Block Processor: A Resource-distributed Architecture

Zeke Wang Feng Yu
Institute of Digital Technology and Instrument
Zhejiang University
Hangzhou, China
wangzeke@zju.edu.cn, osfengyu@zju.edu.cn

Xue Liu
Institute of Cyber-Physical Systems Engineering,
Northeastern University
Shenyang, China
liuxue0412@tom.com

***Abstract*—We present the architecture of Block Processor, task-level coprocessor, to execute vectorizable computing task migrated from main processor via command bus. The Block Processor is designed around 32 high-MVL block registers, which can be direct operands of vector instruction and be local cache of the Block Processor. The corresponding unique conflict-solving mechanism scales with the various implementations and easily supports chaining by adding extra execution states. The architecture distributes the block registers, ALUs and control logic. We implement the Block Processor which maps efficiently into the FPGA since the FPGA also distributes its inner resource. Each block register requires two FPGA Block RAM to be 2-read-1-write-port, 1024-depth and 32-bit-width. With the enhanced chaining and decoupling, it might hinder the latency of vector memory instructions and then sustain the computing abilities. With the little resource occupied, 1024-point radix-2 DIF FFT costs 11348 cycles on one Block Processor.**

Keywords—block register; chaining; vector; co-processor

I. INTRODUCTION

Modern vector architectures [11] (e.g. VIRAM [1], Tarantula [2], CODE [3], T0 [4]) can offer significant performance advantages over superscalar processors for compute-intensive applications. However, there are three well-known obstacles [3] to the widespread adoption of vector architectures: the complexity of a multi-ported centralized vector register file (VRF), the difficulty of implementing precise exceptions for vector instructions, and the high penalty of on-chip vector memory systems.

Espasa introduced Tarantula [2] to improve the performance of vector architecture via a centralized vector register file, large on chip L2 cache and multiple lanes. Nevertheless, Tarantula is limited to 3 VFUs due to the VRF complexity. Kozyrakis [1, 3] presented CODE to overcome the limitations of conventional vector processors in terms of clustered vector register file, decoupling, and renaming table. Clustered vector register file not only reduces the VRF complexity, but also supply more VRF ports to execute more vector instructions in flight. Decoupling makes sure that stalls in long latency operations do not affect instructions in other clusters, especially rigorously long-latency vector memory instructions. Renaming table determines the exact cluster where the corresponding instruction executes, and eliminates the name dependencies (WAW and WAR register hazards). However, CODE does not offer powerful chaining to further reduce the latency between some otherwise chain-able instructions. The cluster in the CODE cannot support out-of-order execution which can eliminate a large percentage of unnecessary dependencies. Out-of-order execution is necessary not only between load/store instructions and arithmetic instructions, but also between two arithmetic instructions, especially for the vector processor with higher-MVL(maximum vector length) vector register file.

In the paper, we present the flexible architecture of Block Processor. It is a task-level coprocessor to execute computing task migrated from main processor via command bus. The vector unit of the Block Processor is designed around 32 block registers. The block register must have high MVL to serve as local cache of the architecture, not only instruction operand. No block registers combine their read/write ports to form distributed register file (DRF) for any cluster. Hence, no block register is dedicated to any cluster, which must arbiter for the ownership of the read/write ports of the corresponding block registers. The unique conflict-solving mechanism is presented to handle the related dependencies. Further, it can easily support enhanced chaining by adding a few extra execution states, and scales with the various implementations which have different data-paths among the block registers. In sum, The architecture of Block Processor distributes the block registers, ALUs and control logic.

As CMOS technology advances, modern FPGA can offer more and more reprogrammable fabric, including on-chip block memories [12], DSPs, and logic. And the above resources distributes uniformly in FPGA. As a sequence, how to architect FPGA resources to meet the requirement of a range of application gains more and more attention. In particular, implementing a soft processor [5-8] in an FPGA can have considerable computing capacities, especially soft vector processor [6-8] with multiple vector lanes.

We implement the Block Processor which maps efficiently into an FPGA and then provides a significant performance for a reasonable amount of area. The Block Processor is designed around 32 block registers, which are implemented with dual-ported block RAMs [12] in an FPGA. That is, the block register is 1024-MVL and 32-bit width. In order to achieve three ports (2-read and 1-write), block RAMs are replicated to implement each block register. Furthermore, the enhanced chaining is implemented to eliminate unnecessary latencies of vector instruction which is executed in terms of 1024-MVL block registers.

The rest of the paper is organized as follows. Section Ⅱ presents the basic features of the Block Processor. Section Ⅲ presents the Vector Unit implemented on the FPGA, Section Ⅳ presents the block register conflict-solving mechanism. Section Ⅴ evaluates the Block Processor via FPGA-based implementation. And section Ⅵ concludes the paper.

978-1-4799-1365-7/13/$31.00 ©2013 IEEE

II. BLOCK PROCESSOR ARCHITECTURE

Block Processor, task-level coprocessor, is crystallized around the need to execute computing task migrated from the main processor via command bus.

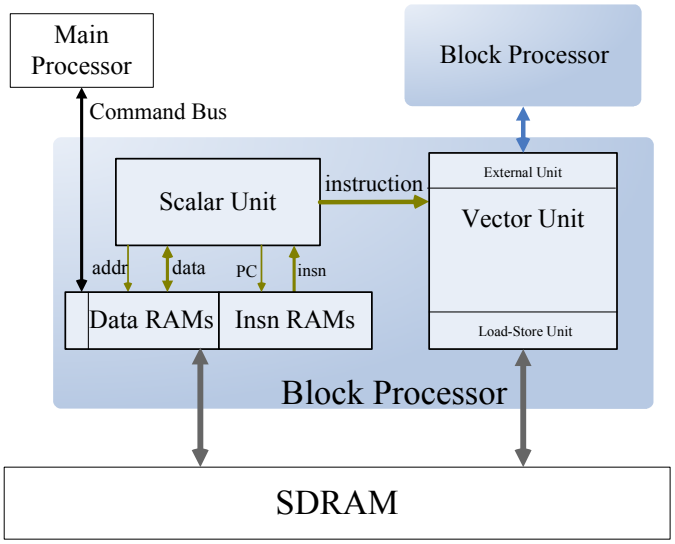

Fig.1. The block diagram of Block Processor. Block Processor consists of the scalar unit and vector unit.

Block Processor mainly consists of a scalar unit and a vector unit, as shown in Fig.1. The Block Processor employs decoupling execution to separate an instruction sequence into multiple streams, to execute maximum number of instructions in flight. Furthermore, the decoupling exists not only between the scalar unit and vector unit, but also between any two clusters in the vector unit.

The Block Processor can support precise interrupt, but the respond time is relatively long, since the vector unit cannot execute new vector instructions until all previous pending vector instructions complete the executions.

A. *Scalar Unit*

The basic role of scalar unit is to communicate with main processor and to serve the vector unit via supplying the vector instructions with the corresponding decoded general-purpose scalar register values.

The scalar unit has a classic five-stage RISC pipeline. The corresponding scalar ISA is a subset of that of MicroBlaze[5] which excludes floating point and virtual memory instructions. We also add some special instructions to communicate to vector unit, e.g. statusForm. The instruction (statusForm) is used to combine the vector length and vector stride (not valid for all vector instructions) into one 32-bit register. Hence, we can employ general-purpose register to pass parameters (vector length and vector stride) to the vector unit. That is, no special vector length/stride registers [1] are required. Destination register (we call, STATUS) of the instruction (statusForm) stays in last 5 bits of the vector instruction. When a vector instruction is recognized, the vector instruction, together with the corresponding 32-bit value of STATUS, is passed to vector unit when the instruction queue between the vector unit and the scalar unit is not full. Otherwise, the scalar unit must stall until the full signal negates.

After powered up, the scalar unit is always waiting for the computing task from main processor. Once a computing task arrives, it begins to fill the Insn RAMs with the corresponding instructions (e.g. .text) via requesting DMA transfer, as well as Data RAMs with the corresponding global variables (e.g. .data) of the executable file. Then, the scalar unit fetches and executes instructions from its Insn RAMs. It loads and stores data from and to its Data RAMs, and issue vector instructions to the vector unit.

When the computing task completes, the scalar unit would inform the main processor and waits for another computing task. Due to the decoupling nature between scalar unit and vector unit, the scalar unit can accept another computing task, even when the scalar unit has finished the scalar code and the vector unit has not yet finished computing job. That is, it can hide the startup time of another computing task and then keep the vector unit always busy.

B. *Vector Unit*

The vector unit is the cornerstone of the Block Processor to execute computing task. It usually contains one dispatch unit and several clusters. It receives vector instruction from the scalar unit and then issues to the corresponding cluster for computation. Each cluster contains one instruction queue to manage the incoming vector instructions. The instruction queue supports out-of-order execution to maximize the arithmetic resource utilization.

It is critical to note that each block register has specified number of ports (e.g. 2 read and 1 write ports), not the entire register file; that is, no block registers combine their read/write ports to form distributed register file (DRF) for any cluster. Hence, no block register is dedicated to any cluster, which must arbiter for the ownership of the read/write ports of the corresponding block registers.

In each cluster, the block register file is proposed to combine the read/write ports of the corresponding block registers. The block register file does not contain any dedicated block register, but shares with the block register files in the other clusters. Then, the instruction in the cluster must arbiter for the ownership of the corresponding block registers first, and then read from and write to them.

Because of flexible property of block-register-based data-path, the data-path can be easily tuned for performance balance and various block-register-based implementations exist. We present our implementation of the vector unit in the next section. We also present the corresponding conflict-solving mechanism to scale with the various implementations in the Section Ⅳ.

III. THE PROPOSED VECTOR UNIT

The proposed vector unit consists of the dispatch unit and 8 clusters, as shown in Fig.2. The dispatch unit receives vector instruction from the scalar unit and then issues to the corresponding cluster. The 8 clusters includes 1 load/store unit, 6 compute units and 1 external unit. The 8 clusters are designed around 32 block registers, whose read/write ports are

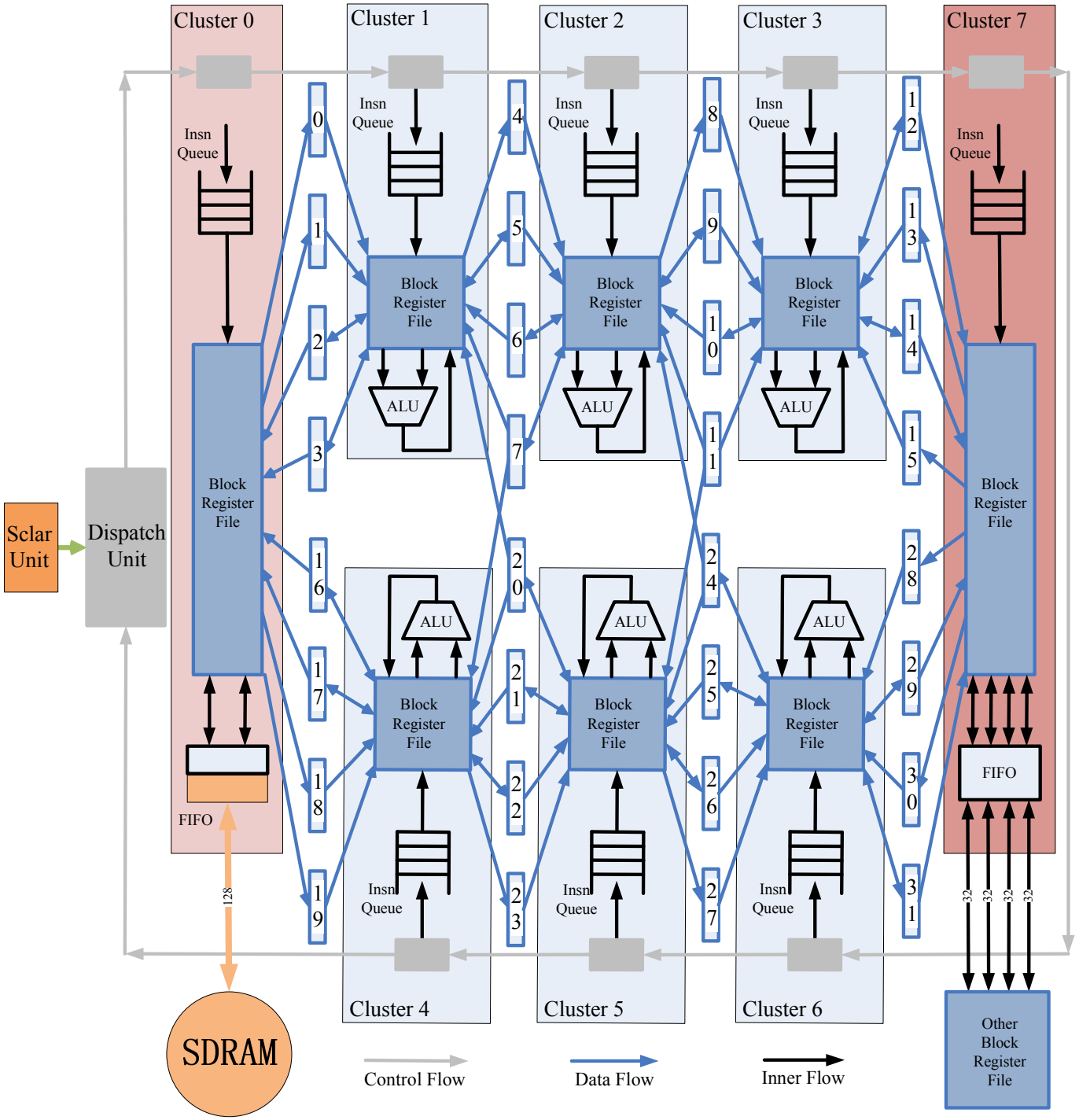

Fig.2. The proposed vector unit contains one dispatch unit and 8 clusters, which has 1 load/store unit (cluster 0), 6 compute units (clusters 1-6) and 1 external unit (cluster 7). The cluster 1 reads from block registers 0,1,2,3,5,6,7,and 20, writes to block registers 2,3,4,and 5.

exposed to 8 block register files in the 8 clusters to source operands and store results. Each block register has 2-read-1-write ports and 1024 32-bit elements. Each block register requires two FPGA Block RAMs to implement since current FPGAs have only dual-ported block RAMs. To the best of our knowledge, the block register is implemented with largest MVL 1024.

Our implementation can execute at most 22 vector instructions simultaneously. We also propose enhanced chaining for block register to further maximize the utilization of arithmetic resource.

The detail of the vector unit is presented as follows.

A. Dispatch Unit

The dispatch unit contains an instruction queue to receive vector instructions from the scalar unit, which issues instructions to vector unit in strict program order. When the instruction queue is not empty, the dispatch unit starts to issue vector instruction (in-order), together with necessary decoded information and execution condition, to the corresponding cluster via control network.

There are a wide variety of alternatives for the control network implementation: ring, bus, crossbar, and so on. The basic trade-off in selecting a network implementation is between performance, cost of hardware resources, and power consumption.

We select ring as our control network implementation for its high clock cycles and low hardware resource cost. The dispatch unit passes each vector instruction (with corresponding decoding information) to the cluster 0, which determines whether the instruction belongs to it. If hit, the instruction is passed to the instruction queue in the cluster 0. Otherwise, the instruction is passed to the cluster 1, which is next stop of the ring, as shown in Fig.2. This process is repeated until the instruction reaches the cluster 4, which is the last stop of the ring. There is a data path from the cluster 4 to the dispatch unit, since the 8 clusters must inform the instruction completions to the dispatch unit via the same ring.

Besides, we choose vector instruction decoding in the dispatch unit since no decoding logic is needed in each cluster [3]. That is, the centralized decoding policy reduces 7 copies of decoding logic while not violating issuing correctness.

However, the proposed dispatch unit cannot generate the proper inter-cluster transfers when the source operands of a vector instruction are not in the selected cluster [3]. It is important to easy-programming. Now we resort to software to detect operand mismatch during programming.

B. Compute Unit

Clusters 1-6 are compute units where we implement single-precision floating-point operations. Each cluster supports one floating-point instruction and one move/logical instruction simultaneously.

The corresponding instruction queue in the compute unit receives vector instructions from the dispatch unit. Once the instruction queue is not empty, the cluster can execute instructions out-of-order via looking ahead in the instruction queue, where instruction window (we choose, 8) can be parameterized. The out-of-order property is critical for large MVL (1024), since each unnecessary dependency might cost at most 1024 cycles. We want to maximize resource utilization by executing the instruction whose operands are available no matter where it locates. Espasa [13] presented out-of-order-issue in-order-commit vector architecture via register renaming and reordering buffer, while we resort to in-order-issue out-of-order-commit vector architecture to boot the vector performance.

Each compute unit has one the block register file which combines the read/write ports of the corresponding block registers. In particular, the block register file in the cluster 1 reads data from 8 block registers (0,1,2,3,5,6,7 and 20), and writes to 4 block registers (2,3,4 and 5), as shown in Fig.2. Two block register files arbiter for the read/write ports of one block register. For example, the block register files of clusters 1 and 2 content for the block register 5. It means that the read/write ports of the block register 5 are shared by the clusters 1 and 2.

C. Load/Store Unit

The cluster 0 is dedicated to load/store unit which loads and stores data to and from SDRAM. The decoupling nature between the load/store unit and the other units is to tolerate long memory latency [1, 3, and 9].

Hot channel conflict, bank conflict and bus read/write transition conflict are inevitable for DRAM operations. So memory operations should be in long burst mode to reduce the above three conflicts. In particular, once one device wins the arbitration, continual granting of mastership should be given to load (or store) data, under the assumption that unit-stride load (or store) always occurs. Of course, each device must cooperate to request unit-stride load (or store), since the device is the master who initiates the DRAM operations.

We should also pay attention to two rate mismatch factors: clock frequency and bus width conversion, e.g. 200M clock frequency and 128-bit width for SDRAM controller versus 250M clock frequency and 32-bit width for the block register. Two mismatch factors combine dynamically together and can result in a mismatched transaction rate between devices at any given instant in time. So we employ FIFOs for command and data buffering to absorb temporary differences in transaction rates.

The load/store unit supports 2 load and 2 store operations simultaneously, while it can support 4 load and 4 store operations in theory. In order to reduce DRAM conflict, there is only one load (or store) instruction accessing SDRAM at one time, while the parallelization exists between block registers and FIFOs. In particular, two block registers in the load/store unit can write data to FIFOs simultaneously.

D. External Unit

The cluster 7 is external unit which exchanges data between two block registers of two external units. The corresponding instruction queue also supports executing instructions out-of-order, while the instruction widow is 8.

The clusters 0-6 cannot directly communicate data with the other Block Processors and must move data to the corresponding external unit first. Inter-communication takes place by executing a tagged write-fifo instruction (BwrFifo) of the sending Block Processor and a tagged read-fifo instruction (BrdFifo) of the receiving Block Processor respectively. The external unit can support 4 pairs of external instructions in flight, as shown in Fig.2.

The external unit also supports 2 pairs of move instructions simultaneously in the corresponding block register file. It is also possible to support floating-point operations to enhance the floating-point computing performance.

IV. BLOCK REGISTER CONFLICT-SOLVING MECHANISM

As we all know, the classic data hazards include RAW, WAW and WAR, and the conventional dependence-eliminating mechanism is Tomasulo's approach [10]: register renaming and reservation station. The basic requirement of register renaming is more physical registers to temporally store results of executing instructions. It is not costly for general-purpose register file, but costly for block register with its large MVL. According to the properties of block register, we propose the unique conflict-solving mechanism to address the three data hazards, in the scoreboard-like way. The mechanism can scale with different implementations since the cluster would collect the accompanying execution states of the corresponding block registers to determine whether the pending instruction could execute; that is, no extra execution states are included if the corresponding block register is not referenced by the cluster. Execution states, whose physical representations (for example, PWC), are combining the meanings of containing symbols (for example, P and WC) shown in Table.1.

A. Basic mechanism

In the dispatch unit, the execution condition is copied and then passed to the cluster (in strict program order), including PWC for VA/VB, PRC and PWC for VD. After the execution condition is copied, the execution condition (PRC and PWC) should be updated before the next instruction is issued. The condition updating includes increasing PRC by one for VA/VB, and increasing PWC by one for VD. Increasing PRC by one for VA/VB means the VA/VB is read once more; that is, the next write to VA/VB should make sure that the read has been finished. Similarly, increasing PWC by one for VD means the VD is written once more; that is, the next read to VD should make sure the write has been finished.

Table 1. conflict-solving symbol conventions.

symbol	full name	Description
RC	read count	number of times the corresponding block register (source) is read
WC	write count	number of times the corresponding block register (destination) is written
P	predicted	dispatch-generated execution condition
E	executed	execution status from block registers
S	start	updated at the start of instruction
VA/VB	source	source register where ALU gets data
VD	destination	destination register where ALU stores data

When the instruction, together with the corresponding execution condition, is passed to the corresponding cluster via control network, they are added to the corresponding instruction queue. Once the queue is not empty, the cluster would determine whether its execution condition is satisfied.

The cluster would collect the up-to-date accompanying execution states, EWC and ERC, of the corresponding block registers to judge whether the pending instruction is executable. The execution states accompany along with its corresponding block registers, and then scales with the architecture.

RAW hazard can be eliminated when the condition, VA/VB_PWC == VA/VB_EWC, is satisfied, shown in "basic" of Fig.3a; that is, the source block register VA/VB could read the correct data if the previous write to VA/VB has finished.

WAW hazard can be eliminated when the condition, VD_PWC == VD_EWC, is satisfied; that is, the write to the destination block register VD must be in order and no out of order is allowed.

WAR hazard can be eliminated when the condition, VD_PRC == VD_ERC, is satisfied, shown in "basic" of Fig.3b; that is, the write to VD is allowed only if the previous read to VD has finished.

After the instruction in the cluster has finished computation, it is necessary to update the execution states, ERC for VA/VB and EWC for VD, to relax the dependencies of the pending instructions in the concerned clusters. In particular, increasing ERC by one means the VA/VB has finished the corresponding read operation, and increasing EWC by one means the VD has finished the corresponding write operation.

B. Chaining-enable mechanism

The chaining is of extreme importance to our design due to its large MVL, and each unnecessary dependency would cost as many as 1024 cycles, so we implement enhanced chaining, which can relax unnecessary dependencies to a certain extent between block registers, as long as data dependencies between elements are preserved. Chaining [1, 3] allows dependent instructions to overlap their execution by forwarding the result of element operations before the whole instruction completes.

The basic conflict-solving mechanism is to distinguish when the block register completes read (or write). However, the key rule to implement the chaining is to additionally distinguish when the block register begins read (or write), together with when the block register completes read (or write). Hence, we import additional execution states, SEWC, SERC, in_order_rd and in_order_wr, to determine when the block register begins read (or write). No change in the generation (updating) of execution condition is needed in the dispatch unit, just the same as the basic mechanism.

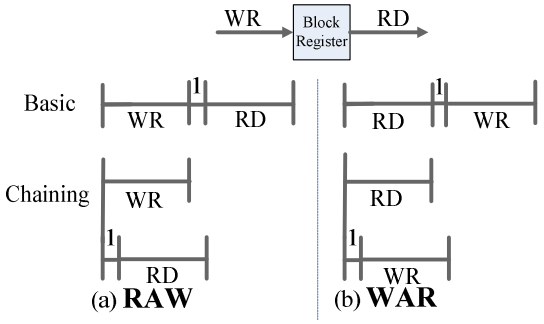

Fig.3. Timings between write port and read port of the block register, both basic and chaining. The 1-clock cycle delay is the latency of crossover between read and write operations.

As long as the dependency between read port and write port is preserved, the dependency should be relaxed to maximize the utilization of arithmetic resource. By importing chaining technology, the three data hazards are relaxed as follows.

RAW hazard can also be eliminated when the additional condition, VA/VB_in_order_wr & VA/VB_in_order_rd & (VA/VB_PWC == VA/VB_SEWC), is satisfied, shown in "chaining" of Fig.3a. The three conditions should be satisfied to enable chaining. (VA/VB_PWC == VA/VB_SEWC) means the previous write has begun, VA/VB_in_order_wr, indexed by VA/VB in in_order_wr, determines whether the previous write to VA/VB is in-order, and VA/VB_in_order_rd, natural instruction property, determines whether the read is in-order.

WAW hazard cannot be relaxed by importing chaining technology, since each block register has only one write port.

WAR hazard can also be eliminated when the additional condition, VD_in_order_wr & VD_in_order_rd & (VD_PRC == VD_SERC), is satisfied, shown in "chaining" of Fig.3b. (VD_PRC == VD_SERC) means the previous read has begun. VD_in_order_rd, indexed by VD in in_order_rd, determines whether the previous read to VD is in-order, and VD_in_order_wr, natural instruction property, determines whether the write is in-order.

When the instruction in the cluster begins computation, it is necessary to update the execution states, SERC and in_order_rd for VA/VB, and in_order_wr and EWC for VD, to relax the dependencies. Increasing SERC by one means the VA/VB has begun the corresponding read; in_order_rd indexed by VA/VB is set 1(0) when the VA/VB is in-order (out-of-order) read. Increasing SEWC by one means the VD has begun the corresponding write operation; in_order_wr indexed by VD is set 1 (0) when the VD is in-order (out-of-order) write.

We can implement enhanced chaining by importing more execution states, which determine another condition to eliminate dependence, based on the existing conditions.

Table 2. Block Processor Resource occupied.

	Available	Occupied	Percent
LUTs	149,760	19,818	13.2%
FFs	149,760	19,719	13.2%
BlockRAMs	516	90	17.4%
DSP48Es	1056	97	9.2%

V. Evaluation and Performance

We implement the Block Processor in Xilinx Virtex-5 XC5VSX240T, and the resource occupied is shown in Table 2. The BlockRAM utilization is relatively high, since the BlockRAMs must be replicated to implement 2-read and 1-write ports of block register. However, the DSP48 utilization is relatively low, and we could compute two single-precision floating-point numbers per cycle in a vector instruction. We can implement several copies of Block Processor to improve the temporal locality in the data references, and then significantly reduce memory references via exchanging data between external units.

We implement 250 clock frequency of the vector unit and 200M for the scalar unit. The basic required elements (e.g. block register and instruction queue) are easy to implement, it could be implemented as an ASIC prototype.

We implement 1024-point radix-2 DIF FFT on one Block Processor. It costs 11348 cycles with sufficient memory bandwidth, while the start time is that the scalar unit begins to execute FFT code and the stop time is that the vector unit finishes storing data back to SDRAM.

FFT algorithm needs to load data and twiddle factors into block registers, and then start the vector arithmetic operations. Sequentially, it is hard to sustain the computing logic at the loading stage of the FFT, though the decoupling exists between the load/store unit and compute unit. The same thing also occurs in the storing stage. Obviously, the loading and storing stages take roughly one third of overall cycles, which can be reduced by a factor of 2 via supporting 4 load and 4 store vector instructions in flight in the load/store unit, since more than 4 load (or store) instructions are available in the loading and storing stages.

VI. Conclusion

We have presented the Block Processor to boot the vector performance. It is designed around 32 block registers, whose read/write ports are exposed to programmer or compiler to maximize executing instructions in flight. The corresponding MVL is 1024, and we implement enhanced chaining to service the 32 Block registers. In each cluster, it also supports out-of-order execution.

References

[1] C. Kozyrakis (2002). Scalable Vector Media-processors for Embedded Systems. PhD thesis, Computer Science Division, University of California at Berkeley.

[2] R. Espasa, et al (2002). Tarantula: a Vector Extension to the Alpha Architecture. In the Proceedings of the 29th Intl. Symp. on Computer Architecture, Anchorage, AL, pp.281–291.

[3] C. Kozyrakis and D. Patterson (2003). Overcoming the limitations of conventional vector processors. SIGARCH Comput. Archit. News, 31(2), pp.399–409.

[4] K. Asanovi´c(1998). Vector Microprocessors. PhD thesis, Computer Science Division, University of California at Berkeley.

[5] Xilinx, Inc(2008). MicroBlaze Processor Reference Guide

[6] P. Yiannacouras, et al(2008). Vespa: Portable, scalable, and flexible fpga-based vector processors. In CASES'08: International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM.

[7] J. Yu, et al (2008). Vector processing as a soft-core cpu accelerator. In Symposium on Field programmable gate arrays, New York, NY, USA, ACM, pp.222–232 .

[8] P. Yiannacouras, et al(2009). Fine-Grain Performance Scaling of Soft Vector Processors. In CASES'09: International Conference on Compilers, Architecture and Synthesis for Embedded Systems. ACM.

[9] J. Smith(1984). Decoupled Access/Execute Computer Architecture. ACM Transactions on Computer Systems, 2(4): pp.289–308.

[10] K. Asanovi´c. Vector Microprocessors. PhD thesis, Computer Science Division, University of California at Berkeley, 1998.

[11] J. Hennessy, D. Patterson (2007). Computer Architecture: A Quantitative Approach. fourth edition. Morgan Kaufmann, San Francisco, CA.

[12] Xilinx, Inc(2009). Virtex-5 FPGA User Guide.

[13] R. Espasa, et al (1997). Out-of-order Vector Architectures. In the Proceedings of the 30th Intl. Symp. on Microarchitecture, Research Triangle Park, NC, pp.160–70.