

Digital Signal Processing at 1GHz in a Field-Programmable Object Array

D. Helgemo

MathStar, Inc., 5900 Green Oak Dr., Minneapolis, MN 55343

Abstract

Autonomous MAC and ALU processors and register files (three types of Silicon Objects) are implemented with custom logic to achieve 1GHz operation. Processor, I/O, and memory objects are arrayed for fabrication, then configured at runtime, yielding a field-programmable object array (FPOA). Consider the applicability of the programmable array for numeric processing such as the Fast Fourier Transform (FFT): Synchronous programmable interconnect and embedded storage reduce the need for difficult index calculation and the use of external memory for intermediate values. The flexibility of the objects and their interconnect allows the level of parallelism to be chosen freely based on performance requirements and resource constraints. Arraying hundreds of objects in parallel in a single chip enables incredible DSP performance from a flexible, in-circuit reprogrammable architecture. For example, a 1024-point radix-2 FFT with (16+16)-bit complex samples can be completed every 160 clock cycles (i.e., every 160 nanoseconds) using 64 butterflies (128 MAC, 128 ALU, and 64 RF objects) assisted by 128 ALU and 64 RF objects for inter-stage data routing.

I. INTRODUCTION

The Field-Programmable Object Array (FPOA) offers a massively parallel high-performance computation fabric. Individual processing units, called Silicon Objects, are programmed individually and act autonomously. The interconnect is programmed to construct custom computation macro blocks – composing simple scalar operations (addition, multiplication, logic) into complex functions (e.g., 1024-point FFT). Interconnect and instructions are programmed at runtime by loading configuration bits from a PROM or via JTAG.

II. SILICON OBJECT COMMUNICATION

Silicon Objects communicate via 21-bit buses composed of the following: sixteen bits of data; one bit indicating the validity of the data (e.g., for event-driven programming); and four bits of user-defined side-band control signals.

Communication proceeds synchronously and cooperatively. Buses are driven by registers; values of interest are read by a cooperating receiving object. Thus data is pulled rather than pushed through the architecture. All registers of all objects are synchronized to the global 1GHz clock; objects identify particular clock cycles via user programming of control signals and/or recognition of data patterns.

Objects are interconnected two ways: as adjacent neighbors and via party lines.

A. Neighbors = Fastest

Each Silicon Object can read the neighbor bus output from each of its *eight* adjacent neighbors (Manhattan directions and diagonals). However, to match silicon resources to expected applications, a Silicon Object transmits *four* registers toward its neighbors, one for each pair of neighbors. Thus, when a Silicon Object updates a register that drives a neighbor bus, two corresponding neighbor objects perceive the new value immediately – i.e., as fast as their own internal registers.

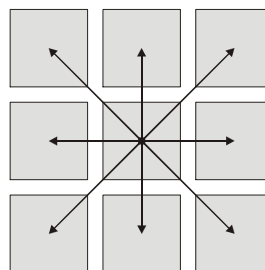


Figure 1: Neighbor Communication – Same Speed as Local Registers

Neighbor communication offers zero-latency interconnect under the constraint of adjacency. Any size computational neighborhood can be constructed, one hop at a time. In practice, most complex functions can be arranged such that the majority of inter-object communication can be supported via neighbor interconnect. However, to facilitate access to distant resources and/or to interconnect multiple dense functions, an escape from this dense two-dimensional universe (a crowded flat land) is provided.

B. Party Lines = Farthest

Party lines reach beyond the eight-object neighborhood: Objects launch register values onto a party line bus so that distant objects can land the value. To maintain deterministic digital synchronization, bus values are retimed as needed via pipeline register stages in intermediate objects, and finally are landed into an object register to be read. At maximum clock frequency, party lines must be retimed after passing to a third object. Thus party lines have unlimited range at the expense of deterministic (pipelined) latency.

Party lines are partitioned into layers. Each layer transmits a bus value in each Manhattan direction (north, south, east, west). There are two full layers and one half-layer. (The half-layer communicates only in the north-south direction.) Thus there are a total of ten party lines (4+4+2). Each party line can be configured to pass along bus values in the direction received, to turn ninety degrees in either direction, to land the received value into a register for retiming or for object input, or to transmit the value of a local object register.

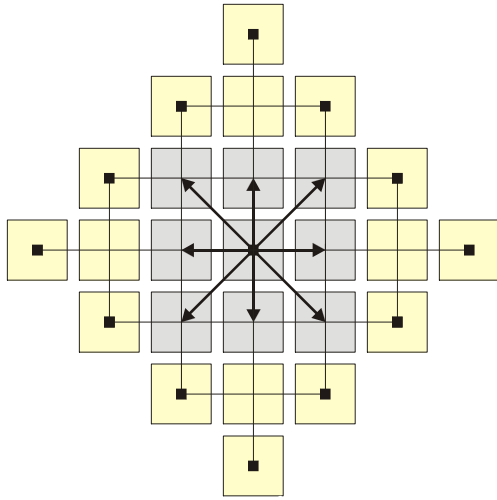


Figure 2: Party Line Communication – Neighborhood within One Extra Clock Cycle

The configuration flexibility of party lines gives rise to derivative capabilities. Because landing and launching are independently controlled, party lines can be used to broadcast values to multiple objects. (If there is no intervening retiming register, then all of the objects will perceive the multicast value in the same clock cycle.) With per-object configuration granularity, complex data movement can be configured (e.g., a chess “knight’s move” can be configured with minimum latency). Alternatively, extra retiming registers can be inserted into a path to evoke intentional additional pipeline delay (for example, to stall data that otherwise would have arrived earlier than a related control signal).

C. Communication Summary

Overall, the communication infrastructure is smooth and scalable, to any shape or size of computation resources. (In contrast, hierarchical communication schemes impose bandwidth and/or latency penalties whenever resources exceed the shape and/or number of a given hierarchical level.) Additionally (and particularly relevant to DSP kernels), the programming and pipelining of the interconnect facilitates the design of arbitrary spatial and timing communication patterns to keep computation resources fully utilized.

III. SILICON OBJECT TYPES

With an understanding of the interconnect between Silicon Objects instances, the internal capabilities of each type of Silicon Object can be discussed.

A. Multiply-Accumulate Object (MAC)

The MAC object type multiplies two 16-bit integers and adds the 32-bit result into the accumulator. The accumulator can be configured either to saturate or to wrap into an 8-bit over-/underflow counter, thereby accommodating up to 40-bit magnitude intermediate results. The operation can be configured for integer or fractional (Q15) accumulation. The

MAC object consumes fresh inputs and generates a result every clock cycle, with a processing latency of two clock cycles. That is, a sum of 100 products requires 101 clock cycles (101 nanoseconds).

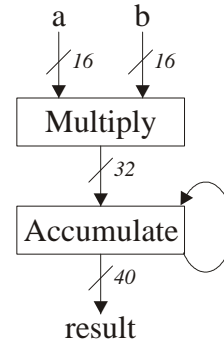


Figure 3: Multiply-Accumulate (MAC) Object

The MAC interacts with control signals for algorithmic integration. Configurable control inputs choose the operand input sources, choose signed/unsigned treatment for each operand, reset the accumulator, negate the current product (useful for complex multiply), enable the operator, update an output register, and round to a 16-bit value. All forty bits of the accumulator are available for output, and 16-bit values can be sampled from several different bit-shifted positions. Using a third operand, the accumulator can accept an arbitrary bias value or resume a suspended sum of products with no loss of precision.

B. Arithmetic-Logic Unit Object (ALU)

The ALU is the most general-purpose object type. It employs a 16-bit add, shift, and logic operator controlled by an 8-instruction state machine. Each instruction selects three input words and one carry input bit, configures the operator (a.k.a. opcode), selects result destination register(s), and specifies conditional execution and branching options. Conditional execution and branching is guided by up to four control inputs that are conditioned by a truth table (an arbitrary function 4:4 lookup table).

This object type contains nine working registers (four for neighbors, five for party lines); two programmable constant registers; and two wired constants. In total there are twenty-one possible inputs and nine possible outputs. In a single clock cycle, the current instruction is fetched and decoded, the operator is executed, the result is stored (subject to conditional execution), and the next instruction is selected per branching.

Four of the working registers are placed in the corners of the ALU object square; each one’s location determines its neighbor connectivity: The northwest register can be read by neighbors to the north and northwest, and can be launched on north- and westbound party lines. The pattern is repeated for the other three corners. Focusing on just neighbor communication, ALU objects thus output four values and input eight values.

The remaining five working registers are coupled to the party lines. Specifically, each register can land and launch a pair of party lines – same layer, opposing directions. Thus these registers are named NS1, EW1, NS2, EW2, and NS3 (where the digit indicates the party line layer number). Each register is configured to receive either party line data (for landing or retiming) or ALU result data. Each register can be selected for launch onto its corresponding party lines.

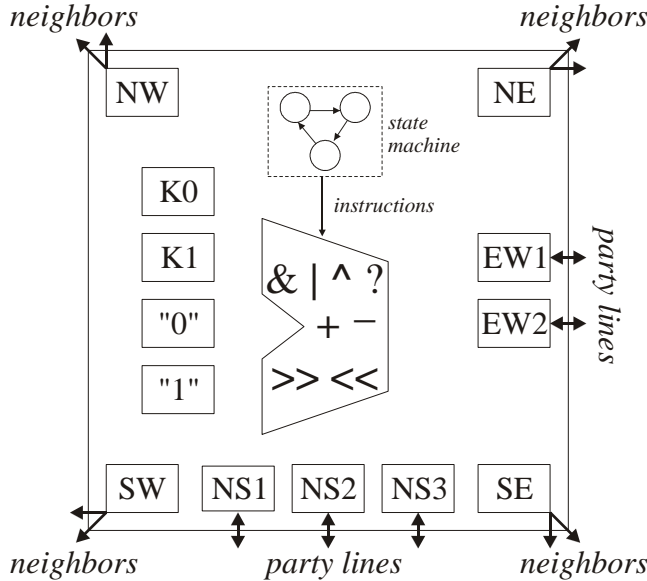


Figure 4: Arithmetic Logic Unit (ALU) Object

C. Truth Function Object (TF)

Four 4:1 lookup tables each consume up to four control inputs and then drive an output control bit. Each 16-entry lookup table allows any 4-input arbitrary logic function to be implemented. The TF object is integrated with the ALU object type to provide control bit generation.

D. Register File Object (RF)

The RF object type provides fast storage within the array. Up to two 20-bit values can be read and two 20-bit values written simultaneously every clock cycle, with a latency of two clock cycles. Storage capacity is sixty-four 20-bit words. If 40-bit units are required, the contents can be internally remapped as thirty-two double-words. An RF object is configured in one of three modes: random-access read/write, first in first out (FIFO), or a hybrid of the two: random-write sequential-read (particularly useful for FFT).

Random access read/write mode allows simultaneous read and write to different addresses.

FIFO mode provides a functional shock absorber. Programmable watermarks indicate fullness via control bit outputs.

Random-write sequential-read allows values to be sorted: write in an arbitrary index order, then read as a sequence without the burden of address generation. That is, values are

written to arbitrary addresses in anticipation of the order in which they will be read out.

IV. ALGORITHMS VIA OBJECTS

To demonstrate the utility and flexibility of programmable Silicon Objects, several practical algorithms are implemented. Note especially how the designer can choose smoothly between maximum performance with maximum resource consumption and lower performance with resource conservation.

A. Sum of Products: Dot Product, Matrix Multiplication, Convolution, Finite Impulse Response (FIR) Filter

A single MAC object is designed to support any finite length sum of products: $a_0b_0+a_1b_1+\dots+a_{n-1}b_{n-1}$. However, calculating the sum of n products with only a single MAC object can be initiated only every n clock cycles. If higher performance is desired, multiple MAC objects can execute subsequences in parallel, then use ALU objects in a binary tree (which halves the number of results each stage) to calculate the final aggregate.

Table 1: Sum of Products Parallelism
(Rate is results per clock cycle, Latency is clock cycles)

	MACs	ALUs	Rate	Latency
Slowest/Smallest	1	0	1/n	n+2
Intermediate	sqrt(n)	sqrt(n)-1	1/sqrt(n)	2+1/2log ₂ n
Fastest/Largest	n	n-1	1	2+log ₂ n

Surrounding ALU objects then provide the data iteration patterns needed both to scan the data and coefficients for a single sum of products and to sequence outer loops (e.g., scanning rows and columns for matrix multiplications). RF objects can provide addressable storage for input data, coefficients, intermediate results (if any), and output results.

B. Complex Multiplication

MAC objects can be ganged to multiply complex numbers: $(a+bj)(c+dj) = (ac-bd)+(ad+bc)j$. Thus four multiplications are performed, two of which are differenced, two of which are summed. Thus two MAC objects can generate a complex result every two clock cycles, with a latency of three clock cycles.

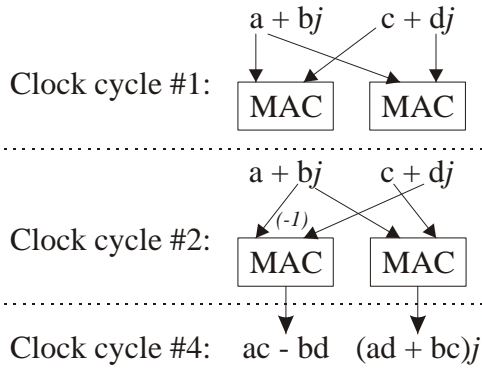


Figure 5: Efficient Complex Multiplication

Bandwidth can be doubled, but at a loss of efficiency: Four MAC objects perform the four multiplications in parallel: Intra-MAC accumulation is no longer possible, so two ALUs must be appended to difference and sum the pairs of multiplications. In this fashion, four MAC objects and two ALU objects generate a complex result every clock cycle, with a latency of three clock cycles.

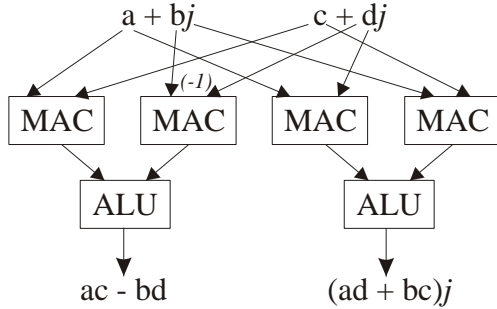


Figure 6: Fastest Complex Multiplication

C. Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is a computationally efficient means of calculating the Discrete Fourier Transform (DFT) of a discrete sequence. The radix-2 FFT analyzes successive pairs of the input sequence via an FFT “butterfly” operation, yielding pairs of complex numbers. (Thus the output sequence is the same size as the input sequence.) This completes the first stage of the FFT. Progressive stages choose different pairings of the previous stage’s results (as well as different butterfly parameters) until all of the FFT inputs affect all of the FFT outputs (i.e., 2^n points require n stages).

Each butterfly of each stage employs a complex constant number called a “phase factor” or “twiddle factor.” Rather than calculating phase factors on the fly (a complex sine), these numbers are calculated at design time and preloaded into a Register File (RF) object. Each RF object can store up to thirty-two complex constants. (If additional constants are required, RF objects can be ganged together via party lines and ALU objects.) Due to the predictable order of using these constants (because of the predictable order of the butterflies and the stages containing them), the RF object is configured into sequential read mode – and address generation is not

required. Also, because the timing of the sequence is perfectly predictable, the RF object supplying constants can be distant and communicate via high latency paths (e.g., party lines with many hops).

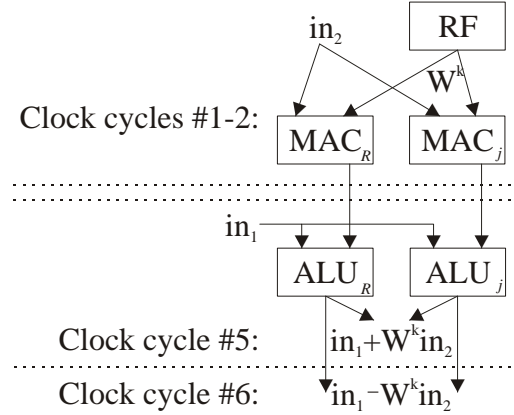


Figure 7: Decimation-In-Time Butterfly

Consider the implementation of a decimation-in-time butterfly: $out_1 = in_1 + W^k in_2$, $out_2 = in_1 - W^k in_2$, where W^k is the complex phase factor (for a particular butterfly of a particular stage). The following sequence generates the two complex butterfly outputs: 1. Fetch the precalculated W^k from the RF object. 2. Perform the complex multiplication $W^k in_2$ via two MAC objects (as described above). 3. Use two ALU objects to sum and difference with in_1 to generate complex out_1 and out_2 , respectively. Thus, a butterfly is completed every two clock cycles, with a latency of six clock cycles.

While a single butterfly can be leveraged to any size FFT, multiple parallel instantiations of the butterfly (in powers of two) increase the theoretical computational performance dramatically.

Table 2: Butterfly Parallelism for 2^n -point FFT (n stages)
(Rate is results per clock cycle, Latency is clock cycles)

# Butterflies	MACs	ALUs	RFs	Rate	Latency
1	2	2	1	$1/(n2^{n+1})$	$4+n2^n$
2	4	4	2	$1/(n2^n)$	$4+n2^{n-1}$
2^{n-2}	2^{n-1}	2^{n-1}	2^{n-2}	$1/4n$	$4+4n$
2^{n-1}	2^n	2^n	2^{n-1}	$1/2n$	$4+2n$
$n2^{n-1}$	$n2^n$	$n2^n$	$n2^{n-1}$	$1/2$	$4+2n$

Fortunately, practical performance does not substantially lag the theoretical ceiling. Butterflies are kept 100% utilized within an FFT stage by providing two new complex inputs every two clock cycles. Either an RF object or two ALU objects can sustain this bandwidth indefinitely – and at any distance via pipelined party lines. The trick lies in efficient transitions between FFT stages.

Every butterfly result is used precisely twice as an input into the next stage. Therefore, the butterfly results (one complex result per clock cycle) are routed via ALU objects (with stage-specific directions) toward the two butterflies for the next FFT stage. An RF object sorts the complex data values into the correct order for the next stage using a nearby

ALU object to generate stage-specific write addresses into the RF object.

Performance is lost between stages only if the RF object cannot be loaded in time to start the next stage. In practice, index analysis of the data dependencies between stages allows the next stage to be started while the previous stage completes. (Ironically, fully parallelized butterflies cannot avoid stalling between FFT stages: Because all of the butterflies are calculated in parallel, none of them can proceed until the previous stage completes.)

By employing 64 butterflies (128 MAC, 128 ALU, and 64 RF objects), a 1024-point FFT with (16+16)-bit complex samples can be completed every 160 clock cycles (i.e., every 160 nanoseconds), assisted by 128 ALU and 64 RF objects for inter-stage data routing.

V. RESULTS

The object approach to silicon provides high performance

and high density. An array of 400 objects (20x20) fits within 100 square millimeters (10x10), providing up to 400 billion operations per second, consuming 20 watts of power. The resulting chip is field-programmable (configuration is loaded at run-time into latches) with optional AES encryption and bond-selectable copy protection.

Field-programmable object arrays (FPOAs) compare favorably to both FPGA products and ASIC methodology. Standard FPOAs can be purchased commercially off the shelf (COTS) at a fraction of the price of an FPGA. Yet FPOA performance, density, and performance per power is dramatically improved (FPOA is more coarse-grained than bit-programmable FPGAs, so electrical signals are subject to less loading between functions, yielding higher speed with less power per function). FPOA can also be used as an ASIC methodology: Choose custom mixes of proven object types (processors, memory, and I/O) to yield a low-risk custom ASIC (system-on-chip via objects) which enjoys the further risk reduction of field programmability.