# SSS Strawman Architecture

Bill Dally Ujval Kapasi Mattan Erez Ben Serebrin Timothy Knight
Jung Ho Ahn

Version 0.61
April 18, 2002

# Contents

**1 Introduction** **5**
1.1 Document Structure . . . 5

**2 System Overview** **5**
2.1 System Architecture . . . 5
2.2 Stream Processor Architecture . . . 6
2.3 Outline of Operation . . . 7
2.4 Programming the SSS . . . 7
2.5 Multiuser / Multiprogramming Support . . . 8

**3 Stream Instruction-Set Architecture** **8**
3.1 Machine State . . . 8
3.1.1 Scalar Program Counter (PC) . . . 8
3.1.2 Scalar Processor Registers (SPR) . . . 8
3.1.3 Stream Register File (SRF) . . . 9
3.1.4 Stream Descriptor Registers (SDR) . . . 9
3.1.5 Memory Address Registers (MAR) . . . 9
3.1.6 Memory Stream Control Registers (MSCR) . . . 9
3.1.7 Stream Cache (SC) . . . 9
3.1.8 Global Memory (M) . . . 9
3.1.9 Segment Registers (SEG) . . . 9
3.1.10 Synchronization Primitive State (SYNC) . . . 9
3.2 Instruction Set . . . 9
3.2.1 Scalar Instructions . . . 9
3.2.2 Stream Load and Store . . . 9
3.2.3 Stream Cache Prefetch . . . 10
3.2.4 Invalidate Cache Entries . . . 10
3.2.5 Execute Kernel . . . 10
3.2.6 Global Sync . . . 10
3.2.7 Segment Register Manipulation . . . 10
3.3 Memory Model . . . 10
3.4 Global Communication and Synchronization Mechanisms . . . 12
3.5 Exception / Interrupt Handling . . . 12

**4 Kernel Instruction-Set Architecture** **12**
4.1 Machine State . . . 12
4.1.1 Cluster Local Registers (LR) . . . 12
4.1.2 Cluster Scratchpad Register File (SP) . . . 12
4.1.3 Cluster Condition Code Registers (CCCR) . . . 13
4.1.4 Microcontroller Register File (MCR) . . . 13
4.1.5 Microcontroller Condition Code Registers (MCCR) . . . 13
4.1.6 Microprogram Counter (MPC) . . . 13
4.1.7 Microcode Store (MCS) . . . 13
4.2 Microcode Instruction Execution . . . 13
4.3 Conditional Execution . . . 14

**5 Microarchitecture** **14**
5.1 Scalar Execution Unit . . . 14
5.1.1 Scalar Processor (SP) . . . 14
5.1.2 Stream Controller (SCO) . . . 14
5.1.3 Processor-Network Interface (PNI) . . . 14
5.1.4 Programmable Interrupt Controller (PIC) . . . 16
5.2 Stream Execution Unit . . . 16

5.2.1 Arithmetic Cluster (CL) . . . 16
5.2.2 Microcontroller (MC) . . . 16
5.2.3 Inter-cluster Switch (ICS) . . . 16
5.3 Stream Register File (SRF) and Stream Buffers (SB) . . . 16
5.4 Memory System . . . 18
5.4.1 Address Generators (AG) . . . 18
5.4.2 Reorder Buffers (ROB) . . . 18
5.4.3 Stream Cache (SC) . . . 18
5.4.4 DRAM Interface (DI) . . . 19
5.4.5 Memory-Network Interface (MNI) . . . 19
5.4.6 Processor-Memory Interface (PMI) . . . 19
5.4.7 Memory Unit (MU) . . . 19
5.5 Network and Network Interface (NI) . . . 20

**6 Feeds and Speeds 20**
6.1 Speeds within the Node . . . 21
6.1.1 Local Bandwidth . . . 21
6.1.2 Stream Register File . . . 21
6.1.3 Cache . . . 21
6.1.4 DRDRAM Local Memory . . . 21
6.2 Speeds across the system . . . 21
6.2.1 Card-level Speed . . . 21
6.2.2 Backplane Speed . . . 21
6.2.3 System Speeds . . . 21

**7 Area and Power Estimates 22**

**8 Parameters and Ranges 22**

**9 Issues 22**
9.1 Open . . . 22
9.1.1 Virtual Memory . . . 22
9.1.2 Memory Coherence . . . 22
9.1.3 Dependency Information . . . 22
9.1.4 Aspect Ratio and Layout of Execution Units . . . 22
9.1.5 Conditional support . . . 23
9.1.6 Inter-cluster Switch Implementation . . . 23
9.1.7 Register File Organization . . . 23
9.1.8 Integer Unit in Clusters . . . 23
9.1.9 Memory Locking . . . 23
9.1.10 Number of Stream Buffers . . . 23
9.2 Closed . . . 23
9.2.1 SRF, SB, SC . . . 23
9.2.2 Stream Cache . . . 23
9.2.3 Parallelism Within the Node . . . 23
9.2.4 Fetch-and-Op . . . 24
9.2.5 Inter-cluster Switch Placement . . . 24
9.2.6 Memory Segment Sizes . . . 24
9.2.7 Support for Debugging of Software . . . 24
9.2.8 General Purpose Timers . . . 24
9.2.9 Multicasting . . . 24
9.2.10 SIMD Floaing Point Units . . . 25
9.2.11 L2 Cache . . . 25
9.2.12 SRF Random Access . . . 25
9.2.13 Multidimensional Strides . . . 25

# List of Figures

1 SSS system architecture . . . 5
2 SSS node architecture overview . . . 6
3 Language heirarchy . . . 8
4 Mapping virtual addresses to physical addresses, assuming that there are at most $2^{14}$ nodes, each node has $2^{35}$ bytes of local memory, and there are 32 segment registers. . . . 11
5 SSS node block diagram . . . 15
6 Cluster block diagram . . . 17

# List of Tables

1 Memory unit requesters . . . 20
2 Memory unit responders . . . 20
3 SSS parameters . . . 22

# 1 Introduction

The SSS is intended as a cost-effective target for numerical applications expressed as stream programs in a language such as Brook [6]. The goal is to achieve a cost/performance ratio 100x better than conventional cluster based supercomputers on both arithmetic limited and memory-bandwidth limited computations.

The SSS exploits the stream model [5] to expose the locality available in an application, uses a stream architecture [1] to exploit this locality, and provides a large number of floating-point units per node to convert the resulting high-level of arithmetic intensity into performance.

The latency hiding of stream memory operations in combination with a high-bandwidth memory system achieves cost effectiveness on memory-limited applications.

## 1.1 Document Structure

Section 2 gives a high-level description of the SSS, covering the various aspects of the system breifly. Sections 3 and 4 present the programmer's model of the machine, and 5 specifies the SSS architecture. Sections 6 and 7 summarise the speeds of various components of the SSS and present estimates of the area and power consumption of the chip specified in 5. Section 8 enumerates various parameters of the SSS which will be tuned based upon the results of simulations, and section 9 lists all of the open issues related to this document.

# 2 System Overview

## 2.1 System Architecture

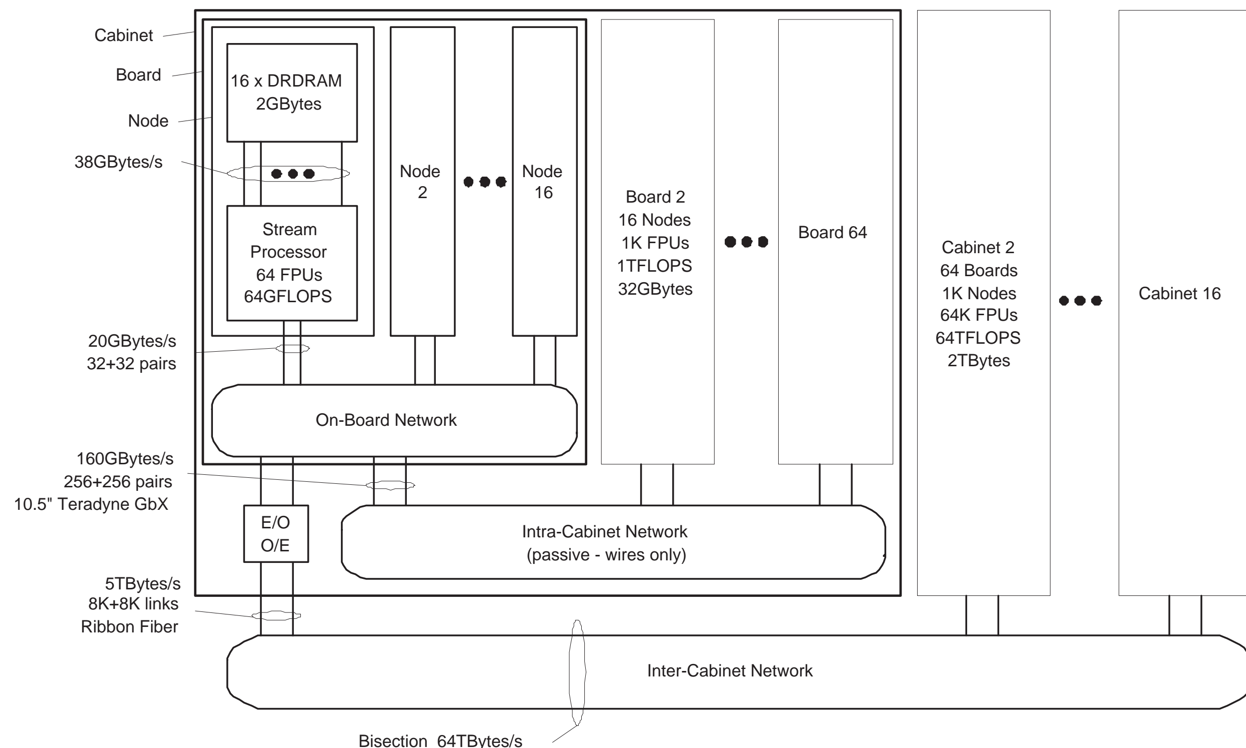


Figure 1: SSS system architecture

The SSS is a parallel stream processor with a shared memory [2]. Figure 1 illustrates the system architecture of the SSS, along with some bandwidth and performance targets. From the perspective of the programming system, the SSS is a collection of nodes, each consisting of a stream processor and a portion of the global memory; a node can reference any location in global memory but has higher bandwidth to its own portion.

## 2.2 Stream Processor Architecture

Figure 2: SSS node architecture overview

The basic architecture of a stream processor (node) is illustrated in figure 2. There are five main components.

- The scalar execution unit contains a scalar processor that executes standard scalar instructions (e.g. MIPS) in addition to stream instructions which configure and control the other units. The stream instructions, from the scalar processor's point of view, are simply writes to registers in the SSS blocks, and are interleaved with normal scalar instructions in the scalar program being executed.
- The stream execution unit consists of a microcontroller and a set of parallel clusters, each of which contain several arithmetic units and local register files. The microcontroller executes a sequence of VLIW microcode instructions that specify how the clusters should be controlled.
- The memory system handles requests for data which may be in a local DRAM, may be cached locally, or may be stored in a remote node's memory. It controls the transfer of data between the various levels in the memory heirarchy.
- The stream register file is a staging area for the clusters. The aim of the stream architecture is to be able to keep the clusters busy using just the data in the stream register file for long enough that the data transfers between the SRF and global memory are hidden.
- The network serves two functions: it provides the appearance of a single large shared global memory, by forwarding requests by the memory system of one node for data stored on another, and it enables signalling between the scalar processors on each node. The network guarantees delivery of packets.

The SSS exploits parallelism at three levels. At the outermost level the nodes execute in parallel, each operating on its portion of a set of input streams to generate its portion of a set of output streams. These

streams need not be entirely local to the node, but performance will benefit if they are. The nodes communicate and synchronize with one another at the end of each parallel stream operation.

At the middle level, the clusters of the stream processor operate on several adjacent elements of the local portion of the stream simultaneously. This inner-loop parallelism exploits the on-chip communication that exists within a single node.

Finally, at the innermost level, the floating-point units within each cluster exploit instruction-level parallelism.

## 2.3 Outline of Operation

The following is a high level description of the intended operation of the SSS.

- The compiler takes a user application and outputs a scalar program and a set of microcode programs (kernels). The scalar program, which will run on the scalar processor in each node, the microcode for each kernel, and the program data are loaded into global memory, and the scalar processors begin execution. The program data streams will have been partitioned such that the elements assigned to each node exhibit locality of reference (e.g.) neighboring points in a grid or mesh; this is called domain decomposition.

- A stream load instruction in the scalar program running on a node causes some data in the global memory to be fetched by the memory system and copied into the SRF on that node.

- A kernel load instruction in the scalar program causes some microcode to be copied from the SRF into the microcontroller's program memory, and a subsequent kernel execute instruction causes the microcontroller to begin executing the sequence of instructions.

- The microcontroller program is the kernel - it copies data between the SRF and the local register files in the arithmetic clusters and it controls the individual arithmetic units which process the data. Multiple stream elements are processed in parallel by the node's clusters, which execute the same microcode program in lockstep.

- The kernel computation produces a new stream of values, which are written to the SRF by the microcode program and then potentially copied back to the global memory by a stream store instruction in the scalar program. During the execution of the kernel the scalar program will handle transferring chunks of the data streams between memory and the SRF using streeam load and store instructions, attempting to keep the arithmetic units busy.

- If global synchronization is required, for example a barrier signifying the end of one global iteration and the beginning of a new one, the scalar program can send signals to other nodes and/or wait (stall) until it recieves a signal.

## 2.4 Programming the SSS

Figure 3 illustrates the heirarchy of languages used to program the SSS. At the highest level, the user writes applications either directly in Brook or in an application specific programming language which is converted to Brook. These Brook programs work at the level of streams and the kernels which operate on them; they don't deal with memory management (e.g. copying data between global memory and the SRF), nor do they worry about how the parallel clusters will be programmed.

At the lowest level, the user's program is compiled to a scalar processor binary (e.g. MIPS) and a set of microcode programs (kernels). The scalar program initiates transfers of stream data between the SRF and global memory, causes kernel programs to be run by the microcontroller, handles prefetching data into the cache as well as invalidating entries, and sends/receives synchronisation signals to/from the scalar processors in other nodes. The microcode programs, which are executed by the microcontroller, specify the kernel operations being performed on the stream data; they control the transfer of data between the SRF and the local register files, control the arithmetic units which operate on the data in parallel, and initiate inter-cluster transfers of data.

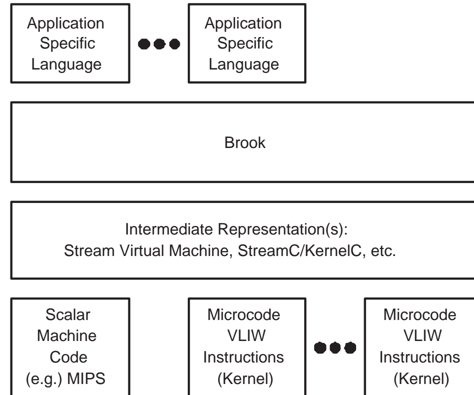


Figure 3: Language heirarchy

The compilation and runtime systems are expected to be able to perform the following transformations on the high-level program written in Brook:

- Partition the streams of data across nodes (domain decomposition). For example, if a program uses a $1024 \times 1024$ stream and runs on a 64-node SSS, the programmign system could assign a $128 \times 128$ submatrix to each node. Unlike domain decomposition on cluster machines, the submatrices on each node need not be overlapped since the node can reference data in the adjoining submatrices if it is needed.
- Schedule the arithmetic units and allocate local registers for the execution of kernels, including splitting or merging kernels to maximize the ratio of computation to memory operations.
- Schedule the transfers of stream data between the SRF and global memory, ensuring that the sizes of the transferred chunks don't cause the kernel to spill tremporary variables (strip-mining).
- Insert instructions in the scalar code to synchronize the nodes where necessary.
- Convert if-statements and conditional loops to either predicated execution or conditional streams.

## 2.5 Multiuser / Multiprogramming Support

Each node in the SSS can run at most one program at a time, and several nodes can be running the same program collaboratively. Different programs can be executing on different nodes, however, and these programs share the complete global memory space of all the nodes combined. Memory segmentation, specified via a set of segment registers, is used to protect different programs from each other.

# 3 Stream Instruction-Set Architecture

## 3.1 Machine State

Each stream processor operates on the following state.

### 3.1.1 Scalar Program Counter (PC)

Contains the memory address of the next instruction to be executed by the scalar processor.

### 3.1.2 Scalar Processor Registers (SPR)

The register set of the scalar processor - a standard MIPS or ARM set with some stream extensions.

### 3.1.3 Stream Register File (SRF)

A single $S_{SRF}$-word register file. Contiguous data can be transferred in parallel between the SRF and the stream buffers.

### 3.1.4 Stream Descriptor Registers (SDR)

The $N_{SDR}$ stream descriptor registers are used to hold descriptions of streams in the SRF.

### 3.1.5 Memory Address Registers (MAR)

The memory address registers hold descriptions of vector address streams for the memory system.

### 3.1.6 Memory Stream Control Registers (MSCR)

Each address generator contains a memory stream control register, which contains information about the stream the address generator is fetching or storing. When a stream instruction initiates a stream transfer via the address generators, the MAR with the stream info is specified along with details of the operation to perform, and this information is stored in the address generator's MSCR.

### 3.1.7 Stream Cache (SC)

A $S_{SC}$-word register file that acts as a special cache and is accessed with memory addresses. It is intended to support graph streams efficiently. It is exposed in the ISA but is not software managed.

### 3.1.8 Global Memory (M)

The contents of all of the memory in the machine. Physical memory addresses are partitioned into node and offset.

### 3.1.9 Segment Registers (SEG)

A set of $N_{SEG}$ special registers for implementing the memory model of section 3.3.

### 3.1.10 Synchronization Primitive State (SYNC)

State kept by the system to allow the implementation of the synchronization primitives described in section 3.4.

## 3.2 Instruction Set

The scalar processor in each node executes a single instruction sequence indexed by the program counter. The instruction set includes the following instuction types.

### 3.2.1 Scalar Instructions

A standard RISC instruction (e.g. MIPS). Note that from the perspective of the scalar processor, the other instructions in this list are all regular RISC instructions; they are implemented as writes to the stream controller.

### 3.2.2 Stream Load and Store

Copy a stream of records between the SRF and global memory. The records can be of arbitrary length and can be located in memory on a regular stride. Additionally, an index stream can be used to specify a list of addresses of memory locations to copy data to or from. Streams in both memory and the SRF/SC are identified by register-mapped stream descriptors (SDRs). Each stream operation specifies whether or not the stream cache (SC) may be used.

#### 3.2.3 Stream Cache Prefetch

Prefetches a set of records into the stream cache.

#### 3.2.4 Invalidate Cache Entries

Support for single line invalidates, as well as setting bits used for gang-invalidates [7].

#### 3.2.5 Execute Kernel

Execute a kernel (specified by a register-mapped kernel instruction pointer) on a set of streams (specified by register-mapped stream descriptors). Dependency information will be passed to the stream controller explicitly in the instruction(s) from the scalar processor.

#### 3.2.6 Global Sync

A set of instructions to set up and invoke the synchronization primitives described in section 3.4.

#### 3.2.7 Segment Register Manipulation

OS priviledged instructions for reading and setting the segment registers described in section 3.3.

### 3.3 Memory Model

To allow multiple jobs to be run on the SSS simultaneously - the intended mode is space shared - memory is protected via segmentation. The 64-bit virtual addresses used by the SSS are absolute across an entire process (a collection of nodes executing the same program), and segment registers in each node are used to convert the virtual addresses to 64-bit physical addresses which are absolute across the entire system. Note that although both the physical and virtual addresses refer to a byte offset in memory, they are restricted to being word aligned - the lower 3 bits of each are assumed to be 0 and are ignored by the memory system. Figure 4 illustrates the mapping process, which is summarized as follows:

- The *SegNum* field of the virtual address is used to index the segment register file.
- The $n_0$ and $n_1$ fields of the segment register are used to partition the *SegAddr* field of the virtual address into the physical memory offset (*PhysOffHi:PhysOffLo*) and the *NodeOff* field.
- The node to which the virtual address is referring is calculated by adding *NodeBase*, the start of the contiguous range of nodes in the segment, to *NodeOff*.
- The absolute physical address (*PhysAddr*) within the determined node (*Node*) is computed by adding *PhysOff* to *PhysBase*.
- The physical address must fall within the segment memory range in the node, namely *PhysBase* to $PhysBase + 2^{PhysLen} - 1$. Note that the minimum segment size allowed is $2^3$ bytes, (i.e.) *PhysLen* must be greater than or equal to 3, and the base physical address (*PhysBase*) must be an integer multiple of the segment size.

Within each segment, the virtual addresses (*SegAddr*) are contiguous and run from 0 up to $2^{PhysLen+x}-1$, where $x = n_1 - n_0$ and $2^x$ is the number of nodes in the segment. Physically, however, due to the way the virtual addresses are converted to physical addresses, the segment data may be interleaved across a number of nodes. For example, a segment register could specify that a segment over 16 nodes, starting at node 32 ($NodeBase = 32$), is 4 GBytes in size (total) and hence uses 256 MBytes from each node ($PhysLen = 28$), starts at physical address 1 GBytes within each node ($PhysBase = 2^{30}$), and is striped across the nodes in 1 MByte increments ($n_0 = 20$ and $n_1 = 24$).

The segment registers also specify the stream cache policy for their addresses via the *Cache* field. A segment which is marked as cacheable allows data to be cached (optional, depending on the instructions),

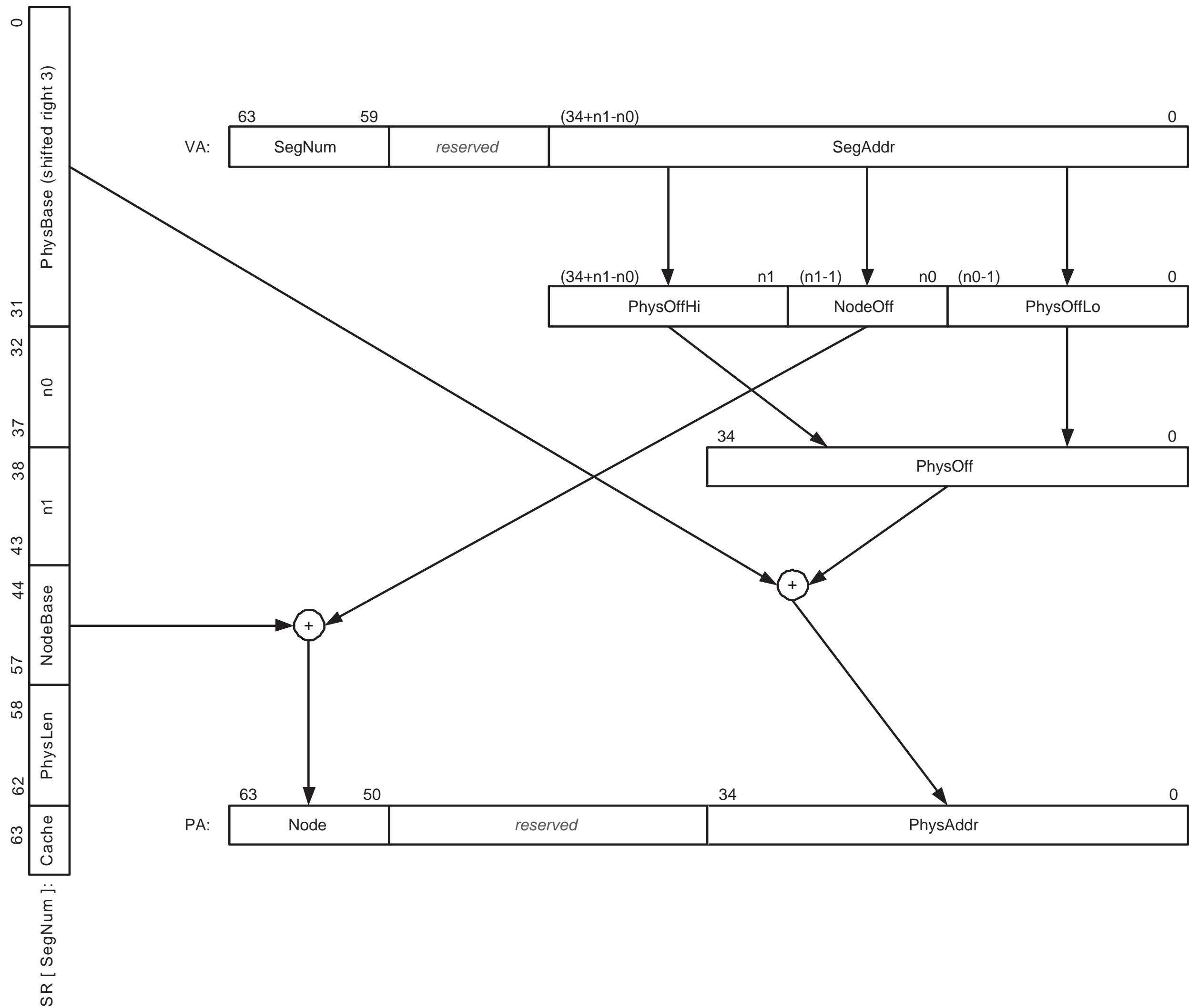


Figure 4: Mapping virtual addresses to physical addresses, assuming that there are at most $2^{14}$ nodes, each node has $2^{35}$ bytes of local memory, and there are 32 segment registers.

and a segment which is marked non-cacheable will never allow data to be cached, regardless of the instructions issued. The stream cache supports read-only semantics with explicit invalidation.

The scalar processor can issue both virtual and physical addresses; the processor-memory interface accepts a special "bypass address translation" bit in the address issued, and interprets the *SegAddr* field as simply the physical address within the current node. This feature is intended for OS use only, however, and is only enabled when the scalar processor is in privileged mode.

## 3.4 Global Communication and Synchronization Mechanisms

The scalar processor in each node can send and receive messages over the network to and from the scalar processors in other nodes. Packets which are related to memory requests and responses are handled by the memory-network interface. The messaging packets which are sourced/sinked by the processor-network interface are of the following types:

- **Fetch-and-Op:** A node can request that a remote node perform an operation on its local data, possibly returning the result. Each fetch-and-op request can carry an arbitrary word of data, allowing the efficient implementation of reduction operations. Supported operations with explicit hardware support are add-and-store, fetch-and-add, and compare-and-swap.
- **Barrier Signal:** The processors can signal when they reach a given point in their programs, and/or they can wait until they receive a signal from another node. The signal will refer one of a small set of named barriers, which could be a general global barrier or a specifically tuned pairwise barrier.
- **General Purpose Message:** The processors can also send messages to each other for user-defined purposes. An example of an application of these messages would be in the case of multiple nodes searching for a solution within their own local portion of the data; once one node finds a match, it can signal the other nodes to notify them to stop searching.

In addition to sending messages over the network, processors can communicate and synchronize with each other by using the global shared memory.

## 3.5 Exception / Interrupt Handling

Conceptually, kernel and scalar operations are atomic and exceptions can be checked after each stream operation. Error conditions which arise inside the cluster arithmetic units, such as divide-by-0, will cause a status bit to be set, and will possibly result in a NaN floating point value.

The scalar processor can be interrupted by both the stream controller and the processor-network interface. These interrupts are maskable, and the processor can poll a status register in each instead of being asynchronously interrupted if the programmer so desires.

# 4 Kernel Instruction-Set Architecture

This section presents the programmer-visible elements of the architecture which are related to the processing of stream elements within the clusters.

## 4.1 Machine State

### 4.1.1 Cluster Local Registers (LR)

Each of the arithmetic clusters contain a set of $S_{LR}$-word two-port local register files, one on each input of each functional (arithmetic) unit.

### 4.1.2 Cluster Scratchpad Register File (SP)

A $S_{SP}$-word indexable register file in each cluster that is used for temporary storage such as local register spills and lookup tables.

### 4.1.3 Cluster Condition Code Registers (CCCR)

Each arithmetic cluster also contains a condition code register which contains the usual arithmetic condition codes and is used for predicated execution as well as conditional streams.

### 4.1.4 Microcontroller Register File (MCR)

The $S_{MCR}$-word microcontroller register file can hold arbitrary data written by the microcontroller. Specifically, it is used to hold loop counters as well as data to be broadcast to some or all of the clusters. It can also be used to pass values from a microprogram to the scalar processor and vice versa. There is a single microcontroller register file in each node.

### 4.1.5 Microcontroller Condition Code Registers (MCCR)

The microcontroller condition code registers are used as branching conditions for loop operations in microprograms. There is a single microcontroller condition code register file in each node.

### 4.1.6 Microprogram Counter (MPC)

The microprogram counter is located in the microcontroller, and is simply a pointer into the microcode store which indicates the currently executing microcode instruction. There is a single microprogram counter register in each node.

### 4.1.7 Microcode Store (MCS)

The microcode store is a $S_{MCS}$-byte register file that holds the microcode instructions which are to be issued by the microcontroller to the arithmetic clusters. Microcode is loaded into the microcode store via the stream register file.

## 4.2 Microcode Instruction Execution

The microcontroller executes a single instruction sequence located in the microcode store. Each instruction is issued simultaneously to all clusters in the node, which execute it in lockstep. The only type of non-sequential control flow supported within the microcontroller is looping. Microcode instructions can specify the following operations:

- Select the operations performed by the arithmetic blocks inside the clusters.
- Configure the inter-cluster switch, controlling how data is routed between clusters.
- Cause the microcontroller to loop by adding an immediate value to the microprogram counter register. The branch to the top of the loop is conditional, based on the value of a microcontroller condition code register.
- Set a microcontroller condition code register, based on the test of a microcontroller register.
- Write an immediate value to a microcontroller register.
- Stall the microcontroller until a synch signal is received from the stream controller.
- Set a microcontroller condition code register, based on the test of any or all of the cluster condition code registers.
- Transfer streams of data between the SRF and the cluster registers.

## 4.3 Conditional Execution

The only conditional operation supported at the microcontroller level is looping, due to the fact that a single stream of microcode instructions is broadcast to all clusters within a node, which execute them in lockstep. At the cluster level, predicated execution is used to enable if-then-else constructs without branching; both halves of the if-then-else construct are executed in lockstep by all clusters, but the register write operations are only committed in either the if-block or the else-block on a per-cluster basis, depending on the result of the boolean condition evaluation.

Support for conditional streams [3] is also provided, by allowing the stream buffers to be independant across clusters and using the inter-cluster switch to route the data between the stream buffers and the clusters.

# 5 Microarchitecture

The top-level node block diagram is shown in figure 5. This section describes the functionality of each block.

## 5.1 Scalar Execution Unit

### 5.1.1 Scalar Processor (SP)

The scalar processor executes a standard RISC instruction set (e.g. MIPS). Stream instructions, as described in section 3.2, are implemented as writes to the stream controller block, and are interleaved with the 'scalar' instructions in a single logical thread of execution.

A minimal run-time OS will execute on the scalar processor. The OS is responsible for loading and executing the scalar programs and may be involved in the implementation of the synchronisation primitives discussed in 3.4.

The scalar processor has its own private I- and D-caches. It shares the same global memory space as the streaming portion of the architecture, into which it can make byte-wise memory references through the processor-memory interface.

### 5.1.2 Stream Controller (SCO)

The scalar processor issues stream commands to the stream controller, which performs the following tasks.

- Decodes the command from the scalar processor, which includes explicit dependency information.
- Keeps a scoreboard of pending instructions and their dependencies, as well as available resources. An instruction will be issued to the SSS blocks when its dependencies are met and the resources it requires are free.
- A mask of available scoreboard entries is provided to the scalar processor to enable the program to determine when the stream controller can accept a stream instruction.
- Optionally, the stream controller could interrupt the scalar processor when the scoreboard transitions from full to not full.

### 5.1.3 Processor-Network Interface (PNI)

The processor-network interface block is responsible for enabling the scalar processor to send and receive messages over the network. The types of message packets which can be sent and received are enumerated in section 3.4. The processor-network interface performs the following two tasks.

- The scalar processor can write a message-send command to the block, which will package the message into the network packet format and transmit it to the destination node.

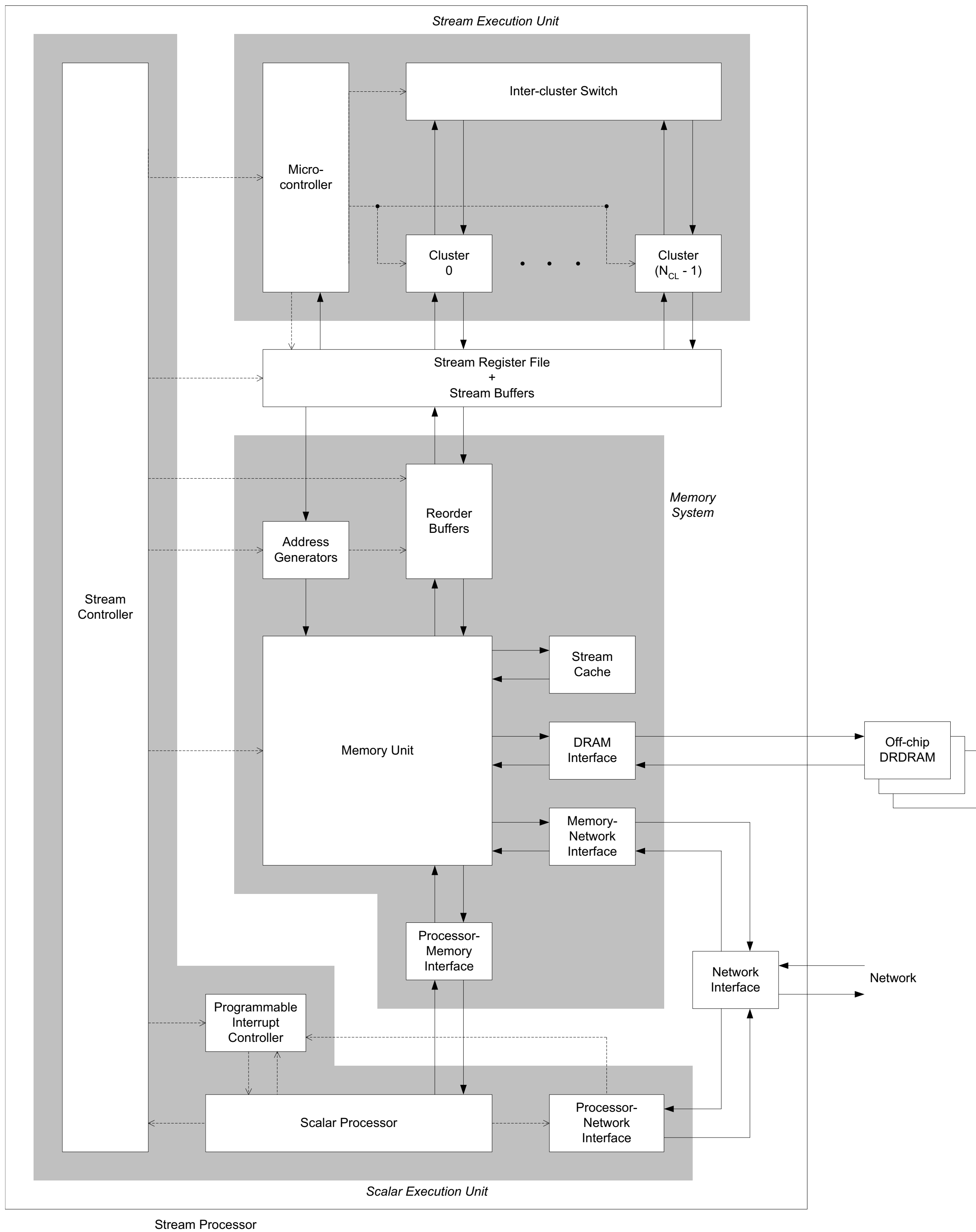


Figure 5: SSS node block diagram

- Upon reception of a message packet from another node, the block will decode the packet and store the message in any of several sets of registers which is readable by the scalar processor; multiple received messages may be outstanding at any given time. In addition, there is a maskable interrupt which may be asserted to notify the processor of the arrival of a message.

#### 5.1.4 Programmable Interrupt Controller (PIC)

The scalar processor can potentially receive interrupts from multiple sources, including the processor-network interface and the stream controller. This block would provide a simple interface for the scalar program to mask and unmask the individual interrupt sources, as well as providing a global interrupt enable/disable bit that the program could write.

The processor will receive status signals from various blocks in the chip, and these signals will be presented as status bits to the program. To ensure that the software developers and compiler writers using the SSS have maximum flexibility, these status bits should also generate maskable interrupts from the interrupt controller when they transition from 0 to 1. The status bit interrupts would be deasserted by the processor via a clear-on-write protocol.

### 5.2 Stream Execution Unit

#### 5.2.1 Arithmetic Cluster (CL)

Each node contains $N_{CL}$ identical arithmetic clusters which consist of several functional units, local register files, and an intra-cluster switch. Figure 6 illustrates the cluster architecture.

The functional units can only read from their own local register files, and the functional unit outputs are written back to one of the local register files in the cluster. The microcontroller is responsible for transferring data between the local register files and the SRF.

Each cluster consists of two double-precision floating point adders, two multipliers, and one divide/square root unit. Each local register file contains $S_{LRF}$ 64-bit entries. All functional units other than the divide/square root unit are fully pipelined.

In addition, each cluster contains a scratchpad which is a set of $S_{SP}$ 64-bit indexible registers used for temporary storage such as register spills and lookup tables. The scratchpad is controlled similarly to an execution unit and can process one read and one write per cycle.

#### 5.2.2 Microcontroller (MC)

The functional units, local register files, intra-cluster switches, and inter-cluster switch are controlled by the microcontroller, which executes a statically scheduled VLIW instruction sequence. The instructions, which constitute a kernel, are located in the microcode store. The microcontroller broadcasts the same instruction to all the clusters, which operate in a SIMD fashion.

The microcode program executed by the microcontroller is responsible for initiating data transfers between the local register files and the SRF, and also between the clusters via the inter-cluster switch. Transfers to and from the SRF are via the stream buffers, and the microcontroller will stall execution if either the data being read from the SRF is not yet available in the SB or if the SB is full and cannot accept the data being written to the SRF.

The microcontroller has a set of registers which can store loop counters and other values required by the microcode program.

#### 5.2.3 Inter-cluster Switch (ICS)

The inter-cluster switch is implemented as a crossbar, with each cluster able to write one word per cycle and read one word per cycle (throughput).

### 5.3 Stream Register File (SRF) and Stream Buffers (SB)

The stream register file is a $S_{SRF}$-word memory that is accessable by the arithmetic clusters, memory system (address generators) and scalar processor. The basic unit of access is a stream transfer, which is initiated

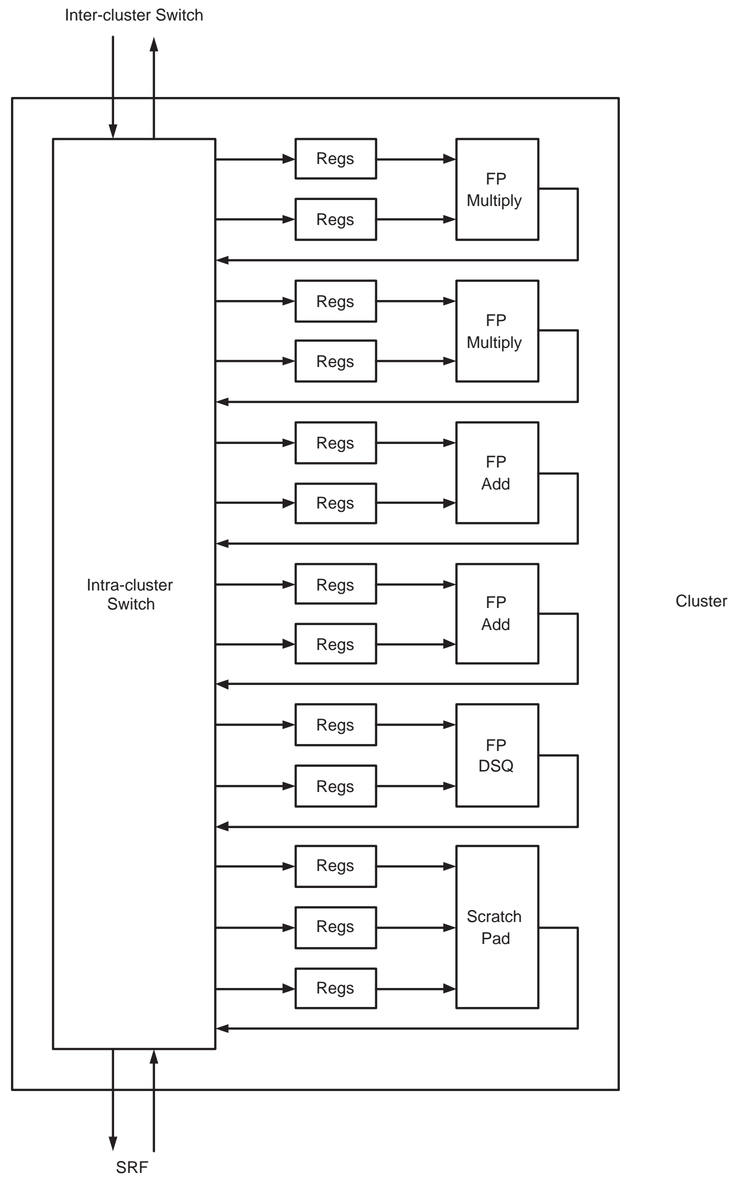


Figure 6: Cluster block diagram

by a stream instruction executed by the scalar processor. Each stream operation will specify a direction, a base address into the SRF, a count ($n$), and whether the count is variable. The direction parameter specifies whether the stream transfer is reading from or writing to the SRF core memory.

If the number of elements accessed is predictable then the stream operation will terminate after $n$ elements have been transferred; otherwise, if the number of elements is variable, then the count represents a maximum length. In the latter case, the stream operation will terminate either when the client asserts an end-of-stream (EOS) signal or if $n$ elements have been transferred. Accesses to the SRF are linear, with $n$ consecutive stream elements transfered starting at the base address.

A set of stream buffers handles all flow control and data transfers between the SRF core and clients. Stream buffer access to the SRF is time-multiplexed but very wide, avoiding the necessity of having a multi-ported SRF to support multiple clusters accessing it simultaneously. Essentially, each stream transfer is assigned to a particular SB, and the connectivity of SBs to clients is such that each client has its own dedicated set of SBs. The SBs logically function like a FIFO between the SRF and the client. The stream buffers are also used to implement conditional streams as described in [3].

Note that the stream buffers are not all identical; certain stream buffers can be configured to transfer data in both directions, while some are limited to only one. Additionally, the amount of bandwidth between the SB and the client is potentially different for each client.

The SRF is partitioned into multiple banks, and each cluster is only able to access data aligned to its partition; random access is not supported.

## 5.4 Memory System

### 5.4.1 Address Generators (AG)

Each node accesses memory by issuing stream load and store commands to the address generators. The role of the address generators is to issue single word memory requests to the memory unit. These are based either on a strided access by specifying a base, record size, and number of records, or on an indirect access via a stream of indices. The AGs also support bit-reversed addressing. There are $N_{AG}$ address generators, each of width $W_{AG}$, supporting a total of $N_{AG} \times W_{AG}$ word requests per cycle.

### 5.4.2 Reorder Buffers (ROB)

Along side the address generators is a set of memory reorder buffers that accept/supply data to/from the SRF, essentially acting as an indexable stream buffer. Each ROB holds the identifier of the stream it is buffering. When an AG sends a request to the memory system it first allocates an entry in the ROB and sends the entry number along with the request to the memory unit, which appends the AG number and the ROB entry number to all of its requests and replies.

Note that this mechanism avoids the need for the MSHRs and associative memory which were used in Imagine [4]. The penalty for discarding MSHRs, though, is that multiple outstanding requests for the same word result in multiple memory accesses, but we expect these to be few and that the stream cache will capture the bulk of repeatedly accessed memory locations. The ordering is up to the user, and can be done by an an "ordering" kernel.

### 5.4.3 Stream Cache (SC)

The stream cache may be useful in the following ways.

- Graph streams exhibit temporal locality rather than the spatial locality exploited by the SRF. The stream cache will be used to selectively cache graph elements.
- There are applications that don't know which particular elements of a stream they will be accessing and in what order a priori, but whose accesses do exhibit neighbor locality. Examples are the class of problems that require a PDE solver to traverse an unstructured graph or mesh as well as texture map and z-buffer accesses in graphics programs.

The stream cache is physically-addressed and physically-tagged. The segment registers in each node contain a bit which indicates the caching policy of the segment, either cacheable or non-cacheable. Requests from the address generators will use the cache only if the segment is marked cacheable and the request specifically instructs that the cache should be used; if either of these conditions are not met, the cache will not be accessed. Memory requests from other blocks (MNI, PMI) never use the cache.

The coherence policy for the stream cache, described in [7], is summarized as follows.

- Cache entries can be marked as read-only, guaranteeing that when they are flushed from the cache no write-back is required. The stream model, in which data is not modified in-place, enables the use of read-only cache entries without losing any generality.
- Gang invalidation is used to invalidate a large portion of the cache at once. This is useful due to the fact that many applications contain different computation phases separated by synchronization boundaries. The cache supports two distinct logical regions ($A$ and $B$), and allows the program to invalidate an entire region at once.
- Cache entries which are not marked read-only use a write-back policy. There is a danger here of the different nodes in the system containing inconsistent copies of the same global memory value, and the scalar program, which has explicit control over which addresses may be cached and which may not by the stream load/store commands it issues to the stream controller, is responsible for ensuring that non-read-only cache entries are not cached in multiple nodes.

#### 5.4.4 DRAM Interface (DI)

The local memory capacity is 2GBytes and the local memory bandwidth is 38GB/s, which are achieved by 16 128Mbytes Direct RDRAM chips with a bandwidth of 2.4GB/s each. DRDRAM chips are contained in the RIMM modules and fed by the Direct Rambus Clock Generators which produce the 600MHz differential clock. Each DRDRAM chip is connected to each Rambus ASIC Cell (RAC), which encodes and decodes transaction packets of the information transmitted over the chip.

The local DRAM controller is located between the RACs and the address generators. It receives the memory requests from the address generators and the scalar processor, and makes the transactions needed to communicate with the RDRAM chips for the read and write jobs.

The local DRAM controller is optimized for stream load and save operations, so it concentrates on the stream throughput, rather than the throughput of individual, independent accesses. The memory requests from the scalar processor will be treated as linear stream accesses so that they will be handled same as the other stream operations. It also deals with the DRDRAM maintenance operations.

#### 5.4.5 Memory-Network Interface (MNI)

The memory-network forwards memory requests and responses between the local memory unit and remote nodes. Note that the requests sent over the network use physical memory addresses rather than virtual addresses.

#### 5.4.6 Processor-Memory Interface (PMI)

The scalar processor is able to issue requests for byte-aligned memory addresses, but the memory system is word-aligned (64-bit). The processor-memory interface is responsible for providing the appearance of a byte-addressable memory to the processor. Additionally, the PMI functions as the processor's L2 cache.

#### 5.4.7 Memory Unit (MU)

The memory unit serves two functions, which are to translate the virtual addresses requested by the address generators and processor memory interface to physical addresses, and to coordinate the use of the stream cache, local DRAM, and network. Tables 1 and 2 specify which blocks request data from the memory unit and which blocks respond to requests; the W/C column is the maximum number of words per cycle the blocks can issue requests for or respond to, respectively.

| Requesting block | W/C (throughput) | Requests sent to |
|---|---|---|
| AG/ROB | 8 | DI, MNI, SC |
| PMI | 1 | DI, MNI |
| MNI | 1 | DI |

Table 1: Memory unit requesters

| Responding block | W/C (throughput) | Responses sent to |
|---|---|---|
| SC | 8 | AG/ROB |
| DI | 1 | AG/ROB, PMI, MNI |
| MNI | 1 | AG/ROB, PMI, SC |

Table 2: Memory unit responders

When a request is made by an address generator, the memory unit checks the stream cache tags for a valid copy of the requested address while the address translation is occurring and supplies/stores the data if possible. If the address isn't in the cache and the translated address lands in the current node's DRAM, the memory request is issued to the local DRAM controller. Otherwise, if the translated address refers to a remote node, the request is sent to that remote node via the memory-network interface. The process is the same for a request from the processor-memory interface, with the exception that the stream cache is not checked.

Requests are also accepted from the memory-network interface but in these cases the address requested is a physical address rather than a virtual address and the stream cache, whose function is to cache data from remote nodes, isn't used.

### 5.5 Network and Network Interface (NI)

The network employs a hierarchical topology, uses high-speed (5 Gbits/s per signal) signaling to give high global bandwidth and uses flit-reservation flow control [8] to minimize memory latency.

The network organization, sketched in figure 1, matches the physical packaging hierarchy of the machine. The network is composed of channels connected by routers. Each channel consists of eight 5 Gbits/s differential signals giving it a raw bandwidth of 40 Gbits/s. Messages are switched between channels by routers. There are four routers on each circuit card. Corresponding routers are connected together across the circuit cards to form four completely independent routing planes.

Each router connects to 28 bidirectional channels (eight signal pairs in each direction). Sixteen of the channels are local channels, eight of the channels are backplane channels, and the remaining four channels are global channels. One local channel is connected to each of the sixteen streaming processors on the circuit card. Processors on a circuit card can communicate directly with one another by traversing one router and two local channels and, using all four planes, have a raw bandwidth of 20 GBytes/s over each of these connections. This permits processors to access the memory of other processors on the same circuit card with half the bandwidth that they can access their own memory. The local channels of the 64 circuit cards in a backplane are connected together in a backplane interconnection network (details remain to be worked out). This permits all nodes in a cabinet to sustain a usable bandwidth of 10 GBytes/s each to random locations in the cabinet. Finally, the global channels are converted on the backplane to ribbon fibers and connected in a global interconnection network that permits all nodes in a system to sustain 4 GBytes/s of global memory bandwidth each.

## 6 Feeds and Speeds

The Streaming Supercomputer has a hierarchy of speeds in its connections. System-wide bandwidth provides 4 Gbytes per second per channel. As paths become more local, they increase in bandwidth until the local

register files have 1520 Gbytes per second per node.

## 6.1 Speeds within the Node

This section begins with the most local of data paths and expands out to the global multi-cabinet system. The local data paths are shortest and have the highest aggregate bandwidth, while the global paths scale to longer distances and smaller capacity.

### 6.1.1 Local Bandwidth

The ultimate consumers and producers of data are the Arithmetic Logic Units. Each node of the SSS has multiple clusters, each containing multiple ALUs, each of which requires input and output bandwidth in order to provide sustained computation. The stream model strives to keep computation local, so it is desirable to have fast communication between ALUs within a cluster. Thus, the local bandwidth to the ALUs is orders of magnitude greater than any other data path in the Streaming Supercomputer. In one node, the SSS can provide 1.5 TB/s of data among local registers.

### 6.1.2 Stream Register File

The Stream Register File resides above the local registers in the hierarchy. Streams or pieces of streams reside in the Stream Register File to quickly provide data to the ALUs and to take advantage of Producer/Consumer locality between kernels. The Stream Register File provides 256 GB/s of bandwidth.

### 6.1.3 Cache

The Cache serves as a staging area for data whose access pattern is not sufficiently predictable to allow the data to be loaded directly into the Stream Register File. This includes data from non-grid graphs of arbitrary degree. The Cache provides 64 GB/s of bandwidth.

### 6.1.4 DRDRAM Local Memory

External Rambus memory chips provide 2 GB of random-access storage at a speed of 38 GB/s.

## 6.2 Speeds across the system

Memory requests to any random access are expected to have less than 1 $\mu$sec latency.

### 6.2.1 Card-level Speed

Processors on a circuit card can communicate directly with one another by traversing one router and two local channels and, using all four planes, have a raw bandwidth of 20GB/s over each of these connections. This permits processors to access the memory of other processors on the same circuit card with half the bandwidth that they can access their own memory.

### 6.2.2 Backplane Speed

Within a cabinet, all nodes can sustain a usable bandwidth of 10 GB/s each to random locations within the cabinet.

### 6.2.3 System Speeds

The global channels are converted on the backplane to ribbon fibers and connected in a global interconnection network that permits all nodes in a system to sustain 4GB/s of global memory bandwidth each.

| Parameter | Preliminary value | Description |
|---|---|---|
| $N_{CL}$ | 16 | Number of clusters. |
| $N_{SB}$ | ? | Number of stream buffers. |
| $N_{SEG}$ | 32 | Number of segment registers. |
| $N_{SDR}$ | 32 | Number of stream descriptor registers. |
| $N_{AG}$ | 4 | Number of address generators. |
| $W_{AG}$ | 2 | Width of the address generators. |
| $S_{LR}$ | 32 Words | Local register file size. |
| $S_{SP}$ | 128 Words | Scratchpad register file size. |
| $S_{MCR}$ | 32 Words | Microcontroller register file size. |
| $S_{MCS}$ | 512 KBytes | Microcode store size. |
| $S_{SB}$ | 128 Words | Stream buffer size. |
| $S_{SRF}$ | 64 KWords | Stream register file size. |
| $S_{SC}$ | 8 KWords | Stream cache size. |
| $S_M$ | 2 GBytes | Globally addressable memory per node. |

Table 3: SSS parameters

# 7 Area and Power Estimates

# 8 Parameters and Ranges

Table 3 lists the parameterized elements of the SSS architecture. Note that all words are 64-bits wide unless otherwise specified.

# 9 Issues

## 9.1 Open

### 9.1.1 Virtual Memory

Currently no support for virtual memory is provided since the cost of swapping to disk is extreme and will not allow efficient operation. The question is whether the OS can provide these services without hardware support in a way that hides the cost of I/O.

### 9.1.2 Memory Coherence

There is no hardware support for fine-grained memory coherence, and the intention is for programs to work in phases with gang invalidation of stale data. Is this a poor choice? Both the scalar processor cache and stream cache have coherence issues to consider.

### 9.1.3 Dependency Information

Studies are planned to determine the most efficient and expressive way of specifying the dependence information to the hardware.

### 9.1.4 Aspect Ratio and Layout of Execution Units

Explore the layout of execution units across three axes of parallelism: $ILP$ (the number of arithmetic units per cluster), $DLP$ (the number of clusters run together in a SIMD fashion), and $TLP$ (the number of MIMD blocks per node). Currently we suggest having $ILP = 5$, $DLP = 16$, and $TLP = 1$.

#### 9.1.5 Conditional support

How do the clusters support conditional operations?. Are predication and conditional streams enough? Do we need separate instruction sequencers for each cluster? We need to quantify the differences in cost and performance here.

#### 9.1.6 Inter-cluster Switch Implementation

Should there be two levels of switches for the inter-cluster paths? One for local neighbours and a larger, slower global one?

#### 9.1.7 Register File Organization

Is the two ported local register file still the optimal design point as it was in Imagine?

#### 9.1.8 Integer Unit in Clusters

Should the clusters contain an integer unit in addition to the double precision floating point units? Some of the operations required for conditional support will not require floating point arithmetic.

#### 9.1.9 Memory Locking

Need to spec out how the memory system will support locking of addresses. Perhaps a small table in each node of physical addresses in that node which are locked? This is useful for fetch-and-op, and maybe for other things too.

#### 9.1.10 Number of Stream Buffers

How many stream buffers should be specified in the SSS parameters table? Need to catalog the different stream buffers in the system similarly to the catalog in the Imagine ISA.

### 9.2 Closed

#### 9.2.1 SRF, SB, SC

Determine the arrangement of these three elements of the machine state. Do we allow the clusters to directly reference the SRF or only via the SBs? Can the clusters reference the SC or do they need to generate an index stream and then have the AGs reference the SC?

**Resolution:** The blocks are arranged as illustrated in the block diagram.

#### 9.2.2 Stream Cache

Do we want the cache to be integrated with the SRF so we can fetch blocks from the cache into the SBs? Or alternatively, do we want the cache on the far side of the AGs so that the cache just filters memory references? Probably the latter, but we need to quantify the differences.

**Resolution:** Cache is on far side of AGs.

#### 9.2.3 Parallelism Within the Node

We need to quantify the advantages of having the clusters operate in parallel on adjacent elements of a stream as opposed to subdividing the stream and having each cluster operate on a contiguous substream. Having the clusters work on adjacent elements incurs more inter-cluster communication but enables efficient stream access.

**Resolution:** This is a software issue, but we need to ensure that the provides this flexibility to the software.

### 9.2.4 Fetch-and-Op

Is it efficient for the scalar processor to interrupt the program it is running to process a fetch-and-op request from a remote node? There would be overhead related to it's pipeline and also to message handling in general. Might it be better in the case of a large user-defined operation to not have this feature and just have the remote node do a fetch, operate on it locally, and then write the copy back? Is the atomicity of this operation an important feature? Or - should the processor not be involved in fetch-and-ops at all?

**Resolution:** There will be fetch-and-op, but the exact specs for this are yet to be done. The memory system should support some level of memory locking, guaranteeing atomicity.

### 9.2.5 Inter-cluster Switch Placement

Can the inter-cluster switch also route information between the SRF and the clusters? Is this kind of routing needed to implement conditional streams?

**Resolution:** The switch will not route data between the SRF and the clusters. Data can be communicated between clusters; this is sufficient for the implementation of conditional streams.

### 9.2.6 Memory Segment Sizes

The specification currently in this document is that a segment must be of length $2^k$ bytes in each node (and hence is of size $n * 2^k$ bytes total for a segment over $n$ nodes), and the base physical address of the segment memory in each node must be a multiple of the segment size ($2^k$). Is this correct? If so, it seems that a lot of memory will be wasted between segments. What is the problem with having arbitrary segments? That is, both the segment size and segment base address are only restricted to being a multiple of 8 bytes (1 word).

**Resolution:** The description is correct; hopefully, not too much memory will be wasted due to segmentation, but this specification allows efficient implementation.

### 9.2.7 Support for Debugging of Software

Should the hardware support breakpointing etc. to enable user software to be debugged more easily?

**Resolution:** The scalar processor will support breakpointing (e.g. MIPS), but this won't effect the other blocks, which will keep on executing despite the scalar processor pausing.

### 9.2.8 General Purpose Timers

A small question: Would it be useful to have a set of general purpose timers in the hardware? From the scalar program's perspective, these would be registers that it could write with a number, and the value would decrement by 1 every clock cycle, interrupting the processor once it reached 0. The timers would enable the program to have a real-time component; does this matter at all?

**Resolution:** The scalar processor (e.g. MIPS) will most likely already implement some timers, but we should add some more.

### 9.2.9 Multicasting

Can packets be milticast over the network? Specifically, can a node send a signal of some kind to a set of other nodes?

**Resolution:** No, the network won't allow the processor to specify multiple destinations for a given signal.

### 9.2.10 SIMD Floaing Point Units

Should the floating point units (adder, multiplier, and divide/square-root) in each cluster be able to operate as both 64-bit units as well as paired 32-bit units? In the latter mode, each 64-bit operand would be split into two 32-bit operands, and the floating point operation would be performed on each pair of 32-bit operands in parallel. If this were used, the clusters may also need a hardware floating point truncation/packing and padding/unpacking block, as the one program may use some single-precision kernels and some double-precision kernels.

**Resolution:** This could be a performance-improving feature, but it would add to the complexity without contributing to research, so will not be included in the SSS.

### 9.2.11 L2 Cache

Should the processor-memory interface be an L2 cache? Or should the interface simply handle the byte/word alignment issue?

**Resolution:** Yes, the PMI will be the processor's L2 cache.

### 9.2.12 SRF Random Access

Will random mode accesses to the SRF be supported? Will there be a mini-memory access scheduler between the SBs and the SRF to maximize the SRF bandwidth during random mode transfers?

**Resolution:** Random access to the SRF is not supported.

### 9.2.13 Multidimensional Strides

Brook supports multidiminsional strides as well as stream grouping (specifying which elements of one stream are grouped to form a single element of a new stream), and Ian said that many applications will make use of at least the multidimensional strides. Should the SSS hardware support more than the one dimensional strided access currently specified? How about the grouping?

**Resolution:** Index streams can efficiently support multidimensional and grouped streams (as specified in Brook) with the exception of constant (clamped) boundary conditions. To support these, the index streams should be extended to allow elements to be either offsets (as in Imagine) or absolute virtual addresses.

## References

[1] B. Khailany, B. Dally, et al., "Imagine: Media Processing with Streams", *IEEE Micro*, March-April 2001, pp. 35-46.

[2] B. Dally, P. Hanrahan, R. Fedkiw, "A Streaming Supercomputer", September 2001.

[3] U. Kapasi, B. Dally, et al. "Efficient Conditional Operations for Data-parallel Architectures", *Proceedings of the 33rd Annual International Symposium on Microarchitecture.*

[4] B. Dally, S. Rixner, et al., "The Imagine Instruction Set Architecture", June 2001.

[5] (Peter Mattson's PhD)

[6] (Ian Buck's Brook document)

[7] (Ben Serebrin's cache coherency memo)

[8] (Document describing Flit Reservation Flow Control)