A Prototype Multithreaded Associative SIMD Processor

Kevin Schaffer and Robert A. Walker
Department of Computer Science
Kent State University
Kent, Ohio 44242
{kschaffe, walker}@cs.kent.edu

Abstract

The performance of SIMD processors is often limited by the time it takes to transfer data between the centralized control unit and the parallel processor array. This is especially true of hybrid SIMD models, such as associative computing, that make extensive use of global search operations. Pipelining instruction broadcast can help, but is not enough to solve the problem, especially for massively parallel processors with thousands of processing elements. In this paper, we describe a SIMD processor architecture that combines a fully pipelined broadcast/reduction network with hardware multithreading to reduce performance degradation as the number of processors is scaled up.

1. Introduction

Single Instruction Multiple Data (SIMD) computers are often used to solve computationally intensive problems that exhibit a high degree of fine-grain data parallelism. The appeal of the SIMD model is its simplicity, from the perspective of both software and hardware. A SIMD computer maintains the same implicit synchronization as a sequential computer, making it easier to program

At the hardware level, a SIMD computer is built up from relatively simple pieces, making it easy to adapt the SIMD model to new technologies, such as systems-on-a-chip. Commercial single-chip SIMD processor arrays are available for a variety of applications [1, 2].

However, the performance of a SIMD processor is often limited by the broadcast/reduction bottleneck [3]. As the number of processing elements (PEs) increases, the wires connecting the control unit and the PEs become longer and signals take longer to propagate. Since each instruction must be broadcast, this delay increases the clock cycle time, limiting system performance. Similarly, most SIMD systems support AND, OR, or other reduction operations, which also act as a bottleneck. Pipelining the broadcast/reduction network helps to keep the clock cycle time short, but it introduces hazards that can severely limit system performance (as described later in Section 4).

In this paper we will describe a prototype associative SIMD processor architecture that uses a combination of pipelining and multithreading to maintain high performance as the number of PEs increases.

2. Associative Computing

Developed within the Department of Computer Science at Kent State University, the associative computing (ASC) model grew out of work on the STARAN and MPP SIMD computers at Goodyear Aerospace [4]. In a SIMD computer, there is a single control unit, also known as an instruction stream, and multiple processing elements (PEs) combined with local memory, known collectively as cells. The control unit fetches and decodes program instructions, and broadcasts control signals to the PEs. The PEs, under the direction of the control unit, execute these instructions using their own local data. SIMD computers are particularly well suited for massively data parallel programming, where the same operation is applied to each element of a data set.

An ASC processor is a SIMD computer that has additional hardware, in the form of a broadcast/reduction network, to support specific high-speed global operations. An ASC computer must be able to: broadcast from the control unit to all PEs; search all PEs in parallel; detect the presence of PEs whose search was successful (called *responders*); pick one responder; reduce a value from the PEs to the control unit using either bitwise AND or OR; and find the maximum or minimum value across all PEs. The ASC computing model extends the data parallel SIMD programming by adding these associative operations.

3. ASC Processor Prototypes

Several students at Kent State have developed a prototype ASC Processor that implements the associative computing model.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

The first version of the ASC Processor [5] had an 8-bit control unit, 4 8-bit PEs, and a broadcast/reduction network capable of maximum/minimum search, responder detection, and multiple responder resolution. The instruction set was modeled after RISC processors such as MIPS and DLX with associative features similar to the STARAN. The processor was designed in VHDL and targeted at an Altera FLEX 10K70 FPGA, however this design was never fully implemented.

The ASC Processor was later redesigned [6] and made scalable, so that the number of PEs could be changed easily. To make the processor scalable, the broadcast/reduction network had to be redesigned, as the network in the first processor could only handle four PEs. There were no major changes to the instruction set architecture, so this scalable ASC Processor could run most of the software written for the first one. The new scalable ASC Processor was then implemented in a larger Altera APEX 20K1000 FPGA, which could hold the control unit and up to 50 PEs.

To further improve performance, the instruction execution of the scalable ASC Processor was pipelined [7]. The pipelined ASC Processor used a classic RISC pipeline with five stages: instruction fetch, instruction decode, execution, memory access, and write back, though that pipeline spanned the control unit and PE array. While this pipeline did improve performance over the original non-pipelined implementation, it still suffered from the broadcast/reduction bottleneck because the broadcast and reduction operations were not pipelined.

4. Pipelining

We can overcome the broadcast/reduction bottleneck in the ASC Processor (or any other SIMD processor) by pipelining the broadcast and reduction networks. A pipelined broadcast network is a k-ary tree with a register at each node. It can accept a new instruction each clock cycle and it delivers an instruction to the PE array after a latency of $\log_k n$ cycles, for a machine with n PEs. A pipelined reduction network is similar except that data flows in the opposite direction and at each node a functional unit combines k values together before storing the result in a register.

Since the distance that signals must propagate in a pipelined broadcast or reduction network is shorter, the network can operate at a much higher clock rate than a non-pipelined network. Unfortunately, the pipelined networks introduce additional hazards as described next.

4.1. Pipeline Organization

Instructions in a SIMD processor can be classified into three types: *scalar instructions* execute within the control unit; *parallel instructions* execute on the PE array and require the use of the broadcast network; and *reduction instructions* execute on the PE array and require the use of both the broadcast and reduction networks. The pipeline has separate paths for each instruction type, so that an instruction does not waste time in unused pipeline stages.

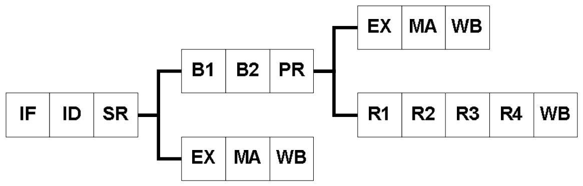

Figure 1. Pipeline organization.

Figure 1 shows the pipeline organization. After the scalar register read stage (SR) the pipeline splits, with scalar instructions taking the lower path, moving to the execute stage (EX), and parallel and reduction instructions taking the upper path, moving to the first broadcast stage (B1). The parallel pipeline then splits once again after the parallel register read stage (PR), with parallel instructions taking the upper path, moving to the execute stage (EX), and reduction instructions taking the lower path, moving to the first reduction stage (R1).

The number of broadcast and reduction stages is variable, depending on the number of PEs. To simplify the figures, two broadcast stages (B1-B2) and four reduction stages (R1-R4) are assumed.

4.2. Hazards

Broadcast hazards can occur when a parallel instruction uses the result of an earlier scalar instruction. With the split pipeline organization, classic data forwarding can eliminate stalls caused by broadcast hazards. As shown in Figure 2, the result of the scalar `SUB` instruction is not available until it enters the memory access stage (MA), while the parallel `PADD` instruction needs that result by the first broadcast stage (B1). In this case a stall can be avoided by forwarding the result from the scalar EX stage to the parallel B1 stage.

Reduction hazards can occur when a scalar instruction uses the result of an earlier reduction instruction. As shown in Figure 2, the scalar `SUB` instruction consumes its operands in the execute stage (EX), but the reduction `RMAX` result is not available until the write back stage (WB). In this case, the scalar instruction has to stall for up to $b + r$ clock cycles, where b is the latency of the broadcast network and r is the latency of the reduction network. In the figure, a stall is indicated by having the instruction repeat the instruction decode (ID) stage.

Broadcast-reduction hazards can occur when a parallel instruction uses the result of an earlier reduction instruction. The effect of a broadcast-reduction hazard is a combination of previous two types of hazards. The parallel

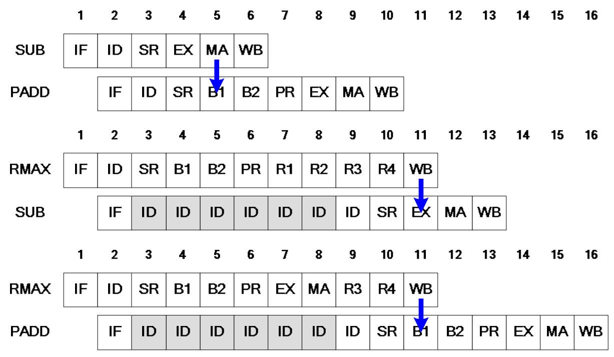
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
SUB IF ID SR EX MA WB
PADD IF ID SR B1 B2 PR EX MA WB
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
RMAX IF ID SR B1 B2 PR R1 R2 R3 R4 WB
SUB IF ID ID ID ID ID ID ID SR EX MA WB
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
RMAX IF ID SR B1 B2 PR EX MA R3 R4 WB
PADD IF ID ID ID ID ID ID ID SR B1 B2 PR EX MA WB

Figure 2. Examples of pipeline hazards (from top to bottom): broadcast, reduction, broadcast-reduction.

`PADD` instruction's operand must be available by the first broadcast stage (B1), but the reduction `RMAX` instruction's result is not available until the write back stage (WB), as shown in Figure 2. In in the case of reduction hazards, the parallel instruction must again stall, this time for at most $b + r$ clock cycles.

5. Multithreading

As illustrated in the bottom two examples in Figure 2, reduction instructions can stall the processor for several cycles. The compiler or programmer could schedule the instructions in order to diminish the number of stall cycles, but the exact latency of reduction instructions depends on the number of PEs, which is generally not known at compile time. Furthermore, for a large machine, the latency could be much higher than the degree of instruction-level parallelism (ILP) in the code.

Hardware *multithreading* is a better solution to reducing the number of stalls. A multithreaded processor dynamically schedules instructions from multiple threads in order to avoid stalls caused by high latency operations. Multithreading exploits thread-level parallelism (TLP), which scales much better than ILP.

Coarse-grain multithreading [8] allows a thread to run until it encounters an operation that would cause it to stall, at which point the processor switches to a different thread. During a thread switch, the pipeline is flushed and the machine state is updated before fetching instructions from the new thread. It takes many cycles to perform a thread switch, so this switch is performed only for high latency operations that occur infrequently, such as cache misses or remote memory accesses. At any given point in time, all the instructions in the pipeline are from the same thread, so the amount of additional hardware necessary to support coarse-gain multithreading is relatively small.

Fine-grain multithreading [8], in contrast, switches threads on every clock cycle, so it can improve throughput even in the face of shorter, more frequent stalls such as those from branch mispredictions and data hazards. So long as there is at least one thread that is not stalled in every cycle, a fine-grain multithreaded processor will never stall. Instructions from multiple threads coexist in the pipeline at the same time, so machine state must be replicated for each thread. This results in a higher hardware cost than coarse-grain multithreading.

Simultaneous multithreading (SMT) [9] differs from coarse-grain and fine-grain multithreading in that it can issue instructions from multiple threads in the same clock cycle. Even with aggressive out-of-order issue logic, it is difficult to find enough instructions in a single thread to fill all the issue slots in a wide-issue superscalar processor, but by looking at instructions in multiple threads, a SMT processor is better able to fill its issue slots each cycle. In addition to the machine state, pipeline resources such as instruction windows and reorder buffers must also be replicated. SMT has the highest hardware cost of all three approaches.

Recall that the reason for using multithreading in the pipeline of a SIMD processor is to avoid stalls caused by reduction hazards. The latency of a reduction operation depends on the number of PEs and can vary from a few cycles for a small machine to tens of cycles for a larger one, so fine-grain multithreading or SMT is necessary to effectively eliminate stalls in the SIMD pipeline. The Multithreaded ASC Processor described in this paper uses fine-grain multithreading.

6. Multithreaded ASC Processor Architecture

In order to measure the performance and hardware costs of a multithreaded associative SIMD, a prototype processor is currently under development, targeted for an Altera Cyclone II (EP2C35) FPGA. The Multithreaded ASC Processor consists of a control unit, which handles instruction fetch and decode and executes scalar instructions; an array of PEs, which execute parallel instructions; and a broadcast/reduction network to connect the two. In order to support multithreading, machine state, such as the PC and register files, must be replicated, and all the functional units, including the broadcast/reduction network, must be pipelined.

6.1. Instruction Set Architecture

The Multithreaded ASC Processor uses an instruction set architecture similar to, but not compatible with, the ISA used in the previous ASC Processors described earlier. The new ISA is a RISC load-store architecture similar to MIPS, but with extensions for SIMD data-parallel computing, associative computing, and multithreading.

Parallel Instructions. Arithmetic, logic, and comparison operations are available in both scalar and parallel instructions. The parallel instructions operate on a separate parallel memory space with a separate set of registers.

Broadcast/Reduction Instructions. Most parallel instructions allow one of the operands to be a scalar value that is broadcast to the PE array prior to performing the operation. There are also a number of reduction instructions that combine parallel data values from the PE array into a scalar value.

Flags. Logical results from comparisons, which play an important role in associative computing, become a first-class data type with their own set of registers and instructions.

Multithreading. The ISA provides instructions to allocate and release hardware threads and to communicate data between threads.

6.2. Processing Element (PE) Organization

Each processing element (PE) in the PE array consists of a local memory, a general-purpose register file, a flag register file, an arithmetic/logic unit (ALU), a multiplier, and a divider.

Local Memory. Each PE has a small amount of local memory that acts as a programmer- or compiler-managed cache. The local memory is shared between threads at the hardware level, though it may be partitioned under software control. The local memory is implemented as one or more block RAMs. The exact number of block RAMs used in each local memory is a configuration issue — a larger memory will reduce off-chip memory traffic, but reduce the number of PEs that can fit on a single FPGA.

General-Purpose Register File. As a load-store architecture, all arithmetic instructions require their operands to be stored in the register file. Load and store instructions transfer data between the register file and the local memory. The register file is split between threads at the hardware level, so that a thread can only access its own registers. Interthread communication instructions, however, can transfer data between registers in different threads.

Implementing the register files efficiently in an FPGA is a significant challenge. The need to fit as many PEs on a chip as possible precludes the use of flip-flop arrays because they waste logic resources. A distributed (LUT-based) RAM implementation is also ruled out due to the need for large register files, in order to support a large number of hardware threads. Block RAMs are the best way to implement the register files since they do not waste logic resources and support sufficient depths. However, the number of block RAMs available on an FPGA chip will likely set the limit on the number of PEs.

Flag Register File. The flag register file is used in much the same way as the general-purpose register file, except that the values stored in the flag register file are 1-bit flags. Just like the general-purpose registers, the flag registers are also split between threads. Implementing the

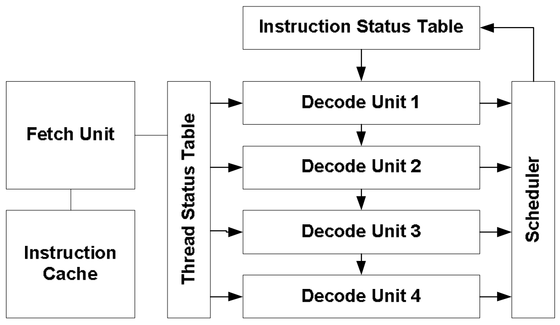

Figure 3. Organization of the control unit.

flag register file efficiently is an even greater challenge than the general-purpose file because the amount of data is so much smaller, using an entire RAM block for a single flag register file would be a waste. The solution is to share one RAM block between multiple PEs.

ALU. The ALU supports a standard set of arithmetic, logic, and comparison operations. Logic operations are supported for both integers (bitwise logic) and flags. Comparisons operate on integers and produce flag results. The ALU has an initiation rate of one operation per cycle and a latency of one cycle. Forwarding paths are provided so that the results of an ALU operation can be sent back to the ALU before they are written into one of the register files.

Multiplier. The multiplier is optional and can be implemented in one of two ways. If the FPGA chip supports hard multiplier blocks then those blocks can be used to implement a fast, fully pipelined multiplier. However, the number of PEs may be restricted if there are insufficient multiplier blocks available on the FPGA. The other option is a sequential multiplier that uses fewer FPGA resources, but is slower and cannot be used by multiple threads simultaneously.

Divider. The divider is also optional, but is only available as a sequential unit. As such, multiple threads cannot use it at the same time. However, since division is an uncommon operation, structural hazards for the divider should not degrade performance significantly.

6.3. Control Unit Organization

The control unit is essentially a multithreaded scalar processor with a few additions to support parallel instructions. The control unit consists of a fetch unit, a decode/issue unit, and a scalar datapath. The organization of the control unit is illustrated in Figure 3.

Fetch Unit. The fetch unit fetches instructions from the instruction cache/memory and places them in an instruction buffer. Each thread's instruction buffer, PC, and state are recorded in a data structure called the thread status table, which is shared between the fetch unit and the decode unit.

Decode Unit. The decode unit decodes the instructions in the buffers and determines which ones are ready to execute. This unit is replicated for each hardware thread so that instructions from different threads can be decoded in parallel.

Scheduler. The scheduler selects a thread that has an instruction ready to execute and issues that instructions to either the scalar datapath or the PE array. A rotating priority selection policy is employed to ensure fairness between threads. The scheduler also maintains the instruction status table, which keeps track of all the instructions currently executing and is used by the decode unit to detect hazards.

Scalar Datapath. The scalar datapath executes scalar instructions and has an organization nearly identical to the PEs. There are additional components to handle scalar-specific instructions such as branches, forks, and joins.

6.4. Broadcast/Reduction Network

A key element of this Multithreaded ASC Processor, the broadcast/reduction network is fully pipelined so that threads never contend for its use. The broadcast network is not pipelined as deeply as the reduction network, since the broadcast network does not perform any computation. The reduction network supports the bitwise AND/OR and maximum/minimum functions, which are required by the ASC model, as well as the count responders and sum functions.

Broadcast Unit. The broadcast unit broadcasts instructions and data from the control unit to the PE array. It is implemented as a pipelined *k*-ary tree with registers at each node. The unit has an initiation rate of one operation per cycle and a latency of $\lg_k p$ cycles, where *p* is the number of PEs. The arity (*k*) of the tree in used in the broadcast network is variable and is chosen so as to maximize system performance.

Logic Unit. The logic unit performs bitwise reduction of integers and flags and supports both the AND and OR functions. The logic unit is implemented as a pipelined tree of OR gates with bypassable inverters before and after the tree. The unit has an initiation rate of one operation per cycle and a latency of lg *p* cycles, where *p* is the number of PEs.

Maximum/Minimum Unit. The maximum/minimum unit supports maximum and minimum reduction of signed and unsigned integers. The previous ASC Processors performed maximum/minimum reductions using the Falkoff algorithm, which processes one bit of the data word each cycle. In order to avoid stalls in the event that multiple threads attempt to perform a maximum or minimum operation at the same time, the multithreaded processor uses a pipelined tree-based structure. Each node in the tree computes the maximum or minimum of its inputs and passes that result on to the next node after one clock cycle. The unit has an initiation rate of one operation per cycle and a latency of lg *p* cycles, where *p* is the number of PEs.

Sum Unit. The sum unit produces the sum of a set of data words located in the PEs. While the ASC model does not require this function, it is used in a number of image and video processing algorithms. If overflow occurs while computing the sum, the result is saturated to the largest or smallest representable value. The sum unit is implemented as a pipelined binary adder tree. The unit has an initiation rate of one operation per cycle and a latency of lg *p* cycles, where *p* is the number of PEs.

Multiple Response Resolver. The multiple response resolver identifies the first responder in a set, and is used to implement sequential and single selection modes of responder resolution. The multiple response resolver is implemented as a pipelined parallel prefix network. The unit has an initiation rate of one operation per cycle and a latency of lg *p* cycles, where *p* is the number of PEs. Unlike the other reduction units, the output of the multiple response resolver is a parallel value.

Response Counter. The response counter counts the number of PEs whose responder bit is set. The ASC model only requires the ability to do a binary count (some/none) of the responders, however this unit goes a step further and produces an exact count of the number of responders. Due to the pipelined implementation, the simpler counter would not have been only faster than the exact one. The response counter is implemented as a pipelined binary adder tree. The unit has an initiation rate of one operation per cycle and a latency of lg *p* cycles, where *p* is the number of PEs.

7. Synthesis Results

The first prototype of the Multithreaded ASC Processor was implemented in VHDL and targeted for an Altera Cyclone II (EP2C35) FPGA. This prototype implements the basic architecture specified in Section 6, though a few features (e.g., multiplier/divider and interthread communication) are still missing.

This first prototype Processor has 16 16-bit PEs, 1 KB of local memory per PE, and supports 16 hardware thread contexts. The entire Processor requires 9,672 logic elements (LEs) and 104 RAM blocks. It operates at a clock speed of approximately 75 MHz on the EP2C35 device.

Table 1 shows the resource usage for each of three main subsystems: control unit, processing element, and broadcast/reduction network. The main factor that limits the number of PEs is the availability of RAM blocks; the critical path that limits the clock speed is the forwarding logic in the PE.

8. Related Work

An FPGA-based SIMD processor is described in [10]. This processor is implemented in a Xilinx Virtex XCV1000E FPGA, has 95 8-bit PEs with 512 bytes of memory per PE, and can operate at a maximum clock speed of 68 MHz. Because the instruction broadcast network is not pipelined, the clock speed is limited by the time is takes to distribute instructions to the PEs. Their processor is larger than our prototype (though our next version will be larger), but is not pipelined or multithreaded.

Another FPGA-based SIMD processor is described in [11]. This processor is implemented in an Altera Stratix EP1S80 FPGA, has 88 8-bit PEs, and can operate at a maximum clock speed of 121 MHz. This processor does use a pipelined instruction broadcast network to improve clock speed. However, it does not pipeline instruction execution, which limits throughput.

9. Future Work

The first prototype of the Multithreaded ASC Processor described in this paper has not yet been fully optimized. Future work will focus on trying to fit more PEs on a FPGA single chip and on improving clock speed. As shown in Table 1, the main factor limiting the number of PEs is the number of RAM blocks. Future versions of the processor may explore alternative PE organizations that require fewer RAM blocks and take advantage of unused logic resources.

Future plans also include implementing software for the architecture in order to better show the performance advantages of multithreading and to explore possible application areas for the architecture.

Component	LEs	RAMs
Control Unit	1,897	8
PE Array (16 PEs)	5,984	96
Network	1,791	0
Total	9,672	104
Available	33,216	105

Table 1. Resource usage for initial processor prototype implemented in EP2C35 FPGA.

10. References

[1] ClearSpeed Technology, "Products Overview," [2006 Dec 18], Available at HTTP: http://www.clearspeed.com/products/overview

[2] WorldScape, "Massively Parallel Computing," [2006 Dec 18], Available at HTTP: http://www.wscapeinc.com/technology.html

[3] James D. Allen and David E. Schimmel, "Issues in the Design of High-Performance SIMD Architectures," *IEEE Transactions on Parallel and and Distributed Systems*, vol. 7, no. 8, Aug., pp. 818–829, 1996.

[4] Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, "ASC: An Associative-Computing Paradigm," *Computer*, vol. 27, no. 11, Nov., pp. 19-25, 1994.

[5] Meiduo Wu, Robert Walker, and Jerry Potter, "Implementing Associative Search and Responder Resolution," in *Proc. International Parallel and Distributed Processing Symposium: Workshop on Massively Parallel Processing*, 2002, p. 246.

[6] Hong Wang and Robert Walker, "Implementing a Scalable ASC Processor," in *Proc. International Parallel and Distributed Processing Symposium: Workshop on Massively Parallel Processing*, 2003, p. 267a.

[7] Hong Wang and Robert Walker, "A Scalable Pipelined Associative SIMD Array with Reconfigurable PE Interconnection Network for Embedded Applications," in *Proc. International Conference on Parallel and Distributed Computing and Systems*, 2005, pp. 667–673.

[8] Anant Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 5, Sept., pp. 525–539, 1992.

[9] Henry M. Levy, Susan J. Eggers, and Dean M. Tullsen, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," in *Proc. International Symposium on Computer Architecture*, 1995, p. 392.

[10] Stanley Y.C. Li, Gap C.K. Cheuk, K.H. Lee, and Philip H.W. Leong, "FPGA-based SIMD Processor," in *Proc. Symposium on Field-Programmable Custom Computing Machines*, 2003, p. 267.

[11] Raymond Hoare, Shenchih Tung, and Katrina Werger, "An 88-Way Multiprocessor within an FPGA with Customizable Instructions," in Proc. *International Parallel and Distributed Processing Symposium: Workshop on Massively Parallel Processing*, 2004, p. 258b.