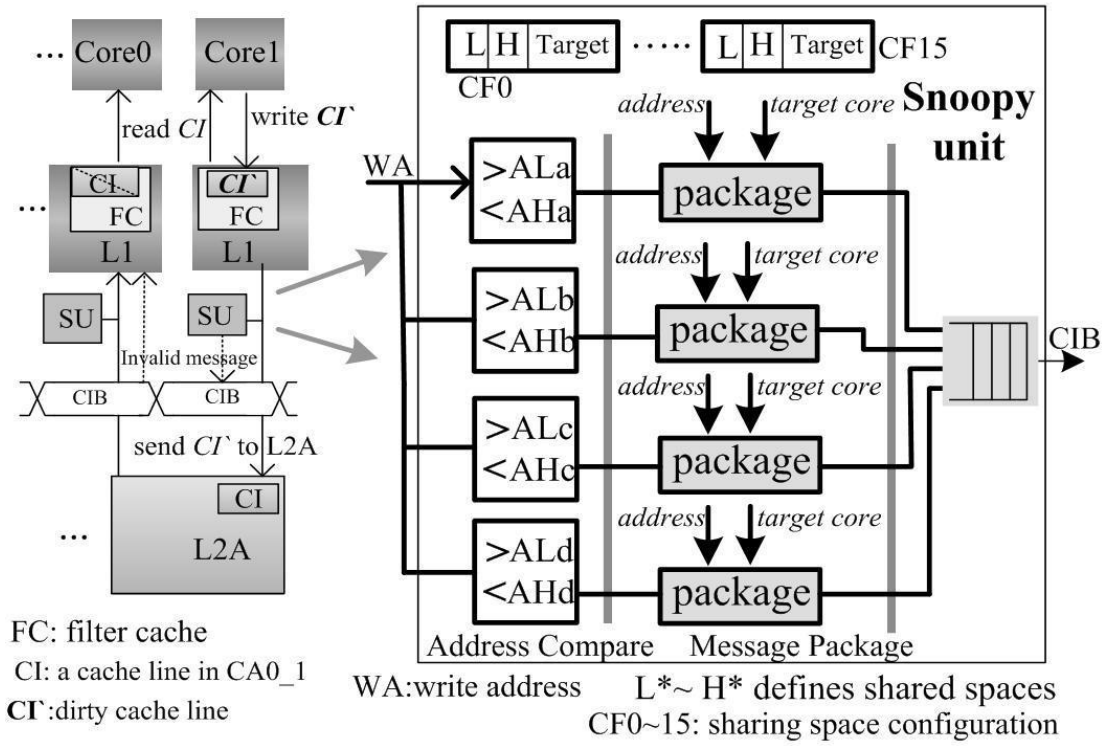


Figure 3-4 Access of shared data in MCI and SU's architecture.

Figure 3-4 also shows an example about how MCI protocol works on shared data access. At first, Cache line I (CI) in shared space "CA0_1" is required by core 0 and 1, and then CI is maintained by L2A and two L1 units. When core 1 writes new data (CI`) to L1, CI` is stored in filter cache (FC) unit. FC unit has eight entries with LRU replacement policy and is also used to combine small write operations. When CI` in FC is replaced and sent back to L2 cache, SU checks the address of this write request. As core 1 writes to a sharing space with core 0 ("CA0_1"), SU of core 1 should send an invalid message to core 0's L1 cache (to guarantee data coherence in CA0_1). Address of CI` and targeted core number are packaged in SU and sent out through CIB. Then the old data (CI) in core 0's L1 caches becomes invalid. Core 1 needs to send out a synchronization message, when core 0 and core 1 need to synchronize the sharing data. In this example, only one message is sent between two cores, and no broadcasting message is sent to the other 30 cores. Coherence operations are reduced from 31 to 1.

3.4.2 Cache Design with MCI

In MCI, the invalid state means empty block in L1 cache. A read request at an invalid block produces a read miss, and then produces a data request to lower cache. It means that the new data should be sent to lower cache firstly, and then sends out invalid messages to other cores. If write back policy is used in L1 cache, then a lot of dirty data blocks will be stored in L1 cache. Then synchronization operation will have to flush all dirty data in L1 and then sends out invalid messages. It takes thousands of cycles for such kind of flush. In MCI, synchronization operation is more frequently than others. Thus write back is not suitable for MCI.

A write through based L1 cache with a special filter cache (FC) is designed for MCI as Figure 3-5. FC unit is used to reduce the energy of L1 data cache. FC is different from conventional write buffer as it can be used to combine small write operations.

It maintains eight entries for recently accessed data cache line (16-byte). Write-back data is also temporally kept in FC to combine small write-back data. When dirty cache lines in FC are displaced (LRU replacement policy), they are written through to L2 cache. Synchronization operation flushes out all of the dirty cache lines to keep the data coherence. In experiments, it can save about 87% power of L1 data cache in MP3 decoder application.

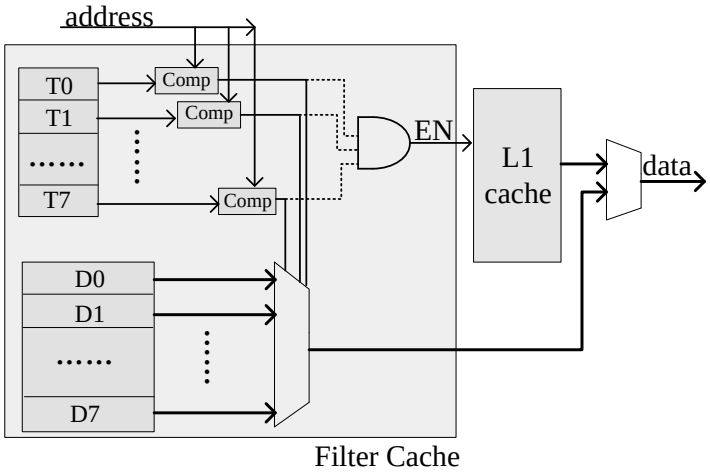


Figure 3-5 The architecture of Filter cache before L1 cache

3.5 Performance Evaluations

MCI needs to add some memory configurations for every program. Thus it's not compatible with conventional software on SMP (memory configurations must be added for MCI). General CPU for PC can't use MCI, as software compatibility is necessary. This is the restriction for our coherence protocol and cache design. Thus we call it as "Application Specified" protocol. However, all of software can be mapped to MCI based processor with some modifications. A lot of embedded processors can also use this protocol, as compatibility isn't so critical for them. Therefore, we drive the experiments with several application of SPLASH2-benchmark. We select a signal processing

application, e.g., FFT, a multimedia application e.g., Raytrace, and a data sorting application, e.g., Radix as our benchmark for L2 and MCI. On the other hand, vector core of SVP is designed for multimedia applications, thus applications of image/video processing are selected for evaluating vector core.

To evaluate MCI, We use Simics [54] to simulate a 32 cores SMP processor, with 8 KB L1 data cache per core and 1 MB shared L2 cache. In experiments, only snoopy operations of data cache are measured. In experiments, four tasks are mapped to four clusters, and each task has eight cores. As snooping based MOESI is widely used in products, we evaluate the original MOESI, MOESI with Jetty and MCI. Jetty is configured with 16-bit address, 32 entries, 2-way associative [27]. In MCI, we define four sharing spaces for four tasks, and 32 private spaces are also used for each core. Sharing data such as “global_memory” in Radix is mapped into one of sharing spaces, and the other local variables are mapped into private space. Then eight cores within cluster work together and broadcasting messages are limited by MCI. Snoop-used L1 cache’s energy is measured for them. In original MOESI, snoopy operations occupy a lot of L1 data cache resource (more than 50%) as in Figure 3-6. Jetty can reduce about 24.6% in average of snooping used energy. MCI can reduce about 67.8% of L1 cache energy in average. Snooping operation of MCI is about 43.2% less than MOESI with Jetty.

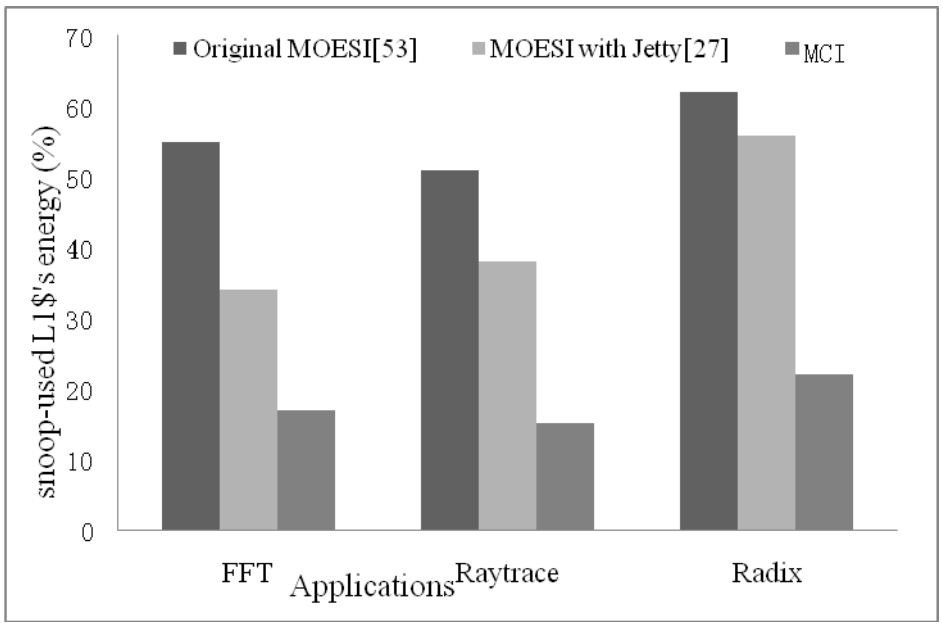


Figure 3-6 Snoop-used percentage of L1 cache energy.

3.6 Chapter Summary

Jetty just can reduce tag lookup operation caused by snooping; the transmission of messages can't be reduced. Compared with Jetty, MCI can reduce both the tag lookup operations and BW cost. The drawback of MCI is software compatibility and complexity. In MCI, cost of snooping is mainly determined by data sharing method, not the number of cores. Large scale SMP processor can achieve good performance in MCI. The software impacts of MCI are the two configuration steps, and the complexity depends on the memory sharing patterns. Thus it's not compatible with conventional software on SMP (memory configurations must be added for MCI). General CPU for PC can't use MCI, as software compatibility is necessary. This is the restriction for our coherence protocol and cache design. A lot of embedded processors can also use this protocol, as compatibility isn't so critical for them.

4 Communication Network

In this chapter, we explore the design of the on-chip communication network and provide insight into its communication method, latency and BW. Section 5.1 presents the background of interconnection network, including the BW decreasing problem for small size transmission. Section 4.2 shows the proposed Core Interconnection Bus (CIB), including topology, data format and packet routing method of CIB. Each flit in CIB is independent and transmitted by time-division multiplexing. Latency of each node in CIB is only 1 cycle. Section 4.3 presents how to connect four CIB for the whole chip. Section 4.4 shows the bandwidth and latency performance of CIB, including a comparison with CELL's EIB [33] and NoC. Then section 4.5 gives a summary of this chapter.

4.1 Background of Interconnection Network

In our previous work on CELL processor, small DMA transmission becomes the bottleneck of performance [55], as CELL just can provide high BW for large data transmission. The cost of starting a small size transmission is very critical for effective BW and latency. This section presents the details of data flow of CELL's EIB and a typical NoC design, and then analyses the cost of starting a transmission of them.

There are two kinds of communication networks for multicore processors: shared bus and NoC. A shared bus based network always consists of bus and central arbitrary. To build a connection with bus based network, cores need to send a bus request to the arbitrary, and then the arbitrary dispatches the available bus to each request and also

manages target and source address of data transmission. For NoC, data packets can route automatically, and transmissions need some extra flits for allocating a channel. Therefore, both of them have some extra cost for starting a transmission. For a large transmission, this cost just occupies every few BW. However, it becomes intolerable for small data transmissions.

The EIB network of CELL processor is consisted by four 16-byte wide data rings, and each ring can support three concurrent data transfers at the same time, when their paths don't overlap. To initiate a transfer in CIB, bus elements must send a message to the arbitrary and request data bus at first. Then the arbiter processes these requests and dispatches the bus. In CELL processor, DMA is used for data transfer between cores. One transmission needs about 7 steps by EIB, including address translation, bus request and coherence checks. These process costs more than 50 cycles [33]. It means that the effective BW for small data is very low. When the data size is less than 16 bytes, EIB just can provide less than 2 GB/s BW which is just about 8% of maximal BW.

On the other hand, NoC doesn't have central arbitrary and connection with NoC becomes simple. NoC is consisted by topology, routing, and channel control. Topology is the arrangement of nodes and channels into a system. Routing method defines how a packet chooses a path in these nodes. Channel control deals with the allocation of channel and buffer resources to a packet. Virtual channel is always used for resolving dead lock in NoC. As the bus width is fixed in NoC, packet is always divided into flits, which have the same width with bus. To build a connection, two extra flits are needed for sending destination memory address in [43], routing information. Figure 1-6 shows a basic packet and the data format of each flit in [43]. One short data transmission needs three flits at least. Six bits is used for controlling the channel. Comparing with bus based network, NoC just needs two extra flits and 6 extra bits for one transmission.

Theoretically, the effective BW of 4 bytes transmission is about 33% of maximal effective BW.

NoC is expected to achieve better performance than bus based network for small transmission, but virtual channels need a lot of buffer resources. The overheads of allocating and managing a channel of NoC are still very high. To resolve this problem, a combined solution of bus and NoC is proposed. A short routing header is added for every flit, and then data paths can be used as time-division multiplexing (TDM). Command and data is transmitted with separated paths. Short data transmission can achieve the same BW as large data transmission.

4.2 Core Interconnection Bus

To take advantage of all the computation power of this processor, workloads must be distributed to 32 cores. In this processor, cluster with a shared L2 cache is the basic processing unit. Data sharing within cluster is supported by the sharing cache. However, it's inefficient to use the shared cache for sharing data transfer between VM, as it needs a write back and read operation to L2. To enhance the data exchange ability for vector core, a ring based network (CIB) is proposed in this section, which can offer very low latency data transfer within clusters. In CIB, every flit has routing header, and they share the links in time-division multiplexing method. One transmission can be divided into independent flits. Each flit can route in CIB independently. It means that virtual channel is unnecessary in CIB. The overhead of building a connection is very few. Thus CIB can achieve more than 10 GB/s BW for all size vector transfers, which is more than 2.5 times better than EIB [33] and NoC [43]. When the injection rate is less than 8%, CIB's latency is less than 4 cycles in average.

4.2.1 Topology of CIB

Clusters need a low latency and high BW network for exchange data within cluster. Thus latency and BW is the most critical factor of CIB. Bus based network can offers lower delay, but the bus requirement brings some delay. For example, CELL processor applies a ring bus based EIB network. It consists of four 16-byte data rings: two running clockwise and two counterclockwise. To initiate a data transfer, each core must send requests to a central bus arbiter, which controls all of the data transfers. The set-up process of a transfer causes a lot of cycles in EIB. Thus the pure bus based network can't satisfy our requirements. In NoC, each data link can be used by all cores, and the data is routing automatically. And then the set-up process becomes simple. However, NoC architectures need a lot of large FIFOs to store data packages in network. In Intel's 6x4 mesh NoC, one router costs about 1.1 mm2 and 500 mW in 45 nm [56]. Obviously it can't be used for this embedded processor.

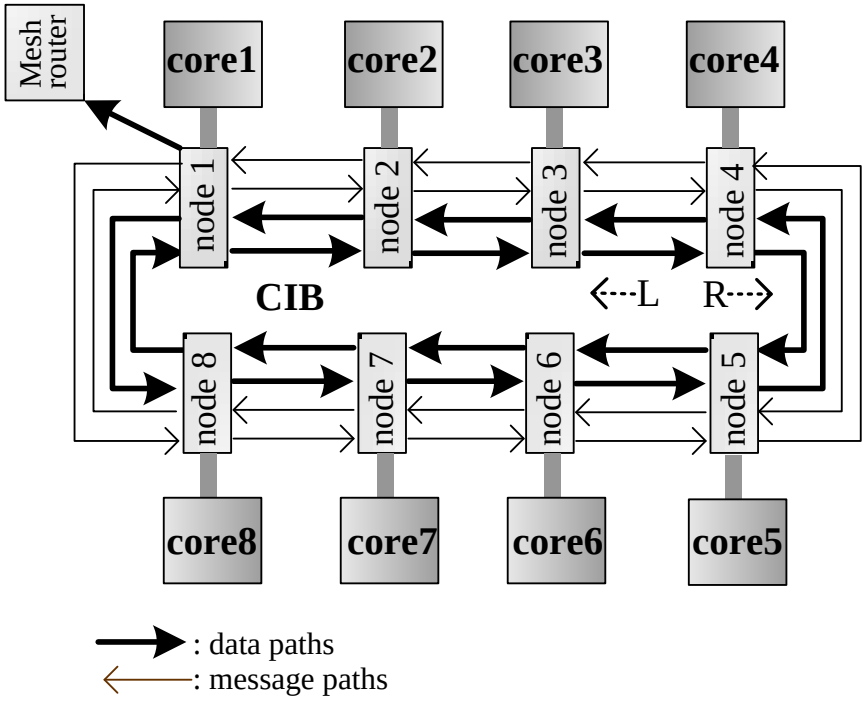


Figure 4-1 Core Interconnection Bus for a cluster.

To take advantage of all the computation power on SVP, CIB is designed to combine bus and NoC together, which can achieve low delay as bus based designs and keep the scalability as NoC. Figure 4-1 shows the CIB, the heart of the SVP's communication architecture.

It consists of two 16-byte data rings and two 4-byte message rings. It supports separate communication paths for 32-bit commands (use 4-byte message rings) and large blocks data movement (use 16-byte data rings). Eight bus nodes in a cluster divide CIB into eight segments. Large data is also divided into 16-byte data elements. Data elements are packaged together with target cores and cluster tag, and automatically transmitted in CIB. To support broadcasting data to all cores, 14 bits extra package header is used for automatically routing in the whole communication network. Bus Nodes in CIB are the basic transfer stations, which can accept four packages (two data packages and two message/command packages), send the arrived packages to core and send out the others to next Bus Nodes. Bus node always deals with the received data packages first, and then sends out the new data packages from core at free slots. Figure 4-2 shows the details of two data links in Bus Node (message links is similar).

Comparing with conventional NoC, there is no input FIFO for incoming data from CIB, only a small output FIFO for core. Bus node also doesn't have large cross bar Data in CIB isn't temporarily stored in FIFO unit. Figure 4-2 presents the details of one bus node. CIB has an output unit, an input unit and a bandwidth management unit (BMU) for data links. Message links is similar with data links, but without BMU unit. The output unit checks the data of two input links. If the routing tag includes this core, then output unit sends it to the core. If current core is the only target of the arrived data, then it will ends in this node. Otherwise, this node needs to modify the routing tag and sends out the regenerated data. More details are shown in next section about routing tag. If

current core isn't the target of the arrived data, arrived data will send out in next cycle without any modification.

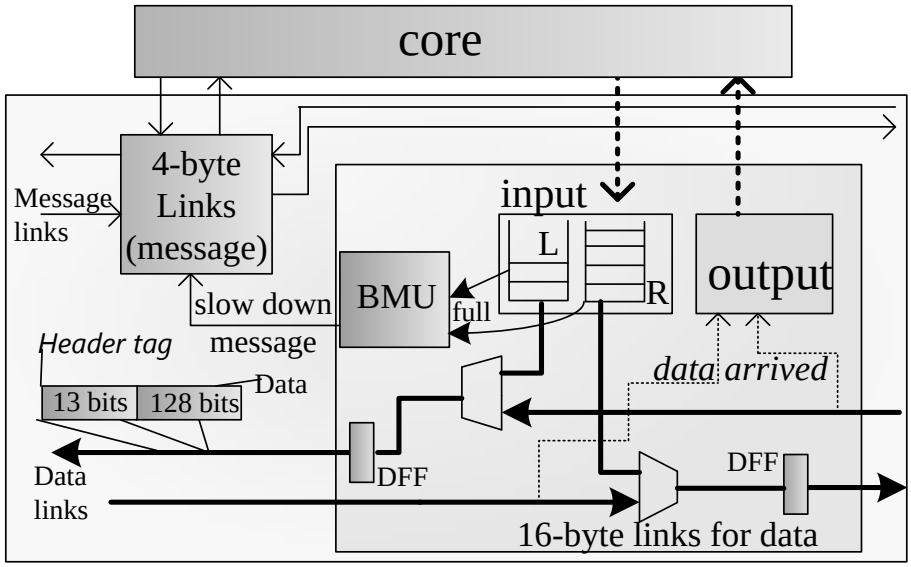


Figure 4-2 Bus Node architecture in CIB

4.2.2 Bandwidth Management of CIB

BMU is used for resolving the resource conflicts and reducing delay for heavy traffic cases. In CIB, one bus node has two output paths (left and right paths) for sending out data. Bus node selects direction which has the near distance. For example, if core 6 needs to send data to core 4, then it always uses the right side path. Thus one data links can be required by four cores at most. For example, the data link from core 4 to core 5 may be required by core 1/2/3/4. Figure 4-3 shows that nodes 1/2/3/4 need to send data to node 5, and all of them are full of data in the nodes. If there aren't BMU, core 1 keeps on sending data and it can achieve the maxim BW: 12 GB/s. However, the other cores have to wait until core 1 is all over, as CIB just can send out data at free bus slots. To smooth the BW and reduce the latency for core 2/3/4, node 2 sends out a slow down message to left side (including core 1) firstly. As node 2 doesn't know which cores are using the bus, slow down message is broadcasted to all possible nodes (nodes

1/8/7). And then data issue rate (right side only) of node 1 is reduced to 25% (only right side path). Node 3 and node 4 also send out a slow down message to left side. And then the whole 12 GB/s BW is assigned to four cores in average. If core 4 doesn't need to send data to core 5, and then core 3 can have more BW as in Figure 4-3.

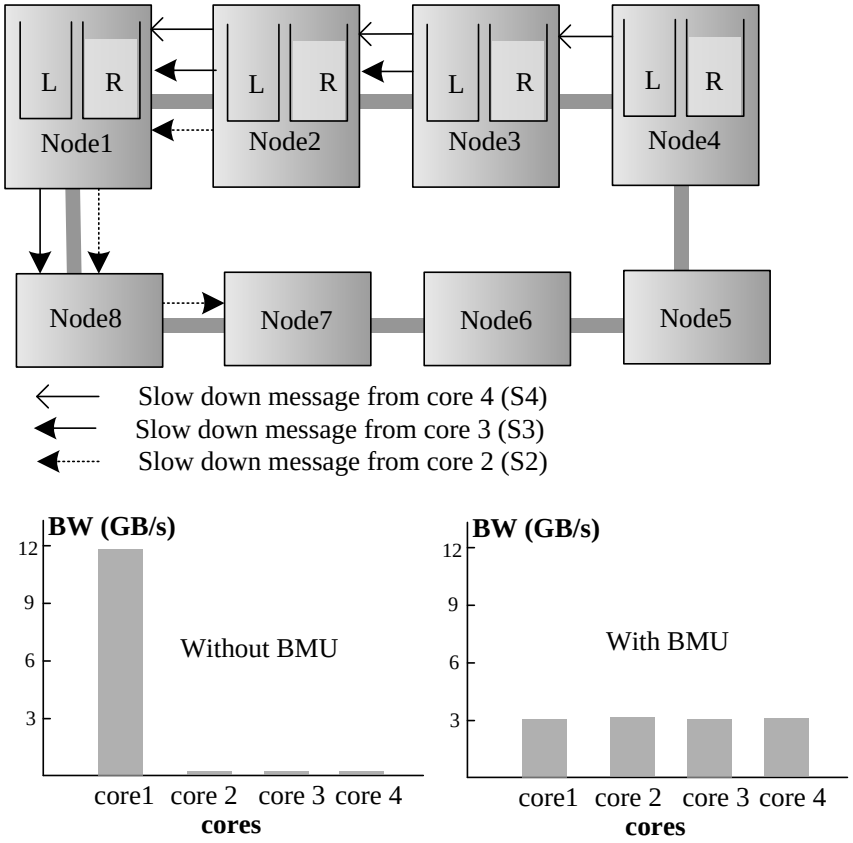


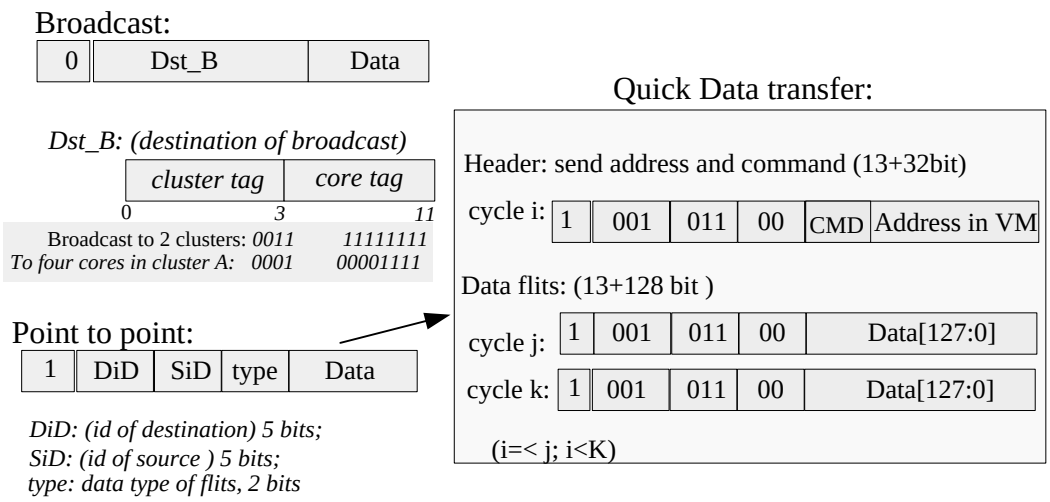
Figure 4-3 Bandwidth management in CIB

4.2.3 Data Routing and Transmission in CIB

In NoC, there aren't dedicate core to core wires, data links are shared by all cores. NoC applies channels for each packet, and data packets are always divided into small flits. There are always extra some flits for allocating channels and routing. Data flits can't route independently, and they must follow the header one by one. Though virtual channel can resolve the dead lock problem of channel based communication, hardware cost is very huge.

CIB is similar with NoC, routing header and routing algorithm are needed. There are two kinds of requirement in CIB: point to point transfer and broadcasting. Each flit can route independently in CIB. It means that each flit should have some extra bits for routing (In NoC, each flit needs some bits for controlling channels). We define two kinds of routing headers for broadcasting and point to point transfers as Figure 4-4.

In MCI, broadcasting is a basic operation, thus the proposed network directly supports to broadcast a packet to all cores or some cores as in Figure 4-4 (a). 14 bits are used for denoting target cores. Cluster tag denotes the target clusters. In 32 cores system, it has 4 bits. Each cluster is corresponding to one bit. For example, "0001" means cluster A is the target, and "1111" means this message is broadcasted to all clusters. Within cluster, eight bits is used for denoting eight cores, which is similar with cluster tag. The target nodes modify "Dst_B" (set the corresponding bit to "0"), and then send it out to the next node until that "Dst_B" equals to zero.



(**a**) Data format of CIB for 32 cores processor (**b**) Data transfer with CIB (for core 3 to core 1)

Figure 4-4 Data format and quick data transfer with CIB.

For point to point data transfer, such kind of target tags is redundant. We have to define a new format for point to point data transfer. As the two kinds of header route in

the same bus, the bits of them should be the same. In Vangal, S.R's 80-Tile processor, the designed NoC applies 6 bits for controlling NoC, and 30 bits for routing. This kind of routing header costs too much. In [43], destination's address is presented by its position in NoC. For an 8x8 mesh network, destination needs 3 bits for X and 3 bits for Y coordinate [57]. This method can reduce some overhead for routing and very suitable for CIB. Thus we define data format for point to point data flits as Figure 4-4 (a).

The header part of a packet includes identification for destination ("DiD"), identification for source ("SiD") and data type flag. DiD is used for routing in CIB. SiD is used for identifying which core sends out this data. CIB supports four kinds of short message: 16 bits synchronization message, 32 bits message for DMA including VM memory's address (16 bits), 32 bits coherence message and user defined command for hardware accelerators. For 128 bits bus, CIB also supports DMA data flits and hardware accelerator's data flits. Type flag is used for denoting the message types. Different units deal with different type of flits.

A completely DMA transfer needs two flits at least: one 32 bits header flits and one 128 bits data flits. Header flits and data flits use different paths and can transfer in parallel. Figure 4-4 (b) shows a small data transfer (two data flits) with CIB. A header including 4 bits command ("CMD") and address (28 bits) are sent out firstly. And then 128 bits data packages can be sent out by 128 data links. Within CIB, 32 bits and 128 bits hops can work at the same time, thus the first 128 bits data package can be sent out with the 32 bits package at the same cycle ("i=<j"). This supports sending out a 128 bits vector within one cycle, which means the register file level data exchange instruction can be supported.

CIB works in a time-division multiplexing method, and there is no dead lock as NoC. Comparing with NoC, CIB can save a lot of hardware resource by reducing virtu-

al channel and large crossbar for all input ports. CIB also doesn't have central arbitrary unit as EIB, which can simplify the transmission process.

4.3 2x2 Mesh Network for Four Clusters

Comparing with NoC, CIB's delay is very short (1 cycle per node). However, the scalability of CIB is very poor. If CIB is extended to 64 cores case, the maxim distance between cores is 32 hops. Both BW and delay becomes bottleneck. Thus, we apply mesh network for connecting 8-core CIB. Then the hops can be reduced from 32 to 12. In this processor, 2x2 mesh NoC is used for connecting four clusters. Figure 4-5 shows the whole network and routers in mesh NoC.

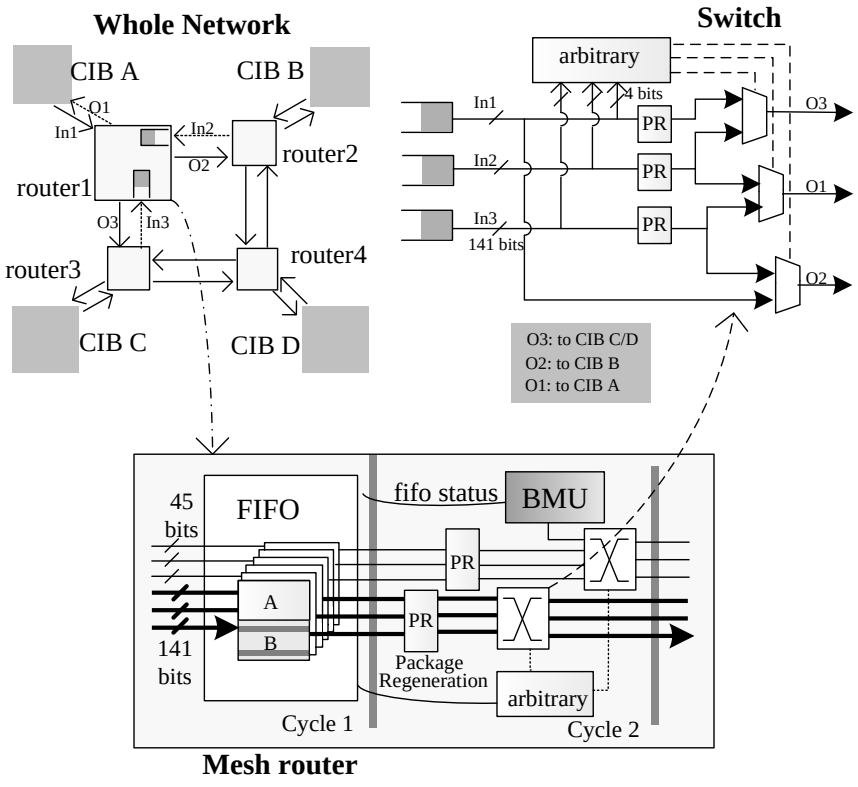


Figure 4-5 The three ports router for connecting CIBs

Each router has 3 data ports and 3 message ports. All input packages are stored in FIFO and then automatically send to other nodes according to package header. Package

regeneration (PR) unit is used to modify the cluster tag when the data is arrived. The router uses fixed routing method. For example, only output port O3 is used for transmission to CIB D. Then, the multiplexer units just need to select one from two input port as Figure 4-5. For example, two input ports (“In3/In2”) of router 1 may needs to use output ports “O1” at the same time. On the other hand, input port “In3” may just be used by CIB C and CIB D. Then CIB D and C may complete for port “In3”. NoC applies virtual channel for resolving this problem. But virtual channel needs 6 bit for controlling and increases a lot of complexity. In this design, the input FIFO is divided to two parts “A” and “B”, and they are automatically allocated to two competitors of the input ports. Arbitrary unit is used for resolving resource conflicts. For example, two input ports (“In3/In2”) of router 1 may needs to use output ports “O1” at the same time. Then arbitrary unit randomly select one of them. If CIB A is always busy, then the two input FIFOs may become full. BMU unit is used for broadcasting slow down message to CIB according to the FIFO’s status.

In this communication network, every data package has destination and source tags. Every competitor for the ports can have a buffer; data packets don’t have any control bits for virtual channel. Two level pipelines are enough for our design’s time requirement.

4.4 Performance Evaluation and Comparisons

Bandwidth and latency are the most important guideline for a communication network. They are measured in this section for different data sizes. We also measure them of a single ring based CIB. Results show that two rings CIB can satisfy the requirement of 128 bits vector cores. In the applications, small size data transmissions are very common. We also measure the performance of CIB, 2x4 NoC as [43] and CELL’s EIB

[33] for small size data transmissions. CIB is optimized for quick data transmission. Transmission process is simple in CIB, and the delay of each node is only one cycle. Then CIB can achieve about 10 GB/s for 128 bits vectors transfer, which is about 83% of its maximum BW. The 2x4 NoC just can achieve 4 GB/s for small size transmissions, which is only 25% of the maximum BW. EIB also just achieve less than 2 GB/s for 16 bytes DMA, which is about 8% of the maximum BW.

4.4.1 Performance Evaluation of CIB

With a clock speed of 750MHz, One Bus Node can support a peak bandwidth of 24 GB/s for inter-core data transfers. However, one data transfer from core 1 to core 4 needs to be propagated three times in CIB. It means that the effective BW for each core depends on the transfer distance. In experiments, we use random traffic pattern to measure the actual BW in a CIB. The destination is also randomly selected. Moreover, we assume that any packet injected in the network is independent, and the arrived packets are consumed immediately. The average packet latency is measured under different packet injection rate (16-byte data packets per cycle). Table 4-1 shows that latency increase rapidly, when injection rate is larger than 70%. When injection rate is less than 50%, latency is less than 8 cycles. It means that each core has 6 GB/s low latency inter-core communication BW in average (192 GB/s for whole chip). This is enough for most of applications. If we just use one 16-byte data ring in CIB, the latency becomes very large when injection rate is larger than 20%. The maximum BW decreases more than 3 times for single ring case, as the average core distance is also increased than dual rings case.

Table 4-1 BW and latency (cycles) for single/dual rings CIB

Injection Rate*	1%	30%	40%	50%	70%	75%	80%	85%
Dual Rings	3.2	5.1	6.5	7.5	18	40	140	490
Dual Rings+BMU	3.2	4.8	5.7	6.8	14	27	78	218
Injection Rate*	1%	8%	12%	16%	18%	20%	23%	25%
Single Ring	4.8	5.35	6.4	8.7	13	27	121	448
SingleRing+BMU	4.8	5.32	6	7.8	11	21	87	210

*: BW= Injection Rate x 16x0.75 GB/s. (45 bits short messages are excluded for this injection rate, as it can sends in parallel)

4.4.2 Performance Comparisons for Short Transmissions

Previous work about parallel HD encoding on CELL processor shows that small size data transfers are very common in a video encoder, which occupies about 13% of whole process time. However, both NoC and CELL's EIB can't provide enough BW for small size data transfers. When data size is less than 16 bytes, EIB just can achieve less than 8% of maximal BW (when message size larger than 1024 bytes) [33]. In this section, we measure the BW and latency performance of NoC and CIB for short transmissions.

Noxim NoC simulator is used for simulating a 2x4 NoC. The frequency of NoC, latency of router and data format is configured as [43]. As the packets size of Noxim is limited to 10 flits per packet, data size per packet is measured from 1 to 8 words in this experiment. XY routing algorithm and random spatial distribution of traffic is applied. Figure 4-6 shows the maximal BW comparisons at different data size. The 2x4 NoC operates at 4 GHz as in [43], and the width of bus is 38 bits. Without considering the

channel and buffer conflict, it can achieve about 16 GB/s (4 GHz*4 bytes) for large packets. However, it achieves less than 4 GB/s in these experiments.

EIB operates at 1.6 GHz with 128 bits ring bus. It can achieve 25.6 GB/s for local memory access, when data size is larger than 1024 bytes. However, it just can provide less than 2 GB/s for a 128 bits vector.

CIB operates at 750 MHz with 128 bits bus. Without considering the resource conflict, it can achieve a maximal BW as 12 GB/s (0.75 GHz x 16 bytes). In experiments, we measure the BW of different data size from 1 to 8 words at a random spatial distribution of traffic. It can achieve more than 10 GB/s for vector data transfer. The BW also decreases when the data size is less than bus's width. For small vector data transfers, CIB can achieve more than 2.5 times better performance than EIB and NoC.

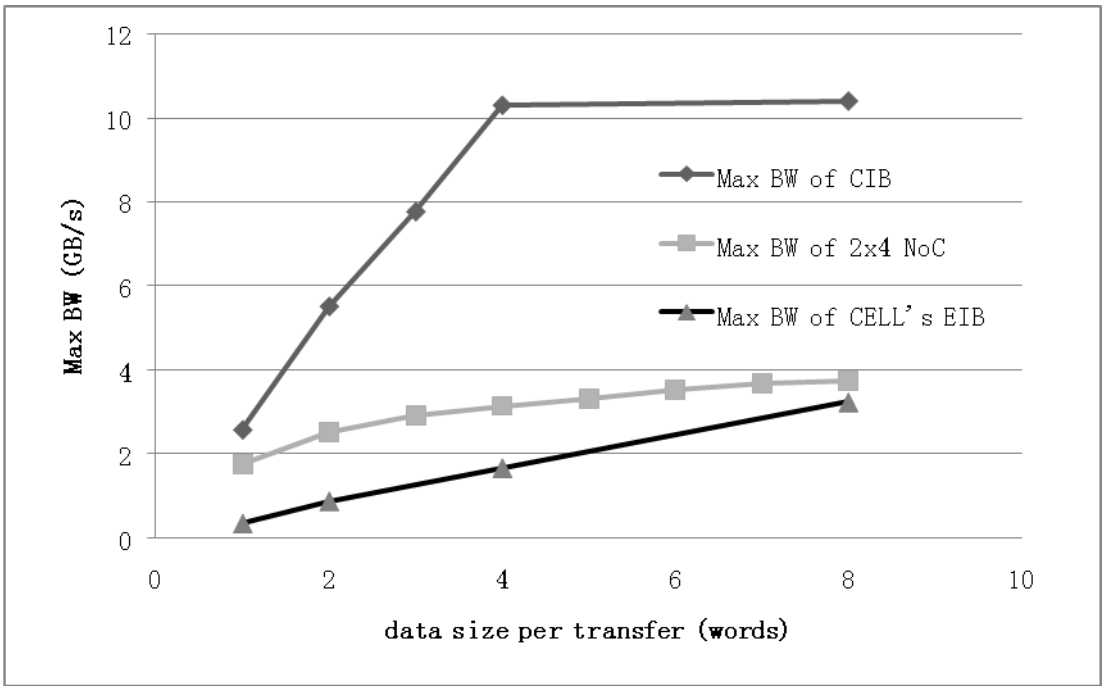


Figure 4-6 Maximal BW performance of 2x4 NoC, CIB and EIB

We also measure latency of NoC and CIB at different flits injection rate. Figure 4-7 shows the results. When the injection rate is less than 8%, the latency of CIB is less than 4 cycles in average. For NoC, the latency is more than 8 cycles in average. Thus CIB

can achieve more than 2 times better performance in latency, when the injection rate is less than 8%. This is very important for instruction level data exchange and sharing hardware accelerator. As EIB is based on conventional shared bus architecture, there is no injection rate concept in EIB. We can't compare it with NoC and CIB. But one transmission in EIB needs 50 cycles at least [33]. Thus the proposed network can achieve lower latency than NoC and EIB.

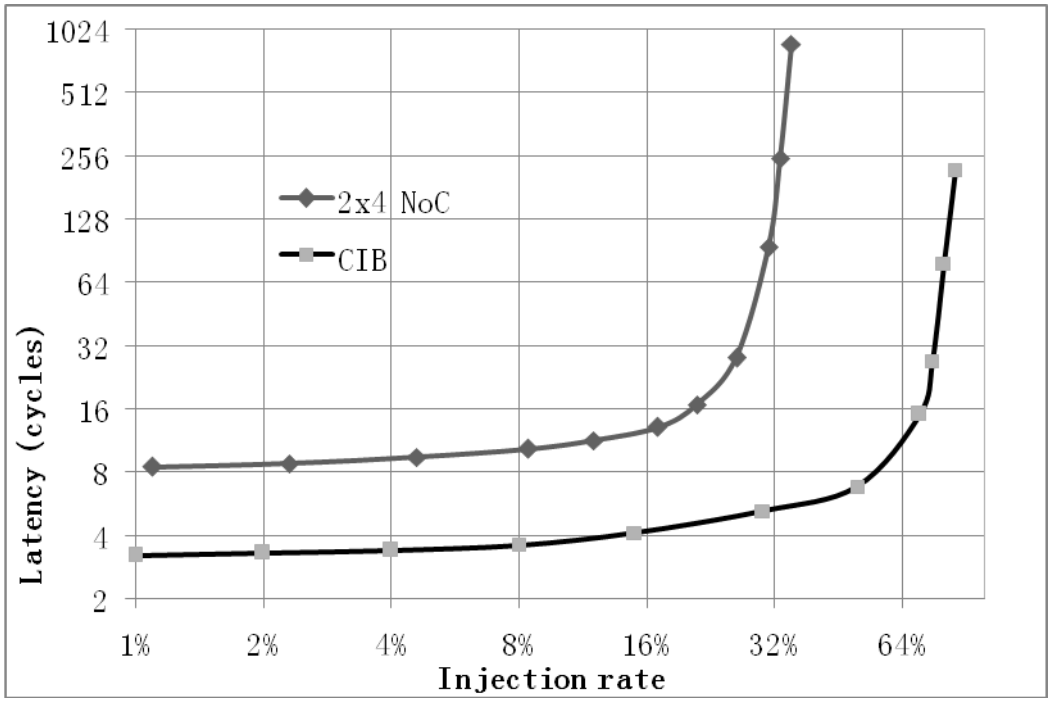


Figure 4-7 Latency of CIB and 2x4 NoC at different injection rate

On the other hand, hardware costs of CIB based network is less than other NoC based network. Table 4-2 shows the costs and architecture detail of them. As there are no details about the hardware costs of EIB, we can't compare with it. Comparing with [43], one bus node in CIB can save about 61% of transistors. Router in the 2x2 mesh network also can save about 30% of transistors. The latency and transmission overhead are lower in our work. And the data format is also defined according to NoC's bus width. In [43], routing information needs 30 bits, and the destination memory address needs about 32 bits. Then one data transmission needs three packages at least: one for

routing information (FLIT0), one for memory address (FILT1) and one for data. It means that BW for small data transmissions is reduced by 75%. In CIB, 128 bit vector core needs to send 128 bit data, and then the width of bus in CIB is 141 bits (13 bits header+128 bits data). The 45 bits bus can be used to send the destination memory address. Thus the BW for small data transmissions is the same as huge data transmissions.

Table 4-2 Hardware costs and architecture comparisons

Networks	NoC [43]	CIB's Bus Node
Input FIFOs	5 ports x 2 FIFO (2 virtual channels)	1 FIFO
Crossbar	Five 5 to 1 multiplexer	Two 2 to 1 multiplexer
Pipelines (=latency)	5 (cycles)	1 (cycles)
Transistors	210K	81 K
Overhead per flit*	6 bits for control	13 bits for routing

*: overhead for data flits. NoC needs two flits for allocating a channel, and the data packet also needs 6 bits for routing. The proposed network needs 13 bits for routing and a 45 bits header for one packet.

In CELL's EIB bus, there are also similar problem for small DMA transmission. A DMA transmission consists of DMA issue, DMA to EIB, List element fetch and Coherence protocol steps, which costs more than 50 cycles. Our previous works about parallel encoding on CELL show that EIB is poor for small data transmissions and DMA costs about 25% of the whole encoding time [55]. In CIB, one 45 bits packages with one 141 bits data package can be sent out at the same cycle, and routing in CIB independently. DMA and MP pipeline in vector core can issue a transmission directly to other core. There aren't steps such as Coherence protocol or List element fetch steps. Vector core can exchange data in instruction level.

4.5 Chapter Summary

In this chapter, a time-division multiplexing based communication network is introduced. It applies ring bus based topology and low latency bus nodes. Two kinds of data format are designed for supporting core to core communication and broadcasting. With a 13 bits header, data flits can route automatically and independently in CIB. Then the data paths can be used by different cores in a time-division multiplexing way. Bus nodes accept new data flit at free bus slot, and then the flits in CIB wouldn't be stopped or blocked. Input flits can be sent forward immediately. BMU is applied for allocating the BW for each core.

Comparing with conventional NoC [43], CIB can save a lot of hardware resource by reducing virtual channel and large crossbar for all input ports. One bus node in CIB can save about 61% of transistors than a 5 ports router in [43]. As the delay of CIB's bus nodes is only one cycle, CIB's latency is more than two times better than a 2x4 NoC or EIB. CIB can achieve more than 10 GB/s BW for all 128 bits vector transfers. Comparing with EIB, CIB just uses half wire resource and achieve more than 2.5 times higher BW for small size data transfers (<32 bytes).

In this communication network, every data package has destination and source tags. Every competitor for the ports can have a buffer; data packets don't have any control bits for virtual channel. Two level pipelines are enough for our design's time requirement. Broadcasting is supported in CIB, which is very importing for snooping based cache coherence protocols. Channel based NoC designs, can't support this.

5 Chip Implementation and Performance Evaluation

After functional check with Modelsim, the chip was synthesized by Synopsys Design Compiler in bottom-up compile strategy. The final pos-layout simulation also performed with Modelsim to check the timing requirements. The design is fabricated in SMIC 65 nm CMOS technology. This chip can achieve 750 MHz at 1.2 V core power. Section 5.1 presents the design flow and measured results. In Section 5.2, application level performance is presented and compares with some common DSP processors and a massive-parallel SIMD processor.

5.1 Chip Implementation and Measurement

The whole design flow mainly contains four steps: RTL level design and function verification, synthesis and formal verification, back-end design including floorplan and routing, and post-layout simulations. After these steps, the designed are fabricated in factory. The first step performs the functional verification with Modelsim SE. Then the finished modules are synthesized by Synopsys Design Compiler with SIMC 65 nm. If the modules can satisfy the timing constrains, then they are finished in these steps. Otherwise, we have to modify the RTL to reduce the delay. When all modules satisfy these time constrains, a bottom-up compile strategy is applied for compiling the whole chip. We also need to do the formal verification by using the Standard Delay Format file again for the whole chip. And then the netlist file and Synopsys Design Constrain files are fed into back-end design steps. Synopsys IC Compiler is used for floorplan, place-

ment and routing. After all Design Rule Check (DRC) errors are fixed, the post-layout simulation is performed for checking the function and timing again.

To verify the fabricated chips, a test unit is added. We also design a testboard as Figure 5-1. A FPGA is used for testing in the testboard. A parallel test port through which a host can access the processor's memory space and registers is designed for testing. FPGA can control the processor by this test port. PLL units are configured by a dedicated I^2C serial port. Four test modes are supported: repeating single instruction, repeating a short code, accessing register files and memory. The first mode is used for measuring power of each instruction. The second mode is used for the average power measuring power of some small applications. Accessing register files and memory can be used for verification the function.

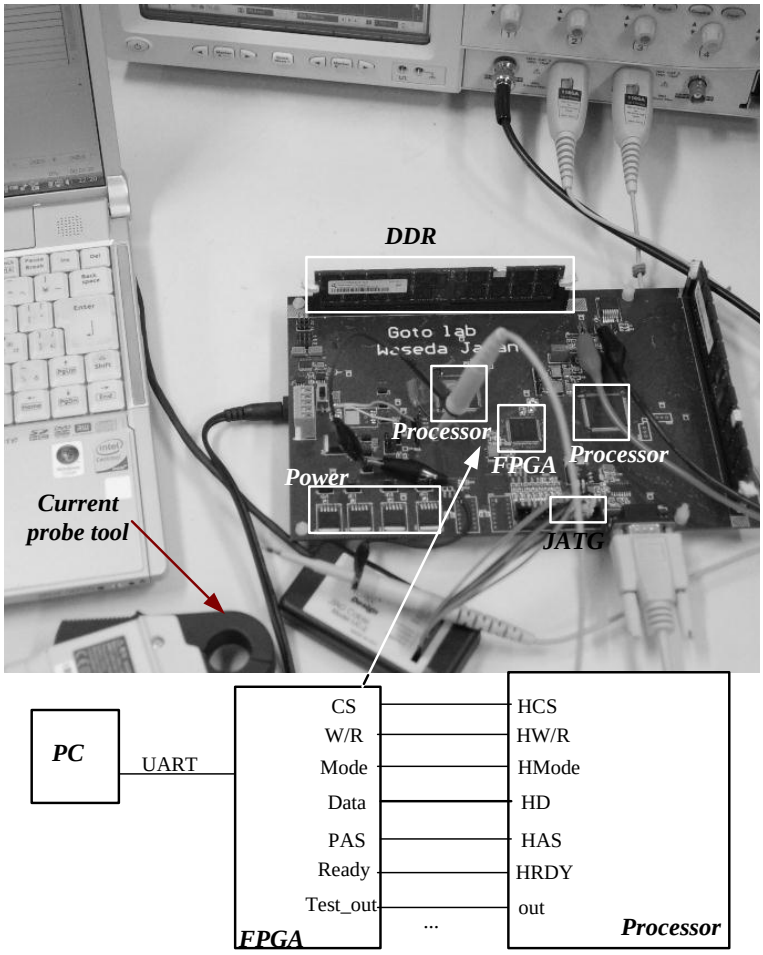


Figure 5-1 Test systems and parallel interface between FPGA and processor

Figure 5-2 shows the die micrograph of the 32-core dual-issue processor. The die occupies 25 mm2, including 32 vector cores, 256 KB L2 cache, 64-bit DDRII's PHY and two PLL units. Each cluster with CIB and a mesh router occupies 4.4 mm2. Extensive clock gating is applied at cluster level, core level, and register level to reduce the power consumption. The power consumption at different supply voltage is measured by the evaluation board.

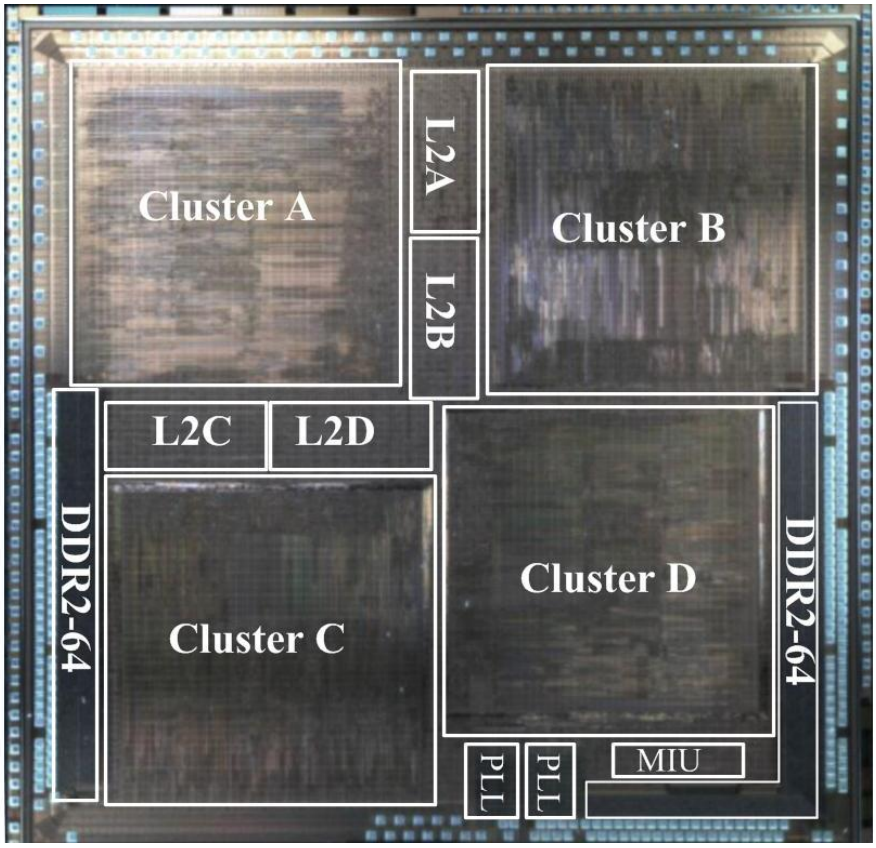


Figure 5-2 Photograph of SVP in 65nm CMOS (5x5 mm^2)

Figure 5-3 shows measured power and frequency from 0.8 to 1.2V. The power consumption is measured with 100% utilization rate of 32 cores and about 50% utilization rate of L2 caches at 25℃. The maximum speed of 750 MHz is achieved at 1.2 V with 3.7 W. DDRII PHY is also verified at 533MHz, 800MHz and 1066MHz. The power consumption of different instructions is also measured by forcing all cores to repeat

the same instruction. The chip can also achieve better power efficiency at lower power supply. It's not stable at 0.8 Voltage. Then the recommended power condition for better power efficiency is 0.9 Voltage. This chip consumes about 0.5 W while operating at 220 MHz and 0.9 V. There are two PLL units for cores and DDRII PHY unit respectively. To save power consumption, clusters can work at lower frequency. DDRII can also work at lower frequency to save power by configuring the other PLL.

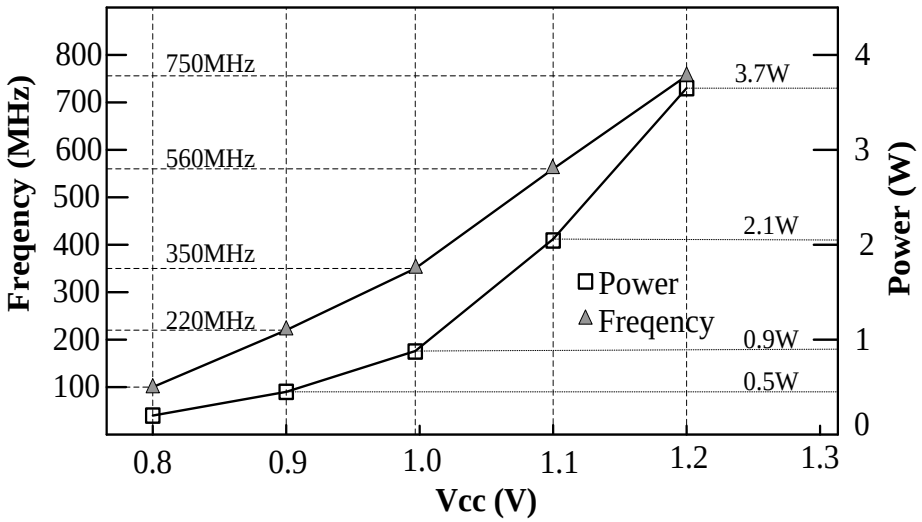


Figure 5-3 Measured power consumption and maxim frequency

Table 5-1 also shows average power consumption (per core) of several instructions, including L2 cache's power (about 50% utilization rate of L2 caches). They are measured by repeating single instruction for each core. 16-bit SIMD MAC operations cost more power than the others. The RISC pipeline is idle for measuring SIMD instructions, except the instruction fetch and decoding stages.

Table 5-1 Power consumption of instructions

Instructions	16-bit MAC SIMD	8-bit MAC SIMD	16-bit ADD SIMD	Vector Load	Vector Store	RISC ALU	NOP
Power(mW)	149	131	105	116	129	51	29

5.2 Performance Evaluation and Comparisons

The performance of MAC instruction is widely used in marketing literature as an indicator of processor's performance [48], which is also widely offered. Then we measured the GMAC performance of this chip, considering the overhead of setting up data for processing. The whole chip can achieve a peak performance of 375 GMACs, or 750 GOPS of 8-bit data operations. GMACs/W is treated as a general indicator of power efficiency. The power of GMACs is measured with a loop of several 8-bit MAC instructions (Utilization rate of L2: 0%), and then this large-scale SMP processor can achieve 98 GMACs/W at 1.2V. Table 5-2 shows performance and power efficiency comparison in GMACs/W. The chip is compared with two similar high performance processors, which are also designed for multimedia applications. For eight bits pixel based data processing, it can achieve 1.9 times higher GMACs performance than the 167 many core chip. For 16 bit GMACs/W, this chip can still achieve about 60% lower power consumption than [16].

Table 5-2 Performance comparison in GMACs/W

Comparison	Proposed	JSSC09[16]	ISSCC07[11]
Cores	32 cores	167 cores	16 SIMD
Architecture	SIMD & SMP	RISC(16bit MAC)	SIMD Plane
Frequency	750MHz/1.2V	1.2GHz/1.3V	800MHz/1.0V
Performance	375GMACs	197GMACs*	128GMACs*
Technology	65nm	65nm	130nm
GMACs/W	98	19	12

*: don't support 8-bit MAC instructions

The proposed SIMD core architecture is designed for multimedia applications. Max Baron's report [48] shows that DCT and SAD kernels occupy about 45% of total cycles for MPEG4 codec. SAD becomes dominant in video codec. Thus, we select SAD, DCT and a 3x3 matrix multiply [48] as our benchmark for evaluating cycle level performance. VCP [13] and several common used DSP processors [48] are compared with proposed design. Table 5-3 shows the cycle count comparison. For SAD and 3x3 Matrix Multiply, the proposed work's cycle count is reduced by 37% than TI C6415 DSP. As instruction parallelism of DCT 8x8 is very high, VLIW based DSP can achieve better performance than our design. VCP has three RISC and SIMD pipelines which is three times of our design, thus it's excellent in both data and instruction parallelism. As most of applications don't have enough parallel SIMD instructions and sequential parts become the bottleneck. Thus, it just can achieve about 1.5 times better performance than the proposed work in this experiment.

Table 5-3 Cycle count comparison of three applications from [13]

Kernels /Primitive	VCP[13] (3-SIMD)	TI C6415 (8-VLIW)	TI C5502 (2 MAC)	BF533 (2 MAC)	Proposed (1 SIMD)
DCT 8x8	113	116	1078	293	171
3x3 Matrix Multiply	14	38	57	143	24
SAD 8x8	14	31	N/A	190	15

Details of performance and power of an edge detection application (frame: 720x480) are presented in [13]. We also measure time and power of this application and compare with VCP in Table 5-4. Application level energy efficiency is defined as the energy cost for processing one frame in uJ. Table 5-4 shows that VCP (3 RISC + 3 SIMD) is about 4.4 times larger than our design. Our design uses smaller architecture and higher frequency for achieve higher application level performance. Time cost per

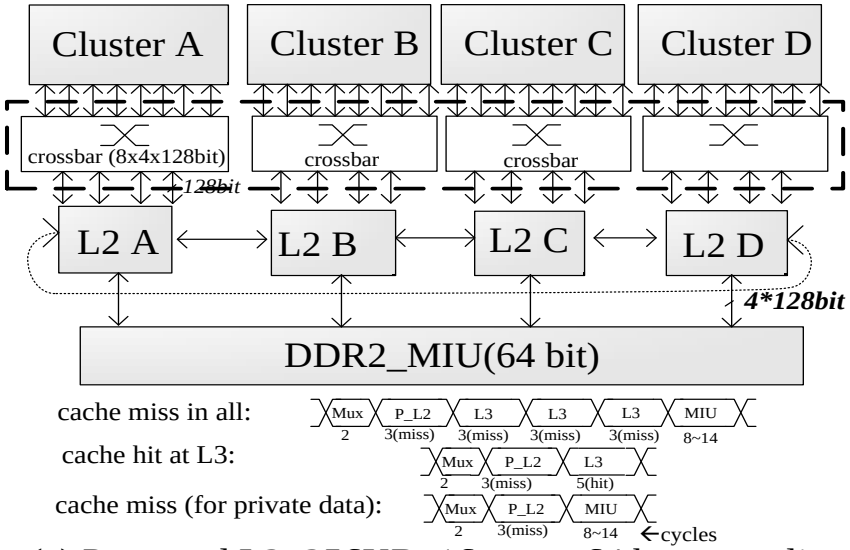
frame is about 1.4 times better than VCP. Then proposed work can achieve about 2.1 times better energy efficiency than VCP.

Table 5-4 Application level performance and power comparison

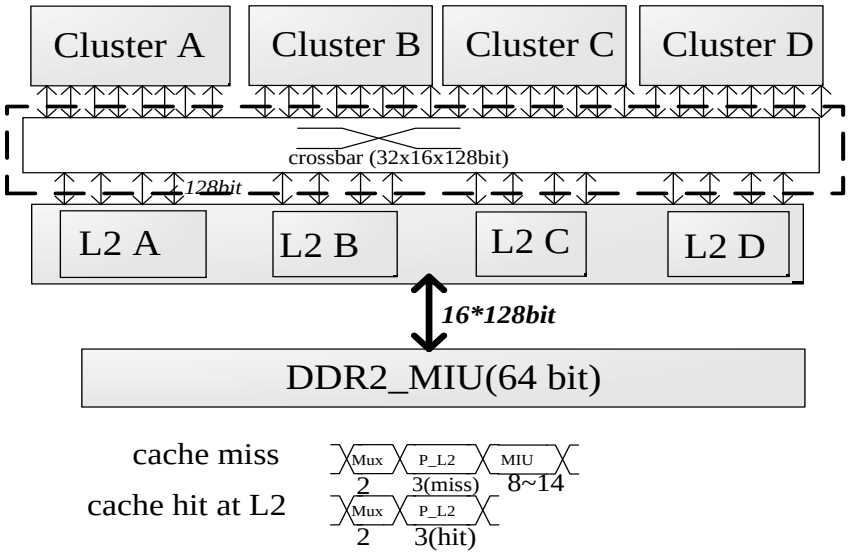
Edge Detection	Time/frame	Power	Energy	Frequency	Gates
VCP[13]	3.52ms	173mW	610 uJ	300MHz	1268K
Proposed	2.55ms	112mW	285 uJ	750MHz	286K

To evaluate the performance of the proposed L2 cache, we compare it with a unified L2 cache as [36]. To guarantee fairness, they are designed with the same number of ports, bank architecture and capacity for the same L2 BW. The same bank architecture as Figure 5-4 is also used in this experiment. The architectures of them are shown as Figure 5-4. Unified L2 cache is organized as 256 bytes per line, which may get better performance for some applications.

Hardware cost, average memory access delay (in cycles) and miss rate (MR) are evaluated. Hardware cost only includes the arbitrary and multiplexer units. Overhead of cache coherence for unified L2 cache design is not considered in this experiment. The same applications as Section 3 are used. Applications work in parallel within 32 cores. Table 5-5 shows that unified L2 cache uses a lot of hardware resource for the crossbar and arbitrary. Cache miss rate of proposed work increases about 0.12%, comparing with unified L2. In proposed work, cache miss needs about 3~9 extra cycles. In all, the proposed work sacrifices 0.08 cycles memory access time for reducing 63% hardware cost in average. As DMA can hide the memory access delay for vector data, this cost becomes acceptable.



(a) Proposed L2: 256KB, 16 ports, 64 bytes per line



(b) Unified L2: 256KB, 16 ports, 256 bytes per line

Figure 5-4 Architecture of the proposed L2 cache design and unified L2

Table 5-5 Performance comparisons of two L2 cache designs

Applications	Unified [8] Gate count: 233.6K		Proposed Gate count: 87.1K	
	MR	delay	MR	delay
Radix	1.84%	2.35 cycles	1.92%	2.44 cycles
FFT	1.31%	2.21 cycles	1.47%	2.30 cycles
Raytrace	0.41%	2.12 cycles	0.53%	2.17 cycles

5.3 Chapter Summary

A large-scale SMP computational platform that is well-suited for video and image processing has been fabricated and verified in 65 nm CMOS. Details about design flow and testing system are also introduced. This processor can achieve a maximum speed of 750 MHz at 1.2 V core power. The whole chip can achieve a peak performance of 375 GMACs, or 750 GOPS of 8-bit data operations. This vector core provides widely SIMD instructions for multimedia applications. For eight bits pixel based data processing, it can achieve 1.9 times higher GMACs performance than the 167 RISC cores chip. Three typical kernels of multimedia applications are also evaluated. For SAD and Matrix Multiply kernels, the proposed work can achieve more than 1.5 times better performance than the other DSPs. Though it costs more cycles than the massive parallel SIMD processor (VCP), VCP costs about 4.4 times gate counts than our design. To evaluate the energy efficiency, an edge detection application is evaluated. Then proposed work can achieve about 2.1 times better energy efficiency than VCP with less SIMD plane per core.

6 Extended Processor with Hardware Accelerator

A high performance multicore processor has been designed for multimedia applications by maximizing on-chip data level and task level parallelism. However, there are limitations for high parallel system: sequential functions with less parallelism become the bottleneck, and floating point based functions becomes very slow for this integer processor. In previous single core processors, hardware accelerators are added for resolving these problems. As the usage rate of accelerators is very low, it's unnecessary to add the same accelerators for every core of multicore processor. Sharing resources and reducing the cost becomes a hot topic recently. AMD also proposed a new architecture for desktop PC, which can share the floating point units. Section 6.1 presents the background of this area.

Section 6.2 introduces our previous intra decoding engine for Ultra HD video decoder. Section 6.3 shows how to use it as a hardware accelerator in this platform and how to share it within cluster. As the shared intra decoder processes MB without data dependency problem, it can achieve about 4 times better performance with less gate counts and SRAM. It can process one MB within 16 cycles, which means it can satisfy eight channels HD decoding with 32 MHz. And section 6.4 gives the summary of this chapter.

6.1 Background

For mainstream CPUs, power consumption and gate counts determine the performance. Shares hardware resources for multicore processor becomes a hot topic recently.

To deliver higher performance with limited budget, Advanced Micro Devices' Bulldozer architecture [58] combines two independent cores intended to deliver higher throughput with less area and power consumption. Bulldozer shares hardware if it's affordable and profitable. The floating-point unit (FPU) is treated as affordable for sharing, as integer instructions are domain in dominated in most applications. Bulldozer module has two integer pipelines and can execute two threads via a combination of shared and dedicated resources. The FPU is a coprocessor model shared between two integer cores. The FPU unit has its own out-of-order engine along with the execution units and register file. From the software point of view, Bulldozer appears as two fully capable, independent cores.

Bulldozer is instruction level sharing architecture, and the shared FPU must be closely coupled with integer cores. Thus, sharing hardware influences timing and complexity of critical hardware paths. The overhead of communication between cores and shared units becomes the bottleneck. Therefore it's hard to share a FPU unit between four cores.

However, hardware accelerators in embedded systems don't need to be closely coupled with cores. Accelerators always need dozens of cycles to deal one job. FPU is instruction level coprocessor, but hardware accelerators are function level coprocessors. Several cycles delay for communication is accepted. Then it becomes possible for more cores to share hardware accelerators. In this chapter, a new sharing architecture is proposed.

6.2 Introduction of Previous Intra Decoder

A high profile intra decoder in H.264/AVC video coding standard is firstly designed for 4Kx2K@60fps Ultra High Definition (UHD) Decoder [59] [60]. The pro-

posed architecture can provide very stable throughput, which can predict any H.264 intra prediction mode within 66 cycles [61]. Compared with previous design, this feature can guarantee the whole decoding pipeline to work efficiently. The intra prediction engine is divided into two parallel pipelines, one is used for 4x4 block prediction loops and the other is used to prepare data for MB loops. It can overlap data preparing time with prediction time, which can finish data loading and storing within 2 cycles. Comparing with MB pipeline only architecture, it can achieve more than 3.2 times higher throughput with 29.8K gates cost. The proposed architecture is verified to work at 175 MHz for our UHD Decoder by using TSMC 90 nm technology.

6.2.1 Design Requirements of the UHD Decoder

In our UHD decoder, it's capable of 4096x2160@60fps H.264 decoding at 175MHz, which is at least 4.3x better than the state-of-the-art [62] [63]. Comparing with previous decoders, the most important feature is that the external memory bandwidth (BW) requirement is very low. As high frequency DDR memory costs a lot of power, we just used a 64bit 166MHz LPDDR. Comparing with DDR/333MHz, the power consumption can be reduced a lot. Low BW can save a lot of power. For a 4096x2160 H.264 decoding, even if each pixel is read only twice and written once, the basic BW requirement is more than 2GB/s. So the 166MHz LPDDR is completely not enough for UHD. A lot of on-chip memory is also applied to reduce BW requirement. The decoder core contains 662K logic gates and 59.6KB on-chip memory which is implemented by TSMC 90 nm technology. In intra decoding part, 16K bytes on-chip memory is used and all of off-chip BW requirement is reduced.

Another feature is that the main decoder is required to process one MB in around 64 clock cycles. This is useful to achieve a stable throughput. In our implementation, intra decoding time is stable and the pipeline can be fully utilized. Figure 6-1 presents

the decoding diagram of our UHD decoder. Most of pipeline components are designed to process each MB in around 64 clock cycles. There is only one component with variable throughput for Intra decoding: the CAVLC block (ED 0 and ED 1) in Figure 6-1. Its throughput is dependent on the bits of each MB.

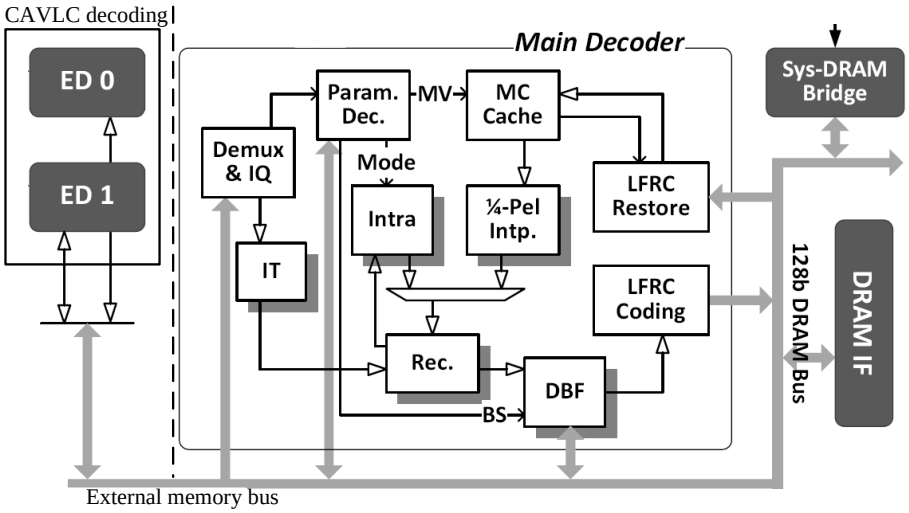


Figure 6-1 Decoding diagram of H.264/AVC UHD decoder [59]

Thus CAVLC's throughput can't match the other units. To solve above problems, a ping-pong buffer was applied between CAVLC and Intra Prediction Engine [61]. It enables the overlapped execution of intra prediction and entropy coding without large cycle increase. But in our constant throughput Intra decoding pipeline, stable throughput requirement is very critical. Small ping-pong buffer is not enough for our decoder. A new architecture is proposed for resoling the mismatch problem for CAVLC. In this architecture, CAVLC decoding is completely asynchronous with Intra decoding. CAVLC decoding component starts to decode at first, and writes the decoded data to off-chip memory. After enough MBs have been buffered in off-chip memory, Intra decoding pipeline will start decode. This asynchronous decoding architecture makes it possible to design a new decoding order for Intra decoding parts.

In previous works [64] [65], each MB is decoded line by line according to H.264 standard order. They can't change the MB decoding order as CAVLC part must decode

according to H.264 standard order1). In this decoding order, MB4 is dependent on MB0/1/2, and MB4 is just decoded behind MB2. Thus, MB pipeline must wait for MB2's output before start to decoding MB4. This data dependence problem causes a lot of pipeline bubbles. It's impossible for them [64] [65] to resolve the data dependent problem by changing MB prediction order. In our proposed architecture, CAVLC part is working asynchronous with Intra prediction. Intra prediction part doesn't need to follow the original decoding order. A new Zig-Zag MB level decoding order is proposed as in Figure 6-2. Four MB lines compose a basic Zig-Zag scanning unit for this decoding order. In each basic unit, MB is decoded in a Zig-Zag scan order. In this new decoding order, MB4 is decoded behind MB3 and MB0, MB1 and MB2 is decoded before MB3. When MB pipeline starts to decode MB4, reference pixels from MB2 are available. Thus, MB pipeline can start to predict new MB when current MB is still under processing in our decoding order.

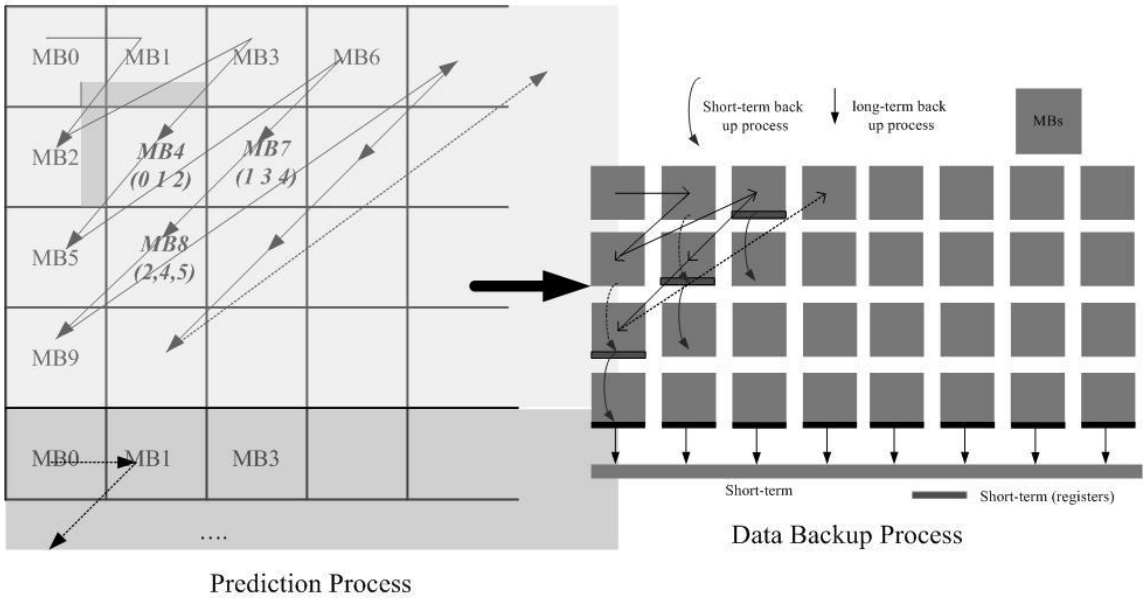


Figure 6-2 Zig-Zag MB level decoding order in UHD decoder

6.2.2 A high Performance Intra Decoder for UHD

In intra prediction decoding process, we need reference pixels from upper and left MB. Most of proposed Intra Prediction architectures [66] [67] apply a small SRAM buffer to store current MB's reconstructed pixels, but don't apply large internal SRAM to the upper MB's reference pixels. In the decoding process, it is used to store reconstructed pixels and then output them to external memory at the end of decoding. In their designs, the reconstructed pixels are buffered in the external DRAM. Considering the writing operation of that reconstructed pixels, about 260 MB/s BW is needed for 4Kx2K@60fps decoder.

In the proposed architecture, a large pixel line buffer is used for luma and chroma intra decoding. It costs about 16K bytes internal memory to save 10% chip's BW. As we have enough area budget but limited memory BW, this approach is very suitable for UHD. The data loading time is also reduced from 101 cycles [68] to 2 cycles. In the proposed MB decoding order, last line pixels of MB1 is stored in a short-term buffer in Figure 6-2, as MB4 needs these pixels soon. Short-term buffer is consisted by several registers. When MB4 is finished, these pixels are useless and the last line of MB4 will replace MB1 in short-term buffer. At last short-term buffers are stored in a SRAM unit, which can save about 260 MB/s BW.

Based on the new decoding order, an Intra decoder is presented as Figure 6-3, which can provide very stable throughput for the UHD decoder. The intra prediction engine is divided into two parallel pipelines, one is used for block prediction loops and the other is used to prepare data for MB loops. The block pipeline is applied for 4x4 blocks prediction and can produce 4 predictors per cycle for all of Intra modes. MB pipeline is applied to make the data preparing and prediction process work in parallel as in Figure 6-3. An internal SRAM is utilized to store depended reconstructed pixels.

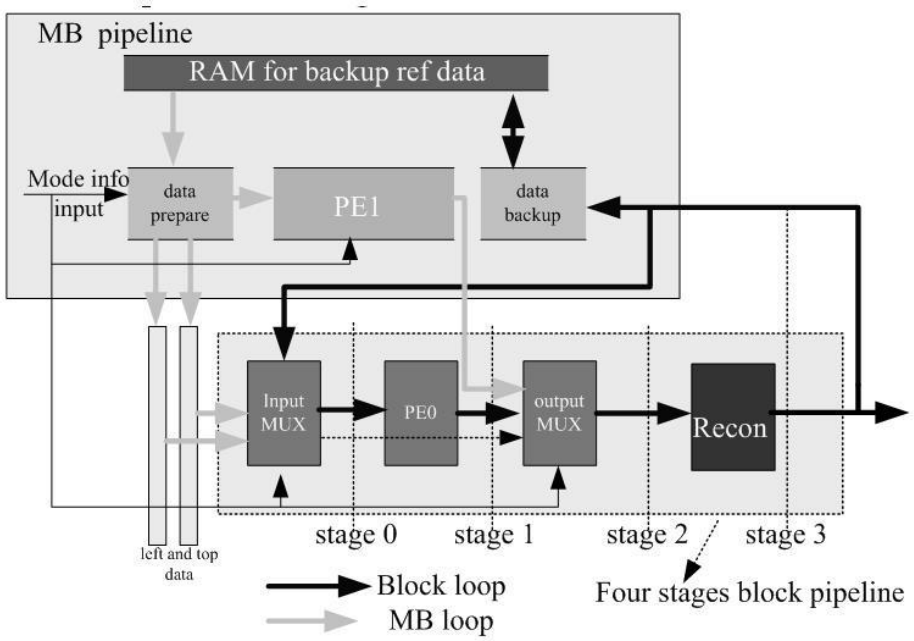


Figure 6-3 MB and block pipelines of Intra Decoder

The upper part in Figure 6-3 is MB pipeline, including data preparing unit, PE1 unit and data writing back unit. Data backup unit in Figure 6-3 is in charge of choosing depended reconstructed pixels from Recon unit's output data and writing back to internal RAM unit. Reconstructed output data is also sent to input MUX unit in block pipeline, as there are serious data dependent problems between blocks. When there is available input mode data, the data preparing unit will start to read related pixels from RAM unit. After finishing preparing work, it will send ready signal to block loop if the input mode is I4x4 mode. If the input mode is I8x8 mode or 16x16 DC/Plane modes, data preparing unit sends ready signal to PE1 after getting the required pixels for PE1. The lower part in Figure 6-3 is block pipeline for I4x4 mode, including input MUX, PE0 and output MUX unit.

In previous designs [66-68], they follow the original MB decoding order and data dependence problem has not been resolved. Data preparing and prediction can't work in parallel, which limits the performance. As we have resolved data dependence problem in Zig-Zag decoding order, data preparing work and H, V value computation in I16x16

plane mode can work parallel with prediction process. As we apply internal memory for reconstructed pixels backup, MB setup and end just need two more cycle in our pipeline architecture. We can reduce the data backup and preparing work from 128 cycles [66] to 2 cycles at the cost a parallel working MB.

And next, we need to design the prediction engines for all intra modes, which must be able to process one MB with 64 cycles. The prediction engine for 4x4 and 8x8 modes are presented in details.

After analyzing all of 4x4 modes [61], we found that only 18 adders are needed to design a combine PE for all 4x4 block prediction. To reduce area cost, we design a combine PE0 as Figure 6-4. PE0 has nine input ports (I0-I8) and 7 output ports. PE0 can compute one 4x4 block prediction in only one cycle. Figure 6-4 shows that the whole 4x4 block prediction pipeline is consisted of three stages: Input MUX, PE0 and output MUX stage. It needs about 13 multiplexers (13 to 1), 4 multiplexers (7 to 1) and 18 adders.

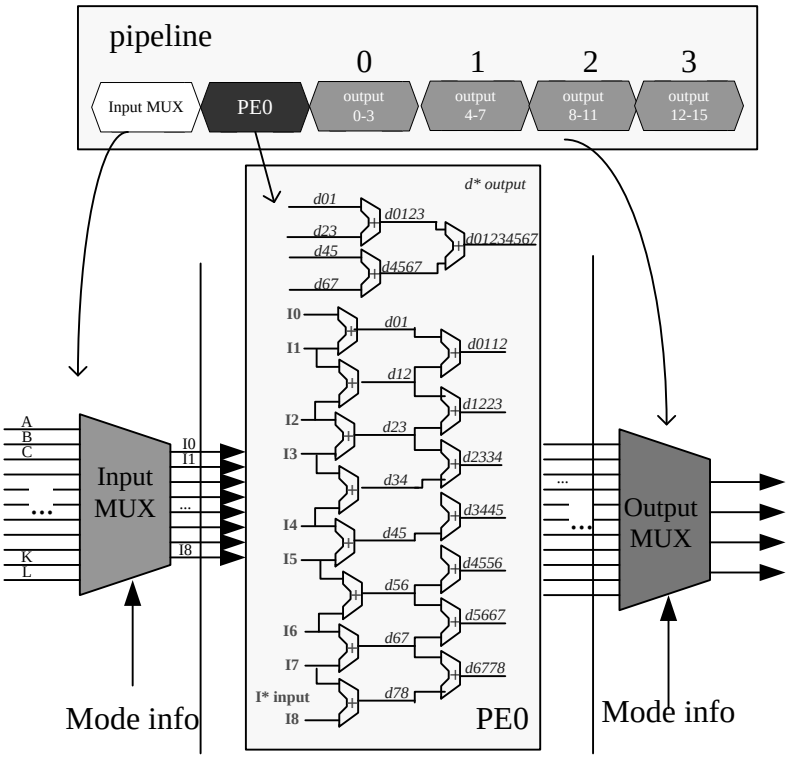


Figure 6-4 Combine PE for Intra Prediction

For 8x8 modes, each 8x8 block need about 25 pixels from adjacent blocks as in Figure 6-5 and these pixels must be filtered before prediction. Figure 6-5 shows the filtering process and the dependent pixels of first 4x4 block in 8x8 blocks. Most of previous architectures [65] [68] [69] just used one pipeline for Intra prediction, and the filtering process can't work parallel with prediction process. The filtering process occupies 7 extra cycles for the four data paths architecture [68] [69] (25 pixels/4 data paths). The whole MB prediction process for I8x8 modes costs more than 90 cycles in their design. As our proposed Intra prediction architecture has two pipelines working together, the filtering process in MB pipeline can work parallel with prediction process in block pipeline. Thus, MB pipeline has 16 cycles (cycles for prediction one 8x8 block) for filtering process. It means the filtering unit just needs to filter 2 pixels (25pixels/16cycles) per cycle for each 8x8 block as in Figure 6-5. This filtering process just needs 5 adders and guarantees the block pipeline to produce one MB within 64 cycles. To combine with 4x4 modes, the output format of 8x8 blocks follows the order as 4x4 modes. And then the 4x4 modes can share use PE0 of 4x4 modes.

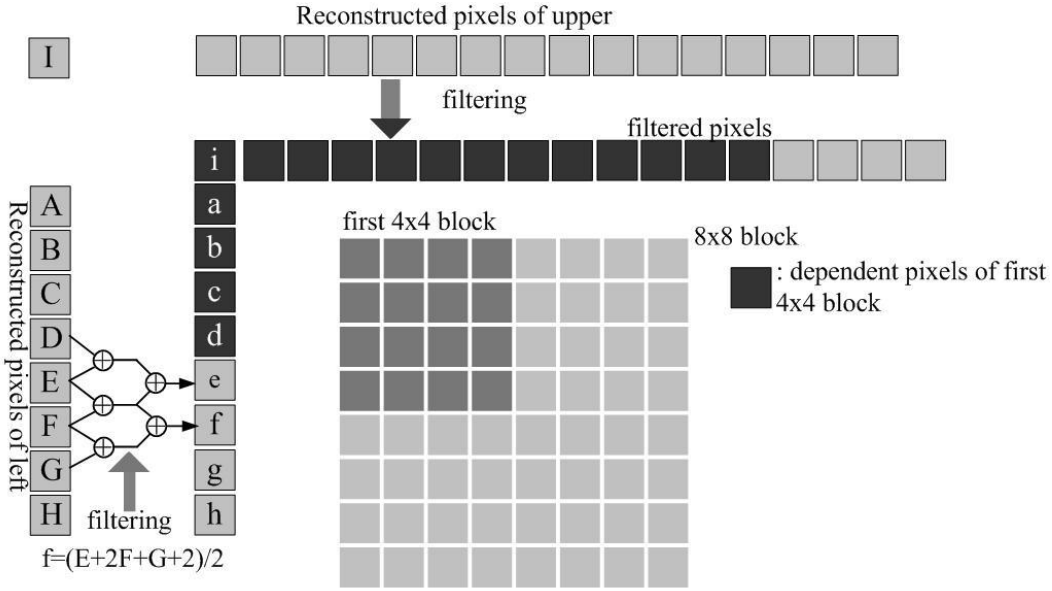


Figure 6-5 8x8 Filtering Process in high profile H.264/AVC

6.3 Shared Intra Decoder within Cluster

Hardware accelerator is a very efficient way for exploring processor's application fields. They work in a flow as: receiving tasks from cores, processing received data and sending back results to cores. Communication with cores is different from the data flow in ASIC designs. For example, hardware accelerator in an ARM based SoC design should connect to the AMBA bus. In this processor, accelerators should connect to CIB bus, and then they can be shared by cores. Data transfers are divided into small flits and automatically routing in CIB. Therefore, accelerators must have an identifier for routing and be compatible the data format of CIB. As several cores may send tasks to accelerator in random order, it requires that accelerators must be able to process tasks independently.

This section presents how to design a shared accelerator in this multicore platform. The benefits by sharing accelerators are also shown.

6.3.1 Interface and Data Flow of the Shared Decoder

Data exchange in CIB is based on data routing. Flits of point to point transfer must have a source identifier and a destination identifier. Data format of CIB determines that it just can have eight users in one CIB bus. Thus the accelerator can only share identifier with a core. Figure 6-6 shows the hardware architecture of the interface with CIB. The output unit in bus node needs one more output port for accelerator ("HA"). As data flits of core and HA may arrive at this node at the same time from different rings, core and HA don't share the output ports. The output unit dispatches the arrived flits to core or HA, according to the type flag in the header. A cross path is built between input ports and output ports. Then the core and HA can communicate directly.

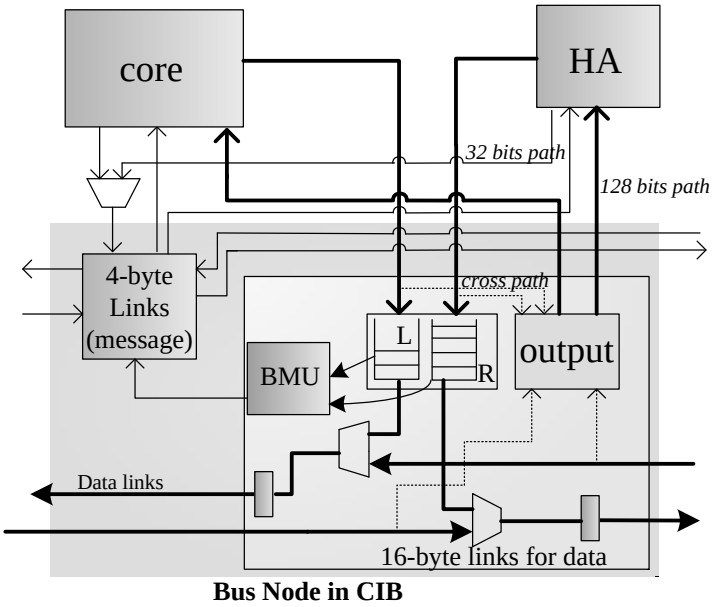


Figure 6-6 Connecting the shared hardware accelerator in CIB

HA doesn't have a SRAM unit as core's VM, and incoming data is stored in a command queue. Thus data communication with HA doesn't need to send out address information. The core unit keeps the destination address. Both the 32 bit message paths and 128 bits data paths can be used for sending data to HA. The dual-issue vector core supports sending out one 32 bits register and one 128 bits register to CIB within one instruction. In the Intra decoder, 16x16 modes need to send 33 pixels and one mode information to the accelerator. 8x8 modes need about 17 or 25 33 pixels. 4x4 modes need 9 or 13 pixels. Thus, it just needs only one or two instruction for sending out a task to the accelerator for sending a task to the accelerator. In this design, cores are in charge of preparing pixels for prediction. The shared Intra decoder doesn't have about the data dependency problem.

6.3.2 New Pipeline Design for Accelerator

In previous design for UHD decoder, a data preparing pipeline and a special output order are applied. However, they are unnecessary for the accelerator. In accelerator, in-

put data is prepared by the cores, and the output order should be suitable for storing in vector memory. According these new requirements of accelerator, some parts of the intra decoder engine is redesigned.

The vector core can send out one 16 bytes vector and one 4 bytes scalar by one instruction. CIB also supports to send out them at the same time. For Intra4x4 prediction, neighboring 13 pixels including 8 upper pixels and 4 left pixels are needed for prediction. However, they aren't used at the same time. For example, I4x4_ Diagonal_Down_Left mode just needs 9 upper pixels, and I4x4_ Diagonal_Down_Right mode just needs 5 upper pixels and 4 left pixels. Considering the mode information, cores just needs to send ten bytes to the accelerator. For Intra8x8 prediction, cores just need to send seventeen bytes to the accelerator. For Intra16x16, 34 bytes are needed, which means two cycles are needed for send out data. On the other hand, accelerator can send back 16 bytes per cycle to cores.

Thus the output pipeline of Intra4x4 as Figure 6-4 is not suitable for this accelerator. Figure 6-7 shows that the output MUX unit is modified for outputting sixteen pixels per cycle. The Input MUX and PE0 unit is the same. There are two extra stages: "Core sends CMD" and "CIB". The delay of the whole process is about 13 cycles for Intra 4x4 modes. Data transmission causes about 10 cycles delay. Therefore, the latency of communication network is very critical for sharing a hardware accelerator. For conventional NoC, the communication delay becomes the bottleneck for sharing.

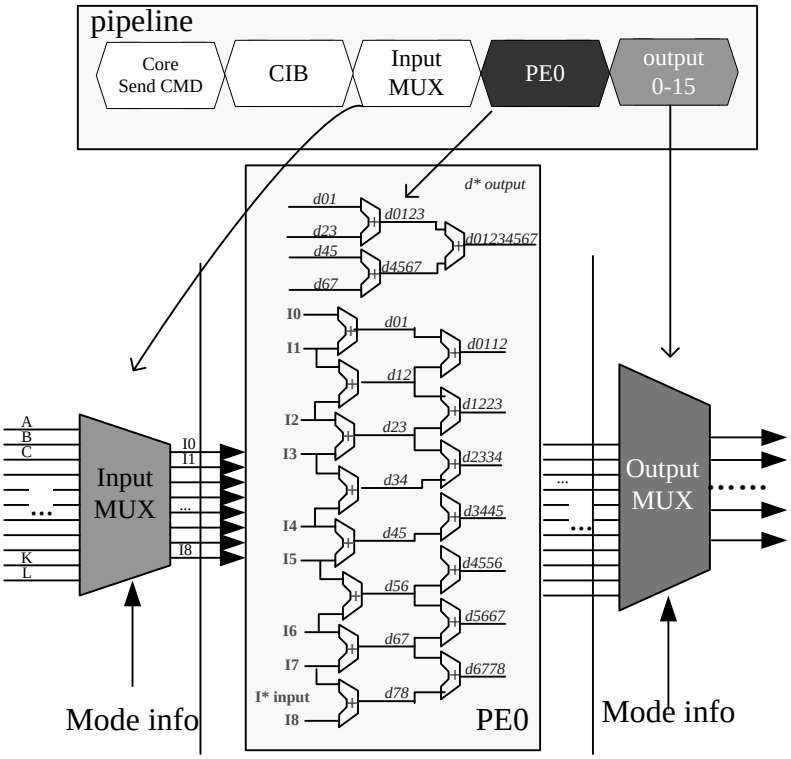


Figure 6-7 The prediction pipeline of intra4x4 modes in accelerator

This accelerator is able to produce 16 pixels per cycle, thus Intra8x8 modes just have four cycles per block. The filter process in previous work needs more than nine cycles for one block, which is not enough for this accelerator. Three more filter units are added for achieving 5 pixels/cycle. Intra8x8 and Intra4x4 modes can share the calculation unit (PE0) as Figure 6-8. Intra 8x8 needs an extra Filter process per block. Figure 6-8 shows that the accelerator can deal with Intra8x8 and Intra4x4 blocks with out any pipeline bubble when different cores are using accelerator. If there is only one core is using the accelerator, the using core needs a lot of cycles for preparing data for the each block, and the accelerator can't work with the maximal performance. Therefore this accelerator can achieve the maximal performance as 16 pixels per cycle, when it's shared by different cores. It can satisfy eight channels parallel HD decoding at 31 MHz. When it synthesized by SIMC 65 nm, it can achieve 300MHz at 1.2V, which is more than ten times of the HD decoding requirements.

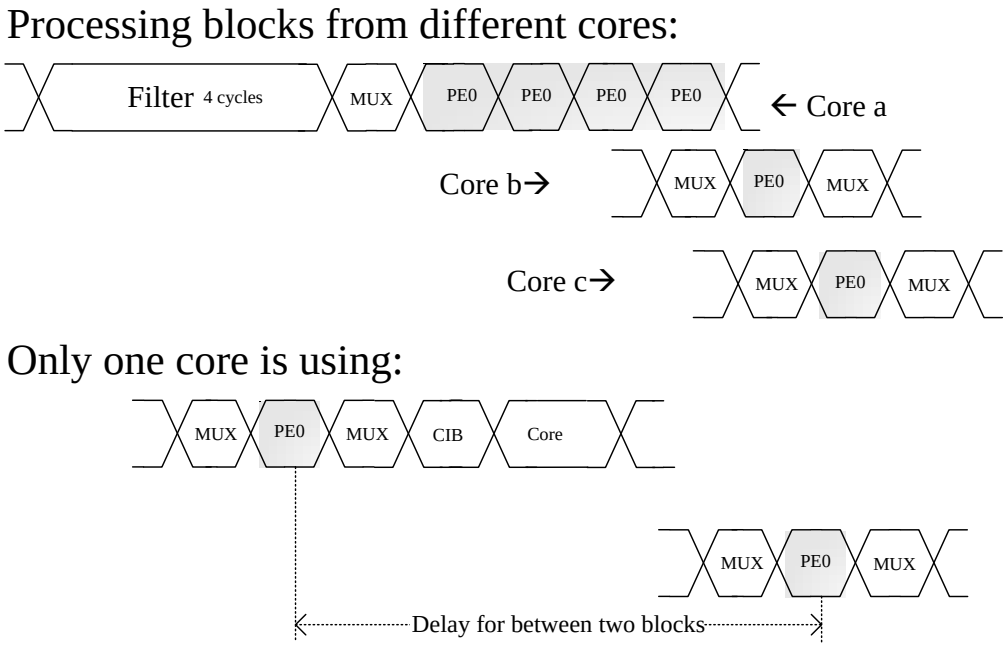


Figure 6-8 The pipelines of accelerator for different cases.

As data preparing work is performed by the cores, accelerator doesn't need any SRAM unit for storing temporary pixels. The output unit also becomes simple. Table 6-1 shows the proposed accelerator uses less hardware resources and achieves more than four times throughput than [61].

Table 6-1 Performance comparisons of Intra decoders

	Cycle/MB	Gates	RAM	Modes
[66]	236	14.9K	384	Base profile
[61]	66	29.8K	8K RAM	High profile
Proposal	16	21.6K	NO	High profile (without data backup)

6.4 Chapter Summary

A shared hardware accelerator is proposed in this chapter. The interface design for sharing accelerator is also presented. Comparing with our previous design for UHD decoder, the pipeline becomes simple and the throughput is more than four time higher. It can satisfy eight channels HD decoding with 31 MHz. The CIB network can guarantee a low latency access cost for sharing accelerators. Together with shared hardware accelerator, this multicore platform can become more powerful for special applications. Comparing with dedicated hardware accelerator, sharing hardware accelerator design can reduce a lot of hardware cost.

7 Conclusion

A large-scale SMP computational platform that is well-suited for video and image processing has been fabricated in 65 nm CMOS. This chip contains 32 dual-issue cores, which supports 128-bit SIMD instructions (including 8-bit MAC) and vertical vector access. Optimization with these features can achieve 2.3 times speedup in average for several 2D image processing kernels. A filter cache is utilized to reduce L1 data cache access and combine write operations. In large-scale SMP systems, traditional snoop protocols cause a lot of snoopy operations, which occupy a huge BW and L1 cache resources. A new application specified cache coherence protocol is proposed to reduce BW costs and L1 cache's energy. Compared with Jetty, MCI can reduce both the tag lookup operations and BW cost. For a 32-core CMP processor, Jetty with MOESI can reduce about 24.6% snooping operations, and our proposed MCI can reduce about 67.8% snooping operations. The drawback of MCI is software compatibility and complexity. The software impacts of MCI are the two configuration steps. MCI is not compatible with conventional software on SMP (memory configurations must be added for MCI). General CPU for PC can't use MCI, as software compatibility is necessary. This is the restriction for our coherence protocol and cache design.

Four CIB networks are used for clusters, and providing 192 GB/s inter-core communication BW in average. Fixed rout mesh network is also applied. This hierarchical network can achieve lower latency and better energy efficiency than NoC. In CIB, every data flits are independent. There are no channels and buffers for in-network data flits. Different transmissions can share links in CIB. The overhead of CIB is that the bus needs about 13 extra wires for routing header. For NoC, 13 routing header is too huge as

NoC needs to store them in buffer. For 32-bit NoC, the hardware increases about 28% for more 13 bits.

Based on those architectures, this chip can achieve good energy efficiency and DLP as MP-SIMD, and also provide high ILP and TLP as CMPs. However, this processor is just suitable for vector processing. To explore the application fields of the proposed SIMD based multicore processor, a shared hardware accelerator is proposed.

For future work, we need to develop a compiler for this processor. For software development, the efficiency of compiler is very critical. For vector based processor, compiler can’t automatically make a loop based function to vector instructions. Programmer needs to spend a lot of time for optimization. Thus, the requirement of compiler for this processor is not very high. We also need to add more interface, such as PCI-E or inter-chip interface. Then different chips can be connected together for achieving higher performance.

Acknowledgement

First and foremost, I would like to gratefully and sincerely thank Professor Satoshi Goto of Waseda University for his constant encouragement, guidance, and support during my research. Professor Goto has wonderful personal fascination and gives me a great guide in my growth and research.

I would like to thank Professor Peilin Liu of Shanghai Jiao Tong University for the continuous encouragement and support throughout my work. I also thank Professor Togawa Nozomu, Professor Shinji Kimura and Professor Takahiro Watanabe for giving me continuous encouragement and precious advices during my research.

I would like to appreciate Dr. Dajiang Zhou and Dr. Jinjia Zhou of Waseda University. They gave me direct guidance and tremendous help during my research. And also the other students in Goto-Lab, they give me a lot of help both in my research and life. Under these people's help, I can enjoy the research life in Waseda University.

I also thank to the support from Waseda University Ambient SoC Global COE Program of MEXT, Japan, and from the CREST project of Japan Science and Technology Agency.

Finally, an honorable mention goes to my family for their understandings and supports.

References

[1] David A. Patterson and John L. Hennessey, *Computer Organization and Design: the Hardware/Software Interface*, 2nd Edition, Morgan Kaufmann Publishers, Inc., San Francisco, California, p.751, 1998

[2] RG Hintz and DP Tate, "Control data STAR-100 processor design," in *Proc. COMPCON 72, IEEE Comput. Soc*. Conf. Proc, pp. 1-4, Sept. 1972

[3] M. Eden, M. Kagan, "The Pentium(R) processor with MMX technology", *IEEE Compcon '97. Proceedings*, pp.260-262, Feb. 1997

[4] Intel Corporation. Intel Developer Web Site. http://developer.intel.com

[5] J. Tyler, J. Lent, A. Mather, Huy Nguyen, "AltiVec: bringing vector technology to the PowerPC processor family", *PCCC*, pp. 437-444, Feb.1999

[6] BTMS320C6414, TMS320C6415, TMS320C6416, Fixed-Point Digital Signal Processors, Texas Instruments Incorporated, 2005.

[7] Y. Lin, H. Lee, M. Who, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, "SODA: A low-power architecture for software radio," in Proc. *Int. Symp. Computer Architecture (ISCA)*, pp. 89‑101, June. 2006

[8] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," *IEICE Trans. Inf. Syst.*, vol. E82-D, no. 2, pp. 389‑397, 1999.

[9] K. Schaffer, R. A. Walker, “A Prototype Multithreaded Associative SIMD Processor,” *IEEE International Parallel and Distributed Processing Symposium*, pp.1-6, Jan. 2007

[10] A. Abbo, et al., “Xetal-II: a 107 GOPS, 600 mW massively parallel processor for video scene analysis”. *IEEE Journal of Solid-State Circuits*, pp.192–201, Feb. 2008

[11] Khailany, B.K, et al., “A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing,” *ISSCC Dig. Tech.* Papers, pp.272-273, Feb. 2007

[12] J. A. Fisher, “Very long instruction word architectures and the ELI-52,” in Proc. 10th Annu. *Int. Symp. Computer Architecture*, pp.140-150, 1983

[13] Wada, T, et al, “A VLIW Vector Media Coprocessor with Cascaded SIMD ALUs," in *IEEE trans on VLSI Systems*, Vol: 17, pp. 1285-1296, 2009

[14] M. Gschwind, H.P. Hofstee, B.K. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, “Synergistic processing in Cell’s multicore architecture,” *IEEE Micro* 26, pp. 10–24, Dec, 2006

[15] Bell, S, et al., “TILE64 - Processor: A 64-Core SoC with Mesh Interconnect,” *ISSCC* Dig. Tech. Papers, pp.88-598, Feb. 2008

[16] Truong, D.N, et al., “A 167-Processor Computational platform in 65 nm CMOS,” *IEEE Journal of Solid-State Circuits*, vol. 44, pp. 310-319, April, 2009

[17] Kahn, Gilles. “The Semantics of a Simple Language for Parallel Programming.” In *IFIP Congress*, pp. 471–475, Aug. 1974

[18] D.C. Cann, J.T. Feo, T.M. DeBoni, "SISAL 1.2: high-performance applicative computing ," *IEEE Symposium on Parallel and Distributed Processing*, pp. 612 – 616, Dec. 1990

[19] Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” In OSDI '04: 6th *Symposium on Operating Systems Design and Implementation*, pp 137–150, Dec. 2004

[20] K.Z. Ibrahim, G.T. Byrd, "Extending OpenMP to support slipstream execution mode" *International Parallel and Distributed Processing Symposium*, pp. 10-14, Jan. 2003

[21] http://openmp.org/wp/2008/10/openmp-tutorial-at-supercomputing-2008 OpenMP Tutorial at Supercomputing 2008

[22] Sato, M, "OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors", *International Symposium on System Synthesis*, pp. 109 – 111, Oct. 2002

[23] O. Hernandez, R.C. Nanjegowda, B. Chapman, "Open Source Software Support for the OpenMP Runtime API for Profiling," *International Conference on Parallel Processing Workshops*, pp. 130 – 137, 2009

[24] C. Saldanha and M. Lipasti. “Power Efficient Cache Coherence,” *Workshop on Memory Performance Issues*, in conjunction with ISCA, pp. 63-78, June 2001

[25] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. “Timestamp Snooping: An Approach for Extending SMPs,” In Proceedings of the *Ninth International Con-*

ference on Architectural Support for Programming Languages and Operating Systems, pp. 25–36, Nov. 2000

[26] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. “Bandwidth Adaptive Snooping.” In Proceedings of the Eighth *IEEE Symposium on High-Performance Computer Architecture*, pp. 251–262, Feb. 2002

[27] A. Moshovos, G. Memik, B. Falsafi and A. Choudhary. “JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers”. Proceedings of the 7th *International Symposium on High-Performance Computer Architecture*, pp.85-96, January, 2001

[28] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. “The Stanford DASH Multiprocessor”. *IEEE Computer*, 25(3), pp:63-79, Mar. 1992

[29] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. “The Alpha 21364 Network Architecture,” In Proceedings of the 9th *Hot Interconnects Symposium*, pp: 35-40, Aug. 2001

[30] W.-D. Weber, S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. “The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers,” In Proceedings of the 24th Annual *International Symposium on Computer Architecture*, pp: 120-124, Jun. 1997

[31] D. Chaiken, D, C. Fields, K. Kurihara, A. Agarwal, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, Volume: 23 , Issue: 6, pp:49 – 58, 1990

[32] T. Lovett and S. Thakkar. “The symmetry multiprocessor system,” in *ICPP*, pp:21-26, Aug. 1988

[33] M. Kistler, M. Perrone, F. Petrini, “Cell Multiprocessor Communication Network: Built for Speed” *IEEE Micro*, pp:10-23, 2006

[34] A.Wilson. “Hierarchical cache/bus architecture for shared memory multiprocessors,” In *ISCA-14*, June 1987.

[35] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *IEEE Computer*, 30(9), 1997.

[36] Shin, J.L, et al, “A 40 nm 16-Core 128-Thread SPARC SoC Processor,” *IEEE Journal of Solid-State Circuits*, Vol : 46 , Issue:1, pp:131 – 144, Jan.2011

[37] D. Bertozzi and L. Benini, “Xpipes: a network-on-chip architecture for gigascale systems-on-chip,”, vol. 4, no. 2. *IEEE in Circuits and Systems Magazine*, pp. 18–31, 2004

[38] W. J. Dally and B. Towles, “Route packets, not wires: on-chip interconnection networks,” in Proceedings of the *Design Automation Conference*, Las Vegas, NV, pp. 684–689, June 2001

[39] A. Hemani, A. Jantsch, S. Kumar, A. Postula, J. Oberg, M. Millberg, and D. Lindqvist, “Network on chip: An architecture for billion transistor era,” in Proceeding of the *IEEE NorChip Conference*, pp.250-255, Nov. 2000

[40] Dally, W.J, “Wire-Efficient VLSI Multiprocessor Communication Networks,” Proceedings of the Stanford Conference on *Advanced Research in VLSI*, Paul Losleben, ed., MIT Press, pp. 391-415, March 1987

[41] W. J. Dally, “Virtual-Channel Flow Control,” In Proceedings of the 17th Annual *International Symposium on Computer Architecture* , pp. 28 –34, Jan .1990

[42] Dally, W.J. and Seitz, C.L, “Deadlock Free Message Routing in Multiprocessor Interconnection Networks,” *IEEE Transactions on Compute*rs, Vol C-3G, No5, pp. 547-553, May 1987

[43] Vangal, S.R, et al, “An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS,” *IEEE Journal of Solid-State Circuits*, Vol : 49 , Issue:1, pp. 29 – 41, Oct. 2008

[44] Damjan Lampret et al., "OpenRISC 1000 Architecture Manual", Rev 1.3, 15 Nov 2007. Available from the OpenCores website

[45] P. Merkle, K. Mueller, A. Smolic, and T. Wiegand, “Efficient Compression of Multi-view Video Exploiting Inter-view Dependencies Based on H.264/MPEG4-AVC”, *ICME*, pp. 68-73, July. 2006

[46] A. Smolic, K. Mueller, N. Stefanoski, J. Ostermann, A. Gotchev, G.B. Akar, G.A. Triantafyllidis and A.Koz: “Coding Algorithms for 3DTV-A Survey,” *IEEE Trans. on CSVT*, Vol 7, Issue 11, pp. 1606-1621, Nov. 2007

[47] Amdahl, Gene, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities". *AFIPS Conference* Proceedings (30), pp. 483–485, 1998

[48] M. Baron, Applications Define DSP Speed, Microprocessor Report, Apr.2005, pp. 3–12.

[49] Jiayi Zhu, Peilin Liu, Dajiang Zhou, "An SDRAM controller optimized for high definition video coding application", *ISCAS*, pp.3518–3521, May. 2008

[50] Yu Shengfa, et al, "Instruction-Level Optimization of H.264 Encoder Using SIMD Instructions" *International Conference on Communications, Circuits and Systems*, pp:126 –129, May, 2006

[51] A.K.Jones, R.Hoare, I.S. Kourtev, J. Fazekas, D. Kusic, J. Foster, S. Boddie, A. Muaydh, "A 64-way VLIW/SIMD FPGA architecture and design flow," *ICECS*, Page(s): 499 – 502, Feb. 2004

[52] M. Ekman, F. Dahlgren, and P. Stenström: "Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors," In Proc. of the First *Workshop on Duplicating, Deconstructing, and Debunking*, May 2002.

[53] P. Sweazey and A. J. Smith. "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Future Bus," Proceedings of the 13th International *Symposium on Computer Architecture*, pages 414-423, May 1986.

[54] P. S. Magnusson, F. Larsson, A. Moestedt, B. Werner, F. Dahlgren, M. Karlsson, F. Lundholm, J. Nilsson, P. tenstr öm, and H. Grahn. SimICS/sun4m: A virtual workstation. Proceedings of the USENIX 1998 Annual T*echnical Conference. USENIX Association*, pages 119-130, June 1998.

[55] Xun He, Xiangzhong Fang, Ci Wang, Goto, S, "Parallel HD encoding on CELL," *ISCAS*, Page(s): 1065 – 1068, May. 2009.

[56] Howard, J, et al., "A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS" *ISSCC Dig.* Tech. Papers, pp.108 -109, Jan. 2010.

[57] Yiping Dong, Hua Zhang, Zhen Lin, T. Watanabe, “A novel hardware method to implement a routing algorithm onto Network on Chip,” *ICCCAS*, July. 2010

[58] M. Butler, L. Barnes, D.D. Sarma, B. Gelinas, "Bulldozer: An Approach to Multi-threaded Compute Performance ," *IEEE*, *Micro*, Volume: 31 , Issue: 2, pp. 6-15 , pp. 6 – 15

[59] Dajiang Zhou, Jinjia Zhou, Xun He, Jiayi Zhu, Ji Kong, Peilin Liu, Goto, S, "A 530 Mpixels/s 4096x2160@60fps H.264/AVC High Profile Video Decoder Chip", *IEEE Journal of Solid-State Circuits*, Volume: 46, pp. 777-788, 2011

[60] Dajiang Zhou, Jinjia Zhou, Xun He, Ji Kong, Jiayi Zhu, Peilin Liu, Goto, S, "A 530Mpixels/s 4096×2160@60fps H.264/AVC high profile video decoder chip," 2010 *IEEE Symposium on VLSI Circuits (*VLSIC), Page(s): 171 – 172, Feb, 2010

[61] Xun He, Jinjia Zhou, Dajiang Zhou and Satoshi Goto, "High profile intra prediction architecture for UHD H.264 decoder", *IPSJ Transactions on System LSI Design Methodology*, Vol. 3, No. 2, pp. 303-313, Aug. 2010

[62] C.-C. Ju, et al., “A 125Mpixels/sec full-HD MPEG-2/H.264/VC-1 video decoder for Blu-ray applications,” in *A-SSCC Dig*. Tech. Papers, pp.9–12, Nov. 2008

[63] C. C. Lin, et al., “A 160k gates/4.5kB SRAM H.264 video decoder for HDTV applications,” *IEEE J. Solid State Circuits*, vol. 42, no. 1, pp.170-182, Jan. 2007

[64] E.Sahin, I.Hamzaoglu, “An Efficient Intra Prediction Hardware Architecture for H.264 Video Decoding,” *DSD*, pp.448–454, Aug. 2007

[65] T.-A. Lin, S.-Z.Wang, T.-M. Liu, and C.-Y. Lee, “An H.264/AVC decoder with 4x4-block level pipeline,” in Proc. *IEEE Int. Symp. Circuits Syst*., pp. 1810–1813, May 2005

[66] C. W. Ku, C. C. Cheng, G. S. Yu, M. C. Tsai, and T. S. Chang, “A high-definition H.264/AVC intra-frame codec IP for digital video and still camera applications,” *IEEE Trans. Circuits Syst. Video Technol*, vol. 16, pp.917–928, Aug. 2006

[67] T.-A. Lin, S.-Z. Wang, T.-M. Liu, “Architecture Design of H.264/AVC Decoder with Hybrid Task Pipelining for High Definition Videos,” *ISCAS*, pp.1810–1813, May 2005

[68] Y. Huang, B. Hsieh, T. Chen, and L. Chen, “Analysis, Fast Algorithm, and VLSI Architecture Design for H.264/AVC Intra Frame Coder”, *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 15, Mar. 2005

[69] W.T. Staehler, A.A. Susin, “Real-Time 4x4 Intraframe Prediction Architecture for a H.264 Decoder,” in *ITS*, pp.416–421, Feb. 2006

Publications

Journals:

[1] **Xun He**, Jinjia Zhou, Dajiang Zhou and Satoshi Goto, " A 98 GMACs/W 32-Core Vector Processor in 65nm CMOS ", *IEICE Transactions on Fundamental*, Vol.E94-A, No.12, Dec. 2011.

[2] Dajiang Zhou, Jinjia Zhou, **Xun He**, Jiayi Zhu, Ji Kong, Peilin Liu, and Satoshi Goto, "A 530Mpixels/s 4096x2160@60fps H.264/AVC high profile video decoder chip," *IEEE Journal of Solid-State Circuits*, Volume: 46 , Issue: 4. Page(s): 777 – 788, 2011

[3] **Xun He**, Jinjia Zhou, Dajiang Zhou and Satoshi Goto, "High profile intra prediction architecture for UHD H.264 decoder", *IPSJ Transactions on System LSI Design Methodology*, Vol. 3, No. 2, pp. 303-313, Aug. 2010

[4] Jinjia Zhou, Dajiang Zhou, **Xun He** and Satoshi Goto, "A bandwidth optimized, 64 cycles/MB joint parameter decoder architecture for ultra high definition H.264/AVC applications", *IEICE Transactions on Fundamentals*, Vol. E93-A, No. 8, pp. 1425-1433, E93.A.1425, Aug. 2010

International Conference Papers:

[1] **Xun He**, Dajiang Zhou, Xin Jin and Satoshi Goto, "A 98 GMACs/W 32-Core Vector Processor in 65nm CMOS ", *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 373 – 378, Aug. 2011

[2] **Xun He**, Xin Jin, Minghui Wang and Satoshi Goto, "A Novel Depth-Image Based View Synthesis Scheme for Multiview and 3DTV," *The 17th International Conference on MultiMedia Modeling* (MMM), Taipei, Taiwan, Part I, LNCS 6523, pp. 161-170, Jan. 2011

[3] Dajiang Zhou, Jinjia Zhou, **Xun He**, Ji Kong, Jiayi Zhu, Peilin Liu and Satoshi Goto, "A 530Mpixels/s 4096x2160@60fps H.264/AVC high profile video decoder chip", *Symposium on VLSI Circuits 2010*, pp. 171-172, June, 2010

[4] Gang He, **Xun He** and Satoshi Goto, "The Hybrid of dynamic and static allocation directory for cache coherence", *ITC-CSCC 2010*, pp.154-157, July, 2010

[5] **Xun He**, Dajiang Zhou, Jinjia Zhou and Satoshi Goto, "A New Architecture for High Performance Intra Prediction in H.264 Decoder", *International Symposium on Intelligent Signal Processing and Communication System*, pp. 41-44, Dec. 2009

[6] **Xun He**, Xiangzhong Fang, Ci Wang, Satoshi Goto, "Parallel HD Encoding on CELL," *International Symposium on Circuits and Systems*, pp. 1065 – 1068, MAY 2009

[7] **Xun He**, Xianmin Chen, Peilin Liu, Satoshi Goto, "A New DCT-Domain Distortion Model for MB-Level Quality Control", *ICCMS'* 09. 20-22, pp. 69 – 72, Feb. 2009