Architectural Exploration

Smita Bakshi
Hsiao-Ping Juan
Daniel D. Gajski

March 29, 1993

Technical Report #93-10

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

sbakshi@ics.uci.edu
hjuan@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this report we present a strategy for architectural exploration with the view of reducing the time a designer spends in selecting an implementation that best suits his/her requirements and constraints. We develop an architecture taxonomy and relate designs to different architectural classes. In order to demonstrate the effectiveness of our exploration strategy we have tested it using four examples: a timer system, an FIR filter, an FFT datapath, and a robot kinematics system. We conclude the report by presenting results of our experiments.

Contents

List of Figures

List of Tables

1 Introduction

In this report we attempt to map design descriptions to one or more points in an architectural space consisting of different hardware implementations. We believe that architectures can be categorized and a systematic way of searching the architecture space can be established.

We first develop a taxonomy of architectures and justify it through several "well chosen" examples. Next, we propose a strategy for exploring the architectural space and test its validity with the help of our examples.

We classify architectures on the basis of three factors: the use of standard vs. custom components, the level and degree of parallelism and the direction of slicing and blocking the computation. The input of the exploration strategy is a data flow graph representation of the design. We first determine the critical path and the communication pattern of the design in order to categorize the design. This indicates the extent of data dependence and hence, the inherent parallelism in the design. It also gives us an idea of how we can slice the computation into smaller blocks or units. We then systematically partition the data flow graph into blocks of different sizes, and vary design parameters in order to obtain different datapaths.

Current synthesis environments ([2], [8]) rely heavily on tasks such as transformations (loop unrolling, software retiming etc), scheduling and allocation. The result of the synthesis is largely dependent on the input description. The architectural exploration methodology proposed in this report performs the task of first determining the optimal direction of slicing. It then partitions the data flow graph in the selected direction. We believe this strategy results in a more exhaustive search of the design space and is not so dependent on the input description.

The report is organized in the following manner. Section 2 describes the architecture taxonomy and Section 3 gives an outline of the architecture exploration strategy. A description of the examples is given in Section 4 and our experimental results are presented in Section 5. In Section 6 we propose a memthodology for architectural exploration and discuss the automation of various tasks in the design process.

2 Architecture Classifications

In this section we describe a taxonomy of architectures and also relate different designs with architectural styles. The categories used to classify architectures are as follows:

1. **Standard vs. Custom Designs:** We classify a standard design as one in which we have to mold or "tweak" the input specification to fit an already existing architecture. Standard components are not tuned to the problem unlike custom components which can be "optimally" designed for a given set of requirements. Though standard components are relatively inexpensive and are well documented, they are often not suited for performance critical applications.

   Thus designs may be classified, with respect to customization, as follows:

   (a) Standard Datapath and Control

   (b) Standard Datapath with Custom Control

   (c) Custom Datapath and Custom

   (d) Custom Datapath with Standard Control

2. **Parallel vs. Serial Designs:** Designs can have varying degrees of parallelism at different levels. Consider a design that consists of N independent m-bit ADD operations. Depending on the availability of resources, anywhere from 2 to N additions can be done in parallel. Furthermore, the adder can range from 1 to m bits in length. Therefore, in this design we can introduce parallelism at two levels, the bit level and the word or block level.

   In general, we can vary the following:

   (a) *Level of Parallelism:* Bit level or Block level.

   A block of computation refers to a group of operations that can be executed as one "indivisible" unit. For example, using two multipliers and two adders, a computation can be performed in blocks of 4 operations. This level of parallelism can also be referred to as word-level parallelism. Bit level parallelism, on the other hand, is dependent on the functional units that compose the datapath: they can be bit serial or bit parallel.

   (b) *Degree of Parallelism:* We can achieve varying degrees of parallelism by:

   - varying the number of bits computed in parallel,
   - varying the block size, or
   - varying the number of blocks.

   Selecting a block size is dependent on the constraints and the availability of resources, for example, the number of available memory modules or memory ports in the design.

Once the block size is fixed so are the number of functional units, and to a large extent, the memory ports (or memory modules) and the pipelining of the datapath.

3. **Direction of Slicing and Blocking:** For a given dataflow graph, a block of computation is selected, based not only on the availability of resources and the performance requirements, but also on the communication flow of the computation. The communication flow or the input-output dependence of a design falls in three categories:

   (a) **Point Flow**: The input consists of totally independent sets of data without any notion of causality or sequence. The ith output depends only on the ith input.

   (b) **One-dimensional Flow**: The input consists of a sequence of data values and each output depends on a certain section or window of the input sequence. For example, in a 4-order FIR filter the ith output depends on the $(i-3)rd$, $(i-2)nd$, $(i-1)st$ and the ith input.

   (c) **Two-dimensional Flow**: As before, the input consists of a sequence of data values but the outputs need not depend on a fixed section or window of the inputs. In this system, data flows in more than one dimension (eg. right and down) and hence it may not be possible to determine a direction of communication flow along which the computation may be sliced (e.g. FFT datapath).

   For designs with a one-dimensional flow, the entire computation can be sliced in one of four directions:

   (a) Horizontal

   (b) Vertical

   (c) Main Diagonal

   (d) Reverse Diagonal

   The optimal direction of slicing for a design depends on its direction of data flow. When we compute against the dataflow of the design the partial results that are generated in one "block of computation" cannot be readily used by the next block of computations and have to be stored back to the memory. On the other hand, when we perform the computation in the direction of the dataflow, the partial results of one block of computations can serve as inputs for the next block of computations. We can thus store these partial results in registers instead of in the memory. Since memory accesses are time consuming, slicing in the direction of the dataflow yields more cost-effective designs.

For designs with a two-dimensional flow there is no preferred direction of slicing, and the direction and order of blocking the computation is greatly limited by data dependencies (e.g. dependencies between two consecutive stages of an FFT datapath).

3 Exploration Strategy

This section outlines the design exploration strategy used to implement the four examples. The timer example was described using VHDL code. This was sufficient since it consists of independent sets of computations without any data flow. For the other three examples, our starting point was a data flow graph, since the communication pattern of one-dimensional and two-dimensional designs can be readily recognized using a data flow graph.

The result of our exploration is a set of implementations of the design and an estimate of the cost and performance of the different implementations.

By first analyzing the description of the design, we determined its critical path and communication flow. We classified the design as having a point, one-dimensional or two-dimensional flow. We also determined the direction of flow (if one existed) and the data dependencies in the design. From this analysis, we determined whether or not to slice the design as well as the direction of blocking and slicing.

Next, we extracted the maximum amount of parallelism from the design if no upper bounds on the cost or the number of resource were specified. This was the starting point of the design exploration. For example, the most parallel implementation of a design with say, 256 independent multiplications consists of 256 multipliers working in parallel. Note that the most parallel design may be highly impractical. However, it serves as a good starting point for breaking the design into a series of smaller computations.

If the designer specified an upper bound (as in the example of the robot kinematics system) on the resources to be used (number of functional units, number of memory ports, etc.) then the starting point of the design exploration consisted of a datapath with the maximum number of resources that still respected the resource constraints. We thus started from the most efficient end of the design spectrum and proceeded towards less costly and less efficient designs.

Next, we varied design parameters in a step-wise fashion to obtain implementations of varying costs and performance. The parameters that were varied include:

1. Block size

2. Number of blocks

3. Degree of bit parallelism

4. Degree of pipelining

It is to be noted, that in some cases it was not possible to vary the degree of pipelining and bit-parallelism independently of the block size.

Finally, using suitable assumptions of the area and delay of resources we estimated the cost and performance of the designs. In all the examples the cost has been approximated by a gate count of the datapath and the performance has been estimated by the total time taken in *ns* for the entire computation.

It is also to be noted, that our design exploration is by no means complete since we have focused on a small section of the design spectrum, namely designs with custom datapaths. In the concluding section, this strategy is extended to include designs with standard datapaths.

4 Description of Examples

In order to demonstrate the effectiveness of our architecture taxonomy and exploration strategy we used them explore the design space of four examples. In this section we give a brief description of the examples.

1. *N independent modulo-m Timer System:* The Timer System consists of N-independent counters where each counter has a separate *load* signal, *enable* signal, m-bit *limit* signal and a *time_out* signal. (The output, thus, consists of the N *time_out* signals). The *ith* output depends only on the *ith* input. The data flow of such a system can be classified as a *point* flow.

2. *Finite Impulse Response (FIR) Filter:* The FIR filter response ([5]) is given by the following equation: $y[i] = \sum_{k=0}^{M-1} x[i-k]b[k]$ where x[0..N] is the input stream, y[0..N] is the output stream and b[0..M-1] is the array of filter coefficients (M is the order of the filter and N is the number of inputs). In this system, an output signal, y[i], depends on M inputs, x[i], x[i-1], x[i-2],....,x[i-M]. The dataflow of this system can be classified as having a *one-dimensional* flow since an output depends on a window of input values (in this case, M previous input values).

3. *Fast Fourier Transform (FFT) Datapath:* An N-point FFT consists of N inputs labeled x_0 to x_{N-1}, N outputs labeled y_0 to y_{N-1} and $\lg N$ stages of computations. Each stage of computation requires $\frac{N}{2}$ butterflies where a butterfly consists of one

addition, one subtraction and one multiplication. Furthermore, two consecutive stages are connected to each other via a shuffle network.

The data flow in this system can be categorized as *two-dimensional* since an output depends on all the inputs; thus, there is no direction of "communication flow" as in the example of the FIR filter.

4. *Robot Kinematics:*

   This example evaluates the Jacobian of an open kinematic chain, whose representation is based on the *product-of-exponentials* (POE) formula. Details of this representation can be found in [7].

   Informally, the Jacobian of a robot is the linear transformation relating joint rates to end-effector rates. The input to the algorithm consists of a set of n scalar joint variables corresponding to the n joints of the robot and a set of n 4×4 matrices, A_1 to A_n, characterizing the joints ([7]). The matrices are of a "special form" and they can also be represented as 6-element vectors. The result of the Jacobian computation is a set of n 4×4 matrices, J_1 to J_n, which are similar in form to the A_i matrices and can also be represented as 6-element vectors.

   We now outline the steps of the algorithm for determining the Jacobian, J, of a mechanism. Let G denote a 4×4 identity matrix. The first element of the set of Jacobian matrices, J_1, is simply equal to the elements of A_1. The remaining $n-1$ matrices, J_2 to J_n, are evaluated as follows:

$$
\begin{aligned}
&\textit{For } (i = 2; i \leq n; i++) \textit{ do} \\
&\qquad G := G \exp^{A_{i-1} x_{i-1}}; \\
&\qquad J_i := G A_i G^{-1};
\end{aligned}
$$

   The matrix exponentials can be computed via a table lookup or directly from a closed-form expression for the exponential. We have computed the exponentials by a table lookup.

   This system can be categorized as *one-dimensional* since the direction of data flow from the *ith* input to the *ith* output is restricted to one dimension.

```
TIMER : process
        variable counter : array(7 downto 0) of integer;
        variable i : integer;
begin
        Wait until CLOCK = '1' and not CLOCK'stable;
        For i in 7 downto 0 loop
                If LOAD(i) = '1' and LOAD(i)'active then
                        counter(i) := LIMIT(i);
                elsif ENABLE(i) = '1' then
                        If counter(i) /= 0 then
                                counter(i) := counter(i) - 1;
                        else TIME_OUT(i) <= '1'; end if;
                end if;
        end for;
end process;
```

Figure 1: VHDL Description of Timer System

5 Experiments

In this section we describe the different implementations of the four examples given in the previous section. We also give the estimated cost and performance of the implementations and establish why some designs are more cost effective than others.

5.1 N independent Modulo-m Timers

The timer system consists of N inputs and N outputs where there is a one to one correspondence between the inputs and the outputs, that is, the *ith* output depends only on the *ith* input and on no other input. The design essentially consists of N totally independent sets of computations which can be performed in parallel or in series. Figure 1 gives a portion of a VHDL sequential description of the timer system.

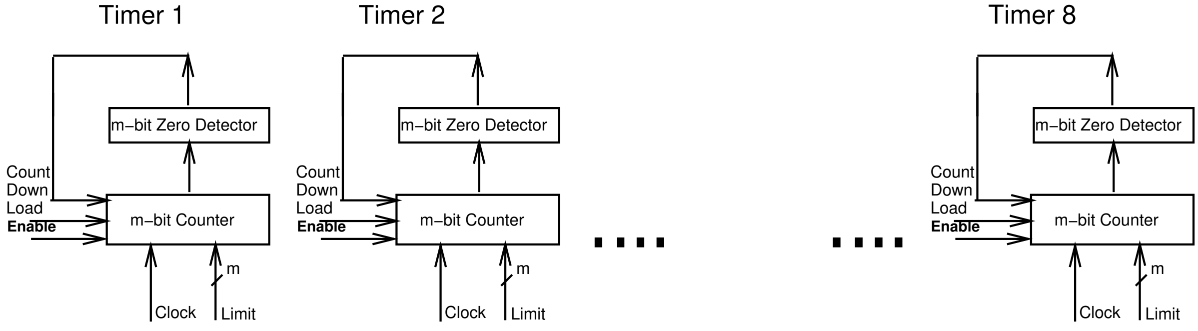


Figure 2: 8 parallel 16-bit Counters

We recognize that the timer system has a point flow and hence we cannot consider the direction of slicing as a possible variant of the implementations. We first find the most parallel implementation of the design and then we consider blocking the design into series of one or more computations (hybrid designs). The final implementation computes the N operations completely serially (i.e. in N time steps). Here the blocking is at a word level.

Next, we briefly describe various implementations of a system with 8 modulo-16 timers.

1. *Design 1:* This design (Figure 2) consists of 8 counters and 8 16-input NOR gates which act as zero detectors. This is the most parallel implementation of the system.

2. *Design 2:* This design is identical to the first design except that we now reduce the number of counters and zero detectors to 4 (instead of 8). Since there is more serialism in this design and the number of resources is smaller than the size of the problem it becomes necessary to introduce more storage elements so that the computations can be done in blocks of 4 each. We thus introduce an 8×16 memory in this design. We can generalize and say that serialism in a design necessitates the use of storage.

3. *Design 3:* Finally we have just one counter and zero detector in our design. This is a completely serial design.

4. *Design 4:* The previous 3 designs were bit parallel designs and now we consider bit serial designs. Design 4 consists of 8 independent "computation units" of a shift register, a 1-bit adder and a zero detector. This design can be classified as a word parallel, bit serial design.

5. *Design 5:* This design is similar to Design 4 except that it contains 4 instead of 8 "computation units" and a memory unit.

6. *Design 6:* In this design we reduce the number of computation units to one. Hence, the design is a bit serial word serial design.

7. *Design 7:* This design consists of 8 computation units where each computation unit contains a 2^{16} shift register, a 16×2^{16} decoder, 2^{16} AND gates and a 2^{16} bit zero detector. The design uses unary encoding and thus requires a 2^{16} bit register to encode all possible states of a 16-bit counter.

8. *Design 8:* This design is similar to Design 7 except that it contains 4 instead of 8 "computation units" and a memory unit.

Design #	Resources in Design	Total Cost in gates	Total Time for one operation (ns)
1	8 counters, 8 zero detectors	3064	60
2	4 counters, 4 zero detectors	1344	520
3	1 counters, 1 zero detectors	695	880
4	8 shift reg, 8 zero detectors, 8 one-bit adder	1512	470
5	4 shift reg, 4 zero detectors, 4 one-bit adders	1068	1340
6	1 shift reg, 1 zero detectors, 1 one-bit adder	501	4100

Table 1: Evaluation of Timer Implementations

9. *Design 9:* In this design we reduce the number of unary encoded computation units to one.

10. *Design 10:* So far we have considered designs with custom datapaths. We now evaluate designs with standard datapaths. Design 10 consists of the Intel 8086 micro-processor connected to a memory and an I/O interface via a data and address bus. This design is also serial at the word level.

11. *Design 11:* In this design the computations are done using 4, 4-bit ALUs (AMD 2901). This design is also word serial and requires a 8×16 memory and a control unit.

12. *Design 12:* In this design we utilize two 8086 Intel micro processor chips with a dual ported memory. This design can be classified as a hybrid design.

We evaluated the cost (in terms of # of gates) and the performance (in terms of the number of *ns* required to compute the complete loop in Figure 1) of the designs (Table 1). We determined though word parallelism increases the cost it also brings about a considerable increase in the performance. Furthermore, we realized that bit serialism reduces the cost but also brings about a more than proportionate increase in the total computation time.

Table 1 lists some of the design evaluations and Figure 3 gives the cost vs. delay of the designs. From the graph we conclude that Design #1 is nearly twice the cost of Design #2 but its speed is about eight times greater than Design # 2. This is because in Design #1 we have as many counters as there are timers and hence, there is no need for a memory. In Design #2 we reduce the number of components by half and we introduce a memory unit. Due to extra memory accesses, the computation time now increases by about eight times. It is to be noted that Design #3 has the lowest cost-performance product.

Furthermore, we see that Design #4 is about half the cost of Design #1 and about six times as slow. This is because it computes the bits of the timer serially using a one-bit

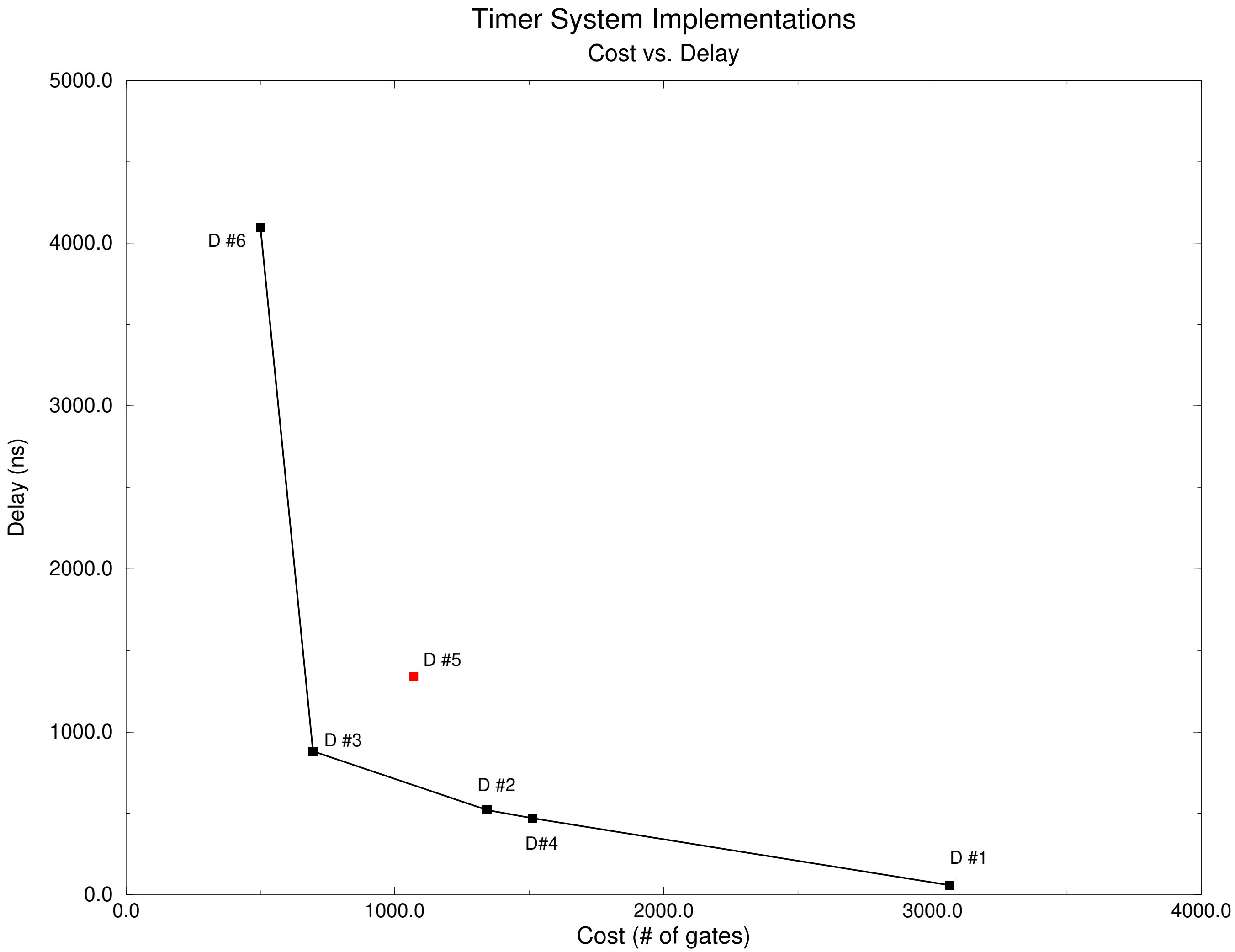


Figure 3: Cost vs. Performance of Timer Circuits

adder rather than in parallel using a counter. For the timer system we can conclude that bit parallel designs are more cost effective than bit serial designs, where the term "cost-effective" can be defined as the percentage increase in performance due to a unit increase in the cost (in our case, the gate count) of the design.

5.2 FIR Filter

We explain the implementations of the FIR filter by taking a specific example of an 8-order, 16-input FIR filter. It consists of 16 inputs labeled x_0 to x_{15}, 16 outputs labeled $y_0$to y_{15} and 8 filter coefficients labeled b_0 to b_7. Each output consists of a sum of products involving the 8 coefficients and eight previous input values. Figure 4 shows that communication follows a horizontal path from the input to the output and that input values flow along the main diagonal.

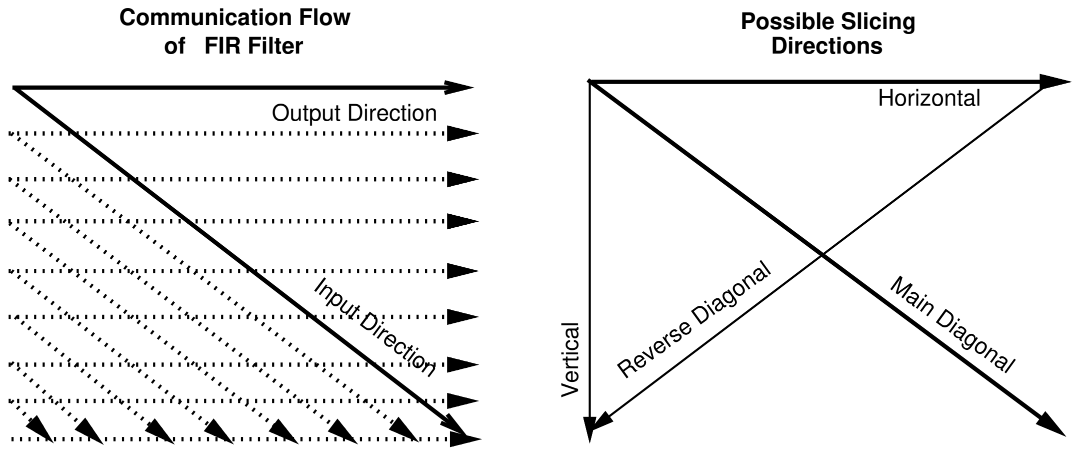


Figure 4: FIR Communication Flow and Slicing Directions

The FIR filter has a one-dimensional flow; this makes it possible to slice the computation in different directions. We considered four slicing directions: horizontal, vertical, main diagonal and reverse diagonal (Figure 4). For each of the slicing directions we first implemented the most parallel design and then blocked the computation in different sizes and different directions. This is indicated in Figures 5, 6, 7 and 8. Thus the variable factors in the implementations are: the direction of slicing, the block size and the number of blocks working concurrently. We also varied the direction of blocking in some implementations.

Table 2 gives the resources and an estimate of the cost and performance of a number of implementations for an 8-order FIR filter. Given a number of resources, we can implement a design to perform the computation in either of the four directions. We found that designs with horizontal slicing had the fewest number of memory accesses, and consequently these designs gave the highest performance. The reverse diagonal implementations, on the other hand, were the least cost effective.

In Figure 9 we show the cost vs. delay of a number of the implementations for four different slicing directions. For all the implementations we see that horizontal slicing gives the best performance, reverse diagonal slicing gives consistently bad results and vertical and main diagonal slicing yield intermediate (and similar) results. Figure 10 gives the effect of increasing resources on the cost, delay and number of memory accesses required. (Since vertical and main-diagonal slicing techniques are very similar we have shown the results of only one of them.)

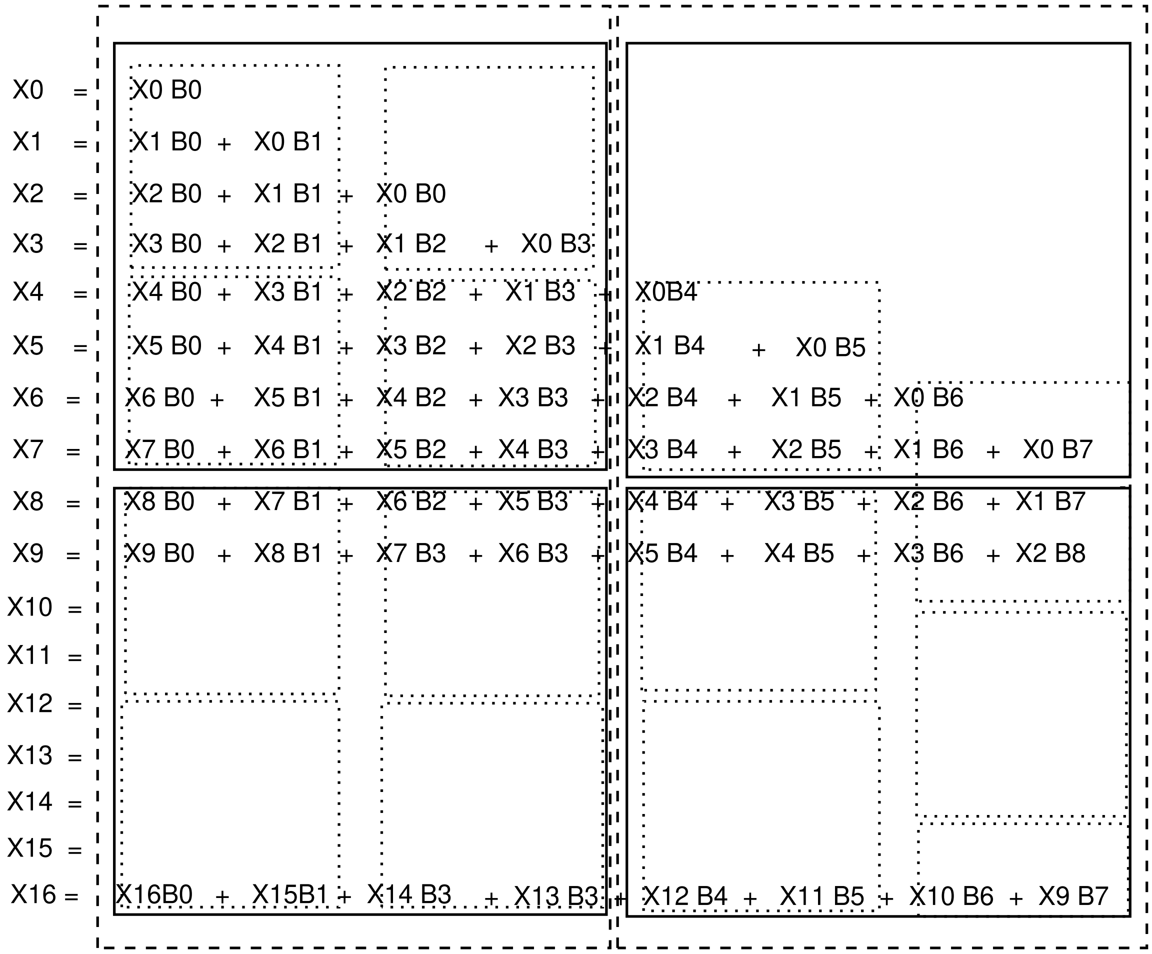


Figure 5: Different Block Sizes of FIR Filter Computation

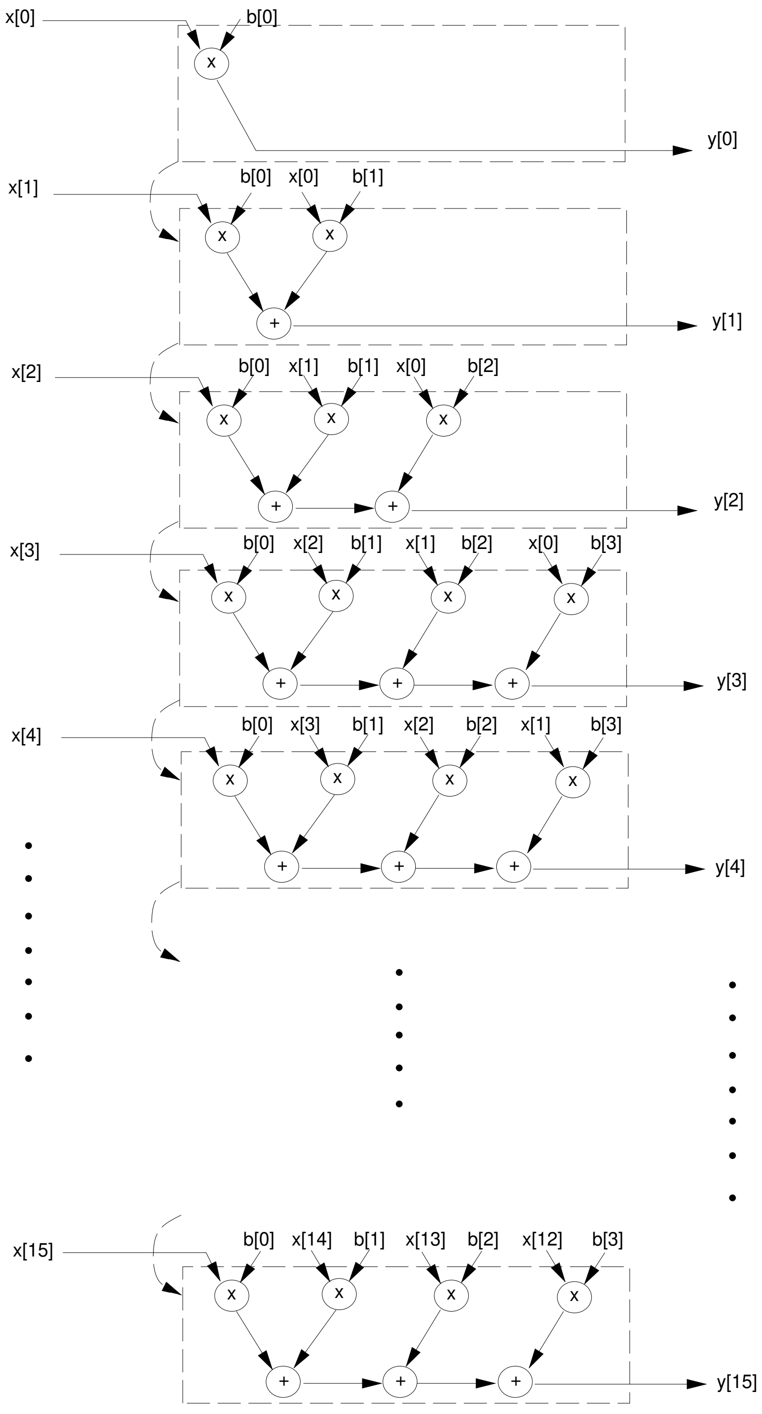


Figure 6: Horizontal Slicing of FIR Filter Computation

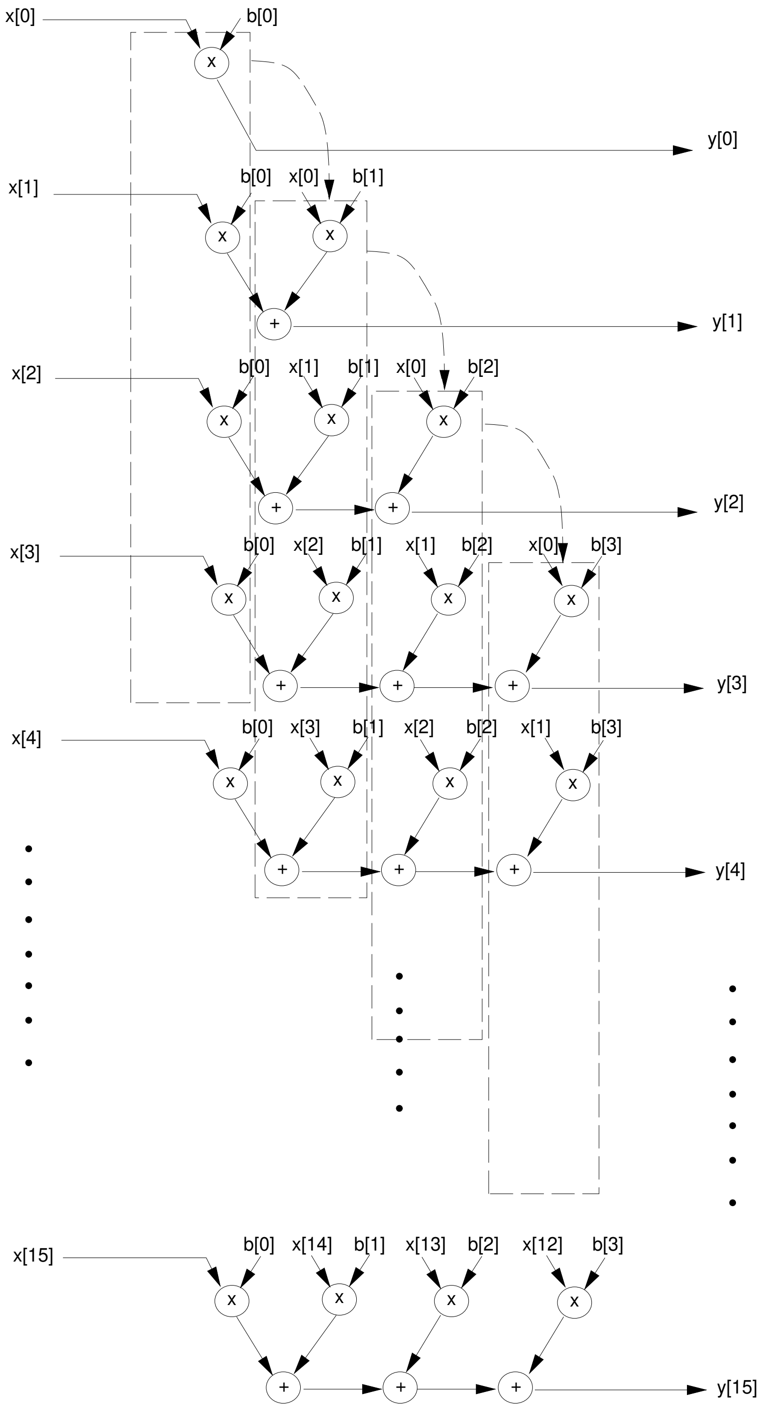
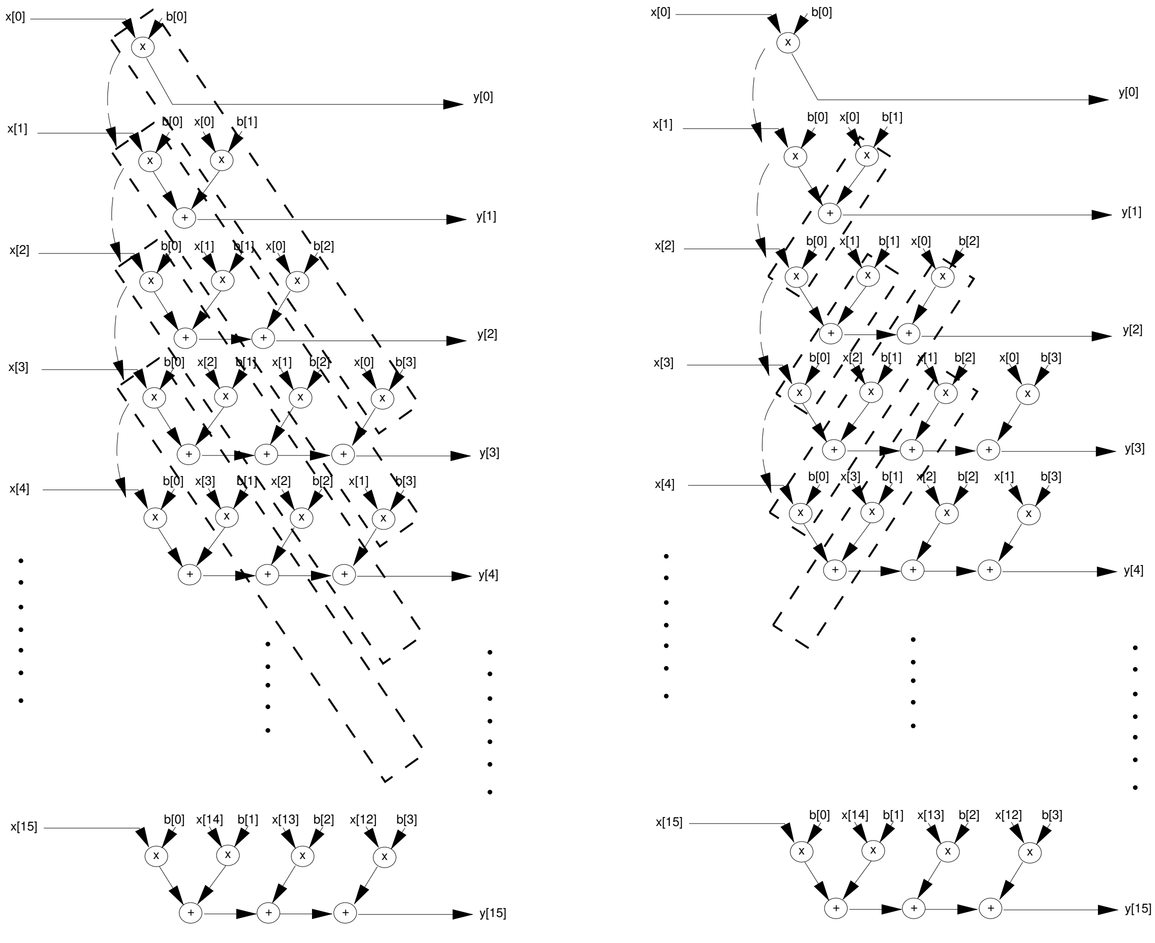


Figure 7: Vertical Slicing of FIR Filter



(a) (b)

Figure 8: Main and Reverse Diagonal Slicing of FIR Filter

Slicing Direction	# of Mult.	# of Adders	# of Reg	# of Memory Ports	# of memory accesses	Total Cost # of gates	Total Computation Time (ns)
Horizontal	1	1	3	1	200	8395	34200
Horizontal	2	2	4	1	148	9918	25200
Horizontal	2	2	3	2	200	11114	19700
Horizontal	4	4	8	1	76	13220	14600
Horizontal	8	7	14	1	40	19153	8000
Horizontal	8	8	14	2	80	20892	8800
Vertical	1	1	2	1	292	8267	44800
Vertical	2	2	4	2	292	11242	22400
Vertical	4	4	7	1	84	13604	16800
Vertical	8	7	8	1	60	20945	8400
Main Diag.	1	1	3	1	280	8395	44050
Main Diag.	2	2	4	1	148	9918	26700
Main Diag.	4	4	11	1	76	13604	16000
Main Diag.	8	7	24	1	40	20433	9700
Main Diag.	8	8	20	2	85	21660	10900
Rev. Diag.	1	1	3	1	400	8395	55000
Rev. Diag.	2	2	4	2	400	11242	27800
Rev. Diag.	4	4	8	4	400	15204	14500

Table 2: Evaluation of FIR Filter Implementations

From this we conclude that slicing a computation in the direction of its flow results in a more cost effective implementation since in one "block" a fewer number of memory accesses are required. On the other hand, if we slice the computation against its natural flow (along the reverse diagonal for example) a lot of the partial results that are generated in each block of computation have to be stored back to the memory rather than be combined to yield an output or a fewer number of larger partial products. This consequently, reduces the performance of the design. Figure 10 also shows that there is a direct relationship between the number of memory accesses in the design and the total computation time. Thus, the larger the number of memory accesses the larger will be the total execution time.

5.3 FFT Datapath

We explain the implementations of the FFT datapath by taking the example of a 64-point FFT. This is shown in Figure 11. As we mentioned previously, an output of the FFT depends on all the inputs and the communication flow is two dimensional rather than one dimensional. Hence, we cannot slice the computation in different directions as we could for the FIR filter. Also, there is a lot of data dependence between two consecutive stages of computations. (We define node j in the dataflow graph to be dependent on node i if the result of node i is used as an input for node j. Thus node j can be computed only after

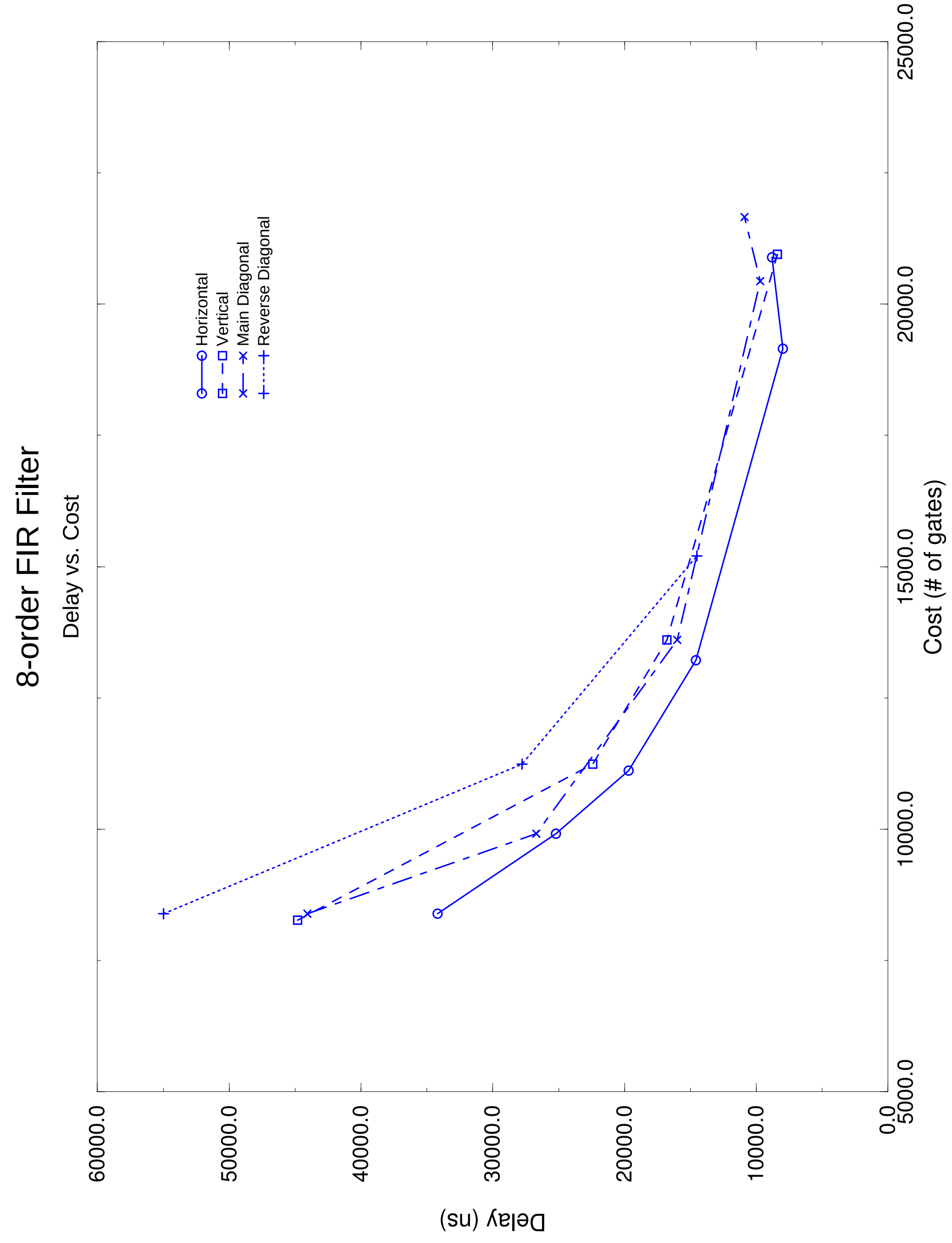


Figure 9: Cost vs. Delay of 8-order FIR Filter Implementations

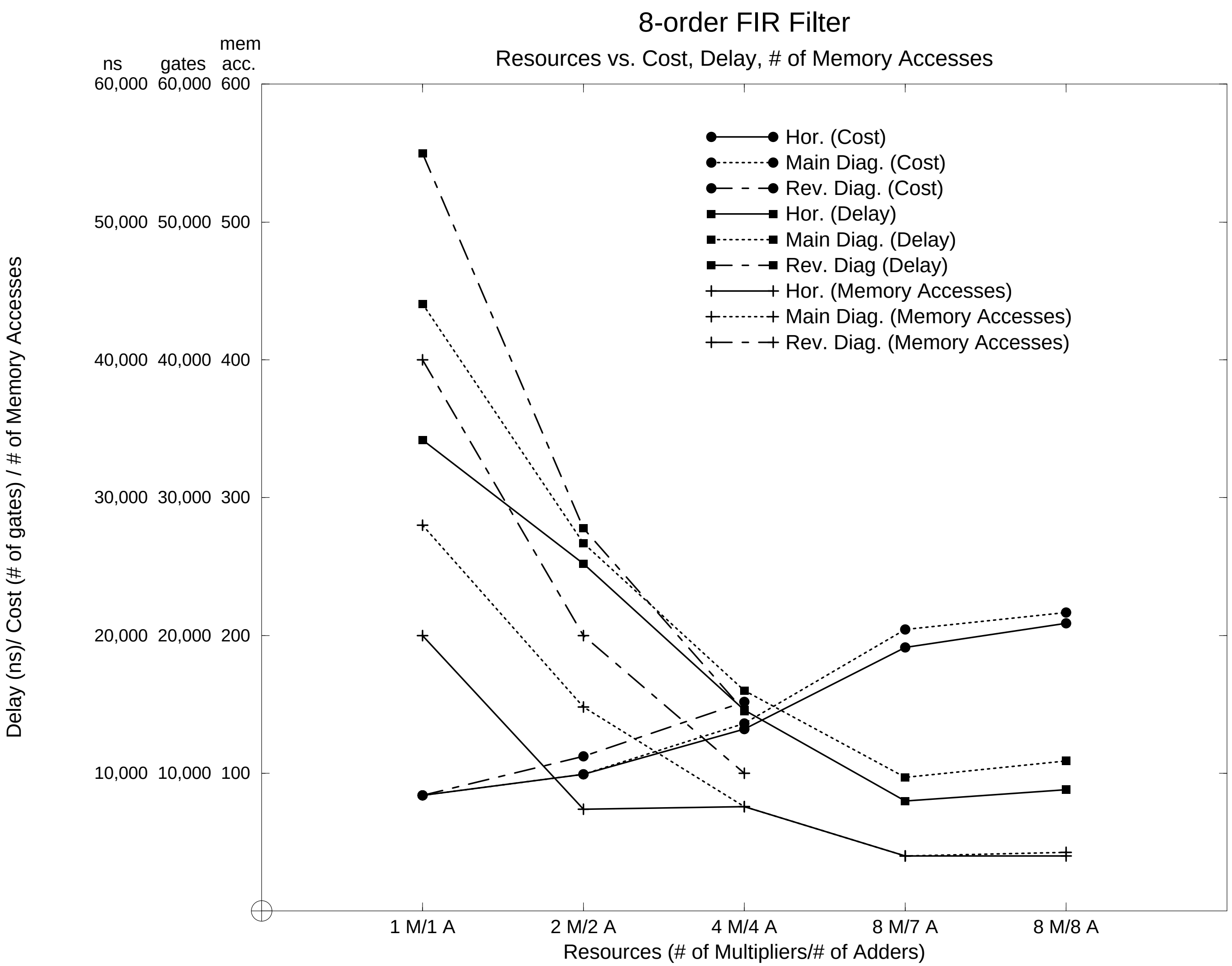


Figure 10: Resources vs. Cost, Delay and # of Memory Accesses of -order FIR Filter Implementations

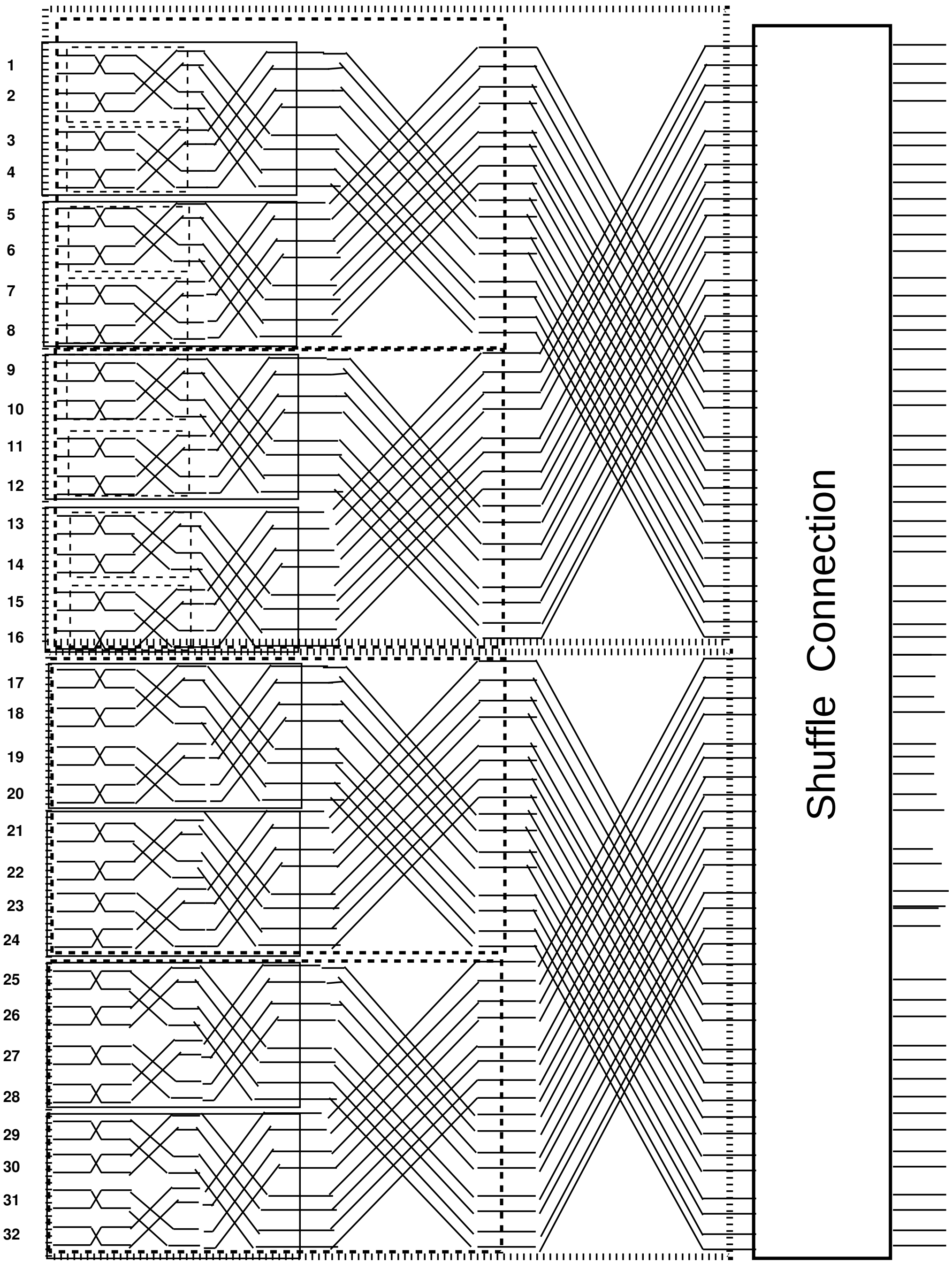


Figure 11: 64-point FFT

Block Size	# of Blocks	# of Memory Ports	# of Multipliers	# of Adders	# of memory accesses	# of Gates	Total Computation Time	Utilization of FFT Chip (%)
4	1	1	4	8	384	27200	6720	100
4	1	4	4	8	96	42200	3840	100
4	4	4	16	32	96	63800	1600	100
4	4	16	16	32	24	108800	960	100
4	8	16	32	64	24	137600	840	100
8	1	1	12	24	256	41600	4000	100
8	1	4	12	24	64	56000	2080	100
8	1	8	12	24	32	81600	1760	100
8	2	8	24	48	32	103200	960	100
8	2	16	24	48	16	123200	880	100
8	4	16	48	96	16	166400	480	100
16	1	4	32	64	160	92600	4000	30
16	1	16	32	64	40	137600	2800	30
16/4	1/1	4	36	72	64	99800	2080	46
16/4	1/4	16	48	96	16	166400	880	54
16	2	16	64	128	40	195200	1500	30
32	1	4	80	160	544	179000	10000	7
32	1	16	80	160	136	224000	5820	7
64	1	16	192	384	8	368000	260	100
64	1	64	192	384	2	448000	200	100

Table 3: Evaluation of FFT Implementations

node i has been computed). In the FFT, the results of one stage of butterflies are required by the next stage. Hence, we do not have a lot of flexibility in the direction or order of evaluation of the butterflies.

The factors that vary across different implementations are:

1. the block size (i.e the number of butterflies that can be evaluated by one chip),

2. the number of such blocks (or chips), and

3. the number of memory modules or memory ports in the design.

For the 64-pt FFT the most parallel implementation is obtained by doing the computation in one block of size 64. This requires the least number of memory accesses to store intermediate results, and depending on the number of ports on the memory, the input and output data can be loaded in one or more batches.

Next, we serialize the design by using block sizes of 32, 16, 8 and 4. We also vary the number of blocks and, to a limited extent, the number of memory ports. For each of these implementations we estimate the cost, performance and the utilization of the blocks. This is given in Table 3 and illustrated in Figure 13.

Figure 12 gives the estimated cost vs. delay of the different implementations. The designs in which the block size is 16 or 32 have a low utilization compared to designs with

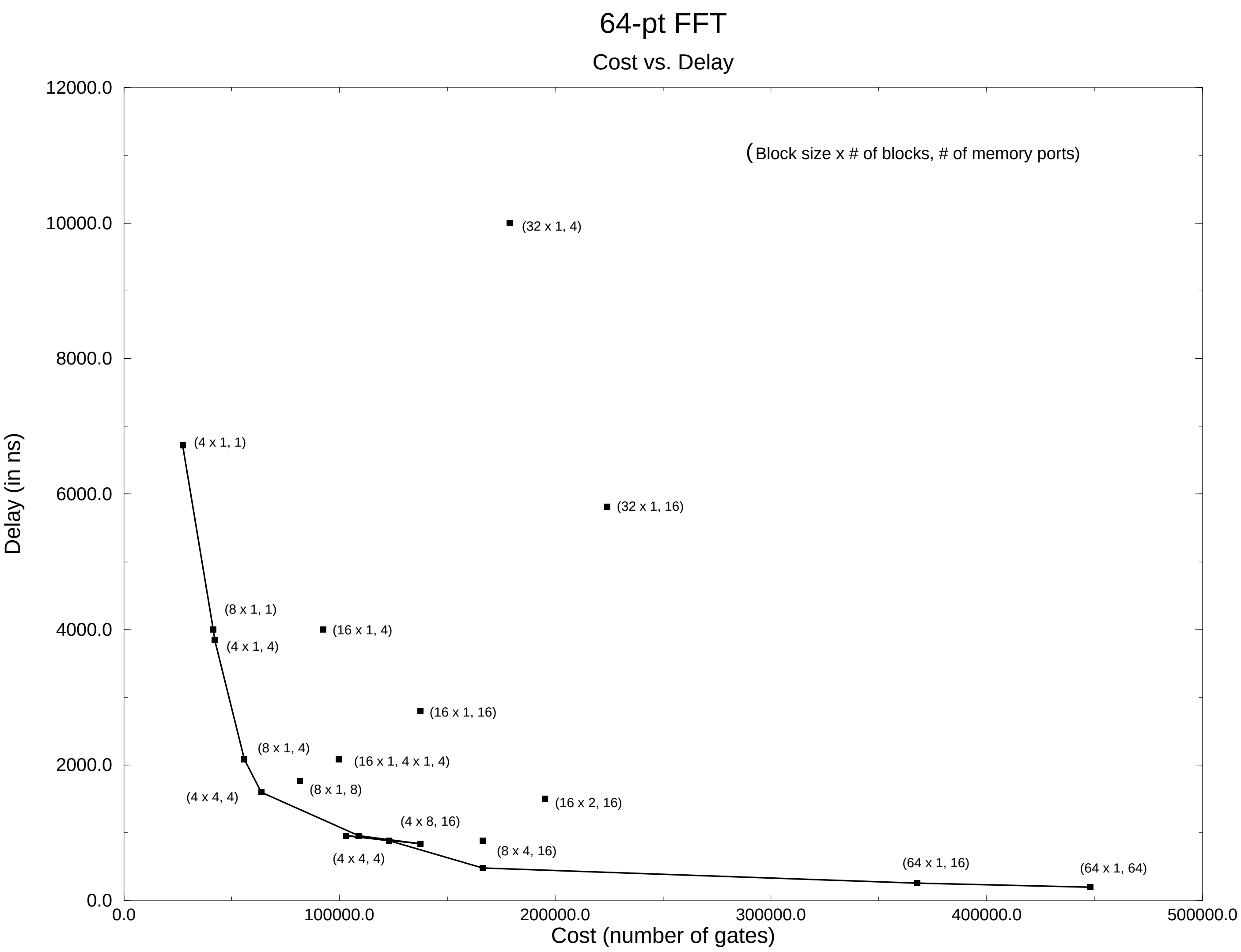


Figure 12: Cost vs. Performance of 64-point FFT Implementation

block sizes of 4, 8 or 64. This consequently increases the total computation time of such designs. The reason for the low performance is that the 64-point FFT does not map well to blocks of size 16 or 32 and a lot of redundant butterflies are required in order to complete the FFT using these blocks. For example, when we use a chip of size 32, the first 5 stages of the butterflies are computed without any redundant computations but evaluation of the sixth stage results in about 90% extra butterflies. This reduces the effective utilization of the chip and consequently its performance. (In general, an N-point FFT can be computed with an M-point FFT chip without any redundant butterflies (100% utilization) provided $\lg N$ is a multiple of $\lg M$.)

5.4 Robot Kinematics

We implemented the datapath for the computation of the Jacobian of a robot with 8 joints. A block diagram of the entire computation is shown in Figure 14. It is to be noted that all computations involve floating point numbers. Block 1 consists of the multiplication of a 6-element vector (A_i matrix) with a scalar (x_i). Block 2 is a table lookup and Block 3 consists of the multiplication of two 4×4 matrices. Block 4 is similar to Block 3, except that it has a fewer number of computations. It is easy to see that the critical path lies through the Block 3 computations since the output of the ith computation serves as an input for the $(i+1)th$ computation.

Figures 15, 16 and 17 give the signal flow graph of Blocks 1, 3 and 4 respectively. In Figure 16, *R11, R12* etc. represent elements of the matrix obtained from the table lookup in Block 2. (The signal flow graphs were derived manually from a high-level description (in C-language) of the Jacobian computation.)

The table of exponentials for Block 2 is estimated to have approximately $n \times 10^6$ 16-bit entries. For a robot with 8 joints this amounts to approximately 20 Mbits of ROM. In our designs we have assumed that the ROM is off-chip and thus, Block 2 requires 12 off-chip memory accesses.

In this example, our design exploration was limited due to resource constraints. We assumed that the datapath was to be implemented on a single chip using gate array technology [9] and thus, we had an upper bound on the total gate count of the implementations.

We first determined the direction of dataflow in Blocks 1, 3 and 4. This is indicated in the figures. We then partitioned (or blocked) each of the signal flow graphs to obtain the "largest" block size without violating the resource constraints. (This corresponds to obtaining the "most parallel" implementation for each of the blocks). With this information

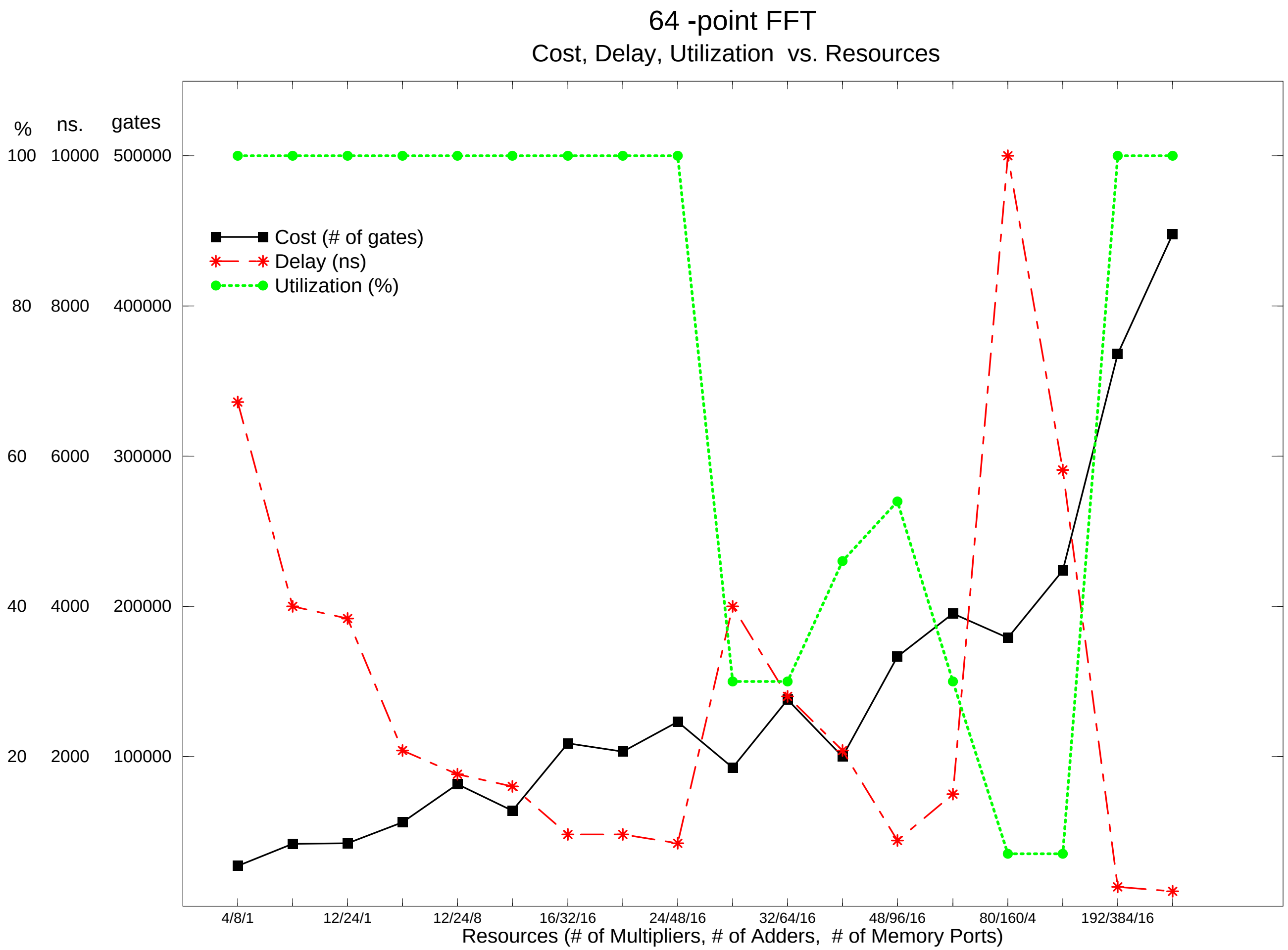


Figure 13: Resources vs. Cost, Delay and Utilization of FFT Implementations

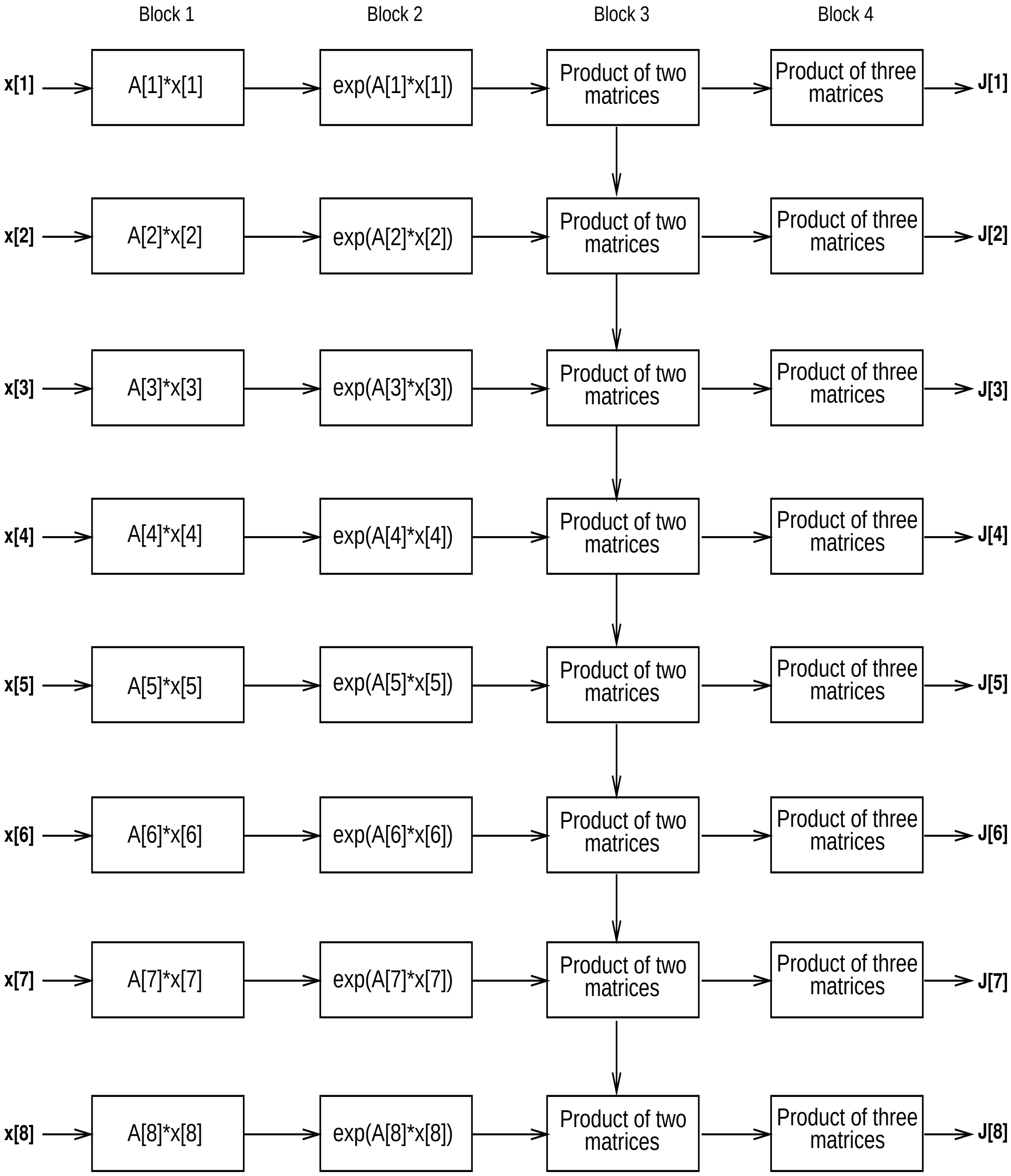


Figure 14: A Block Diagram of the Jacobian Computation

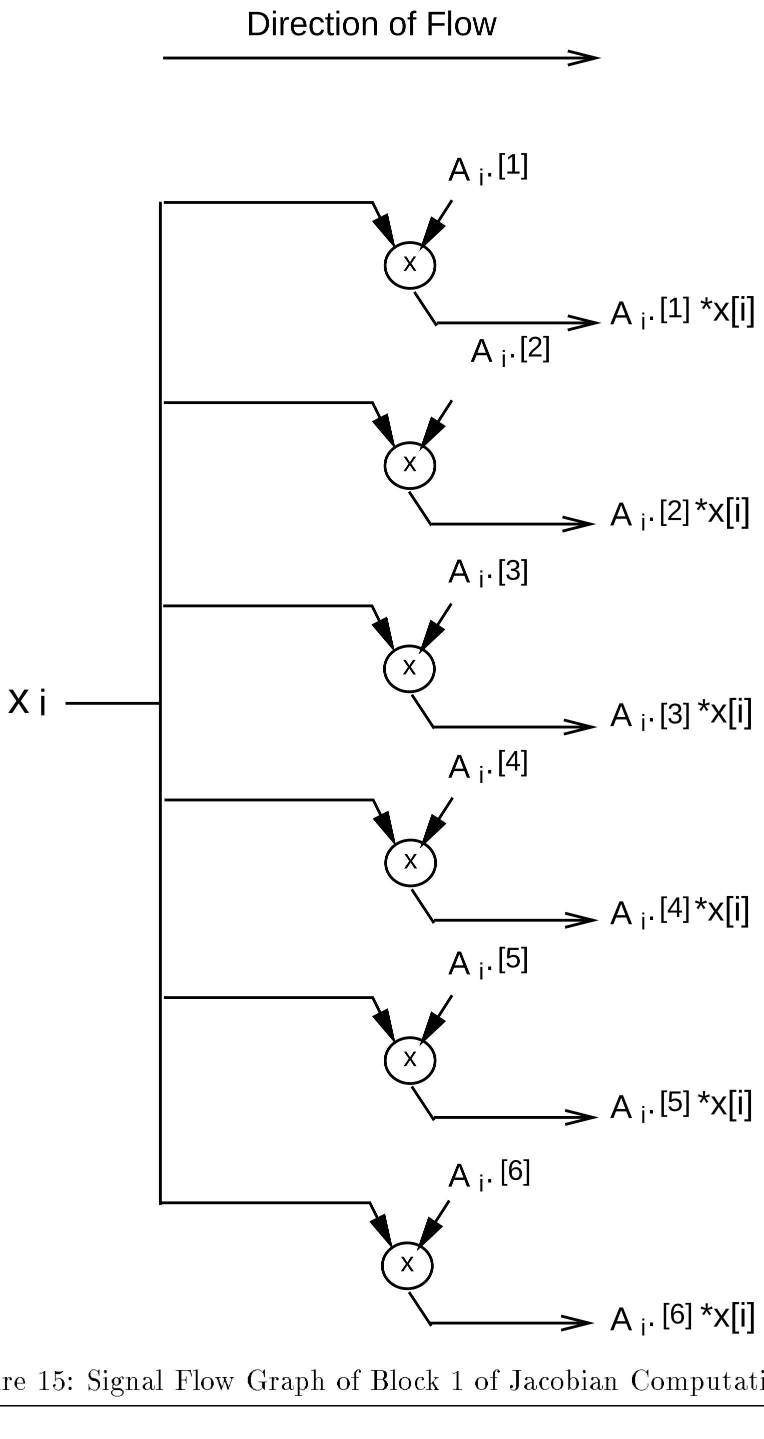


Figure 15: Signal Flow Graph of Block 1 of Jacobian Computation

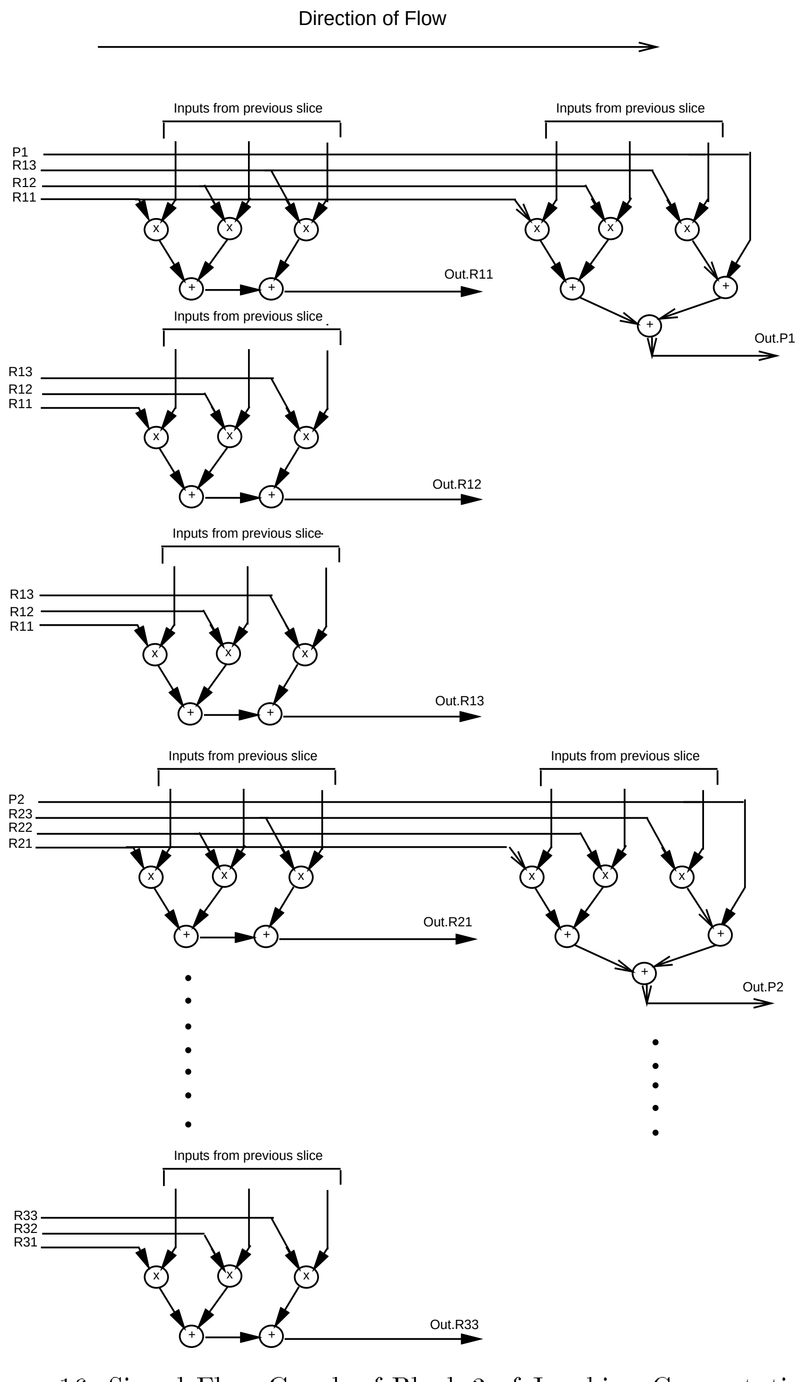


Figure 16: Signal Flow Graph of Block 3 of Jacobian Computation

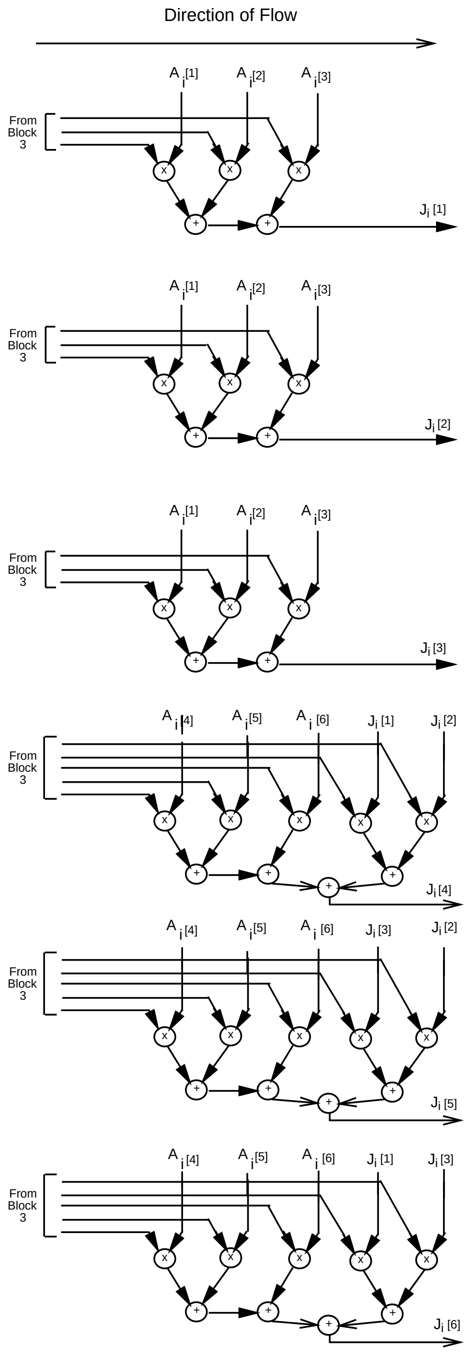


Figure 17: Signal Flow Graph of Block 4 of Jacobian Computation

Design #	# Multipliers	# Adders	# Memory Ports	Total # of gates	Total Computaiton Time (μsec)
1	3	3	3	36,000	21.6
2	2	2	1	25,000	27.3
3	1	1	1	16,000	32.0

Table 4: Evaluation of Jacobian Implementations

we obtained a datapath for each of the blocks. As can be seen from Figures 15, 16 and 17, all the Blocks have similar computations and thus, the datapaths for each of them are very similar. Next, we merged the datapaths to obtain a single datapath for the entire computation. The datapath was optimized for Block 3 since, in addition to being the most computation-intensive block, it also lies on the critical path of the computation.

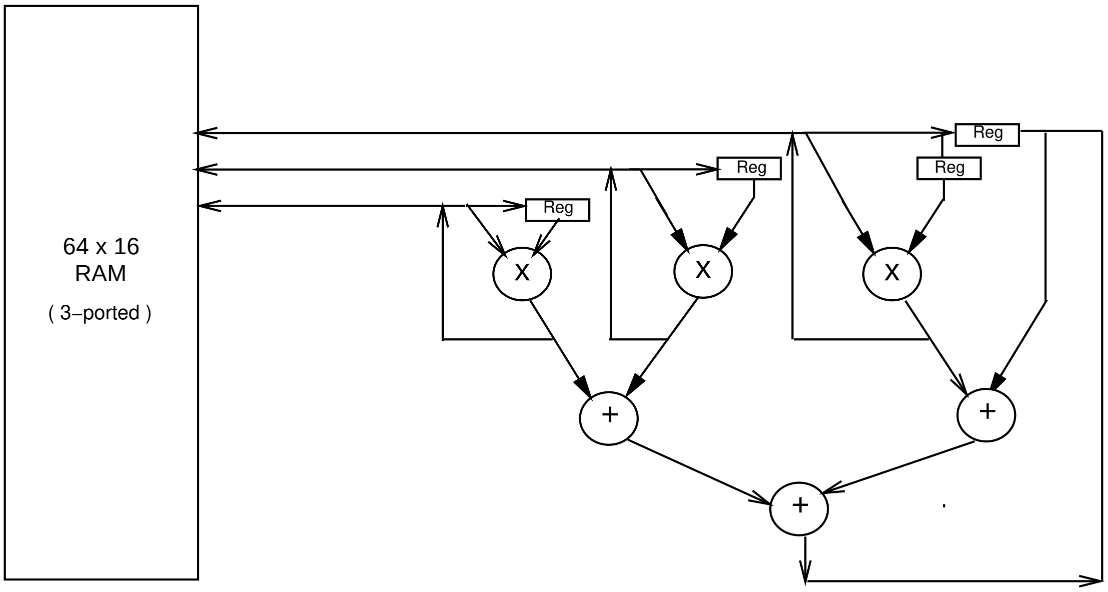


Figure 18: Datapath of Design #1

The largest datapath that we could afford with the constraint of using a single chip consists (Figure 18) of 3 16-bit floating point multipliers, 3 32-bit floating point adders and a 64×16 3-ported RAM. This appears as Design#1 in Table 4. Table 4 also lists the cost (total # of gates), and the performance (time taken for evaluating the Jacobian for one set of inputs, x_1 to x_8) for two additional designs, Design #2 and #3 which were obtained by reducing the block size to contain a fewer number of functional units.

6 Conclusions

In this report we have presented an architecture taxonomy and a strategy for exploring the architecture space of a design. In order to validate the feasibility of our taxonomy and exploration strategy we have investigated the architecture space of four examples: a timer system, an FIR filter, an FFT datapath, and a robot kinematics system. It is to be noted that we have only dealt with custom datapath and control architectures.

We now propose a methodology for architectural exploration and give a description of the design tasks required at each step. Figure 19 gives an overview of the proposed methodology.

1. The designer input consists of a high level description of the design (in VHDL, C, etc.) and a set of constraints (cost of design, power dissipation, number of memory modules, expected performance, etc.). The high level description is parsed into a hierarchical data flow graph which serves as an intermediate representation. Alternatively, for small designs, the user can input his description directly in this format.

2. Next, the hierarchical flow graph is analyzed in order to determine

   (a) the critical path,

   (b) the data dependencies, and

   (c) the communication flow of the design.

   The results of this analysis are used to slice and block the computation.

3. The design can be implemented using a standard or a custom datapath. This decision can be made by a designer, or alternatively, a tool can be employed to make a decision based on designer constraints. If the designer opts to use a standard datapath, the next task in the design process consists of selecting a processor from the component library and compiling the input description to the assembly code for the chosen processor. A software estimator ([4]) can then be used to verify that performance constraints are met. If constraints are not met, the designer (or the tool) can chose an alternative target processor or datapath and repeat the estimation.

   The tasks of processor selection, compiling, and software estimation can be automated to facilitate rapid exploration of design alternatives.

4. If the designer opts to customize the datapath, the next task involves pipelining the dataflow graph. This is done by inserting pipelining stages so as to divide all paths

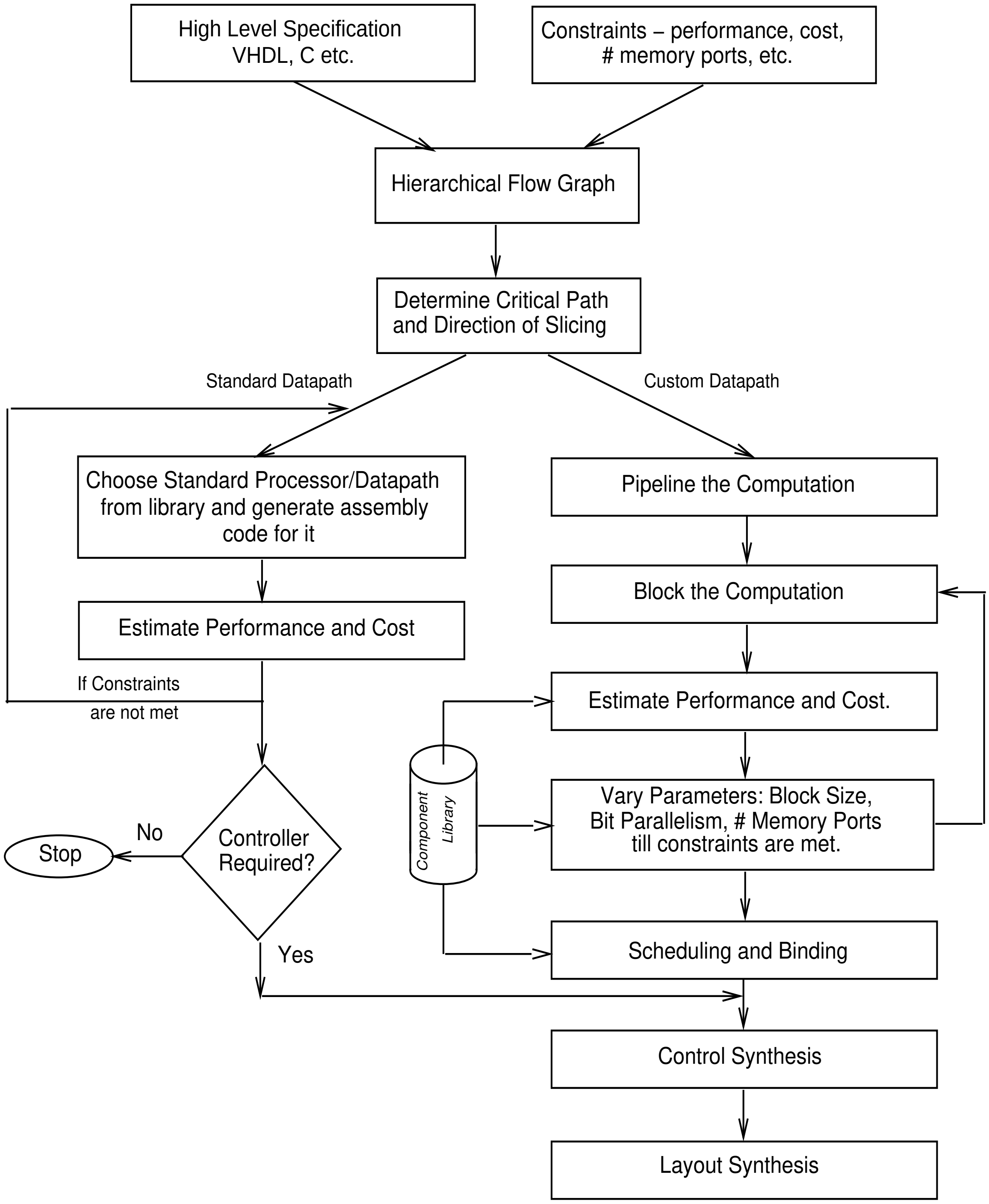


Figure 19: Design Exploration Methodology

of the computation into stages of equal "time steps". Once again, this task can be automated.

5. Next, based on given constraints, the library of available components, the direction of slicing and the critical path, a computation block is selected. This is equivalent to partitioning the data flow graph given a set of constraints and a preferred direction of partitioning (or slicing). A hardware estimator is them employed to estimate quality metrics such as the cost and performance of the design. If design constraints are violated the computation is repartitioned by varying design parameters such as the size of the block, the extent of parallelism etc.

   The process of choosing an "optimal" block is an iterative one as described in the previous sections and estimation is an important step in this process.

6. After determining the computation block, the final design can be obtained by a series of well known tasks such as scheduling, binding, ([3]) control synthesis and finally, layout synthesis ([6]).

The exploration methodology described above allows a designer to rapidly evaluate a large spectrum of implementations. We not only believe that considerable designer effort and time can be saved by automating and integrating these design tasks, but also that the resulting design will be more cost-effective since it would be selected only after an exhaustive search of the design space.

7 Acknowledgements

This work was partially supported by the Semiconducter Research Corporation grant #92-DJ-316, and we gratefully acknowledge their support. We also extend our gratitude to Frank C. Park for his help in providing the example of the Jacobian Computation.

References

[1] E.Bidet, C. Joanblanq, and P.Senn, *GENRIF: An Integrated FIR Filter Compiler*, Proceedings of EDAC, 1993.

[2] C.M. Chu, M. Potkonjak, M. Thaler and J. Rabaey, *HYPER: An Interactive Synthesis Environment for High Performance Real Time Applications*, Proceedings of the International Conference on Computer Design, pp. 432-435, 1989.

[3] D. D. Gajski, N. Dutt, A. Wu, and S. Lin, *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers, 1992.

[4] J. Gong, D. Gajski, and S. Narayan, "Software estimation from executable specifications", UC Irvine, Dept. of ICS, Technical Report 93-5, 1993.

[5] A. Oppenheim, A. Willsky and I. Young, *Signals and Systems*, Prentice Hall Inc.,1983

[6] B. Preas and M Lorenzetti, *Physical Design Automation of VLSI Systems,* Benjamin/Cummings, 1988.

[7] F.C. Park and A.P. Murray, *Computational and Modeling Aspects of the Product-of-Exponentials Formula for Robot Kinematics*, to appear in IEEE Transactions on Automatic Control, 1993.

[8] A Tutorial on *Implementation and Synthesis of VLSI Signal Processing*, presented by Keshab K. Parhi and Jan Rabaey, and produced by IEEE Educational Activities Board in cooperation with the IEEE Signal Processing Society.

[9] Toshiba, *TC140G/14L Series Megacell Megafunction* ASIC Gate Array Library, 1990