Implementing Associative Processing: Rethinking Earlier Architectural Decisions

Robert A. Walker, Jerry Potter, Yanping Wang, and Meiduo Wu

Kent State University
Mathematics and Computer Science Department
Kent, OH 44242
{walker, potter, yawang, mwu}@mcs.kent.edu

Abstract

This paper describes an initial design of an associative processor for implementation using field-programmable logic devices (FPLDs). The processor is based loosely on earlier work on the STARAN computer, but updated to reflect modern design practices. We also draw on a large body of research at Kent State on the ASC and MASC models of associative processing, and take advantage of an existing compiler for the ASC model. The resulting design consists of an associative array of 8-bit RISC Processing Elements (PEs), operating in byte-serial fashion under the control of an Instruction Stream (IS) Control Unit that can execute assembly language code produced by a machine-specific back-end compiler.

1 Introduction — the KSU MASC model

At Kent State University (KSU), research has long focused on a unique variant of data parallel processing known as *associative processing*. Associative processors provide the capabilities of massive associative memories — the ability to access memory by content rather than address — without the high cost associated with "real" associative memories. Such associative processors are particularly well suited for problems that involve searching through and processing massive amounts of data, such as relational database management, data mining, air traffic control, image processing, and graphics.

The bit-matching circuitry of associative memories is expensive, but *associative processors* are much more cost effective. An associative processor shares Processing Elements (PEs) among the memory, allowing thousands of memory words to be examined in parallel. A global set of *mask bits* selects the memory words to be examined, program logic selects the fields within the words to be compared, and *responder bits* indicate successful matches. A control unit broadcasts the same instruction to each PE, but based on the results of previous searches, a particular PE can decide whether or not to perform that instruction. For example, a search could be performed to locate all burgundy Ford Focus cars in Ohio on a dealer's lot, and then limit further processing to only those cars. Associative computing is discussed in detail in [Potter92] and [Krikelis97].

Developed at Kent State, the *MASC* (*Multiple ASsociative Computing*) model of associative computing grew out of work on the STARAN and MPP computers at Goodyear Aerospace Corporation. In the MASC model, shown conceptually in Figure 1, the associative processing array consists of a set of cells, each containing a PE and a local memory. A tabular data structure is stored in the array as one record per memory, which means the PEs can simultaneously perform various arithmetic, logical, or comparison operations on a particular field in their own local memories in parallel.

In the current MASC model, there are one or more different Instruction Stream (IS) Control Units (but much less than the number of PEs). Each IS contains a copy of the program it is to execute, and can broadcast an instruction to all cells. Each cell listens to only one IS—initially all cells listen to the same IS, but the cells can switch to other ISs in response to their local data and commands from the current IS. An active PE conditionally executes the commands it receives from its IS based on the contents of its mask bits, while an inactive PE simply listens to the commands of its IS until it is reactivated. (A single-IS version of the MASC model is described in [Potter92,] and will be referred to here as the ASC model when the two need to be distinguished.)

The MASC model supports *associative processing* by providing associative searching and selection, logical operations (AND and OR), maximum, minimum, least upper bound, and greatest lower bound. Searching permits

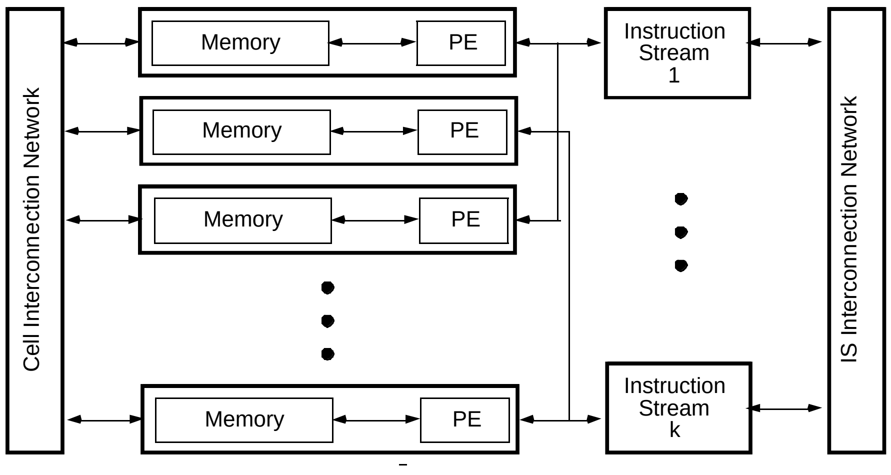

Figure 1 – A Conceptual View of the KSU MASC Model

the simultaneous examination and identification of all those cells that meet the search criteria. The identified cells, called *responders*, can then have their mask bits set to become the new set of active cells. An IS has the ability to detect the presence of responders, send instructions to active cells in parallel or sequentially, and restore previously active cells to the set of idle cells. The maximum (minimum) functions retrieve either the greatest (least) value in a particular field or the index of the PE containing that value.

2 The ASC processor — implementing the ASC model

We are currently re-evaluating associative computing in terms of today's implementation technology. More specifically, the single-instruction stream ASC model is being implemented; later on the focus will be shifted to the multiple-instruction stream MASC model. During this implementation, some ideas from the STARAN and earlier processors have been adopted. However, the architecture has been completely reinvented to accommodate advances in implementation technology (the STARAN was implemented in TTL). Our initial target implementation is on field-programmable logic devices (FPLDs) in the Altera FLEX family; later implementations may be on other FPLDs or even on custom ASICs.

The remainder of this paper is organized as follows. This section presents the overall structure of our associative processor and the major design decisions made. Section 3 describes the architecture of the IS Control Unit for the associative processing array. Section 4 then elucidates the architecture of Processing Elements, along with the circuitry for handling responders and searching for minimum / maximum values. As of this writing (January 2001) the remaining details of this architecture are being defined by test sample programs against the architecture, and coding the architecture in VHDL is in progress.

2.1 Targeting the Altera FLEX 10K20 for the initial implementation

Our initial design will use Altera's MAX+PLUS II design environment to implement a small ASC processor on Altera FLEX 10K20 field-programmable logic devices (FPLDs). This initial prototype will contain a small array of 4 PEs and a single IS control unit. Since this design has very limited PEs and a single instruction stream, the cell and IS interconnection networks shown in Figure 1 will not be implemented yet. The current implementation focuses only on basic ASC functions, such as the broadcast and reduction operations, conditional searching, maximum and minimum field searching, and responder iterations that operate on a small database management application. Although this design is too small to be of great use, it will enable us to test our architecture. However, our ultimate goal is to use a much larger number of more powerful chips, and to implement the more advanced features of the full MACS model.

There are a number of boards from Altera available in our VLSI Design lab; each of these boards includes one Altera FLEX 10K20 FPLD. According to Altera's 10K data sheet [AlteraDoc1], the FLEX 10K20 contains 20,000 logic gates, grouped into 144 Logic Array blocks, each of which contains 8 Logic Elements. Each logic element, in turn, contains a 4-input look-up table and a programmable flip-flop. The FLEX 10K20 also contains 6 Embedded Array Blocks (EAB) that can be used as internal on-chip fast memory — each EAB provides 2048 bits of RAM that can be configured in 256*8, 512*4,

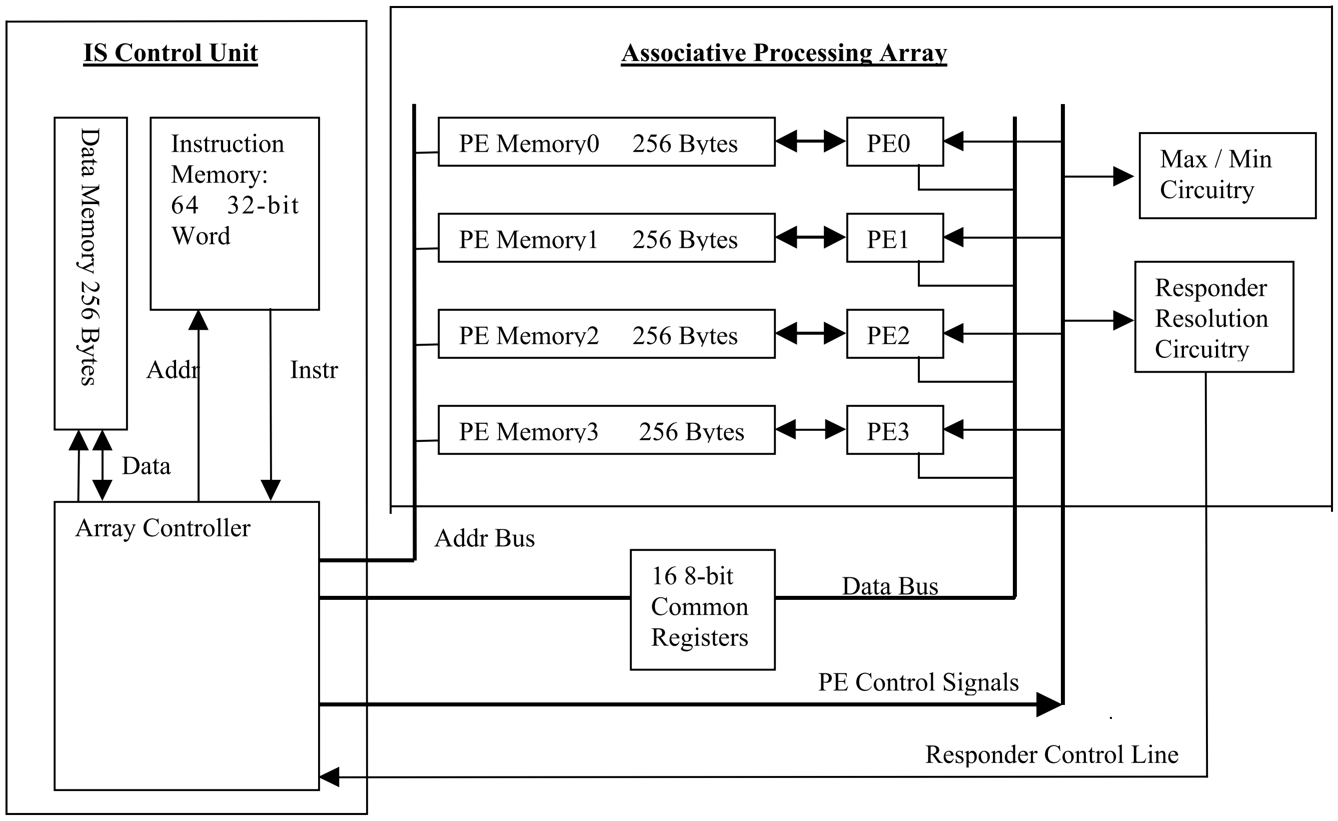

Figure 2 – ASC Processor (Small Initial Prototype)

1024*2, or 2048*1 bits, with an access latency of about 20-50ns.

Based on the characteristics of the FLEX 10K20 FPLD and its limited size, our initial design implements only one IS Control Unit and four PEs in the Associative Processing Array. Since there are 6 EABs available on the chip, each PE uses one EAB as its Data Memory, which is configured as 256 bytes. The two remaining EABs are employed as the Instruction Memory and the Data Memory in the control unit; one is configured as 64 32-bit words (the ASC Control Unit has 32-bit instructions), and the other is configured the same as the PE's Data Memory (which has 256 bytes).

2.2 ASC processor design overview

Having examined the STARAN's design and the features of our FLEX 10K20 target chip, we decided to implement an 8-bit (byte serial) associative processor instead of a bit serial processor like the STARAN. In our implementation, each PE uses a separate EAB as its Data Memory, and our initial experiments seem to indicate that there are sufficient Logic Array blocks in proportion to each EAB to implement a full 8-bit PE for each memory. Each EAB can easily be configured and accessed as an 8-bit memory. Other implications of this choice to implement a byte-serial processor are discussed over the course of this paper.

An overview of our small initial prototype of the ASC processor is shown in Figure 2. It includes a single IS Control Unit, and an Associative Processing Array containing 4 PEs, a set of Common Registers, and some support circuitry.

The IS *Control Unit* processes 32-bit ASC instructions that are produced by a machine-specific back-end compiler that translates the output of the existing ASC compiler into our ASC processor's machine code. Blocks of instructions are stored in its Instruction Memory, and the Control Unit fetches, and decodes those instructions. After decoding, it executes the sequential operations directly, and sends control signals to the PEs in the Associative Processing Array to execute the parallel instructions. During this process, address translation and corner turning are provided as necessary by the Array Controller so that each PE can properly access its Data Memory.

Since the Associative Processing Array is byte-serial, the Control Unit must loop to process any field that is longer

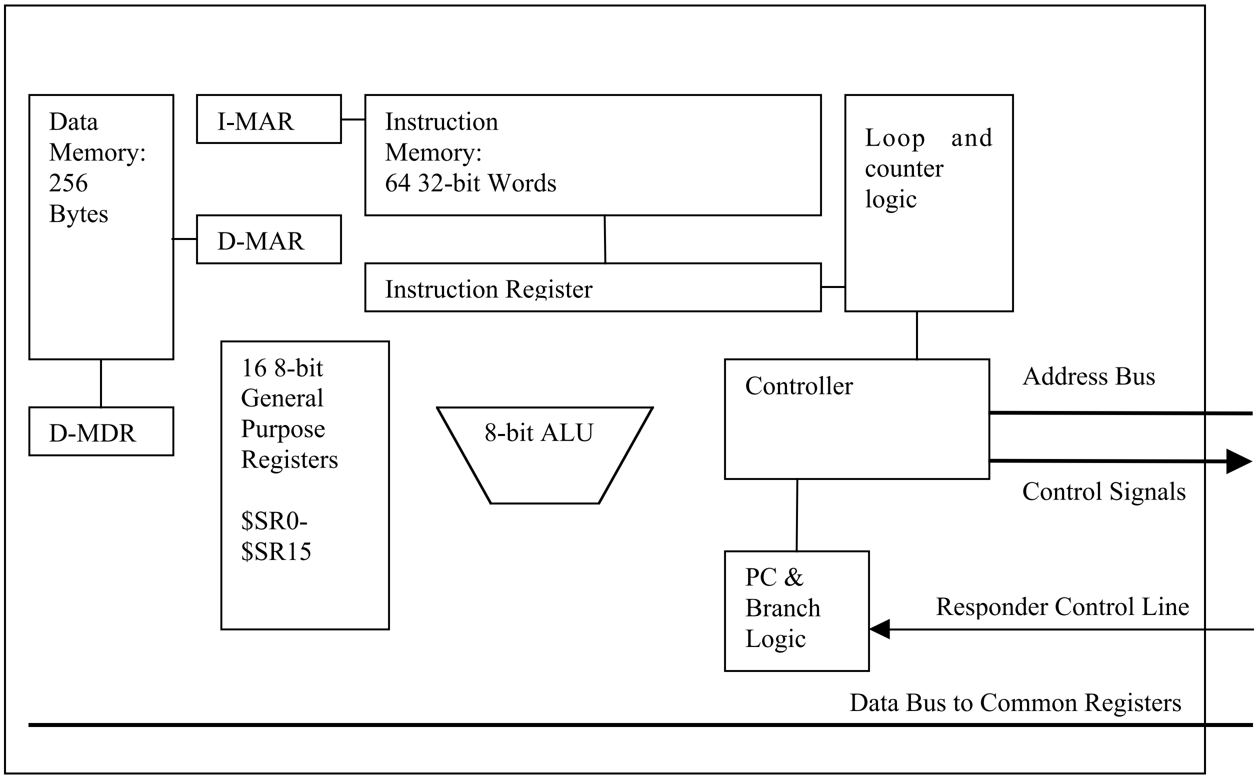

Figure 3 – ASC IS Control Unit

than 8 bits. For example, when processing an integer addition of 32 bits, the Control Unit must execute the addition instruction four times and propagate carry information between bytes. When appropriate, the Control Unit also detects active responders (through the responder control line set by the PE array) and processes them in serially or in parallel. The Control Unit is described in more detail in Section 3.

There are sixteen 8-bit *Common Registers* ($CR0 to $CR15), which serve as intermediate data storage when transferring data between the Control Unit and the Associative Processing Array, and are used for broadcasting and reduction. Since they are common to the Control Unit and to the PEs in the Associative Processing Array, both the Control Unit and the PEs can use the Common Registers as a source of instruction operands.

The PEs in the Associative Processing Array operate in SIMD fashion, with each PE receiving control signals from the Control Unit. Those PEs whose mask bit is true will execute the instructions sent by Control Unit, while the others will remain idle. The array also contains circuitry to detect the existence of responders, and min/max circuitry to perform minimum / maximum value searches. The Associative Processing Array is described in more detail in Section 4.

3 The IS control unit

The IS Control Unit, shown in Figure 3, contains an Instruction Memory, an Instruction Register, a Data Memory, sixteen 8-bit General Purpose Registers, an 8-bit ALU, and assorted logic. This section briefly describes the architecture of the Control Unit and the instructions it supports.

3.1 Control unit architecture

The Control Unit, sketched in Figure 3, contains several components. The major components and their main functionality are as follows:

- ALU: provides 8-bit arithmetic and logic operations as well as comparison operations, and can also be used as a part of global reduction operations such as summation.
- General Purpose Registers: a bank of sixteen 8-bit General-Purpose Registers ($SR0 - $SR15) to provide workspace for arithmetic and data transfer operations.

Table 1 — Machine Instruction Set

Arithmetic Instructions

INSTRUCTION	EXAMPLE	MEANING
Add	ADD $GR1, $CR1, $GR2;	$GR1+ $CR1 = $GR2
Subtract	SUB $GR1, $GR2, $GR3;	$GR1 - $GR2 = $GR3
Multiply	MUL $GR1, $GR2, $GR3;	$GR1 * $GR2 = $GR3
Arithmetic Shift	AS $GR1, $GR2, $GR3	$GR1 * 2^{$GR2} = $GR3

Logical Instructions

INSTRUCTION	EXAMPLE	MEANING
And	AND $LR1, $LR2, $LR3;	$LR1 & $LR2 = $LR3
Or	OR $LR1, $LR2, $LR3;	$LR1 | $LR2 = $LR3
Xor	OR $LR1, $LR2, $LR3;	$LR1 | $LR2 = $LR3
Not	NOT $LR1, $LR2;	! $LR1 = $LR2

Compare Instructions

INSTRUCTION	EXAMPLE	MEANING
Set if greater than	SGT $GR1, $CR1, $LR1;	if $GR1 > $CR1, $LR1=1; else $LR1=0.
Set if greater equal	SGE $GR1, $CR1, $LR1;	if $GR1 >= $CR1, $LR1=1; else $LR1=0.
Set if less than	SLT $GR1, $CR1, $LR1;	if $GR1 < $CR1, $LR1=1; else $LR1=0.
Set if less equal	SLE $GR1, $CR1, $LR1;	if $GR1 <= $CR1, $LR1=1; else $LR1=0.
Set if equal	SEQ $GR1, $CR1, $LR1;	if $GR1== $CR1, $LR1=1; else $LR1=0.
Set if not equal	SNE $GR1, $CR1, $LR1;	if $GR1!= $CR1, $LR1=1; else $LR1=0.

Mask Stack Instructions

INSTRUCTION	EXAMPLE	MEANING
Copy top of mask stack	TOPMSK $LR1;	Copy top of the mask stack to $LR1.
Pop the top of mask stack	POPMSK $LR1;	Pop the top of mask stack to $LR1.
Replace top of mask stack	REPLACEMSK $LR1;	Replace top of stack with $LR1.
Push into mask stack	PUSHMSK $LR1;	Push $LR1 into mask stack.
Push to mask stack and them	PUSHMSKTHEM $LR1;	Push $LR1 to mask stack and "Them" register.

Data Transfer Instructions

INSTRUCTION	EXAMPLE	MEANING
Load a byte from memory to GPR	LD 0X2F, $GR1;	Load content of (0X2F) to $GR1.
Store byte from register to memory	ST $GR1, 0X05;	Store $GR1 to memory address (0X05).
Load register to register	LDRR $GR1, $CR1;	Move data from $GR1to $CR1
Load Immediate value to register	LDI ImmVal, $CR0;	Load immediate value to common register $CR0

- Buses: a unidirectional 8-bit address bus that connects to each PE memory, and an 8-bit bi-directional data bus that connects to the Common Registers.

Table 1 — Machine Instruction Set (cont.)

Reduction Instructions

INSTRUCTION	EXAMPLE	MEANING
Select a responder	FIND $LR1;	According to the value of $LR1, find a responder from a group without clear their responder registers.
Select a responder and clear it	STEP $LR1;	According to the value of $LR1, find a responder from a group, and clear the responder's $LR1.
Select a responder and clear all others	RESVFST $LR1;	According to the value of $LR1, find a responder from a group, and clear all others $LR1.
A step for searching minimum	MIN $GR1, $LR1;	Compare one bit of $GR1 at a time from a selected group in searching for the minimum field, store result in $LR1.
A step for searching maximum	MAX $GR1, $LR1;	Compare one bit of $GR1 at a time from a selected group in searching for the minimum field. Store result in $LR1.

- Loop & Count Logic: used to process data fields longer than one byte, as described in Section 2.2.
- PC & Branch Logic: controls program counter and provides branches around parts of the instruction stream if there are no responders.
- Controller Logic: coordinates the Loop & Count Logic and the PC & Branch Logic and sends addresses and control signals to the PEs in the Associative Processing Array.

3.2 Machine instruction set

The IS Control Unit is designed to process assembly language instructions produced by our newly developed back-end compiler, which takes advantages of an existing ASC compiler. Although the ASC processor has never been implemented, our existing ASC compiler was supported on the STARAN, CM-2, and Wavetracer, and simulators have been designed to run on a variety of SISD machines. As a result, the back-end compiler takes the output of our existing compiler's intermediate code, and translates that into our ASC processor's machine code that is based on our specific architecture. More detailed information about this back-end compiler can be found in [Wang01].

There are two categories of instructions supported by our ASC processor: masked instructions and unmasked instructions. Masked instructions are executed only by those PEs that have a "1" on the top of their mask stack, while unmasked instructions are executed by all PEs. In each category, there are arithmetic instructions, logical instructions, compare instructions, mask stack instructions, data transfer instructions, and reduction instructions, as shown in Table 1.

3.3 Comparison to STARAN

The ASC Control Unit is roughly similar to that of STARAN, in that it processes a similar instruction set (although a number of simplifications were made). At the moment, we have a small Control Unit with a limited Instruction Memory and Data Memory instead of a full host controller, but that smaller Control Unit should work well when we move to the Multiple-Instruction-Stream MASC model. The other major difference between our ASC processor and STARAN is that STARAN was bit-serial, whereas our ASC processor is byte-serial, which gives us some additional processing power.

4 The associative processing array

The Associative Processing Array consists of an array of Processing Element (PE) cells, along with min/max circuitry and responder circuitry as shown in Figure 2. Each PE cell consists of a PE and a local memory. The local memory usually stores one record out of a tabular data structure, where a record usually consists of several variable-size fields. The PEs can thus simultaneously perform various arithmetic, logical, or comparison

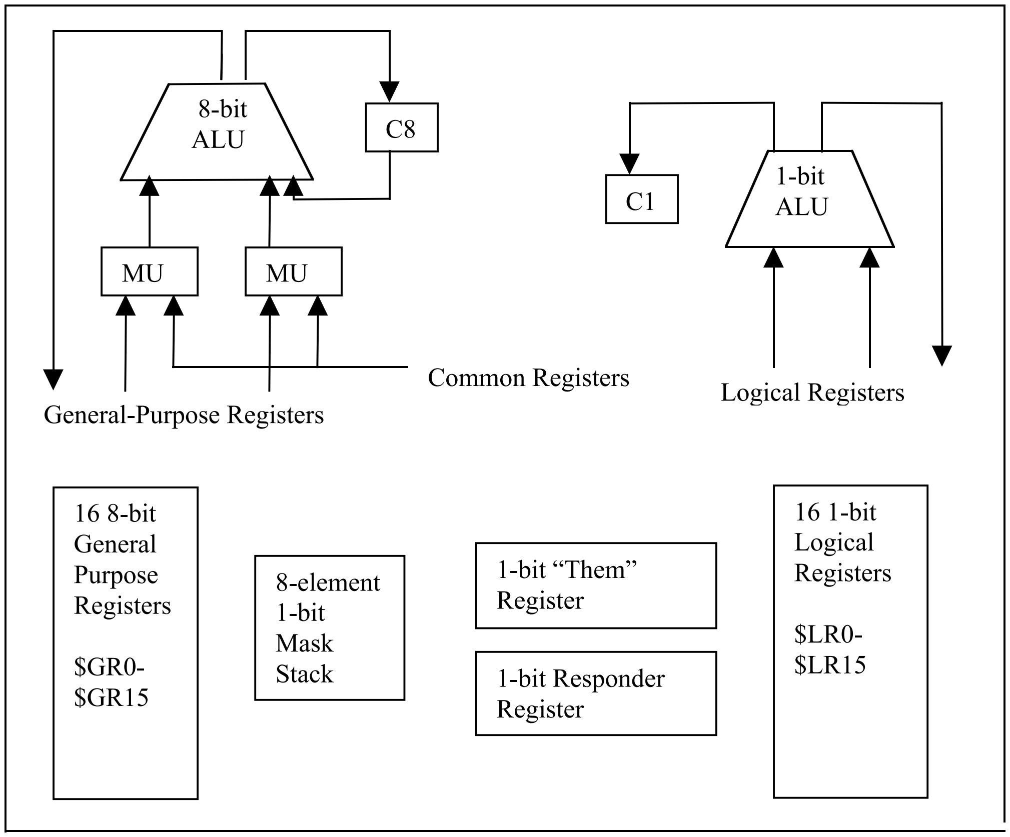
8-bit ALU
C8
C1
1-bit ALU
MU
MU
Common Registers
Logical Registers
General-Purpose Registers
16 8-bit General Purpose Registers $GR0- $GR15
8-element 1-bit Mask Stack
1-bit "Them" Register
1-bit Responder Register
16 1-bit Logical Registers $LR0- $LR15

Figure 4 – One PE in the ASC Associative Processing Array

operations on those fields in their own local memory under the control of the IS Control Unit.

4.1 PE architecture

The architecture of one PE in the Associative Processing Array is shown in Figure 4. This PE is composed of an 8-bit ALU, a 1-bit ALU, sixteen 8-bit General-Purpose Registers, sixteen 1-bit Logical Registers, a mask stack, a special purpose 1-bit "Them" register, and a 1-bit responder register.

The 8-bit ALU and the sixteen 8-bit *General-Purpose Registers* ($GP0 to $GP15) are primarily used for byte-serial arithmetic and comparison operations. The operands for the 8-bit ALU come from either the General-Purpose Registers or from the Common Registers, and the output from the ALU goes to one of the General-Purpose Registers or to the Common Registers. Data transfer instructions permit memory access.

If operands are larger than one byte, the Control Unit must process each byte in turn. For instance, an addition of two 4-byte integers is processed by four 1-byte additions from the least significant byte to the most significant byte. Each time, the carry out from one byte is stored in the C8 register, and the sum is stored in one of the General Purpose Registers.

The 1-bit ALU and the sixteen 1-bit *Logical Registers* ($LR0 to $LR15) are used to perform logical operations, and in conjunction with the responder circuitry (the *responder register* and the *mask stack*) are used to support associative processing. To perform an associative search operation, a broadcast value (stored in a Common Register) is compared with the corresponding data stored in the local memory of each PE (after loading it into a General Purpose Register). For example, it may be necessary to search for those cars whose color is "burgundy" and are located on a dealer's lot in "Ohio". Those two comparisons are performed, and each result is stored temporarily in a Logical Register. After both comparisons have been performed, the results are logically ANDed together to produce the final result, which is "1" if the both comparison searches are successful and otherwise it is "0". This result is then used to set the responder register, and is pushed into mask stack. In certain special cases, it is necessary to store this result for later use; if so, the result can be pushed into the mask stack and also be stored into the "Them" register using the "PUSHMSKTHEM" instruction .

The mask stack is particularly useful when multiple levels of association exist. In ASC, the IF-THEN statement might include several nested conditions, which would incur multiple levels of association. Using the mask stack, we can distinguish these different association levels efficiently. The mask stack in a PE, which can contain up to 8 1-bit logical values, is capable of indicating up to 8 levels of association groups (assume the nested conditions never exceed 8 levels). The farther the stack grows, the more restricted association group it indicates.

When a program begins, the first item of the stack is initialized to 1 for all memory cells. This means that all cells are in the active group at the top level. A new active association group in the mask stack can be set up by IF statement in ASC. The top of the mask stack always indicates the current association group. During the THEN statement-block, all of the current responders (which are indicated by the responder bits) are processed either in parallel or serially, but is possible that the responder bit may change afterward. For example, the responder bits will be cleared one by one during serial processing of the responders. However, the current association group is kept in the stack until the corresponding ENDIF is encountered, then the top of mask stack is popped. Before processing the next association group, the responder bit will be updated using the top value of the mask stack. So with the mask stack, it is efficient to perform as many operations as desired on an association group.

4.2 The resolution and min / max circuitry

In addition to the PE cells, the Associative Processing Array includes the resolution circuitry and min/max circuitry as shown in Figure 2. The responder circuitry is used by the Control Unit to process responders, and the min/max circuitry is used to find the minimum or maximum value of the specified field among the responders.

The responder resolution circuitry gives the Control Unit a signal to indicate whether or not there is at least one successful responder after searching. If there are multiple responders, they will be processed by the Control Unit either serially or in parallel. If serial processing is needed, the responder resolution circuitry is responsible for selecting the successful responders in turn for the Control Unit to process, and once a responder has been processed, for clearing the corresponding responder bit. After all the responders are processed, the signal provided by the responder resolution circuitry to the Control Unit will indicate that there is no responder.

The min/max circuitry searches for the largest and smallest value in a field among all active responders. The general idea is to use bit slices as masks for the extreme value. It is performed in parallel as follows:

1. Searching the bit slices from most significant bit to least significant bit. As each bit slice is processed, it is logically ANDed with the corresponding logical register indicated by instruction stream, and the result is stored back to that register. This operation is shown in Table 1 under the Reduction Instruction category. A 1-bit MAX register is used to indicate whether the data in the specified field is the maximum.
2. Checking all the results in that logical register in parallel to ensure that at least one new maximum value remains (at least one result is 1). If it is true, then the MAX bit is updated; if all the results are 0, then all current entries considered have the same bit value 0 and remain tied (the MAX bit is not updated at this time).
3. Continue to process the remaining bit slices as above until all are processed.
4. Once all bit slices have been processed, if only one MAX bit is 1, it marks the largest number; if more than one MAX bit is 1, those cells are tied for the maximum.

The minimum value can be obtained in a similar way, but we use the MIN bit instead of the MAX bit, and logically NOT the bit slices first, then logically AND the result with the logical register.

4.3 Comparison to STARAN

The ASC Processing Array is very different from that in STARAN, and is essentially a totally new design. The processors in ASC, which work in byte-serial fashion, have a RISC load / store architecture and instruction set of our own design, based roughly on a "typical" modern RISC processor. Furthermore, unlike STARAN, our ASC processor has General-Purpose Registers in each PE, and a stack of mask bits instead of a single mask bit. Finally, each PE has its own local memory, instead of sharing a common memory, which obviates the need for the complex corner-turning circuitry that STARAN needed for efficient memory access.

5 Conclusions and future work

At this point, we have completed a preliminary paper design of the ASC processor. We have also developed a back-end machine-specific compiler. An initial VHDL version of the array controller and some parts of PE are implemented on the Altera FLEX 10K20 FPLD. Over the next few months, we expect to code the whole design in

VHDL, simulate to verify its functionality on a small database application, and evaluate how well it fits on the Altera 10K FPLD chip groups. Then we will purchase PCI boards with larger FPLD and with 32MB or more of SDRAM, and construct a larger associative processor. Eventually, we expect to implement the full multiple-instruction-stream MASC model.

6 Acknowledgements

The authors thank Professor Kenneth Batcher and other members of the Parallel Processing Group, for many helpful discussions.

7 References

[AlteraDoc1] Altera 10K data sheet http://www.altera.com/document/ds/dsf10k.pdf

[Batcher74] K. E. Batcher, "*STARAN Parallel Processor System Hardware*", 1974 National Computer Conference, AFIPS Conference Proceedings, vol.43, pp. 405-410.

[Batcher80] K. E. Batcher, *"Architecture of A Massively Parallel Processor"*, IEEE Computer Society, 1980.

[Krikelis97] A. Krikelis and C.C. Weems, *Associative Processing and Processors*, IEEE Computer Society, 1997.

[Potter92] J.L. Potter, *Associative Computing*, Plenum Publishing, New York, 1992.

[Potter94] J.L. Potter, J. Baker, S. Scott, A. Bansal, C. Leangsuksun, and C. Asthagiri, *"ASC: An Associative Computing Paradigm,"* IEEE Computer, November 1994, pp. 19-26.

[Staran77_1] Goodyear Aerospace Corporation, "*STARAN Reference Manual*", November 1977.

[Staran77_2] Goodyear Aerospace Corporation, "*STARAN Programming Manual*", November 1977.

[Wang01] Y. Wang, Design Documentation, "A Machine-specific Back-end Parallel Compiler Design – Target ASC Machine Architecture, Assembling Language, and Binary Machine Code Format",, http://www.mcs.kent.edu/~yawang/ASC-compiler.html