FCCM'96 -- IEEE Symposium on FPGAs for Custom Computing Machines
April 17-19, 1996, Napa, CA

MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources

Ethan Mirsky
eamirsky@ai.mit.edu
(617) 253-8597
NE43-793

André DeHon
andre@mit.edu
(617) 253-5868
NE43-791

545 Technology Sq.
Cambridge, MA 02139
FAX: (617) 253-5060

Abstract

MATRIX is a novel, coarse-grain, reconfigurable computing architecture which supports configurable instruction distribution. Device resources are allocated to controlling and describing the computation on a per task basis. Application-specific regularity allows us to compress the resources allocated to instruction control and distribution, in many situations yielding more resources for datapaths and computations. The adaptability is made possible by a multi-level configuration scheme, a unified configurable network supporting both datapaths and instruction distribution, and a coarse-grained building block which can serve as an instruction store, a memory element, or a computational element. In a 0.5μ CMOS process, the 8-bit functional unit at the heart of the MATRIX architecture has a footprint of roughly 1.5mm$\times$1.2mm, making single dies with over a hundred function units practical today. At this process point, 100MHz operation is easily achievable, allowing MATRIX components to deliver on the order of 10 Gop/s (8-bit ops).

1 Introduction and Motivation

General-purpose computing architectures must address two important questions:

1. How are general-purpose processing resources controlled?
2. How much area is dedicated to holding the instructions which control these resources?

There are many, different possible answers to these questions and the answers, in large part, distinguish the various general-purpose architectures with which we are familiar (*e.g.* word-wide uniprocessors, SIMD, MIMD, VLIW, FPGA, reconfigurable ALUs). The answers also play a large role in determining the efficiency with which the architecture can handle various applications. We have developed a novel reconfigurable device architecture which allows these questions to be answered by the application rather than by the device architect.

Most of the area on a modern microprocessor goes into storing data and instructions and into control circuitry. All this area is dedicated to allowing computational tasks to heavily *reuse* the small, active portion of the silicon, the ALUs. Consequently, very little of the capacity inherent in a processor gets applied to the problem — most of it goes into supporting a large operational diversity. Further, the rigid word-SIMD ALU instructions coupled with wide processor words make processors relatively inefficient at processing bit or byte-level data.

Conventional Field Programmable Gate Arrays (FPGAs) allow finer granularity control over operation and dedicate minimal area to instruction distribution. Consequently, they can deliver more computations per unit silicon than processors on a wide range of regular operations. However, the lack of resources for instruction distribution make them efficient only when the functional diversity is low — *i.e.* when the same operation is required repeatedly and that entire operation can be fit spatially onto the FPGA or FPGAs in the system.

Dynamically Programmable Gate Arrays (DPGAs) [14] [7] dedicate a modest amount of on-chip area to store additional instructions allowing them to support higher operation diversity than traditional FPGAs. The area necessary to support this diversity must be dedicated at fabrication time and consumes area whether or not the additional diversity is required. The amount of diversity supported – *i.e.* the number of instructions – is also fixed at fabrication time. Further, when regular datapath operations are required many instruction stores will be programmed with the same data.

Rather than separate the resources for instruction storage and distribution from the resources for data storage and computation and dedicate silicon resources to them at fabrication time, the MATRIX architecture unifies these resources. Once unified, traditional instruction and control resources are decomposed along with computing resources and can be deployed in an application-specific manner. Chip capacity can be deployed to support active computation or to control reuse of computational resources depending on the needs of the application and the available hardware resources.

In this paper we introduce the MATRIX architecture. The following section (Section 2) provides an overview of the basic architecture. In Section 3, we show usage examples to illustrate the architecture's flexibility in adapting to varying application characteristics. Section 4 puts MATRIX in context with some of its closest cousins. We comment on MATRIX's granularity in Section 5. In Section 6, we highlight implementation characteristics from our first prototype. Section 7 concludes by reviewing the key aspects of the MATRIX architecture.

2 Architecture

MATRIX is composed of an array of identical, 8-bit functional units overlayed with a configurable network. Each functional unit contains a 256×8-bit memory, an 8-bit ALU and multiply unit, and reduction control logic including a 20×8 NOR plane. The network is hierarchical supporting three levels of interconnect. Functional unit port inputs and non-local network lines can be statically configured or dynamically switched.

2.1 BFU

The Basic Functional Unit (BFU) is shown in Figure 1. The BFU contains three major components:

- **256×8 memory** – the memory can function either as a single 256-byte memory or as a dual-ported, 128×8-bit memory in register-file mode. In register-file mode the memory supports two reads and one write operation on each cycle.

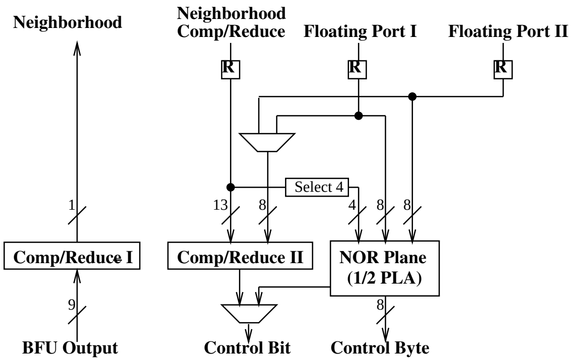


Figure 2: BFU Control Logic

- **8-bit ALU** – the ALU supports the standard set of arithmetic and logic functions including NAND, NOR, XOR, shift, and add. With optional input inversion, this extends to include OR, AND, XNOR, and subtract. A configurable carry chain between adjacent ALUs allow cascading of ALUs to perform wide-word operations. The ALU also includes an 8×8 multiply-add-add operation; the multiply operation takes two operating cycles to complete producing the low 8 bits of the product on the first cycle and the high 8 bits on the second cycle.
- **Control Logic** – the control logic is composed of: (1) a local pattern matcher for generating local control from the ALU output (Figure 2 Left), (2) a reduction network for generating local control (Figure 2 Middle), and (3) a 20-input, 8-output NOR block which can serve as half of a PLA (Figure 2 Right).

MATRIX operation is pipelined at the BFU level with a pipeline register at each BFU input port. A single pipeline stage includes:

1. Memory read
2. ALU operation
3. memory write and local interconnect traversal – these two operations proceed in parallel

The BFU can serve in any of several roles:

- **I-store** – Instruction memory for controlling ALU, memory, or interconnect functions
- **Data Memory** – Read/Write memory for holding data
- **RF+ALU slice** – Byte slice of a register-file-ALU combination

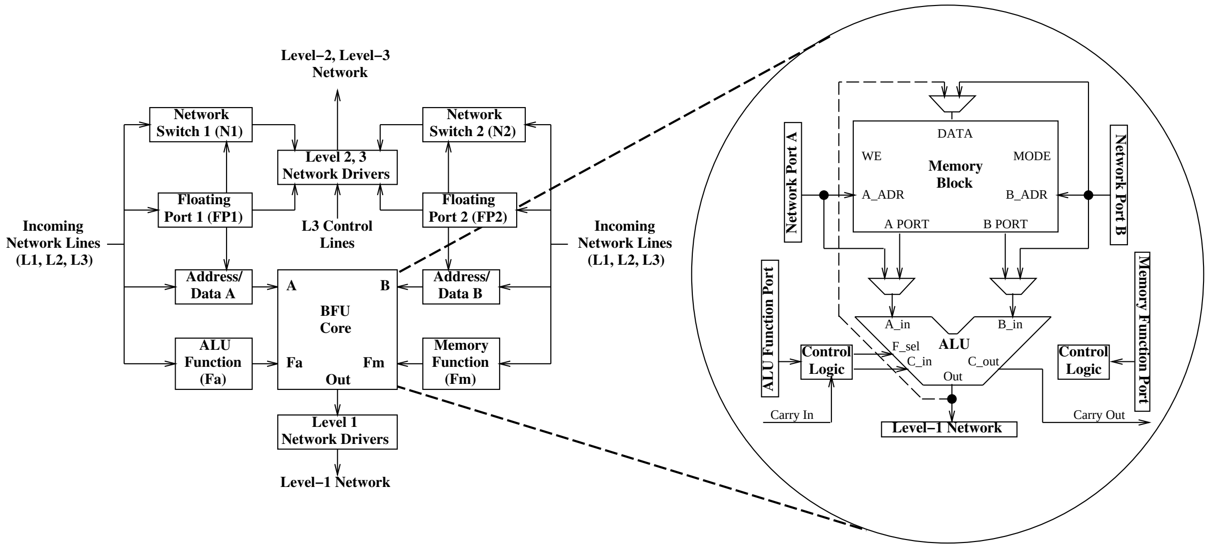


Figure 1: MATRIX BFU

- **ALU function –** Independent ALU function

The BFU's versatility allows each unit to be deployed as part of a computational datapath or as part of the memory or control circuitry in a design.

2.2 Network

The MATRIX network is a hierarchical collection of 8-bit busses. The interconnect distribution resembles traditional FPGA interconnect. Unlike traditional FPGA interconnect, MATRIX has the option to dynamically switch network connections. The network includes:

1. **Nearest Neighbor Connection** (Figure 3 Left) – A direct network connection is provided between the BFUs within two manhattan grid squares. Results transmitted over local interconnect are available for consumption on the following clock cycle.

2. **Length Four Bypass Connection** (Figure 3 Right) – Each BFU supports two connections into the level two network. The level two network allows corner turns, local fanout, medium distance interconnect, and some data shifting and retiming. Travel on the level two network may add as few as one pipeline delay stage between producer and consumer for every three level two switches included in the path. Each level two switch may add a pipeline delay stage if necessary for data retiming.

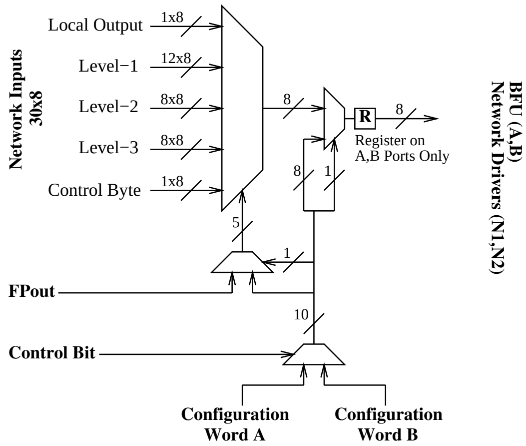


Figure 4: BFU Port Architecture

3. **Global Lines –** Every row and column supports four interconnect lines which span the entire row or column. Travel on a global line adds one pipeline stage between producer and consumer.

2.3 Port Architecture

The MATRIX port configuration is one of the keys to the architecture's flexibility. Figure 4 shows the composition of the BFU network and data ports. Each port can be configured in one of three major modes:

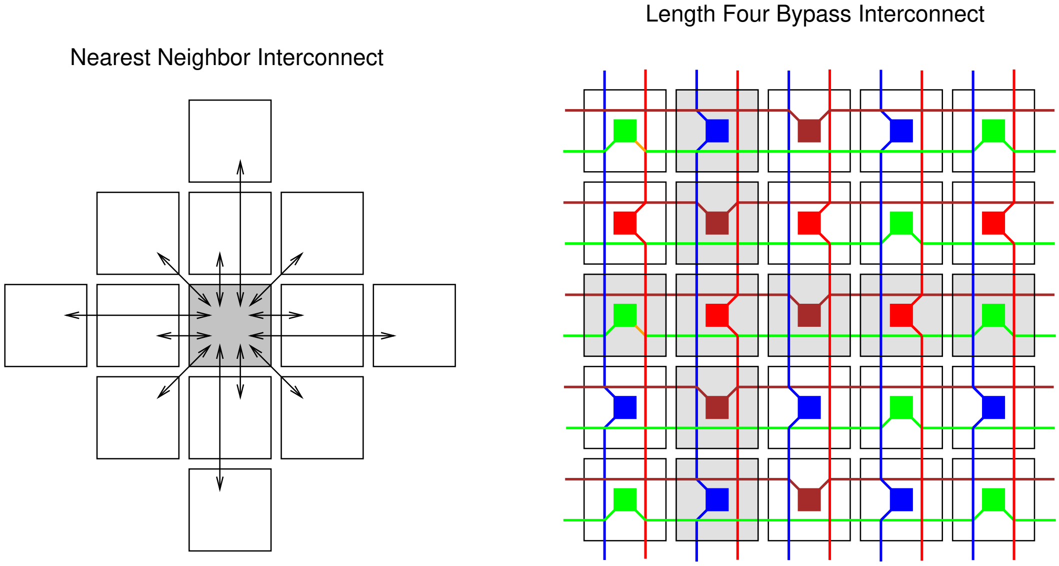


Figure 3: MATRIX Network

1. **Static Value Mode** – The value stored in the port configuration word is used as a static value driven into the port. This is useful for driving constant data or instructions into a BFU. BFUs configured simply as I-Stores or memories will have their ALU function port statically set to flow through. BFUs operating in a systolic array might also have their ALU function port set to the desired operation.

2. **Static Source Mode** – The value stored in the port configuration word is used to statically select the network bus providing data for the appropriate port. This configuration is useful in wiring static control or datapaths. Static port configuration is typical of FPGA interconnect.

3. **Dynamic Source Mode** – The value stored in the port configuration word is ignored. Instead the output of the associated floating port (see Figure 1) controls the input source on a cycle-by-cycle basis. This is useful when datapath sources need to switch during normal operation. For example, during a relaxation algorithm, a BFU might need to alternately take input from each of its neighbors.

The floating port and function ports are configured similarly, but only support the static value and static source modes.

3 Usage

For illustrative purposes, let us consider various convolution implementations on MATRIX. Our convolution task is as follows: Given a set of k weights $\{w_1, w_2, \ldots w_k\}$ and a sequence of samples $\{x_1, x_2, \ldots\}$, compute a sequence of results $\{y_1, y_2, \ldots\}$ according to:

$$y_i = w_1 \cdot x_i + w_2 \cdot x_{i+1} + \cdots + w_k \cdot x_{i+k-1} \qquad (1)$$

Systolic Figure 5 shows an eight-weight ($k = 8$) convolution of 8-bit samples accumulating a 16-bit result value. The top row simply carries sample values through the systolic pipeline. The middle row performs an 8×8 multiply against the constants weights, w's, producing a 16-bit result. The multiply operation is the rate limiter in this task requiring two cycles to produce each 16-bit result. The lower two rows accumulate y_i results. In this case, all datapaths (shown with arrows in the diagram) are wired using static source mode (Figure 4). The constant weights are configured as static value sources to the multiplier cells. Add operations are configured for carry chaining to perform the required 16-bit add operation. For a k-weight filter, this arrangement requires $4k$ cells and produces one result every 2 cycles, completing, on average, $\frac{k}{2}$ 8×8 multiplies and $\frac{k}{2}$ 16-bit adds per cycle.

In practice, we can:

1. Use the horizontal level-two bypass lines for pipelining the inputs, removing the need for the top row of BFUs simply to carry sample values through the pipeline.

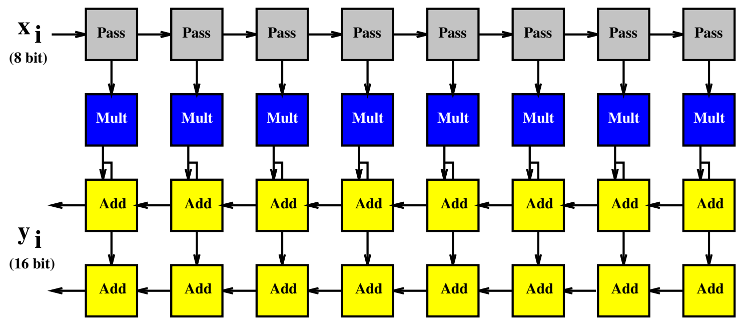


Figure 5: Systolic Convolution Implementation

2. Use both the horizontal and vertical level-two bypass lines to retime the data flowing through the add pipeline so that only a single BFU adder is needed per filter tap stage.

3. Use three I-stores and a program counter (PC) to control the operation of the multiply and add BFUs, as well as the advance of samples along the sample pipeline.

The k-weight filter can be implemented with only $2k + 4$ cells in practice.

Microcoded Figure 6 shows a microcoded convolution implementation. The k coefficient weights are stored in the ALU register-file memory in registers 1 through k and the last k samples are stored in a ring buffer constructed from registers 65 through $64 + k$. Six other memory location (Rs, Rsp, Rw, Rwp, Rl, and Rh) are used to hold values during the computation. The ALU's A and B ports are set to dynamic source mode. I-store memories are used to drive the values controlling the source of the A and B input (two I_{src} memories), the values fed into the A and B inputs (I_a,I_b), the memory function (I_{mf}) and the ALU function (I_{alu}). The PC is a BFU setup to increment its output value or load an address from its associated memory.

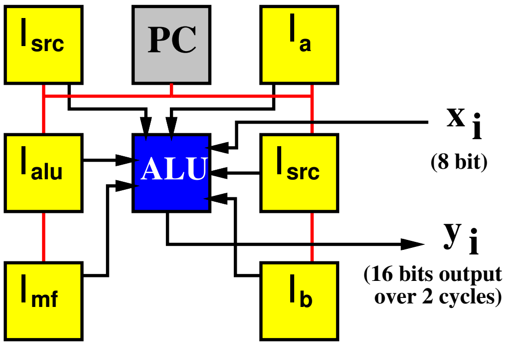
The implementation requires 8 BFUs and produces a new 16-bit result every $8k + 9$ cycles. The result is output over two cycles on the ALU's output bus. The number of weights supported is limited to $k \leq 61$ by the space in the ALU's memory. Longer convolutions (larger k) can be supported by deploying additional memories to hold sample and coefficient values.

Custom VLIW (Horizontal Microcode) Figure 7 shows a VLIW-style implementation of the convolution operation that includes application-specific dataflow. The sample pointer (Xptr) and the coefficient pointer (Wptr) are each given a BFU, and separate ALUs are used for the multiply operation and the summing add operation. This configuration allows the inner loop to consist of only two operations, the two-cycle multiply in parallel with the low and high byte additions. Pointer increments are also performed in parallel. Most of the I-stores used in this design only contain a couple of distinct instructions. With clever use of the control PLA and configuration words, the number of I-stores can be cut in half making this implementation no more costly than the microcoded implementation.

As shown, the implementation requires 11 BFUs and produces a new 16-bit result every $2k + 1$ cycles. As in the microcoded example the result is output over two cycles on the ALU output bus. The number of weights supported is limited to $k \leq 64$ by the space in the ALU's memory.

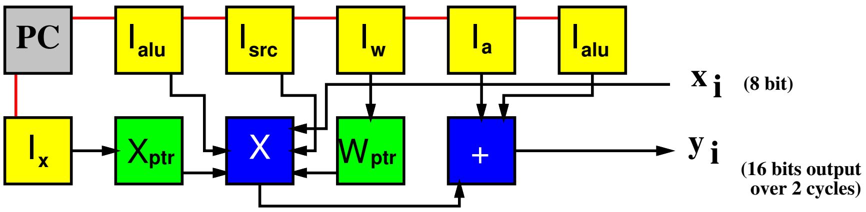
VLIW/MSIMD Figure 8 shows a Multiple-SIMD/VLIW hybrid implementation based on the control structure from the VLIW implementation. As shown in the figure, six separate convolutions are performed simultaneously sharing the same VLIW control developed to perform a single convolution, amortizing the cost of the control overhead. To exploit shared control in this manner, the sample data streams must receive data at the same rate in lock step.

When sample rates differ, separate control may be required for each different rate. This amounts to replicating the VLIW control section for each data stream. In the extreme of one control unit per data stream, we would have a VLIW/MIMD implementation. Between the two extremes, we have VLIW/MSIMD hybrids with varying numbers of control streams according to the application requirements.



Label	ALU Op	PC
newsample	Rxp ← Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont1
	$<$ Rxp $>$ ← new x_i	(pipelined branch slot)
	Rxp ← 65	
xpcont1	$<$ Rxp $>$ ← new x_i	
	Rs ← $<$ Rxp $>$	
	Rwp ← 1	
	Rw ← $<$ Rwp $>$	
	Rs ← Rs × Rw	
	Rw ← ×-continue	
	Rl ← Rs; Match false	BNE enterloop
	Rh ← Rw	(pipelined branch slot)
innerloop	Rs ← Rs × Rw	
	Rw ← ×-continue	
	Rl ← Rs + Rl	
	Rh ← Rw +-continue Rh	
enterloop	Rxp ← Rxp + 1 ; Match $(k+1)$ (6 bits)	BNE xpcont2
	Rs ← $<$ Rxp $>$	(pipelined branch slot)
	Rxp ← 65	
	Rs ← $<$ Rxp $>$	
xpcont2	Rwp ← Rwp + 1 ; Match $(k+1)$ (6 bits)	BNE innerloop
	Rw ← $<$ Rwp $>$	(pipelined branch slot)
last	read Rl ; Match false	BNE newsample
	read Rh	(pipelined branch slot)

Figure 6: Microcoded Convolution Implementation



Label	Xptr unit	Wptr unit	PC	MPY unit	+-unit
firstsample	Xptr←64	Wptr←0			
	output Xptr	output Wptr		$<$ Xptr $>$ ← new x_i	
nextsample	Xptr++ MOD $k \mid 64$	Wptr++		$<$ Xptr $>$ × $<$ Wptr $>$	
	output Xptr	output Wptr		×-continue	Rlow ← MPY-result
	Xptr++ MOD $k \mid 64$	Wptr++		$<$ Xptr $>$ × $<$ Wptr $>$	Rhigh← MPY-result
	output Xptr	output Wptr		×-continue	Rlow ← Rlow + MPY-result
innerloop	Xptr++ MOD $k \mid 64$	Wptr++; Match k	BNE innerloop	$<$ Xptr $>$ × $<$ Wptr $>$	Rhigh← Rhigh + MPY-result
	output Xptr	output Wptr	(pipelined branch slot)	×-continue	Rlow ← Rlow + MPY-result
last	output Xptr	output Wptr		$<$ Xptr $>$ × $<$ Wptr $>$	Rhigh← Rhigh + MPY-result
	Xptr++ MOD $k \mid 64$	Wptr←0; Match false	BNE nextsample	×-continue	$\boxed{\text{Rlow}}$ ← Rlow + MPY-result
	output Xptr	output Wptr	(pipeline branch slot)	$<$ Xptr $>$ ← new x_i	$\boxed{\text{Rhigh}}$ ← Rhigh + MPY-result

Boxed values in last are the pair of y_i output bytes at the end of each convolution.

Figure 7: Custom VLIW Convolution Implementation

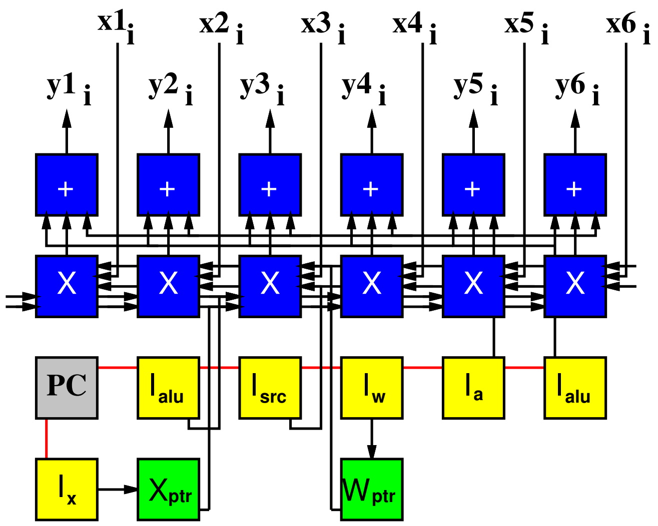


Figure 8: VLIW/MSIMD Convolution Implementation

Comments Of course, many variations on these themes are possible. The power of the MATRIX architecture is its ability to deploy resources for control based on application regularity, throughput requirements, and space available. In contrast, traditional microprocessors, VLIW, or SIMD machines fix the assignment of control resources, memory, and datapath flow at fabrication time, while traditional programmable logic does not support the high-speed reuse of functional units to perform different functions.

4 Relation to Existing Computing Devices

Owing to the coarse-grain configurability, the most closely related architectures are PADDI [4], PADDI-2 [15], and PMEL's vDSP [6]. PADDI has 16-bit functional units with an 8-word deep instruction memory per processing element. A chip-wide instruction pointer is broadcast on a cycle-by-cycle basis giving PADDI a distinctly VLIW control structure. From what little public information is available, the vDSP appears to have a similar VLIW control structure with 4 contexts and 8-bit wide functional units. PADDI-2 also supports 8 distinct instructions per processing element but dispenses with the global instruction pointer, implementing a dataflow-MIMD control structure instead. While the MATRIX BFU is similar in functional composition to these devices, MATRIX is unique in control flexibility, allowing the control structure, be it SIMD, VLIW, MIMD, systolic, or a hybrid structure, to be customized on a per application basis.

Dharma [1], DPGA [14], and VEGA [9] demonstrate various fixed design points with dedicated context memory for reusing the computing and interconnect resources in fine-grained programmable arrays. Dharma has a rigid decomposition of resources into computational phases. The DPGA provides a more flexible multicontext implementation with a small context memory (*e.g.* 4 in the prototype). At the other end of the spectrum, VEGA has 2048 context memory words. The differences in these devices exhibit the tension associated with making a pre-fabrication partitioning and assignment of resources between instruction memory, data memory, and active computing resources. While necessarily granular in nature, MATRIX allows the resource assignment to be made on a per application basis.

The proposed DP-FPGA [5] controls multiple FPGA LUTs or interconnect primitives with a single instruction. However, the assignment of instructions to functional units, and hence the width of the datapath of identically controlled elements is fixed at fabrication time. MATRIX allows a single control memory to control multiple functional units simultaneously in a configurable-SIMD fashion. This provides a form of instruction memory compression not possible when instruction and compute resources have fixed pairings.

As seen in Section 3, MATRIX can be configured to operate in VLIW, SIMD, and MSIMD fashion. Unlike traditional devices, the arrangement of units, dataflow, and control can be customized to the application. In the SIMD cases, MATRIX allows the construction of the master control and reduction networks out of the same pool of resources as array logic, avoiding the need for fixed control logic on each chip or an off-chip array controller. Like MSIMD (*e.g.* [3, 11]) or MIMD multigauge [13] designs, the array can be broken into units operating on different instructions. Synchronization between the separate functions can be lock-step VLIW, like the convolution example, or completely orthogonal depending on the application. Unlike traditional MSIMD or multigauge designs, the control processors and array processors are built out of the same building block resources and networking. Consequently, more array resources are available as less control resources are used.

To handle mixed granularity data efficiently, a number of architectures have been proposed or built which have segmentable datapaths (*e.g.* [13] [2]). These generally exhibit SIMD instruction control for the datapath, but can be reconfigured to treat the n bit datapath as k, $\frac{n}{k}$-bit words, for certain, restricted, values of k. Modern multimedia processors (*e.g.* [12] [8]) allow the datapath to be treated as a collection of 8, 16, 32, or 64 bit words. MATRIX handles mixed or varying granularities by composing BFUs and deploying instruction control. Since the datapath size and assignment of control resources is not fixed for a MATRIX component, MATRIX has greater flexibility to match the datapath composition and granularity to the needs of the application.

The LIFE [10] VLIW architecture was designed to allow easy synthesis of function-specific micro-programmed architectures. The number and type of the functional units can be varied prior to fabrication. The control structure and resources in the architecture remain fixed. MATRIX allows the function-specific composition of micro-programmed functional units, but does not fix control or resource allocation prior to fabrication time.

Reviewing the microcoded example in Section 3, we can see both how microprocessors manage to achieve more functional diversity than FPGAs and how FPGAs and other reconfigurable architectures can achieve higher performance than processors on highly repetitive computing tasks with limited functional diversity. In order to support heavy reuse of a functional unit, a considerable fraction of the resources must go into controlling the functions and datapaths including the memory to hold programs and data, as we see in the microcoded example. This is one of the primary reasons that the performance provided by microprocessors is small compared to their reconfigurable counterparts — most of the device capacity in microprocessors is dedicated to memory and control **not** to active computation required by the task. Furthermore, in cases such as this one, most of the cycles on the active resources are dedicated to control and bookkeeping.

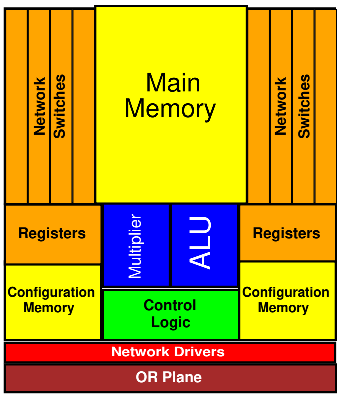
Table 1 presents an instruction stream taxonomy for multiple data computing devices. Owing to their fixed instruction control structure, all traditional computing devices can be categorized in this taxonomy. MATRIX is unique in that its multi-level configuration allows it, post fabrication, to implement any of these structures and many hybrids. Independent of an application configuration, MATRIX defies strict classification based on the number of instructions and threads of control.

5 Granularity

The 8-bit granularity used in MATRIX is a convenient size for use in datapaths, memory addressing, and control. Using the configurable instruction distribution, wide-word operations can be cascaded with reasonable efficiency. The overhead for configuring and controlling datapaths is significantly reduced compared to a bit-level network configuration. Owing to the 8-bit granularity, MATRIX will not completely subsume bit-granularity reconfigurable devices for irregular, fine-grained operations.

Control Threads (PCs)	**Instructions**	**Architecture/Examples**
0	n/a	Hardwired Functional Unit Group (*e.g.* ECC/EDC unit, FP MPY, hardware systolic)
	n	FPGA, Programmable Systolic Array
1	1	SIMD
	n	VLIW, PADDI, DPGA
n	n	MIMD (traditional), PADDI-2

Table 1: Taxonomy for Fixed Instruction Distribution Architectures



Technology	0.5μ CMOS
BFU Size	1.5mm $\times$ 1.2mm
	($1.8\text{mm}^2 \approx 29\text{M}\lambda^2$)
Data Width	8-bit
Memory	256×8
Cycle	10 ns (estimate)

Figure 9: MATRIX BFU Composition

6 Implementation

Figure 9 shows the composition of a BFU along with its size and performance. As described in Section 2, the BFU is pipelined at the BFU level allowing high speed implementation. The 10 ns cycle estimate is for the university prototype. With only a small memory read, an ALU operation, and local network distribution, the basic cycle rate can be quite small – at least comparable to microprocessor clock rates. The area breakdown is roughly: 50% network, 30% memory, 12% control, and 8% ALU including the multiplier. At 1.8mm^2, 100 BFUs fit on a 17mm$\times$14mm die. A 100 BFU MATRIX device operating at 100MHz has a peak performance of 10^{10} 8-bit operations per cycle (10 Gop/s).

7 Conclusions

Traditional computing devices are configured for an application by their instruction stream. However, the composition of their instruction stream and the resources it occupies cannot be tailored to the application. With a multi-level configuration scheme, MATRIX allows the application to control the division of resources between computation and control. In the process, MATRIX allows the application to determine the specifics of the instruction stream. Consequently, MATRIX to provide:

- **Parallel, Configurable Dataflow** – Datapaths can be wired up in an application-specific manner avoiding serialization of data transfer through memory or global busses. Results can often be delivered directly to their consumers avoiding intermediate operations to route data.
- **As much Dynamic Control as Needed** – Values, operations, and switches which need to change on a cycle-by-cycle basis may be controlled by deploying memory or functional blocks for their control. Values and entities which do not need to change during a computation require the deployment of no additional resources for their control.
- **As much Regularity as Exploitable** – A single instruction may control as many or as few distinct functional units as the task merits.
- **Deployable Resources** – Each BFU can serve as data memory, datapath ALU, control logic, or instruction memory. This allows each application to allocate available resources according to its characteristics. Regular operations may dedicate most BFUs to datapath logic, while irregular and spatially limited applications may dedicate most BFUs to control.
- **Instruction Stream Compression** – Application-specific tailoring of datapaths and instruction distribution exploits application structure to reduce the size of the delivered instruction stream. Most notably, when an operation does not change from cycle-to-cycle it does not require broadcast, and when the same operation occurs at multiple computational sites, only a single copy need be broadcast.

The result is a general-purpose, reconfigurable computing architecture which robustly yields high-performance across a wide range of computational tasks.

Acknowledgments:

This research is supported by the Advanced Research Projects Agency of the Department of Defense under Rome Labs contract number F30602-94-C-0252.

References

[1] Narasimha B. Bhat, Kamal Chaudhary, and Ernest S. Kuh. Performance-Oriented Fully Routable Dynamic Architecture for a Field Programmable Logic Device. UCB/ERL M93/42, University of California, Berkeley, June 1993.

[2] Michael Bolotski, Thomas Simon, Carlin Vieri, Rajeevan Amirtharajah, and Thomas F. Knight Jr. Abacus: A 1024 Processor 8ns SIMD Array. In *Advanced Research in VLSI 1995*, 1995.

[3] Timothy Bridges. The GPA Machine: A Generally Partitionable MSIMD Architecture. In *Proceedings of the Third Symposium on The Frontiers for Massively Parallel Computations*, pages 196–202. IEEE, 1990.

[4] Dev C. Chen and Jan M. Rabaey. A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Data Paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[5] Don Cherepacha and David Lewis. A Datapath Oriented Architecture for FPGAs. In *Second International ACM/SIGDA Workshop on Field-Programmable Gate Arrays*. ACM, February 1994. proceedings not available outside of the workshop, contact author `lewis@eecg.toronto.edu`.

[6] Peter Clarke. Pilkington Preps Reconfigurable Video DSP. *Electronic Engineering Times*, page 16, August 7 1995. Online briefing `http://www.pmel.com/dsp.html`.

[7] André DeHon. DPGA Utilization and Application. In *Proceedings of the 1996 International Symposium on Field Programmable Gate Arrays*. ACM/SIGDA, February 1996. Extended version available as Transit Note #129, available

via anonymous FTP `transit.ai.mit.edu: transit-notes/tn129.ps.Z`.

[8] Dave Epstein. Chromatic Raises the Multimedia Bar. *Microprocessor Report*, 9(14):23 ff., October 23 1995.

[9] David Jones and David Lewis. A Time-Multiplexed FPGA Architecture for Logic Emulation. In *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference*, pages 495–498. IEEE, May 1995.

[10] Junien Labrousse and Gerrit A. Slavenburg. CREATE-LIFE: A Modular Design Approach for High Performance ASIC's. In *Compcon '90: Thirty-fifth IEEE Computer Society International Conference, Digest of Papers*, pages 427–433. IEEE, February 1990.

[11] Gary J. Nutt. Microprocessor Implementation of a Parallel Processor. In *Proceedings of the Fourth Annual International Symposium on Computer Architecture*, pages 147–152. ACM, 1977.

[12] Michael Slater. MicroUnity Lifts Veil on MediaProcessor. *Microprocessor Report*, 9(14):11 ff., October 23 1995.

[13] Lawrence Snyder. An Inquiry into the Benefits of Multigauge Parallel Computation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 488–492. IEEE, August 1985.

[14] Edward Tau, Ian Eslick, Derrick Chen, Jeremy Brown, and André DeHon. A First Generation DPGA Implementation. In *Proceedings of the Third Canadian Workshop on Field-Programmable Devices*, pages 138–143, May 1995.

[15] Alfred K. Yeung and Jan M. Rabaey. A 2.4 GOPS Data-Drivern Reconfigurable Multiprocessor IC for DSP. In *Proceedings of the 1995 IEEE International Solid-State Circuits Conference,* pages 108–109. IEEE, February 1995.