

# Exploring Architectures and Methods for Supporting Wide-bit Arithmetic in Reconfigurable Computing Applications

J.P. Davis, S. Devarkal, N. Sontineni, and A. Kathuria  
*Reconfigurable Computing Laboratory*  
*Department of Computer Science and Engineering*  
*University of South Carolina*  
*Columbia, SC 29208 USA*  
*jimdavis@cse.sc.edu*

## Abstract

*In this paper, we explore some of the architectural issues associated with creating high-performance wide-bit data path arithmetic applications on a reconfigurable computing platform consisting of multiple FPGAs. Many applications of reconfigurable computing for arithmetic, such as for DSP and Cryptography, have been carried out with small bit-widths, thereby avoiding some of the more difficult design tradeoffs associated with architectures that do not fit onto a single FPGA device. In this paper, we explore some of the results in the literature as pertaining to architecture tradeoffs and HDL-based design methods, and apply them to the problem of architecting efficient arithmetic circuits spanning multiple FPGAs in an RCM fabric. We focus on the aspects of realizing Addition and Multiplication operations supporting data paths of 192 and 256-bits in width. We discuss our methods in context of the StarBridge® HAL-15® and HC-36m® platforms for sake of reference.*

## 1. Introduction

Reconfigurable computing machines (RCM) have become recognized as a viable means for achieving orders of magnitude speedup, through the use of fine-grained parallelism on a customized logic substrate, on compute-intensive applications [1, 2]. Algorithms in a wide variety of military and scientific domains and the architectures necessary for their implementation are being implemented, through a series of transformations, on a custom-logic "fabric" consisting of one or more reconfigurable logic devices, such as commercially-available field programmable gate arrays (FPGAs) [3].

The process by which a set of algorithms associated with an application become realized as an architecture bound to an RCM consists of a set of ordered process steps, some of which can partly be automated and some of which are usually done manually. The typical process involves hardware designers and software programmers to create the elements of an application partitioned

between a host-based CPU and an RCM array fabric—the type of RCM-based platform of interest to our research.

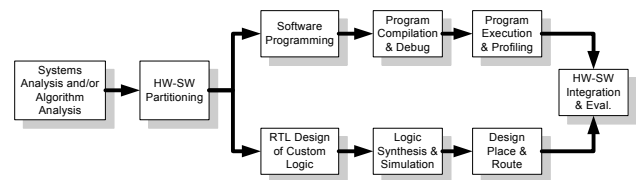


Figure 1. RCM application co-development process

Today, users of an RCM platform can analyze the target application and make the high-level hardware-software partitioning decisions, as well as the lower-level RCM resource mapping, scheduling and allocation decisions, so as to maximize the utilization of available resources and the computational throughput on the RCM fabric. Maximizing throughput also involves minimizing the inherent overhead of the communications channel between the host CPU and the RCM fabric [3].

The application development process is thus divided into two separate but related streams: (1) developing the logic design of that portion of the application to run on the reconfigurable fabric; and, (2) developing the software application program portion that will execute on the host CPU and coordinate that execution with that portion executing as reconfigured hardware logic on the RCM fabric.

The detailed separation of an application's implementation into parts that execute on the host CPU and parts that execute on the RCM fabric is determined by which parts of the application can effectively be made to operate in parallel for execution on the RCM fabric, in a manner that also minimizes the communication overhead of getting data onto and off of the fabric. This fabric is generally suited to fine-grained parallelism; highly repetitive operations, such as pipelined arithmetic calculations on wide data words, and highly parallel operations--such as those found in image processing, phylogenetics and cryptography domains--are a good match for an RCM as long as the data pipe can be filled and results returned to the host application in a timely manner. Invariably some parts of the application

processing are maintained in that portion of the application's code executing on the host CPU.

For scientists and researchers looking for a means to speed up the execution of critical parts of their applications, this process of creating applications for an RCM platform remains something of a black art. As such, the current state of the art in harnessing the power of RCM platforms for productive use in scientific research is only just beginning. Successful applications for the hard sciences have been done [1, 2, 3], but the continuing difficulty associated with taking applications and moving them seamlessly onto an RCM platform combining conventional CPUs and the RCM fabric itself is the major obstacle to their wider use.

In this paper, examine one aspect of this process--the logical transformation from application, to algorithm, to architecture and, finally, to device mapping. The critical problems for the analyst/architect--when considering effective and efficient realization of those portions of the application onto the RCM fabric--primarily focus on how best to map between algorithm, architecture and device realization, once characteristics of the application domain are well understood.

As such, the architect must be concerned with problems associated with partitioning and decomposition of the RCM-bound application parts. This includes identifying how best to maximize concurrency to speed throughput while also conserving resources of the fabric, thus striking the best tradeoff between computation, resource usage and communications overhead. This analysis must be done both between processing elements (PE) of the fabric and between the RCM and the host system. In this paper, we explore some of these issues pertaining to analysis and architectural tradeoffs on the fabric, and formulate a methodology devised for dealing with applications employing wide-bit arithmetic operations. We illustrate these methodological issues with example wide-bit multiplier circuits designed for cryptographic applications.

## 2. Applications and algorithms

It has been understood for some time in the design of VLSI systems that an application's properties—namely algorithms and execution environment (e.g., throughput requirements)—drive the identification and selection of architecture [15]. However, when attempting to harness an existing environment to new applications, such as in the nascent area of reconfigurable computing, we may be presented with a set of fixed constraints and be expected to configure an architecture that meets the application processing requirements in light of these constraints, if one exists. As such, we generally must meet throughput requirements while fitting within the device's available resources.

Consequently, we need a means to characterize the application and its algorithms, mapping them to a set of basic, configurable architectural “patterns”, such that we can quickly assess the ability of the platform to support the application in question, and also to provide guidance in the definition of a logical architecture that can be mapped onto the available physical resources in a manner that satisfies the application's constraints. This is very much a “middle out” style of design strategy, in that we conceptualize the application and its key algorithms top-down, but must consider how best to assemble an appropriate architecture from available or new components, such that device constraints are not violated.

In our consideration of applications, we have been exploring wide-bit arithmetic circuits for use in cryptography. Many of the major computational problems in modern data communications involve the issue of how to provide information over a channel securely. For a number of years, the National Institute of Standards and Technology (NIST) has promulgated standards for cryptographic protocols, some of which involve public key algorithms [6]. At the heart of public key computations such as these are basic arithmetic operations, either for multi-precise integers or for elements in finite fields of characteristic GF(2).

The primary operation considered in the scope of this paper is the multiplication operation, of which there are 14 required for adding points on an elliptic curve. We need schemes for the wide multiplication operations at 192, 256, 384 and 521 bits. Also, due to the nature of most of the multiplication algorithms, there is considerable nesting of operations at different levels of the architecture. Some of the multipliers to be considered are themselves composed of smaller-width multipliers, of which the models at all levels comprise other arithmetic operations, such as addition and subtraction.

The process of building an architecture laid onto the RCM fabric involves building models for the various units at these different widths and levels of composition to understand their behavior, and to identify one or more composite architectures that meet the throughput and area constraints imposed by the application and by the RCM fabric and its constituent devices.

## 3. Architecture analysis method

Our objective is to create VLSI circuit architectures that can be efficiently implemented on reconfigurable computing devices, such that we can realize high-performance wide-bit arithmetic processing for a range of scientific applications. As an example, given the processing characteristics of many cryptography algorithms, we have focused our attention on understanding the requirements for multiplier circuits realizable on an RCM platform. To that end, there are

many multiplier algorithms and architectures to choose from—so, which are appropriate? Also, given the run times for creating such models, is there a means whereby we can estimate the costs of different architectural choices, so as to better focus our design compiles on those that are most promising?

We seek to articulate a set of heuristics that guide us to appropriately use different multiplier architectures in varying applications to crypto-arithmetic processing, given a range of RCM fabrics with differing characteristics—such as number of devices and device resource characteristics. To that end, our focus of study is outlined as follows:

1. Identify and model different Multiplier circuits – here, we are interested to identify and select appropriate schemes that can be used on wide-bit unsigned integer arithmetic that is appropriate for multi-precise and elliptic curve arithmetic.
2. Select an appropriate unit size – we are interested to select a basic unit for constructing large-width multiplier data paths. Since it is impractical to construct monolithic multiplier circuits of very large widths [7] [15], we must model our different multiplier schemes as basic units, from which we will construct larger units.
3. Having modeled the basic multiplier units of some standard width, we then subject these units to benchmarking. As pointed out in Chen and Mead [4], what might be considered an efficient or optimal algorithm in a mathematical or software sense may not be so from the standpoint of its architecture because of physical cost considerations. Therefore, we must subject each model to cost evaluation. For synthesized circuits, this is generally done in terms of area, speed (both in terms of maximum delay, and best clocking speed) and power characteristics. Since we are considering our implementation on a reconfigurable parallel computing platform, based on an array of Xilinx® FPGAs as processing units not destined for low-power mobile or wireless applications, we will for the moment ignore the power issue and focus on area and speed. Area is most important, as we need to be able to assess the ability of a multiplier unit to scale to size, while also continuing to “fit” on the given PE device. Otherwise, if the design doesn’t fit, we incur performance penalties by having to go “off chip” to complete the arithmetic function. Speed is important for obvious reasons, in that minimizing computation time (in the form of worst-case delay through the unit) affects the speed at which we can clock the design reliably, and thus minimize latency, and

maximize throughput, for the overall time for computation.

4. Having characterized each of the identified and modeled multiplier units, we then devise scaled models of the computation architecture for the data path widths that are of interest. We are dealing with unsigned integer arithmetic of the following data path widths: 192, 256, 384 and 521 bits in width. (Note that the 521-bit computation is generally rounded up to 544 bits, as this allows use of a standard unit configuration for this purpose.) We assume at present that the architectures for each of the bit widths may dictate a different—or even more than one—multiplier architecture for use in a given reconfigurable application. Therefore, to appropriately account for this possibility, we are analyzing the larger multiplier architectures using different basic multiplier units, and evaluating the performance of each wide-bit configuration.

It is impractical to construct wide-bit multipliers of 192, 256 or 384-bits as monolithic circuits. As discussed in [4] and [7], a hierarchical design approach allows better management of design complexity by exploiting the regularity of large numbers of identical structures, and also affords a better chance that a usable and efficient circuit can be synthesized. Therefore, we explore a number of different multiplier architectures, identifying a set of basic 32-bit units, using these basic units to construct the larger multiplier circuits with wide data path widths. The selection of 32-bit operations over 64-bit or larger was done for intuitive reasons that were borne out in the resultant data collected. As will be shown in the next section, the structure of the wide-bit multiplier architecture dictated 32-bits as a logical starting point.

#### 4. Architectural treatment

In the figure below, we depict one of two wide-bit architecture schemes we have been exploring as the basis for wide-bit computation. In this “broadcast” scheme, we break up a 192-bit multiplication into six separate 32-bit multiplication operations, scheduled in parallel, whose outputs are fed into 64-bit pipeline registers, according to a “perfect shuffle” permutation scheme [15].

The partial product registers are grouped into 3-units, where each group is treated as a unit for purposes of shifting partial product data. The leftmost 3-unit, labeled as Registers Rb, Rd, and Rf, form a 192-bit shift register, on which a 32-bit SHL operation occurs after each multiplication cycle.

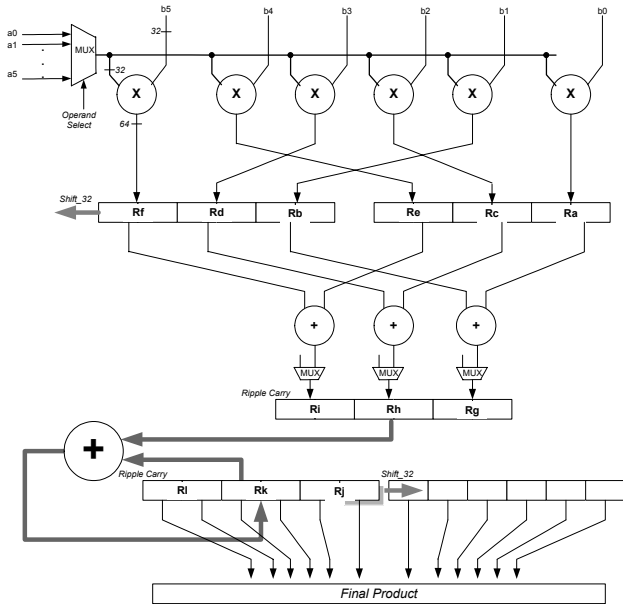


Figure 2. Broadcast-style 192-bit multiplier datapath

The data stored in these registers is then clocked into three 64-bit adder units, organized into an “exchange permutation” scheme [15] whose outputs are fed into a second stage of pipelined registers. The following stage adds the results of these three registers, grouped as a 3-unit, with an “accumulator” register 3-unit consisting of three additional 64-bit pipelined registers. During this final addition, the contents of the permuted partial product addition registers are added to the running value stored in the accumulator. Between each accumulator add, the contents of the 3-unit accumulator are right-shifted 32-bits into an additional 3-unit register group. After six such accumulator-adds and 32-bit shifts, a complete final product is contained within the running 384-bits of 64-bit registers organized into two 3-unit registers. A final product register, consisting of three 128-bit registers, stores the product for latching by another stage of the application.

Based on preliminary assessment reported elsewhere [5], we have settled on evaluating each of the following 32-bit multiplier units as base components to be used in the larger multiplier circuit. We’ve started with a basic set of multiplier architectures for the Shift-Add [9], the Pencil and Paper method [10], the Booth algorithm [11], and the Montgomery approach [12] and the Divide and Conquer scheme [8]. Whereas other algorithms exist, each of these techniques has been used for unsigned integer arithmetic, and suits our needs as initial units for study of the problem of scaling to wide data paths, and realizing these circuits in an RCM fabric consisting of multiple FPGAs.

#### 4. Realizing logical architecture on the fabric

There are two different hybrid RCM platforms on which we have been exploring the architecture and design of parallel high-performance solutions to scientific applications: the HAL-15® and HC-36m®, both from Star Bridge Systems. The figure below depicts the style of topology that exists as the RCM fabric of this class of machine (courtesy of Star Bridge Systems) [13].

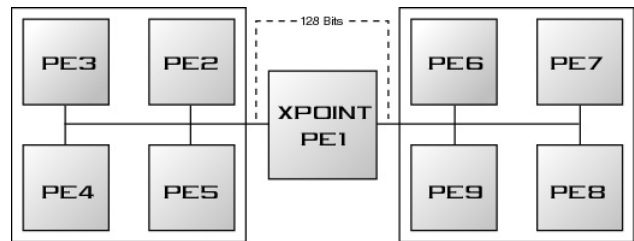


Figure 3. Device topology of Star Bridge RCM fabric

Namely, the hybrid system consists of a host system based on the “Wintel” platform, loosely coupled with an RCM fabric consisting of a number of Xilinx® XC4062 (HAL-15®) or Vertex-II 6000 (HC-36m®) FPGAs as processing elements (PEs), arranged in a cross-bus configuration reminiscent of Splash-II [1]. The two components—host and RCM fabric—are connected through a common PCI-X bus supporting transfer of program and data bit streams for configuration or execution on the fabric. Specialized FPGAs are used to manage the staging and redirection of program and data to one or more PE units through a crossbar bus, as well as manage the communication between portions of a design residing on different PEs. Each PE has its own local memory, and I/O resources are allocated to support these connections. Such constraints of the architecture of the platform must be taken into consideration when carrying out the estimation, partitioning and re-architecting activities.

For purposes of limiting scope of coverage in this paper, we assume that there is some number of cycles taken to stage data both onto and off of the PE units (through their local memory storage areas), and that bus transfers are 32-bits in width. Therefore, some portion of the architecture’s logic must be allocated and designed to manage the staging of operands for the wide-bit arithmetic units. We will not consider the details of this activity in this paper, other than to note the fact that the bus transfer and staging of operands can assume considerable overhead in the throughput of the resultant RCM application. We are currently evaluating this overhead for two separate applications—one in phylogenetics and another in cryptography. Suffice to

say that one of the architectural objectives is to bury as much of this set-up time as possible through pipelining.

For purposes of this paper, we will concern ourselves with the I/O resources available to use on the Xilinx® devices, and the estimated number of cycles required to pass data off-chip between different components of the wide-bit arithmetic units.

## 5. Estimation and analysis

As we look to explore the overall architecture of wide-bit arithmetic units, namely multipliers, with widths in excess of 192-bits, we understand that the architecture arrived at for a particular application will be hierarchically defined [7][14], so as to be as regular and as modular as possible. It is given that we need to be able to quickly retarget our arithmetic units to different reconfigurable devices, and this involves different partitioning and exploitation of resources to maximize concurrency and minimize resource usage on the target fabric.

We start by examining the performance characteristics of a set of base units to be incorporated into the larger wide-bit units. These base units include smaller multipliers and adders. There is a question we wish to answer regarding the adder units, which is whether examining the costs associated with different adder units is relevant to the overall cost of the multipliers.

To that end, we characterize a set of base multiplier and adder units, of differing architecture and bit-widths, so that we may use this data for estimation of unit cost in the construction of different wide-bit arithmetic units on different RCM fabrics. In addition, we characterize the cost of the individual arithmetic operations that are executed as part of the control regimes associated with the sequencing and scheduling of the operations in the larger units.

### 5.1. Characterizing the base multiplier units

As discussed earlier, all of the multiplier schemes chosen for integer-based, wide-bit multiplication require smaller multiplier and adder units embedded within them. To understand the behavior of each style of unit, we need to know what happens with each of the architectures as it grows in width, so as to understand its scaling properties. For adders, the critical factor is the propagation delay of the carry signal passed between computations of the adder bits. For the multipliers, it is the cost of the various constituent computations—in terms of resources and delay—that affect selection of a clocking scheme. In addition, given that we are not sure what results we'll obtain, we must consider the possible impact of parallelism and pipelining (to improve throughput) on

resource usage and availability (Slices and IOBs on the Xilinx® parts.)

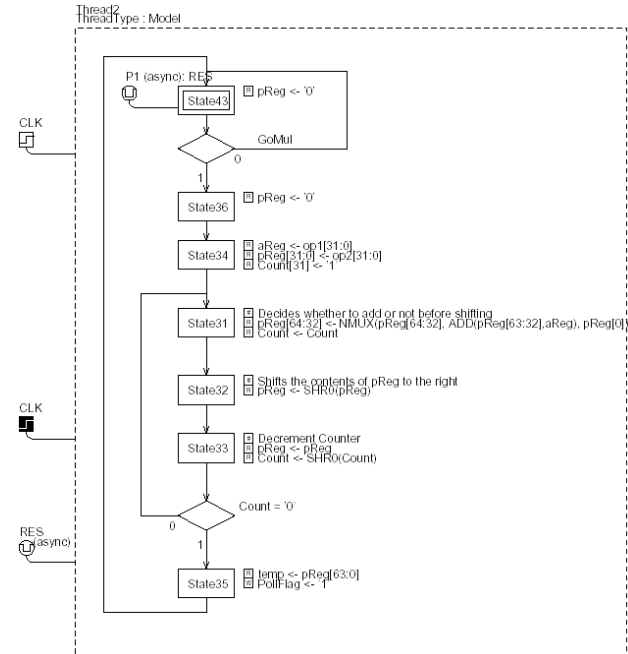


Figure 4. Algorithmic CDFG for shift-add multiplier

Our ability to estimate and evaluate possible architectures is affected by several factors: (1) our ability to “visualize” the control flow, in addition to the data flow, of the architecture under consideration, (2) our ability to relate this control flow to the original algorithm, (3) our ability to explore architecture tradeoffs, in terms of cycle scheduling, resource allocation and sharing, and use of pipelining and parallelism. For this purpose, we adopt the algorithmic state machine (ASM) notation, and employ it as a means to represent cyclic control-data flow graphs (CDFG). CDFGs have been used in the analysis of systolic arrays [7] and in more general custom logic designs [15]. However, our adoption of the ASM notation is consistent in its use as a means for doing top-down HDL-based design [16] [17].

In Figure 4, we depict the ASM diagram for the 32-bit Shift-Add multiplier unit. From the model, we notice several properties about the architecture. First, there are 7 states defined, and one loop involving 3 of these states (the others being primarily concerned with initialization and posting of data). The initial poll loop causes the multiplier unit to stay in its initial wait state until it is enabled. Second, the specific control and data path operations are partitioned among the states, where those operations co-resident in a state are concurrently scheduled for execution. Third, the operational expressions on each state involve assignments between defined buses, and some involve the use of functional operators. These operators map to behavioral VHDL

functions written in a package library. Finally, symbols attached to the ASM thread’s bounding box denote the clocking and reset signals associated with the control and data paths of the multiplier unit.

It should be noted that most other treatments of such architectures ignore the control path (such as in [12]). We find that considering the control scheme along with the data path architecture allows us to consider the impacts of tradeoff analysis—pipelining and parallelism versus resource sharing—in a more explicit manner. This provides considerable advantage when considering the estimation of different architectures for wide-bit arithmetic, as these architectures can have non-trivial control regimes requiring careful consideration of the scheduling and resource allocation issues. These factors can be easily considered using the ASM modeling as discussed.

Multiplier Architecture	IOBs	Slices	Max Delay (ns)
VHDL MUL Operator (**)	130	1067	16.969
Booth (One-hot)	135	233	8.110
Booth (Gray)	135	230	9.067
Knuth (pencil & paper)	135	425	7.243
Shift-Add (One-hot)	131	182	6.500
Shift-Add (Gray)	131	181	7.207
Shift-Add (Binary)	131	180	9.150
Montgomery (MUL op) <sup>1</sup>	200	12950	47.850
Montgomery (Shift-Add) <sup>1</sup>	200	12045	42.720
Montgomery (MUL op)	98	1956	19.879
Montgomery (Shift-Add)	98	505	11.654
Divide and Conquer (Shift-Add)	130	498	6.716

**Table 1. Baseline cost data for multiplier architectures**

Table 1 presents the data set collected for a range of multiplier architectures for the different algorithms referenced earlier, based on mapping to the Vertex-II xc2v6000 part. This data was collected according to the following procedure: (1) graphical ASM models were created using the flowHDL® tool set, and were subjected to cycle-level simulation and debugging in that environment; (2) VHDL code was automatically generated, in a number of different coding styles, from the source ASM model; (3) the VHDL code and associated libraries were synthesized using Synopsys® FPGA Compiler-II® synthesis tools, using both the Xilinx® XC4062 and Vertex-II® XC2v6000 libraries. The subsequent netlist was sent to the Xilinx® place and route tools, whereupon data was collected from the PAR reports, and used as a basis for comparison.

The questions we were asking of the data were these: (1) which multiplier consumed the least amount of resources, and (2) which multiplier had the least worst-case delay. The issue of resource usage is simply one of addition: if we need to employ a number of these base units in a larger wide-bit multiplier design, which one would have the least cumulative cost in resource usage. The issue of delay allowed to estimate at what clock

frequency we could run the unit so as to include it into the larger wide-bit scheme—thereby being able to determine the fastest speed at which we could clock the larger unit.

From this data, we make the following observations: First, the Shift-Add model provided close to the smallest area, but also offered the smallest delay of any of the units. We considered this scheme against the behavioral multiplier operator—which we assumed would be synthesized and mapped most efficiently to the Xilinx® parts. But this was not the case, for which further questioning is required.

Second, using several of the models, we created resultant VHDL descriptions that used different coding styles for the state machines. It has been reported in the literature recently that coding styles of source VHDL significantly affected quality of resultant architecture on FPGAs [14]. Although, the number of states is not large in the example Shift-Add model, we were surprised at the difference in delay between the One-hot encoding (using a separate register for each state variable output) versus either the standard Binary coding or the more optimal Gray coding schemes. With minimal variance in used area, we observed a 3 ns reduction in the circuit delay by adopting a One-hot coding of the state machine for this multiplier. Therefore, for the Shift-Add model, One-hot encoded, we assume we can clock the model with a 10ns clock (allowing for the setup and hold times for registers in the Vertex-II device).

Third, we anticipated that the Montgomery multiplier scheme would be close in area and delay, so that the use of its reduction scheme on the width of its output bus (outputting a 32-bit product as opposed to a 64-bit product) would ultimately result in less routing resources and IOB consumption when sending the data off-chip. We developed 4 different Montgomery models: a pair using the behavioral VHDL MUL operator and another pair using the Shift-Add as its base unit; and, a pair with these base multipliers using different schemes for the division in the modulo reduction [12]. The Montgomery multiplier uses different multiplier units of the same radix to perform its calculations; the 32-bit Montgomery employs 3 separate 32-bit Shift-Add multiplication operations, along with the modulo reduction operations, to carry out its algorithm. In the design created, the Shift-Add unit was resource-shared, as the different multiplication computations occur on different time steps. The use of shifting instead of division to achieve modulo reduction was able to reduce the resource usage by an order of magnitude. The Shift-Add version with shifting for modulo reduction was considerably smaller and faster, as seen from the data in Table 1.

Since the Montgomery scheme we employed includes one Shift-Add multiplier, resource-shared over three different executions, we expected the Montgomery to be larger. However, it is larger by a factor of 2.5, and slower

by a factor of 2. It remains an issue as to whether the benefit of reduced multiplication Product width (where resultant data path operations are halved downstream from the multipliers as a result of using the Montgomery scheme) is offset by its cost. This issue will be discussed again later when we consider the construction of the 192-bit and 256-bit width multipliers.

## 5.2. Characterizing the base adder units

The adder units we explore are some of those whose algorithms are found in [8]: Ripple-Carry (RPC), Carry-Select (CS), Carry-Look-Ahead (CLA), and Bit-Serial (BS). We have created multiplier designs in VHDL and have used the behavioral Addition operator to specify these functions. Our question is whether the cost of this behavioral VHDL Add operation (directly affected by the quality of optimization carried out by the synthesis tools) is in line with that associated with standard structural Adder models that we might use.

To find out, we followed the same development process employed for the multipliers. Creating models for each of the addition schemes, we collected data on the various runs from which we could compare and check our assumptions. The base data we collected for the Adders is given in Table 2.

64-bit Adder Architecture	IOBs/Nets	Slices	Max Delay (ns)
VHDL Add Operator (+)	193	85	6.144
Ripple Carry	193	64	6.386
Carry Look Ahead	226	136	6.404
Carry Select	175	175	5.892
Bit Serial (One-Hot)	100	62	7.024
Bit Serial (Gray)	100	63	4.899

Table 2. Baseline cost data for adder architectures

As discussed earlier, all of the multiplier schemes chosen for integer-based, wide-bit multiplication require smaller adder units embedded within them. To understand the behavior of each style of unit, we need to know what happens with each of the architectures as it grows in width, so as to understand its scaling properties. For adders, the critical factor is the propagation delay of the carry signal passed between computations of the adder bits [8].

If we are to use the 64-bit Add units, as shown in the Broadcast multiplier in Figure 1, then the behavioral 64-bit ADD is considerably smaller than the other schemes, with only a 2/10 of a nanosecond loss in delay over the next-best unit, the Carry Select. So, we continue to use the behavioral Add operator within the various Multiplier units.

When considering the use of the 32-bit Montgomery multiplier as the base unit, its constituent adders would only need to be 32-bits in width, as the partial products of the Montgomery would be 32-bits rather than 64-bits in

width. This cost difference is considered later in this paper.

## 5.3. Characterizing the macro primitives

As mentioned earlier, the multiplier units and adder units (at all levels of abstraction and composition) use a number of behavioral macro-function primitives to carry out operations required in the control and manipulation of the data paths. Another aspect of the estimation and exploration process is to examine the costs associated with these macros at different operand widths, to understand their contributing cost as the design units scale in size.

Macro function	Macro Cost			Macro function	Macro Cost		
	CLB/slice	IOB	delay (ns)		CLB/slice	IOB	delay (ns)
AND_4	4	12	1.478	CAT_16	0	64	2.678
AND_8	8	24	1.792	CAT_32	0	128	6.56
AND_16	16	48	1.877	CAT_64	0	256	6.665
AND_32	32	96	4.205	ADD_4	4	13	1.241
AND_64	64	192	6.025	ADD_8	11	25	1.846
OR_4	4	12	1.478	ADD_16	21	49	2.501
OR_8	8	24	1.792	ADD_32	43	97	3.685
OR_16	16	48	1.877	ADD_64	85	193	6.144
OR_32	32	96	4.205	SUB_4	4	13	1.241
OR_64	64	192	6.025	SUB_8	11	25	1.846
XOR_4	4	12	1.478	SUB_16	21	49	2.501
XOR_8	8	24	1.792	SUB_32	43	97	3.685
XOR_16	16	48	1.877	SUB_64	85	193	6.144
XOR_32	32	96	4.205	INCR_8	6	17	1.855
XOR_64	64	192	6.025	INCR_16	13	33	2.239
NMUX_8	4	25	1.513	DECR_8	6	17	1.452
NMUX_16	8	49	3.204	DECR_16	13	33	2.239
NMUX_32	16	97	5.875	DECRNC_8	5	16	2.21
NMUX_64	32	193	7.502	DECRNC_16	12	32	1.975
NMUX_128	128	385	16.965	MUL_4	17	16	2.52
MUX2_8	4	25	1.513	MUL_8	79	32	3.362
MUX2_16	8	49	3.204	MUL_16	339	64	6.645
MUX2_32	16	97	5.875	MUL_32	1067	130	16.969
MUX2_64	32	193	7.502	MUL_64			
SHRIN_8	0	16	1.517	MUL_128			
SHRIN_16	0	32	1.602	REM_16	576	48	7.864
SHRIN_32	0	64	2.662	REM_32	2776	96	18.355
SHRIN_64	0	128	6.114	REM_64			
SHRO_8	0	15	0.78	SHL0_8			
SHRO_16	0	31	1.224	SHL0_16			
SHRO_32	0	63	2.98	SHL0_32	0	63	3.545
SHRO_64	0	127	5.847	SHL0_64	0	127	5.889

Table 3. Base cost data for behavioral macros by width

Table 3 presents the cost data associated with the behavioral macro-functions of differing widths, many of which are used in the ASM/VHDL models created in the various Adder and Multiplier architecture considered in this paper. Note that some of the data was unavailable at the time of printing. Also, the Xilinx® PAR reports for certain functions, such as shifting, reported zero Slice usage. This may be because of the fact that the Vertex-II architecture supports shifter and shift-carry configurations directly from its LUT structure [18]. Regardless, we assume the area cost of shifting, in terms of CLB resources is based on the published data of up to 32-bits of shifting per Vertex-II Slice.

Our interest is such low-level cost data is that the use of behavioral macros is ubiquitous throughout our VHDL models. We have adopted a behavioral/RTL code style and method of VHDL development, as is common in the industry for ASIC designs. We wish to confirm that our

coding practices are appropriate for building FPGA designs in reconfigurable computing applications. As reported in [14] [15], we cannot take code style and level of design abstraction for granted.

At this juncture in our research, we have not compared the efficacy of these behavioral macros, built into a Synopsys® library, versus those mappings provided in Xilinx’ own development environment. Of specific interest is the comparison of the mappings to dedicated Multiplier resources that are available within the Slice structure of the Vertex-II architecture [18]. However, since we are considering the problem of mapping to any family of devices (the other at present being the Xilinx XC4062 device used in the Star Bridge HAL-15® RCM platform), we have not focused on these issues as part of the current efforts reported in this paper.

In performing an inventory of the various macro operations used in a base model, say, the 192-bit Broadcast model of Figure 1, our objective is to insure that the specified operations—which can be nested to arbitrary levels of depth in the ASM notation—are computable within the clocking cycle times we are considering as part of this estimation process. We come back to this problem when we consider the clock rate for the wide-bit multiplier in the next section.

## 6. Estimating the wide-bit architecture

To this point, we have concentrated on our methodology of collecting data on the various functional units that are used to construct our candidate base unit architectures. For this purpose, we decided on a unit width of 32-bits, from which we will construct the wider bit-width units. The wide-bits we examine in this paper are 192-bits and 256-bits, respectively. First, by running a 192-bit Broadcast model through the process discussed, we obtain data on the design for the Vertex-II device, on which this width of multiplier fits without modification. The ASM model for this multiplier is shown in Figure 5.

The representation shown in Figure 5a for the multiplier corresponds to the CDFG for the data path diagram of Figure 1. The CDFG in Figure 5 consists of a main thread, and a sub-thread consisting of demultiplexing logic executed as Mealy-style conditional outputs via a multiway branch taken while in state ‘State6’. The ASM diagram shows an initial version, taken from the algorithmic description, and therefore does not employ explicit pipelining, since we have a single control thread indicated. Obviously, to make this a fully pipelined control structure, we would refine this model such that the control states associated with each processing step in the data path corresponded to a separate thread of execution—meaning there would be a separate CDFG for each processing thread in the pipeline.

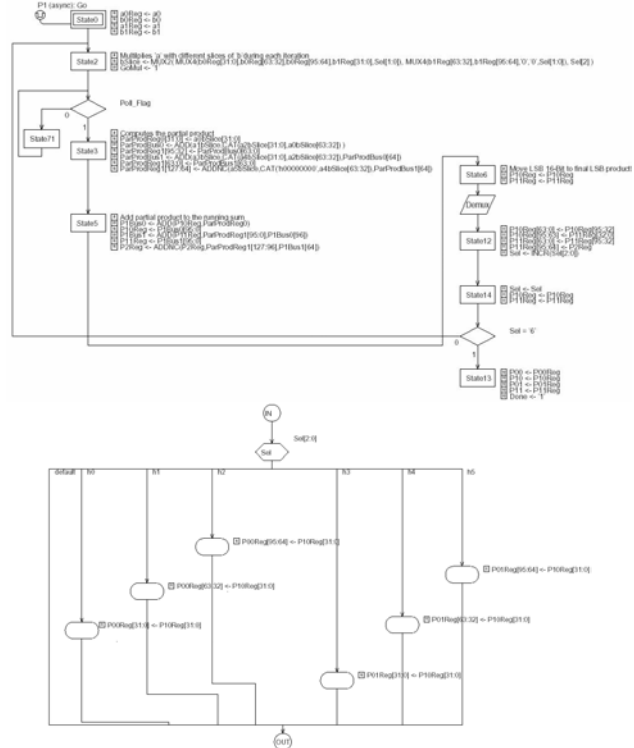


Figure 5a&b. Algorithmic CDFG for BC-192 multiplier

However, note that in the ASM diagram of the CDFG, we have a single signal that enables all six of the 32-bit Shift-Add multipliers in parallel (which exist as six separate instances of the Shift-Add model shown in Figure 4, with appropriate instantiation of the signals for each unique multiplier component).

Also, we estimate this model to compute the multiplication in some number of clock cycles that we can arrive at by counting cycles implied by the presence of the discrete states in the ASM diagram, plus those of the diagram in Figure 4. We do this estimation and analysis as follows, in a similar scheme as those presented in [21] [22] [23]. For the 32-bit Shift-Add units, each will run through its sequence of states while the main BC-192 control sequencer is in its poll state. As such, we assume the poll state gets executed at least once, based on the extra cycle required to latch the control signals.

Multiplier Unit	Non-loop states	Loop states	Loop count	Cycles
Shift-Add-32	4	3	32	100
Broadcast-192	2	7	6	44
Total cycles				144

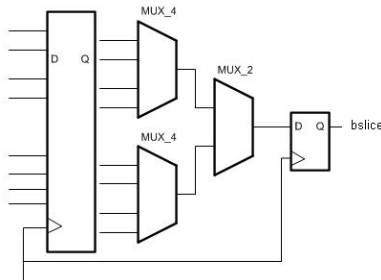
Table 4. Clock cycle data for Broadcast-192 multiplier

In Table 4, we show this estimation, given assumptions of the current model, to run a 192x192 multiplication in 144 clock cycles, which is also the latency of the unit. Next, before considering the full cost of the unit, we must settle on a clock frequency estimate,

which we will derive based on our delay data for the various elements of the BC-192.

As previously mentioned, we need to consider the delay of individual CDFG statements that are scheduled for execution in each given state, to insure that the construction of these expressions insure they can be completed within the allotted time frame of the clock period. From the CDFG in Figure 5a, we have one nested macro operation defined (corresponding to an appropriate VHDL statement, not shown) for the multiplexing in of multiplier operands. Now, the multiplicand operands are successively loaded into holding input registers (which is effectively one 32-bit operand chunk of the 128-bit whole per PCI-X bus cycle, given the current Star Bridge busing architecture [13]), and the next 6 cycles are required to “broadcast” the 32-bit operand chunk of the 128-bit multiplier operand to all six Shift-Add units in parallel. However, we have a nested MUX structure indicated in the CDFG for which we need to check our delay constraints against clock cycle time assumptions. The data path for this nested MUX structure, similar between the BC-192 and BC-256, is shown in Figure 6.

```
bslice <- MUX2 (MUX4 (in11, in12, in13, in14, sel_1),
                MUX4(in21, in22, in23, in24, sel_2),
                sel_0)
```



**Figure 6. BC-192 multiplier nested MUX structure**

From Table 3, we see that the basic MUX macro primitive has a calculated PAR delay of 5.875 ns. The Vertex-II CLB structure [18] defines a set of MUX configurations whereby nested 2:1 MUXes can be chained to create larger MUX structures. If we assume that our mapping of this structure, sans optimization, leads us with a structure that is 3-levels deep, we would estimate a total delay for this operation to be about 17.625 (also assuming the routing information is embedded in the delay calculation returned in the Xilinx® PAR report).

At this juncture, and given the data reported thus far, we have a choice of options regarding our clocking and operation scheduling. One option is the obvious—our minimum clock cycle would be on the order of 20 ns, including timing for setup and hold. Another option is to break up the staged MUX into two separate operations synchronized on different clock cycles, or even clock

edges, whereby we could run the clock at a considerably higher speed.

Given the data in Tables 1, 2 and 3, we see that the maximum delay incurred through various elements (including the 32-bit base Shift-Add multiplier) is on the order of 6-7 ns, so that we might be able to estimate a clock cycle of 10 ns instead, incurring the extra cycle penalty for the MUX operation on loading of the 32-bit chunks of the broadcasted 192-bit multiplier operand, resulting in 6 additional cycles involved in the multiplication to achieve the partial products. On simply relying on these estimates, it appears that breaking up the operation across cycles wouldn’t save us anything—unless we were to explore a two-phase or inverted edge clocking scheme, both of which are possible using the Vertex-II device architecture [18].

The resultant data collected for a fully synthesized and mapped BC-192 are shown in Table 5. We note that, given the FSM control structure, a Gray coding yields a faster circuit. This data also indicates that we likely could not use a 10 ns cycle time, but perhaps a 15 ns time might be feasible—allowing our design to operate at roughly the same 66 Mhz clock cycle time as our PCI-X bus feeding us data. This has many benefits, namely, being able to take in data and operate on it, at-speed, on input.

BC-192 Architecture Run	IOBs/Nets	Slices	Max Delay (ns)
Flat, Gray Encoded	740/824	2227	11.451
Flat, One-hot Encoded	740/824	2178	14.578
Hierarchical, One-hot	740/824	2228	16.455

**Table 5. Cost data for mapped Broadcast-192 multiplier**

In discussing the same architecture extended for the BC-256 case, the Xilinx® tools indicate that there are insufficient IOBs (which is expected [14], as a 256x256 would yield 512-bit output, consuming most of these scarce resources. In considering the design options in this scenario, we either would need to explore use of the Montgomery scheme, and thus reduce the bit-width through the Adder stages and, ultimately, to the output. However, we have opted for a different solution involving resource sharing of the 128-bit output registers; therefore, instead of using a 512-bit final product register, we define a single 128-bit register that is MUXed across cycles.

This is possible, and perhaps preferable, due to the fact that, once pipelining, the final-stage accumulation will be shifting its data to the right by 32-bits, after each final Add operation. Once we have shifted 4 cycles of 32-bits, we would be able to start outputting the first 128-bits of the final product. After 4 additional cycles, we’d be able to output the second 128-bits of the product. For the next 2 cycles we’d write the low and high-order longwords of the Accumulator register itself into the MUX for outputting to the 128-bit holding register.

## 7. Summary

In this paper, we have presented an analysis and estimation method for assessing the cost tradeoffs and clocking characteristics of wide-bit arithmetic units. Although the immediate application for this work is in constructing large multipliers of varying bit-widths for crypto-arithmetic applications, we believe this analysis method—consisting of creating ASM diagrams as CDFGs, and using collected library data for behavioral macros and units of differing widths, as the basis for estimation and tradeoff analysis—can be applied to other applications. We discussed this method in the context of devising architectures for 192-bit and 256-bit multipliers.

For multipliers larger than 256-bits, we might want to reconsider the cost tradeoff of the Montgomery against the cost associated with more resource sharing. It is clear from the Slice numbers that the Vertex-II is quite able to handle the logic of the wide-bit multipliers. The critical resource is the IOBs for outputting data—which we can effectively resource-share. At issue, however, are the requirements for the number of each such unit configured onto the RCM fabric such that all of the multiplications required of the multi-precise and elliptic curve arithmetic can be effectively parallelized and pipelined to maximize throughput while limiting off-chip communication costs.

## 8. References

- [1] D.A. Buell, J.M. Arnold, and W.J. Kleinfelder (eds.), *Splash-2: FPGAs in a Custom Computing Machine*, IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [2] S. Hauck, “The roles of FPGAs in reprogrammable systems,” *Proceedings of the IEEE*, Vol. 86, 1998, pp. 615-639.
- [3] K. Compton and S. Hauck, “Reconfigurable computing: A survey of systems and software,” *ACM Computing Surveys*, Vol. 34, No. 2, June 2002, pp. 171-210.
- [4] Chen, M. C., and C. A. Mead, “Concurrent Algorithms as Space-Time Recursion Equations”, in Kung, S. Y., H. J. Whitehouse, and T. Kailath (eds.), *VLSI and Modern Signal Processing*, Prentice-Hall Publishers, 1985.
- [5] D.A. Buell, J.P. Davis, and G. Quan, “Reconfigurable Computing Applied to Problems in Communications Security”, in *Proceedings MAPLD-02*, Laurel, MD, 2002.
- [6] National Institute for Standards and Technology, “Recommended elliptic curves for federal government use,” *Technical Report*, NIST, [www.csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.ps](http://www.csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.ps), 1999.
- [7] S.Y. Kung, *VLSI Array Processors*, Prentice Hall Publishers, Englewood Cliffs, NJ, 1988.
- [8] Parhami, B., *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, 2000.
- [9] Carpinelli, J. D., *Computer Systems Organization and Architecture*, Addison-Wesley Publishing Co., 2001.
- [10] Knuth, D. E., *The Art of Computer Programming, Volume 2, Semi-numerical Algorithms*, 3<sup>rd</sup> ed., Addison-Wesley Publishing Co., 1997.
- [11] Flynn, M. J., and S. F. Oberman, *Advanced Computer Arithmetic Design*, John Wiley and Sons, Inc., 2001.
- [12] A.F. Tenca, and C.K. Koc, “A Scalable Architecture for Montgomery Multiplication”, in C.K. Kok and C. Paar (eds.), *Proceedings CHES '99*, LNCS 1717, Springer-Verlag Publishers, Berlin, Germany, 1999, pp. 94-108.
- [13] Star Bridge Systems, *Description of the Hypercomputer® Hardware*, at <http://www.starbridgesystems.com>.
- [14] W.J. Fang, and A.C.H. Wu, “Multiway FPGA Partitioning by Fully Exploiting Design Hierarchy”, *ACM Transactions on Design Automation of Electronic Systems*, Association for Computing Machinery, Vol. 5, No. 1, Jan. 2000, pp. 34-50.
- [15] S. Bakshi, and D.D. Gajski, “Performance-Constrained Hierarchical Pipelining for Behaviors, Loops, and Operators”, *ACM Transactions on Design Automation of Electronic Systems*, Association for Computing Machinery, Vol. 6, No. 1, Jan. 2001, pp. 1-25.
- [16] J.P. Davis, S. Nagarkar and J.K. Matthewes, “High-level Design of On-chip Systems for Integrated Control and Datapath Applications”, in *Design Supercon-96: On-Chip System Design Conference*, Hewlett Packard Company, Inc., 1996.
- [17] C.H. Roth, Jr., *Digital Systems Design Using VHDL*, PWS Publishing Company, Boston, MA, 1998.
- [18] Xilinx, Inc., *Vertex-II 1.5V Field Programmable Gate Arrays: Advance Product Specification*, DS031-1, DS031-2, and DS031-3, September 22, 2002.
- [19] F.R. Boyer, and E.M. Aboulhamid, “Optimal Design of Synchronous Circuits Using Software Pipelining Techniques”, *ACM Transactions on Design Automation of Electronic Systems*, Association for Computing Machinery, Vol. 6, No. 4, Oct. 2001, pp. 516-532.
- [20] M. Balakrishnan, and H. Khanna, “Allocation of FIFO Structures in RTL Data Paths”, *ACM Transactions on Design Automation of Electronic Systems*, Association for Computing Machinery, Vol. 5, No. 3, Jul. 2000, pp. 294-310.
- [21] A. Darte, R. Schreiber, B.R. Rau, and F. Vivien, “Constructing and Exploiting Linear Schedules with Prescribed Parallelism”, *ACM Transactions on Design Automation of Electronic Systems*, Association for Computing Machinery, Vol. 7, No. 1, Jan. 2002, pp. 159-172.