The CAM2000 Chip Architecture[1]

D. Smith J. Hall K. Miyake
Laboratory for Computer Science Research
Department of Computer Science
Rutgers University

January 28, 1994

[1]This works was supported by the Defense Advanced Projects Agency and the National Aeronautics and Space Administration under NASA-Ames Research Center grant NAG 2-668

Abstract

Effective use of a processor requires the delivery of instructions and data at a sufficiently high rate to prevent stalls. Consequently, memories must be both large and fast. Current technology trends show that within the next few years the processor/memory interface will become a serious bottleneck that severely limits system performance.

Though DRAM speed will hold the processor/memory data bandwidth below acceptable levels it is possible to significantly increase the processor/memory information bandwidth. This can be done by increasing the *quality* of the data passed between memory and processor. The concept of increased data *quality* is the basis for the Rutgers CAM2000 design.

The CAM2000 architecture is a tree connected state machine consisting of four tightly coupled components; tree, leaf, memory, and I/O. It combines features of Associative Processing(AP), Content Addressable Memory (CAM), and Dynamic Random Access Memory (DRAM) in a single chip package that is not only DRAM compatible but capable of applying *simple* massively parallel operations to memory.

Contents

List of Figures

List of Tables

1 Introduction

This report described the architecture and instruction set of the Rutgers CAM2000 memory chip. The CAM2000 combines features of Associative Processing(AP), Content Addressable Memory (CAM), and Dynamic Random Access Memory (DRAM) in a single chip package that is not only DRAM compatible but capable of applying *simple* massively parallel operations to memory.

This document reflects the current status of the CAM2000 architecture and is continually updated to reflect the current state of the architecture and instruction set.

1.1 Processor/Memory performance gap

Ideally a processor should be supplied with instructions and data at a rate sufficiently high to allow the processor to execute without pause. In order to meet the demands of the processor, memories must be large enough to contain the necessary information and fast enough to deliver it as required. Memories technologies have been developed that support either the size (e.g., DRAM) *or* the speed (e.g., SRAM) demanded by processors but there is no available technology that supports memories that are *both* large enough and fast enough to satisfy processor demands.

The need for large, fast memories has been circumvented by cleverly designed memory systems and hierarchies that employ interleaved memories, static column DRAM[Hennessy and Patterson, 199 and multiple level caches[Hennessy and Patterson, 1990]. These systems rely on the local nature of information usage particular to the VonNeumann architecture to provide a memory system that performs as though it were a single large, fast memory. They are implemented hierarchically using smaller and faster *caches* near the processor and larger slower memories near the *main memory*. The performance of such memory systems has been studied in detail over the last 10 years[Hennessy and Patterson, 1990] with the results indicating that *typical* programs produce memory access patterns with a high degree of clustering. Memory hierarchies designed to take advantage of this clustering have been able to hide the latency of DRAM; however, current technology trends are making it unlikely that such memory systems will be able to keep pace with the data rates required by the next generation of processors. Table 1 summarizes the current technology trends that highlight this anticipated bottleneck.

- Processor speed is growing 75% per year
- Memory size is growing 60% per year
- Memory speed is growing 7% per year

Table 1: Technology trends for Processor and Memory Growth

As seen from Table 1, DRAM size is growing at a rate comparable to processor speed; however, DRAM speed is growing at only one tenth this rate. Consequently, although stan-

dard memory systems will scale up in size as required, they will not scale up in speed and the resulting lack of processor/memory data bandwidth will become a serious bottleneck that limits system speed. However, even though DRAM speed will hold the processor/memory data bandwidth below acceptable levels it is possible to significantly increase the processor/memory information bandwidth. This can be done by increasing the *quality* of the information passed between memory and processor while maintaining the same data bandwidth.

The concept of increased information *quality* is the basis for the Rutgers CAM2000 design. The design places many small processors on the memory chip allowing it to perform simple massively parallel computations on the memory contents before forwarding the results to the processor. For example, consider finding the average of k numbers using DRAM with a cache-based processor. Since the addition of the k numbers can only be done in the processor, all k numbers must be sent through the processor/memory bottleneck creating a heavy load on this critical resource. If CAM2000 memory is used, the sum of the numbers could be computed in the memory and then passed to the processor. In this case only one number flows through the bottleneck: the *quality* of information has been increased and the load on the critical processor/memory bottleneck has been reduced.

The CAM2000 architecture is a tree connected state machine consisting of four tightly coupled components; tree, leaf, memory, and I/O. The tree component is composed of tree cells organized as a binary tree. It performs global operations over data contained in the leaf cells. In addition it provides a data path from each leaf cell to the processor's memory bus. The leaf component is composed of leaf cells that contain one leaf processor, a bank of local registers, one partition of the on-chip memory, and one register of a parallel shifter that forms the I/O component. The leaf cells perform local operations on that cell's memory partition and register set. A four leaf cell example of the CAM2000 architecture is shown in Figure 1.

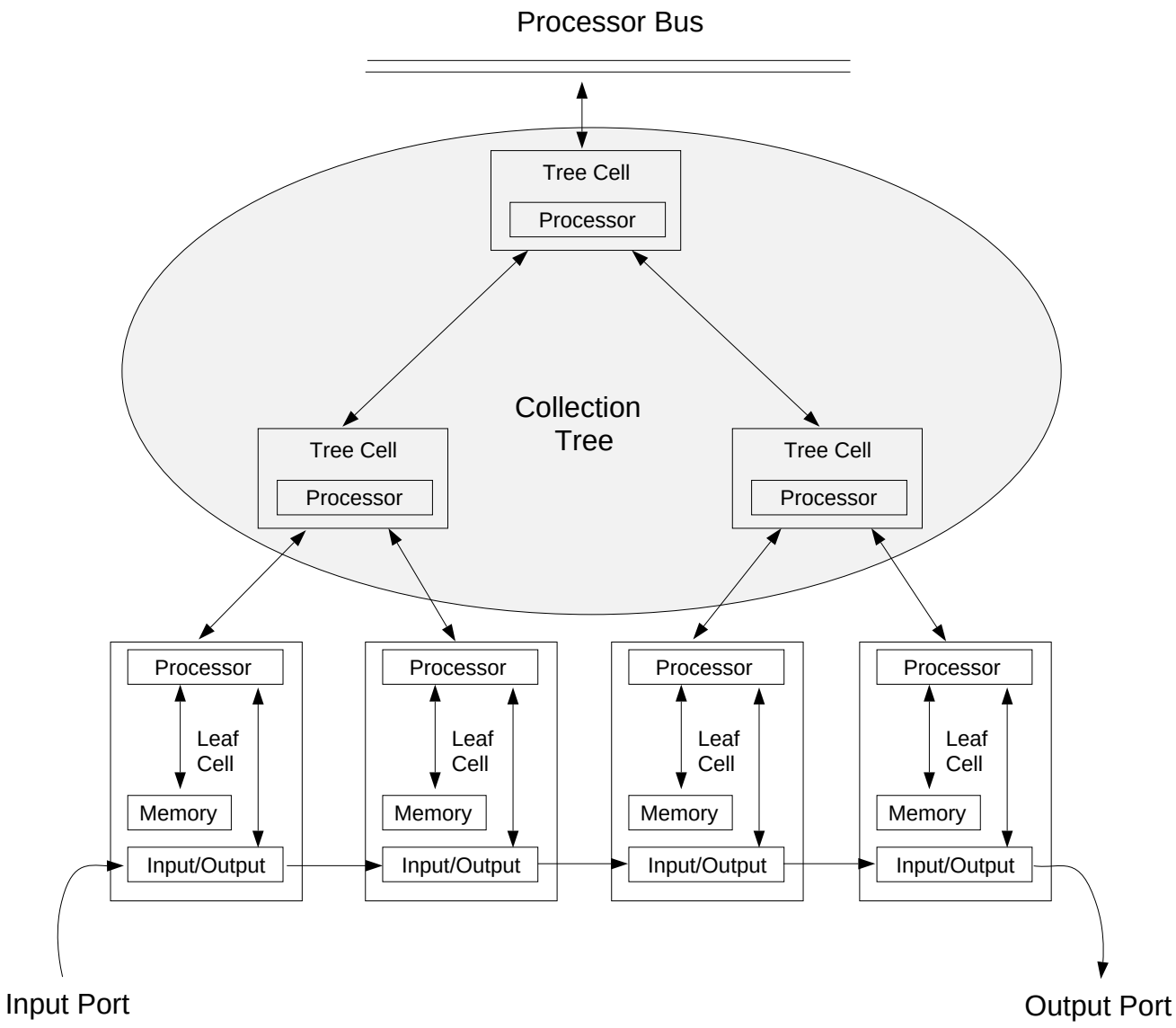

Figure 1: Rutgers CAM2000 chip Architecture

2 Functional Capability

The Rutgers CAM2000 uses enhanced version of many *classical* CAM features to improve usability and performance. Of these enhancement, the following four are critically tied to the performance of the CAM2000 architecture:

Wide Words:

Classical CAMs have typically employed many 1-bit processors. However, in practice, many algorithms perform better on architectures with fewer wider processors than on architectures with more, narrow processors. The CAM2000 architecture is based on balancing the number of processors against the width of each processor and is parameterized to allow experimentation with alternative design points.

Global Operations:

Classical CAMs perform operations such as count-responders and sum-all-values in a word-parallel bit-serial manner using software to perform the bit-serial portion of the computation. In practice the time required for a *classical* CAM algorithm is often dominated by the software emulation of its bit-serial global operations. The CAM2000 provides hardware to perform such global operations and consequently provides the speed of *classical* CAM without the overhead of software emulated global operations.

Segmentation:

Segmented operations are performed in a *classical* CAM model using activity control to *sequentially* process one segment at a time. This approach is extremely slow since in addition to the penalty of using software emulation, operations on the segments must be performed sequentially. The CAM2000 provides hardware control that supports arbitrary segmentation over all global operations thus allowing segmented global operations to execute in the same time as their unsegmented version.

Local addressing:

Classical CAM uses a SIMD model and operates on the same word in each leaf cell.[1] The design of the CAM2000 allows the choice of words to be varied across leaf cell thus improving the performance of expression matching and making the CAM2000 architecture well suited for higher-level models of computation.[2]

2.1 Supported Operations

Functionally the Rutgers CAM2000 provides *standard* DRAM operations as well as local vector operations, global collection operations, and parallel prefix/suffix operations. The parallel prefix/suffix operations are similar to those reported by Blelloch's [Blelloch, 1987, Blelloch, 1990], but the CAM2000 architecture differs from Blelloch's in the fundamental regard that the CAM2000 architecture is a memory architecture and not a model for parallel computation. As such, the CAM2000 focuses on implementing memory-based prefix and suffix operations with simple, dedicated, efficient hardware.

Most operations complete in a single CAM2000 cycle, which is roughly the time required for one DRAM memory access; however, the complexity of some primitive operations require more than one cycle.[3] In addition there are a few higher level operations, such as vector-valued collective functions, that are composed of two multicycle primitive operations. As a group, the operations supported by the CAM2000 provide substantial algorithmic advantages over systems that use *conventional* DRAM. These additional operators are described below.

[1] We have adopted a terminology more in line with that used in the massively parallel literature rather than the CAM literature. In the CAM literature this sentence would read, CAM uses a SIMD model and operates on the same field (in contrast to word) in each CAM word(in contrast to leaf cell).

[2] At present, we are considering several implementation alternatives to support local addressing and have not committed to any one.

[3] In our design, the number of cycles for any primitive operation is bounded by a constant. Based on estimates of year 2000 technology, multicycle primitive operations will require between 3 and 5 cycles.

2.1.1 Parallel Vector Operations

Parallel Vector operations are performed independently by each leaf cell on its local memory partition and/or registers in a SISD manner. The Rutgers CAM2000 extends these operations in two ways. The first is to allow one of the operands to be a *global* value (i.e., the same value in each cell). The second extension is the *classical* SIMD extension of activity control allowing each leaf cell to be enabled or disabled on a per-cell per-operation basis. If the cell is inactive the operation is ignored and state is maintained, if the cell is active the operation is performed as specified. This is a important difference between CAM and *simple* vector style computation.

The supported operations are: integer addition, subtraction, comparison, bitwise boolean functions, and shift left[4]. Integer multiplication, division, and floating point are not supported although they can be emulated in software.

Two examples of activity controlled parallel vector operations are shown in Figure 2. Both examples add 2 to each CAM cell: one shows the results when all cells are enabled (i.e., activity=1) while the other shows the results when some cells are disabled (i.e., activity=0).

	Without activity control All cell enabled								With activity control Some cells disabled							
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
input	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
result	4	5	6	7	8	9	10	11	4	3	6	7	6	9	10	9

Figure 2: Parallel Vector Add-2 on an 8 cell CAM

2.1.2 Scalar-valued and vector-valued collective operations

Collective operations are associative and belong to one of two classes, operations that return a scalar and operations that return a vector. These operations accept their inputs from the root and leaf cells, carry out the necessary computation, and place the final result back in the root and leaf cells.

The supported scalar-valued collective functions are: integer sum, max, and min for both positive integer and 2's complement form. Also supported are the binary bitwise boolean functions OR, AND, XOR as well as the bitwise selection functions *left* and *right* which route the data of the named child (i.e., *left* or *right*) to the parent. These collective functions are termed complete if they depend on all leaf cells or partial if they depend on a subset of leaf cells. Participation in partial functions is determined on a per-cell per-operation basis as determined by control bits in the leaf cells.

All scalar-valued collective functions have vector-valued counterpart that are also supported in hardware. As with the scalar-valued collective functions, the vector-valued func-

[4]Other shifting and byte extraction operations are being considered. Additional shifts and/or extraction operation will be implemented based on use patterns typical of applications.

tions can be complete or partial. In addition, The vector-valued operations are partitioned into prefix and suffix forms and independently into inclusive and exclusive forms. Prefix and suffix forms differ in the vector's *direction* while inclusive and exclusive forms differ based on the relationship between the i^{th} result and the i^{th} input[5]. The CAM2000 architecture provides hardware in the tree cells that support the exclusive form of both the prefix and suffix variant of these collective operators. Support for the inclusive forms is provided through hardware in the leaf cells that constructs an inclusive result from its exclusive counterpart. Figure 3 shows the four vector-valued results when the function $\diamondsuit$ with identity element ϕ is applied to the input $\langle x_1, x_2, \ldots, x_n \rangle$.

Input:	$\langle$	x_1	x_2	$\cdots$	x_{n-1}	x_n $\rangle$
Exclusive Prefix:	$\langle$	ϕ	x_1	$x_1 \diamondsuit x_2$	$\cdots$	$x_1 \diamondsuit x_2 \cdots \diamondsuit x_{n-1} \rangle$
Inclusive Prefix:	$\langle$	x_1	$x_1 \diamondsuit x_2$	$x_1 \diamondsuit x_2 \diamondsuit x_3$	$\cdots$	$x_1 \diamondsuit x_2 \cdots \diamondsuit x_n$ $\rangle$
Exclusive Suffix:	$\langle x_2 \diamondsuit x_3 \cdots \diamondsuit x_n$	$\cdots$	$x_{n-1} \diamondsuit x_n$	x_n	ϕ	$\rangle$
Inclusive Suffix:	$\langle x_1 \diamondsuit x_2 \cdots \diamondsuit x_n$	$\cdots$	$x_{n-2} \diamondsuit x_{n-1} \diamondsuit x_n$	$x_{n-1} \diamondsuit x_n$	x_n	$\rangle$

Figure 3: Vector-valued collection operations

In most cases the result of a vector-valued collective operation is obvious from its scalar-valued counterpart; however, the scalar operations *left* and *right* have unusual vector-valued counterparts. We term these vector-valued functions skip-shifts since they move a value from each active leaf cell to the next active leaf cell. The time taken by this operation is independent of the distance between active leaf cells or the direction of the shift. An example of an activity-controlled right-moving skip shift implemented using the exclusive prefix operation *right* with identity element ϕ is shown in Figure 4.

activity bit	1	0	1	1	0	1	1	0
input	2	3	4	5	6	7	8	9
result	ϕ	3	2	4	6	5	7	9

Figure 4: Right-moving Skip Shift: exclusive prefix *right*

2.1.3 Segmented and Unsegmented collective operations

Both scalar-valued and vector-valued collective functions can be arbitrarily segmented. The segmentation points, like the activity, are dynamically determined on a per-cell per-operations basis by each leaf cell. In an unsegmented collective operations the leaf cells belong to one contiguous segment and the operation is performed as described in section 2.1.2. Segmented

[5]The operation is termed inclusive if result r_i depends on input x_i. If r_i does not depend on x_i the operation is termed exclusive.

operations partition the leaf cells into independent segments and perform the same operations, one on each segment in parallel. As with local operations, leaf cells specify on a per-cell per-operation basis if their data will participate in the operation.

Two examples of an unsegmented exclusive prefix sum with identity element 0 are presented in Figure 5.[6] Both compute all partial sums, one with all cells enabled and another with some cells disabled. A value of 1 in the activity bit indicates that the cell is enabled and will participate in the computation. A value of 0 in the activity bit indicates that the cell is disabled and will *not* participate in the computation. Notice that when a cell is disabled it maintains its previous state.

	All cells enabled								Some cells disabled							
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
input	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
result	0	2	5	9	14	20	27	35	0	3	2	6	6	11	18	9

Figure 5: Unsegmented exclusive prefix sum with identity element 0

Segmented global operations use a segment bit in each leaf cell to determine if a leaf cell is the first element of a new segment or an internal element in the current segment. If the segment bit is 1 the leaf cell is treated as the first cell of a new segment, if it is 0 the cell is a continuation of the current segment. Figure 6 present two examples of a segmented exclusive prefix sum. As with the unsegmented example, one is an example without activity control while the other is an example with activity control.

	All cells enabled								Some cells disabled							
segment bit	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
activity bit	1	1	1	1	1	1	1	1	1	0	1	1	0	1	1	0
initial state	2	3	4	5	6	7	8	9	2	3	4	5	6	7	8	9
final state	0	2	5	9	14	0	7	15	0	3	2	6	6	0	7	9

Figure 6: Segmented exclusive parallel prefix sum on an 8 cell CAM

2.1.4 Input/Output

The CAM2000 provides simple dedicated hardware that allows I/O to be done in parallel with all other operations. IO operations provide leaf-at-a-time word-parallel shifting independent of activity control and segmentation. They are implemented using a single register in each leaf cell that can function either as one element of a shift register spanning all leaf cells or as a local register to the leaf cell in which it is contained.

[6] In the CAM2000 architecture, the tree nodes determine if a global operation produces a result that is the identity element. The value of the identity element is inserted by the leaf node - the tree nodes know only that a result is the identity element and not the actual value of that element.

IO shift operations run asynchronously and independent of leaf operations employing a cycle that is roughly 5 times smaller than a leaf cycle. The IO shift operations require time proportional to number of cells shifted across and though essentially an I/O facility can be used to move data between leaf cells.

3 Architecture

The CAM2000 memory architecture combines processor and memory components on a single chip. Each chip, regardless of the amount of CAM onboard, has 4 busses: a bidirectional data bus connected to the main processor(s), an input-only instruction bus, an input-only I/O bus, and an output-only I/O bus.[7] The architecture is currently parameterized in terms of datapath width allowing us to investigate arbitrary data-widths up to 32-bits: we believe that by the year 2000 technology will support a 32-bit datapath CAM2000 architecture. For a given datapath width, the chip interface is constant size and the CAM architecture is consequently *scalable* in the strongest sense of the word.

Internally the chip's architecture is built around two main components: the leaf cell and the tree cell. The leaf cells handle all local operations and interfaces to the I/O component, the memory system, and the tree component. The tree cells function as a unit handling global operations and providing datapaths among leaf cells and the system memory bus. Both leaf and tree cells are controlled by the instructions presented on the instruction bus.

3.1 Timing Estimates

We have simulated our design using an event driven gate level simulator to obtain estimates of performance and based on these estimates we have refined the design to improve performance.

In the current design a leaf cycle requires 50 gate delays using 20 of these 50 in decoding, routing, and other *support* circuitry and the other 30 for execution in the ALU. Since all ALU operations except addition/subtraction complete in far fewer than 30 gate delays, we were presented with several design choices including pipelining and variable time instructions. We chose to keep the design simple and to use fixed time operations thus making the system speed critically dependent on the time required for the ALU to perform addition. Due to the speed and size issues constraining the addition operation we chose to use a carry-skip circuit optimized to produce a 32-bit result along with ALLZERO and OVERFLOW status bits in minimal time.[8]

The speed of tree operations is related to word width w and tree depth d. In order to meet performance goals the tree is designed as a combinational circuit using a two-dimensional ripple that simultaneously ripples results across words and across tree levels. This technique allows the tree to complete *slow* operations (e.g., addition, minimum and maximum) in roughly $2w + 10d$ gate delays and *fast* operations (e.g., OR, AND, XOR) in roughly $10d$ gate delays. For the CAM2000's target design of 32-bit word and 1024 leaves per chip, a tree operation would require between 120 and 184 gate delays, including 20 gate delays for *support* circuitry.

[7] The input and output I/O busses are opposite ends of the I/O shift register joining the leaf nodes.

[8] This optimization lead to a design in which addition incurs an 8% overhead to compute ALLZERO and OVERFLOW (i.e., the sum is available after 25 gate delays and the status bits are available 2 delays later) while *simple* operations such as OR incurred a 100% overhead (i.e., the result is available after 10 gate delays and the status bits are available only after another 10 delays).

Again we faced a multiplicity of design choices and opted for simplicity by limiting the time for a tree operation to an integral number of leaf cycles (i.e., 50 gate delays). However, since the speed of the tree is critical to many applications we decide to allow tree operations to take a variable number of leaf cycles and designed the instruction set with tree status information that specifies, for each leaf cycle, if a tree operation is starting, continuing, or ending. The design decision for the circuit, architecture, and instruction set provide a general framework in which tree operations may take as few as 2 cycles or as many as required. Based on technology trends we believe that any tree operation will require at least 2 leaf cycles and at most 5. In our current design tree operations require either 3 or 4 leaf cycles.

3.2 The Leaf Processor Component

Each leaf cell is responsible for performing local CAM-like operations and is composed of a main processor connected to three support processors via multi-ported interface registers: the tree register(TR), the refresh register(RR), and the I/O register(IOR). The leaf cell's main processor is composed of two communicating components; a w-bit system[9] for *standard* operations and a 1-bit system used to manage status and control activity within a leaf cell. Figure 7 shows the interconnection of these components.

3.2.1 The w-bit ALU

The w-bit system employs a three bus (i.e., a primary input bus GB_p, a secondary input bus GB_s, and a result bus GB_r) architecture and is composed of:

- a w-bit ALU containing a carry-skip adder optimized for 32 bit computation,
- five general purpose w-bit registers GR_1, GR_2, GR_3, GR_4, and GR_5,
- a dual ported w-bit flag register(FR) directly coupled to the writable 1-bit registers,
- a dual ported (w+1)-bit register TR that provides the interface to the tree processor,[10]
- a tri-ported w-bit refresh register register(RR) that provides the interface to the memory, and IO processors.
- a dual ported IO register(IOR) that provides the interface between the IO processor and the refresh register (RR).

The set of implemented operations includes AND, OR, XOR, addition, and subtraction as well as transfer-primary and transfer-secondary. These last two instructions route the data from the indicated input bus to the result bus. In addition to the primary and secondary

[9] We expect the w-bit processor and its associated registers to be 32-bits wide; however, our design is not restricted to 32-bit widths but parameterized as a function of word width.

[10] The extra 1-bit of width is used to communicate overflow information.

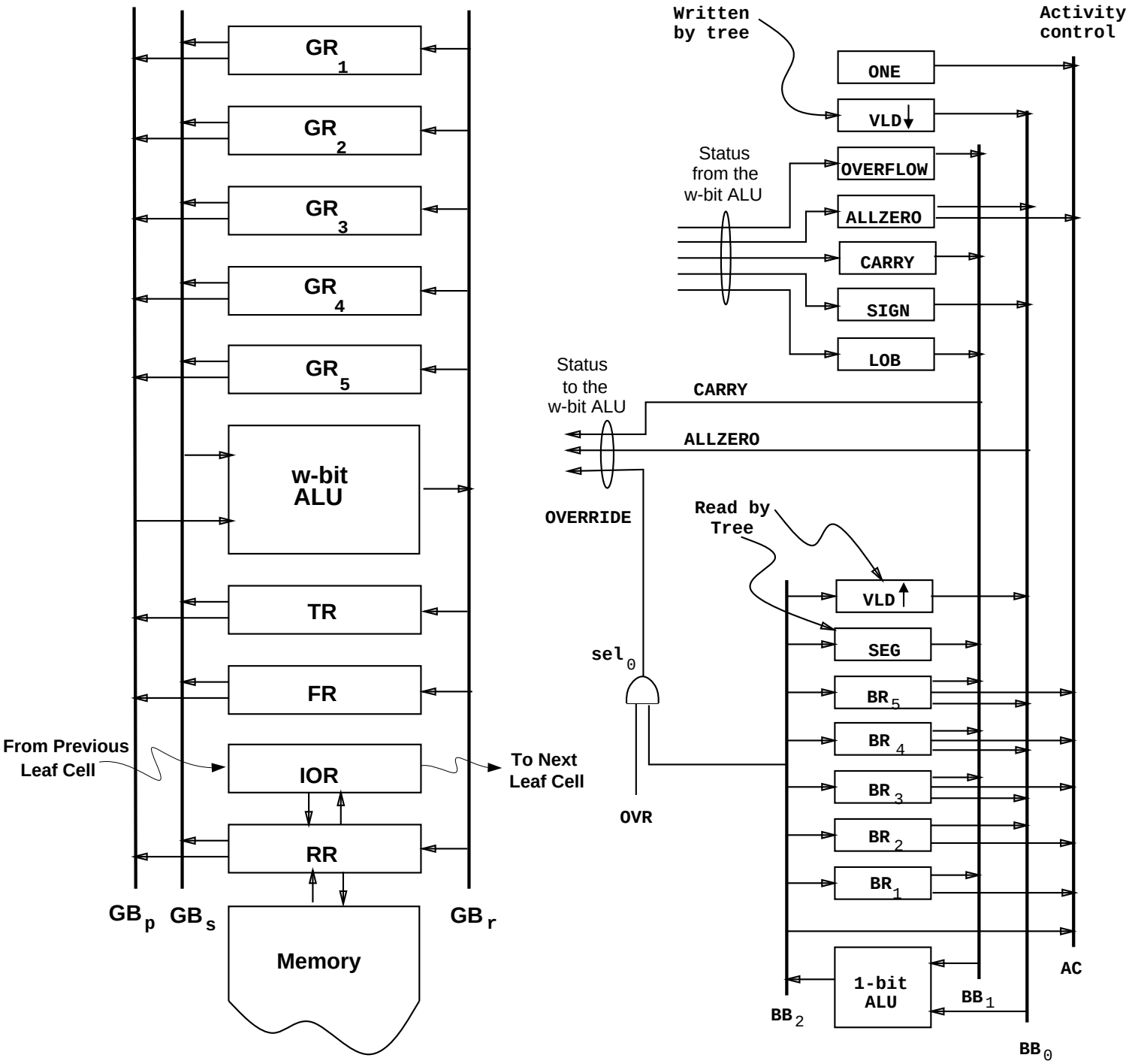

Figure 7: Leaf cell architecture

w-bit input busses there are 3 1-bit input lines: CARRY, ALLZERO, and OVERRIDE. CARRY and ALLZERO are used for extended precision operations and allow the leaf cell to efficiently produce results for fields wider than w-bits. The OVERRIDE allows the ALU to efficiently handle inclusive prefix and suffix operations and provides a single-cycle test-and-set operation.

The output of the w-bit ALU is connected to the w-bit result bus GB_r and to five 1-bit status bits: OVERFLOW, ALLZERO, CARRY, SIGN, and LOB (low order bit).[11] The CARRY, SIGN, and LOB bits are copies of other bits computed by the ALU; the ALLZERO bit is zero if all output bits are zero and 1 otherwise; and the OVERFLOW bit indicates if an overflow was detected during the computation of the value on bus GB_r.

All operations except addition and subtraction can be performed bit-parallel and consequently require little time to complete. Addition and subtraction cannot be performed bit-parallel and therefore require a *special* design to provide the desired speed. The CAM2000 architecture strikes a balance between the speed and size of the w-bit ALU by using a carry skip adder optimized for a 32-bit result. The adder is partitioned into 9 stages sized in bits from high order to low order as 3,4,4,5,5,4,3,2,2. The sum is available 25 gate delays after the input is applied with the ALLZERO and OVERFLOW available 2 gate delays later (i.e., 27 gate delays after the input is applied).[12] In comparison, a ripple carry adder would be over 2.5 times slower requiring at least 68 gate delays to produce an OVERFLOW indication and an optimal speed dedicated carry-lookahead circuit would provide no more than a 14% improvement in speed, requiring at least 25 gate delays to produce the OVERFLOW signal.

Override control

Exclusive prefix and suffix operations are performed using only the tree cells; however, inclusive prefix and suffix operations require both the tree and leave cells. In order to provide efficient inclusive operations the leaf cell's w-bit ALU has been designed with an override control. When asserted this line causes the ALU to ignore the operation specified on the instruction bus and to execute a transfer-primary operation instead. This feature not only provides efficient exclusive scans but can also be used as a single cycle test-and-set operation or to support certain MUX-like operations. Sections 3.2.7 and 3.2.7 contains examples showing the use of the Override feature.

Overflow condition

The overflow condition computed by the w-bit ALU is the *standard* one used for integer arithmetic except in the case when the result is dependent on the value in TR. This case is exceptional since TR can contain one word of the vector-result computed by the tree cells

[11] At present the CAM2000 design requires the status bits to be changed on every leaf cycle. The generation of optimized code is adversely affected by this decision and we are considering enhancing the architecture to allow selective writing of these bits.

[12] In current design OVERFLOW is not available until 2 delays after the ALLZERO; however, a simple enhancement can reduce this by 2.

and as such will be depend on the overflow(s) generated by that tree operation.[13] When a leaf cell uses the result in TR the hardware assumes that it is finishing a computation begun in the tree and that the associated overflow computed by the leaf cell should reflect the entire computation and not just the last step performed in the leaf cell.

Consequently, the overflow condition of every leaf operation that uses TR as a source register, including simple data movements such as $GR_i \leftarrow TR$, is set if either TR's overflow bit is on or if the w-bit ALU operation generates an overflows. This idea is conceptually clean; however, the interactions between the override signal and the transfer-primary and transfer-secondary operations make deciding if TR is a source to the w-bit ALU surprising complex. The following boolean expression indicates when TR should be considered an input to the w-bit ALU.

$$(TR_p \wedge ovr) \vee (TR_p \wedge OP'_s) \vee (TR_s \wedge OP'_p \wedge ovr')$$

where: TR_p is true if the value of TR is on bus GB_p
TR_s is true if the value of TR is on bus GB_s
OP_p is true if the operation is transfer-primary
OP_s is true if the operation is transfer-secondary
ovr is true if the OVERRIDE line to the ALU is on

3.2.2 The 1-bit system

The 1-bit system is a four bus (i.e., BB_0, BB_1, BB_2, AC) architecture composed of:

- a 1-bit ALU
- five general purpose 1-bit registers BR_1, BR_2, BR_3, BR_4, and BR_5
- a dual ported 1-bit segment register(SEG), read by the tree nodes, that indicates if a leaf cell is (SEG=1) or is not (SEG=0) the first element in a new segment.
- a dual ported 1-bit status register(VLD$\uparrow$), read by the tree cells, that indicates if the tree component should use (VLD$\uparrow$=1) or ignore (VLD$\uparrow$=0) the data in TR.
- five 1-bit status registers(OVERFLOW, ALLZERO, CARRY, SIGN, LOB) that contain the status of the w-bit ALU
- a dual ported 1-bit status register(VLD$\downarrow$), written by the tree nodes, that indicates if the result computed by the tree nodes and stored in the tree register should be used (VLD$\downarrow$=1) or ignored (VLD$\downarrow$=0) by the leaf cell.
- a 1-bit constant (ONE) used to activate leaf cells

[13] The overflow computed by the tree cells indicates for each value produced if that value depends on any computation that overflowed. It is stored in the *extra* bit of the (w+1)-bit tree register TR.

The 1-bit system controls both the leaf cell's activity using line AC and, in conjunction with the instruction bus OVR signal, the w-bit ALU's override input. The 1-bit ALU implements all 16 2-input 1-bit functions using a 4-bit operation code with table lookup.

Three of its fourteen registers form an interface to the tree cells. Two of these, SEG and VAL↑, are read by the tree cells and one, VLD↓ is written by the tree cells. All three may used as an input to the 1-bit ALU.

Five of its registers form an interface to the w-bit ALU. These registers, OVERFLOW, ALLZERO, CARRY, SIGN, and LOB are written by the w-bit ALU as status bits and can be read, but not written, by the 1-bit ALU. Two of these registers, CARRY and ALLZERO, can be directly fed back to the w-bit ALU thus providing efficient extended precision operations.

3.2.3 The 1-bit and w-bit system Interface

The 1-bit and w-bit systems communicate through the processor status registers OVERFLOW, ALLZERO, CARRY, SIGN, and LOB, the activity control, the dedicated connections joining the 1-bit registers to w-bit register FR, and the override(OVR) signal.

Activity control is determined by the value applied to wire AC by the the 1-bit system and is used to enable or disable the writing of both 1-bit and w-bit registers from their respective output buses, GB_r and BB_2. The value placed on AC can be obtained directly from any of the general purpose 1-bit registers BR_1, BR_2, BR_3, BR_4, BR_5, or from special purpose 1-bit registers ALLZERO, or ONE. In addition, the output of the 1-bit ALU (BB_2) can also be routed to AC allowing values stored in registers other than those mentioned above, for example SIGN, to be placed on AC. In these cases the 1-bit ALU is part of the signal path to AC and cannot, therefore, be used to perform a simultaneous independent computation.

The dedicated connections between the 1-bit registers and FR will provide an additional interface that increases the bandwidth between the 1-bit and w-bit components. Currently our design implements FR as a general purpose register and does not provide connections to the 1-bit registers; however, the final design will introduce connections between FR and the 1-bit registers. The specific connections included will be determined by the requirements of typical applications.

3.2.4 Leaf and tree cell Interface

The interface between the leaf and tree cells is composed of registers TR, SEG, VLD↑, and VLD↓. TR acts as a bidirectional data port connecting the leaf and tree cells. When the tree accepts data from the leaves it reads VLD↑ and SEG. VLD↑ indicates if the data in TR should, or should not, participate in the tree operation while SEG indicates if the leaf is, or is not, the first in a segment. These two 1-bit registers can be read and written by the 1-bit ALU; however, they may only be read by the tree.

When the tree provides data to the leaves it use VLD↓ to indicates if the data in TR

should, or should not, be used by the leaf. VLD$\downarrow$ can only be read by the leaf processor and can only be written by the TREE processor. There is dedicated hardware in the tree that computes VLD$\downarrow$ as a function of the SEG, VLD$\uparrow$, and the direction of the operation type (i.e., prefix or suffix). Changing any of these three fields will cause VLD$\downarrow$ to immediately change.

3.2.5 Leaf cell and memory interface

The read and write commands form the conceptual interface between the leaf cell and memory. They are executed by the memory processor and cause data to be transferred between RR and the memory. The read command moves data from memory to RR while the write command moves data from RR to memory. These commands causes the memory processor to transfer the data to or from the location specified by ADDR. If the read is destructive, as will be the case with a DRAM implementation, the main control unit will need to issue a write command to rewrite the contents of RR back to memory. Neither the read or write command is affected by the processor's activity control.

Once initiated by either a read or write the memory processor performs the data transfer asynchronously. The leaf processor should not change RR while the memory processor is busy and is thus limited to using RR only when the memory processor is idle.

3.2.6 Instruction Bus Fields

Operation of the leaf cells are encoded on the instruction bus in the fields described below. The estimated width of each field is shown in parentheses. A summary of all fields of the instruction bus is provide in Figure 14 on page 22. This description is tentative; refinements to field size and instructions will be made as algorithms are implemented on this architecture.

GOP(4):

GOP specifies the operation to be performed by the w-bit ALU.

R_{GB_p} and R_{GB_s} (3):

R_{GB_p} and R_{GB_s} specify which register will be routed to busses GB_p and GB_s, respectively. Each field encodes one of the refresh register, one of the w-bit general registers, the tree register, or the flag register. Our current design using 5 general registers is encoded as follows:

000:RR 001:GR_1 010:GR_2 011:GR_3 100:GR_4 101:GR_5 110:TR 111:FR

R_{GB_r} (3) :

R_{GB_r} specifies which register will write the value on GB_r. It encodes one of a null destination(λ), a w-bit general registers, the tree register, or the flag register. The value is recorded if and only if the leaf processor's activity control, as determined by

AC, is set. This field cannot specify the refresh register. Our 5 register design is encoded as:

000:λ 001:GR_1 010:GR_2 011:GR_3 100:GR_4 101:GR_5 110:TR 111: FR

RRC(2):

RRC is a 2 bit field that specifies what data, if any, is written to the refresh register. The field encodes one of four possibilities: a noop(λ), write from GB_r, write from memory, or write from IOR. Specifying this field independent of the R_{GB_r} field allows the refresh register to be written in parallel with any of the destinations specified by R_{GB_r}. It also isolates the leaf processor from the memory and IO processors allowing each to run at its own optimal speed. Activity control is used when RR is being written from GB_r - it is ignored for all other cases. Our current design encodes RRC as follows:

00: λ 01: RR $\leftarrow$ GB_r 10: RR $\leftarrow$ M[ADDR] 11: RR $\leftarrow$ IOR

ADDR(lg(n)**:**

ADDR is a lg(n) bit field, where n is the number of words of memory per leaf cell, that specifies the memory location to be read from or written to. This field is required only when a read or write is being performed.

WR(1):

WR is a 1-bit field that indicates when data is to be written from the refresh register to the memory (M[ADDR] $\leftarrow$ RR). This operation can be performed in parallel with other operations that access the refresh register.

IOC(1):

IOC is a 1-bit field that specifies when data is written from the refresh register to the IO register (IOR $\leftarrow$ RR).

OVR(1):

OVR is a 1-bit field that enable or disables the override capability of the w-bit ALU. When OVR is set the 1-bit ALU determines if the w-bit ALU performs either the operation specified by field GOP or overrides that specification and transfer data from input bus GB_p to output bus GB_r. The override capability is used for computing inclusive scans from exclusive scans as well as providing MUX-like capabilities to the w-bit ALU (see Sections 3.2.7 and 3.2.7).

BOP(4):

BOP specifies the operation to be performed by the 1-bit ALU. The 1-bit ALU is implemented as a lookup table using using BB_0 and BB_1 to index the truth table whose entries are given by the 4 bits of field BOP.

R_{BB_0}, R_{BB_1}, and R_{AC}(3):

R_{BB_0}, R_{BB_1}, and R_{AC} specify which register will be routed to wires BB_0, BB_1, and AC respectively. Each field encodes one of 8 registers possible registers. These encodings are based on the interrelations between the 1-bit registers and a *typical* instruction mix. The specific registers connected to each wire and their encodings will be altered as experience is gained with algorithms on this architecture. The following are design goals which influence these choices as well as an initial assignment of registers to busses.

- Extended precision requires that the CARRY and ALLZERO status outputs from the w-bit processor be fed back into the processor's CARRY and ALLZERO inputs. Consequently, data paths that allow parallel routing of the CARRY and ALLZERO registers to the w-bit processor must be supported.
- Inclusive operations completed in the leaf processors must be efficient.[14] This is accomplished using the 1-bit ALU, the override feature of the w-bit ALU, the 1-bit registers VLD↓, and SEG with VLD↓ and SEG presented in parallel to the 1-bit ALU. Details of these operations are described in Section 3.2.7.

R_{BB_0} is encoded as:

000:BR_2	001:BR_3	010:BR_4	011:BR_5	100:VLD↑
		101:SIGN	110:ALLZERO	111:VLD↑

R_{BB_1} is encoded as:

000:BR_1	001:BR_3	010:BR_4	011:BR_5	100:SEG
		101:LOB	110:CARRY	111:OVERFLOW

R_{AC} is encoded as:

000:BB_2	001:BR_1	010:BR_2	011:BR_3	100:BR_4
		101:BR_5	110:ALLZERO	111:ONE

R_{BB_2}(3);

R_{BB2} specifies the register that will record the value on BB_2. This value is recorded if and only if the processor's activity control, as determined by AC, is set. Our initial design is encoded as:

000:λ 001:BR_1 010:BR_2 011:BR_3 100:BR_4 101:BR_5 110:SEG 111:VLD↑

3.2.7 Using the 1-bit Override feature

Any w-bit operation can be executed as specified or overridden. When overridden, an operation is replaced by the transfer-primary operations. Figure 8 shows the operation $\diamond$ and its overridden counterpart. Notice that the data transfer that replaces the specified operations ignores the fields R_{GB_s} and GOP but uses the fields R_{GB_p} and R_{GB_r} without alteration.

[14]The current design requires at worst 2 leaf cycles to complete an inclusive operation.

	Operation Mnemonic	Relevant instruction fields
Operation	$GR_i \leftarrow (GR_j \diamond GR_k)$	R_{GB_p}=j, R_{GB_s}=GR_k, R_{GB_r}=i, GOP=$\diamond$
Overridden Counterpart	$GR_i \leftarrow GR_j$	R_{GB_p}=j, R_{GB_r}=i

Figure 8: Comparison of the operation $\diamond$ and it overridden counterpart

Forming Inclusive results from their Exclusive Counterparts

When an inclusive result is desired, the leaf processor must compute it from the TREE component's exclusive result and its own internal data. If the operation is unsegmented with all leaf cells participating the conversion of an exclusive result to its inclusive form is straight forward. However, when the operation is segmented and/or has some non-participating leaf cells the conversion is more complex. In general, the conversion is done using the override capability of the w-bit ALU to either use or ignore the exclusive result reported by the tree in TR.

In order to produce inclusive scans from exclusive scans two cases must be considered. One, when the tree processor provides data on which the inclusive scan depends (i.e., VLD$\downarrow$=1 $\wedge$ SEG=0) and two, when it does not (i.e., VLD$\downarrow$=0 $\vee$ SEG=1). Notice that these conditions are functions not only the VLD$\downarrow$ signal produced in the tree but also of the segmentation bit held in the leaf processor. This latter dependence is due to the fact that the first leaf processor in a segment (SEG=1) must ignore the data it receives from the tree since this data is from a different segment. The leaf processor must be able to distinguish between these cases and complete the inclusive scan appropriately. Figure 9 shows instructions that the leaf processor can execute to complete the inclusive operation $\diamond$ for these two cases.

Tree data should be used	$GR_i \leftarrow (GR_j \diamond TR)$
Tree data should be ignored	$GR_i \leftarrow GR_j$

Figure 9: Leaf operations for forming inclusive result from exclusive result

Since the second of these operations is an overridden version of the first, the choice between the two operations can be made by controlling the override capability of the w-bit ALU with the 1-bit ALU. Figure 10 shows the 1-bit and w-bit instructions that achieve this result when the OVR filed of the instruction bus is set.

Using the w-bit ALU as a Multiplexor for MIN and MAX operations

The override capability can also be used to cause the w-bit ALU to function as a MUX routing one of its inputs, GB_p or GB_s, to GB_r. This is accomplished by using OVR and the 1-bit ALU in conjunction with the w-bit operation transfer-secondary as shown below. Since the transfer-secondary routes the secondary bus to the output and its overridden counterpart

w-bit ALU	1-bit operation
$GR_i \leftarrow (GR_j \text{ op } TR)$	$VAL\downarrow \wedge \overline{SEG}$

Figure 10: Using the w-bit override to form an inclusive result

routes the primary bus to the output this operation can be used as a multiplexor. The syntax may seem awkward since it specifies two arguments to a function with arity one; however, it makes it clear that the override capability of the w-bit ALU does provide MUX-like performance.

$GR_i \leftarrow$ transfer-secondary (GR_j,GR_k)
where: transfer-secondary performs $GR_i \leftarrow GR_k$

The ability to use the w-bit ALU as a MUX allows the leaf to efficiently form inclusive results from their exclusive counterparts. It is especially useful for MIN and MAX operations because these operations are performed differently in the tree and leaf cells[15]. In a TREE cell MIN and MAX are functions that output either the MIN or MAX of their inputs but in a leaf cell, MIN and MAX are *emulated* by comparing the two datum and selecting one based on the result of the comparison. This difference requires that inclusive MIN and MAX operations use the two steps show in Figure 11 to convert an exclusive result to an inclusive result.

$GR_i \leftarrow MIN(GR_j,TR)$

OVR field	w-bit operation	1-bit operation
off	$\lambda \leftarrow (GR_j$ - TR)	$BR_1 \leftarrow (\overline{VLD\downarrow} \vee SEG)$
on	$GR_i \leftarrow$ transfer-secondary(GR_j,TR)	$\lambda \leftarrow (BR_1 \vee SIGN)$

Figure 11: leaf cell steps to complete MIN/MAX inclusive scans

In the first step, while the TR is being subtracted from GR_j in the w-bit processor, the 1-bit processor decides if TR's contents are required to form the inclusive result, storing this in BR_1. In the second step the 1-bit processor controls the override setting of the w-bit processor so that it acts as a multiplexor selecting either GR_j or TR and routing it to GR_i. The override condition is computed in the 1-bit ALU based on the result stored in BR_1 and the sign bit produced by the subtraction operation. GR_j is selected either when the tree data should not be used or the contents of GR_j is less than TR.

[15] Addition is performed identically in the leaf and TREE cells.

3.2.8 Inserting identity elements

Tree cells perform segmented exclusive operations to which leaves selectively contribute and must, therefore, be able to determine when a result's value is the operation's identity element. Rather than designing the tree to insert the required identity element when needed we have opted to use a single bit to indicate whether a value should be used as is or interpreted as the operation's identity element. Consequently, when a resulting value should be interpreted as the operation's identity element the leaf processor must insert it.

This is accomplished using the MUX-like capabilities of the leaf processor to choose between the data provided by the tree and the identity element for the operation. Four different constants are required to provide the identity elements for all supported tree operations. These constants in addition to other general purpose constants encoded in a 3-bit field in which one bit specifies the high order bit, one the internal bits, and one the low order bit. Figure 12 shows the eight possible constants and some of their uses.

	Constant		Use as identity	General use
0	00..00	0	+, OR, XOR, positive integer MAX	
0	00..00	1		increment
0	11..11	0		
0	11..11	1	2's complement MIN	
1	00..00	0	2's complement MAX	
1	00..00	1		
1	11..11	0		subtract 2
1	11..11	1	AND, positive integer MIN	decrement

Figure 12: Important Constants

3.2.9 Activity controlled write to memory

Activity control is used on all w-bit and 1-bit registers. These registers latch their input values at the end of each execute cycle if and only if they are selected by R_{GB_r} or R_{BB_2} and AC is set. The memory system is not activity controlled and consequently there is no activity controlled write-to-memory. However, the effect of this operation can be obtained by using the CAM2000's standard memory operations in conjunction with an activity controlled register operation on RR. When used with a DRAM memory performing destructive reads this composite operation takes only slightly longer than a *typical* memory cycle. Figure 13 shows the three steps required to perform an activity controlled write-to-memory operation on the CAM2000.

$\underline{\text{M[ADDR]} \leftarrow \text{GR}_i \text{ in active leaf nodes}}$

$\text{RR} \leftarrow \text{M[ADDR]}$
$\text{RR} \leftarrow \text{GR}_i$ mediated by AC
$\text{M[ADDR]} \leftarrow \text{RR}$

Figure 13: Activity Controlled write to memory

- Tree Instruction:

  TOP(3): tree operation

  LR(1): tree operation type (prefix/suffix)

  UD(1): tree operation direction (up/down)

  U2(1): tree operation data-type

  TS(2): tree operation status (starting,continuing,stopping)

  ES(2): tree operation extended status(first nibble, middle nibble, last nibble)

- w-bit Instruction:

  GOP(4): w-bit operation

  OVR(1): Override enable for w-bit ALU's

  $\mathbf{R}_{\mathrm{GB}_p}$**(3):** primary argument to w-bit ALU

  $\mathbf{R}_{\mathrm{GB}_s}$**(3):** secondary argument to w-bit ALU

  $\mathbf{R}_{\mathrm{GB}_r}$**(3):** destination for w-bit result

  RRC(2): input to refresh register

  WR(1): write refresh register to memory

  IOC(1): write refresh register to IO register

  ADDR($\lg(n)$)**:** memory address

- 1-bit Instruction:

  BOP(4): 1-bit operation

  $\mathbf{R}_{\mathrm{BB}_0}$**(3):** first argument to 1-bit ALU

  $\mathbf{R}_{\mathrm{BB}_1}$**(3):** second argument to 1-bit ALU

  $\mathbf{R}_{\mathrm{BB}_2}$**(3):** destination for 1-bit result

  $\mathbf{R}_{\mathrm{AC}}$**(3):** input to AC

Figure 14: Instruction Fields

3.3 The Tree Processor Component

The tree processor is conceptually a single large ALU that supports segmented partial scalar and vector global operations. The processor is organized as a binary tree of identical cells which compute locally and communicate with parent and children. The two major components of the tree processor, control and data, are distributed throughout the tree. The data processor is essentially a combinational circuit; however, in order to reduce circuit complexity as well as provide for extended precision operations, state information is maintained within the data component of the tree processors.

Circuit complexity is controlled by conceptually dividing the data component into two parts and using a single component for both phases. Consequently, a parallel prefix operation, such as all partial sums, requires two passes through the tree. The first phase, termed up, moves information *up* the tree storing results one per tree cell while the second phase, termed down, uses this stored information and produces the final result at the leaf cells. The data component is composed a w-bit ALU, three (w+1)-bit MUXes, one (w+1)-bit latch, and several tri-state drivers.[16]

The control component is purely combinational employing separate circuits corresponding to the up and down phases of the data path. Each tree cell's control component accepts seven 1-bit inputs and 1 encoded input that specifies the desired operation. It produces four 1-bit outputs that describe the segmentation among and participation of leaf cells in the global computation as well as an encoded output specifying which operation the w-bit ALU should perform. Figure 15 provides a high-level view of the interconnection between these tree cell components.

3.3.1 Control Component

The control component of the tree processor is a purely combinational circuit distributed across the tree cells. This component determines segmentation among and participation of leaf cells in the global computation and uses this information to control the operation of each tree cell's ALU.. The results are functionally dependent on the two 1-bit registers, SEG and VLD$\uparrow$, in each leaf cell, and the 1-bit root register VLD$\downarrow$. The design of the control component is similar in style to a carry-lookahead circuit but with the additional requirement that the propagated signals may move either from left-to-right or right-to-left with left-to-right propagation used for prefix operations and right-to-left for suffix operations. To avoid confusion, the following discussion uses the term downstream and upstream to refer to the relative location of data. For prefix operations, upstream can be read as to-the-left and downstream as to-the-right; for suffix operations reverse these directions.

The control component is distributed among the tree cells with each cell accepting segment information from its two children and activity information from its children and parent. Each cell produces segment information that is routed to its parent and activity information

[16] We expect the w-bit processor and (w+1)-bit MUXes and latches to be 32-bits and 33-bits wide, respectively; however, our design is not width restricted but parameterized as a function of word width.

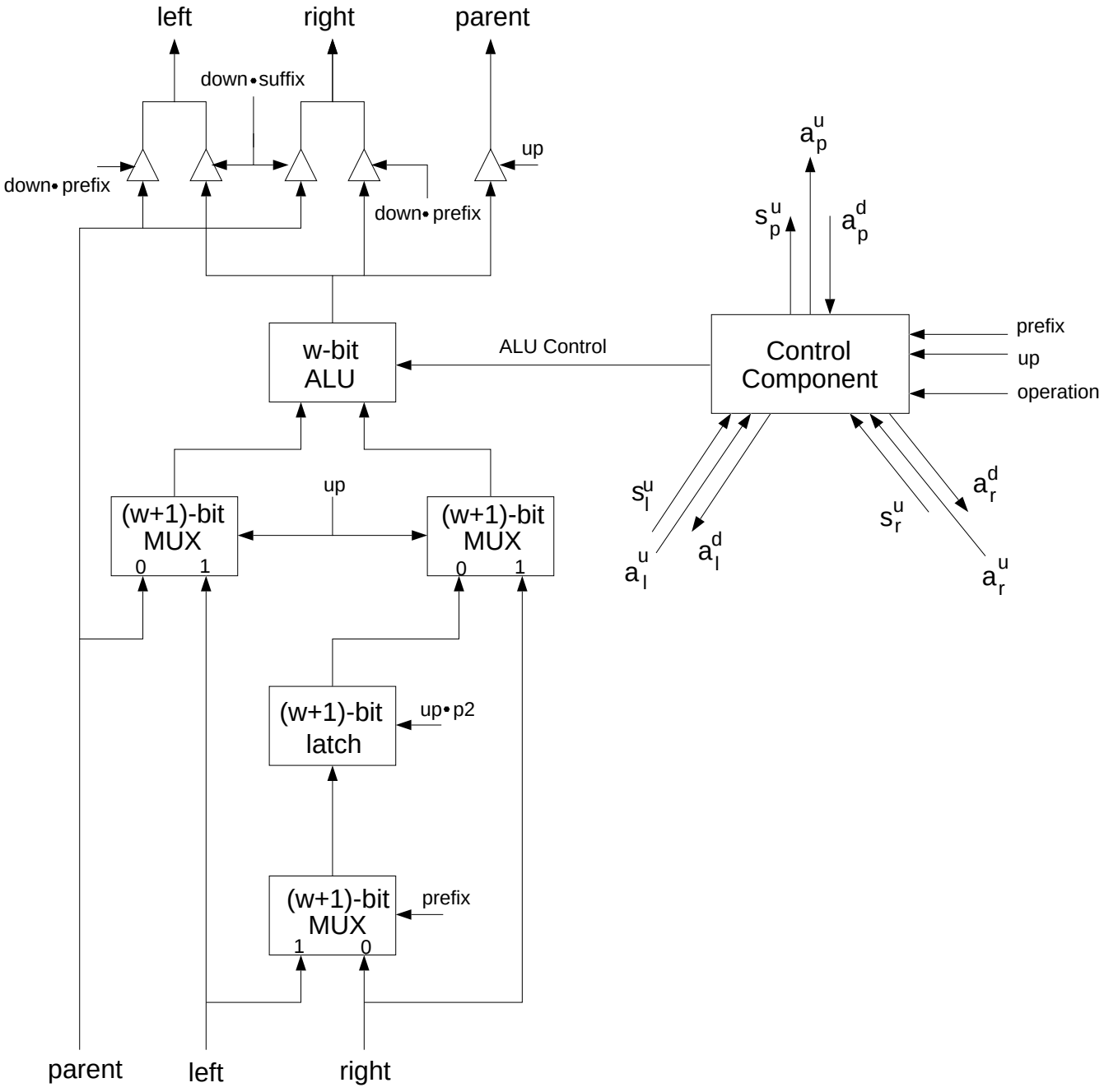

Figure 15: Tree cell architecture

that is routed to its parent and children. These signals are described in Figure 16.

3.3.2 Data Path component

The Data Path component is used to process, store and route the data during the up and down phases. It is connected to its parent and children using bidirectional busses that provide the pathways for the data values involved in the computation.

During an up phase it accepts inputs from its children, stores information from the upstream subtree in its local register, and computes and outputs results to its parent. The results output are determined by the globally specified tree operations and mediated by the control component based on segmentation and participation information.

During a down phase the Data Path component accepts inputs from its parent and local register, passes the upstream information coming from its parent to its upstream subtree, and computes and outputs results to its downstream subtree. The results output are determined by the globally specified tree operations and mediated by the control component based on segmentation and participation information.

In the current design the w-bit ALU supports eight different functions (i.e., MIN, MAX, SUM, AND, OR, XOR, LEFT, RIGHT,) allowing twos complement and positive integer data types for MIN, MAX, and SUM. Of these operations, all except LEFT and RIGHT are commutative allowing the inputs to the ALU to be presented on either input. Since LEFT and RIGHT are not commutative they require special treatment which is provided by a compiler or assembler. These software systems select between LEFT and RIGHT, not only based on the user specified tree operation but also with the knowledge of the data paths supported in the tree cell. Without this *division of labor* the tree cell would require a crossbar at the input to the w-bit ALU.

3.3.3 Overflow in the Tree Component

When used in a plus-scan or plus-reduction the tree component acts as a massively parallel processor and consequently can generate overflows within each tree cell. In order for the user to determined if an overflow has occurred, it is necessary to provide this overflow information both at the root of the tree and at the leaf cells. This is done by extending the data paths by 1 bit and using this line to indicate if the data associated with it has or has not overflowed. The w-bit ALU supports this feature by ORing and its two input overflows with the overflow produced by the ALU's computation to produce the output overflow.

Since overflow can be produced within a tree cell and since twos complement and positive integer have different overflow conditions it is necessary to either provide two separate overflow conditions or to specify the data type for the tree operation SUM - we chose to do the later.

- Input Signals:

  s_l^u - Segment information from left subtree. Signal has value 1 iff a new segment begins in left subtree.

  s_r^u - Segment information from right subtree. Signal has value 1 iff a new segment begins in right subtree.

  a_l^u - Participation information from left subtree. Signal has value 1 iff data contained in most downstream segment of the left subtree contributes to results downstream of the tree rooted at this node.

  a_r^u - Participation information from right subtree. Signal has value 1 iff data contained in the most downstream segment of the right subtree contributes to results downstream of the tree rooted at this node.

  a_p^d - Participation information from parent. Signal has value 1 iff upstream data contributes to result in the tree rooted at this node.

  prefix - Global information specifying the *type* of the tree operation. Signal has value 1 iff the operation is prefix, 0 if the operation is suffix.

  up - Global information specifying the direction of the tree operation. Signal has value 1 iff the operation is moving data up the tree, 0 if data is moving down the tree.

  operation - Global information that specifies the operation to be performed by the tree ALU.

- Output Signals:

  s_p^u - Segment information to the parent. Signal has value 1 iff a new segment begins in this tree.

  a_l^d - Participation information to the left subtree. Signal has value 1 iff data upstream of the left subtree contributes to the result in the left subtree.

  a_r^d - Participation information to the right subtree. Signal has value 1 iff data upstream of the right subtree contributes to the result in the right subtree.

  a_p^u - Participation information to the parent. Signal has value 1 iff data in the most downstream segment of the tree rooted at this node contributes to results downstream of the tree.

  ALU control - Local information that determines which function the w-bit tree ALU performs. For unsegmented total operations (i.e., operations in which every leaf participates) this signal is identical to the input signal labeled operation. For partial, segmented operations this signal may override the operation specification causing the w-bit ALU to act as a MUX routing either its left or right input to its output.

Figure 16: A Tree Cells Control Component

3.3.4 Two Dimensional Ripple

The efficiency of this architecture is critically dependent on the time required to move information through the tree. The delays associated with this information movement come from two sources, 1) the time to move information between levels in the tree, and 2) the time to ripple information across a word within a level.

By designing the architecture to simultaneously move information both across levels of the tree and across bits of a word, performance can be greatly improved with little additional hardware cost. Consider an n leaf tree (i.e., $\lg(n)$ levels) with word width w that uses ripple carry adders in each tree cell. If information propagation is required to complete within each word of a level before it proceeds to the next level, the time required for a SUM operation over the entire tree is $w \cdot \lg(n)$. If information is allowed to propagate in parallel both within and across levels this time is reduced to $w + \lg(n)$. Supporting two dimensional ripple is extremely simple if the only operation is SUM; however, when MIN is also supported a design choice must be made that balances the performance of MIN against the additional circuitry required to improve its performance.

A straight forward approach that supports the MIN function is to compare the two inputs using subtraction and then to route the smaller to the output. However, if the SUM circuitry is used for a subtraction based comparison, the comparison will require w time at each level before it can output the first bit of the result. Consequently, $w \cdot \lg(n)$ time will be required for a MIN operation over the entire tree. This can be improved using a similar approach to that previously mentioned for SUM; however, in this case the information ripple within a level is from the high order to the low order bit - opposite that required by SUM.

The CAM2000 architecture implements MIN as a high-to-low rippling function thus enabling two dimensional ripple in the tree and reducing the time required for MIN over the entire tree to $w + \lg(n)$[17]. In addition, and due to the representation of twos complement and positive integer forms, MIN must be aware of the data type of its inputs. Consequently, when the tree is asked to perform a SUM, MIN, or MAX the data type of the operation must also be specified.

Optimal is not much better

We have compared our design choices and the resulting performance with theoretically optimal circuits and found that our design provides an attractive balance between hardware cost and performance. Where a straight forward approach would provide SUM, MIN, and MAX in a time of 320 (i.e., $32 \cdot \lg(1024)$) our design requires a time of only 42 (i.e., $32 + \lg(1024)$), while an optimal circuit would require a time of 15 ($\lg(32) + \lg(0124)$).

In addition we have considered practical circuits based on optimal asymptotic designs (e.g. carry-lookahead and carry-free circuits) and found that our design uses significantly less circuitry and performs nearly as well as these asymptotically optimal designs. In fact, for values of word size and number of processors near our target design the two dimensional

[17] MAX has been implemented similarly and shares circuitry with MIN.

ripple is superior in both cost and performance to their *optimal counterparts.*

3.3.5 Extended Precision

Extended precision operations are performed using software emulation in both the tree and leaf nodes. The leaf node uses *standard* techniques such as add-with-carry; however, due to the number and organization of the processors in the collection tree specialized hardware is required to provide efficient support for software emulation of extended precision.

This hardware is distributed throughout the tree nodes requiring each tree node to record the status of any operations that can be extended (i.e., ADD, MIN, and MAX). Due to the split phase nature of the tree, up-status and down-status are maintained separately. When an extended operation is performed, each tree processor *seeds* itself with the status from the previous computation and records the new status for the next computation. This approach is similar in concept to the add-with-carry approach employed in the leaf node and combined with the distribution of status information throughout the tree provides global extended precision operations that are as efficient as their local counterparts.

3.3.6 Global Operations - Hardware Implementation

These operations require data to be processed in the collection tree. As previously noted, the technique we have adopted performs a scan in two phases; an up phase during which data is processed, stored, and propagated up the collection tree; and a down phase during which the stored data and a value injected at the root are processed and propagated down the collection tree.

Total Unsegmented Scan Operations

During each phase of a scan, tree cells computes two values in parallel determined by the type of scan (i.e., up or down) and the operation broadcast to the cells. During the up phase each tree cell stores the data from its upstream subtree in its internal register, applies the specified operation to the data from its children, and routes this result to its parent. The down phase works similarly; each tree cell routes the upstream data from is parent to its upstream subtree, applies the specified operation to the data from its parent and internal register, and routes the result to its downstream subtree.

Figure 17 provides a detailed example showing the data transmission and storage of the two phases of an unsegmented parallel prefix exclusive PLUS operation without activity control(i.e., all cells participate). In this example each tree cell is shown as an oval, the value stored in a cells internal register during the up phase are shown inside the oval, values transmitted during the up phase are represented by integers placed at the top of an edge, and values transmitted during the down phase are represented by integers placed at the bottom of an edge. The tree cells contain the input data used by the tree component as well as the results computed by this component. Notice that the up phase produces an output value

from the tree of 44 and that the down phase excepts an input value of 0 to the tree.

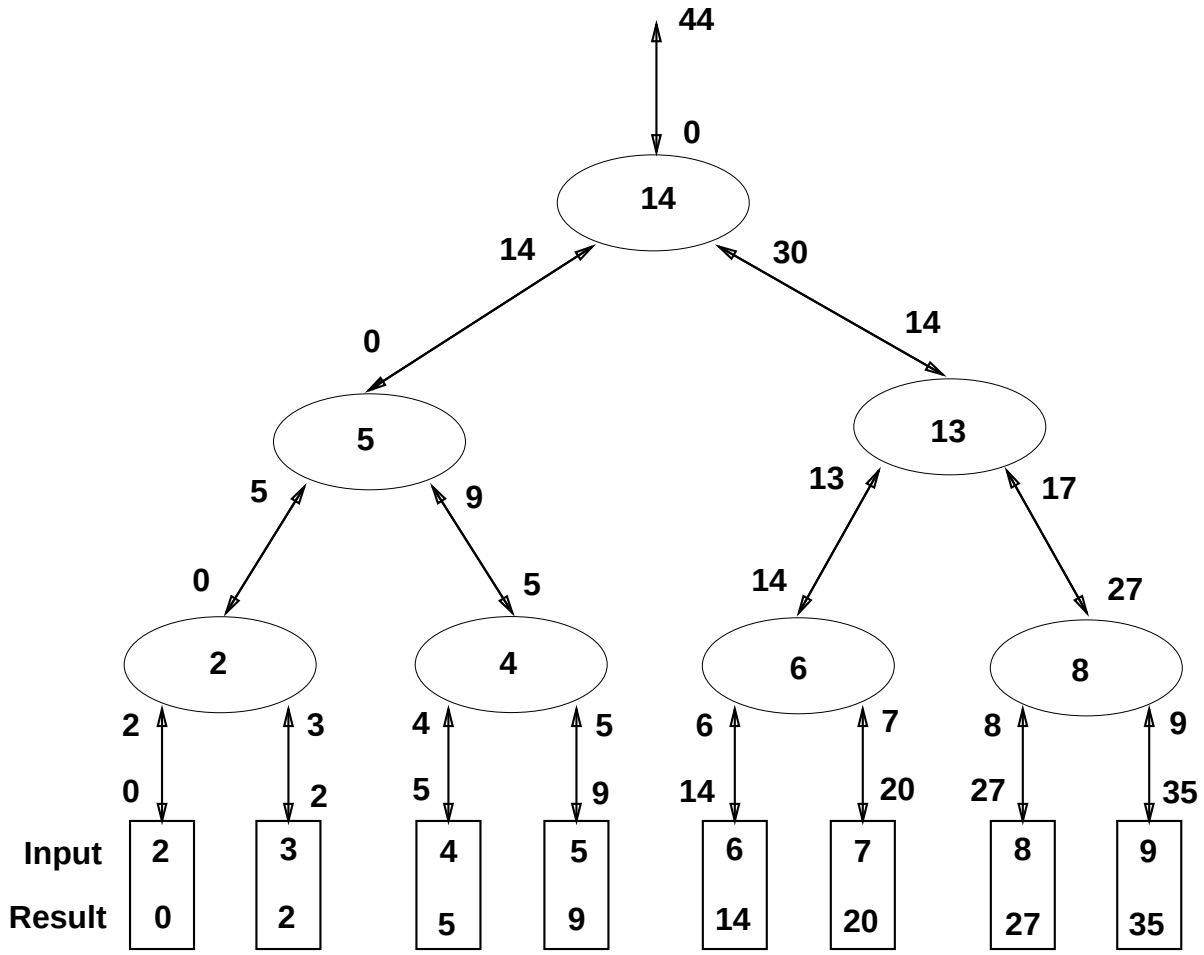

Figure 17: Computation and Communication for both phases of a plus scan

Partial Unsegmented Scan Operations

When a partial unsegmented scan is performed some leaf cells are disabled and the computation and communication of data in the collection tree must change. One way to accommodate this change without altering the operation of the tree cells is to preprocess the data in the leaf cells replacing inactive date with the tree operation's identity element. Once this is done the tree operation can be performed as though all leaf cells were active. This approach permits CAM cells to appear disabled in a partial unsegmented scan; however, it does not generalize to allow segmented scans or operations with no identity element (e.g., LEFT or RIGHT). An alternative approach adopted in the CAM2000 architecture, is to pass activity control information up and down the collection tree. This approach requires additional hardware in each tree node but provides uniform support for total and partial operations as well as segmented and unsegmented operations.

Partial Segmented Scan Operations

Partial Segmented operations cover all forms of global operations. They require each tree cell to be aware of the segment and activity relationships that exists among the input data. For example, during the up phase of a plus scan the data arriving from the children must be added together if the children belong to the same segment but must not be added if the children belong to different segments. The necessity for a tree cell to react to segmentation information requires that the segment bits stored in the CAM cells be transmitted to and within the collection tree.

This architecture implements segment computation and communication with dedicated hardware in each tree cell. The hardware accepts segment data from its two children, computes its own segment data, and passes this result to its parent. This representation provides a common semantics for segment information across CAM and tree cells. CAM cells store this information in their segment bit while tree cell compute this information as the **OR** of their children's segment data.

Activity information must also be propagated within the collection tree; however, in contrast to segmentation information that only is passed up the tree, activity information must be passed both up and down the tree. The activity information is encoded as a 1-bit field that is conceptually attached to each data value passed in the tree. A value of 1 indicates that the data is significant and must be used in the computation while a value of 0 indicates that the data is irrelevant and should not be used.

If all scans were unsegmented, activity information could be handled in the same manner as segment information with the exception that activity information must also be propagated down the tree. However, segmented scans cause the computation of activity information to be more complex. Activity information, as was the case with segment information, can be given unified semantics that encodes activity in a single bit field indicating if the associated data is relevant. Figure 18 shows the relationship between the activity and segment bits for partial segmented scans. Subscripts l, r, and p denote information about the left child, right child, and parent of a node while superscripts u and d denote whether the activity information pertains to the up or down phase of a scan [18].

Figure 15 on page 24 shows the control component that communicates the segment and activity information to a tree cell. These components compute segment and activity as specified in Figure 18 and transmit this information as indicated by the edges of Figure 15. The segment and activity control over the tree as a whole is determined by this hardware, the segment and activity bits in the CAM cells, and the activity bit, a^d, introduced at the root of the collection tree. Recall that the signal prefix is 1 if the left subtree is upstream of the right subtree and it is 0 if the right subtree is upstream of the left subtree.

Data Computation and Communication:

Once each tree cell has established its segment and activity information the data for the

[18] Both up and down activity information are computed in parallel using combinational circuits. Only the data paths require separate up and down phases.

$$
\begin{aligned}
s_p^u &\leftarrow s_l^u \vee s_r^u \\
a_p^u &\leftarrow \text{prefix} \wedge [a_r^u \vee (\overline{s_r} \wedge a_l^u)] \quad \vee \quad \overline{\text{prefix}} \wedge [a_l^u \vee (\overline{s_l} \wedge a_r^u)] \\
a_l^d &\leftarrow \text{prefix} \wedge a_p^d \quad \vee \quad \overline{\text{prefix}} \wedge [a_r^u \vee (\overline{s_r} \wedge a_p^d)] \\
a_r^d &\leftarrow \text{prefix} \wedge [a_l^u \vee (\overline{s_l} \wedge a_p^d)] \quad \vee \quad \overline{\text{prefix}} \wedge a_p^d
\end{aligned}
$$

Figure 18: General left-to-right segment and activity

corresponding operation is processed and routed. Figures 19 and 20 show the computation performed by a TREE cell as a function of the activity and segment information during the up and down phases. These functions have been developed using the activity and segment specifications to identify don't care situations which in turn have been used to reduce circuit complexity. A careful inspection of these functions will reveal that they identify what must be computed when a result is required but produce unpredictable, and sometimes conflicting, values when no result is required.

$$
v_i \leftarrow \begin{cases} v_l & \underline{\text{prefix}} \\ v_r & \overline{\text{prefix}} \end{cases}
\qquad
v_p \leftarrow \begin{cases} v_l & \overline{a_r^u} \vee (s_l^u \wedge \overline{\text{prefix}}) \\ v_r & \overline{a_l^u} \vee (s_r^u \wedge \text{prefix}) \\ v_l \diamondsuit v_r & a_l^u \wedge a_r^u \wedge ((\text{prefix} \wedge \overline{s_r^u}) \vee (\overline{\text{prefix}} \wedge \overline{s_l^u})) \end{cases}
$$

Figure 19: Datapath routing for up phase of a partial segmented operation

$$
v_l \leftarrow \begin{cases} v_p & \text{prefix} \vee (\overline{\text{prefix}} \wedge \overline{a_r^u}) \\ v_i & \underline{a_p^d} \\ v_p \diamondsuit v_i & \overline{\text{prefix}} \wedge a_p^d \wedge a_r^u \end{cases}
\qquad
v_r \leftarrow \begin{cases} v_p & \overline{\text{prefix}} \vee (\text{prefix} \wedge \overline{a_l^u}) \\ v_i & \underline{a_p^d} \\ v_p \diamondsuit v_i & \text{prefix} \wedge a_p^d \wedge a_l^u \end{cases}
$$

Figure 20: Datapath routing for down phase of a partial segmented operation

Figures 18, 19, and 20 provide a complete description of how the TREE cells implement a general left-to-right segment scan with activity control. Scans without activity control as well as unsegmented scans are specialized version of the general scan and can be performed by setting the CAM cell's segment and activity bit as required.

References

[Blelloch, 1987] Blelloch, G. (1987). Scans as primitive parallel operations. In *Proceedings of the 15th International Conference on Parallel Processing*, pages 355–362, University Park, PA. Pennsylvania State University Press.

[Blelloch, 1990] Blelloch, G. (1990). *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, MA.

[Hennessy and Patterson, 1990] Hennessy, J. L. and Patterson, D. A. (1990). *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA.