A SCALABLE PIPELINED ASSOCIATIVE SIMD ARRAY WITH RECONFIGURABLE PE INTERCONNECTION NETWORK FOR EMBEDDED APPLICATIONS

Hong Wang
Kent State University
Computer Science Department
Kent, OH 44242
USA
001-330-672-3123
honwang@cs.kent.edu

Robert A. Walker
Kent State University
Computer Science Department
Kent, OH 44242
USA
001-330-672-9055
walker@cs.kent.edu

ABSTRACT
This paper describes the FPGA implementation of a specialized SIMD processor array for embedded applications. An alternative to traditional SoC or MPSoC architectures, this array combines the massive parallelism inherent in SIMD architectures with the search capabilities of associative computing, producing a SIMD *Processor Array System on a Chip* (PASoC) well suited for applications such as data mining and bioinformatics. This paper first describes the architecture of the system, and then the pipelining of the processing elements and the addition of a reconfigurable interconnection network. Finally, the paper concludes with a brief description of a string-matching algorithm that can be used for the applications cited above.

KEY WORDS
SIMD, Associative Computing, Reconfigurable Network, Embedded Systems, String Matching

1. Introduction
As ASICs and FPGAs continue to grow, a wide range of architectures are being developed to make effective use of the millions (soon to be billions) of transistors available on those chips. System on a Chip (SoC) architectures are increasingly common, and Multiprocessor SoC (MPSoC) architectures [1] combine a tens, possibly hundreds, of powerful processors, a network, and shared memory on a single chip.

This paper describes a third alternative — a SIMD *Processor Array on a Chip* (PASoC). In contrast to the small number of powerful, independent CISC processors supported by a MPSoC, the PASoC supports a large number (hundreds or thousands) of simple RISC processors operating in lock-step SIMD fashion under the direction of a single control unit. While SIMD may have been out of fashion in the 1990s, such SIMD PASoCs are now commercially available [2] and packaged into systems for a variety of applications [3].

These SIMD PASoCs can be further augmented with support for associative computing [4]. *Associative computing* references memory by content rather than address. In its simplest form, each memory cell is associated with a flag bit, and if a search for key data is successful, this bit is flagged, leaving that memory cell to be processed further as appropriate by the algorithm.

An associative SIMD PASoC [4,5] adds associative computing to a SIMD PASoC, assigning a dedicated Processing Element (PE) to each set of memory cells. In associative SIMD computing, PEs search for a key in their local memories, and PEs whose search is successful are designated *responders*. *Masked instructions* can then be used to limit further SIMD processing to only those responders. This SIMD associative computing model has been explored by researchers at Our State University for over 30 years. Referred to as the ASC model [5,6], it is particularly well-suited for applications such as data mining, bioinformatics, image processing, and air traffic control.

We have implemented several prototypes of an FPGA-based associative SIMD PASoC, called the *ASC Processor* [7]. Like a traditional SIMD architecture, the ASC Processor has one Control Unit (CU) and an array of simple PEs, each containing an 8-bit processor and memory. The Control Unit decodes the instructions and broadcasts control signals to the PE array. Even a small million-gate FPGA can implement one CU and around 70 PEs, at clock speeds comparable to other processor cores on FPGAs.

The ASC Processor implements associative operations using a dedicated comparison unit in each PE, a mask stack to indicate when the PE is a responder, and special masked instructions to limit further sequential or parallel processing to only the responders. Additional hardware supports max/min search, giving the PE array the ability to flag as responders those PEs that have a maximum (or minimum) value in a particular memory location.

In any SIMD array, the network design is crucial for PE communication. This network is typically a linear array or

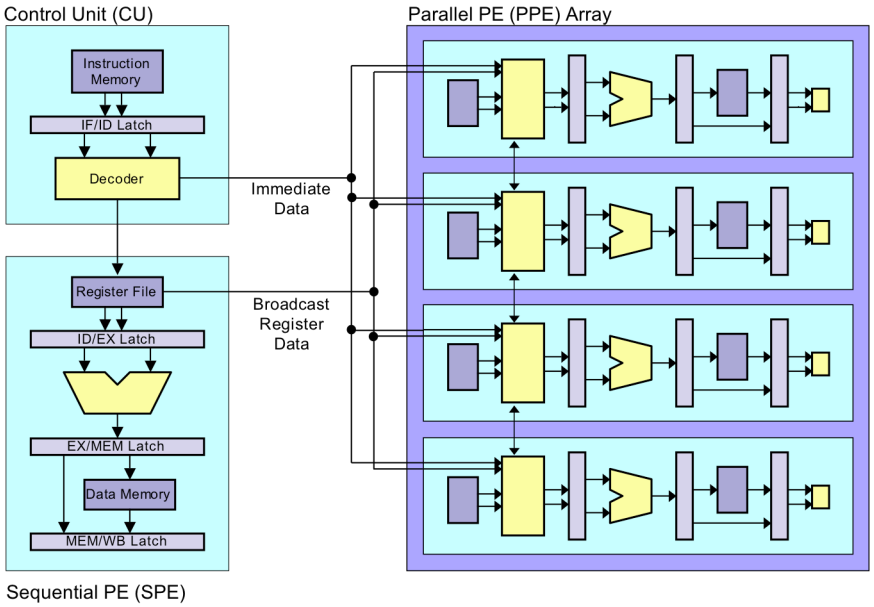

Figure 1. ASC Processor's Pipelined Architecture

mesh, possibly augmented by complicated switches or routers. Herrmann's Asynchronous Reconfigurable Mesh [8] was the first to use a reconfigurable network to connect associative processors, mainly used for image processing. Their system allows processing cells in the PE to reconfigure the mesh network depending on the content of that cell, thus allowing cells to disconnect one or more of its four connections to its east, west, south and north neighbors. This support for reconfiguration allows a group of cells to form a local region of connection. In Section 3 of this paper, we describe another type of reconfigurable network that takes advantage of the associative nature of our system.

Our previous ASC Processor prototypes [7] also have a shortcoming typical of virtually all SIMD architectures — the system speedup comes entirely from increasing the number of PEs, since the PEs lack the pipeline architecture used by most modern processors. While a pipelined architecture has been proposed in SIMD systems [9], there have been no pipelined SIMD processors implemented to date that we are aware of, including the commercial SIMD PASoCs cited above. In Section 2 of this paper, we describe a pipelined associative SIMD PASoC array.

The remainder of this paper will be organized as follows. Section 2 will describe the architecture and functionality of our ASC Processor's pipelined SIMD array. Section 3 will then describe the implementation of a reconfigurable network and its functionality. Each section will also comment briefly on the performance of our new ASC Processor with those additions. Finally, Section 4 will present an application that takes advantage of associative computing and the reconfigurable network.

2. Pipelined SIMD Array

2.1 Pipeline Architecture

There are two main types of pipelining that can be supported by a SIMD system: pipelined broadcast and pipelined PEs [9]. Although broadcast to PEs will be a concern as the number of PEs increases, this paper will assume a relatively small number of PEs (on the order of hundreds of PEs) and concentrate on the pipelining of the PEs, leaving pipelined broadcast for future work. The SIMD array in our ASC Processor, described in this section, has five stages of pipelining: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Data Write Back (WB). Each stage is one clock cycle in length.

The overall structure of the ASC Processor's architecture and pipeline is shown in Figure 1. On the top left side of the figure is the *Control Unit* (CU), which fetches and decodes instructions. Below the Control Unit is a *Sequential PE* (SPE), which executes any scalar instructions, and handles control flow such as branches. On the right side of the figure is the SIMD processor array, consisting of a set of *Parallel PEs* (PPEs), which executes any parallel instructions. Note that this figure emphasizes the pipelined implementation; other architectural details such as the components needed to support associative computing are not shown and have been described in previous papers [7].

The IF stage and part of the ID stage are implemented in the *Control Unit* (CU), shown at the top left of Figure 1. In the IF stage, the Control Unit fetches one instruction from its Instruction Memory, and stores it in the inter-stage IF/ID latch. In the next clock cycle, the ID stage decodes the instruction and sends any immediate data via a data bus and any appropriate control signals (not shown)

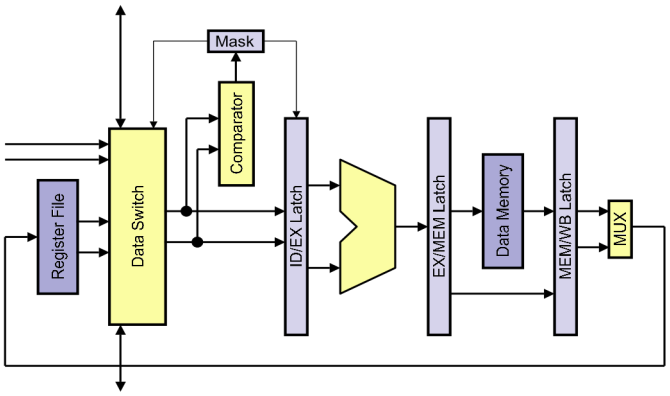

Figure 2. Parallel PE's Pipelined Architecture

via an instruction bus to either the Sequential PE or the Parallel PEs in the PE array. (Recall that in a SIMD system the Control Unit broadcasts hardwired control signals to the PEs, each of which contains only a data path.) The remainder of the ID stage, along with the last three stages, is implemented in the PEs.

The *Sequential PE* (SPE), shown at the lower left of Figure 1, performs scalar computation and handles control flow (e.g., branches). It can also broadcast register data via a data bus to the parallel PE array, for example broadcasting a search key to the PEs for associative search. The SPE has four stages of pipelining: the remainder of the ID stage, the EX stage, the MEM stage, and the WB stage. The last three stages are similar to those in most pipelined sequential processors. The EX stage performs arithmetic and logic operations. The MEM stage contains a 256 x 8 bit data memory, and the WB stage writes back either memory data or ALU data to the register file.

The SPE's portion of the ID stage is more specific to its role in a parallel system. At the front end of the stage is a register file, which stores decoded instructions and data. Most scalar instructions will read data from that register file into the ID/EX latch. However, if the SPE must execute a broadcast instruction, the data is instead placed onto a data bus and broadcast to the parallel PE array. This portion of the ID stage also includes a comparator (not shown) for testing branch addresses.

The parallel PE array is the heart of SIMD parallel computing. In the ASC associative computing model, data are stored in a tabular format across the entire PE array. For example, in a relational database the data are stored in a table, perhaps with each PE storing one row of that table. If there are more rows than PEs, each PE can act as multiple "virtual PEs". For simplicity, we will assume here that we always have enough PE for specific applications, with the understanding that PEs can act as virtual PEs when necessary.

The architecture of the *Parallel PE* (PPE) *array* is shown on the right side of Figure 1. The relationship of the PE array to the Control Unit and SPE has already been discussed. An instruction bus (not shown) allows control signals to be sent from the Control Unit to the PE array, and a data bus allows immediate data and broadcast data to be sent from the CU and SPE, respectively, to the PE array. Each Parallel PE (PPE) in the PE array also includes a Data Switch to facilitate communication between PEs (see Section 3 for more information on the Data Switch and reconfigurable PE network).

The pipelined architecture of a Parallel PE (PPE) is shown in detail in Figure 2, and consists of four pipelined stages. As in the SPE's, the PPE's portion of the ID stage is more specific to its role in a parallel system, this time with an emphasis on associative computing. The PPE's ID stage consists of a 16 x 8 bit register file, a Data Switch, a comparator, and an 8 x 1 bit mask stack. The *mask stack* has two roles: limiting associative computing to only responders, and configuring the interconnection network. In this latter role, the top of the mask stack is sent to the Data Switch and used to configure the interconnection network, as explained later in Section 3.

The mask stack is also used to implement *responder processing* — a fundamental operation in SIMD associative computing. Whenever an associative search is performed, the search key is sent to the PPEs, that key is compared to local data, and the result is pushed onto the top of the mask stack. The top of the mask stack is used to control the ID/EX latch, so if the top of the mask stack is '0', meaning that PPE is not a responder, the instruction will not go through the ID/EX latch.

Using the top of the mask stack in this way, further associative processing can be limited only to responders. If a PPE is a responder, instructions and data move through the ID/EX latch to the EX stage, where ALU arithmetic and logical operations are performed. Then memory access operation can be performed in the MEM stage if necessary, and finally, the WB stage can write network data, ALU results, or data read from memory back to the register file.

In these PPEs, pipeline control hazards are also regulated using the mask bit. If the PE is not a responder, the mask bit will be used to deactivate the ID/EX latch, thus

preventing any following signals or data from going through to the EX or following stages. However, any preceding instructions will still be allowed to finish. Sequential control hazards can be treated in the traditional way; combined sequential-parallel hazards can be dealt with by pushing no-op bubbles through the pipe. Other optimizations, such as register forwarding could also be added, but for now will be left for future work.

2.2 Pipeline Performance

We have implemented the pipelined ASC Processor on an Altera APEX20K1000C FPGA. This FPGA is not particularly large by today's standards, comparable to only one million gates, but sufficient at present for a proof-of-concept. Altera's web site [10] lists various 8-bit processor cores implemented on this particular FPGA and speed grade, with clock speeds ranging from 30 to 106 MHz, typically 60-68 MHz.

Our previous non-pipelined ASC Processor [7] was not tuned for performance, mainly being used to test the implementation of associative SIMD computing, and as such had a clock speed of only 15 MHz. The new pipelined ASC Processor described here was designed more carefully, with shorter critical paths, and as such has a clock speed of 56.4 MHz, comparable with the 8-bit processors cited by Altera. However, with the 5-stage pipeline in our ASC Processor, peak performance can approach 300 MHz, much faster than any of those processors.

As stated above, our ASC Processor is implemented on a million-gate Altera APEX20K1000C FPGA; this FPGA contains 38,4000 Logic Elements (LEs). Without a multiplier, the Control Unit and SPE take up about 200 logical elements (LEs), and each PPE requires about 500 LEs. Thus a one million-gate FPGA of this type can hold 1 Control Unit, 1 SPE, and about 70 PPEs.

3. Reconfigurable PE Interconnection Network

3.1 Need for a Reconfigurable Network

Virtually every parallel system, whether SIMD or MIMD oriented, supports some form of PE interconnection network. A linear array or 2D array (also referred to as a "mesh") is quite common, often with wrap-around at the ends to form a ring or torus. Some systems add row and column broadcast buses. Others add a hypercube for long-distance communication among PEs, or specialized hardware for arbitrary point-to-point communication. As may be expected, there is a tradeoff between added functionality and added hardware.

Associative computing has many advantages, but also has the disadvantage of disrupting any fixed network pattern, since after an associative search the PEs to be processed further are in arbitrary locations in the array. One solution to this problem is to try to avoid using a PE network in associative computing, and it is indeed possible to write many useful associative algorithms [4,5,6] that do not use a network. Another solution is to allow arbitrary PEs to connect to their neighbors to form so-called "coteries" [11], though that solution requires the PEs to be adjacent, which may not be the case after a random associative search.

For our ASC Processor, we have developed a reconfigurable network to support PE communication for associative computing. (Technically these are PPEs, using the implementation-centric terminology of Section 2, but we will revert here to the more common term "PE" for simplicity.) This reconfigurable PE network allows arbitrary PEs in the PE array to be connected via either a linear array (currently implemented) or a 2D mesh (to be implemented soon), without the restriction of physical adjacency. More specifically, each PE in the PE array can choose to stay in the fixed existing network, or opt out of the network so that it is bypassed by any communication. While this solution does not allow the responders to be connected into an arbitrary configuration, it does support communication sufficient for many algorithms.

This reconfigurable PE network will also be crucial for a future version of our ASC Processor, designed for the more powerful *Multiple Instruction Stream ASC* (MASC) model. In MASC, multiple Control Units, referred to as Instruction Streams (ISs), are supported, each of which can control arbitrary PEs / responders in the PE array. To support the MASC model, each IS will require a separate reconfigurable PE network, which can reconfigure dynamically as each IS's responder set changes.

3.2 Reconfigurable Network Architecture

The reconfigurable PE network is currently implemented as a linear array, with support for a 2D mesh planned. Using this network, each PE can directly read its neighbor's register file and store the data into its own ID/EX latch for further processing. For example, consider the following instruction:

Add $left($R1) $R2 $R3

This instruction, executed in lockstep by a set of responder PEs under the direction of the Control Unit, will cause each PE to add the data from its left neighbor's register R1 to the data in its own register R2, and store the result into its own register R3, all in one clock cycle. Similarly, data from the left neighbor could be added to broadcast data and stored in the PE's register file; data from the left and right neighbors could be averaged and stored locally, etc.

To transfer data between these different sources efficiently in the network, a dedicated *Data Switch* unit (see Figure 3) is placed before the ID/EX latch in the PEs (the PPEs, using the terminology of Section 2). For the linear array network currently implemented in our ASC Processor, the Data Switch acts as a network unit,

connecting each PE's Data Switch to its two neighbor's Data Switches.

The Data Switch also implements responder processing — the associative processing functionality described earlier in Section 2.1. If the top of the mask stack is '1', meaning the PE is a responder, the Data Switch routes data from its input ports to its output ports and processes data locally as described below. If the mask bit is '0', however, the Data Switch acts as a bypass, transferring data directly from its left to right neighbors without any local processing. Thus the whole network is reconfigurable, with only responders "connected" to the network and with non-responders being bypassed.

Each Data Switch has six input ports and four output ports, as shown in Figure 3. Two input ports provide local data from the PE's register file. One input port connects via a data bus to the Control Unit's ID stage to provide immediate data. Another input port connects via a data bus to provide register data broadcast by the Sequential PE. The final two input ports provide data from the PE's left and right neighbors.

The Data Switch has four output ports. Two output ports connect to the ID/EX latch to provide data for calculation, and two output ports provide data to the left and right neighboring PEs.

The Data Switch routes data to its output in one of four "modes". In *Computation Mode*, the two input data from the register file are connected to the two output data ports for calculation. In *Broadcast Mode*, immediate data from the Control Unit's ID stage or register data broadcast by the Sequential PE is connected to one of the output data ports, and one of the register file inputs is connected to the other output data port.

In *Data Movement Mode*, there are three options: Left, Right, and Both. In *Data Movement Left* Mode, every PE's left register file input connects to its left neighbor, its right register file input connects to one ID/EX latch output, and data from its right neighbor connects to the other ID/EX latch output. *Data Movement Right Mode* is similar.

Data Movement Both Mode blends the two. Every PE's left register file input connects to its left neighbor, and every PE's right register file input connects to its right neighbor. Further, every PE connects data received from both neighbors to its two ID/EX latch outputs.

Finally, the Data Switch provides one last mode, *Bypass Mode*, to support responder processing for associative computing as described earlier in Section 2.1. This mode is set by the top of the mask stack. If the top of the mask stack is '0', meaning this PE is not a responder, then the left neighbor's output is connected directly to the right neighbor's input, and vice versa. In this way, the PE is bypassed from the network, leaving only responders connected by the network.

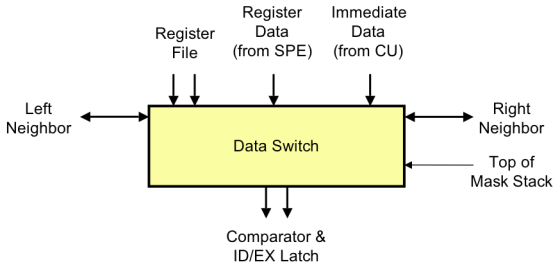

Figure 3. Structure of Data Switch

After moving through the ID/EX latch and leaving the ID stage, data goes on to EX stage, where ALU arithmetic and logical operations are performed, and then on to the MEM and WB stages as described earlier. Recall that the top of the mask stack also controls the ID/EX latch, as described earlier in Section 2.1 — if the PE is not a responder, then data does not pass through the ID/EX latch, limiting further processing only to responders.

3.3 Network Performance

Unfortunately, the performance of our ASC Processor does degrade as the number of PEs is increased with Bypass Mode present due to the long path from the first PE to the last PE in the array. Implemented on a million-gate Altera APEX20K1000C FPGA as described earlier, a 4-PE processor requires 2152 LEs, and runs at 56.4 MHz, comparable to other processors implemented on this FPGA. If we do not include Bypass Mode in the reconfigurable network, the processor frequency remains the same when the number of PEs is increased from 4 to 50 PEs. However, with Bypass Mode added, when the number of PEs is increased to 50, the frequency drops to 22 MHz.

In the future we hope to reduce this delay due to the addition of Bypass Mode by using a pipelined or other multi-hop architecture. For example, we could place intermediate registers every ten PEs in the network. Data being moved would then be stored temporarily in these intermediate registers before moving toward the destination. A MOVE operation would then require several clock cycles to complete, but the overall processor frequency will be improved. Alternatively, we are currently exploring a multi-threaded version of this architecture which will switch to another thread when one stalls due to network or memory delays, again improving overall processor performance even in the presence of Bypass Mode delays.

4. Example: String Matching

A good solution to string matching is useful in a variety of applications, including packet processing, genome processing, etc. The VLDC associative SIMD algorithm [12] can find all instances of a pattern string within a larger text string. Though we discuss only the exact-match version of the algorithm here, extensions support single-character and variable-length "don't cares". Note that this algorithm requires associative search, responder

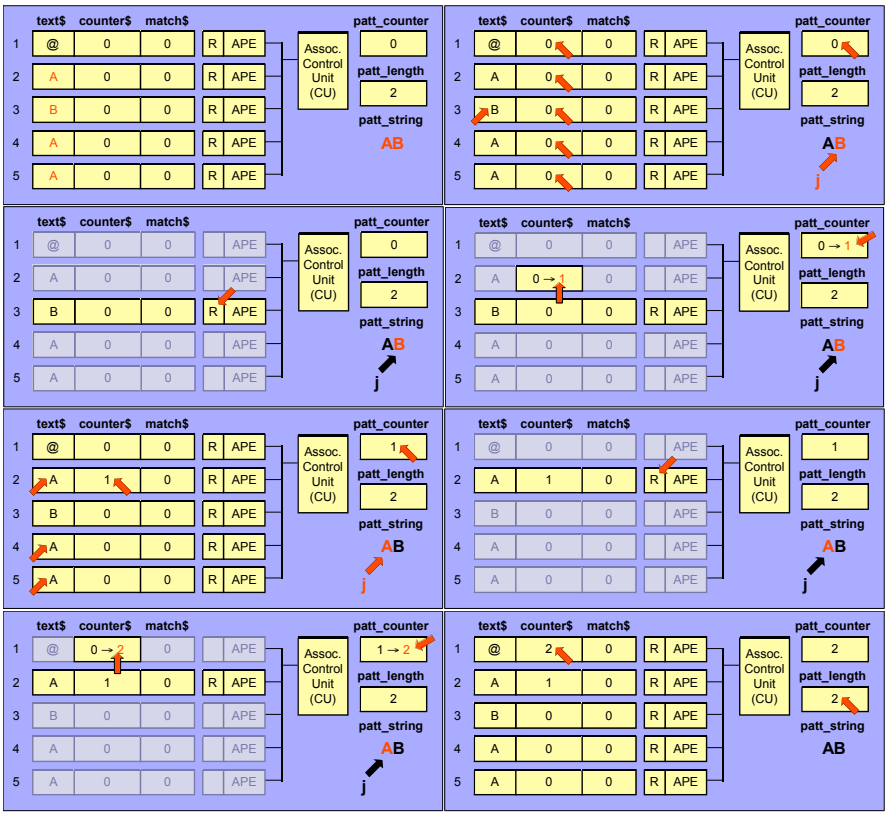

Figure 4. String Matching using Associative Computing

processing, and a linear interconnection network. Although it does not strictly require a reconfigurable network, that functionality might be required by a MASC version that can search multiple text strings simultaneously.

Figure 4 demonstrates the VLDC algorithm by example. In this figure, 5 PEs are shown. "APE" represents an associative PE, and "R" is its responder bit. Figure 4a shows the variables initialized as explained below.

The cells to the left of each PE, with names ending in "$", represent parallel variables stored in the PE's local memory: "text$" stores the text string to be searched ("ABAA" in this example"), "counter$" indicates the number of previous characters matched, and "match$" flags the beginning of any pattern matches.

In addition, the Control Unit stores three scalar variables: "patt_string" is the pattern to be found ("AB" here), "patt_length" is its length, and "patt_counter" indicates the number of characters matched in the pattern.

The algorithm processes the pattern string from right to left (using index variable "j"), beginning with an associative search (see Figure 4b) for a text$ value of "B" and a counter$ value equal to patt_counter, meaning a "B" preceding 0 previously-matched characters. The single successful PE sets its responder bit (see Figure 4c).

Limiting further processing to only that responder, that responder adds 1 to its counter$ value and sends the result of 1 to the counter$ variable of its predecessor (see Figure 4d), using the linear PE interconnection network. This indicates that 1 cell following the predecessor has had a successful match.

The algorithm then processes the next character in the pattern string, performing an associative search (see Figure 4e) for a text$ value of "A" and a counter$ value equal to patt_counter , meaning an "A" preceding 1 previously-matched characters. While there are three "A"s in the text string, only one precedes a "B" — that single successful PE sets its responder bit (see Figure 4f).

Limiting further processing to only that responder, that responder adds 1 to its counter$ value and sends the result of 2 to the counter$ variable of its predecessor (see Figure 4g). This indicates that 2 cells following the predecessor have had a successful match.

Once both characters in patt_string have been searched, the algorithm performs one last associative search (see Figure 4h), looking for a counter$ value equal to patt_length, meaning a cell for which the next 2 preceding cells matched. Although not shown in Figure 4 due to space limitations, the resulting responder sends a 1 to the match$ variable of its successor, using the linear PE interconnection network, to flag the beginning of a successful pattern match.

This algorithm can be modified to run even faster on our pipelined SIMD array, by inserting bubbles before the masked operations to avoid data hazards.

5. Conclusion & Future Work

This paper has presented an alternative to SoCs and MPSoCs — a SIMD Processor Array System on a Chip (PASoC) that combines the massive parallelism of SIMD architectures with the search capabilities of associative computing. This paper describes the implementation of the third generation of our ASC Processor, and the addition of pipelining and a reconfigurable network to that processor — both important steps forward for SIMD systems. Future work will include support for the multiple CUs / ISs of the MASC model, as well as performance improvements to support broadcast to processors with a large number of PEs.

6. References

[1] Ahmed Jerraya and Wayne Wolf, *Multiprocessor Systems-on-Chips*, Morgan Kauffman, 2004.

[2] ClearSpeed Products. http://www.clearspeed.com/products

[3] WorldScape Defense Massively Parallel Computing. http://www.wscapeinc.com/technology.html

[4] Anargyros Krikelis and Charles C. Weems (Eds), *Associative Processing and Processors*, IEEE CS Press, 1997.

[5] Jerry L. Potter, *Associative Computing: A Programming Paradigm for Massively Parallel Computers*, Plenum Press, 1992

[6] Jerry Potter, Johnnie Baker, Stephen Scott, Arvind Bansal, Chokchai Leangsuksun, and Chandra Asthagiri, "ASC: An Associative Computing Paradigm," *IEEE Computer*, 27(11):19-25, November 1994.

[7] Hong Wang and Robert A. Walker, "Implementing a Scalable ASC Processor", *Proc. of the 17th International Parallel and Distributed Processing Symposium (Workshop in Massively Parallel Processing),* abstract on p. 267, full text on accompanying CDROM, April 2003.

[8] Frederick P. Herrmann and Charles G. Sodini, "A Dynamic Associative Processor For Machine Vision Applications", *IEEE MICRO*, 12(3):31-41, June 1992.

[9] James D. Allen and David E. Schimmel, "The Impact of Pipelining on SIMD Architectures", *Proc. of the 9th International Parallel Processing Symposium*, pp. 380-387, April 1995.

[10] Altera IP MegaStore. http://www.altera.com/products/ip/processors/ipm-index.jsp

[11] Martin C. Herbordt, Charles C. Weems, and Michael J. Scudder, "Nonuniform Region Processing on SIMD Arrays using the Coterie Network", *Machine Vision and Applicatons*, 5(2):105-125, March 1992.

[12] Mary C. Esenwein and Johnnie W. Baker, "VLDC String Matching for Associative Computing and Multiple Broadcast Mesh", *Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems*, pages 69-74, 1997