STATIC TIMING ANALYSIS BASED TRANSFORMATIONS OF SUPER-COMPLEX INSTRUCTION SET HARDWARE FUNCTIONS

by

Colin J. Ihrig

B.S. Computer Engineering, University of Pittsburgh, 2005

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Master of Science

University of Pittsburgh

2008

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This thesis was presented

by

Colin J. Ihrig

It was defended on

March 6th 2008

and approved by

Alex K. Jones, Assistant Professor, Department of Electrical and Computer Engineering

Steven Levitan, John A. Jurenko Professor, Department of Electrical and Computer Engineering

Jun Yang, Assistant Professor, Department of Electrical and Computer Engineering

Thesis Advisor: Alex K. Jones, Assistant Professor, Department of Electrical and Computer Engineering

STATIC TIMING ANALYSIS BASED TRANSFORMATIONS OF SUPER-COMPLEX INSTRUCTION SET HARDWARE FUNCTIONS

Colin J. Ihrig, M.S.

University of Pittsburgh, 2008

Application specific hardware implementations are an increasingly popular way of reducing execution time and power consumption in embedded systems. This application specific hardware typically consumes a small fraction of the execution time and power consumption that the equivalent software code would require. Modern electronic design automation (EDA) tools can be used to apply a variety of transformations to hardware blocks in an effort to achieve additional performance and power savings. A number of such transformations require a tool with knowledge of the designs' timing characteristics.

This thesis describes a static timing analyzer and two timing analysis based design automation tools. The static timing analyzer estimates the worst-case timing characteristics of a hardware data flow graph. These hardware data flow graphs are intermediate representations generated within a C to VHDL hardware acceleration compiler. Two EDA tools were then developed which utilize static timing analysis. An automated pipelining tool was developed to increase the throughput of large blocks of combinational logic generated by the hardware acceleration compiler. Another tool was designed in an attempt to mitigate power consumption resulting from extraneous combinational switching. By inserting special signal buffers, known as delay elements, with preselected propagation delays, combinational functional units can be kept inactive until their inputs have stabilized. The hardware descriptions generated by both tools were synthesized, simulated, and power profiled using existing commercial EDA tools. The results show that pipelining leads to an average performance increase of 3.3x, while delay elements saved between 25% and 33% of the power consumption when tested on a set of signal and image processing benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

As the density of chips increases, the use of application specific hardware processing blocks is becoming more commonplace, particularly in embedded systems. These hardware blocks can be used to achieve higher performance and lower power consumption; both of which are precious, particularly in embedded systems. This technique, known as *hardware acceleration*, has been implemented in a variety of fashions. The approach presented in this thesis converts portions of applications written in the C programming language into synthesizable VHSIC hardware description language (VHDL) blocks of combinational logic. By making the combinational hardware blocks accessible to a general purpose processor, they can be thought of as super-complex instructions. The resulting architecture is referred to as a super complex instruction set computer (SuperCISC) [18].

There are a number of existing circuit transformation techniques commonly implemented in commercial EDA tools. Retiming and pipelining are just a few examples of such techniques. These and a number of other techniques both require that the tool has detailed knowledge of the timing characteristics of the circuit. A common technique for gathering this timing information is known as static timing analysis. Static timing analysis is used to determine the worst-case timing characteristics of a digital circuit without requiring simulation [6].

Large blocks of combinational logic are prone to having long latencies. These long latencies impede the throughput of the circuit. One method to alleviate this problem is to pipeline the combinational logic. By pipelining the circuit, multiple data sets can be operated on in parallel. Large combinational circuits also suffer an increase in dynamic power consumption due to extraneous glitching. However, it is possible to attack this problem by applying a number of circuit techniques. This thesis tackles this problem by using static timing analysis applied at the behavioral synthesis level. In particular, this thesis provides the following contributions.

1. *Contribution 1*: A static timing analyzer for the SuperCISC compiler. The annotations created by the static timing analyzer allow for the design and use of timing based tools such as the ones described in this thesis.

2. *Contribution 2*: An automated pipelining tool. The pipelining tool allows for the rapid development of pipelined circuits, requiring no design effort on the part of the developer. Additionally, pipelines of different frequencies can be tested via a user defined input parameter.

3. *Contribution 3*: A greedy heuristic algorithm for delay element insertion. The greedy algorithm attempts to judiciously insert delay elements into a SuperCISC data flow graph. A brute force delay element insertion algorithm was also designed. Additionally, a tool for manual placement of delay elements was developed for testing purposes.

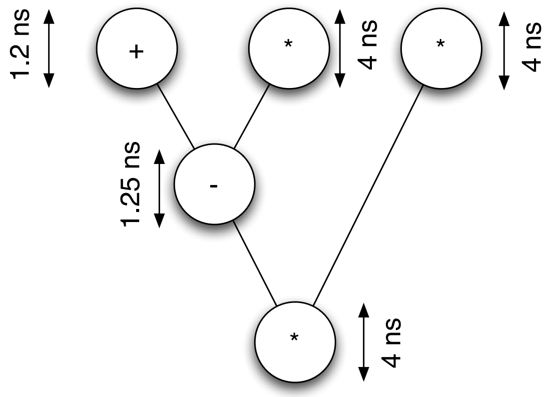
4. *Contribution 4*: A complete post-SuperCISC tool flow for delay element based designs. The tool chain performs synthesis, simulation, and power estimations on the delay element circuits. This contribution consists of a Design Compiler [34] technology library of delay elements, synthesis timing constraints, and a delay element VITAL [5] simulation library.

Increasing performance through software code optimization has yielded disappointing results compared to hardware acceleration. Modern processors include vector processing engines based on single instruction, multiple data (SIMD) or very long instruction word (VLIW) engines [22]. However, to effectively utilize these architectures for acceleration requires special assembly-level programming and the parallelism is often quite limited. For example, even when scaling VLIW processors to handle four or more instructions simultaneously, due to dependencies in the software, instruction level parallelism (ILP) rarely reaches two [22].

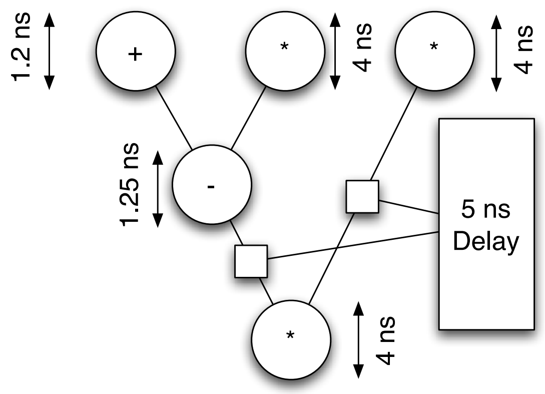
Implementing multiple software instructions as a block of custom hardware can significantly improve the effective parallelism, while consuming less power. While the parallelism achieved with custom hardware is superior to that achieved with software alone, this thesis attempts to further the amount of parallelism seen. Large blocks of combinational logic typically have long critical paths. Unfortunately, these critical paths tend to get longer as the hardware blocks get bigger. Because the critical path defines the maximum performance of a digital system, these long latencies can severely impede performance. One of the contributions of this thesis is a pipelining tool used to increase throughput by overlapping execution of multiple sets of data.

While custom hardware can provide significant avenues to increased performance, this does not come without a cost. Increases in the amount of computation completed in parallel can often significantly increase the power consumption of the processing element. Typically this increased power consumption is due to increased length of the combinational paths in the hardware blocks as the number of nodes is increased. As these paths increase, functional units down the line glitch for longer and longer periods of time leading to increased power consumption.

This thesis focuses on mitigating the effects on power consumption caused by the increase in switching activity due to hardware acceleration techniques. It seeks to achieve improved power consumption without sacrificing the large improvements in performance achieved. The proposed solution makes use of special circuits referred to as delay elements. These delay elements are designed to act as signal buffers with preselected propagation delays. Delay elements, in conjunction with latches, are inserted into combinational logic blocks in order to reduce switching, the source of dynamic power consumption in modern circuits, by keeping functional units inactive until their inputs are available. The delay elements are controlled with a global enable signal. Once the enable signal propagates through the delay elements, the latches are turned on, allowing computed data values to pass through. An example is shown in Figure 1.



(a) Without a delay element.



(b) With a delay element.

Figure 1: Impact of Delay Elements on a Hardware Graph

If the hardware from Figure 1(a) begins calculating at 0 ns, bit switching begins to reach the multiplier in the third row at $0 + \delta$ where δ is assumed to be negligible. However, it takes 4 ns for the right operand to become stable and 5.25 ns for the left operand (4 ns for the multiplier and an additional 1.25 ns for the subtractor) to become stable. Thus, the multiplier is operating and consuming switching power for 9.25 ns. Unfortunately, the multiplier is only performing useful computation for the last 4 ns.

However, if a delay element is inserted as shown in Figure 1(b), it is connected to a latch which gates the left and right operands to the third row multiplier. Thus, for the first 5 ns the multiplier is blocked from calculating a new value. After 5 ns these latches are enabled by the delay element and the already computed value from the right is passed to the multiplier. The left operand requires an additional 0.25 ns prior to stabilizing. Thus, the multiplier switches for 4.25 ns rather than 9.25 ns, saving power without a significant decrease in performance.

The remainder of this thesis is organized as follows: Section 2 provides a background. More specifically, Section 2.1 focuses on the creation of SuperCISC hardware from high level C code, and Section 2.2 explores related previous work. In Section 3, the algorithm used to perform static timing analysis is described. In Section 4, the static timing analyzer is used to create pipelined SuperCISC hardware. Section 5 introduces the technology dependent delay elements. An algorithm for inserting the delay elements into combinational logic is described in Section 5.2. The augmented delay element tool flow from synthesis through simulation and power estimation is described in Section 6. Section 7 briefly describes the benchmark suite, and provides results for pipelining as well as delay elements. Delay element and pipelining results are compared in Section 7.3. Section 8 discusses the future directions of this work. Finally, Section 9 provides the conclusions.

2.0 BACKGROUND

2.1 SUPERCISC HARDWARE FUNCTIONS

Processors used for general purpose computing sacrifice performance and power consumption in exchange for flexibility. In addition to this flexibility, processors allow for very fast design times. While flexibility is desirable in desktop computing, this tradeoff often proves to be counter productive in embedded environments where limited battery life and real-time constraints place performance and power consumption at a premium. At the opposite end of the spectrum, ASICs provide the best solutions in terms of performance and power, but have design times ranging from months to years. However, most embedded systems are typically also tailored to be application specific, allowing the use of ASICs to become a viable option.

A general trend seen during application profiling is that 90% of an application's execution time is spent in approximately 10% of the code [14]. In many scientific, multimedia, and signal processing applications the execution time is dominated by computationally intensive kernels within loops. Thus, an efficient heterogeneous solution is to tightly couple reduced instruction set computing (RISC) processors with application specific coprocessors. These coprocessors, which can execute many instructions at once are referred to as super-complex instructions. These instructions are the foundation of the resulting heterogeneous architecture, referred to as the SuperCISC architecture.

In [18] a heterogeneous architecture is introduced which consists of a RISC VLIW processor with application specific SuperCISC combinational *hardware functions* [14]. Figure 2 provides a high level conceptual view of this system. The middle section of the figure represents the four-way VLIW processor. It includes the four ALUs, instruction decoder, controller, instruction RAM, and the shared register file. The shared register file is accessible to each of the hardware functions surrounding the processor. Each of the eight hardware functions shown is comprised of a group of functional units (labeled FU). The SuperCISC functions do not contain any storage elements, but are tightly coupled with the VLIW processor through the use of a shared register file. Because arrays are typically not stored entirely in the register file, the SuperCISC compiler does not support arrays. In order to overcome this, array values are scalarized in the source code as needed. Loops are also not supported by the SuperCISC compiler because their hardware implementations require storage. Currently loops are handled in one of two ways. In the first method, the loop body is implemented as a hardware function, with the loop control handled in software. In the second method, the loop is unrolled and additional variables are introduced into the source code. The first approach requires less effort on the part of the programmer, but requires execution to cross the hardware-software boundary numerous times. The second approach requires the programmer to explicitly modify the code. However, the resulting hardware functions only need to cross the hardware-software boundary once, and are typically much larger.

2.1.1 SuperCISC Tool Flow

By utilizing an automated C to VHDL tool flow based on the Stanford University Intermediate Format (SUIF) research compiler [39], an application written in the high level C programming language can be converted to a hardware description. This abstraction allows programmers with little to no hardware design experience to accelerate the execution kernels of their applications. These execution kernels are first identified via application profiling. Once an application's kernels have been identified, they are marked for hardware acceleration. In the SuperCISC compiler, the segments of code marked for hardware acceleration are annotated with *pragma* directives. Figure 3 contains the C source code for the Sobel benchmark's kernel. The *HWstart* and *HWend* pragmas are used to delimit the boundaries of the hardware function. The code within the pragma directives will be converted into a combinational hardware function.

Once the source code has been properly annotated, the SuperCISC compiler creates a control flow graph (CFG). In a CFG, each node represents a basic block. A basic block is a straight-line section of code with exactly one entry point and exactly one exit point. Because basic blocks boundaries are denoted by control instructions (branches, jumps, etc.) they are typically small in

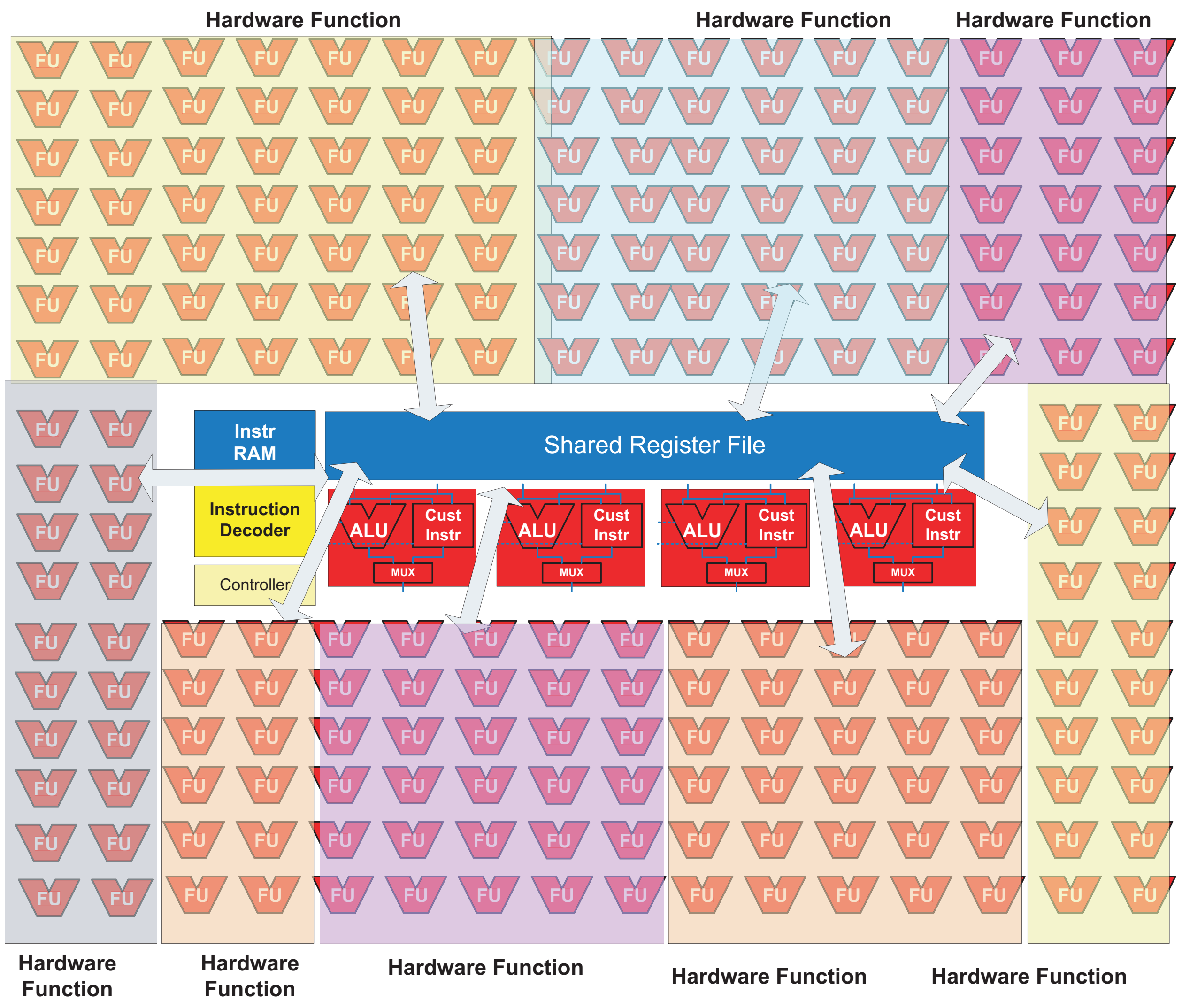


Figure 2: SuperCISC Conceptual View

size. The CFG for the Sobel benchmark is shown in the top left corner of Figure 4. Figure 4 also contains a data flow graph (DFG) for each basic block in the CFG. The combination of the CFG and all of the DFGs is referred to as a control and data flow graph (CDFG). In the CFG, the edges between basic blocks represent control dependencies in the application. These control dependencies correspond to control structures in the source code such as loops and *if-else* statements in the source. On the other hand, the edges in the DFGs represent data dependencies between the different operations in the source code. Each DFG also contains a set of input and output nodes.

```
// Begin SuperCISC Hardware Function
#pragma HWstart

   e1 = x3 - x0;
   e2 = 2 * x4;
   e3 = 2 * x1;
   e4 = x5 - x2;
   e5 = e1 + e2;
   e6 = e4 - e3;
   gx = e5 + e6;

   if ( gx < 0 )
      c = 0 - gx;
   else
      c = gx;

   e1 = x2 - x0;
   e2 = 2 * x7;
   e3 = 2 * x6;
   e4 = x5 - x3;
   e5 = e1 + e2;
   e6 = e4 - e3;
   gy = e5 + e6;

   if ( gy < 0 )
      c += 0 - gy;
   else
      c += gy;

   if ( c > 255 )
      c = 255;

#pragma HWend
// End SuperCISC Hardware Function
```

Figure 3: Sobel Kernel C Source Code

The output nodes labeled *eval* in basic blocks zero, three, and six are of particular importance. In the context of the CDFG, the *eval* outputs are used to determine which control path to follow in the event of a branch.

Working at the basic block level severely limits the achievable size of the synthesized hardware functions. In order to overcome these shortcomings, a technique called *hardware predication* [14] is used. Hardware predication is a method which replaces control dependencies with data

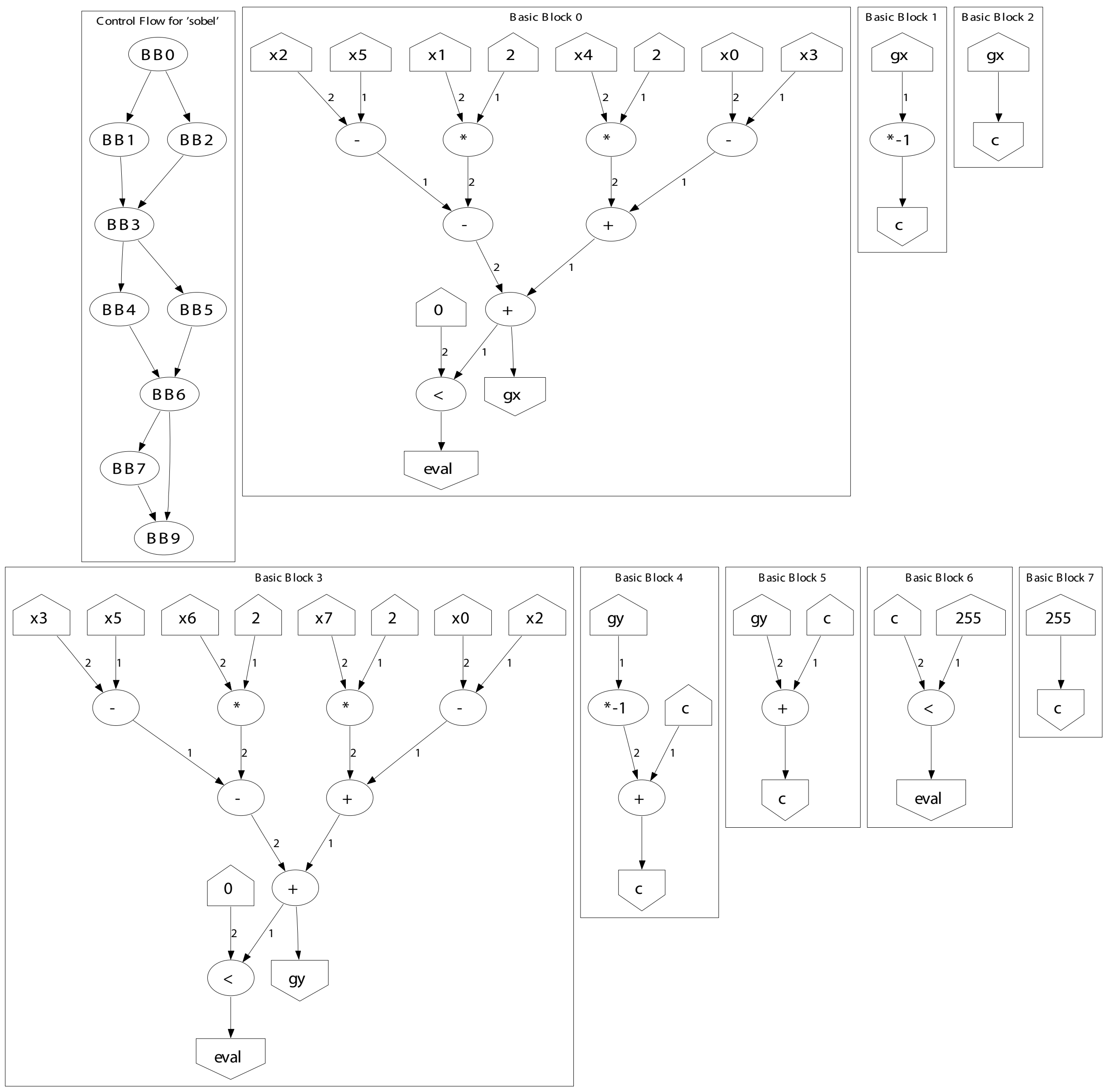


Figure 4: Sobel CDFG Representation

dependencies, which can be used to merge basic blocks. Using hardware predication, both control paths of a branch are executed simultaneously. A multiplexer is then used to select the results from one of the control paths. The multiplexer's select signal is driven by the *eval* output of the basic block where the control branch originated. The post-predication version of the CDFG is called a

Super Data Flow Graph (SDFG). Figure 5 shows the SDFG for the Sobel benchmark following predication of the CDFG in Figure 4. The SDFG contains three multiplexers which are derived from the three control structures in the CDFG (originating in basic blocks zero, three, and six). Additionally, intermediate variables such as *gx* and *gy* have been removed from the graph because they do not live beyond the end of the hardware function, and thus no longer require storage.

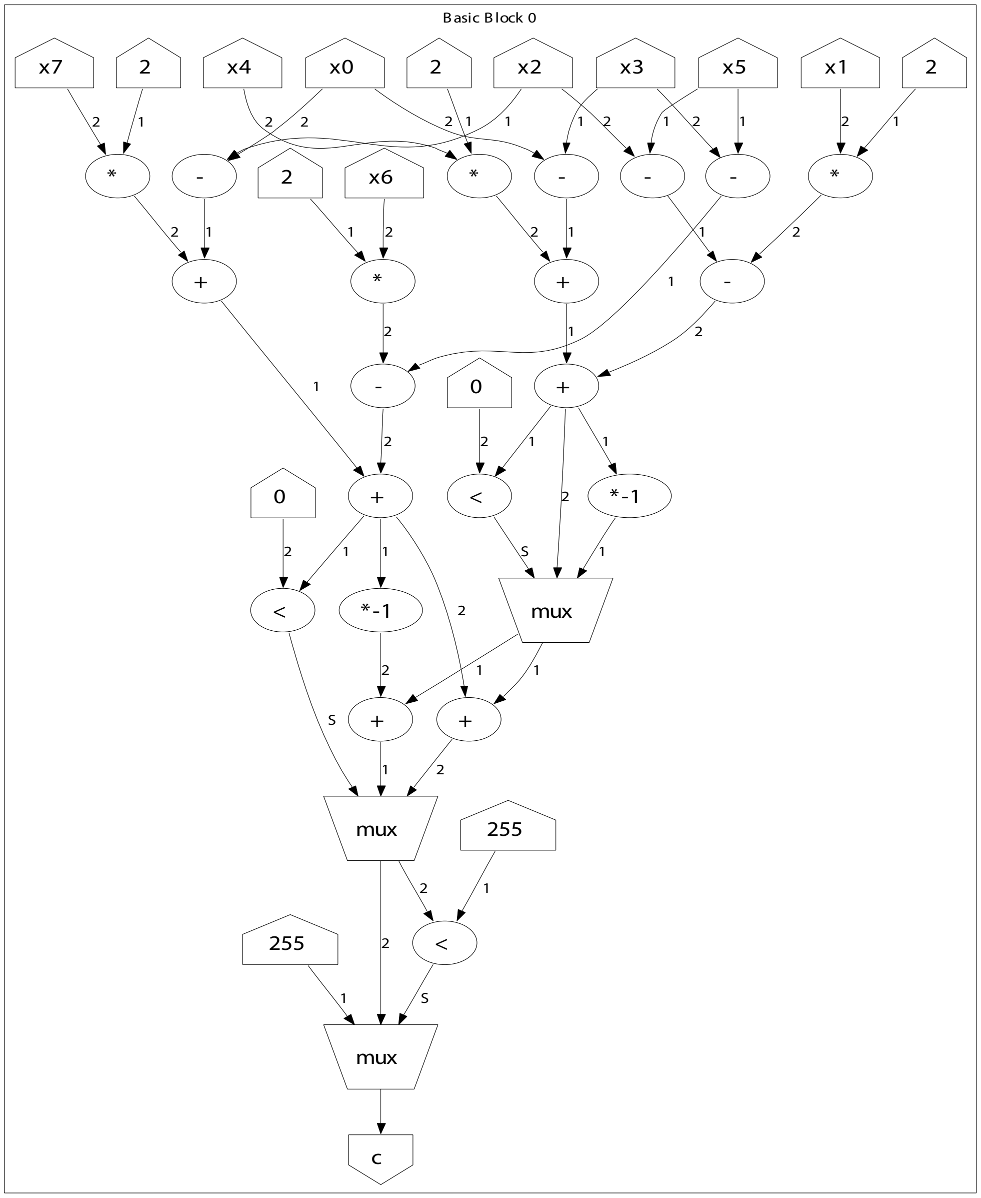


Figure 5: Sobel SDFG Representation

After predication has been performed, the final phase of the compiler generates valid, synthesizable VHDL. A VHDL entity and corresponding architecture is generated for each functional unit present in the SDFG. These functional units are instantiated as components within the top level design. The top level design is comprised of a number of port mappings between the functional units. The connectivity is implemented as a reflection of the SDFG's edges. Finally, the inputs and outputs of the top level design are implemented as VHDL *IN* and *OUT* ports. Figure 6 shows the resulting entity for the Sobel benchmark. The *x0-x7* inputs and *c_out* output correspond to the input and output nodes in the SDFG. The SuperCISC compiler follows a notation in which *_out* is appended to each output's name. The VHDL architecture has been omitted for brevity.

```
entity sobel is
     port(
          signal x0    : IN signed(31 DOWNTO 0);
          signal x1    : IN signed(31 DOWNTO 0);
          signal x2    : IN signed(31 DOWNTO 0);
          signal x3    : IN signed(31 DOWNTO 0);
          signal x4    : IN signed(31 DOWNTO 0);
          signal x5    : IN signed(31 DOWNTO 0);
          signal x6    : IN signed(31 DOWNTO 0);
          signal x7    : IN signed(31 DOWNTO 0);
          signal c_out : OUT signed(31 DOWNTO 0);
     );
end sobel;
```

Figure 6: Sobel VHDL Entity

In [19] and [20], the effects on performance from moving software kernels into SuperCISC hardware functions were studied. The results of this work showed that custom hardware is more efficient than software in terms of both execution time and power consumption. Because hardware executes more quickly than software, the overall time required to do the same amount of processing is reduced. This was shown to result in speedups of 10x and greater. The SDFG based synthesis approach was also shown to be energy efficient. In [19] the energy savings were found to range from 42x to over 418x over software alone. *Power compression* is the primary source of power savings in SuperCISC hardware. Power compression is the result of reducing the functionality from general purpose ALUs to application specific functional units such as adders and multipliers [19].

2.2 RELATED WORK

2.2.1 Static Timing Analysis

Static timing analysis is critical in the design and optimization of modern digital circuits. There are two broad categories of static timing analysis. The first is the worst-case response technique, also referred to as deterministic static timing analysis. This technique is relatively simple not computationally intensive. The worst-case technique has typically been the standard in industry. However, because it relies on the worst-case, this method of performing static timing can be inaccurate. The SuperCISC timing analyzer presented in this thesis utilizes the worst-case response technique.

The other primary method for performing static timing analysis is known as *statistical static timing analysis* (SSTA). SSTA replaces fixed gate delays with probability distributions. These probabilities can be used to model variations in semiconductor devices that are not handled by standard static timing analysis. This method is more computationally intensive than the worst-case response. Some algorithms suffer from exponential complexity due to *reconvergent fanouts* [6]. Reconvergent fanout occurs when the outputs of a node follow different paths through the circuit and reconverge downstream as the inputs to another gate. Although SSTA has been the topic of much research in recent years, it is still the target of much criticism. It is often viewed as being too complex, with only small benefits being seen over its deterministic worst-case behavior alternative.

In [6] a statistical static timing analysis technique is introduced which models delay and arrival times in a circuit as random variables. This technique uses Cumulative Probability Distribution Functions (CDFs) to model arrival times, and Probability Density Functions (PDFs) to model gate delays. The authors present a framework which can be used to integrate deterministic static timing analysis into a SSTA system. The accuracy of the timing analyzer can also be varied according to the piecewise linear CDF model. This method suffers from the SSTA drawbacks described above. For example, this technique incurs a 10-30% performance penalty in order to handle reconvergent fanout.

Although not directly related to the techniques described in this thesis, static timing analysis has also been used to gauge the performance of software in embedded systems. In [26] the authors investigate the relevance of software static timing analysis. In this approach, software statements take the place of functional units. Path analysis is performed on *if-else* statements by calculating according to the branch path with the longest maximum delay. Software static timing analysis suffers from a number of drawbacks. These drawbacks restrict the type of code that a programmer can write. For example, the analyzer provided in this paper cannot handle unbounded loops, recursive function calls, or dynamic function calls (i.e. function pointers). Additionally, it is often impossible to calculate a program's worst case behavior because it is effectively the same as the halting problem [26].

2.2.2 Automating Pipelining

A common problem in the domain of hardware-software codesign is partitioning, or determining which parts of the application will be implemented in hardware and which parts will be implemented in software. The tool flow used in this paper relies on profiling to determine the execution kernels which will be implemented in hardware. Behavioral synthesis techniques can then be used to generate hardware descriptions from high-level languages such as C [36]. High-level synthesis is an increasingly popular technique in hardware design. Mentor Graphics' Catapult C product creates synthesizable hardware descriptions directly from a C/C++ front end [27]. The PACT project from Northwestern University creates power and performance optimized hardware descriptions from C programs [17]. The SPARK project from UC Irvine also converts C code to hardware, while attempting to achieve greater performance by extracting parallelism [11]. In contrast, SuperCISC generates entirely combinational Super Data Flow Graphs (SDFGs) through the use of hardware predication.

Several projects such as PipeRench [9] and HASTE [24] have investigated the mapping of execution kernels into coarse-grained reconfigurable fabrics utilizing ALUs. The RaPid project [7] presents another coarse-grain, pipelined, configurable architecture. The SuperCISC approach, in contrast, by generating completely application specific hardware, attempts to minimize the amount of area and energy used while sacrificing reconfigurability. The custom hardware lends itself towards a flexible, datapath driven pipelining algorithm. The SuperCISC method also suffers less cycle time waste than coarse-grain fabrics because the pipeline is tailored specifically to the datapath.

Other projects have studied ways to reduce power in high-performance systems without sacrificing speed. In [32] and [38] power and performance tradeoffs are explored using methods such as dual speed pipelines. By using high-speed and low-speed pipelines in conjunction, performance is increased, while area is sacrificed. In contrast, SuperCISC functions use hardware predication to achieve high-performance by expanding the kernel in a single very large dataflow graph. This graph is then rebuilt with pipelining to retain as much latency reduction as possible while simultaneously expanding the parallelism.

2.2.3 Reduction of Switching Power

Many techniques have been devised in an effort to reduce power consumption resulting from spurious logic transitions in digital circuits. Numerous techniques have been implemented at varying stages in the design process. The delay element based approach presented in this thesis is implemented at the behavioral synthesis level. Techniques such as this, which occur in the beginning of the design flow, tend to provide the most flexibility to the latter stages of the design process. Unfortunately, they also tend to be the least accurate.

In [33] the authors explore a behavioral synthesis methodology that attempts to lower power consumption in data-dominated circuits. Their approach works at the register-transfer level. The idea is to reduce power consumption by reducing the transition activity in the data path. Similar to the SuperCISC compiler, their behavioral synthesis tool operates on CDFGs. However, the targeted hardware in this approach is assumed to be clocked. The synthesis tool attempts to schedule the functional units on a cycle-by-cycle basis. This technique requires liveness analysis to be performed on the functional units in order to schedule the operations. The scheduler also attempts to schedule operations in a fashion such that there are fewer bit-level transitions between consecutive input sets to each functional unit. This technique involves simulating a CDFG with random input vectors and recording the number of bit transitions. While this technique saves on area by sharing functional units, it is not applicable to combinational hardware functions such as those in the SuperCISC system.

In [37] guarded evaluation is used to reduce power consumption. Guarded evaluation dynamically detects portions of a circuit that are inactive at each clock cycle. These identified sections can be turned off via clock gating. The authors provide a power optimized ALU as an example. An ALU supports a number of operations. In an unoptimized ALU, typically each operation is executed for every set of inputs. However, during any given clock cycle, only one of these operations' result will be used. By inserting guard logic on the functional units within the ALU it is possible to turn off all of the operations that will not be used. Similar to the delay element approach, guard logic is implemented using latches which selectively turn off portions of a circuit. In contrast however, guarded evaluation targets clocked circuits and occurs at the logic level synthesis design stage.

In [1] a technique called gate freezing is introduced. Gate freezing replaces gates suffering from large amounts of switching with functionally equivalent ones that can be ”frozen” by asserting a control signal. These equivalent gates are referred to as F-Gates. This approach requires the creation of a library of F-Gates. This can also lead to larger technology libraries, depending on the number of F-Gates created. Gate freezing is applied at the layout level. Because gate freezing occurs late in the design process, it is very accurate. However, at this late stage the opportunities for optimization are much more limited. In addition, this technique requires that a placed and routed circuit be power profiled prior to inserting any F-Gates. After power profiling, a user-defined percentage of the gates are replaced with F-Gates. The control signals are then added to the design. After making these modifications it is necessary to re-test the circuit to ensure that the functionality is still in tact.

Gate resizing [12] is an alternative method for reducing switching. Gate resizing works by attempting to create path delays through the circuit with matching delays. Gates lying on paths with fast propagation delays (with the exception of critical paths) can be downsized in order to equalize delays. Path equalization takes quite a bit of effort and may not always be possible. Technology libraries also become very bloated due to the various sized gates required.

3.0 STATIC TIMING ANALYSIS

Static timing analysis is a method of estimating worst-case execution times in digital circuits. Static timing analysis is particularly useful in calculating a circuit's timing properties without requiring the circuit to be simulated. Without static timing analysis, it would be necessary to generate a set of input vectors and perform a full timing-driven gate level simulation. The *static* qualifier implies that timing analysis calculations are independent of the design's input stimuli. This implication requires that the calculation must account for all possible input combinations, thus providing the worst-case timing. Static timing can also be used to determine a variety of properties of a circuit. Among these properties are the length of the critical path, false paths, slack associated with paths through the circuit, and required arrival times [23].

Synchronous circuits in particular are often characterized by the clock frequency at which they operate. Static timing can, and often is, performed at various stages of the design flow. These points include but are not limited to the synthesis and place and route stages. Typically, the later the stage in the design process, the more accurate the results from timing analysis. For example, post place and route timing analysis incorporates additional knowledge about wirelength which is not known during timing analysis at the behavioral synthesis level.

Over the past several decades, various techniques for performing static timing analysis have been introduced. Two prominent methods are the Project Evaluation and Review Technique (PERT) [16] and the Critical Path Method [40]. Both of these techniques were developed originally for project management. From a project management standpoint, project tasks are treated as graph nodes. Each task has a delay associated with it. While the Critical Path Method and its variations are currently the most widely used methods, other techniques such as basic graph traversal algorithms are used in various implementations of static timing analyzers.

3.1 STATIC TIMING ANALYSIS FOR SUPERCISC

Static timing analysis in the SuperCISC compiler is performed using a recursive depth-first search, starting at the output nodes and ending at the input nodes. Depth-first search has a space complexity of $O(h)$, where h is the longest path through the graph [4]. It is worth noting that the longest path through the graph is not necessarily the same as the critical path due to the differing weights associated with the nodes. The time complexity of a depth-first search is on the order of $O(N + E)$, where N is the number of nodes in the graph and E is the number of edges [4]. This is also equivalent to $O(b^d)$, where b is the branch factor, and d is the depth of the graph [4]. Because the search is performed starting at the output nodes, the branch factor for each node is equivalent to the fanin of the functional unit represented by that node. Multiplexers have a fanin of three, while all other functional units have either one or two inputs. In an additional effort to minimize the search's execution time, nodes which have already been explored are remembered by annotating the node with timing and power data that will be required in the future for pipelining and delay element analysis. Lastly, depth-first search is guaranteed to be complete in the search space as long as infinite paths are not possible. Because SDFG's do not contain any cycles, infinite paths cannot naturally occur.

The SuperCISC compiler operates at the high level of the C language. As such, static timing analysis within the SuperCISC tool flow also occurs at a high level. For any path through the graph, the worst-case combinational delay is given by Σd_i, where d is the delay associated with a given node, and i spans over each of the nodes in the path. The path with the largest Σd_i value is the critical path and determines the performance limitations of a combinational hardware function. By applying knowledge of how synthesis tools implement and combine certain constructs within a design, it is possible to achieve slightly more accurate static timing results. For example, a shifter with a constant shift amount input can be synthesized as wires and constants, while a shifter with a variable shift amount cannot. Because of this, a constant shift takes less time to execute in hardware. Table 1 lists the delay values associated with each functional unit. The delay values shown were calculated by profiling each type of 32-bit functional unit individually. For example, a 32-bit multipler was found to have a delay of four nanoseconds when analyzed independently.

Table 1: Functional Unit Delays

Operation	**Delay**
ADD	1.20 ns
AND	0.12 ns
EQU	0.70 ns
GT	1.00 ns
GTE	1.00 ns
LT	1.05 ns
LTE	1.00 ns
NE	0.70 ns
MUX	0.15 ns
MUL	4.00 ns
NOT	0.04 ns
OR	0.18 ns
SUB	1.25 ns
XOR	0.19 ns

Figure 7 shows a portion of a data flow graph. The static static timing analyzer begins by examining the multiplier in the bottom row. Upon inspection, the timing analyzer determines that the node represents the hardware implementation of a multiplier. The tool then consults Table 1 to determine the delay to associate with the node. Since the node is a multiplier, a timing annotation of four nanoseconds is attached to the node. The algorithm proceeds by visiting the subtractor parent of the multiplier. The process is repeated for the subtractor and each of its parents. Once the entire graph has been processed, the tool can determine a number of important timing properties. The critical path length is determined to be 9.25 ns. This delay occurs along the path containing the subtractor and two multipliers. In addition to calculating the critical path of the entire circuit, it is also possible to determine the critical path to any node in the graph. During the backtracking phase of the initial graph traversal, each node is annotated with its *critical parent*. For a given node, the critical parent is defined as the parent node with longest cumulative path delay. For example, the critical parent of the subtractor is the multiplier in the top row. Because each conceptual output node in an SDFG has no delay associated with it, the circuit's critical path length can be calculated by finding the output node with the largest critical parent delay.

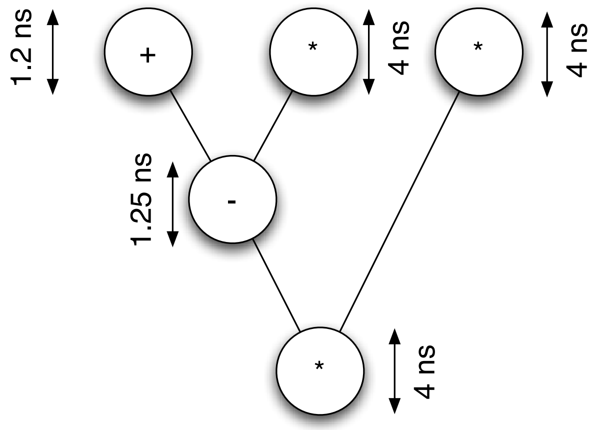


Figure 7: Data Flow Graph With Timing Annotations

Most static timing tools provide information for register to register delays. However, SuperCISC hardware functions are entirely combinational and thus do not have any register to register delays. To account for this, SuperCISC static timing analysis works under the assumption that a circuit's inputs and outputs would be registered, allowing the entire design to be treated as a single register to register delay.

4.0 PIPELINING

Pipelining is a common technique used in modern processors to increase throughput by overlapping multiple instructions in different phases of execution in an assembly line fashion. While pipelining does not improve the latency of a single instruction, the effects of parallel execution can boost the overall throughput of the processor, by allowing a higher maximum clock frequency. The effects of pipelining can be seen best when new inputs can be provided at every clock cycle, and all the stages of the pipeline are doing useful processing [13].

4.1 AUTOMATING PIPELINING

In order to introduce registers into a combinational datapath, it is necessary to add a clock signal. The clock signal frequency is a user-defined input to the tool. The clock period, which is the inverse of the frequency provided by the user, dictates how much combinational logic delay can be placed between any two pipeline registers. Static timing analysis is performed on the design in order to annotate all of the SDFG nodes with timing information. This timing information is then used to divide the design into discrete pipeline stages.

Once static timing is complete, the process of partitioning the SDFG into multiple pipeline stages begins. The algorithm attempts to place nodes into partitions in an as-soon-as-possible fashion. Each functional unit, having some amount of delay associated with it, will consume a certain portion of the available clock cycle time. The pipelining algorithm attempts to fit as many nodes as possible into as few partitions as possible without violating timing restrictions. In order to take full advantage of a pipelined function, it is desirable to be capable of processing new inputs on each clock cycle. For this reason, another constraint on pipelining is that a node cannot be partitioned until all of its parents have been partitioned first to prevent a possible data dependency violation. It is possible for a child node to be placed in the same partition as all, some, or none of its parents.

Partitioning begins by placing all inputs and constant values into the first partition. Each remaining node, *n*, that has not been partitioned is inspected. Each parent node of *n* that has not been partitioned will be visited and partitioned. Once all of the parent nodes have been partitioned, *n* is partitioned. A node will be placed into one of two possible partitions. The first possible partition is that of its parent node with the highest partition number. The first partition created has the lowest partition number, while the last partition created for the circuit has the highest partition number. If the node will not fit into its parent's partition due to timing constraints, then the node will be placed in the next partition in the pipeline. In cases where parent nodes reside in different partitions, it is necessary to align the inputs using dummy registers, whose only purpose is to delay data arrival by a clock cycle.

Figure 8 shows the 150 MHz pipelined equivalent of the SDFG in Figure 5. The combinational example SDFG has been divided into two pipeline stages. A clock input signal and an optional output register has also been included. Each of the benchmarks was pipelined at 200 MHz. The number of 200 MHz pipeline stages for each benchmark is shown in Table 2. Prior to pipelining, each design's maximum frequency was driven by its critical path length. Table 3 shows the critical path length and corresponding maximum frequency for each benchmark. None of the benchmarks' maximum combinational frequencies were even within 50 MHz of the 200 MHz pipeline.

The clock period of the pipeline can be selected arbitrarily, with the minimum possible period achievable governed by the node(s) in the graph associated with the largest delay. Register setup and hold times, as well as routing delays must also be taken into consideration in order to ensure that the requested clock period can be met. It should be noted that while the results of static timing are technology dependent, the algorithm is not. Similarly, the results of pipelining are dependent on the results of static timing as well as the requested clock period. The pipelining algorithm itself is also technology independent.

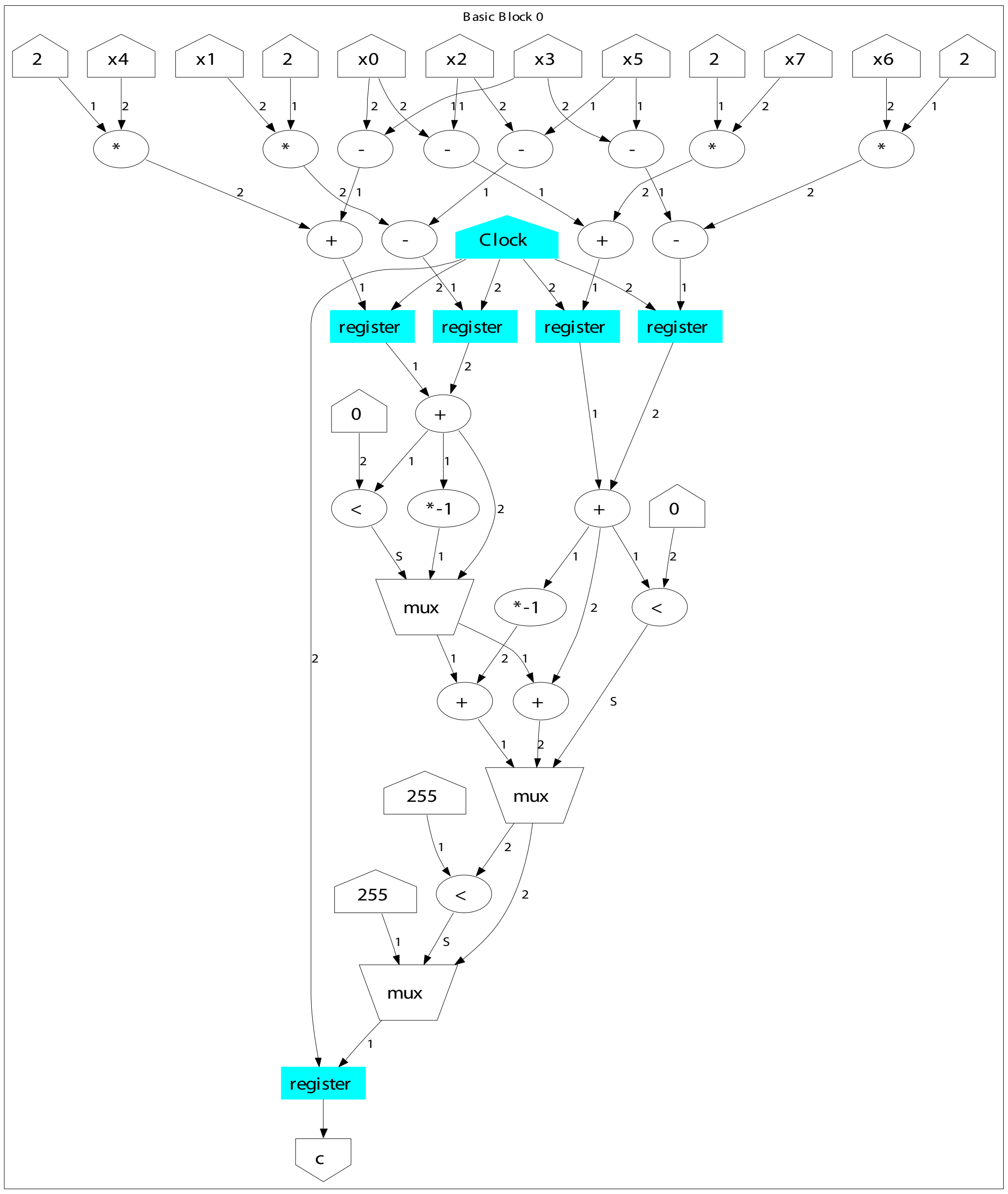


Figure 8: Pipelined Version of the Sobel Benchmark

4.2 DESIGN TRADEOFFS

This section examines the impact of SuperCISC synthesis and pipelining on the parallelism, execution time, area, and energy required for the implementation. The results presented here were

Table 2: Number of Pipeline Stages at 200 MHz

Benchmark	**Pipeline Stages**
Sobel	3
ADPCM Decoder	2
ADPCM Encoder	3
IDCT Row	5
IDCT Column	6
Laplace	2
GSM	4
CRC32	2
MPEG	12
JPEG	9

obtained by applying a pipelining frequency of 200 MHz. The target technology is the 0.13 μm IBM cell-based ASIC technology. Synopsys Design Compiler and PrimePower were used to generate synthesis and power estimations.

Part of the speedup achieved by SuperCISC functions comes from *cycle compression* [22]. Cycle compression is the execution of a sequence of arithmetic operations in a fraction of the time that it would take to execute the same operations in software. This is possible because the operations in the software implementation suffer from *cycle fragmentation* [22]. In a processor, the clock cycle length must be long enough to accommodate the delay associated with the critical path. However, simple operation such as logical OR take only a fraction of the entire cycle time to execute. The remainder of the cycle time is lost to fragmentation. Due to the strictly combinational nature of the SuperCISC hardware, cycle fragmentation is almost completely eliminated. The only remaining fragmentation comes in the final cycle of latency, between the time when the hardware function has completed and the time of the next clock edge. The result is a hardware function that, on average, executes at over 10x the speed of software alone.

Combinational SuperCISC hardware functions suffer from two major drawbacks. First, they incur a latency of several processor clock cycles due to their relatively long critical path lengths. In addition, these functions cannot begin processing a new set of input stimuli until the current set has finished and the result stored. This limitation impedes the throughput metric of the hardware kernels.

Table 3: Critical Path Lengths and Corresponding Maximum Frequencies

Benchmark	**Critical Path Length**	**Frequency**
Sobel	10.4 ns	96.15 MHz
ADPCM Decoder	6.80 ns	147.06 MHz
ADPCM Encoder	10.00 ns	100 MHz
IDCT Row	15.40 ns	64.94 MHz
IDCT Column	16.60 ns	60.24 MHz
Laplace	7.85 ns	127.39 MHz
GSM	15.99 ns	62.54 MHz
CRC32	7.9 ns	126.58 MHz
MPEG	39.60 ns	25.25 MHz
JPEG	25.95 ns	38.54 MHz

Pipelining cannot alleviate the latency problem associated with the SuperCISC functions. In fact, pipelining can cause an increase in the cycle count due to the reintroduction of cycle fragmentation. The amount of cycle fragmentation introduced through pipelining is dependent on the requested pipeline frequency and the delays associated with the varying types of functional units being pipelined. Figure 9 shows the effects of cycle fragmentation on both the combinational and pipelined versions of the ADPCM Decoder benchmark. As the frequency increases, additional cycles of latency are incurred. Between 150 and 200 MHz, the frequency increases, while the cycle latency remains unchanged, resulting in the drop in execution time seen in the graph. The effects of higher frequencies cause the pipelined implementation to become fragmented more quickly. The rise in fragmentation causes additional cycles of latency to become necessary, which increases the overall execution time. Figure 10 illustrates the effects of pipelining on the cycle count latency of the benchmark applications. The cycle count, as opposed to the absolute delay, is used to more accurately describe latency because a function's results will only become useful to synchronous hardware on the first clock edge following execution completion.

While pipelining may degrade the latency of SuperCISC hardware, it can be used to improve the throughput. Combinational implementations of hardware functions can process only a single set of inputs at any given time. This constraint results in a hardware function with a throughput of one result for every number of processor latency cycles associated with the function. By splitting the datapath of a function into two halves with pipeline registers, each half can process different

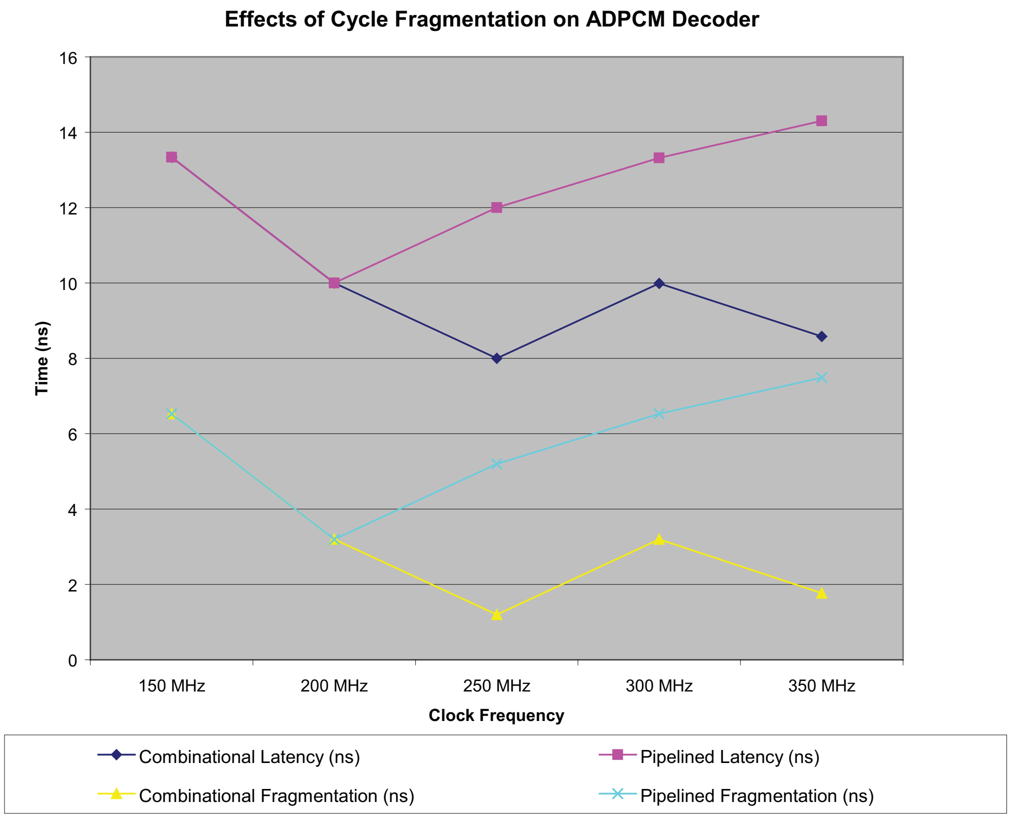


Figure 9: Cycle Fragmentation Losses Incurred in ADPCM Decoder

sets of inputs in parallel, with the two results separated by a single cycle. If new input data can be provided to the function every cycle, then the throughput of this implementation is one result per processor cycle.

Pipelining introduces additional registers into a design. The pipeline registers utilize additional on-chip area. A small optimization that can save on both area and energy is the exclusion of any registers storing constant values. Constant values will never change, making storage redundant. The additional hardware also consumes energy. Additionally, the introduction of a clock signal increases the dynamic power of a circuit. Studies have shown that the clock signals in digital systems can account for between 15% to 45% of the overall system power consumption [35].

Figure 11 and Figure 12 display the execution time and area tradeoffs observed in the pipelined hardware functions versus their combinational equivalents. The execution time, on average, de-

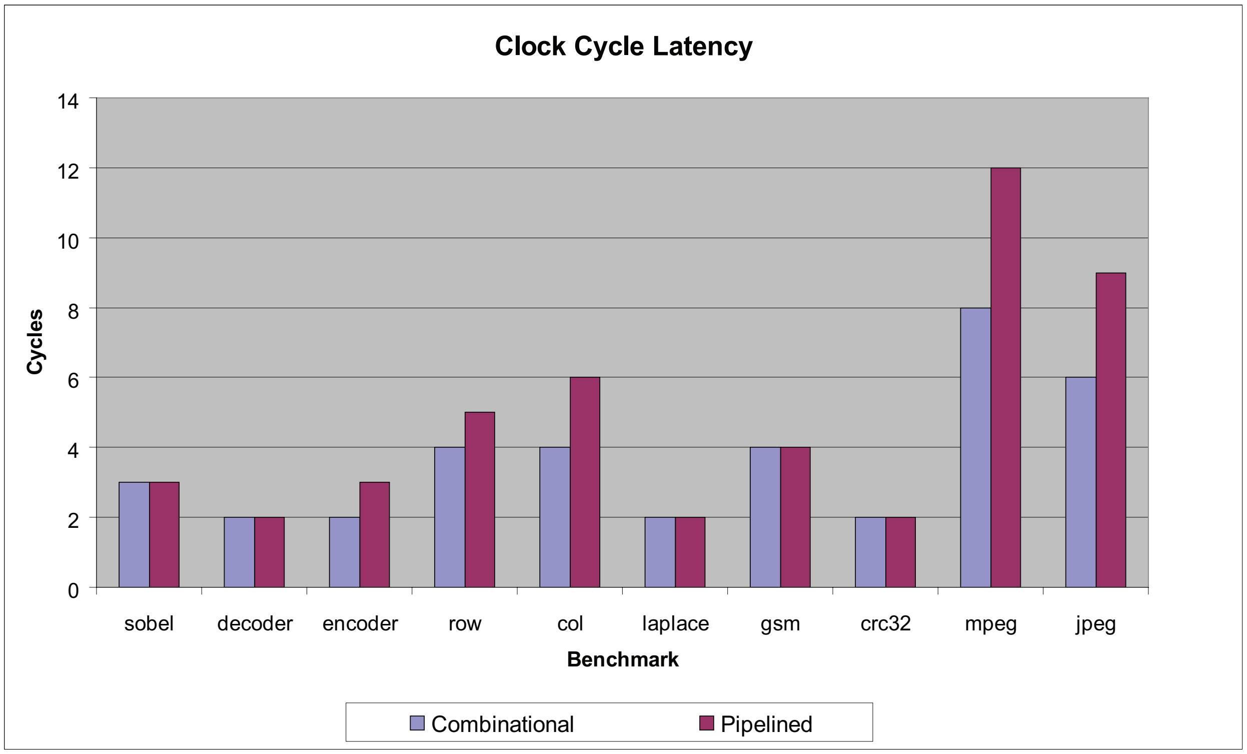


Figure 10: Clock Cycle Latencies

creased by 3.3x while the area only increased by 1.4x. The execution time metric refers to the simulation time required to completely process sets of input vectors. The throughput gains seemed to increase with the length of the critical path, resulting in more savings seen for the larger IDCT Row and Column benchmarks. The area gains did not seem to follow any specific trend. This is most likely because the number of registers added to the design is related to the locations in which the graph is partitioned. IDCT Row and Column saw the largest area increases due to pipelining. Along with having longer critical paths, their graphs are fairly wide. Wider benchmarks require a greater number of partitioning registers.

Power is defined as the amount of energy transferred per time unit. Because pipelining allows multiple sets of data to be processed simultaneously, the amount of time needed for the combinational implementation to complete the same amount of processing can be significantly longer depending on the length of the critical path. For this reason, power consumption cannot be used to achieve a fair comparison. The amount of energy used is the product of power and time, so it is

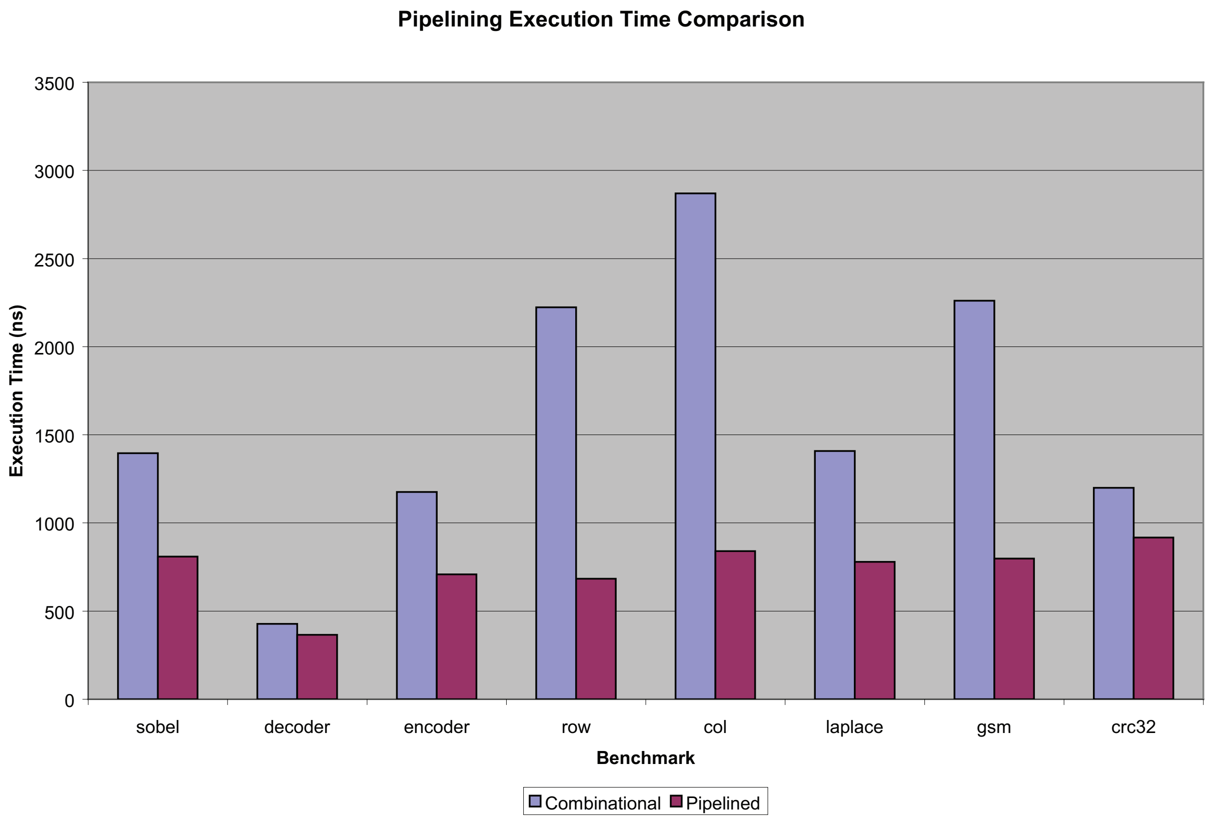


Figure 11: Execution Time Tradeoffs Due to Pipelining

used to provide a better comparison. The energy increases are shown in Figure 13. On average, the total energy consumption for the pipelined designs increased by a factor of 1.7x over the combinational designs. There was no obvious trent in the energy increases. In the combinational versions, IDCT Row consumes nearly three times as much energy as Sobel. However, the pipelined version of Sobel consumes nearly as much energy as the pipelined version of IDCT Row.

4.3 REPLICATION VS. PIPELINING

Most modern soft core processors have the ability to extend their instruction set with custom instructions utilized in working with co-processors. SuperCISC hardware functions, when imple-

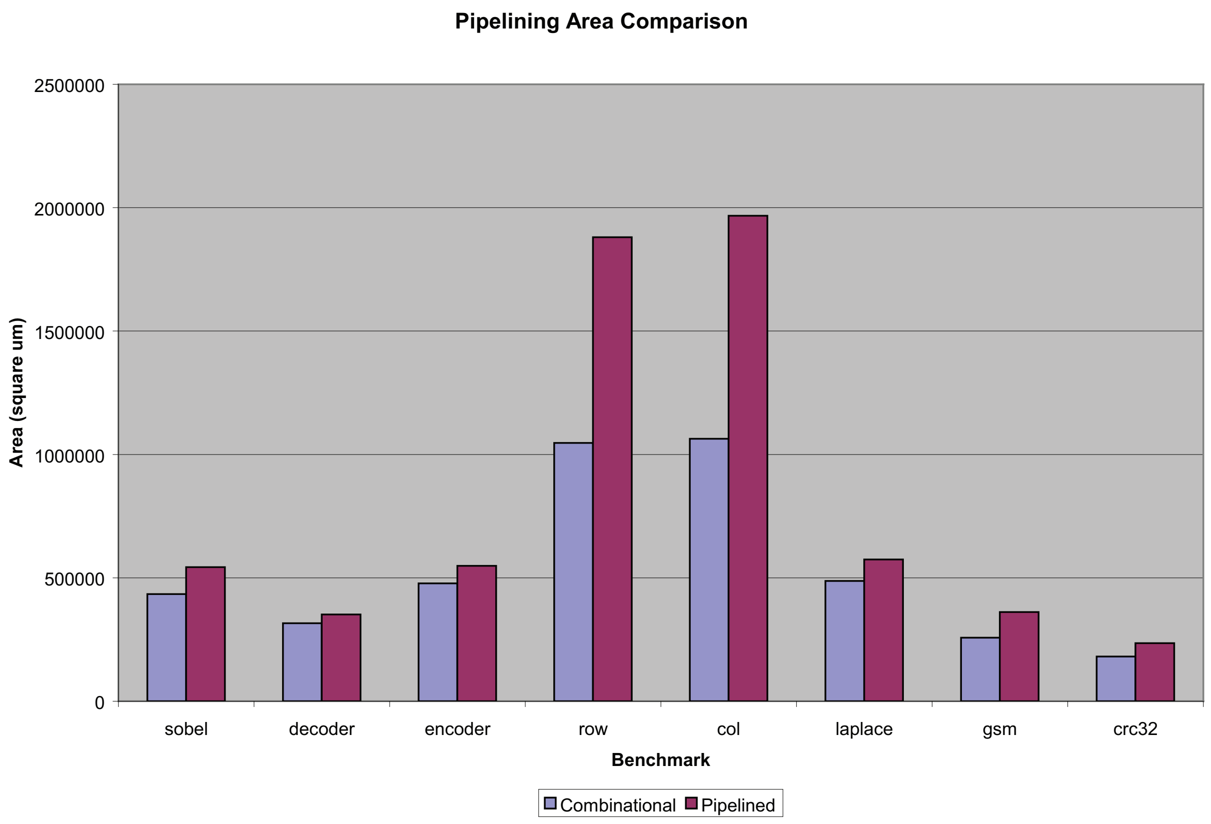


Figure 12: Area Increases Due to Pipelining

mented as custom instructions, can be tightly coupled with RISC soft core processors to provide a massively parallel processing solution. Many SuperCISC functions can be combined on a single chip to exploit an even higher degree of parallelism. Due to the iterative nature of most vector-based signal processing applications, the utilization of coprocessors has been investigated extensively [2]. Replication of hardware functions can be used to unroll time consuming loops and observe greater amounts of speedup. By implementing two instances of the same hardware function, the amount of parallelism is doubled. Unfortunately, the amount of on-chip area and power consumed is also doubled by replicating the kernel. As the size and complexity of the hardware functions increase, overcoming the area and power usage becomes a more daunting task.

Pipelining provides an efficient, scalable solution to the problems facing replication. As hardware functions are replicated, the area and power increase is linear in the number of replicated

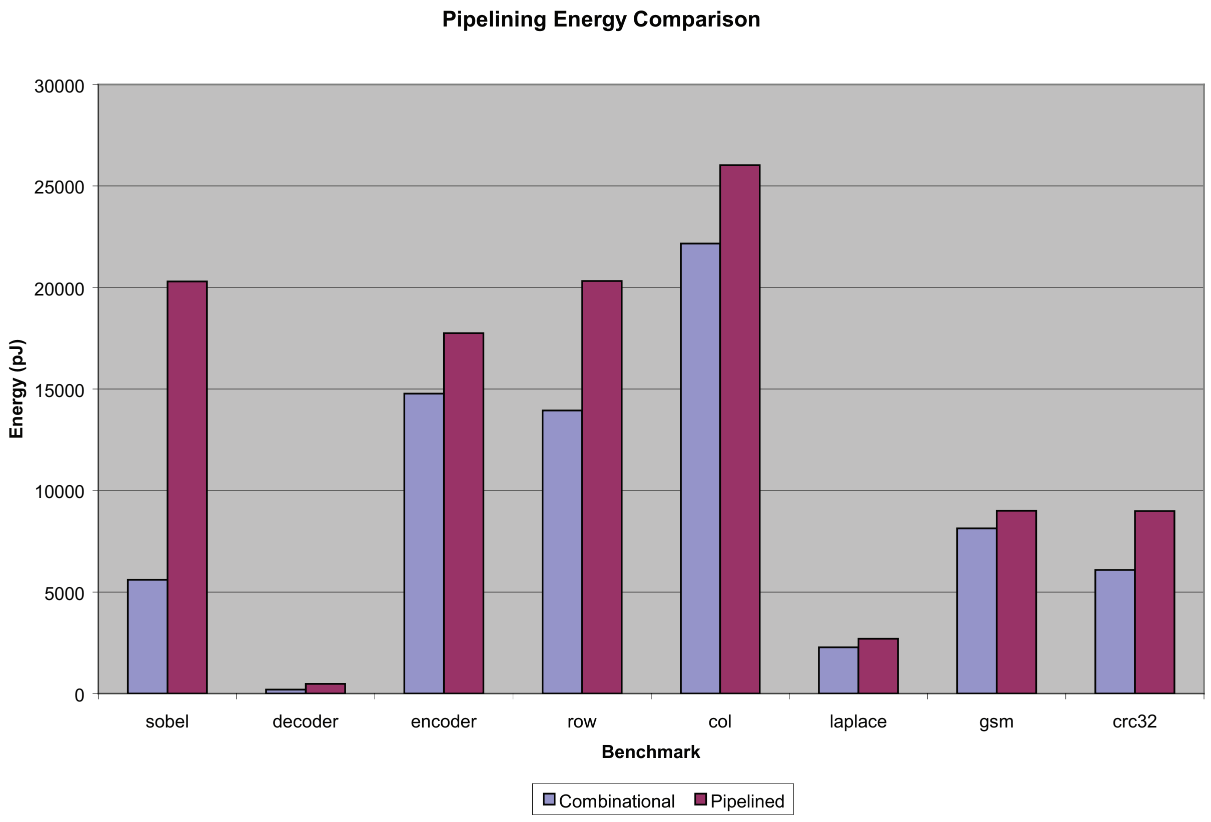


Figure 13: Energy Increases Due to Pipelining

instances. By pipelining a single instance of a hardware function, the only area and energy increases are seen from the inclusion of the pipeline registers and clock signal used to drive them. The benefits to this approach can be most easily seen in larger hardware functions. Larger hardware functions typically result in longer pipelines than those of smaller functions. Pipelined functions with *n* pipeline stages approach the throughput of functions with a replication factor of *n*. In larger applications, the number of instances needed to achieve this level of full replication places heavy demands on area and energy utilization. Figure 14 shows the increases in area and energy usage with full replication compared to an equivalent pipelined implementation. Again, a 200 MHz pipeline is used to generate the results.

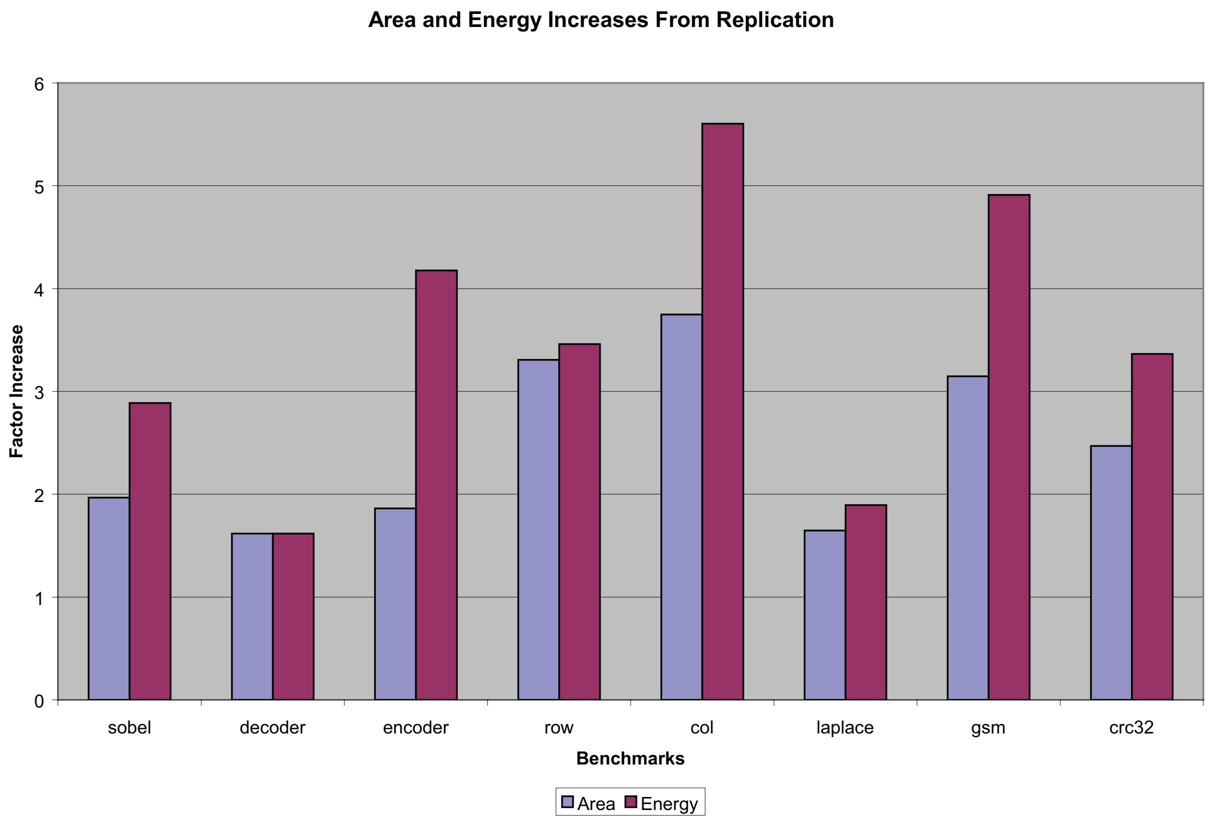


Figure 14: Area and Energy Increases for Fully Replicated Hardware Functions vs. Pipelined Equivalent

5.0 DELAY ELEMENTS

With the increasing use of portable, battery powered devices such as cellular phones, optimizing designs for performance, area, and cost is not enough. Power consumption in these devices is becoming increasingly important and must also be optimized. There are many existing techniques for reducing power consumption in digital circuits. The methodology presented here seeks to reduce total power consumption by reducing the component of event-based power caused by spurious transitions. These spurious transitions, known as glitching or switching, can be reduced by equalizing path delays throughout the circuit. Although perfect equalization is not possible, delay elements can be utilized to alleviate some of the problem.

SuperCISC hardware functions are synthesized into a standard cell representation in order to generate a hardware implementation. To be used in these functions, delay elements must also be synthesized. Delay elements have been synthesized into standard cell designs in the past, however, they are typically built from inverter chains using inverter cells present in the technology library. Inverter based delay elements can provide fairly accurate delays, but are not particularly power efficient. Thus, to include delay elements within SuperCISC designs, a delay element cell needed to be created for inclusion in the technology library. Since each cell, once implemented, has a fixed length delay, several cells were created with differing delay lengths to be used in the system.

The delay element design used to construct the cells for the library is based on a low-power CMOS thyristor based delay element structure as proposed in [41]. As the delay element controls the timing of most of the critical paths in the design, the delay element needs to be robust in terms of power supply sensitivity and temperature variations. As the thyristor based delay element [21] is current controlled rather than being voltage controlled, the design exhibits very little variation with power supply noise and temperature variation. The delay element of [41] has been slightly modified to meet the requirements of SuperCISC designs. SuperCISC designs utilize the delayed rise time to enable latches inserted into combinational paths. In order to process each new set of input vectors, the delay elements must first be reset. Having a long fall time requires the circuit to sit in an idle state while the delay elements are reset. For example, a circuit with a critical path length of 12 ns might contain a 10 ns delay element. Utilizing the 10 ns delay element in such a design would require 20 ns. This means that the design is performing no useful computation for 8 ns. In order to make delay element usage more efficient, the delay elements were adjusted to have a faster fall time. The resulting fall time is on the order of a few hundred picoseconds, as opposed to the rise time of several nanoseconds.

As only the rising edge of the input signal needs to be delayed, a single current source controls the delay on the rising edge. The delay on the falling edge has been minimized as much as possible. Figure 15 shows the timing of the *D* input and the delayed output out of the delay element. The figure shows that the output, *DELAYED_OUT*, follows the pattern of the input. The rising edge is delayed substantially, while the falling edge occurs nearly simultaneously. The slight modification to the design allows for more efficient control of the circuit. This is described in greater detail in Section 6.2.

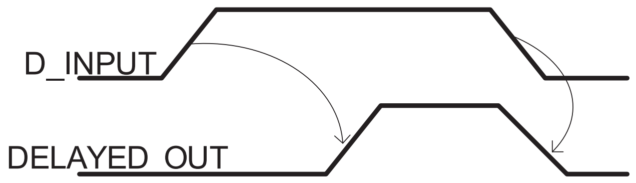


Figure 15: Delay Element Timing

5.1 POWER PROFILING

In order to gain insight into the nature of SuperCISC hardware's power consumption, the individual functional units were power profiled. The power profiling was performed using an established technique known as power macromodeling [3, 10, 25]. Power macromodeling uses three parameters (*p*, *s*, and *d*) to achieve fairly accurate power estimations for digital circuits. The *p* parameter corresponds to the average input signal probability. The signal probability corresponds to the number of 1's to appear in a binary value. The *s* represents the spatial correlation. The spatial correlation represents the probability for 0's and 1's to appear in groups in a binary value. Lastly, *d* represents the transition density. The transition density corresponds to the frequency of bit changes between two or more values [20].

Power results were generated in each of the three dimensions on 0.1 intervals. It is worth noting that there are combinations of *p*, *s*, and *d* which cannot occur. The input stimuli were generated using the Markov sequence generator described in [25]. The mean of all of the experiments is then used as the average power consumption. The power profiling results used in this paper were generated by Gayatri Mehta and published in [20].

5.2 DELAY ELEMENT INSERTION

Two algorithms were tested for inserting delay elements. The first algorithm is a brute force approach. The second algorithm takes a greedy approach to delay element insertion; attempting to place delay elements judiciously in order to achieve the biggest savings with the least amount of circuit modification. Both approaches rely on delay profiling of functional units. The static timing characteristics of the SuperCISC hardware are used to determine the size of the delay to associate with each functional unit. The algorithm used to perform static timing analysis was described in Section 3.

The brute force algorithm places delay elements everywhere possible. For each functional unit, the delay element whose delay length is the closest to the path delay of the functional unit is selected. Additionally, the delay length inserted must be smaller than the path delay of the functional unit. This is done so that performance is unaffected or only affected minimally. Delay elements are not placed on nodes that that do not require gate style implementations. For example, constant values and constant shifters. Both are implemented using wires only.

5.3 A GREEDY APPROACH

The brute force algorithm is very simple and shows interesting results. However, for large designs it may not be feasible to insert latches for every functional unit in the circuit. Thus, an algorithm was developed which attempts to gain the maximal power savings possible at the lowest cost in terms of latch insertion. The resulting algorithm takes a greedy approach to the selection of delay element insertion locations. The heuristic value is based on the amount of energy saved by adding a delay element in a specific location. In order to calculate this heuristic value, static timing information is not enough. Functional units also need to be power profiled in order to calculate energy consumption.

The algorithm begins by selecting a group of nodes in the SDFG as potential locations to insert delay elements. This group initially consists of all of the SDFG nodes which will be synthesized as power consuming functional units. This removes inputs, outputs, constants, and nodes such as sign extension units that are synthesized as wires and consume little power. The next stage of the algorithm relies closely on the concept of strict domination used for control flow analysis in compilers [31].

In a CDFG, a node *d* dominates a node *n*, if every path from the start node to *n* must go through *d*. Every node in the graph dominates itself. A node *d* strictly dominates *n* if *d* dominates *n* and *d* is not equal to *n*. Because an SDFG has no unique start node, some nodes in the graph will not be strictly dominated by any other node. The nodes that are not strictly dominated by any other node are the candidates for having delay elements inserted at their output. These nodes are referred to as the set of switch-blocking nodes. For a given switch-blocking node *s*, adding a delay element to its output will prevent all of the nodes strictly dominated by *s* from switching until the delay has expired.

Once the switch-blocking nodes have been identified, a number of them are selected for delay element latch insertion. The number of nodes to be selected is capped by a maximum value passed as a parameter to the algorithm. Each iteration of the algorithm selects one switch-blocking node, s_i, to add delay latches to based on its heuristic value, $H(s_i)$. The heuristic value is the energy consumption savings estimate by implementing a delay element on s_i.

Energy is used in the heuristic calculation because unlike power, the energy value is dependent on the amount of time that a node switches for. Two multipliers in a circuit will consume roughly the same amount of average power. However, if one of the multipliers is much deeper in the SDFG, then it will switch for a longer period of time and consume more energy. To the determine the power, each functional unit was profiled independently for power under various input stimuli as described in Section 5.1. Then the average power is stored in a table that is accessed by the heuristic. The average power for the individual functional units is shown in Table 4. The energy is determined by the product of the average power consumed and the execution delay. The power estimates shown in Table 4 were calculated using 32-bit functional units. The execution delays used for energy calculation were previously discussed and shown in Table 1.

Table 4: Functional Unit Power Estimates

Operation	**Power**
ADD	5.19 mW
AND	1.98 mW
EQU	0.81 mW
GT	4.42 mW
GTE	3.35 mW
LT	7.41 mW
LTE	3.97 mW
NE	0.94 mW
MUX	30.20 mW
MUL	23.10 mW
NOT	3.00 mW
OR	2.11 mW
SUB	5.86 mW
XOR	12.50 mW

When a delay element is added for node s_i, a number of nodes will be prevented from switching. The nodes strictly dominated by s_i will be blocked by default. However, additional nodes may also become blocked through interactions with delay elements inserted for previous s_i values. Therefore, $H(s_i)$ is calculated as the sum of the energy consumed by the set of strictly dominated nodes as well as the set of newly blocked nodes.

Figure 16 shows a portion of the SDFG for the Sobel benchmark. The switch-blocking nodes are shown in light blue. The yellow nodes below each switch-blocking node represent the nodes that are strictly dominated. The magenta colored nodes will switch unless latches are placed on both of the switch-blocking nodes. This illustrates how the delay elements interact. Adding latches to a single node will block two or three nodes from switching, but adding both latches will stop eight nodes from switching.

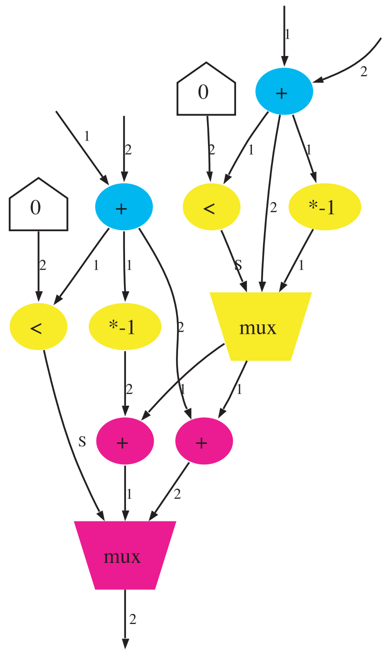


Figure 16: Greedy Algorithm Example

5.3.1 Optimization

Once the greedy algorithm has selected the locations to place delay elements, an optimization algorithm attempts remove redundant delay elements. The optimizer works only with the previously selected switch- blocking nodes and the group of blocked nodes. Each switch-blocking node and the nodes it immediately dominates are treated as a group, with the switch-blocking node referred to as the head node and all other nodes referred to as the children nodes. The algorithm begins by selecting the first two groups and merging them together into one larger group. The new larger group now has two head nodes and the children nodes from the two smaller groups. As the two smaller groups are merged together, additional children may also be added to the larger group, due to the delay element interaction principle explained previously, and illustrated in Figure 16. This process is repeated until all of the groups have been merged into a single large group.

Once all of the nodes have been collected into a single group, the optimizing algorithm begins to manipulate them in an effort to improve the resulting implementation. The algorithm begins by examining all of the head nodes in the group. After all of the merges have taken place, it is possible that one or more of the head nodes might have been demoted. A head node is considered to be demoted if, through the course of merging, it falls into the category of being blocked through the interactions of delay elements above it in the SDFG. Demoted head nodes would only result in redundant delay elements and latches being inserted into the design. However, some caution must be exercised when demoting nodes. For example, the crc32 benchmark is composed of essentially one long path. Inserting a delay element near the top of crc32 would prevent any other delay elements from being inserted because the remaining head nodes would be considered to be demoted. In order to counteract this, a threshold is enforced on demoting. Taking the critical path of the design into consideration, a node cannot demote another node if it is 50% or more of the critical path length away. By allowing for head nodes to be demoted, it is possible for the number of delay elements implemented in the final design to be smaller than the parameter passed to the greedy algorithm. For the Sobel benchmark, calling the greedy algorithm with a parameter of two, three, or four will result in the same delay element placements.

After the removal of redundant head nodes, the algorithm attempts to maximize the benefit of the remaining delays. To do this, the immediate children are inspected for each head node. The immediate children are the set of nodes in the SDFG that are in the adjacency list of the head node. The critical parent for each immediate child node is calculated. The critical parent is the parent node whose output stabilizes at the latest time. The critical parent with the earliest arrival time among all of the immediate children dictates the delay length to be inserted at the output of the head node. The benefit of this approach is seen when the particular head node is not the critical parent. For example, if the head node is ready at time 4 ns, but the earliest arrival time amongst the critical parents is 10 ns, then the delay amount can be increased by 6 ns. This increase in the delay prevents six nanoseconds of switching from occurring, while having no effect on execution time.

Figure 17 shows the resulting implementation for the Sobel benchmark when the greedy algorithm is applied with a parameter of three but with optimizations turned off. Two adders and a multiplexer have been selected as locations to place delay elements. Figure 18 shows the result for the same scenario, but with optimizations applied. In this case, the delay element inserted on the multiplexer has been considered redundant and thus removed. In this example, the two adders are the earliest arriving critical parents of all of their immediate children, prohibiting an increase in delay lengths.

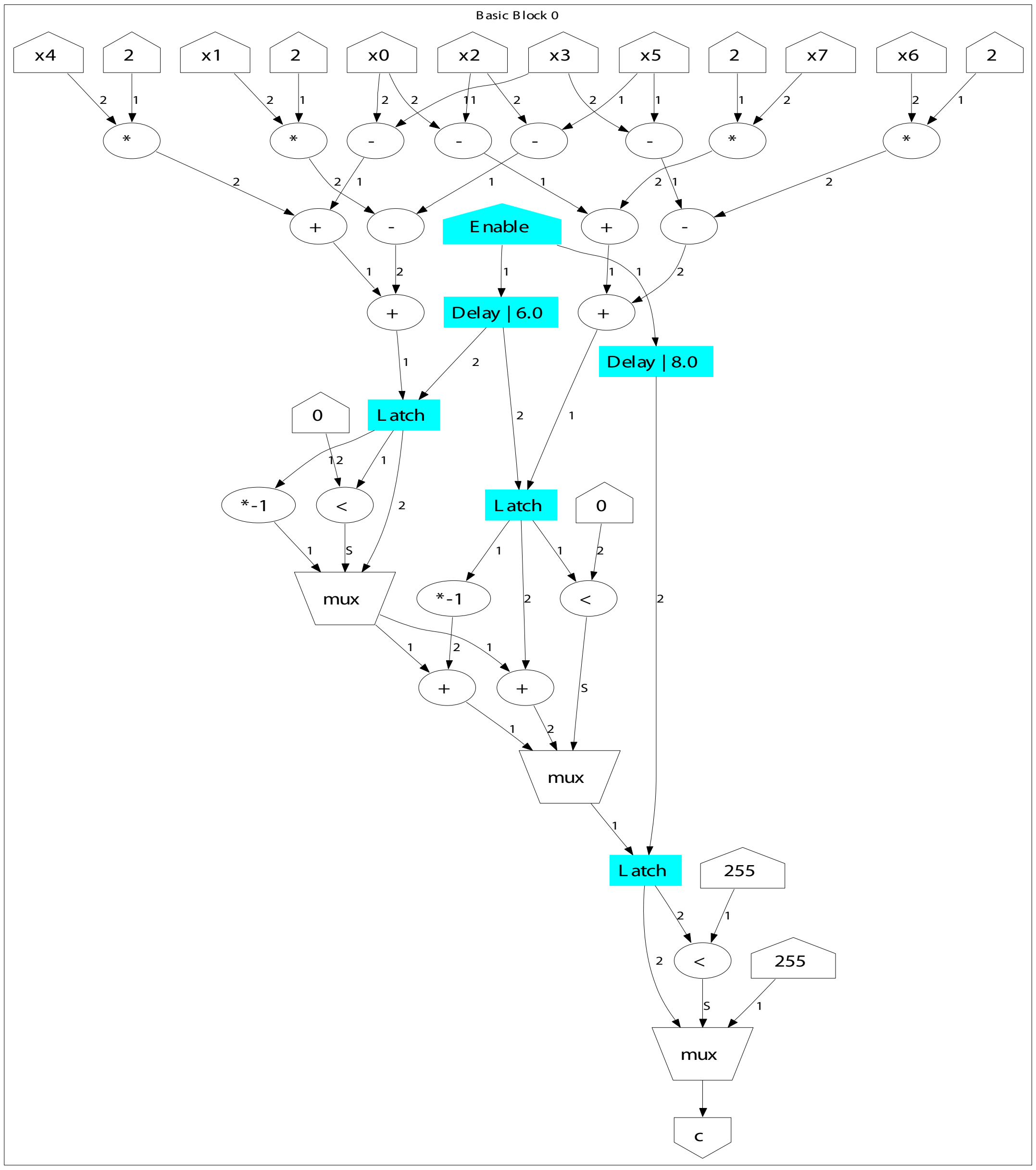


Figure 17: Greedy Algorithm Output With No Optimizations

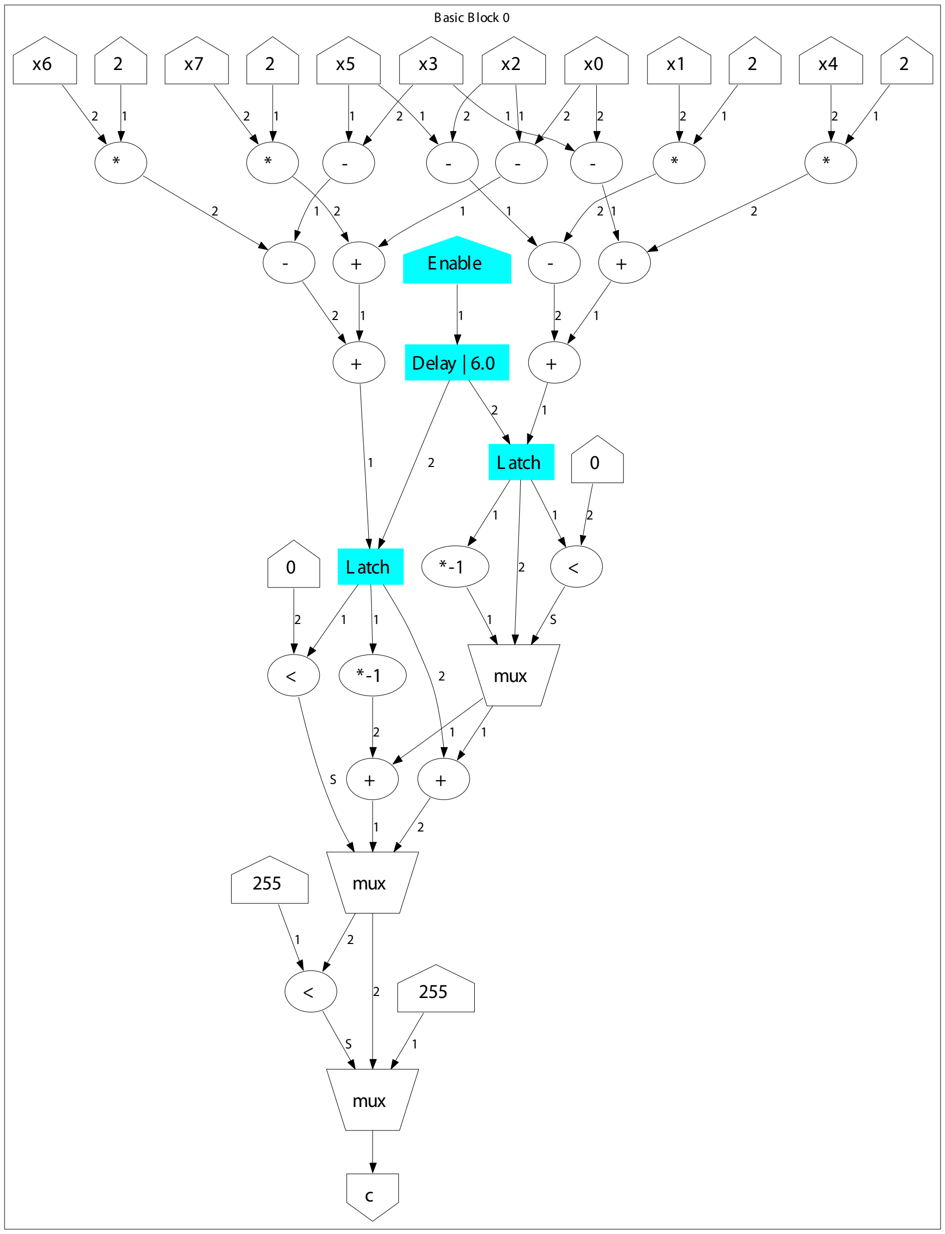


Figure 18: Greedy Algorithm Output After Applying Optimizations

6.0 DELAY ELEMENT TOOL FLOW

The design flow discussed thus far has resided within the SUIF infrastructure. C code is profiled and sent through the SuperCISC compiler to create a hardware description of the software execution kernels. At this point, the remainder of the tool chain consists of a number of commercial EDA tools. At the end of this flow, the designer will have a netlist which has been technology mapped, simulated at the gate level, and profiled for power consumption. In order to achieve this, the existing tool flow needs to be augmented to accommodate delay elements and the issues that arise from using them. The following sections will go into detail about the synthesis, simulation, and power estimation design stages as well as the potential hurdles caused by delay elements in each.

6.1 SYNTHESIS

The SuperCISC compiler generates a VHDL description of a circuit. The next step in the design process is to synthesize the design into a technology mapped netlist. Synthesis is performed using the Synopsys Design Compiler tool. The synthesis process for the delay element based designs is more complex than a standard combinational or clock based design. Each delay element is initially created as a VHDL entity, with an architecture which acts as a wrapper for the actual delay element standard cell. An example of this is shown in Figure 19. In this particular example, the wrapper encloses a four nanosecond delay element cell, defined as *U0*. The *a* input and *yn* output of the delay element cell are mapped to the corresponding *A* input and *C* output of the VHDL entity.

The VHDL description is then passed off to Design Compiler to perform synthesis. Design Compiler has built in scripting capabilities. The design is then synthesized using Design Compiler

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity delay_4_0 is
port (
  signal A : in std_logic;
  signal C : out std_logic
);
end delay_4_0;

architecture cell of delay_4_0 is
  component cell_delay_4_0
    port ( a : in std_logic; yn : out std_logic );
  end component;
begin
  U0 : cell_delay_4_0 port map ( a => A, yn => C );
end cell;
```

Figure 19: Four Nanosecond Delay Element VHDL Description

scripts. The code snippet shown in Figure 20 is taken from one of the synthesis scripts. This particular script sets up the synthesis source and target libraries and then synthesizes the four nanosecond delay element. Two of the commands in the script are noteworthy. The first is the *set_max_delay* command. This command specifies the desired delay through a design. In this case, the four nanosecond delay element is constrained to have a delay of only 3.75 ns. The delay is reduced in an effort to eliminate or minimize the timing effects of the delay element on the rest of the design. The second command to note is the *set_combinational_type* command. This command tells the synthesis tool to use the *cell_delay_4_0* standard cell to implement the *U0* instance of the delay element VHDL model. Failing to specify this allows the tool to implement the design as it sees fit as long as the timing constraint is respected. This often results in the delay elements not being utilized at all.

In the actual tool flow, a single script is used to set up the libraries and synthesize all of the delay elements. This single script is then sourced by each of the benchmarks' synthesis scripts. Not only does this allow for code reuse, but it also simplifies the process of switching between different technology libraries. A similar common script with the delay elements omitted is used to synthesize the combinational designs.

```
synthesis_directory = /vol/delay_elements/
ibm_library         = synthesis_directory + typical.db
delay_library       = synthesis_directory + delay.db
delay_vhd           = synthesis_directory + delay.vhd

link_library = {delay_library} + {ibm_library}
target_library = {delay_library} + {ibm_library}
hdlin_enable_presto_for_vhdl = true

analyze -format vhdl delay_vhd
elaborate delay_4_0

current_design delay_4_0
set_combinational_type -replacement_gate cell_delay_4_0 U0
set_max_delay 3.75 -from {A} -to {C}
set_wire_load ibm13_wl10 -library typical
link
compile -map_effort medium

vhdlout_architecture_name = "SYN"
vhdlout_use_packages = {"IEEE.std_logic_1164.all",
                        "IEEE.std_logic_arith.all"}
```

Figure 20: Design Compiler Script to Include a Delay Element

6.1.1 Constraining the Design

When synthesizing a design using Design Compiler, it is necessary to provide constraints to the synthesis tool in order to realize a circuit with the timing characteristics calculated through static timing analysis. By omitting timing constraints, the designer will receive a circuit with the expected functionality, but whose critical path length and other timing properties might be quite different than expected. Design Compiler provides commands for specifying timing constraints across portions of a design [34]. Constraining a purely combinational SuperCISC hardware function in Design Compiler is trivial. By providing the critical path length calculated via static timing analysis, a single command can be used to constrain the entire design.

After the design has been augmented with delay elements and their corresponding latches, it is necessary to maintain the functional behavior of the original combinational circuit. Inserting delay elements drastically changes the way in which timing constraints are applied to the design.

The problem arises from the insertion of the latches, not the delay elements themselves. In terms of timing calculations, the synthesis tool treats level sensitive latches in a similar fashion to edge triggered registers. This means that user defined timing constraints are nullified upon reaching one of the delay latches. In order to account for this, more elaborate constraints are required. Instead of applying a single constraint to the entire design, individual constraints are applied to each of the latches. For large designs with many latches, defining all of the constraints manually, is not an attractive solution. Instead, the static timing analyzer can be used to generate a file containing all of the constraints. This timing file is then imported into Design Compiler by the primary synthesis script. An example timing constraint script generated by the static timing analyzer for the Sobel benchmark is shown in Figure 21. In this example, a delay of 10.4 nanoseconds is defined from the inputs to the outputs of the design. This delay represents the critical path length calculated through static timing. Applying this constraint ensures that all paths through the design are constrained. Next, the paths containing latches are constrained individually. A delay of 6.45 ns is defined from the design's inputs to the outputs of the latches, *I79* and *I80*. In addition, a delay of 3.95 ns is defined from the latches' outputs to the circuit's outputs. In this particular example, both of the latches are located at a depth of 6.45 ns in the design's timing paths. The two numbers are derived from the individual paths, and are not related. The remaining 3.95 ns is the difference of the critical path length and the amount of delay consumed up to the point of the latches.

```
set_max_delay 10.4 -from {x0,x1,x2,x3,x4,x5,x6,x7}
                   -to {c_out}

set_max_delay 6.45 -to {I79/C*/Q}
set_max_delay 3.95 -from {I79/C*/Q}
set_max_delay 6.45 -to {I80/C*/Q}
set_max_delay 3.95 -from {I80/C*/Q}
```

Figure 21: Design Compiler Timing Constraint Script

Once synthesis has been completed, the design's timing and area statistics were analyzed. After delay element insertion, the timing results were met as expected. However, an unexpected trend was observed upon further analysis of the area results. When applying the brute force insertion algorithm, the size of the designs increased by 7% on average. In some cases, the designs actually became smaller than their combinational equivalents. Logically, this should not happen because the only difference between the designs are the additional delay elements and latches; thus the designs should always get larger when adding delay elements.

To gain further insight into the unexpected decrease in area, the same timing constraints were applied to the combinational versions of the hardware functions. A similar general trend was observed when constraining the combinational designs. An average area decrease of 12% was seen in the constrained combinational designs. When compared with the constrained delay element designs, the area increase due to the additional hardware was apparent. Also, in every case, the constrained combinational circuits were smaller in terms of area, as expected. Figure 22 shows the area consumption trends for the combinational, constrained combinational, and delay element implementations. The modified timing constraints in the constrained designs are inserted in order to equalize path delays between different points in a design. These modified path delays effect the way in which the synthesis tool implements certain functional units. For example, a multiplier in a combinational design might be synthesized in such a way that area and performance optimization is more or less balanced. However, once constrained, that same multiplier may have significantly more slack time to work with. In this case, the tool can focus more on optimizing the multiplier for area.

6.1.2 Delay Element Standard Cell Library

Different versions of the delay element were created for a range of delays. A standard cell was created for each of the delay values ranging from 4 ns to 30 ns in time intervals of 2 ns. For each delay value, a cell was defined in a library of delay elements. Each delay cell is capable of driving the enable signals of multiple latches. The functional units in the hardware functions are at most 32 bits wide. Accordingly, the delay elements can be thought of as controlling sets of latches in multiples of 32. The input capacitance of a latch was estimated using the data given in the standard cell library for the 0.13μm IBM CMOS process. Each delay element was characterized with drive strengths of 32, 64, 128, 256, and 512-bit latches. The physical design of the delay elements was performed by Gerold Joseph. Cadence Virtuoso XL was used to implement the designs. Mentor Graphics Calibre was used to extract parasitic capacitances. Each cell was found to have an area

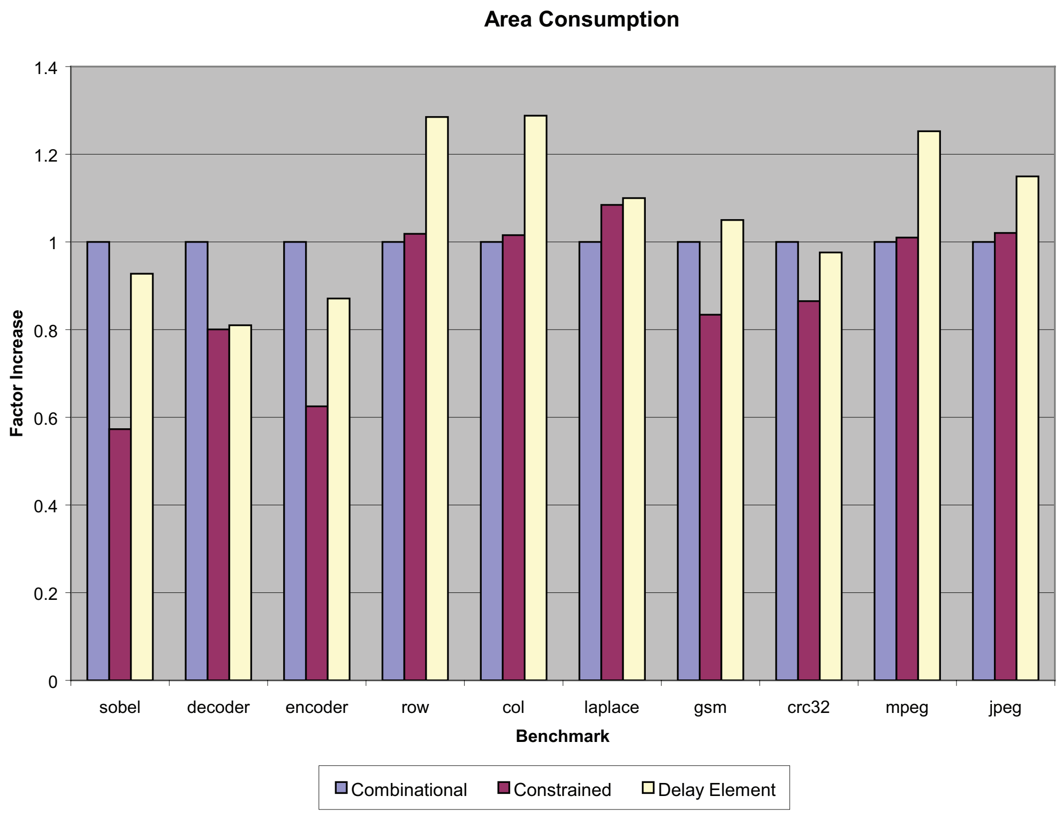


Figure 22: Area Consumption for Combinational, Constrained Combinational, and Delay Element Designs

of approximately 172.8 square microns. Unfortunately, due to the variations in drive strengths, creating designs with an exact delay length was not possible. The result is a set of delay elements with actual delays that are less than the advertised delay lengths. As long as the actual delay is not greater than advertised, no problems are created in the delay element based designs. Unfortunately, because the delay values are not ideal, more glitches are able to pass through the latches, increasing the overall power consumption of the design.

Once the data for each delay element circuit was collected, a delay element cell was created in the Synopsys Liberty file format. The Liberty format characterizes a standard cell in terms of functionality, area requirements, and energy consumption. These properties can then be used later in the design process for simulation and power estimation The Liberty format is described in more detail in Section 6.1.3.

6.1.3 The Liberty File Format

The Liberty format is a file format created by Synopsys for specifying technology libraries. The Liberty format is commonly used in the EDA industry, and is supported by over 100 semiconductor vendors and more than 60 EDA vendors [34]. Liberty is used in over 75 industrial strength commercial tools. Liberty, which is available under standard open-source terms, has been available for more than 15 years. In 2000, Liberty became the one of the first EDA standards available as open source [34]. The format is constantly evolving to address the latest requirements in modeling. The generated technology libraries are used in several stages of the design process. Synthesis and power estimation make extensive use of these libraries. The characteristics described in these libraries can also be used to generate simulation models.

The Liberty format is used to model nearly all aspects of a standard cell based circuit. Some of the properties described in a Liberty style model include, but are not limited to, timing, voltage, current, leakage power, capacitive loads, wire loads, operating conditions, and logic functionality. Figure 23 illustrates the Liberty description of a six nanosecond delay element. The large energy and delay tables have been omitted for brevity. The delay and energy tables are seven by seven tables populated with data from Spice simulations at different drive strengths and load capacitances. The delay element cell's name is *cell_delay_6_0* and has an area of roughly 173 square microns. Next, the characteristics of the pins are described. The *a* pin is described as an input pin with a defined capacitance. The *yn* output pin's model is far more complicated. Output pin models describe the logic function, energy consumption and timing delays in terms of both rising and falling logic transitions. In the case of a delay element, the logic function for *yn* is simply the input *a*. The timing model consists of the cell's delay and transition time. For a delay element cell, the *cell_rise* property will describe the length of the delay. Finally, the cell's leakage power is described for both logic levels. Most tools take the maximum value of the two as the cell's leakage power. The capacitance, energy, timing, and leakage power units are defined elsewhere in the Liberty file.

```
cell(cell_delay_6_0) {
   area : 172.800000;
   pin(a) {
      direction : input;
      capacitance : 0.005427;
   }
   pin(yn) {
      direction : output;
      capacitance : 0.0;
      function : "a";
      internal_power() {
         related_pin : "a";
         rise_power(energy_template_7x7)
            ...
         fall_power(energy_template_7x7)
            ...
      }
      timing() {
         related_pin : "a";
         cell_rise(delay_template_7x7)
            ...
         rise_transition(delay_template_7x7)
            ...
         cell_fall(delay_template_7x7)
            ...
         fall_transition(delay_template_7x7)
            ...
      }
      max_capacitance : 1.023840;
   }
   cell_leakage_power : 1660000;
   leakage_power() {
      when :"!a";
      value : 1660000;
   }
   leakage_power() {
      when :"a";
      value : 1660000;
   }
}
```

Figure 23: Liberty Model of a Six Nanosecond Delay Element

6.2 SIMULATION

In order to simulate ASIC cells, a model must first be created for each standard cell. In the past, these models were created as proprietary implementations. These proprietary models locked designers into a specific set of commercial tools. In order to accelerate the development of high quality ASIC simulation libraries, the IEEE created the VITAL (VHDL Initiative Towards ASIC Libraries) 1076 standard [5]. VITAL models describe all of the timing semantics and functionality of ASIC cells entirely in VHDL. Timing semantics, including glitching characteristics, are implemented as generics in the VITAL model. The actual values passed to these generics are defined in the entity declarations of each standard cell. Design Compiler can be used to generate IEEE compliant VITAL ASIC models based on a Liberty format technology library [34]. Figure 24 shows the VITAL entity for a four nanosecond delay element. The entity defines two generic values, *tpd_a_yn* and *tipd_a*. The *tipd_a* values define the wire delay associated with the *a* input of the delay element. The *tpd_a_yn* values define the delay from the *a* input to the *yn* output for both rise and fall transitions. The first value of 3.134 ns represents the rise time, while the second value of 0.303 ns corresponds to the fall time. For a delay element, the rise time is the transition of interest. Note that the rise delay defined in the entity is only 3.134 ns, even though the delay element length is advertised as four nanoseconds. Although this smaller delay will allow the circuit to switch for a longer time, it is necessary in order to assure that the delay elements will have a minimal performance impact on the design. The fall transition time of 0.303 ns represents the time needed to reset the delay element when a new set of inputs are applied. Figure 25 shows the VITAL architecture corresponding to the entity defined in Figure 24. The architecture maps each cell's ports, generics, and delays using various VITAL data types. The *VitalPathDelay01* tracks the *yn* output. A variable of type *VitalGlitchDataType* is used to process the switching characteristics of *yn*.

The designs are simulated using Mentor Graphics' ModelSim. The detailed step-by-step results of the simulation are written to a Value Change Dump (VCD) file. This file captures the signal switching behavior of the design, which is later used to perform power estimation. The simulations are driven by ModelSim "DO" files. In the combinational circuits, the DO files provide a new set of input stimuli, and advance the simulation until the output values stabilize. However, the designs using delay elements require the enable signal of the delay elements to be controlled.

```
----- CELL cell_delay_4_0 -----
library IEEE;
use IEEE.STD_LOGIC_1164.all;
library IEEE;
use IEEE.VITAL_Timing.all;

-- entity declaration --
entity cell_delay_4_0 is
   generic(
      TimingChecksOn: Boolean    := True;
      InstancePath: STRING       := "*";
      Xon: Boolean               := False;
      MsgOn: Boolean             := True;
      tpd_a_yn                   : VitalDelayType01 := (3.134 ns, 0.303 ns);
      tipd_a                     : VitalDelayType01 := (0.000 ns, 0.000 ns)
   );

   port(
      a                          : in    STD_ULOGIC;
      yn                         : out   STD_ULOGIC
   );
attribute VITAL_LEVEL0 of cell_delay_4_0 : entity is TRUE;
end cell_delay_4_0;
```

Figure 24: Four Nanosecond Delay Element VITAL Entity

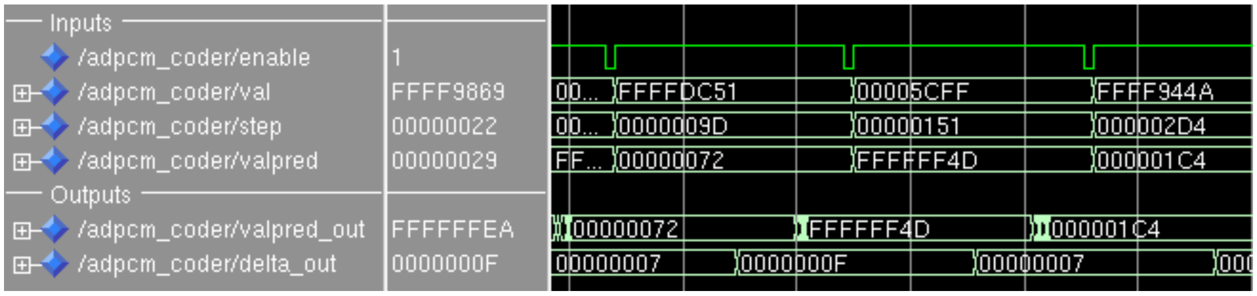
For simulation purposes, the enable signal is treated as a clock signal, but with an unbalanced duty cycle. The enable signal is asserted each time a new set of input stimuli is provided. The signal remains high for the majority of the input data set's simulation time. Finally, the enable signal is forced to a low logic level in order to reset all of the delay elements in the design.

The fast reset time of the delay elements illustrated in Figure 15 allows for long delays to be used in the designs without suffering the performance hit of sitting idle while the delay elements reset. Figure 26(a) illustrates the manipulation of the enable control signal, and the corresponding outputs for the ADPCM Encoder benchmark. When comparing the switching of the outputs with those of the combinational circuit, shown in Figure 26(b), the effects of the delay elements can be seen. The simulation using delay elements shows that there are a fewer number of glitch transitions. Figure 26(a) and Figure 26(b) only illustrate the glitching occurring at the outputs. In fact, similar switching behavior would be occurring in each of the design's internal nodes which are not shown

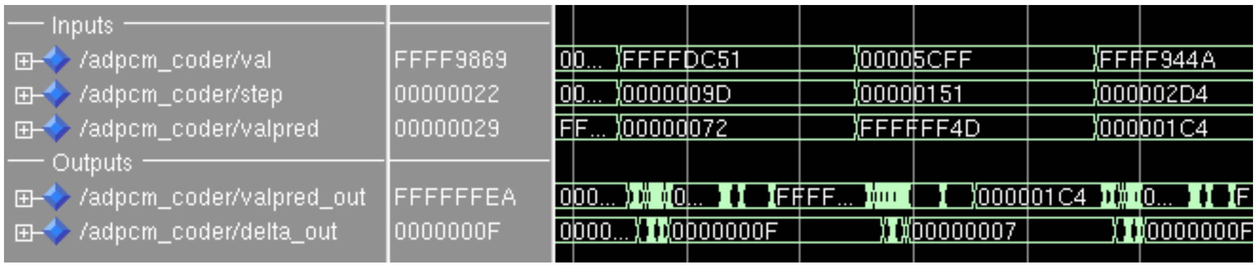
```
-- architecture body --
library IEEE;
use IEEE.VITAL_Primitives.all;

architecture VITAL of cell_delay_4_0 is
   attribute VITAL_LEVEL1 of VITAL : architecture is TRUE;
   SIGNAL a_ipd  : STD_ULOGIC := 'X';
begin
   --  INPUT PATH DELAYs
   WireDelay : block
   begin
      VitalWireDelay (a_ipd, a, tipd_a);
   end block;

   --  BEHAVIOR SECTION
   VITALBehavior : process (a_ipd)

   -- functionality results
   VARIABLE Results : STD_LOGIC_VECTOR(1 to 1) := (others => 'X');
   ALIAS yn_zd : STD_LOGIC is Results(1);

   -- output glitch detection variables
   VARIABLE yn_GlitchData : VitalGlitchDataType;

   begin
      --  Functionality Section
      yn_zd            := TO_X01(a_ipd);

      --  Path Delay Section
      VitalPathDelay01 (
       OutSignal       => yn,
       GlitchData      => yn_GlitchData,
       OutSignalName   => "yn",
       OutTemp         => yn_zd,
       Paths           => (0 => (a_ipd'last_event, tpd_a_yn, TRUE)),
       Mode            => OnDetect,
       Xon             => Xon,
       MsgOn           => MsgOn,
       MsgSeverity     => WARNING
      );
end process;
end VITAL;
```

Figure 25: Four Nanosecond Delay Element VITAL Architecture

here. The simulations also illustrate the effects of delay elements on output availability. The two simulations are shown over the same time period with the same sets of input vectors. While both of the designs meet the synthesis timing constraints, the outputs of the combinational simulation stabilize before those of the delay element based simulation. This is because the delay elements prevent the data from propagating through the design as quickly as possible.


Inputs
/adpcm_coder/enable 1
/adpcm_coder/val FFFF9869
/adpcm_coder/step 00000022
/adpcm_coder/valpred 00000029
Outputs
/adpcm_coder/valpred_out FFFFFFEA
/adpcm_coder/delta_out 0000000F
00... FFFFDC51 00005CFF FFFF944A
00... 0000009D 00000151 000002D4
FF... 00000072 FFFFFF4D 000001C4
00000072 FFFFFF4D 000001C4
00000007 0000000F 00000007

(a) With a Delay Element.


Inputs
/adpcm_coder/val FFFF9869
/adpcm_coder/step 00000022
/adpcm_coder/valpred 00000029
Outputs
/adpcm_coder/valpred_out FFFFFFEA
/adpcm_coder/delta_out 0000000F
00... FFFFDC51 00005CFF FFFF944A
00... 0000009D 00000151 000002D4
FF... 00000072 FFFFFF4D 000001C4
000... 0... FFFF... 000001C4 0...
0000... 0000000F 00000007 0000000F

(b) Without a Delay Element.

Figure 26: Simulation Comparison With and Without Delay Elements

6.3 POWER ESTIMATION

Once simulation is completed, a power estimation for the design is performed. Synopsys' PrimePower tool is used to perform the power estimation. PrimePower is capable of performing statistical activity based power analysis and event- based power analysis. Event-based power analysis provides extremely accurate power analysis with respect to time. The power estimations provided from PrimePower are generally accurate to within 10% of Spice simulations [8]. PrimePower's event-based estimations are based on gate-level simulation activity [34]. PrimePower models a design based on the net capacitances and cell level power characteristics in the technology library, as well as the circuit's connectivity defined in the netlist. PrimePower utilizes the same technology library that Design Compiler uses during synthesis. In addition to the circuit model, the user must also generate switching activity for the design. PrimePower supports several different models of switching activity input. These models range from no defined switching activity to extremely detailed gate level activity. The most accurate power estimations result from of the gate level simulation model. In the delay element tool flow, the activity data is provided via the VCD file generated during simulation.

Once the technology library, VHDL netlist, and VCD file have been provided, PrimePower performs the actual power analysis. First, the transition times for the pins in the design are determined based on the wire capacitance and the connectivity of the design. Next, each node in the design is analyzed to determine the state and path dependent switching properties. Finally, the power consumption for all of the leaf nodes in the design are summed to calculate the total power. The power consumption for each leaf node cell is obtained from the power model in the technology library. This value is calculated for each activity in the VCD file. Once power analysis has been completed, the average and peak power statistics are determined from the power profile.

Based on the results of power analysis, PrimePower reports several types of power consumption. The two broad categories of power consumption are static and dynamic power. The following definitions of static and dynamic power types as well as the ways in which PrimePower calculates each of their values are taken from the PrimePower manual [34]. The static power is the power dissipated by a cell that is inactive. Static power is often referred to as leakage power. Static power consumption comes from two main sources. The first is source-to-drain subthreshold leakage. This results from reduced threshold voltages that stop the cell from turning completely off. This is the largest contributor to static power consumption. The second source of static power comes from current leakage between the substrate and diffusion layers. PrimePower obtains the leakage power for each cell from the technology library. The total static power consumption for a design is calculated by summing the static power of each of its leaf nodes. Static power is typically a very small percentage of the overall power consumption. However, as transistor sizes shrink, leakage power becomes a larger factor.

The primary cause of power consumption comes from dynamic power. Dynamic power is dissipated through circuit activity. Dynamic power is consumed in a cell when there is a voltage change at any of its inputs. An input's voltage level can change slightly without causing a change in its logic level. Because dynamic power consumption is driven by voltage level changes and not logic level changes, it is possible for dynamic power to be consumed even if a new set of input stimuli are not provided. Dynamic power can be further subdivided into internal power and switching power. Internal power is the power dissipated within a cell. Internal power includes the short circuit power consumption which is dissipated during the time that there is a short circuit between the *P* and *N* transistors of a gate. A cell's short circuit power consumption is a byproduct of its transition time. Simply put, fast transitions consume less short circuit power than long ones. PrimePower calculates internal power based on equations which are based on the cell type and characteristics. For simple cells like inverters, the internal energy is the sum of the rise energy and fall energy as defined in the technology library. The internal energy is then divided over the length of the signal's transition pulse to determine the internal power.

The other aspect of dynamic power is switching power. Switching power is dissipated through the charging and discharging of load capacitances at the output of a cell. Logic transitions are the source of switching power consumption. Each time a cell undergoes a logic transition, switching power is dissipated. The more frequently logic transitions occur, the greater the amount of switching power consumed. By adding delay elements to a design, the number of logic transitions can be reduced, leading to a reduction in switching power. PrimePower calculates switching power based on Eq. (6.1) [34], where C_i is the load capacitance of net *i* and V_{dd} is the supply voltage. The sum of the switching power and internal power is reported as the dynamic power. The sum of the dynamic power and static power is reported by PrimePower as the total power consumption of the design.

$$P_s = \Sigma C_i V_{dd}^2 \tag{6.1}$$

7.0 RESULTS

7.1 BENCHMARK SUITE

In order to evaluate the success of the techniques described in this paper, a number of benchmark designs were created. The selected benchmarks come from image and signal processing applications, and represent the execution kernels of their application. Benchmarks of various sizes and code structures were selected in order to test the tools across a variety scenarios. The properties of the benchmarks are observed in an attempt to find trends in the results.

- Sobel - Sobel is an edge detection algorithm. Edges are detected by computing the gradient of 3x3 blocks of pixels. Sobel is comprised of nine basic blocks with three control structures.

- ADPCM Decoder - Adaptive Differential Pulse Code Modulation (ADPCM) Decoder is used in audio and video compression algorithms. Decoder is one of the smaller benchmarks in terms of size, power consumption, and critical path length. Decoder is made up of 18 basic blocks with 8 control structures.
- ADPCM Encoder - ADPCM Encoder is the inverse function of Decoder. Both are part of the Mediabench suite. Encoder consumes the most power out of the smaller set of benchmarks, and is middle of the pack in terms of area. Encoder's source code contains many branches. Encoder is made up of 19 basic blocks with 8 control structures.

- IDCT Row - IDCT Row performs a row wise decomposition of an inverse discrete cosine transformation (IDCT) from an MPEG II decoder. Row and Column are also both from the Mediabench suite. Row is composed of a single basic block with no control structures.

- IDCT Column - IDCT Column performs the column wise decomposition similar to Row. Row and Column are the largest of the smaller set of benchmarks. Like Row, Column is comprised of one basic block and no branching in its source code.
- Laplace - Laplace is another edge detection algorithm. Laplace computes the second derivative of 5x5 blocks of pixels. The SDFG of Laplace forms an inverse tree structure. Laplace contains five basic blocks and two control structures.
- GSM - GSM is used for channel encoding in wireless communication. It is one of the most used cell phone standards in the world. GSM is fairly small in terms of area, but has one of the longer critical paths. GSM is made up of 25 basic blocks and 8 control structures. GSM contains the most complicated control structures, including nested *if-else* statements.
- CRC32 - CRC32 is the table initialization of a cyclic redundancy check. CRC is fairly simple although it contains 25 basic blocks and 8 *if* statements.
- MPEG - The MPEG benchmark is IDCT Row and Column unrolled. JPEG and MPEG comprise the set of big benchmarks. MPEG has the longest critical path of all the benchmarks. MPEG consists of one large basic block.
- JPEG - JPEG is the largest benchmark in the suite in terms of area and power consumption. JPEG's source contains no branching and thus is comprised of a single large basic block.

7.2 DELAY ELEMENT RESULTS

To examine the power reduction resulting from delay element insertion several benchmarks were synthesized as combinational hardware blocks and compared with the same structure with delay elements inserted. In accordance with the delay element designs, the target technology is the 0.13 μm IBM cell-based ASIC process with the updated cell library containing the delay element cells. All hardware designs are VHDL designs automatically generated from C descriptions. The delay elements were inserted automatically using the tool flow previously described.

Results are shown for both the greedy and brute force methods. The greedy method was run with a limit of two delay elements and $\sqrt{N}$ nodes, where N is the number of operators in the hardware design. Using a constant parameter is not an efficient method to automating the process for circuits of varying sizes. $\sqrt{N}$ was chosen as it provided a good trade-off between including enough delay elements and not coming close to an element for each node. The number of nodes for each benchmark is shown in Table 5. The node count includes each node in the benchmark's SDFG which has a static timing delay associated with it. This excludes inputs, constants, output, constant shifters, etc.

Table 5: Number of Nodes Per Benchmark

Benchmark	**Node Count**
Sobel	24
ADPCM Decoder	26
ADPCM Encoder	33
IDCT Row	42
IDCT Column	45
Laplace	29
GSM	26
CRC32	24
MPEG	653
JPEG	672

All the versions of the circuit performed the same amount of processing over the same period of time, thus it makes sense to compare power directly as opposed to energy that introduces a temporal characteristic. The only execution time overhead in the cases with delay elements came from the propagation delay of the latches. This delay was in the picosecond range and had a negligible impact to the simulation times. The delay elements also needed to be reset with each new set of inputs. This was not a problem because the falling edge could be reset nearly instantaneously as shown in Figure 15.

Figure 27 illustrates the actual power savings seen via the use of delay elements. The power scale is logarithmic. All of the benchmarks show power savings, with the exception of ADPCM Decoder. For the smaller benchmarks, the $\sqrt{N}$ greedy heuristic performs very well, certainly beating the constant two limit. The brute force method was less predictable doing well in some cases rather than others.

Scalability is one of the more important characteristics of any hardware solution. In Super-CISC functions, the number of functional units is fairly representative of the size of the resulting hardware block implementation. Of the smaller benchmarks, IDCT Column is the largest with

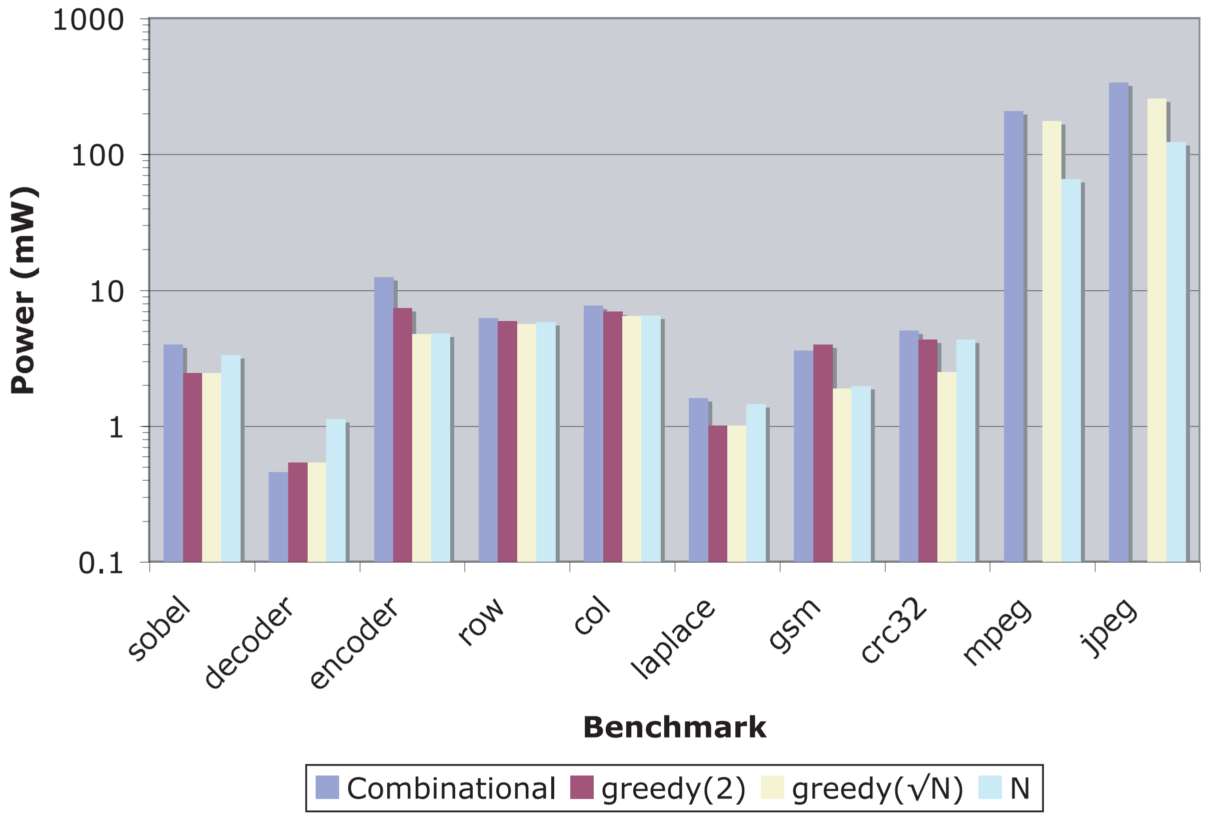


Figure 27: Benchmark Power Consumption With and Without Delay Elements

45 nodes, while ADPCM Encoder consumes the most power at 12.6 mW. However, for the larger benchmarks, MPEG consists of 653 nodes consuming 209 mW and JPEG, the largest benchmark, consists of 672 nodes, consuming 341 mW (27.1x more power than ADPCM Encoder). For these benchmarks, using two delay elements was not tested. For these cases, the brute force technique beat the greedy approach with $\sqrt{N}$ delay elements. The smaller benchmarks save an average of 8% using the brute force method (29% without Decoder). On the other hand, MPEG and JPEG both save over 50%. This is due to their relatively large size. They have much longer critical paths, which results in much more switching activity. Their large size and power consumption also allows them to support a greater number of delay elements and latches relative to their smaller counterparts.

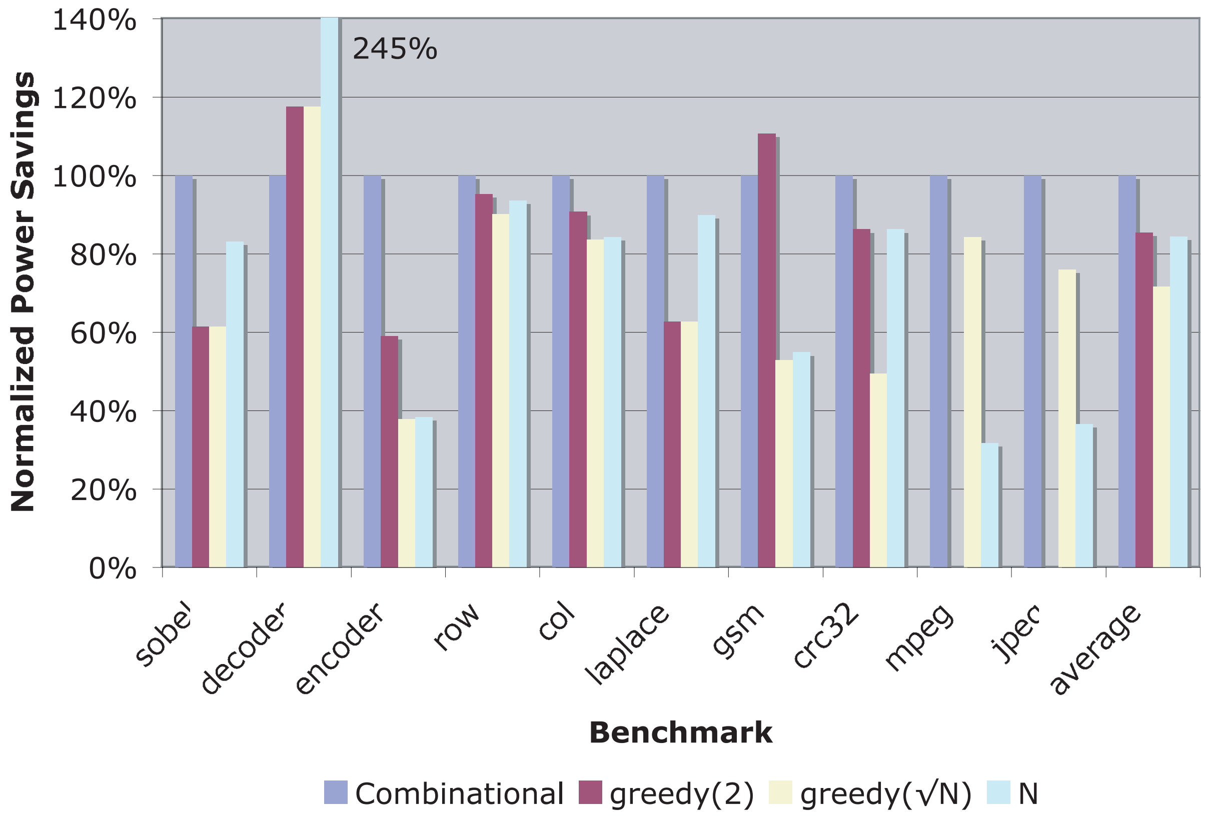


Figure 28: Normalized Power Consumption Comparison

Since the total power varies so highly, we examined a normalized power comparison for all of the benchmarks, shown in Figure 28, to better see the trends. The combinational implementation is used as the baseline case. For the brute force algorithm, an average savings of 19% is seen. The square root greedy algorithm yields a total savings of 28%. The outlier, ADPCM Decoder, particularly skewed the results in the case of the brute force algorithm.

To gain further insight into the effects of delay elements on particular nodes in the system, a visualization tool was developed show the impact of different delay element configurations on the power consumption of each node in the SDFG based on the power reports. The visualization tool is a Perl script which parses the PrimePower report files to extract the power consumption for each node. The power consumption is then normalized and annotated to the corresponding node in the

DOT file as a hexadecimal color value. The script can also be used to normalize power numbers across multiple designs. This is particularly useful because it allows multiple designs to be compared fairly by using the same power scale.

Figure 29 illustrates the power consumption in the Sobel benchmark prior to delay element insertion. The more red in color a node is, the more power it consumes. Similarly, blue nodes consume less power. In this example, the two red adders as well as the multiplexers and the purple comparator consume the most power.

In Figure 30, a delay element has been inserted into the graph. Both designs are normalized to the same power scale. In this example, the two latches that are inserted actually bisect the graph. It can be seen that the previously red and purple nodes in the graph have become much closer to blue in color due to a reduction in switching power. The nodes above the delay element remain unaffected. The delay element and latches are also very blue in color, indicating their low power overhead. For this particular example, total power was reduced by 38%.

7.3 DELAY ELEMENTS VS. PIPELINING

In [15] energy and execution time trade-offs were studied when pipelining SuperCISC hardware functions. Pipelining allows additional levels of parallelism to be achieved by overlapping the processing of multiple sets of data. The studies found that pipelining could be used to achieve a 3.3x reduction in execution time compared to a strictly combinational design. The drawback to pipelining is that the introduction of a clock signal, registers, and faster switching in each pipeline state can cause an increase in energy consumption. Interestingly, the pipeline states prevents long combinational switching times, which often serves the same purpose as the delay element based approach. Unfortunately, this reduction in switching activity was not enough to overcome the additional power overhead due to the pipelining hardware.

Our experiments have shown that the energy consumption of the pipelined designs increases by 70% on average over the combinational approach. Figure 31 shows energy usage comparisons between delay element implementations and their pipelined equivalents. The clock frequency used in the pipelined designs is 200 MHz. Clearly, the choice to implement either pipelining or delay elements depends on the design requirements. Designs that require higher performance may benefit from pipelining, while delay elements would be a better fit for low power designs.

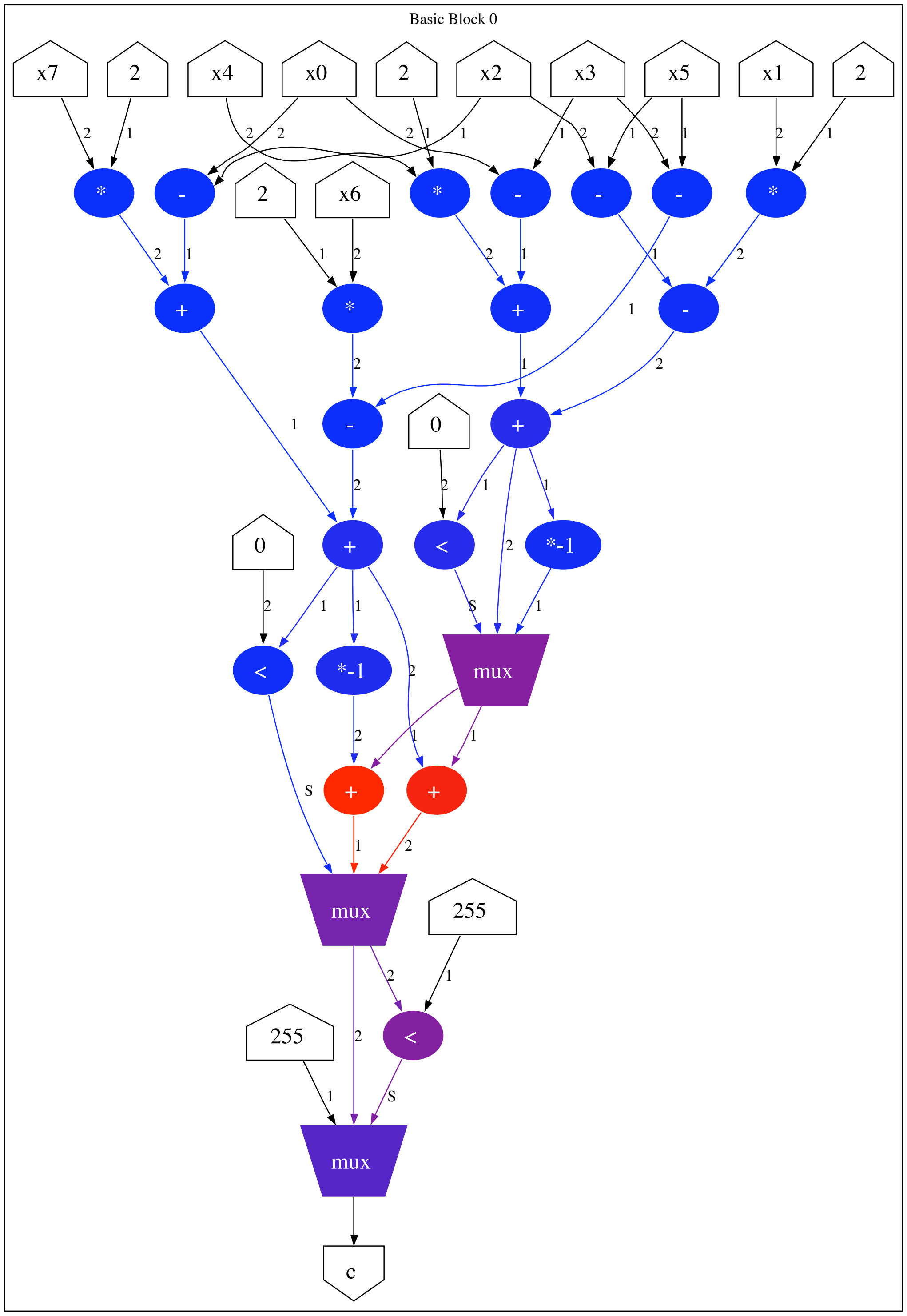


Figure 29: Power Consumption Per Node Prior to Delay Element Insertion

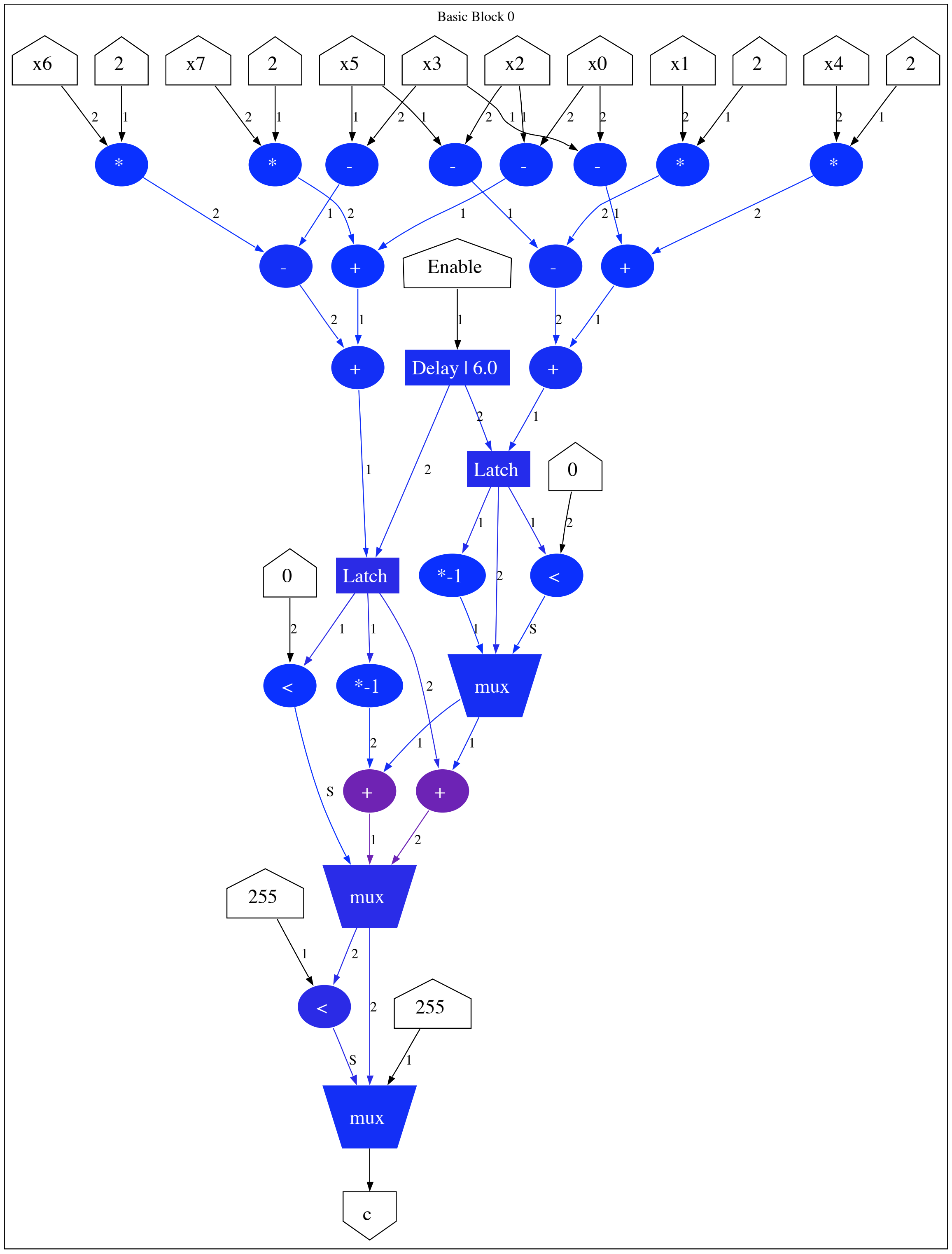


Figure 30: Power Consumption Per Node After Delay Element Insertion

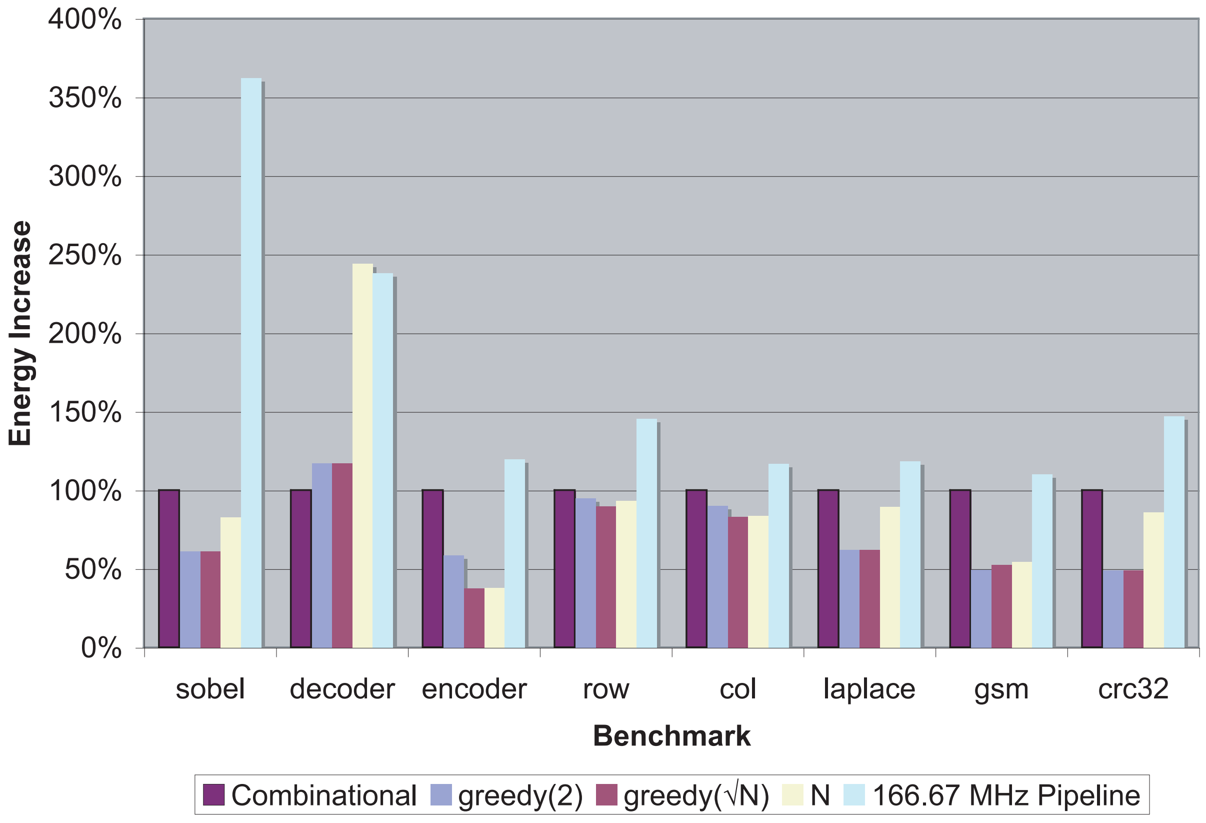


Figure 31: Energy Comparison Between Delay Elements and Pipelined Designs

8.0 FUTURE DIRECTIONS

Future directions of this work involve the incorporation of delay elements into a reconfigurable fabric device. In [30] the authors introduce a coarse-grained, stripe based reconfigurable fabric. The fabric is a structured device consisting of rows of ALUs separated by rows of multiplexer based interconnects used for routing. Figure 32 illustrates the high level conceptual model of the fabric. In [29] several interconnect structures were tested in order to try to reduce energy consumption. After varying the interconnect, the ALUs were examined. In [28] homogeneous ALUs were replaced with heterogeneous ALUs. It was found that not every ALU in the fabric needed to support all of the operations in order to achieve the necessary functionality. This modification proved to be an efficient method of reducing energy in the fabric. In the future, delay elements will be used in the fabric in the hopes of seeing additional energy savings. This type of design lends itself towards the use of delay elements for several reasons. Firstly, in such a structured device, the ALU, interconnect, and wire delays are all well known. This allows for more accurate static timing and delay element insertion. Secondly, the multiple 32-bit functional units in each ALU consume considerable more power than the dedicated ASIC functional units. Thirdly, unlike the full custom ASICs described in this paper, the fabric uses uniform 32-bit data paths. Knowing this information a priori allows for the creation of a delay element more finely tuned to a fixed drive strength .

To be useful in the fabric, the delay elements will also need to be reconfigurable. Each delay element itself is not reconfigurable. However, multiple delay elements can be combined in a single reconfigurable delay module. A control signal in the fabric can be used to select one of the delay values, or no delay at all. The output from each ALU will be latched, with the delay modules controlling these latches. The delay modules will be the focus of the future work. The delay modules will clearly not be homogeneous across the rows of the fabric. Other questions include how many delay elements to include in each module, and what granularity to use between the

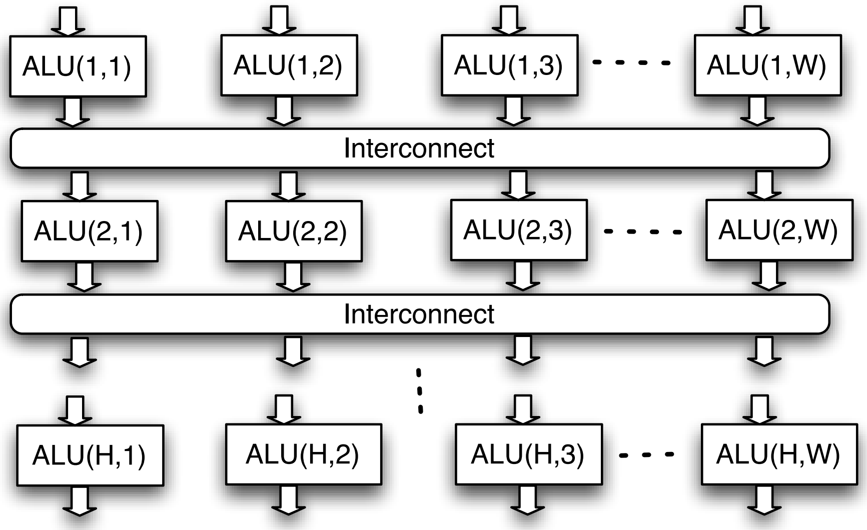


Figure 32: Reconfigurable Fabric Conceptual View

delay elements in each module. Figure 33 shows an example of this reconfigurable delay module. In this example, *DelaySelect* is used to choose between a four, six, or eight nanosecond delay. The *Constant Enable* value is also provided if no delay is desired.

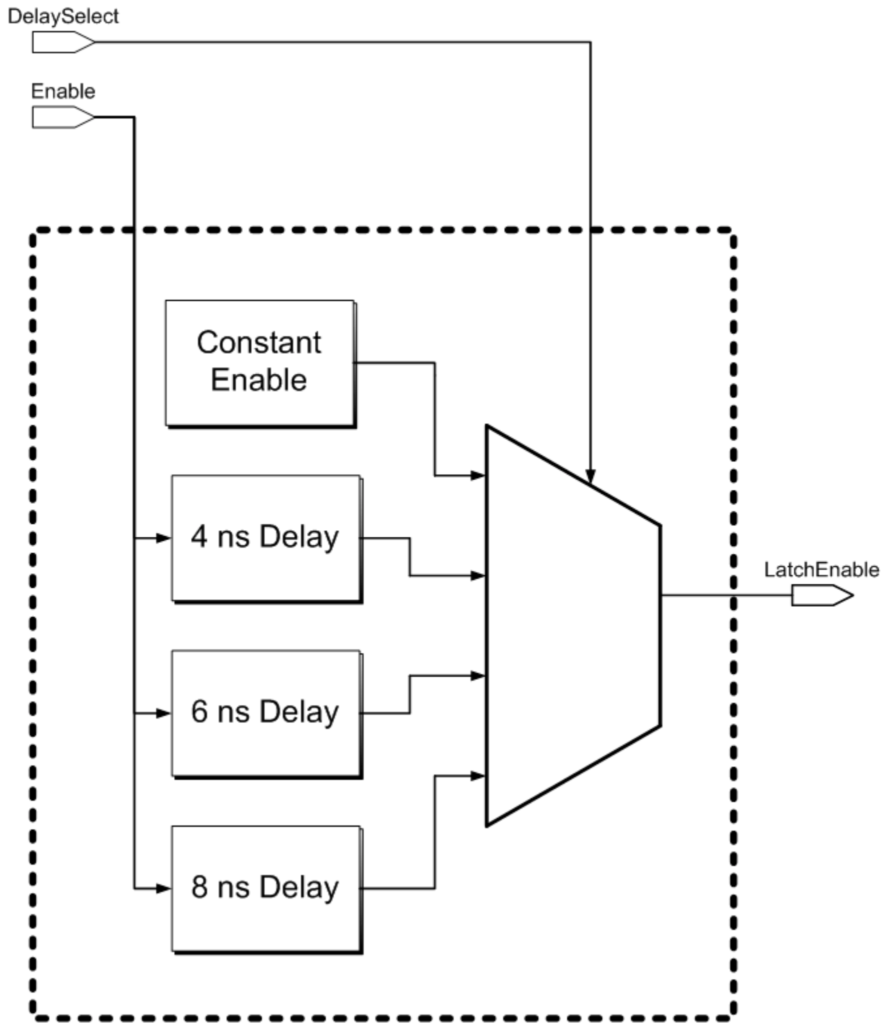


Figure 33: Reconfigurable Delay Module

9.0 CONCLUSIONS

This thesis describes a static timing analyzer as well as two design automation tools which utilize the analyzer. All of these tools were developed within the SUIF research compiler infrastructure, and tested using current industrial strength EDA tools. The particular contributions made by this thesis are as follows:

1. A static timing analyzer for the SuperCISC compiler. The annotations created by the static timing analyzer allow for the design and use of timing based tools such as the ones described in this thesis.
2. An automated pipelining tool. The pipelining tool allows for the rapid development of pipelined circuits, requiring no design effort on the part of the developer. Additionally, pipelines of different frequencies can be tested via a user defined input parameter.
3. A greedy heuristic algorithm for delay element insertion. The greedy algorithm attempts to judiciously insert delay elements into a SuperCISC SDFG. A brute force delay element insertion algorithm was also designed. Additionally, a tool for manual placement of delay elements was developed for testing purposes.
4. A complete post-SuperCISC tool flow for delay element based designs. The tool chain performs synthesis, simulation, and power estimations on the delay element circuits. This contribution consists of a Design Compiler technology library of delay elements, synthesis timing constraints, and a delay element VITAL simulation library.

The static timing analyzer described in this thesis provides a means to create additional timing based optimizations and transformations of SuperCISC hardware functions. The static timing analyzer is based on the depth first search, a well known graph traversal algorithm. Based on this static timing analysis, two tools were developed.

The first tool described in this thesis attempts to increase throughput using synchronous pipelining. Pipelining, a computer architecture technique found in nearly all modern processors, increases performance by increasing the throughput metric of a circuit. Previously published results have shown that performing hardware acceleration can result in performance increases of over 10x compared to equivalent software implementations. By pipelining these hardware blocks, an average additional performance gain of 3.3x was seen due to an increase in throughput. This results in an increase of roughly 33x over software only approaches. Increasing parallelism even more is possible by replicating combinational logic blocks. Replication comes at a steep price in terms of area and power consumption. Pipelining can be used to achieve the same benefits in terms of parallelism and reduced execution time, while area and energy savings increase with the size of the hardware that would otherwise be replicated. The largest observed hardware functions saved over 6x the area and over 10x the energy by pipelining instead of replicating.

This thesis has also demonstrates a technique for reducing combinational switching power in SuperCISC hardware functions. This technique, which utilizes delay elements, allows for a decrease in power consumption while only having a negligible effect on performance. The primary source of power consumption in SuperCISC hardware functions comes from combinational switching. By introducing delay elements into the circuits, the same performance can be attained while consuming roughly two thirds to three quarters of the original power. The use of delay elements has also been shown to be scalable for large circuits capable of supporting the additional hardware's overhead.

BIBLIOGRAPHY

[1] L. Benini, G. DeMicheli, A. Macii, E. Macii, M. Poncino, and R. Scarsi, "Glitch power minimization by selective gate freezing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2000.

[2] T. Bridges, S. W. Kitchel, and R. M. Wehrmeister, "A cpu utilization limit for massively parallel mimd computers," in *Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[3] Z. Chen and K. Roy, "A power macromodeling technique based on power sensitivity," in *DAC '98: Proc. of the 35th annual conf. on Design automation*. ACM Press, 1998, pp. 678–683.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. Cambridge, MA, USA: MIT Press, 2001.

[5] I. S. Department, "Standard vital asic modeling specification." [Online]. Available: citeseer.ist.psu.edu/department95standard.html

[6] A. Devgan and C. Kashyap, "Block-based static timing analysis with uncertainty," in *ICCAD '03: Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 607.

[7] C. Ebeling, D. C. Cronquist, and P. Franklin, "Rapid - reconfigurable pipelined datapath," in *in the 6th Int. Workshop on Field-Programmable Logic and Applications*, 1996.

[8] R. Georing, "Synopsys launches power tool," *EE Times*, 2000.

[9] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. Taylor, "Piperench: A reconfigurable architecture and compiler," *in IEEE Computer*, vol. 33, no. 4, April 2000.

[10] S. Gupta and F. N. Najm, "Power macromodeling for high level power estimation," in *DAC '97: Proc. of the 34th annual conf. on Design automation*. ACM Press, 1997, pp. 365–370.

[11] S. Gupta, R. Gupta, N. Dutt, and A. Nicolau, *SPARK: : A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Kluwer Academic Publishers, 2004.

[12] M. Hashimoto, H. Onodera, and K. Tamaru, "A power optimization method considering glitch reduction by gate sizing," in *ISLPED-98: ACM/IEEE Int. Symposium on Low Power Electronics and Design*, 1998.

[13] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach, Third Edition*. Morgan Kaufmann, 2002.

[14] R. Hoare, A. K. Jones, D. Kusic, J. Fazekas, J. Foster, S. Tung, and M. McCloud, "Rapid VLIW processor customization for signal processing applications using combinational hardware functions," *EURASIP Journal on Applied Signal Processing*, vol. 2006, pp. Article ID 46 472, 23 pages, 2006.

[15] C. J. Ihrig, J. Stander, and A. K. Jones, "Pipelining tradeoffs of massively parallel supercisc hardware functions," in *IPDPS/APDCM Workshop*, 2007.

[16] P. M. Institute, *A Guide To The Project Management Body Of Knowledge, 3rd ed.* Project Management Institute, 2003.

[17] A. K. Jones, D. Bagchi, S. Pal, P. Banerjee, and A. Choudhary, *Pact HDL: Compiler Targeting ASIC's and FPGA's with Power and Performance Optimizations*, R. Graybill and R. Melhem, Eds. Boston, MA: Kluwer Academic Publishers, 2002.

[18] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, and J. Foster, "An fpga-based vliw processor with custom hardware execution," *ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2005.

[19] A. K. Jones, R. Hoare, D. Kusic, J. Fazekas, G. Mehta, and J. Foster, "A vliw processor with hardware functions: Increasing performance while reducing power," *IEEE Transactions on Circuits and Systems II*, 2006.

[20] A. K. Jones, R. Hoare, D. Kusic, G. Mehta, J. Fazekas, and J. Foster, "Reducing power while increasing performance with supercisc," *ACM Transactions on Embedded Computing Systems (TECS)*, 2006.

[21] G. Kim, M.-K. Kim, B.-S. Chang, and W. Kim, "A low-voltage, low-power cmos delay element," *IEEE Journal of Solid State Circuits*, 1996.

[22] D. Kusic, R. Hoare, A. K. Jones, J. Fazekas, and J. Foster, "Extracting speedup from c-code with poor instruction-level parallelism," in *IPDPS Workshop on Massively Parallel Processing (WMPP)*, 2005.

[23] L. Lavagno, G. Martin, and L. Scheffer, *Electronic Design Automation for Integrated Circuits Handbook*. CRC, 2006.

[24] B. A. Levine and H. Schmit, "Efficient application representation for haste: Hybrid architectures with a single, transformable executable," in *IEEE Symposium on FPGAs for Custom Computing Machines(FCCM)*, 2003.

[25] X. Liu and M. C. Papaefthymiou, "A markov chain sequence generator for power macromodeling," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, July 2004.

[26] S. Malik, M. Martonosi, and Y.-T. S. Li, "Static timing analysis of embedded software," in *DAC '97: Proceedings of the 34th annual conference on Design automation*. New York, NY, USA: ACM, 1997, pp. 147–152.

[27] S. McCloud, "Catapult c synthesis-based design flow: Speeding implementation and increasing flexibility," Mentor Graphics, Tech. Rep., 2004.

[28] G. Mehta, C. J. Ihrig, and A. K. Jones, "Reducing energy by exploring heterogeneity in a coarse-grain fabric," in *IPDPS Reconfigurable Architecture Workshop (RAW)*, 2008.

[29] G. Mehta, J. Stander, and A. K. Jones, "Interconnect customization for a coarse-grained reconfigurable fabric," in *IPDPS Reconfigurable Architecture Workshop (RAW)*, 2007.

[30] G. Mehta, J. Stander, J. Lucas, R. R. Hoare, B. Hunsaker, and A. K. Jones, "A low-energy reconfigurable fabric for the supercisc architecture," *Journal of Low Power Electronics*, vol. 2, no. 2, pp. 148–164, August 2006.

[31] S. Muchnick, *Advanced Compiler Design and Implemenatation*. Morgan Kaufmann, 1997.

[32] R. Pyreddy and G. Tyson, "Evaluating design tradeoffs in dual speed pipelines," in *Workshop on Complexity-Effective Design*, Goteborg, Sweden, June 2001.

[33] A. Raghunathan and N. K. Jha, "Behavioral synthesis for low power," in *Proc. of ICCD*, October 1994, pp. 318–322.

[34] Synopsys Inc., "Design compiler and primepower manual," `www.synopsys.com`.

[35] G. Tellez, A. Farrahi, and M. Sarrafzadeh, "Activity-driven clock design for low power circuits," in *International Conference on Computer Aided Design*, San Jose, CA, United States, 1995.

[36] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," in *IEEE Design and Test of Computers*, September 1993.

[37] V. Tiwari, S. Malik, and P. Ashar, "Guarded evaluation: Pushing power management to logic synthesis/design," *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 1051–1060, November 1998.

[38] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez, "Reducing power in high-performance microprocessors," in *Design Automation Conference - DAC*, 1998.

[39] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarsinghe, J. M. Anderson, S. W. K. Tjiang, S. W. Liao, C. W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, "Suif: An infrastructure

for research on parallelizing and optimizing compilers," *ACM SIGPLAN Notices*, vol. 29, no. 12, pp. 31–37, December 1994.

[40] M. B. Woolf, *Faster Construction Projects with CPM Scheduling*. McGraw Hill, 2007.

[41] J. Zhang, S. R. Cooper, A. R. LaPietra, M. W. Mattern, R. M. Guidash, and E. G. Friedman, "A low power thyristor-based cmos programmable delay element," *ISCAS*, 2004.