Reconfigurable Clusters of Memory and Processors Architecture for Stream Processing Systems

Vason P. Srini, Jan M. Rabaey
Berkeley Wireless Research Center (BWRC)
2108 Allston Way
Berkeley, CA 94704

ABSTRACT

A computer systems architecture containing reconfigurable clusters of memory and processors, called RAMP, is proposed in this paper for processing streams of data in high performance image and signal processing and embedded systems. Each cluster in RAMP has one or more memory modules, computation processors, and an I/O processor that allows direct input/output from sensors and actuators. The computation processors receive streams of data and execute functions in a data driven manner. Interprocessor communication is supported using a simple handshake protocol in hardware. A memory module with logic to support random access (RAM), FIFO access (FIFO), delay line implementation (DELAY), and lookup table implementation (LUT) is proposed as a part of the RAMP architecture. The memory module can be reconfigured statically to be in one of the four modes (RAM, FIFO, DELAY, LUT) to support convolution function in image processing, implementing a sample data buffer and digital filters, and color conversion in digital camera and video camera. It is also useful in storing look up tables needed in DFT, FFT/IFFT (twiddle factor), and video compression algorithms. In addition, I/O processors for receiving data from sensors and sending data to displays and actuators is proposed. Software for partitioning applications and mapping them to processors, and routing the processors is also discussed.

A prototype RAMP system containing 16 clusters is described. Each cluster has four processors, one memory module, and one I/O processor. The challenges in interconnecting the clusters in one chip, communicating between processors in one cycle, and the design of the processors is described. Mapping applications such as 2-D convolution to processors in the RAMP architecture is shown.

This research was partially funded by AFRL, Eglin AFB, FL. Contract No. F08630-95-C-101. Email: {srini, jan} @eecs.berkeley.edu

1.0 Motivation

The computing and communication industry continues to face a major challenge in system-level design and adaptation complexity. General purpose microprocessors and digital signal processors do not have the computational density, functional diversity, and energy efficiency to support realtime multimedia communication systems, target recognition systems, Radar and Ladar guidance systems, collision avoidance systems, and reconnaissance systems. Reconfigurable devices such as PLDs, FPGAs, and application specific processors have come closer to meeting the computational density requirements for low granularity operations of some of the above applications by using less instruction memory and simpler architectures. However, meeting the changes in the computational requirements and achieving energy efficiency during the span of a mission by dynamically reconfiguring the available computational resources is still an open problem. Other factors such as fault tolerance, changing environments, and steering the computation based on local information also require dynamic reconfiguration. This paper describes a scalable dynamically reconfigurable computer systems architecture with emphasis on fast runtime reconfiguration and granularity at the block of instructions. It also has facility for steering the computation, multiple instruction and multiple data stream (MIMD) model of parallel computation, and synchronization done in hardware using one cycle delay. Efficient partitioning and placement using architectural details, computational requirements, and multiple granularity is also discussed.

2.0 Economic Opportunities

Portable personal computing (PPC) devices integrate the functions of cellphones, pagers, laptops, global positioning systems, digital cameras and VCRs, and information organizers. Multiple streams of data arrive at the PPC devices and they have to be processed simultaneously in realtime. Data security and integrity, virus vigilance, and

authenticated access are important at all times. Supporting the PPC environment requires processing parallel streams of data in realtime.

This paper proposes a low-power, low-cost, and high performance parallel processing architecture that can be implemented as a VLSI chip and the needed software so that the chip can be used in portable personal computing (PPC), laptop computers, and hand-held devices. Such a system is needed because general purpose microprocessors and their low-power counterparts cannot meet the performance and low-energy requirements of multimedia applications such as secure Web browsing and secure multimedia email with virus vigilance.

The proposed architecture and a VLSI chip based on it will combine parallel stream data processing (SDP) with conventional sequential processing to provide high performance (10 Giga operations per second -GOPS) and low-energy (less than one watt). The parallel SDP system will be based on the VGI (video, graphics, and image processing) architecture and instruction set, and the sequential processing will be done using a low-power MIPS processor available in the public domain. Innovative approaches have been developed for the integration of sequential and parallel stream data processing, compiler technology, partitioning, placement, and routing technology to map applications to the resources in the VLSI chip. A memory system that provides a large common address space with physically distributed memory modules is used. This combination of processors and memory technology will meet performance goals expected by standards such as MPEG-4 and JPEG-2000, encryption standards, and virus vigilant software in PPC devices.

Reconfiguring the computing resources to handle adaptive and evolving algorithms is supported. Some of the algorithms used in high data rate wideband CDMA wireless communication are adaptive in nature and the algorithms are also evolving. Innovative optimizations of the compression and decompression algorithms in the standards also require reconfiguration. Strategies for reconfiguring the computing resources on the fly are discussed.

Smart phones, PDAs, laptops, digital cameras, and other networked portable devices can use the VLSI chip and support video conferences and image processing. Using an array of the proposed VLSI chips in an add-on board, medical images can be processed and communicated in realtime to support telemedicine. New industries could be established to handle digital scanning, data archiving, and delivery of medical images between patients at homes and clinics and hospitals and HMOs. Machine vision applications can use the VLSI chips to improve recognition accuracy. MRI machines can use the VLSI chips to reduce computer hardware cost. High data rate network routers, wireless LAN systems, and electronic instruments such as spectrum analyzers and vector analyzers can use the VLSI chip to reduce size, power, and cost.

3.0 Introduction

The need for high performance and low-power microprocessors for many of the media applications involving images, sound, and video has been recognized by many computer companies and research organizations. Processors such as Philips' TriMedia, TI's TMS320C6000, Analogic's Sharc, HP/ST's Lx, and HP's PICO-NPA [1] have been introduced to meet the performance requirements. The datapath in these processors are optimized to provide extremely high performance on certain kinds of arithmetic-intensive algorithms. However, a powerful datapath is only a part of the solution for high performance. To keep datapath fed with data and to store the results of datapath operations, the processors require the ability to move large amounts of data to and from memory quickly. Thus, the organization of memory, latency, bandwidth, and its interconnection with the processor's datapath are critical factors in determining processor performance. It is known that any off-chip communication is expensive in terms of power, delay, and area. To drive a signal off-chip, large current is required to drive the capacitor on the I/O pads and any external capacitors. Therefore, having to move data to and from chip to off-chip memory will not be an efficient implementation. This observation is recognized in the computer industry and high performance DSP and media processors have on-chip memory. Commercial general purpose digital signal processors like Motorola's DSP56000 family and TI's TMS320C6000 family have separate on-chip memory for program and data. With on-chip memory, the memory bandwidth can be increased and off-chip traffic can be reduced.

FIR implementation is the simplest example to clearly illustrate the memory requirement of DSP processors. A single FIR tap computation requires four accesses to memory - instruction fetch, read data from delay line, read the appropriate filter coefficient, and write data to the next location in the delay line to shift data through the delay line. A similar pattern of memory accesses occurs in 2-D convolution and filter computations in image processing. The memory accesses involved are random access, table lookup, and delay line.

Another important factor is I/O bandwidth. Data coming from multiple A/D converters operating in the range of 20

to 100 MHz have to be processed in realtime. Efficient ways of doing I/O is needed. Most of the DSPs support sending and receiving data from sensors using buses.

In this paper we discuss the architecture of a reconfigurable clusters of memory and processor (RAMP) system capable of delivering tens of giga operations per second and uses low-power techniques. The RAMP system combines on a single chip many stream data processors, memory modules with builtin logic and controllers, I/O processors, and a conventional superscalar processor. The parallel stream data processors compute on the multiple data streams using a MIMD model of computation and the superscalar processor performs the coordination of parallel tasks, sequential computations, and communication with the memory system and I/O system. The execution time for any instruction in the vgiprocessor is one cycle. An overview of RAMP architecture is given in Section 4.0 . An overview of the cluster architecture in the RAMP is described in Section 5.0 . The architecture of the processor and the blocks of the datapath are described in Section 6.0 . The architecture of the memory module, the blocks of the datapath, and memory timing are described in Section 7.0 . The programming of the memory module as well as other processors is done using a scan chain register in each cluster and reconfiguration is achieved using the scan chain register after bringing the processors in a cluster to a freeze state. The 55 bits forming the scan chain register of a memory module is also explained in Section 7.0 . The I/O processor and it details are described in Section 8.0 . The simulator design for RAMP and the simulator for the first implementation, called VGI chip are described in Section 9.0 . The signal flow graph (SFG) for the 2D convolution and mapping it to clusters and processors in the RAMP is also described in Section 9.0 . Future directions for RAMP in the form of extensions to the architecture and its implementation are described in Section 10.0 .

4.0 RAMP Architecture

Realtime audio/video communication and image processing using wired and wireless networks involve handling data coming in streams at a high rate from sensors, cameras, and networks. Physical (PHY) layer components in high data rate wireless networks using standards such as IEEE 802.11ag, Hiperlan2 (wireless LAN), IEEE 802.16ab (wireless MAN) also require processing data coming in streams from one or more antennas. Temporary storage and fast access to data streams are important to keep the overall system performance at a high level. In the case of wireless LANs such as IEEE 802.11ag and wireless MANs such as IEEE 802.161b, other factors such as low-power and low-cost are also important. This means some of the PHY components might operate at different clock rates and some of the components might be turned off to save power. A block diagram of PHY components involved in WLAN/WMAN system is shown in Figure 1. Each component in the diagram gets control tokens from the control processor. This is shown by thin lines with arrows on one end. The point to point interconnection is shown by thick lines between PHY components. Data width conversion, serial to parallel conversion, and buffering functions are shown using thin rectangles. Functions that do computation are shown by squares. The thick lines show data token communication between blocks.

Dealing with different clock domains in a single chip is a challenging activity by itself. One way to handle the communication of data across different clock domains is to use asynchronous FIFOs at the interfaces, shown by thin rectangles in Figure 1.

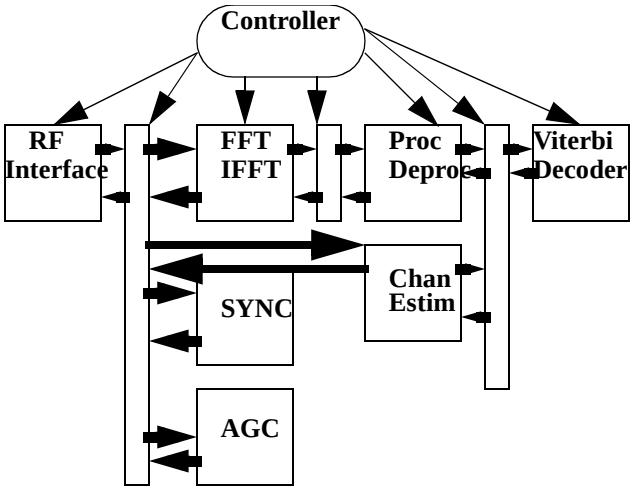


FIGURE 1. Block Diagram of WLAN/WMAN

RAMP is proposed as a low-power and high performance reconfigurable parallel processing system with the needed partitioning, placement, and routing (PPR) software so that the above application can be mapped to processors on one or more chips. A block diagram of RAMP is shown in Figure 2

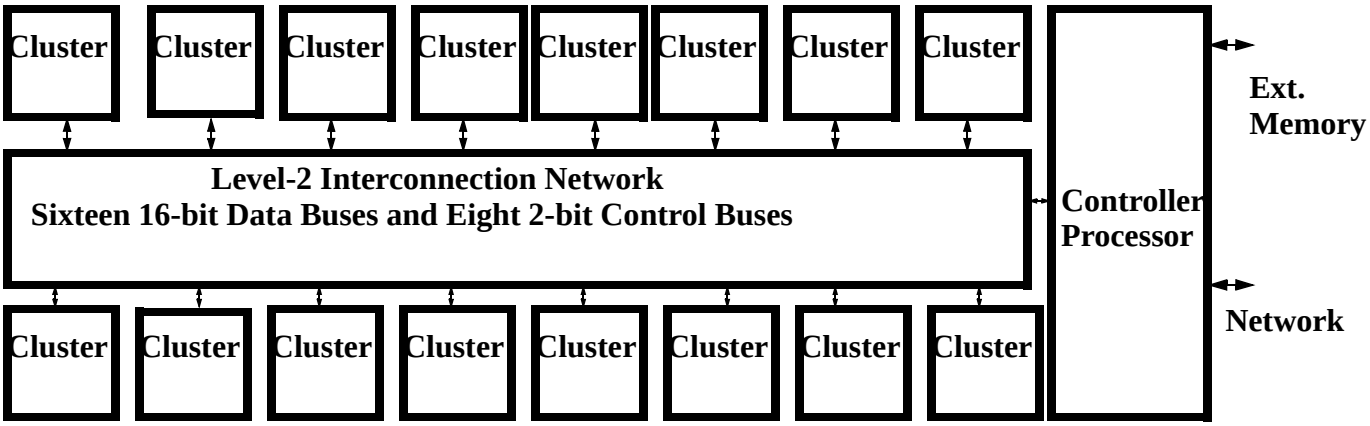


FIGURE 2. RAMP VLSI Building Block with 16 Clusters and External Connections

A RAMP system comprises a hierarchical interconnection of clusters of processors and memory modules. The block diagram shows 16 clusters interconnected by 16 data buses to form a crossbar-like network. Reconfiguration is done at the cluster level. The controller processor in Figure 2 is the unit responsible for dynamically reconfiguring the clusters by sending configuration tokens over 8-bit buses. It also supplies data streams from external memory or network to the clusters, and stores results generated by clusters in external memory. Each cluster has processors, memory modules with logic, and an I/O processor. The memory modules in the clusters can also receive data from cameras, sensors, and supply results to tape recorders or other recorders and display monitors using I/O modules. Each cluster is designed to be reconfigurable independent of other clusters.

One of the key features of RAMP is the separation of information flow into three separate entities: data tokens, control tokens, and configuration tokens. The configuration tokens are used to setup the processors, memory modules, I/O modules, and interconnection networks at load time or reconfiguration time. The control tokens are generated by programs. Another key feature is the ability of all the units within a cluster to communicate data with a delay of one cycle using a simple send/receive handshake protocol. By limiting the number of clusters to 16 in a chip, intercluster communication delay can also be kept at one cycle without slowing the processor. The implementation of the protocol using a single wire, buffers, and queue controllers in sending and receiving units is discussed.

Since the memory module can be configured to be in one of the four modes, control logic is needed to switch between these modes and also to perform the operations in each of the modes. Four finite state machines (FSMs) have been designed to support the four operating modes of the memory modules. The details of the FSMs and other control parts of the memory module are described. The storage array for the memory module is based on a self timed low-power SRAM design. The programming and reconfiguration of the memory module is supported using a scan chain register and control signals. The datapath, controllers, and storage array of the memory module have been integrated and simulated at the VHDL level and transistor level to verify functionality and timing.

RAMP has its foundations in PADDI-2 [2] chip containing 48 nanoprocessors organized as 12 clusters. A nanoprocessor is used to interface the PADDI-2 chip and the external memory. The nanoprocessor handles the single cycle interprocessor communication protocol. In the RAMP system, a part of the memory is on-chip and partitioned so that a small portion of the memory is available in each cluster to support locality of data in applications. Based on analyzing DSP algorithms [3, 4], simulation studies, and experiments [5], it was determined that performance could be improved if memory can be organized to fit the four commonly used memory accesses in signal processing - RAM, Look-up Table, FIFO, and delay line. We decided to enhance the storage array in a memory block with logic and controllers to support the four commonly used accesses and the single cycle interprocessor communication protocol.

5.0 Cluster Architecture

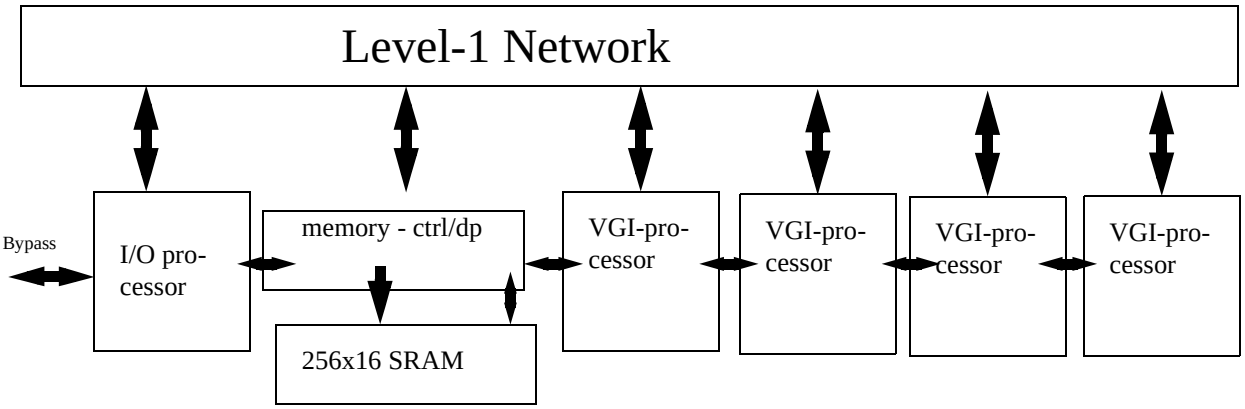


FIGURE 3. Cluster architecture with memory module, vgiprocessors, and I/O processor.

A cluster containing processors, memory, and I/O interface is the building block of a RAMP system. A block diagram of a cluster in RAMP is shown in Figure 3.

It contains four processors that can do video, graphics, image processing, and digital communication computations on streams of data, called vgiprocessors [5, 6]. It is possible to replace the vgiprocessor with a more powerful processor or a matched processor depending on the application. We focus on the use of vgiprocessor in this paper and its details are described in Section 6.0 .

A memory module containing a storage array (256x16 words) and controllers for implementing the four commonly used memory accesses is the second major part of the cluster. The memory module is a fast local memory for supplying data to the processors. Although the prototype memory module has a 256 X 16 SRAM, it is not a limitation. The storage array can be increased in size and the controllers can be resynthesized by changing the parameters of delay lines, queue length, and counters. The memory module design and its datapath are described in Section 7.0 . The details of the array are in [7, 8].

The third part of the cluster is an I/O processor that sends and receives data from external devices such as sensors, digital cameras and video cameras, and A/D and D/A converters using a two cycle delay communication protocol. It also communicates with the external system memory to supply data to the processors. The I/O processor communicates with vgiprocessors and memory modules using a single cycle delay communication protocol. All six processors in a cluster have a three stage pipeline in their datapaths and each one is designed as a synchronous system. In the first implementation of the cluster architecture there is a system clock to synchronize all the processors. It is possible to use a globally asynchronous and locally synchronous (GALS) clocking scheme with the RAMP architecture and it is discussed in the section on future directions, Section 10.0 .

Communication between the processors in a cluster is done through six data token buses (16 bits) and four control token buses (2 bits) in the Level-1 communication network and the associated handshake lines. Adjacent processors in a cluster can communicate data using the bypass buses and thus leaving the resources in Level-1 network for intercluster and non adjacent processor communication.

The single cycle delay communication protocol for memory module and I/O processor is the same as that of the vgiprocessor. From Figure 3, we can see that memory module's control and datapath is the interface between other processors and the storage array. Memory module handles all the timing requirement to read and write into the storage array. It also handles data addressing in FIFO and delay line mode.

Using Level-1 network, the processors in a cluster can communicate with other clusters in a RAMP architecture using an intercluster network, called Level-2 interconnection network. It is a restricted crossbar network containing 16 data token buses and eight control token buses. The communication between Level-1 and Level-2 buses is done by setting the switches (50 switches in all) between Level-2 network and Level-1 network. The switches are n-type transistors in this work. These switches will be replaced by asynchronous FIFOs if the GALS clocking scheme is used for the architecture.

One of the novel features of the RAMP architecture is the single cycle communication protocol between processors in a chip using a send/receive handshake protocol. This protocol is implemented using a bidirectional handshake line for each data token bus and control token bus and the two phases of the system clock. During the execution

stage, a processor wanting to communicate data to another processor pulls the handshake line high when clock is low. If the receiver processor is not ready it can pull down the handshake line when the clock goes high on the next cycle (communication stage). At the end of clock high the success of the transaction is evaluated. If the handshake line is high then the transaction is successful. Otherwise it is unsuccessful. A sender can retry when the clock is low. The details of the protocol and its implementation are in Section 6 and Figure 8.

6.0 Processor Architecture

A vgiprocessor comprises a datapath, a control unit, an instruction memory, and switches for connecting to the Level-1 network. Each vgiprocessor contains its own 16-address instruction memory (40 bits wide), six general purpose data registers that can also be configured as three queues, a control unit for instruction sequencing and three stage pipeline management, a 16-bit datapath, and connections to the Level-1 communication network. A block diagram of the vgiprocessor is shown in Figure 4.

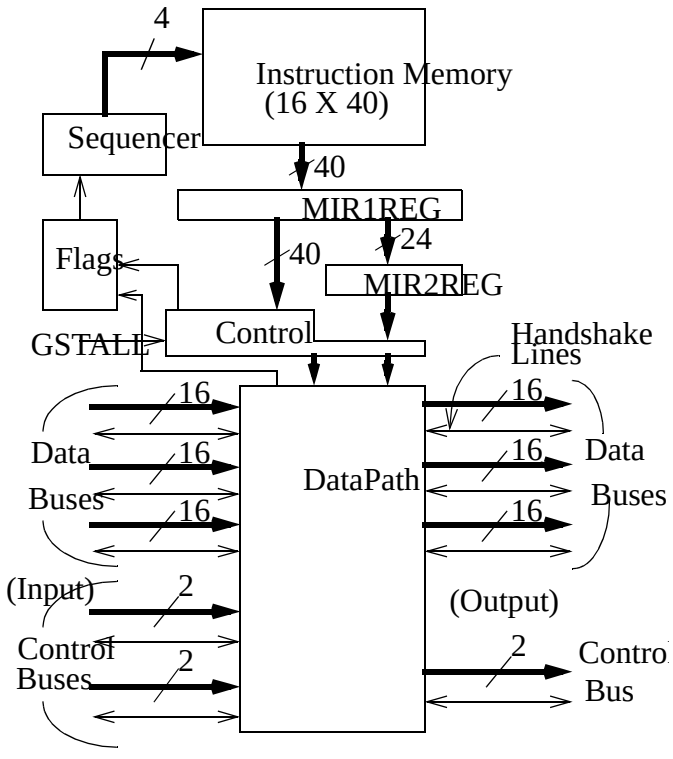


FIGURE 4. VGI Processor Block Diagram

The vgiprocessor is a follow-on to the PADDI-2 processor [2]. The performance improvement is derived mainly from the increase in the functional capabilities of the vgiprocessor. The PADDI-2 processor performs only simple operations such as add, subtract, multiply, a few logical operations, a few shifts, and a select operation. There are only 13 opcodes. In comparison, the instruction set of vgiprocessor contains 194 opcodes. The new instructions are targeted specifically for video and image processing, graphics processing, and signal processing. There are some common operations used in image processing which take several cycles to complete in the PADDI-2 instruction set. New instructions are added to make these common operations faster. Some targeted operations include chroma-keying, sorting, and handling various data operands and sizes.

6.1 VGI Instruction Set

The instruction set of vgiprocessor (VGI instruction set) greatly expands the PADDI-2 [2] instruction set and is designed so that all instructions execute in one cycle (CPI = 1). Additional instructions include saturation arithmetic, packed operations, compare-if-equal, maximum, minimum, high precision multiplies, and MultiOp instructions.

Each instruction has the following parts:

label1: // Instruction address in the range 0 to 15.

dest = op(src1, src2, src3, srcc), // destination for the result of an operation

cc = c/z/s/o, // Set the condition code register to one of the flags - carry, zero, sign, overflow

fo = cc/f0/f1/00/01/10/11, // Set the output flag to one of the seven values

c0 = ra/rb, c1 = ra/rb, c2 = rc, // Set the output register values to ra, rb, or rc, where ra, rb, and rc are internal registers

next = label2; //Address for the next instruction to be executed

The VGI instruction set features arithmetic, shift, and logical operations. Arithmetic operations are done using two's complement format, and there are add and subtract operations with and without carry-in. The carry-in versions are used to handle operands that are multiples of 16 bits. New instructions include increment by one and decrement by one. The logical operations include AND, OR, XOR, INV, and INV1AND, the latter being invert the first operand and AND it with the second operand. Shifts are limited to shift up by one, down by one, or down by two. Arithmetic,

logical, and circular shifts are supported. Shifting up by n bits (1< n < 16) requires n shift up by one instructions.

Saturation arithmetic is a desirable operation in image processing and signal processing to prevent wrap around of data values when doing arithmetic operations. Saturation addition prevents the resulting value from overflowing beyond the maximum representable value, whereas saturation subtraction prevents the result from going below the least representable value. Saturation arithmetic is supported for unsigned adds and subtracts.

Multiplication of two's complement numbers is done using the modified Booth algorithm, and requires eight steps (the number of data bits divided by two bits per cycle) for 16-bit operands. Each step needs to be specified individually in the program code so that a CPI of one can be maintained for all instructions. Multiplication can also be done in a pipelined manner by cascading eight vgiprocessors. The pipeline latency is eight cycles and the throughput is one result every cycle. Since many algorithms on the average do integer multiplications only 10 to 15% of the time, a dedicated hardware multiplier is not included in the vgiprocessor. The result of a multiplication can be lower 16-bits or upper 16-bits. If a full 32-bit precision is required, then the programmer can use two vgiprocessors: one vgiprocessor can calculate the upper 16 bits and the other the lower 16 bits of the result.

Control flow for the vgiprocessor is specified in each instruction using the next field of an instruction. It can be the next sequential address, or the target address of an unconditional or conditional branch. For the unconditional branches, any address location is a valid destination. For conditional branches, there are restrictions on the valid destinations. The requirement is that the destination for a condition of TRUE must be an odd numbered address, and the destination for a condition of FALSE must be that address minus one. The reason for this constraint is to keep the instruction word within 40 bits. To support two independent destination addresses would require an additional four bits in the instruction word, which would increase the instruction memory size by 10%. Despite this constraint on branching, it is still possible to fully utilize all instruction memory addresses (using some clever assembly language programming).

For branches, the possible conditions are limited to the Level-1 input control flag queues or the condition code register (CC). Branches using the CC register are delayed by one cycle, since there is a one cycle latency in setting this register. Hence, branches based on CC would use the value of CC set in the previous instruction rather than the value set in the current instruction.

The data sizes for image processing and signal processing typically vary from eight bits to 32 bits. Many video processing algorithms use 16 bits for representing pixel data (4:1:1 YCrCb, 4:2:2 YCrCb, and 4:4:4 YCrCb) [5]. This creates a dilemma for the processor designer when choosing datapath width. If one selects an 8-bit datapath, then it becomes difficult to operate on larger data sizes, such as 16 bits. This would adversely impact the utilization of chip area since effectively twice as much area would be consumed by control--two processors would be required, hence two control units. If one selects a 16-bit datapath, then operating on eight bit quantities would waste half the bandwidth of the datapath--again under utilizing layout area. One solution to this dilemma is a 16-bit datapath with packed operations. The vgiprocessor includes packed operations so that each byte of the 16-bit operand can be acted upon individually. Therefore two pieces of eight bit data can be concatenated to create one 16-bit operand. This maintains the advantage of minimizing control overhead for the 16-bit datapath, and eliminates the wasted bandwidth of the 8-bit word in a 16-bit datapath. For most operations, the opcode can be applied to the upper eight bits, the lower eight bits, or both the upper and lower eight bits individually. For example, a packed add operation adds the upper and lower halves of the operands separately. To fully support packed operations, the condition flags have been expanded to two bits. The upper bit corresponds to the flag for the upper byte, and similarly the lower bit corresponds to the lower byte. When doing full 16-bit operations, the flag is placed in the upper bit, while the lower bit is undefined.

6.2 Timing

A single clock signal is used for the clusters and the entire RAMP architecture to explain the timing. Both phases of the clock are used in many circuits of the processor and other modules in the first implementation of RAMP. Clock buffers that produce different drive capabilities and produce both phases of the clock have been designed. Each block in the vgiprocessor uses clock buffers and other control signals to produce gated versions of the clock.

Transparent latches are used throughout the design. The pipeline for the vgiprocessor is two stages deep (therefore all instructions have a two cycle latency when executed individually). The control bits for the pipeline stages come from instruction memory, stored in the 40-bit wide microinstruction register (MIR). The activities done in the two stages are shown in Figure 5.

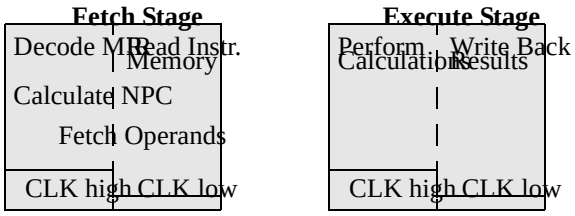


FIGURE 5. Pipeline Divisions

6.3 Datapath

The datapath of the vgiprocessor is fairly complex, due to the presence of packed instructions, saturation arithmetic, and multiplication instructions. The datapath of the vgiprocessor is 16 bits wide, and contains 12 blocks. The blocks of the datapath and the buses used by them are shown in Figure 6.

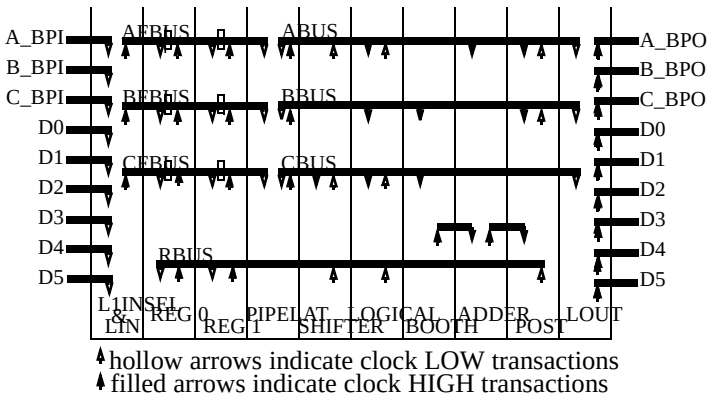


FIGURE 6. Datapath Bus Diagram

To support packed operations, the datapath was basically split in half and then pasted together with some glue logic. Furthermore, additional control was required for doing operations such as packed multiplies, which required an additional booth decoder. A layout of the datapath is shown in Figure 7 to illustrate this.

In Figure 7, there is a gap in the center of the datapath between the low 8 bits and the high 8 bits. This space was required to accommodate the additional glue logic.

6.4 Control

The control unit of the vgiprocessor is divided into functional blocks to simplify the VHDL coding, synthesis, and verification. There are individual blocks for each stage of the pipeline, the Level-2 communication network handshaking, and the next address calculation for the program counter.

6.4.1 Fetch Control

The fetch control (“fectrl”) block is responsible for providing the control points to the datapath for the fetch stage. This includes decoding operand addresses and maintaining the status of the five queues (three for data buses and two for control flag buses).

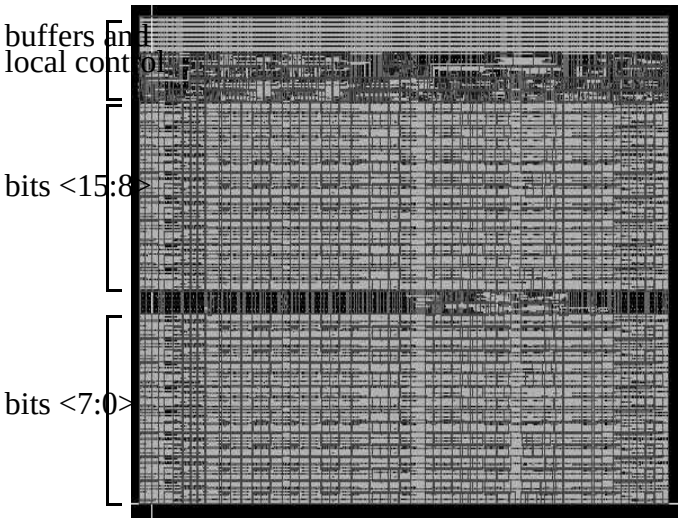


FIGURE 7. Datapath Layout

The queue controllers maintain the state of the two-deep queues which buffer incoming data and control flags. There are five queue controllers in all--one for each of the three data queues, and one for each of the two control flag queues. They operate as regular queues but also have the ability to save values, since values can be read from queues without advancing them, thereby saving the value.

Since each data input queue can be changed to act as two random access registers, decoding for this mode of operation is also done simultaneously. The fectrl block then takes the output from the queue controllers and multiplexes them with the output from the register decoding circuitry. The mode of operation is determined by the state stored in the scan registers of the vgiprocessor.

The fectrl block also generates the stall_empty signal, which stalls the vgiprocessor when values are needed but the queues are empty. This signal effectively creates “bubbles” in the pipeline--the execute stage will finish its current operation and, if needed, will output the data onto the

Level-1 communication network. When data is received by the queue, the stall_empty signal is unasserted and the current instruction will advance through the pipeline.

The fectrl block is synthesized from the behavior description in VHDL using Synopsys' Design Analyzer / compiler software. Then the block layout is generated using Cadence's Cell Ensemble and compacted by hand using interactive Layout Editors.

6.4.2 Execute Control

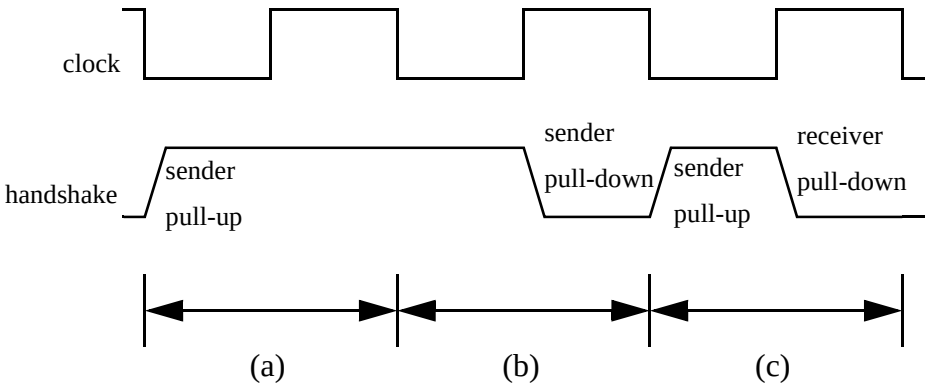
The execute stage control is contained within two blocks. The first block ("exctrl") decodes the opcodes and provides the corresponding control points to the datapath. The opcodes are encoded to minimize area and reduce time penalty. The second block ("postctrl") was created to isolate the critical path of the vgiprocessor. It handles the control of the post-processing unit (see Figure 6). It accepts output flags from the adder along with operand information to determine final execute stage results for operations such as maximum, minimum, and compare-if-equal. Therefore, the critical path for the vgiprocessor begins in the exctrl block.

6.4.3 Program Counter Control

The address of the next instruction to be executed is calculated by the program counter control unit ("npcctrl"). The next value of the program counter for unconditional branches is directly taken from the MIR, that is, no decoding is required. For conditional branches, the bit field corresponding to MIR<34:33> in the MIR is used to select the appropriate flag input for the branch condition. This control block evaluates the flags and determines the next value for the program counter.

6.5 Communication Protocol

The protocol for communicating data uses a single handshake line. The sender precharges this line when it is ready to send, and any of the receivers pulls down the line if it is not ready to receive. Illustrated in Figure 8 are three transactions, a valid transaction is established by a logical one on the handshake line at the end of clock high. This indicates that the sender is ready to send and has precharged the line, and all of the receivers are ready to receive, leaving the line precharged. If the handshake line is low at the end of clock high, a valid transaction has not occurred, either because the sender did not precharge or one of the receivers pulled down the handshake line. Since there are three output ports in a processor, if any of the output ports is blocked, the sender processor is blocked along with the intended receiver processor. If the producers and consumers of data operate at the same rate, the above type of blocking is reduced when running programs.



(a) shows a valid transaction. The sender pulls-up the handshake line while the clock is low and the handshake line remains high while the clock is high, indicating a valid data transfer.
(b) shows the condition when the sender is not ready to send. The sender pulls-down the handshake line when the inverted clock goes low (falling edge).
(c) shows the condition when the sender is ready to send, but the receiver is not ready to receive. The sender pulls-up the handshake line while the clock is low, and the receiver pulls-down the handshake line when the clock is high.

FIGURE 8. Handshake Timing Diagram.

6.6 Scan Chain

A scan chain is used to load the processors in each cluster and read out contents of registers. The scan operation takes place when the processor is not executing a program. The scan chain for the vgiprocessor has 89 bits. It contains 40 bits for instruction, 16 bits for register, and the remaining bits for setting up the switches, queues, and flags. The 16 words of the instruction memory are loaded by repeating the scan chain 16 times with desired data. The GSTALL signal stalls the processor by maintaining the current state of the entire vgiprocessor. The MIR and MIR2 registers, the CC register, the input queue controllers, and the communication network output controller maintain their current state when GSTALL is asserted. Stalling these blocks is sufficient to maintain the exact state of the vgiprocessor.

6.7 Design Verification

All the blocks of the vgiprocessor have been designed and functionally simulated at the RTL level using a VHDL simulator. Many blocks in the control part have been synthesized. The remaining blocks have been custom designed. The buses of the Level-1 network have been integrated in the datapath. The physical layout of the entire processor is shown in Figure 9.

.

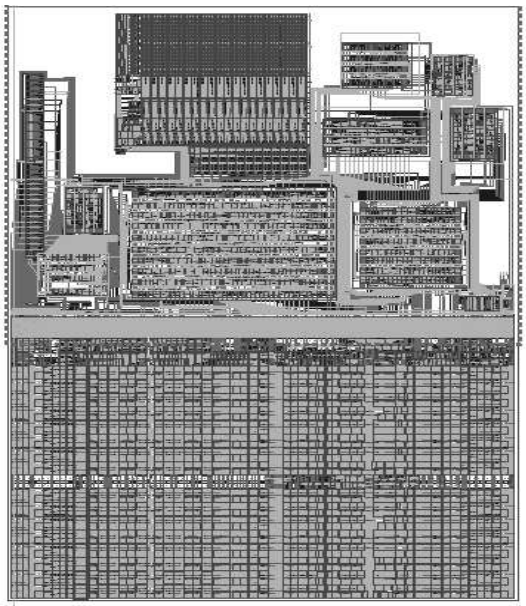


FIGURE 9. Layout of Vgiprocessor

7.0 Memory Module Architecture

The memory module is the second major part of a cluster architecture. It has a datapath and a control section. A block diagram of the memory module is shown in Figure 10. The datapath comprises all the blocks shown in the figure except the SRAM array and the controller. The data queue and the pipelat blocks are similar to those used in the vgiprocessor and the I/O processor. The datapath is described in detail to fill some of the details not covered in the previous section on the processor.

The datapath has a three stage pipeline similar to that of the vgiprocessor and I/O processor comprising fetch, memory access, and communicate stages. The fetch and communicate stages are needed to synchronize the memory module with a vgiprocessor and an I/O processor. This avoids the need for special protocol when a vgiprocessor communicates with the memory. The additional two stages impose a two clock latency when read is requested and one clock latency for write process. Since requests are usually pipelined, the latency occurs only on the first request.

In the following subsections the architecture of the memory module is explained. The datapath structure is described first and then the controller block.

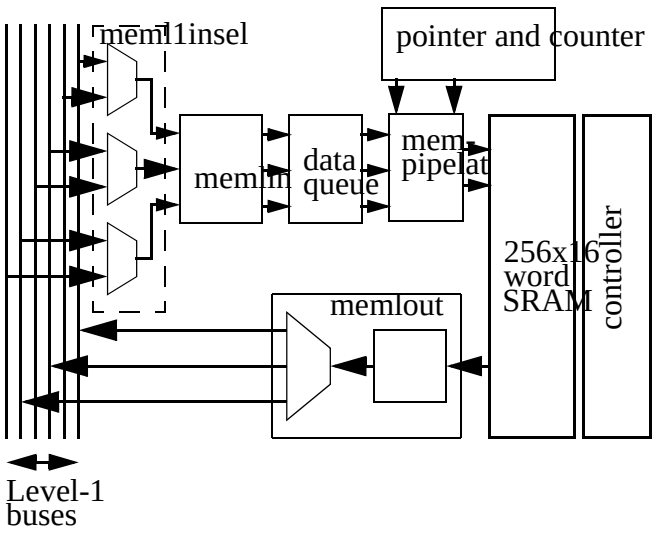


FIGURE 10. Block Diagram of Memory Module

7.1 Datapath Structure

The memory module's datapath consists of the following blocks: meml1insel, memlin, data queue, mempipelat, storage array, memlout, address pointer and counter as

shown in Figure 10. We have retained some of the cryptic names for the blocks to maintain consistency with the names in design schematics and netlists. The following subsections describe in detail the functions of each block.

7.2 Meml1insel Block

Meml1insel is the interface between buses in Level-1 network and memory module. It consists of three multiplexers that select 3 out of 6 Level-1 buses. The three outputs of this block are read address, write address and write data. The bus assignment is done by setting the associated Level-1 configuration switches in the scan chain register (bits memscni<5:0> in the scan register of memory module). Figure 11 shows the Level-1 configuration switches.

The buses in Level-1 network are assigned in the following manner:

- bus (d0 or d1) or bus (d2 or d3) can become read address source.
- bus (d0 or d1) or bus (d2 or d3) can become write address source.
- only bus d4 or d5 can become write data source.

7.3 Memlin Block

This block consists of two 8-bit registers for read and write address and one 16-bit register for write data. During the communicate cycle, the sender drives the output data on Level-1 bus. On the same cycle, the receiver copies the data into the master of the memlin register. During fetch cycle, the data in memlin will be transferred into registers in data queue block. Memlin is the communication data buffer at the receiver. In memlin block, data and address source can be selected from the bypass buses or the Level-1 buses. To select the address or data source from bypass buses, the associated memsbpi bits in the scan chain register (memsbpi<2:0>) should be asserted.

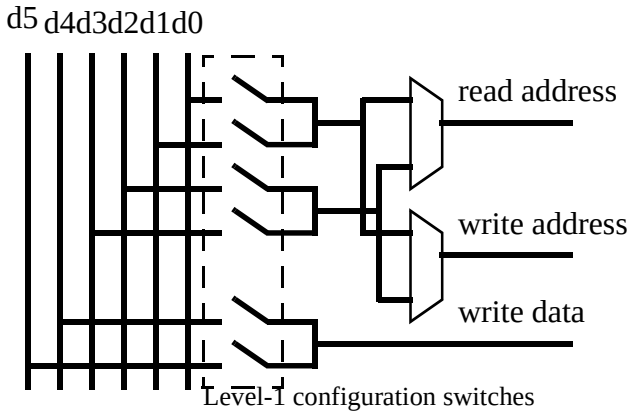


FIGURE 11. Meml1insel structure. Level-1 configuration switches and read and write address

7.4 Data Queue

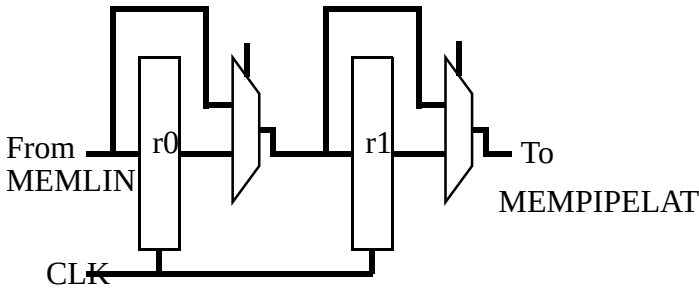


FIGURE 12. Schematic of one queue register. Data queue block consists of three such queue register; they are read address

Data queue consists of three two-deep queues, one for read address, one for write address and the other for write data. The queues are important because of the two cycle delay for a request to be processed. For example, when a read is requested, whether the read data can be sent out or not is not known until two cycles after the request is received. During those two cycles, additional requests are being accepted and processed. If the memory module fails to send the read data the processor will stall; data queues are used to buffer the requests that have been received before the stall comes. The data queue works in the same manner in the vgiprocessor and the I/O processor.

The schematic of the queue is shown in Figure 12, where r0 and r1 are queue registers. When there is no pipeline stall, data will always be placed to the head of the queue, r1. The 2-to-1 mux between r0 and r1 is used to bypass r0, to allow data from memlin to appear directly to r1. The second 2-to-1 mux is used to allow data from memlin to go

directly into the pipeline register (mempipelat). Therefore, when no stall pipe occurs, data from memlin can be fetched into r1 and pipeline register at the same time

7.5 Pipeline register (mempipelat)

The pipeline register plays two major roles. During program execution, mempipelat is used to supply stable address and data to the storage array block, shown in Figure 10. It is the pipeline register of the memory module. Like the data queue, mempipelat consists of two 8-bit register for read and write address and one 16-bit register for write data. The read address register and write data register are also part of the scan chain register and they are used to load data into the storage array during scan in process and to read out the memory content during scan out process. Data register is also used to initialize the read and write pointer for FIFO and delay-line operation.

The two major rolls of the mempipelat block are explained in the next two subsections using the control signals of a cluster and two scan chain register bits.

7.5.1 Mempipelat as scan chain register.

During the scan process data can be loaded into the storage array (e.g. sine or cosine table) or read out the memory content and initialize the address pointers. To store data in the storage array, data and address are scanned into data and address registers. Read/Write enable (rwen) bit and memory update (memupdate) bit of the scan chain register should be asserted as well. When scan update (supdate) control signal comes, data in the scan register will be loaded into the location specified by the address register. Table 1 summarizes the operation of mempipelat as scan chain register

TABLE 1. Summary of mempipelat as scan chain register.

rwen	mem update	Operation
0	0	Read the content of a location in storage array.
0	1	Read the content of counter, read and write pointer

TABLE 1. Summary of mempipelat as scan chain register.

rwen	mem update	Operation
1	0	Initialize counter, read and write pointer
1	1	Store data into a location in storage array.

To read out the content of a memory location, first address has to be scanned in, read/write enable bit (rwen) and memory update (memupdate) bit should be disabled. When supdate signal comes, data in the address location will be loaded into data register which can be scanned out later.

Data register is also used to initialize read and write pointer for FIFO and delay-line operation. Read and write pointers are 8 bits wide each, while data register is 16 bits wide. Cascading read and write pointers, we get 16 bits wide data. Therefore, by connecting the upper byte of data register to write pointer register and the lower byte to read pointer register, we can initialize the pointer registers. Memory update (memupdate) bit of the scan chain is used to distinguish this operation from loading data into storage array. When pointer initialization is intended, memupdate bit is not asserted. Read/Write enable (rwen) bit, however, has to be asserted.Mempipelat as pipeline register

The schematic of mempipelat register when operated as a pipeline register is shown in Figure 13. As pipeline register, mempipelat is used to supply stable address and data to the storage array unit. Whether data in mempipelat can advance or not is controlled by advance signal which comes from queue controller block. Advance signal is asserted when neither stall pipe nor save signal is asserted. Save signal comes from access controller and it is asserted when the access controller cannot serve a request. The section on access controller explains more about the function of this signal. Address output of mempipelat can come from two sources. In the RAM and Look-up table operation, this address comes from Level-1 buses. In FIFO and Delay line operation the address comes from read or write pointer

7.6 Storage Array Block

The 256x16 words SRAM in Figure 10 is the storage array of memory module. This storage array is the self-timed SRAM designed by Burstein [8]. It has separate buses for

input data and output data and a single address bus. The storage consists of memory cells and a dummy circuit which mimics the worst case delay in the memory block. This dummy circuit generates a done signal which indicates that a read/write has been completed. This signal is intended for asynchronous design. In this design, only the memory cell array is used. The done signal is not used. Although the prototype has a 256 X 16 SRAM, a generator is available [9] for extending the size of the storage array.

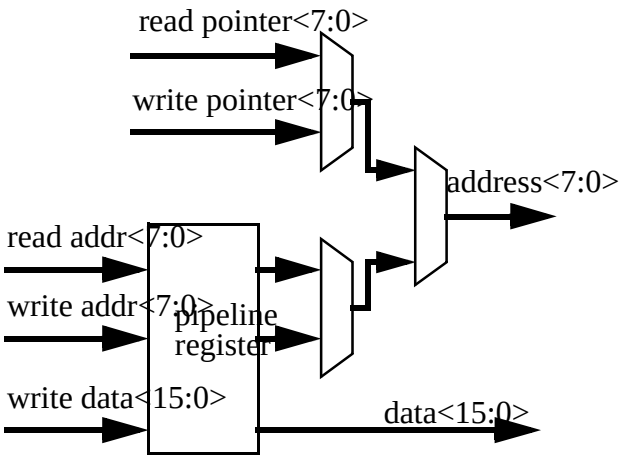


FIGURE 13. Mempipelat schematic as pipeline register. Address source can be read/write address from Level-1 buses (RAM and Look-up Table

7.7 Memlout Block

Memlout consists of a data register to hold data read from storage array and switches that connect to Level-1 buses. Data in the register can be driven to three of the six Level-1 buses so that it can be sent to other processors. Level-1 output switches are configured using scan chain register bits (memscno<2:0>). When a receiver is not ready to receive data in the memlout register, then memlout block will preserve its data to be resent the next cycle.

Output of memlout is also extended to bypass buses. Therefore, if receiving processor is next to the memory module, the data transfer can be done through the bypass buses which free up the Level-1 buses for other processors.

7.8 Pointer and Counter

This block, shown in Figure 14, consist of two up-counters for read pointer and write pointer, one up/down-counter to count the number of data stored in the array, and memflag unit which outputs fullnn and emptynn signal when the memory is full or empty. Read pointer is incremented every time a read request is processed and no stall pipe is present. The same is true for write pointer. It is incremented every time a write request is processed. The pointers are always pointing to the next read and write address to be read and written. Counter is incremented every time a write is executed and decremented when a read is executed. Memflag is used to indicate that FIFO or delay line is full or empty. Full is asserted when counter value is the same as the value of delay set in the scan chain register. For FIFO, in order to use the whole memory as FIFO, one would have to specify the number of delay to 256. Shorter FIFO can be specified by setting the value of delay in the scan chain to be less than 256. Empty is asserted when the counter value reaches one and read request is being executed. The read and write pointer blocks can be initialized during scan chain as described in the mempipelat section (Section 7.5).

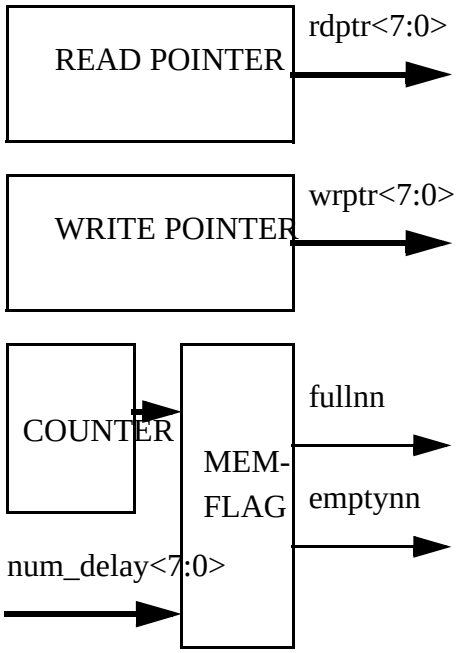


FIGURE 14. Block diagram of the pointer and counter block of memory module. Num_delay<7:0> are scan chain bits that set the size of FIFO and Delay line. Fullnn indicates that FIFO/Delay line is full. Emptynn indicates that FIFO is empty.

7.9 Controller

The controller of memory module (see Figure 10) can be divided into three major blocks, queue controller, memory access controller, and output controller. A block diagram of the controller is shown in Figure 15 with all the control

signals and communication blocks. The control signals are used in the FSMs of the major blocks. The signals gstall and clk are global control signals. The function of each block is described in the subsections.

7.9.1 Handshake queue

Each bus carrying data tokens or control tokens has a handshake signal associated with it. The handshake signal also carries information about what operation to be performed to the data - read or write. Since data in the datapath goes through a queue, there needs to be the same queue for the handshake signal so that the data and its associated handshake signal can propagate in parallel in the memory module.

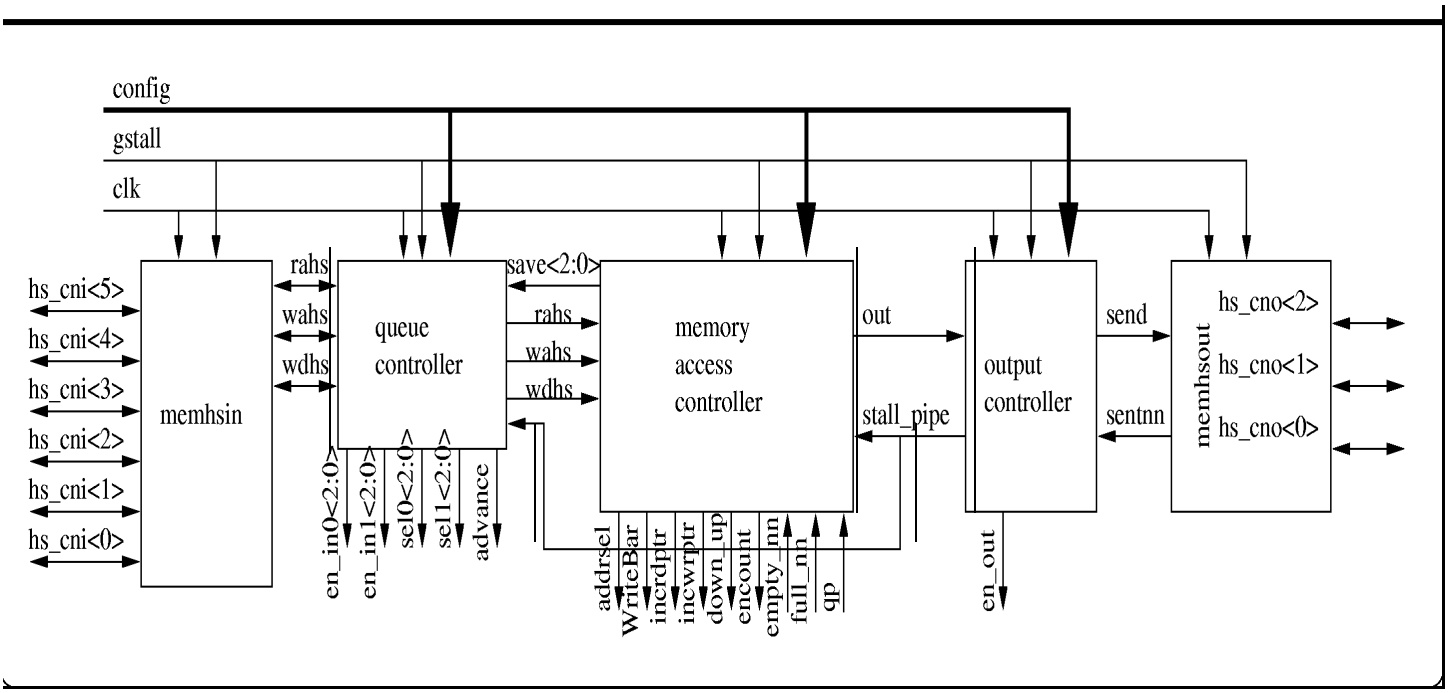


FIGURE 15. Block diagram of the controller unit of memory module.

7.9.2 Queue Controller

The queue controller decides where the data in the memlin register will be stored in the data queue block. If there is no stall pipe, data will be placed at the head of the queue. There are 3 queue controllers, one for read address, one for write address and the other for write data. Queue controllers also decide whether an incoming data can be accepted. When a queue is full, the queue controller will pull down the handshake line to indicate to the sender that no data can be accepted. The states of the queue controller are shown in Figure 16. Please refer to Figure 12 in reading the state diagram.

Advance signal is generated using the following logic:

$$advance = \overline{(stallpipe + save)},$$

where save is a signal from access controller that tells the queue controller that the data at the head of the queue has not been serviced because another process is being executed. As an example consider a situation where read and write requests arrive simultaneously at the access controller. If the priority is given to write over read, then write will be performed and save will be generated for read address queue controller.

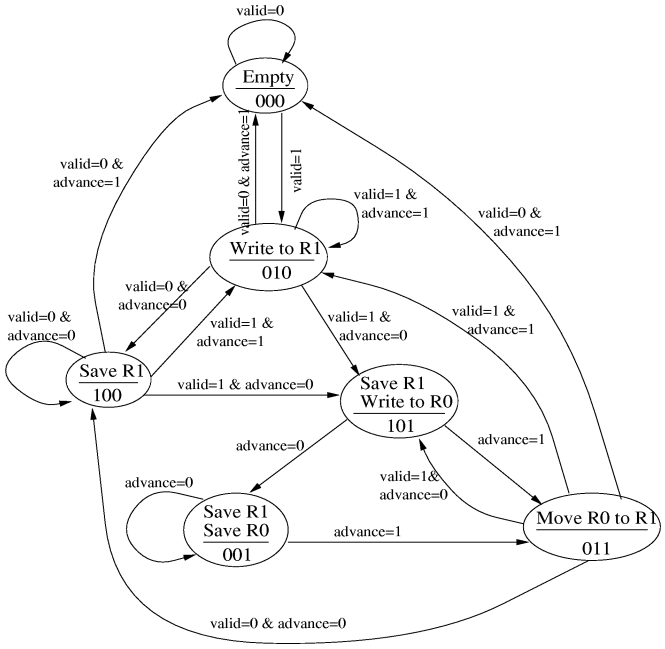


FIGURE 16. Queue controller state transition diagrar
Memory module controller consists of three controlle

7.9.3 Memory Access Controller

The memory access controller is the key block of the memory module. The four modes of the memory module are implemented in this block. The operation mode of memory access controller is configured using configuration (config<1:0>) bits of the scan chain register. Memory access controller communicates with the queue controller using the save signal. Whenever a request (read/write) cannot be serviced by memory access controller, it will assert the save signal. This will result in advance signal being unasserted and the data in data queue and pipeline register blocks will be preserved. For example, if the memory access controller receives simultaneously read and write requests, depending on the priority, either read or write will be selected and the other request has to wait. If write is performed then save signal for read queue controller will be asserted and vice versa for read.

The memory access controller communicates with the output controller through the out signal. When a read is performed, out signal is asserted, signaling the output controller to prepare to send out the data read from storage array. The four modes of the memory access controller are described using FSMs in the next four subsections. The control signals shown in Figure 15 are used in deriving inputs that cause state transitions.

7.9.4 RAM Mode

When configured in RAM mode, memory access controller treats the storage array as a random access memory. It can accept any read and write operation. Simultaneous read and write request is resolved using queue priority bit of scan chain register. This means that the user determines the read or write priority. When queue priority bit is asserted, priority is given to read request and when the bit is low, priority is given to write request. For a write request to be executed, both write address and write data have to be available to the access controller.

7.9.5 Look-up Table Mode

The Look-up Table (LUT) mode is similar to RAM mode except that write is not allowed during execution. In the LUT mode, storage array is loaded with a table during the scan in process. The procedure for scanning is described in mempipelat section (Section 7.5).

7.9.6 FIFO Mode

In FIFO mode, addresses for read and write come from read and write pointer blocks. Only the write data comes from Level-1 buses. Read request is communicated using read handshake line and write request is communicated using write data handshake. Since write address comes from write pointer it is assumed to be always ready. So, in FIFO mode, write address handshake signal is ignored.

Simultaneous read and write request can occur in FIFO mode. This is resolved using the queue priority bit like in the RAM mode. Fullnn and emptynn signal from memflag block (Figure 14) signal the access controller when the FIFO is full or empty. Full occurs when the counter value reaches the number of delay in the scan chain register. Empty occurs when the counter value reaches one and the access controller is in read mode.

7.9.7 Delay Line Mode

Like the FIFO controller, the sources for the addresses of delay line controller are read and write pointer blocks (Figure 14). What makes delay line different from FIFO is that delay line controller does not accept read request. Read will be executed automatically once the delay line is full. This is indicated by fullnn signal from memflag block (Figure 14). Write address is assumed ready every time write data is available. So, write request is communicated using write data handshake signal only. Since simultaneous read and write is not possible when the controller is

operating in delay line mode, the queue priority bit is simply ignored. r.

7.10 Output Controller

The output controller, shown in Figure 15, receives the out signal from memory access controller every time access controller is in read state. When the request comes, the controller will assert the output handshake line to signal the receiver of outgoing data during clock low. When clock high comes, the output controller observes the handshake line to determine if the send is successful. If the handshake line is pulled down any time during clock high, the send is unsuccessful and the controller will try to resend the data the next cycle. In this case stall_pipe signal is asserted for the memory module which will stall other activities.

7.11 Scan Chain Register

Programming the processors in a cluster is done using a scan chain. The scanning process is described using the memory module. The memory module scan chain register is 55 bits long. It is divided into three blocks, memscni (26 bits), mempipelat (26 bits), and memscno (3 bits). The physical location of the bits of the scan chain register is scattered among the blocks of the memory module. Memscni contains 6 bits for selecting Level-1 network's six buses, 3 bits for activating bypass buses to supply data tokens, one bit for selecting read/write to counter, 3 bits for selecting read/write to queue register, 2 bits for selecting read and write address source, 2 bits for operation mode selection (00 = RAM mode, 01= look-up table mode, 10 = FIFO mode, 11 = Delay line mode), 1 bit for queue priority assignment, and 8 bits for FIFO or delay line size. Scan register in mempipelat consists of 8 bit address register, 16 bit data register, read/write enable (rwen) bit, and memupdate bit. Address and data register are used to load storage array with data or to read out content of memory. The second function of the data register is to initialize read and write pointer register in the datapath's counter unit. Read/Write (rwen) bit indicates that the data in data register is to be loaded either into memory or pointer register. Together with read/write bit, memupdate indicates that the data in data register is to be loaded into storage array. Memscno block contains 3 bits to control Level-1 output select switches. The operation of the scan chain register is controlled by two cluster level control signals sen and supdate that are generated from primary input signals gstall, tms, reset, and clk.

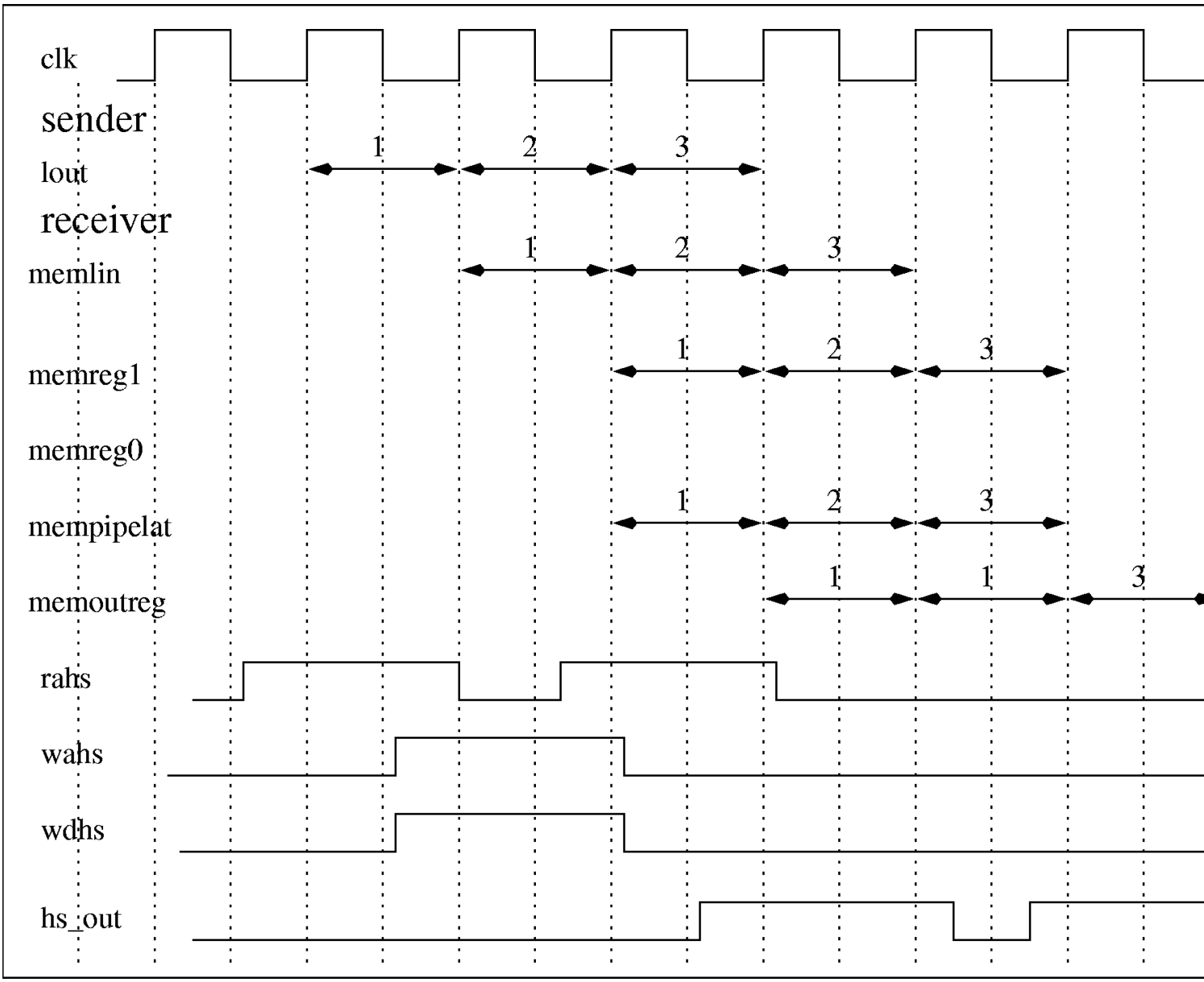


FIGURE 17. RAM Data propagation through memory module datapath.

7.12 Design Verification

All the blocks of the memory module have been designed and functionally simulated at the RTL level using a VHDL simulator. Many blocks in the control part of the memory module have been synthesized. The remaining blocks have been custom designed. The physical layout of the entire memory module has been completed using custom **layout** tools and it is included in Figure 18. The storage array occupies the two quadrants on the right side of the layout. The datapath occupies the bottom left quadrant and the control occupies the top left quadrant. The physical layout has been extracted, verified, and a transistor level netlist generated for use with Timemill simulation. The operation of the memory module has been verified using Timemill simulations.

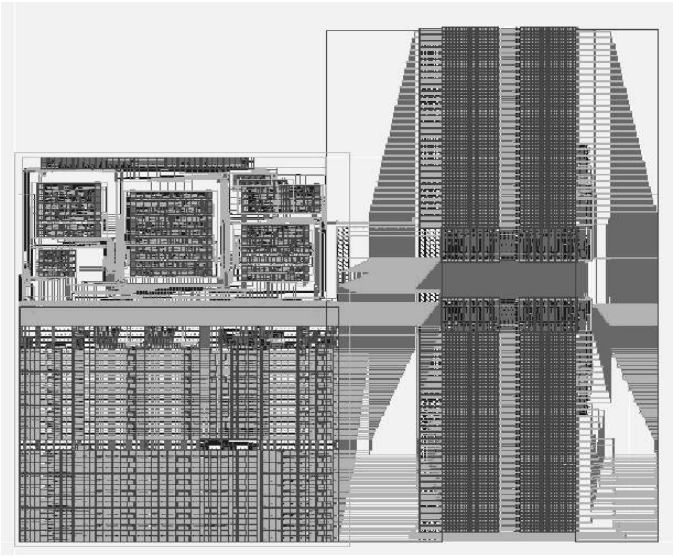


FIGURE 18. Layout of Memory Module

The following example illustrates the data propagation in memory module. We assume the memory module is configured in RAM mode (memconfig = 00). There are three requests that arrive consecutively to the memory module. They are named 1,2 and 3 in Figure 17. Request 1 is a read operation, request 2 is a write operation and request 3 is a read operation. The three requests can come from a vgiprocessor, another memory module, or an I/O processor. For read request, output data is sent out to another processor and the receiving processor is assumed ready to receive the data. Figure 15 shows the timing diagram of data through the memory module's datapath.

During the communication cycle, data 1, are driven onto Level-1 buses to the input of memory module. These data are copied by memlin and driven to data queue in the fetch cycle of memory module. Data 1 is a read request as indicated by the rahs line. Since no data are yet stored in the queue, data 1 will advance to memreg1 and mempipelat. At the same time, memlin is accepting the next data 2. Data 2 is write request as indicated by both wahs and wdhs line. For write request, both wahs and wdhs have to be high to be processed. The next cycle, mempipelat drives data 1 request to storage array and memory read is executed. Data 2 are advanced to memreg1 and the third read request, data 3 are communicated. At the beginning of the next cycle, data 1 result appear at the output register. These data are being driven on Level-1 bus to a receiving processor. Notice that output handshake line (hs_out) has been pulled high during clock low in the previous cycle to tell the receiver processor of the arrival data 1 result. At the same time, write request of data 2 is executed and data 2 operation is completed. Data 3 are transferred to memreg1 and mempipelat during this clock cycle. Since data 2 do not produce any output, hs_out is pulled low by the output controller at the next clock cycle to tell the receiver that no data is being communicated. On the same cycle data 3 is executed and the output is stored in the output register. The output handshake line again is asserted when clock goes low to inform the receiver of the availability of data. During the next cycle, data 3 results are communicated to the receiver.

In this example, we can clearly see the latency of read and write requests. A read request, indicated by data 1 in the example, incurs two clock cycle latency. When data 1 appear at the output of memlin during the first clock cycle, data 1 is transferred to mempipelat. At the second clock cycle, the memory read is performed. At the end of this clock cycle, the data 1 result is available at the output register which will be sent to the receiver when the next clock cycle comes. A write request, indicated by data 2, only incurs one clock cycle latency. When data 2 is available at the output of memlin during the first cycle, data 2 is transferred into mempipelat. At the second clock cycle, the memory write is performed and the request is completed.

8.0 I/O Processor

In a cluster of RAMP, external I/O handling is separated from the vgiprocessors and memory module by the I/O processor. The I/O processor is designed to handle off-chip communication functions, which would otherwise require one of the vgiprocessors as in PADDI-2 [1]. In the case of PADDI-2, the processors had a small instruction set and took a small area. The VGI instruction set is more extensive, so using a vgiprocessor for I/O functions would be a waste of resources. The I/O processor also handles the fact that the off-chip communication protocol can be different from the on-chip communication protocol. To accommodate commercial SRAM chips and other VGI-1 chips, two cycles are allowed for data and control communication.

There is one I/O processor and one memory processor for every four vgiprocessors in a cluster. The I/O processor behavior is set by the scan chain to either take data from an on-chip vgiprocessor and output to a pad, or the other way around. The I/O processor also has a two deep data buffer in case communication is stalled down stream. Figure 19 shows the major blocks of the I/O processor.

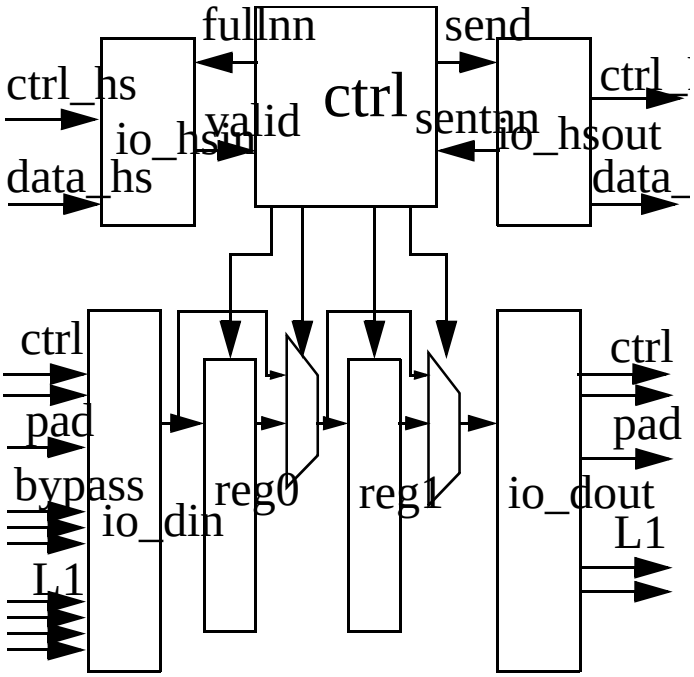


FIGURE 19. Block Diagram of I/O Processor

8.1 Description of I/O Processor Blocks

The datapath of the I/O processor has four major blocks, i0_din, reg0, reg1, and io_dout. The control part of the processor has three major blocks, io_hsin, is_hsout, and ctrl. The two register blocks, reg0 and reg1, were taken directly from the vgiprocessor design. The blocks io_din and io_dout are multiplexers that direct the incoming and outgoing data and control signals as determined by the switch settings. The blocks io_hsin and io_hsout are adaptations of hsin and hsout of the vgiprocessor, respectively. A series of latches with a small amount of logic is included to intercept their communication signals to the ctrl block. These latches are necessary to adapt to the longer delay of the off-chip communication protocol. Since the I/O processor can be configured to take data from off-chip, or to output data off-chip, these latches are enabled or disabled during setup by the sel switch. (See Figure 20).

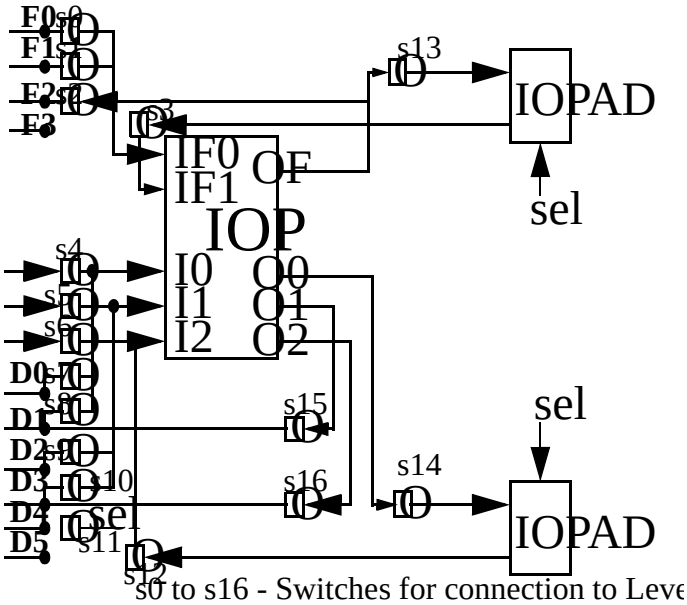


FIGURE 20. I/O Processor

The ctrl block is actually composed of six blocks, as shown in Figure 21. The five qctrl blocks were also taken directly from the vgiprocessor design. They keep track of the states of the registers, determining whether the queue is full or empty, and if data needs to be read or written. The block called ioplogic takes the information it receives from qctrl and io_hsout, and decides if there is data to be sent, and whether or not the communication line is stalled.

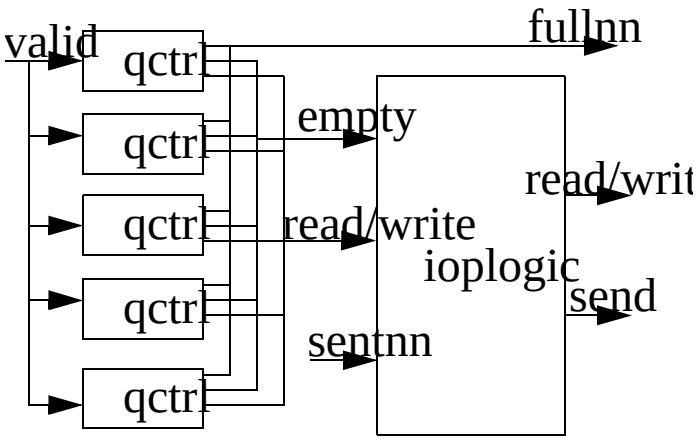


FIGURE 21. Block Diagram of ctrl

The input and output ports of an I/O processor can be connected to the Level-1 network, bypass buses, or pads. There are 17 switches that are configured using a scan chain for each I/O processor. A layout of the I/O processor is shown in Figure 22.

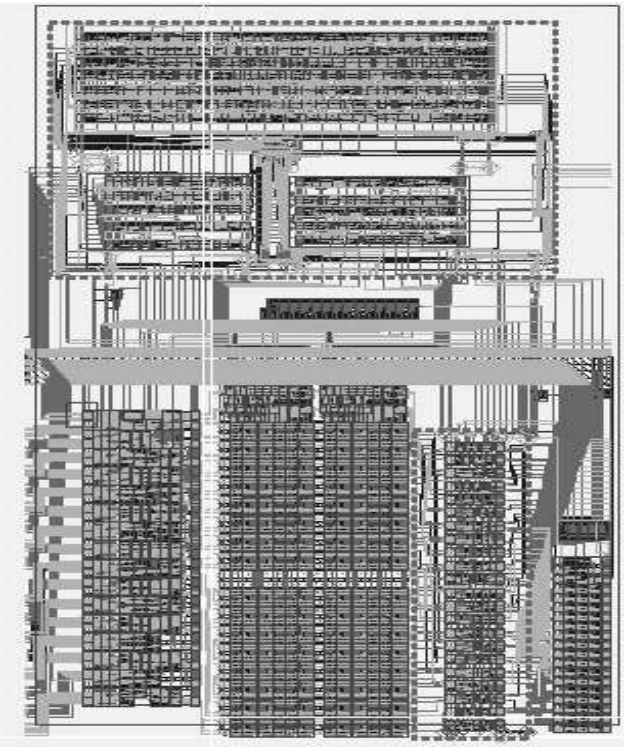


FIGURE 22. Layout of I/O Processor

9.0 Simulator

A functional simulator has been written in Java to assist with program creation and verification for RAMP. Program development using assembly code, then assigning the code to processors and setting route switches manually works fine for small programs, but it scales badly because resource allocation has to be tracked by hand. The simulator is designed to alleviate these limitations and give fast feedback to the programmer. It can be used as a verification tool, allowing the user to load a previously created program, then step through it and view each vgiprocessor's state. It is also a program creation tool with a GUI for assigning processors and finding routes. Development is still a trial-and-error process. If the programmer gets a resource limitation, he/she must back out of the current approach and try a different mapping. Keeping track of resources by hand is almost impossible, so there is a need for tools to keep track of resources and assist with the routing process. The simulator consists of a user interface package, a set of utilities package, a package for representing common features of the parallel architecture, and a RAMP chip-specific package. The simulator is designed so that new parallel architectures can be supported by simply changing the RAMP chip-specific package. The number of clusters in a chip, the number of processors in a cluster, and the types of processors in a cluster can all be changed. The details of the packages forming the simulator are described in the following sections.

9.1. User Interface

The simulator user interface shows a hierarchical display of the first version of RAMP, called VGI chip, on the left, and a display desktop on the right in Figure 23. The tree display shows the 'VGI chip' node, which has 12 'cluster' nodes, an 'input pad' and an 'output pad.' Each cluster node has 4 'vgiprocessors.' Each tree node is associated with a display panel on the right. The displays embed according to the tree structure. Displays include:

VGI chip: Can be used to load and save a program, and to dispatch I/O Code Interfaces:

.set menu: 'Load' reads in the program configuration from a *.set file, which assigns programs to nanoprocessors and sets switches for routing. 'Reload' reads the last loaded file. 'Unload' frees all processor and switch allocations. 'Save' sends the configuration to a.set file on disk.

CI (Code Interface) menu: This is currently the method for simulating a program on the host computer, as it exchanges data with the chip's input/output pads. 'Load' reads and launches a code interface from disk, which is a class file which conforms to VGI_CodeInterface. This class performs operations for displaying input and output data, such as drawing images side-by-side. 'Reload' reads the last loaded file. 'Unload' disables the CI.

VGI nanoprocessor: Displays the state and port information of a nanoprocessor.

File menu: 'Load' reads the.s (assembler) file, and will assemble if the assembler library is present. 'Reload' reads the.bit file. 'Unload' deallocates the processor. 'Edit' pulls up the procressor's assembler code in an editor, which will assemble and update the code when it is saved.

State Panel: displays the current state information of a vgiprocessor. The current state information consists of values for program counter, next program counter, condition code register, flag inputs, scratch-pad registers (ra, rb, rc), general purpose registers (r1 to r6) and communication registers (c0,c1,c2). It also contains information on whether register pairs are configured as registers (big 'R') or queues (big 'Q') (green means connected, red means disconnected).

Port Panel: displays the interconnections of a selected processor. The vgiprocessor has 9 ports (3 data-in,2 flag-in, 3 data-out, 1 flag-out), which are enumerated on the left. If the port has a connection, the connected port will be displayed to the right. Clicking a connection draws a trace to the other element to which it connects. The port panel is also used for routing, which is described below.

Input/Output pads: each has a pad panel, which shows its one internal buffer register, and the number of values communicated. The port panel acts like the nanoprocessor's port panel, with tracing and routing operations. Currently the input pad is hardwired to cluster #6, and the output pad to cluster #4 (making routes possible between ipad and I2-p3c4, and between O1-p3c6 and opad). Currently, the I/O pads are structured internally to the VGI chip, though they will be moved externally when a "board" level of hierarchy is configured. At this point the user will be able to specify the bindings between pads and clusters.

Most levels of hierarchy have a popup menu that is activated by right clicking on the tree node, or on the display if it is showing. For instance, the VGI chip's popup has a "Show Active" function that shows all vgiprocessors that have been assigned programs.

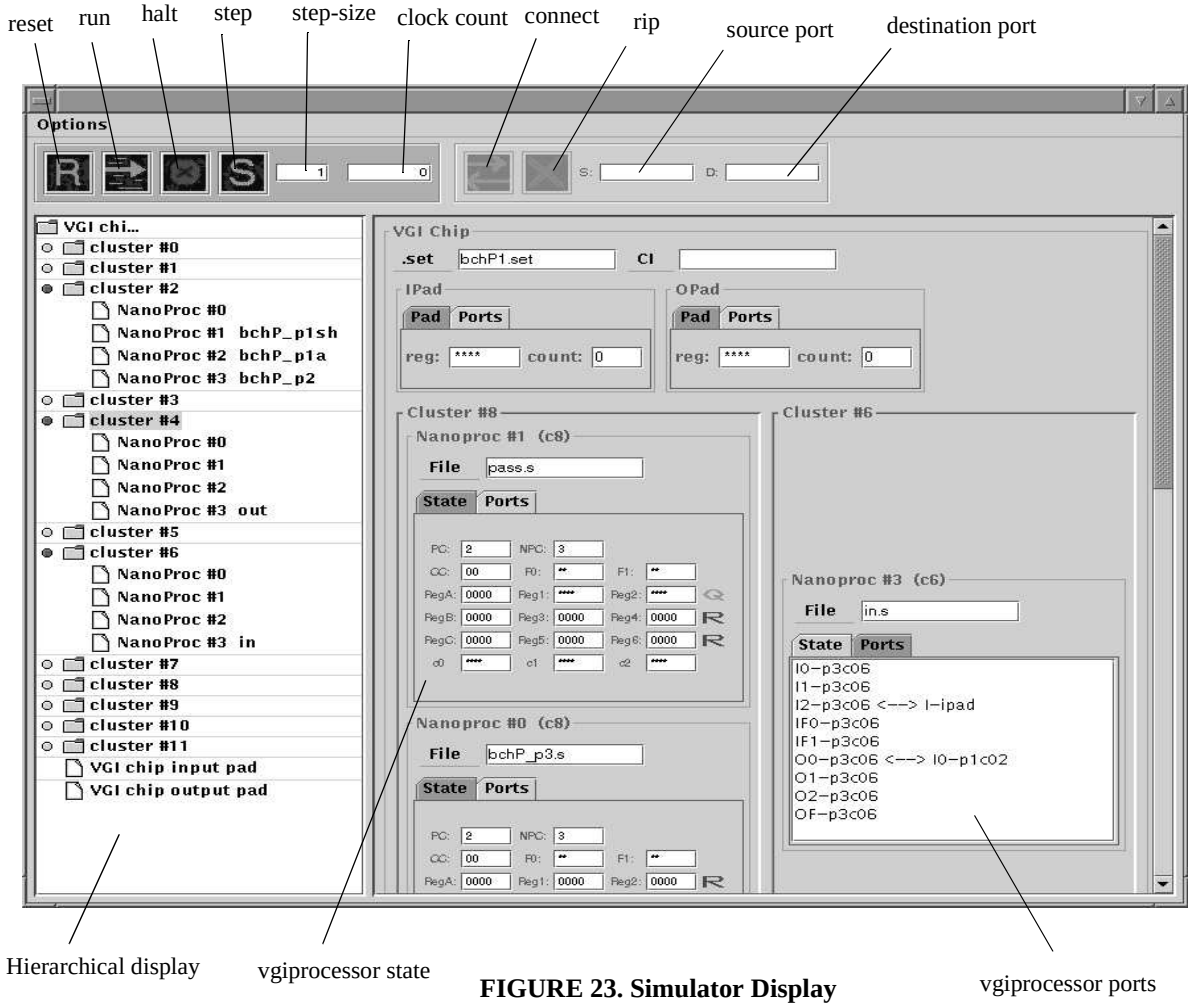


FIGURE 23. Simulator Display

9.1 Program Simulation

The simulator supports program simulation with a toolbar. The toolbar at the top of the VGI simulator panel, shown in Figure 23 has the following actions:

reset: performs a reset operation on all subnets of the VGI chip. For instance, performing reset on a nanoprocessor restores state information from the scan chains.

run: run simulation freely. Note: State information is not updated until a halt is called (it takes too long to redraw).

halt: stops simulation and updates state displays.

step: cycles the number of steps indicated in the adjacent field, and updates state information when finished.

The simulator was used in the design of many programs for applications. The signal flow graph (SFG) for doing a 3 X 3 convolution on a 512 X 512 is shown in Figure 24. The image is supplied as scan lines. To do a 3 X 3 convolution, two scan lines are stored in two delay lines, each having a 512 stage delay. The scan line data is supplied on the top left corner in the figure. The multiplication of pixel values by coefficients is done in the blocks. The result is supplied as a sequence of computed pixel values on the bottom right of the figure. The architecture flow graph (AFG) using VGI chip for the signal flow graph in Figure 24 is shown in Figure 25. The AFG uses 38 vgiprocessors, four memory processors, and two I/O processors.

9.2 Partitioning, Placement, and Routing

The AFG is partitioned using an algorithm similar to the one proposed by Fiduccia and Mattheyses for partitioning nets. It starts with a random partition and then swaps to minimize a cost function. In the case of a RAMP cluster, the cost is defined as the number of Level-1 buses used in a cluster for optimal placement. Chains of processors is considered good since they can use bypass buses (connection between adjacent processors) in a cluster.

The algorithm to place clusters uses swaps to reduce the total communication distances between clusters. This reduces the number of Level-2 bus resources since locality is exploited. After the partitioning and placement of the nodes of the AFG to clusters and processors, detailed routing has to be done.

A simple tool is provided for routing and a sample display is shown in Figure 26. To make a connection, left click on the source port and right click on the destination port (in the port panels of a vgiprocessor or an I/O pad). Neither of these ports can have previous connections. Press the ‘Connect’ button to open a dialog that displays all available routes between the two ports. The search depth can be modified. The default is 4, which is not sufficient for several connections over the Level-2 network. Searching at depth 5 takes about 2 seconds on a fast computer. A search at a depth of 6 or more can take several seconds. Detailed routing has to be done for each cluster with rip up and routing as needed. The placement and routing of a cluster using the routing tool is shown in Figure 27.

9.3 Simulator Design

The simulator is written in Java for portability. The nature of the Java language has proven very useful for design, although experience has shown that the APIs contained in Java Development Kit (JDK) have several problems. A good portion of the simulator code has no knowledge of the VGI chip architecture. It might be possible to construct a simulator for another architecture without rewriting several data structure and user interface classes.

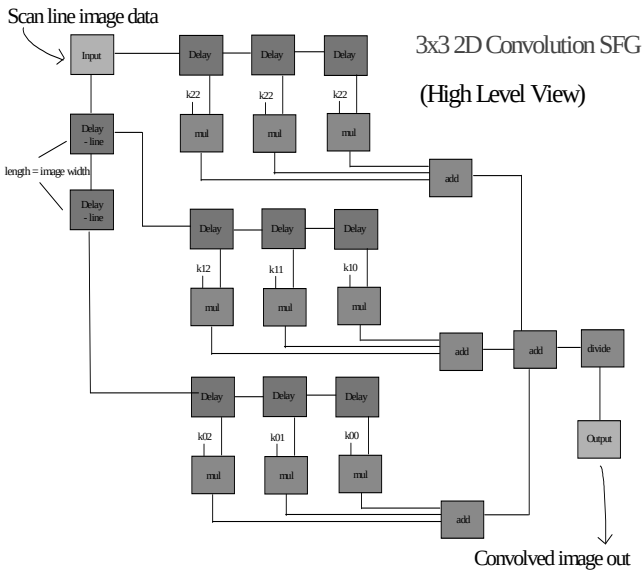


FIGURE 24. 2D Convolution Signal Flow Graph

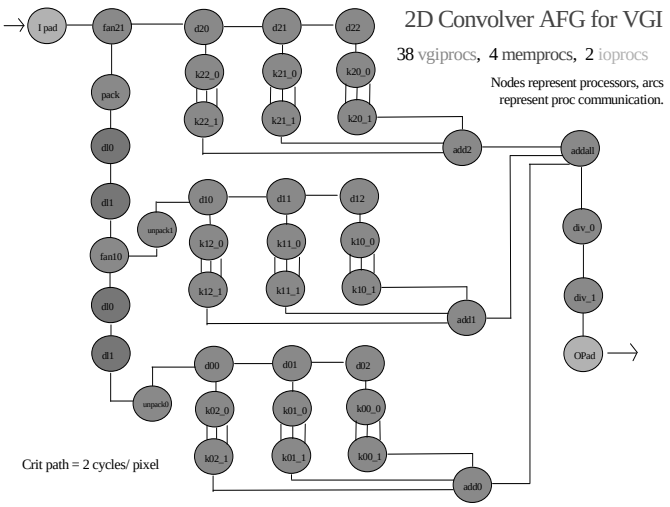


FIGURE 25. Architecture Flow Graph for 2D Convolver

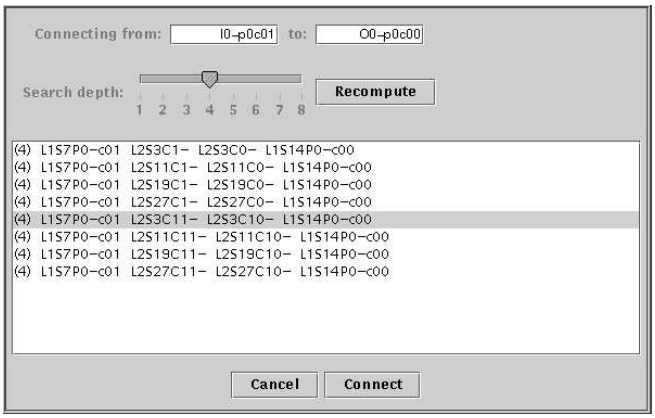


FIGURE 26. Router Display

10.0 Extensions to RAMP

The key elements of the RAMP architecture have been described in this paper. An implementation of the architecture with 16 clusters has been completed and programs have been mapped to the architecture. Applications such as 2D convolution with 3 X 3 kernel and an image size of 512 X 512 pixels can be processed at the rate of one pixel every two cycles. A median filter (nonlinear filter) with a 3 X 3 kernel and an image size of 512 X 512 pixels can be processed at the rate of one pixel every four cycles.

The computation processors in RAMP are homogeneous and tailored to a class of applications. Several extensions are possible to improve performance and reduce power. The first extension relates to the partitioning strategy. If partitioning is based on functions used and same data rate for the functions, then a direct mapping of functions to hardware can be done using a library of hardware components and a compiler, analyzer, generator, and elaborator (CAGE) software. The processors in a cluster can be replaced by processors that are matched to applications. For example, an application can be partitioned so that each cluster of connected nodes is mapped to a cluster. This could be done on a functional basis. The components needed to implement a function can be determined by analyzing the program for the function using CAGE. A direct mapped processor(s) for the function can be assembled using the components and CAGE. The second extension addresses the communication between clusters of RAMP. The first implementation of RAMP uses a simple handshake protocol and a single clock that is distributed to all the clusters. Although clusters can be reconfigured individually they cannot operate at different clock frequencies since the switches used in the Level-1 and Level-2 networks have no buffers and synchronizers. One approach to overcome this limitation is to use a two-phase communication protocol [10] and asynchronous FIFOs in the switches when communicating between clusters. This approach allows globally asynchronous and locally synchronous (GALS) [11] communication in RAMP. The GALS approach has been used in other systems such as the Pleiades [12, 14] and Maia [13, 14]. The communication between clusters in the RAMP takes place using Level-2 buses and switches connecting Level-1 buses to Level-2 buses. Situations requiring clusters to operate at different clock rates (multiple clock domains) can be mapped to RAMP if the GALS clocking scheme is used.

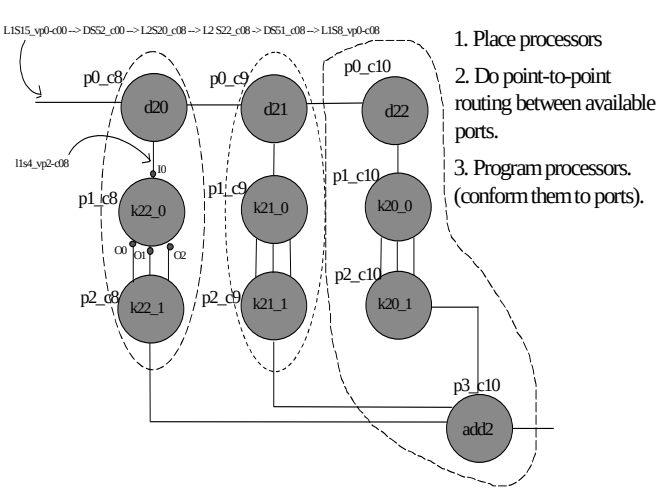


FIGURE 27. Placement and Routing of Processors in Clusters

Commercial systems such as Mercury Computer's Race++ [15] can use the RAMP architecture to enhance the performance of medical imaging systems [16, 17], UAV post-processing, tactical SAR processing, and 3-D image reconstruction.

Acknowledgment

The authors wish to thank John Thendean, David Pini, and S. Ueng for their work on the memory, processor, and I/O processor respectively. The Java simulator for clusters and the core of RAMP was done by Matt Armstrong and the authors thank him for the effort. Roy Sutton, Brian Richards, and many others contributed ideas to get around problems during the design. The authors are grateful to Bob Brodersen for his support and encouragement at the Berkeley Wireless Research Center (BWRC). Additional thanks go to Gary Kelson and Kevin Zimmerman and members of BWRC.

11.0 References

1. R. Schreiber, et al, "PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators", Journal of VLSI Signal Processing Systems, Vol. 31, No. 2, June 2002.

2 A. K. W. Yeung, "PADDI-2 Architecture and Implementation," Ph.D. Thesis, University of California, Berkeley, CA, June, 1995.

3. P. Lapsley, J. Bier, A. Shoham, E.A. Lee, "DSP Processor Fundamentals: Architectures and Features," Berkeley Design Technology, Inc., Berkeley, CA, 1994, Chapter 5.

4. E. C. Ifeachor, B. W. Jervis, "Digital Signal Processing: A Practical Approach," AddisonWesley, Wokingham, England, 1993.

5. V. P. Srini, R. A. Sutton, J. M. Rabaey, "Multiple Processor DSP System using PADDI-2", Proceedings of Design Automation conference, San Francisco, CA June, 1998.

6. D. M. Pini, "A Parallel Processor for High Performance Digital Signal Processing," Master's Thesis, University of California, Berkeley, CA, May, 1997.

7. V.P. Srini, J. Thendean, S. Ueng, J. M. Rabaey, "A Parallel DSP with Memory and I/O Processors", Proceedings of the SPIE Conference, San Diego, CA, July, 1998

8. A. J. Burstein, "Speech Recognition for Portable Multimedia Terminals", ERL Memorandum No. UCB/ERL M97/14, Feb. 1997, University of California, Berkeley, CA.

9. N. Walker, "Integrated Circuit Module Generator", Master's Thesis, University of California, Berkeley, CA, May, 1999.

10. C. Mead and L. Conway, "Introduction to VLSI Systems", AddisonWesley, Reading, MA, 1980.

11. A. Iyer, and D. Marculescu, "Power and Performance Evaluation of Globally Asynchronous Locally Synchronous Processors," The 29th Annual International Symposium on Computer Architecture, Anchorage, Alaska, May 2002, pp 158 - 170.

12. A. Abnous, "Low-Power Domain-Specific Processors for Digital Signal Processing," Ph.D. Thesis, University of California, Berkeley, CA, June, 2001.

13. M. Wan, "Design Methodology for Low Power Heterogeneous Reconfigurable Digital Signal Processors," Ph.D. Thesis, University of California, Berkeley, CA, June 2001.

14. M. Benes, "Design and Implementation of Communication and Switching Techniques for the Pleiades Family of Processors," M.S. Thesis, University of California, Berkeley, CA, June 2000.

15. Mercury Computer Systems Inc., http://www.mc.com/technology_corner/race_plus_plus.cfm

16. M. Trepanier, et al, "Adjunct Processors in Embedded Medical Imaging System," http://www.mc.com/literature/literature_files/MI4681_66_MercTrepanier.pdf

17. I. Goddard, et al ,"High-speed Cone-beam Reconstruction: an embedded systems approach," www.mc.com/literature/literature_files/MI4681-50_MercGoddard.pdf