Divide-and-Conquer Approach for Designing Large-operand Functions on Reconfigurable Computers

Miaoqing Huang, Esam El-Araby, and Tarek El-Ghazawi
Department of Electrical and Computer Engineering, The George Washington University
{*mqhuang,esam,tarek*}*@gwu.edu*

Abstract

Reconfigurable computers (RCs) are gaining rising attention as an alternative to traditional processors for many applications. However, using RCs for operations with either large-size operands or large number of operands can be challenging given the bounded reconfigurable resources. In this paper, we propose a systematic methodology based on a Divide-and-Conquer technique for the implementation of such operations on RCs. A generic architecture is presented where schedulers are necessary to virtualize the usage of the limited resources. Two case studies, namely large-size-operand multiplication and large-number-of-operand FFT representing reduction and transformation operations respectively, were selected to demonstrate the effectiveness of the proposed approach. The experimental work was performed on one of the current RCs, i.e., SRC-6 reconfigurable computer. The tradeoff between performance and resource utilization is also presented.

1. Introduction

Reconfigurable computing has been receiving more attention in recent years [1, 2, 3, 4, 5]. Because of their high degree of fine-grain parallelism, Reconfigurable Computers (RCs) can achieve orders of magnitude speedup, when compared to conventional platforms for certain inherently parallel applications [1, 2, 5]. Nevertheless, researchers have been investigating performance maximization of less inherently parallel applications on RCs [6].

In some scientific fields, certain large-size operations need to be evaluated. For example, in RSA cryptographic system, it is necessary to perform long-integer modular multiplications, e.g., 1024bit$\times$1024bit, and the efficiency of RSA algorithm largely depends on the speed of the multiplication. Moreover, the word-length of current computers is fixed at either 32 or 64 bits. In order to perform a 1024bit$\times$1024bit multiplication, the multiplication unit must be recalled multiple times in sequence. ASICs can be candidates for such a particular arithmetic operation. However, RCs based on FPGAs provide a more flexible solution, which can be used for different operations if a general approach to implement the corresponding algorithms in hardware becomes available.

In general, large-operand functions can be classified into two categories, i.e., large-size-operand functions and large-number-of-operands functions. In the first category, the number of operands may be small while the precision of the operands can be arbitrarily large. One typical example of this category is reduction operations such as multiplication in which there are two input operands and one output while the bit-length of the operands may change according to the needs of different applications. In the second category, the number of operands can be arbitrarily large while the precision of the operands is relatively small. Typically, transformation operations such as FFT and sorting belong to this category. For this type of operations the number of the input operands equals the number of outputs, which can be as large as needed while the precision of each operand is limited by the system word-length, i.e., 32 or 64 bits.

In this paper, we present a generic architecture for large-operand arithmetic developed based on a Divide-and-Conquer technique. In this architecture, the first major component is a small-operand arithmetic unit, which serves as the kernel for the operation under investigation. The second major unit is a scheduler, which is responsible for generating the access patterns for both inputs and outputs from and back to the system memory. Depending on the operation being considered, an optional third component is a merging unit, which is responsible for combining partial and intermediate results into the final outputs.

The paper is organized as follows. In Section 2, related work is described and the contribution of this paper is highlighted. In Section 3, Divide-and-Conquer as a general design technique is discussed and proposed for the implementation of large-operand functions. A

reduction operation, i.e., large-size multiplication, and a transformation operation, i.e., large-size FFT, are selected as case studies and discussed in detail in Section 3 as well. Section 4 presents the implementation results for both case studies on SRC-6 reconfigurable computer. Finally, Section 5 summarizes and concludes this work.

2. Related work

A large body of previous work had investigated the concept of using a small-size unit to perform operations of large operands. In [7], the authors exploited the performance of 32-bit floating point arithmetic in order to obtain a 64-bit accuracy in solving linear algebraic systems using an iterative refinement methodology. In [8], the authors presented a specific CORDIC processor for variable-precision coordinates. The system allowed to specify the precision to perform the CORDIC operation, and control the accuracy of the result. In [9], the authors proposed arbitrary precision packed arithmetic in which the width of the subdatatypes is programmable by the user. They also proposed an implementation for arithmetic on such packed datatypes with marginal hardware overhead. In [10], various integer arithmetic algorithms were presented where a systolic multiplier module and the Chinese remainder algorithm were used to implement multiplication and division respectively. In [11], a 256-bit ALU that could be dynamically reconfigured by a program was presented.

Most of the previous work discussed above either only addressed the accuracy issue of using smaller unit to perform operations of large-size operand [7, 8], or only focused on a particular set of operations [9, 10, 11]. A more generic approach in hardware is desired to cover much broader range of operations. The Divide-and-Conquer approach presented in this paper can be applied to any large-operand functions such as large-size operand functions and/or large-number-of-operands functions, e.g., reduction operations and transformation operations.

3. Divide-and-Conquer Approach

Divide-and-Conquer is an important algorithm design paradigm. A problem, using Divide-and-Conquer, is recursively broken down into two or more sub-problems of the same (or related) type, until these sub-problems become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. A Divide-and-Conquer algorithm is closely tied to a type of recurrence relation between functions of the data in question where data is "divided" into smaller portions and the result is then calculated. This technique is the basis of efficient algorithms for versatile

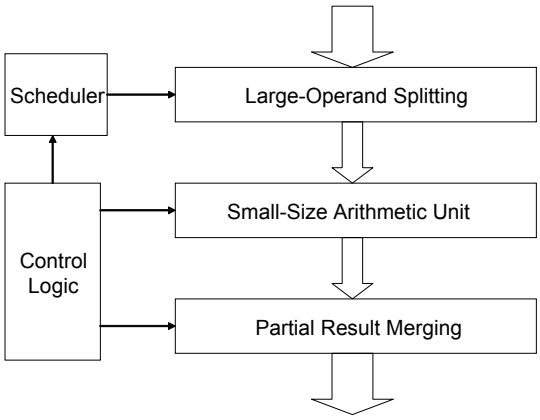

Figure 1. Proposed Hardware Architecture.

kinds of problems, such as sorting (quicksort, merge sort) and the discrete/Fast Fourier transform (FFTs).

In computer science, the pattern for using Divide-and-Conquer in solving a problem is a top-down process that involves a series of recursive function calls. Recursive structure has been shown very difficult to achieve satisfactory performance in hardware implementation [6]. Nevertheless, for the purpose of hardware implementations, using Divide-and-Conquer can follow a bottom-up approach in which a scheduler is responsible for fetching from and storing to memory the pieces of the operands following particular patterns. In this approach, the scheduler is an essential component. We will discuss the construction of this scheduler in detail in this section.

3.1. Proposed Architecture

In order to adopt Divide-and-Conquer for the implementation of large-operand functions, a small-size kernel unit will have to be utilized, and the original operands have to be broken down into small pieces, each of which is either the operand with less precision or a subset of the original total operands. The term "operand" is used to denote the original large operand, and the term "suboperand" is used to denote the operand that is fed to the small-size processing unit. In addition, there must be a scheduler, which generates the pattern to fetch from and/or store to memory these suboperands and feeds them into and/or receives them from the small-size arithmetic unit. These components are shown in Figure 1. For cases such as reduction operations, the partial and/or intermediate results, which are generated by the small-size unit, have to be combined together to form the final result through a merging unit. Other cases such as transformation operations, the merging step is not needed. In general, the merging step is optional depending on the operation

being implemented. In the following two subsections, the two different cases of multiplication as a reduction operation as well as FFT as a transformation operation will be discussed in detail to demonstrate how to generate the fetching schedule for each case.

The scheduler is a key component in the design of large-operand arithmetic units. As explained earlier, the original operands are to be split properly and stored in the memory; and the scheduler follows a particular fetching pattern to read the suboperands simultaneously and feed them into the small-size unit. In implementing the scheduler, two design objectives are to be considered:

1. The whole structure of the arithmetic unit should be easy to scale up or down. In other words, the modification of the overall structure should be minimal in order to support large-size operands.

2. The design of the arithmetic unit can be performance oriented or resource oriented. In the case of performance oriented unit, the merging should be performed on the fly, which means that the small-size unit should never stall and merging should be area-optimal. For the case of resource oriented unit, merging should be performed in memory where the memory space would be used to store all the partial results and the same space would be reused for the merging.

In addition, the scheduler fetching/storing patterns are generated as follows:

1. Divide each operand into two suboperands.

2. Do the calculations among these suboperands, depending on the particular algorithm and following a certain sequence. The calculations can be the real calculations that involve the small-size unit if the size of suboperands matches the acceptable operand for this unit (kernel). Otherwise, calculations can be thought of as imaginary calculations that will be replaced by real ones later. The sequence of calculations should be recorded for later usage.

3. If these suboperands can be divided further, go back to step 1. The new sequence will replace and expand the corresponding part of original sequence.

It is worth mentioning that the input operands are assumed to be stored in separate memory modules providing a high degree of parallelism as needed by FPGAs. Furthermore, it is assumed that the memory modules can take arbitrary addresses every clock cycle with fixed latency.

3.2. Reduction Operations

Reduction operations are those operations whose number of outputs is less than the number of input operands. We will consider these operations as a case study for the first category of large-operand operations, namely large-size-operand functions. In other words, number of operands is limited while the precision of operands is arbitrarily large. For the reduction operations, all components of the general architecture in Figure 1 can be used. By including a small-size processing unit, the first step is to divide the original inputs into small suboperands. This step is quite straightforward, however, the manner through which fetching these suboperands for the small-size processing unit decides the complexity of design of merging step and the performance of the overall structure. All suboperands go through the second step, which is processing through the small-size unit. In order to maximize the performance of the overall structure, this unit is assumed to be fully pipelined, i.e., the unit is capable of taking new suboperands every clock cycle. Although the third step, i.e., merging step, accepts the partial results passively, the structure of this step determines the performance of the complete design. As mentioned earlier, there are two objectives, i.e., performance-oriented or resource-oriented, which can be followed in the design of the merging step. If the best performance is preferred, the merging can be done on the fly, i.e., the merging step is capable of consuming one partial result per clock cycle and never stalls the processing unit. In order to realize this goal, the merging step costs more resources; and the maximum size of the overall design is subject to the logic resources available in the target FPGA. On the other hand, the merging can be done in memory; and the scalability of the design can increase indefinitely. Instead of storing the partial results in registers, they are stored in memory external to the FPGA. The merger starts working on the same scope of memory space used for the input operands after all the partial results are generated. The small-size processing unit must stall until the merging step is finished. Then, it can resume its work for new large operands. In Section 3.2.1, an example of such reduction operations, i.e., 512bit$\times$512bit multiplier, is discussed in details. The discussion would be more focused on the construction of the scheduling and merging steps. The same design flow can be applied to similar algorithms.

3.2.1. Large-size-operand Multiplication

Algorithm 1 as proposed in [12] shows how to divide the original operands into two pieces and obtain the final results based on the partial products. If the suboperand is still too wide after the first splitting, the splitting process continues

Algorithm 1: Multiplication using small multiplier[12].

$A = A_1 \cdot 10^k + A_0;$ $\quad /\star A_0 < 10^k \star /$
$B = B_1 \cdot 10^k + B_0;$ $\quad /\star B_0 < 10^k \star /$
$C = A \cdot B$
$= (A_1 \cdot 10^k + A_0) \cdot (B_1 \cdot 10^k + B_0)$
$= A_1 \cdot B_1 \cdot 10^{2k} + (A_1 \cdot B_0 + A_0 \cdot B_1) \cdot 10^k + A_0 \cdot B_0;$

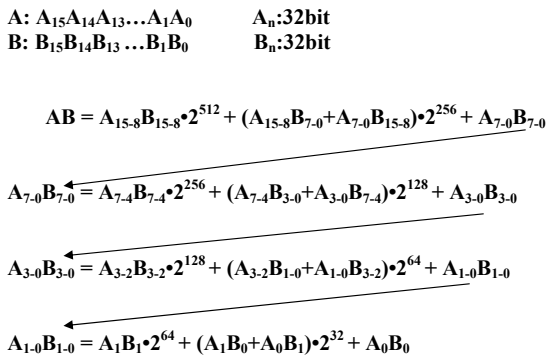

Figure 2. 512×512 Multiplier using Divide-and-Conquer.

until the size of suboperand fits the small-size multiplier that does the real work. In the case of Algorithm 1, the original operands are assumed to be *n*-bit long. Instead of having one $n \times n$ multiplication, there are four $\frac{1}{2}n \times \frac{1}{2}n$ multiplications after the first round of operand-splitting. If the $\frac{1}{2}n$-bit operand still can not be accepted by the small-size multiplier, the splitting can be done further, i.e., $\frac{1}{4}n$ and so forth, until it reaches the acceptable size designed for the small-size multiplier. In addition, the number of partial results decreases as the merging advances until the final result is reached. Generally this scheme can also be applied to other arithmetic problems such as large-size matrix multiplication.

Multiplier Parameters

As an example, we are considering the implementation of a 512bit×512bit multiplier using a 32bit×32bit multiplier. The 32bit×32bit multiplier is the small-size processing unit that does the real multiplication. Figure 2 depicts the process of breaking down the 512bit×512bit multiplication until the suboperands fit the small-size 32bit×32bit multiplier. Firstly, the operands are divided into two halves, and so on, until the length of the suboperands reaches 32 bits. In other words, the operands are split into 16 subsets respectively, i.e., A_{15} through A_0 and B_{15} through B_0, each of which is 32-bit long. In Figure 2, A_{m2-m1} means the concatenation from A_{m2} through A_{m1}.

Table 1. Register Requirement of the Merging Step

Stages	# of register elements	Width	Total bits
4^{th}	4	512-bit	2,048
3^{rd}	16	256-bit	4,096
2^{nd}	64	128-bit	8,192
1^{st}	256	64-bit	16,384

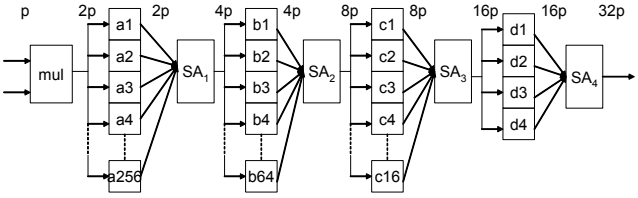

Figure 3. Four Stages of the Merging Step.

The Merging Step

If one traces the steps shown in Figure 2 from bottom to top, it can be found that the process goes through four stages. The partial results entering the four stages are 64-bit, 128-bit, 256-bit and 512-bit wide respectively. The output of each stage is two times wider than the input; and the final result will be 1024-bit wide. The four stages in Figure 2 are labeled as 1^{st} stage, 2^{nd} stage, 3^{rd} stage, 4^{th} stage from bottom to top. The 4^{th} stage involves four partial results of 512-bit wide, and it needs four registers to store these results before merging them into the final result. Each element in the 4^{th} stage requires 4 outputs from 3^{rd} stage, which means that the 3^{rd} stage needs 16 registers of 256-bit wide. The same rule applies to 2^{nd} and 1^{st} stages as well. Table 1 summarizes register requirements for each stage to retain these partial results. Between two consecutive stages of registers, one Shifter-and-Adder (SA) will be inserted to merge the partial results to wider ones. The width of input and output of SA depends on the stage to which it belongs.

–Merging on the fly

In Figure 3, a_n, b_n, c_n and d_n are registers of different sizes. The direct implementation of the above architecture demands a lot of hardware resources and it is also difficult to scale up. However, further optimizations can be performed. It can be seen from Figure 4 that every four register

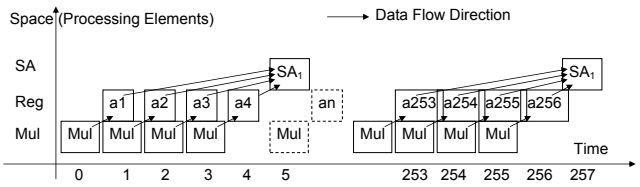

Figure 4. Time-Space Relationship of the First Stage.

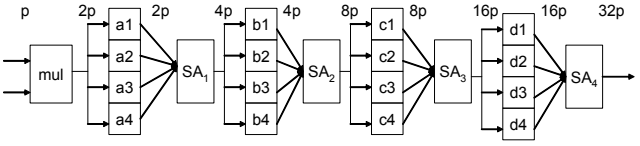

Figure 5. The Optimized Merging Step.

elements form one quadruplet. After the data of one quadruplet are forwarded to the SA, it will not be updated again until the small-size multiplier begins to process the next two large-size operands, which means that it is not necessary to retain these data after the following SA has received them. Instead of retaining 64 quadruplets in the first stage, one quadruplet is enough. The same optimization applies to the following stages as well. No matter how long the original operands are, each stage of the merging step needs only one quadruplet to save the intermediate result. Figure 5 shows the optimized architecture of Figure 3. In Figure 5, there are four different size SA units that do the merging job. The design of large-size SA units can follow the same methodology described at the beginning of Section 3.2, i.e., utilizing a small-size adder in the center of the SA unit.

–Merging in memory

Instead of merging the partial products on the fly, they can be stored in external memory and merged later. The only limitation is the capacity of the available memory, which is quite large compared to the registers in one FPGA device. The 4^{th} column in Table 1 shows the storage requirement of each stage, which is half of the requirement for the preceding stage. Following the same procedure as in *Merging on the fly*, the partial results can be processed in the same memory until the final product is reached. The SA unit that handles these partial results has to double its I/O size when the merging advances from one stage to the next. Additionally, the reading pattern of the scheduler with the SA has to be adjusted for each stage. This particular scheduler generates the addresses of partial results for SA.

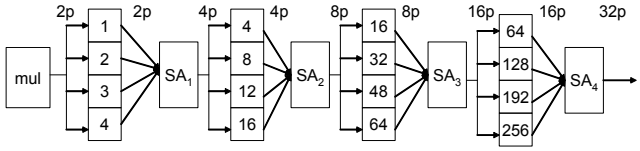

Figure 6. The Data Time Stamps Per Stage.

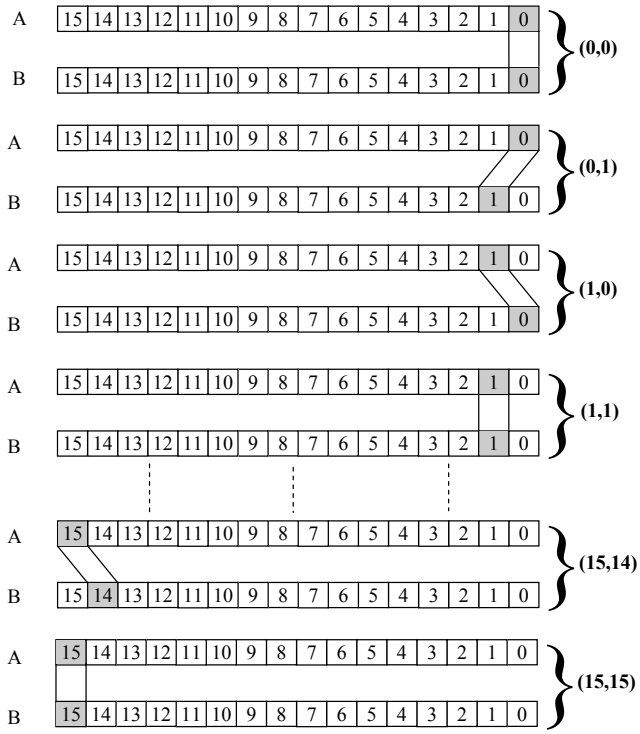

Figure 7. Scheduling Pattern.

The Scheduler

The scheduler is responsible for feeding the small-size multiplier with the proper suboperands. The design of the scheduler and the merger are related and defined by the same pattern. In Figure 6, the numbers in each register present the timing point (as of clock cycles) when the datum is valid for the first time in that register assuming zero latency for SAs. For 1^{st} stage, the data in the registers will be updated every 4 clock cycles. For the two middle stages, the values for the update period are 16 and 64 clock cycles respectively. For the final stage, it will take 256 clock cycles, which is the total number of small-size multiplications, to update the four registers. In the merging step, each element at one certain stage (except the first stage) requires four elements from the previous stage. As shown in Figure 2, the four intermediate results at the

Algorithm 2: Pseudo Code for Generating the Schedule

```
for i3, j3 = 0 to 8 step 8 do
    for i2, j2 = 0 to 4 step 4 do
        for i1, j1 = 0 to 2 step 2 do
            for i0, j0 = 0 to 1 step 1 do
                an = i0 + i1 + i2 + i3;
                bn = j0 + j1 + j2 + j3;
```

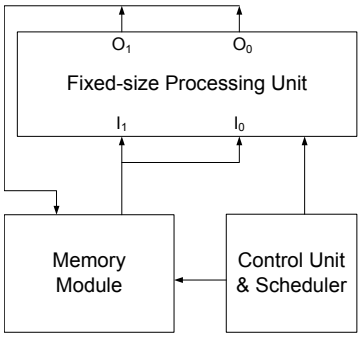

Figure 8. The Architecture for Processing Transformation Operations.

previous stage must relate to each other following one particular pattern. The merging step has no knowledge about the relationships among the data in one quadruplet of registers. The scheduler is responsible for fetching the proper suboperands to feed the multiplier, which in turn generates the results in an appropriate order. In Figure 7, the fetching pattern is drawn in a graphic way. The gray cells show which pairs of suboperands are fetched to be fed to the small-size multiplier. The pseudo code of the scheduler implementation is given as Algorithm 2. Note that a_n and b_n are the addresses of suboperands for the first input and the second input of the small-size multiplier respectively.

3.3. Transformation Operations

Transformation operations are those operations whose number of input operands is equal to the number of outputs. We will consider these operations as a case study for the second category of large-operand operations, namely large-number-of-operand functions. In other words, number of operands is arbitrarily large while the precision of operands is relatively small. More specifically, the intrinsic differences as well as the implementation differences can be summarized as follows:

1. The operands in the second category of operations consist of multiple items, whose length is fixed. However, the quantity of these items can be very large, e.g., more than 2^{10}. The computation applied to the operands does not change the quantity of items; instead, it transforms the operands.

2. In the first category of operations, the partial results are stored in a different memory bank from that used for storing the original operands. For the second category of operations, the processed items are written back to the same memory locations, as shown in Figure 8 in which the I/O size of the processing unit is assumed to be two operands. Two items from the memory module are fed into the processing unit and after a certain latency the two processed items emerge at the output ports and are written back to the memory module. If the memory module has two ports for reading and two ports for writing and the processing unit is fully pipelined, the whole process can operate in a pipelined fashion. If the memory module only has one port for both reading and writing, the performance degrades by a constant factor.

3. The third step in Figure 1 is not necessary for the architecture of the second category of operations. Instead, the Control Unit in Figure 8 will fetch the items in the memory module following a schedule, $Q =< (a_1, b_1), (a_2, b_2), \cdots, (a_q, b_q) >$, to process the operand.

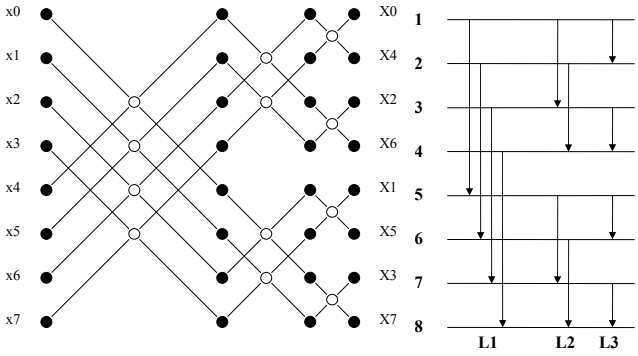

Figure 9. Line Representation of 8-point FFT.

3.3.1. Generating the Schedule

In order to process the operand correctly and efficiently, a proper schedule $Q =< (a_1, b_1), (a_2, b_2), \cdots, (a_q, b_q) >$ needs to be generated. The schedule specifies, from left to right, the pairs of items that are supplied as inputs to the processing unit in a pipelined fashion. Thus, the pair (a_1, b_1) is supplied as the first, followed by the pair (a_2, b_2), and so on.

Olariu *et al.* in [13] discuss the sorting problem, which matches our definition for the second category

of operations. Following the approach in [13], if the processing unit in Figure 8 is denoted by $\mathcal{T}$, the depth of $\mathcal{T}$ can be denoted by *D($\mathcal{T}$)*, which is the number of pipeline stages in $\mathcal{T}$. Let $\mathcal{S}$ be a network of I/O of size *N*, which is the amount of items (suboperands) in the original operand. The network $\mathcal{S}$ implies the order of computation applied to the operand. We show the line representation of $\mathcal{S}$ in Figure 9. Specifically, there are *N* horizontal lines, each labeled by an integer *i*. The left and right endpoints of the line labeled *i* represent, respectively, the i^{th} input and the i^{th} output of the network $\mathcal{S}$. A directed vertical segment originating at line *i* and ending at line *j* represents the pair of items fed into the processing unit $\mathcal{T}$ simultaneously. The schedule *Q* is generated from the line representation of $\mathcal{S}$ following the *greedy algorithm* as proposed in [13]. Let C_1 be an arbitrary FIFO queue of pairs of items at level L_1. We will explain how to obtain the FIFO queue C_{i+1} of pairs of items in level L_{i+1}. First, set C_{i+1} to empty. Second, scan the pair queue C_i in order and, for each pair in C_i, mark its two items. As soon as the two items of a pair c are marked, enqueue c into C_{i+1}. This process is continued, as described, until all queues C_j, such that $1 \leq j \leq D(\mathcal{S})$, have been constructed. Finally, the queues C_j are concatenated in order to obtain a sequence C of pairs such that C_i precedes C_{i+1}. Figure 9 shows the line representation of 8-point FFT. The results of this process is the following schedule:

$$\begin{aligned} Q = <&(1,5),(2,6),(3,7),(4,8),\\ &(1,3),(2,4),(5,7),(6,8),\\ &(1,2),(3,4),(5,6),(7,8)> . \end{aligned}$$

The schedule satisfies the following condition that is necessary to guarantee the correctness of the scheme:

> *No same item should appear more than once in any subsequence Q′ of length D($\mathcal{T}$) of S, such that* $Q' = (a_i, b_i), (a_{i+1}, b_{i+1}), \cdots, (a_{i+D(\mathcal{T})-1}, b_{i+D(\mathcal{T})-1})$

The proof for the above condition is based on the Theorem 2 from [13]. In [13] Theorem 2 states:

> *Let p be the I/O size of the processing unit. If* $N > 2pD(\mathcal{T})$*, then the schedule obtained from the augmented network of* $\mathcal{S}$ *by the greedy algorithm can be used to correctly process the N items.*

3.3.2. Large-number-of-operand FFT

The primitive operation of Fast Fourier Transform (FFT) is the butterfly operation, shown in Figure 10. The butterfly operation involves two points and one *twiddle factor*. After the operation, the results are written back to update the value of these two points. Figure 9 shows an 8-point FFT

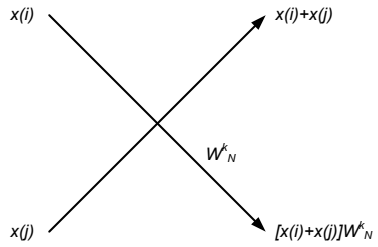

Figure 10. Butterfly Operation.

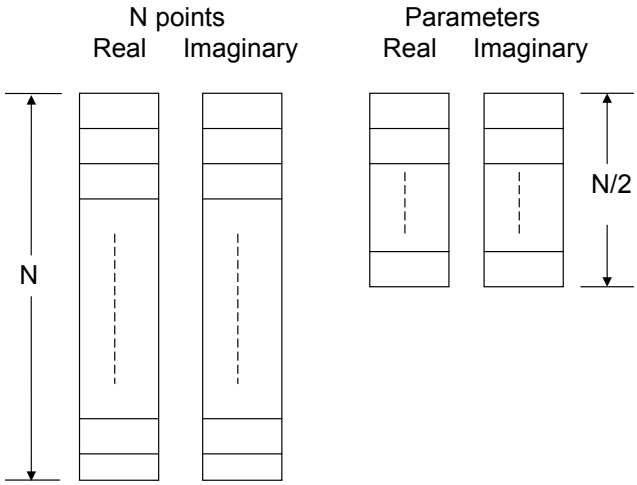

Figure 11. Store N Points and Parameters in Memory.

following Sande-Tukey algorithm. The steps for generating the fetching pattern are shown in Figure 9 and described in Section 3.3.1. The remaining part of this section discusses the mechanism to construct the architecture that can handle large-size FFT using one fully pipelined processing unit, which does the butterfly operation and has a latency of *D($\mathcal{T}$)*.

As illustrated in Figure 11, four banks of memory modules are used to store the data of N points and their twiddle factors. The same location in two memory banks is used to save the real and imaginary parts of one complex number respectively, where both parts of a single point can be fetched simultaneously. When operating in pipelined fashion, in step *i*, the data of two points and their twiddle factor are fed into the butterfly unit. At the end of step *i+D($\mathcal{T}$)+1*, the two results emerge at the output port of the butterfly unit and are written back into the same memory location. This process is continued until all the steps have been processed for N-point FFT. For large-size FFT, the number of iterations of processing N points is $\log N$. In this scenario, the control logic adjusts the patterns to accommodate large sizes.

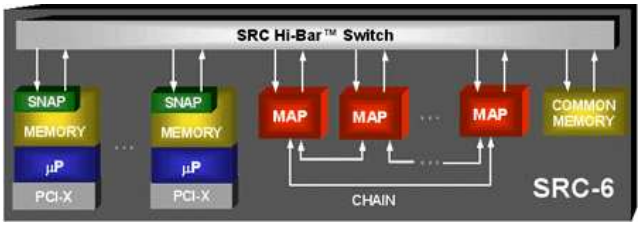

Figure 12. SRC-6 Architecture.

Typically, multiple-point FFTs, e.g., 32-point FFT unit, are required by most applications. However, due to limited hardware resources, it is necessary to utilize smaller-size FFT units, e.g., a k-point FFT unit, to compute the FFT for much larger N points. For the sake of simplicity, k and N are both assumed to be power of 2. The process of N-point FFT consists of $\log N$ layers/stages of computations. The k-point FFT unit can be used to process the last $\log k$ layers/stages of computation. However, for the first ($\log N$ - $\log k$) stages, the k-point unit can not be directly applied without modification. In order to overcome this difficulty, the augmented k-point FFT unit is introduced, which has two modes.

1. When the unit operates in the normal mode, it functions as regular k-point FFT, i.e., it takes the values of k points and applies $\log k$ layers of computations to them.

2. When the unit operates in the augmented mode, it takes the values of k points and $\frac{1}{2}k$ twiddle factors. Only one layer of computation, i.e. the butterfly operations between the first half and second half of the k points, is performed and the result is forwarded to the output port and written back to memory. When processing the N-point FFT, the k-point FFT unit is set to augmented mode from the beginning until the first ($\log N - \log k$) layers of computation are processed, after which it returns to the normal mode. Additionally, the last $\log k$ layers of computation are compressed into one step. To reach the best performance, the data of N points and their twiddle factors should be distributed into the available memory modules as even as possible to maximize the concurrency of memory reading and writing.

4. Implementation and Experimental Work

4.1. Testbed

In order to verify the proposed approach, the previously discussed operations of large-operand multiplication and FFT were implemented on SRC-6 [14]. The SRC-6 architecture consists of Intel microprocessors running the Linux operating system, the MAP processors (i.e., FPGA board), Hi-Bar Switch and Global Common Memory, as shown in Figure 12. In terms of programming entry, the SRC-6 reconfigurable computer is a hybrid-language system. It can accept high level languages, such as C or Fortran as well as HDL language, such as VHDL and Verilog to describe the function running on the MAP processor. In our experiments, the microprocessor prepares the original data (long-size operands) and sends them to the On-Board Memory (OBM), which is accessible by the FPGA directly. For the case of multiplication, the suboperands of each original operand are stored in the OBM. Every clock cycle the scheduler reads one pair of suboperands and feeds them into the small-size multiplier. For the case of FFT, the basic processing unit is a 2-point FFT.

Table 2. Resource Utilization

	Multiplication[a,b]				FFT[b]
	512×512		32,768×32,768		524,288
	App#1	App#2	App#1	App#2	-point
Slices	12,284 (36%)	7,190 (21%)	22,465 (66%)	8,161 (24%)	12,630 (37%)

[a] App#1:Merging on the fly, App#2: Merging in memory.
[b] Vendor-specific service logic is accounted in the above figures.

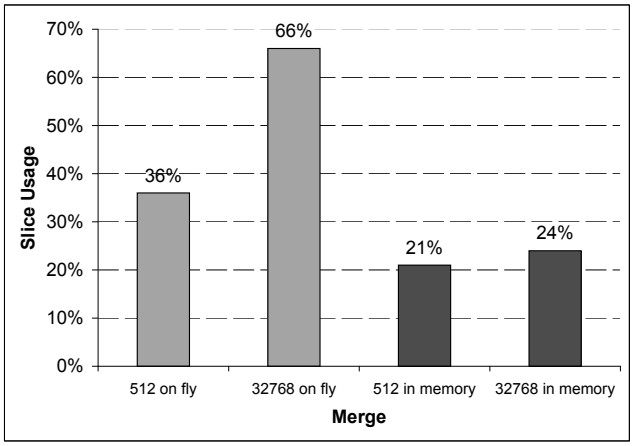

Figure 13. Comparison between two merging approaches.

4.2. Experimental Results

In our experiments, two different multipliers, i.e., 512bit×512bit and 32,768bit×32,768bit, and one 524,288-point FFT are implemented. Table 2 presents

Table 3. Execution Time Comparison of Multiplications between Hardware Implementation and Software Implementation

Operands Length (bits)	SRC-6		μP
	Merge On Fly (μs)	Merge In Memory (μs)	Xeon 2.8GHz (μs)
512	2.56	329.52	35
1,024	10.24	1,318.08	130
2,048	40.96	5,272.32	500
4,096	163.84	21,089.28	2,000
8,192	655.36	84,357.12	8,000
16,384	2,621.44	337428.48	32,800
32,768	10,485.76	1,349,713.92	135,550

the resource utilization on a single FPGA device, which is Xilinx XC2V6000FF1517-4. In the case of multiplication the *merging in memory* approach uses much less resources than the *merge on the fly* approach. As shown in Table 2, although the operand size is increased by 64 times, the increase in resource usage is insignificant. Furthermore, the *merge on the fly* approach can reach the best performance and is easy to scale up. For the case of FFT, the same scheduler can fit arbitrary size input without any change in the hardware architecture.

The results for the multiplication operation show the tradeoff between performance and resource utilization, see Table 2 and Figure 13. Different lengths of operands, ranging from 512-bit to 32,768-bit, are implemented using the two approaches on SRC-6 platform on which the FPGA device runs at clock rate of 100MHz. The performance of software implementation on microprocessor, Intel Xeon 2.8GHz, is given for comparison, see Table 3. The single-thread software implementation of multiplications follows the Karatsuba Algorithm [12] and uses the GMP library [15]. The execution time to perform multiplication at different-length operands is also shown in Table 3 for two platforms. In the approach of *merging on the fly*, the small-size multiplier in the architecture can start the operation for next operands right after the previous one. Because all the intermediate results are stored in the registers and the whole architecture is pipelined, the hardware implementation outperforms the microprocessor by a factor of 10. On the other hand, the microprocessor outperforms the hardware implementation for the other approach, i.e., *merging in memory*, because of its higher clock frequency, and its more mature memory hierarchy with more advanced cache techniques.

It is worth to mention that in Table 3 the data transfer time between main memory of microprocessor and on board memory of FPGA is not taken into account. The impact of data movement on the performance of FPGA can be reduced through a number of techniques. One technique is *Ping-Pong* transfer mode. In this solution the FPGA works on one window of memory and the DMA engine works on another window to overlap the two actions.

5. Conclusions

In this paper, we investigated the problem of large-operand arithmetic by adopting a Divide-and-Conquer approach. We divided this problem into two main sub-categories, namely large-size-operand operations and large-number-of-operand operations. We leveraged the concept of scheduling and merging of the suboperands to perform the overall operation. We selected a representative class of operations for each sub-category. Reduction operations, e.g., large-size-operand multiplication, represented the first category while transformation operations, e.g., large-number-of-operand FFT, represented the second category. In our experiments, two different multipliers, i.e., 512bit×512bit and 32,768bit×32,768bit, and one 524,288-point FFT were implemented to demonstrate the effectiveness of our approach. The SRC-6 reconfigurable computer was used for the experimental verification of the presented concepts.

References

[1] J. Harkins, E. El-Araby, M. Huang, and T. El-Ghazawi, "Performance of sorting algorithms on the SRC 6 reconfigurable computer," in *Proc. IEEE International Conference on Field-Programmable Technology 2005 (ICFPT'05)*, Dec. 2005, pp. 295–296.

[2] E. El-Araby, M. Taher, T. El-Ghazawi, and J. L. Moigne, "Prototyping automatic cloud cover assessment (ACCA) algorithm for remote sensing on-board processing on a reconfigurable computer," in *Proc. IEEE International Conference on Field-Programmable Technology 2005 (ICFPT'05)*, Dec. 2005, pp. 207–214.

[3] C. Shu, K. Gaj, and T. El-Ghazawi, "Low latency elliptic curve cryptography accelerators for NIST curves on binary fields," in *Proc. IEEE International Conference on Field-Programmable Technology 2005 (ICFPT'05)*, Dec. 2005, pp. 309–310.

[4] S. Bajracharya, D. Misra, K. Gaj, and T. El-Ghazawi, "Reconfigurable hardware implementation of mesh routing in number field sieve factorization," in *Proc. Special Purpose Hardware for Attacking Cryptographic Systems, (SHARCS 2005)*, Feb. 2005, pp. 71–81.

[5] T. El-Ghazawi, E. El-Araby, M. Huang, K. Gaj, V. Kindratenko, and D. Buell, "The promise of high-performance reconfigurable computing," *IEEE Computer*, vol. 41, no. 2, pp. 78–85, Feb. 2008.

[6] H. ElGindy and G. Ferizis, "Mapping basic recursive structures to runtime reconfigurable hardware," in *Proc. International Conference on Field Programmable Logic and Applications, 2004 (FPL 2004)*, Aug. 2004, pp. 906–910.

[7] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," in *Proc. the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, Nov. 2006, pp. 113–129.

[8] J. Hormigo, J. Villalba, and E. L. Zapata, "CORDIC processor for variable-precision interval arithmetic," *Journal of VLSI Signal Processing Systems*, vol. 37, no. 1, pp. 21–39, May 2004.

[9] S. Balakrishnan and S. K. Nandy, "Arbitrary precision arithmetic - SIMD style," in *Proc. 11th International Conference on VLSI Design*, Jan. 1998, pp. 128–132.

[10] A. Saha and R. Krishnamurthy, "Design and FPGA implementation of efficient integer arithmetic algorithms," in *Proc. IEEE Southeastcon'93*, Apr. 1993, pp. 8–11.

[11] D. M. Chiarulli, W. G. Rudd, and D. A. Buell, "DRAFT—A dynamically reconfigurable processor for integer arithmetic," in *Proc. 7th International Symposium on Computer Arithmetic*, 1985, pp. 309–317.

[12] A. A. Karatsuba and Y. Ofman, "Multiplication of many-digital numbers by automatic computers," *Doklady Akad. Nauk SSSR*, vol. 145, pp. 293–294, 1962.

[13] S. Olariu, M. C. Pinotti, and S. Zheng, "How to sort N items using a sorting network of fixed I/O size," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 5, pp. 487–499, May 1999.

[14] *SRC Carte^{TM}C Programming Environment v2.2 Guide (SRC-007-18)*, SRC Computers, Inc., Aug. 2006.

[15] The GMP Team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, Free Software Foundation, Inc., Aug. 2007.