Stacked FSMD:
A New Microarchitecture Model for High-Level Synthesis

by

Khushwinder Singh Jasrotia

A thesis submitted in conformity with the requirements
for the Degree of Master of Applied Science in the
Graduate Department of Electrical and Computer Engineering,
University of Toronto

Stacked FSMD:

A New Microarchitecture Model

for High-Level Synthesis

Khushwinder Singh Jasrotia

Master of Applied Science, 2003

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

High-Level synthesis is a process that automates the transformation of an algorithmic description of a digital design into its physical implementation. With digital systems' ever increasing complexity in terms of transistor count and clock speed, it becomes necessary for a high-level synthesis tool to work at higher levels of abstraction in order to effectively cope with the design.

Traditional high-level synthesis tools are unable to efficiently synthesize designs described in high-level abstract languages such as *C* or *Java*. This is because the Finite State Machine with Datapath model (FSMD), the underlying microarchitecture into which the synthesis tool transforms the design, is too simplistic. FSMD is unable to effectively capture high-level constructs such as procedure abstraction and memory allocation.

This thesis makes two primary contributions: First, it proposes an extension of the FSMD model into a Stacked FSMD (SFSMD) model that supports procedure abstraction, and includes support for dynamic memory allocation. Secondly, it describes a behavioral level partitioning technique which leverages the SFSMD model to reduce power consumption.

Acknowledgements

First, I would like to thank my supervisor Professor Jianwen Zhu for his advice, guidance, and support. He was always willing to set aside any amount of time to clarify issues relating to my research. His continual patience and encouragement were invaluable for the successful completion of this thesis.

To my family, thank you for your love, support, patience, and in believing in my ability to go the distance.

To my friends from lab EA306: Zhong, Fang, Rami, Linda, and Dennis, thanks for all your help and support. Val, thanks for providing all those movie passes and for all the interesting discussions we've had. A special thanks to Lesley for taking the time to read my drafts and for being a great friend.

Finally, I would also like to thank Y-Explorations Inc. for donating the *eXcite* synthesis tool which proved invaluable for generating the experimental data used in this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

With transistor densities of over one hundred million gates and clock frequencies in the gigahertz range, digital systems today are truly complex. Systems of such complexity are very difficult to design by hand-crafting each transistor or by defining each signal in terms of logic gates [10]. To cope with this ever increasing complexity, automation tools will need to work at higher levels of abstraction [10] [18].

Working at higher levels of abstraction provides several advantages: firstly, working at lower levels makes a problem humanly intractable and time consuming. This is because at low levels there are too many components and interactions for the human mind to comprehend. At higher levels of abstraction, the number of objects to consider is reduced by orders of magnitude, which allows the designer to design and explore more complicated designs in less time. Secondly, the designer's way of thinking is analogous to higher-level design methodologies. While designing a multi-million gate chip, it is hard to imagine a designer thinking in terms of register-transfer level (RTL) components such as ALUs, multiplexors, registers, memories, etc. Higher algorithmic and process level descriptions are required to comprehend and specify the design. High-level synthesis facilitates the implementation of digital designs at this higher level of abstraction.

High-level synthesis is an automated refining process, from an abstract description of

a digital design to a detailed one [18]. One of the first steps in the creation of a digital system is the modeling of the intended design. Today, most modeling is done using *Hardware Description Languages* or HDLs. The HDL is used to capture the intended functionality of the circuit and it provides the abstract basis on which the high-level synthesis tool operates. Other modeling tools exist such as graphical models, which support flow diagrams, schematic entry and geometric layout. However, the very-large scale nature of the problem forces the modeling to be one that supports hierarchy and high-level of abstraction, and consequently HDL modeling is used for most large-scale designs.

Conceiving an HDL model has similarities to writing a software program. The conciseness of HDL models has made them preferable to the corresponding graphical models [18]. Today, VHDL and Verilog are the most commonly used HDLs. Both languages offer similar functionalities: both HDLs support both behavioral and structural views of a design, i.e, a circuit can be described in terms of its functionality, or can be described as a an interconnection of sub-components. Also, both languages specify semantics that allows circuits to operate in parallel. Having evolved from programming languages [31], both languages support features such as data abstraction (data-types and variables), behavioral operators for transforming data, assignment operators to change values of operators, and control and execution ordering constructs in order to specify flow of control [10].

The next logical evolution for synthesis tools is the support for standard high-level languages (HLLs) such as *C*,*C++*, or *Java*. HLLs have been used for functional validation such as functional modeling of processors, but only recently have tools been available that allows the generation of hardware directly from a high-level language such as *C* [13]. HDLs offer constructs for the control of execution ordering that are lacking in HLLs, however, enhancements to the languages can be made to support these features. Examples are new HDL languages based on *C* such as ESIM [9], HardwareC [19], SpecC

[11], and SystemC [33].

The primary advantage of supporting synthesis from an HLL is the higher-level of abstraction offered over HDLs. HLLs offer constructs for data-encapsulation (structs and classes in *C++*) and method/procedure calls that are more flexible and powerful than corresponding implementations in VHDL or Verilog. They also support constructs that are non-existent in the HDL domain, such as dynamic memory allocation. Additionally, HLLs have the benefit of being familiar to more designers than standard HDLs. Compared to *C*, less are familiar with VHDL or Verilog. The ability to design hardware directly in *C* provides a definite advantage to a designer, who otherwise would need to familiarize himself with the semantics/syntax of VHDL.

Typically, a design is coded in an HLL in order to verify functionality. The code is then given to a hardware designer who is responsible for understanding the code and translating it into an HDL description suitable for synthesis. Being able to synthesize hardware directly from the HLL eliminates the later step and consequently shortens the design cycle, a critical factor in today's competitive electronics industry.

Despite the intensive research efforts invested in the last decade, the notion of high-level synthesis from HLLs unfortunately remains in the hands of academia and a few EDA companies, rather than the design community. The reason for such a reluctance is mainly because the size or complexity of the application that current high-level synthesis tools can accept is too small.

Two main areas need to be examined in order to improve support for synthesis from HLLs : 1) The effective handling of the increased complexity introduced by these languages and, 2) the power-efficiency of the synthesized designs. The study of these two aspects form the basis of this thesis and are described in the next section.

1.1 Motivation

The motivation for this study is to explore the key issues involved in the support of high-level synthesis from HLLs. These issues are described below.

1.1.1 Support for Higher Levels of Abstraction

Supporting synthesis for HLLs requires the ability to efficiently cope with additional abstraction layers. We identify two main areas of abstraction which are not handled effectively by current high-level synthesis tools. These are 1) *procedure abstraction* and 2) *memory abstraction*.

All HLLs and to some level, HDLs, support the concept of procedure and function calls. These constructs allow for program modularity and aid in design reuse. Most current high-level synthesis tools handle procedures by *inlining* them. By inlining, all calls to a procedure are replaced by the body of the procedure. In some cases this can improve the performance of the final design because the synthesis tool can optimize the procedure with the rest of the code. However, for multiple procedure calls, inlining can result in the undesirable effect of increased code size. Furthermore, indiscriminate inlining can place prohibitive demands on memory and runtime.

The inlining of procedures creates a monolithic finite state machine with datapath (FSMD) at the microarchitectural level. As mentioned earlier, high-level synthesis is a refinement process, and the microarchitecture level (also known as the Register Transfer Level) is a "view" of the design at a lower abstraction level. The FSMD consists of two parts, the *controller* and the *datapath*. The datapath circuit is used to store and manipulate data and to transfer data from one part of the system to another [3]. The datapath circuit is comprised of components such as registers, latches, multiplexors, counters, decoders, adders, and so on. The controller circuit is a finite state machine that controls the operation of the datapath circuit.

We extend the notion of this traditional simplistic FSMD model of RTL hardware into a stacked FSMD model (SFSMD) that supports procedure abstraction. In this view, each procedure is implemented as a separate controller with a common datapath shared amongst all the controllers. A hardware stack mechanism is used to control the flow of procedure calls and to allow the datapath to be shared. This approach has several advantages: First, by handling each procedure separately, the divide-and-conquer approach is utilized which reduces the memory and run-time requirements on the synthesis tools. Furthermore, implementing each procedure as a separate controller can simplify the control logic [26], possibly resulting in a smaller and/or faster circuit. Also, significant power savings can be introduced by observing that while one controller is running, the other controllers can be shutdown by stopping their clocks. Lastly, sharing a common datapath introduces opportunities for global optimizations and helps improve circuit performance.

The other abstraction layer studied is memory abstraction, specifically dynamic memory allocation. Support for dynamic memory allocation by current high-level synthesis tools is virtually non existent. Current tools only deal with simple variables and arrays that are statically mapped to registers files and memories in the RTL domain. Memory allocation is traditionally carried out in software, however, the ability to synthesize a dynamic memory allocator can have several benefits: First, a hardware allocator can run much faster than software, and can free up CPU cycles for a software application [14] - in certain memory-intensive garbage collection based programs, memory management can take up to one-third of program time [35]. Second, the abundance of silicon area available in current VLSI technology makes the option of implementing a memory allocator in hardware very attractive. Third, support for dynamic memory allocation would be indispensable for hardware applications that require dynamic memory management such as a TCP protocol engine described in *C*. Finally, dynamic memory management is an integral part of HLLs - this makes the case for hardware allocation all the more

compelling since useful constructs, such as pointers, could be supported.

In this thesis, we study and describe the implementation of a hardware memory allocator. The allocator operates in conjunction with memory devices at the microarchitecture level. The memory allocator is described as a soft IP core. Its quality, performance, and scalability is studied under modern process technologies.

1.1.2 Generation of Power Efficient Designs

Power efficient circuits are an important goal for high-level synthesis tools and much work has been done in this area, as surveyed in [7]. It is well known that by partitioning a large circuit in an intelligent way, power consumption can be reduced. However, such partitioning is traditionally performed at the logic level, or structural RTL level, where information of the application is somewhat lost. Recently, focus has switched to power reduction at the higher levels where large power savings are possible merely by cutting down on wasted switching activities [30] [17] [2]. This can be accomplished by shutting down unnecessary portions of circuits, and the new SFSMD model that has been introduced is very suitable for this type of optimization.

As mentioned earlier, the SFSMD model uses procedures as controller boundaries. It consists of multiple controllers interacting with each other via a stack-controller with a shared datapath. The ability to shutdown inactive controllers is inherent in the SFSMD model, since as each procedure executes, only the controller associated with that procedure is active. However, designers use procedures to enhance readability and maintainability, and the resulting SFSMD circuit from such procedures may not result in the best synthesized design for power. Power consumption can be reduced by intelligently redefining the procedure boundaries of the original specification.

The observation that 10% of a program's instructions account for 90% of its execution time has been used in the context of high-performance processor and compiler design [12] [16]. These observations are due to the prevalence of loops in programs. We introduce

a high-level partitioning scheme which aggressively transforms the original design by discovering *regions*, which can be considered as frequently executed loop kernels. Each region is *exlined* into separate procedures in order to redefine the procedure boundaries of the original specification. By implementing each procedure as separate controllers in the SFSMD model, power consumption can be effectively reduced. This is because the controller for each loop is smaller than the single controller implementing the entire system, and since only one controller is running at any given time, the remaining ones can be deactivated [30].

1.2 Objectives

The goal of this research is to improve synthesis from HLLs by studying and modifying the underlying microarchitecture. The main objectives of this thesis are listed below:

1. To improve support for procedure abstraction in high-level synthesis. This is performed by studying the microarchitecture model and extending it to support procedures by introducing the concept of the stacked FSMD (SFSMD).

2. To add support for dynamic memory allocation in high-level synthesis. This is accomplished by designing and implementing a memory allocator IP core.

3. To investigate partitioning techniques at the behavioral level for the reduction of power. Frequently executed regions of code, such as loops, are extracted to be implemented as separate controllers in the SFSMD model. By implementing the loops as separate controllers, power can be reduced since inactive controllers can be shut-down.

1.3 Contributions

The contributions of this thesis are summarized below:

- The concept of the SFSMD is introduced for the support of procedure abstraction. The SFSMD model is described in detail along with an example. The example is used to highlight the features and functionality of the SFSMD model.

- A memory allocator IP core is described. The quality of the core is evaluated by studying how well it scales with modern process technologies. This is achieved by examining the effects of varying the heap size of the memory controller and employing various compile strategies to synthesize it. The following parameters of the synthesized deign are measured 1) the area, 2) the speed, 3) and the synthesis time.

- A behavioral partitioning technique aimed at reducing power consumption is introduced. A detailed example is used to demonstrate the partitioning methodology along with the resulting power saving figures. The partitioning scheme is also tested on loop-intensive *C* benchmark kernels to demonstrate its effectiveness for designs described in HLLs. Finally, a power-index is developed that can be used to estimate and compare the power of different partitioning styles prior to synthesis.

1.4 Thesis Organization

This thesis is divided into six chapters. Chapter 2 provides background information and reviews related work. Chapter 3 describes the SFSMD microarchitecture and shows how it can be used to support procedure and memory abstraction. Chapter 4 provides a description of the region based partitioning technique. Chapter 5 applies this partitioning technique to various benchmark programs and reviews the experimental results. Also, synthesis results of the memory allocator are presented. Finally, chapter 6 concludes with suggestions for future work.

Chapter 2

Background and Related Work

2.1 Introduction

This section provides background information for the ideas presented in this thesis. It also summarizes related work that has been performed and compares it to the work done in this study. First, an overview of high-level synthesis is presented to familiarize the reader with the synthesis process, issues, and terminology. Then, related work performed in the areas of procedure and memory abstraction is presented and compared. Finally, previous research performed on partitioning techniques for the reduction of power consumption is summarized.

2.2 Overview of High-Level Synthesis

Synthesis can be described as a translation process from a behavioral description to a structural one [10]. The representation of a circuit, or its modeling, is simply an abstraction that shows relevant features without associated details. The task of high level synthesis is to refine a model of higher abstraction into a lower one.

2.2.1 Abstraction Levels

The circuit models can be categorized into different abstraction levels and views. Four main levels of abstraction are traditionally defined, namely: *system level, microarchitectural level, logic level,* and *circuit level*. The levels can be visualized as follows: At *system level*, the circuit can be represented by a set of processes defined typically by an HDL. The processes communicate with each other via shared variables or message passing. At the *microarchitecture level* circuits perform a set of operations, such as computation and transfers at the register level. The *logic level* defines a circuit in terms of boolean equations and logic functions, and the *circuit level* is composed of transistors and other geometric entities [10].

Each abstraction level can be seen under different views as listed in table 2.1 [10]. The views are classified as: *behavioral*, *structural*, and *physical*. The *behavioral view* describes only the function of the intended design without regard as to how the function is implemented. The *structural view* defines the design as an interconnection of components, and the *physical view* defines the design in terms of physical entities (eg. chips, transistors, etc) [18].

Level Name	**Behavioral Representation**	**Structural Representation**	**Physical Representation**
System level	Algorithms Processes Flowcharts	Processors Controllers Memories Buses	PC Boards Chips
Microarchitecture level	Register transfer	Registers ALUs Memories MUXs	Chips Floorplans
Logic level	Boolean functions	Logic Gates Flip-flops	Cells Modules
Circuit level	Transfer functions Equations	Transistors Schematic	Transitory Layouts Traces Contacts

Table 2.1: Abstraction Levels in High-Level Synthesis

Transfer functions, equations, and timing diagrams are used to describe behavior at the circuit level. The behavioral view at the logic level consists of boolean equations and state diagrams. At the microarchitectural level, behavior is defined in terms of register transfers. Execution is divided up into discrete intervals called control states or steps. The register-transfer description is used to specify for each control state 1) what conditions are to be tested 2) which register-transfers are to be executed 3) and the next control state to be entered [10]. The system level behavior is typically defined using HDLs that use algorithms, language operators, processes, and variables to specify functionality. At this level, variables have not been assigned to registers or memories, and operations have not been bound to functional units or control steps. Furthermore, the concept of time is further abstracted to the order in which variable assignments are executed [10].

The structural representation bridges the behavioral and physical representation [10] - it is a mapping of the behavioral view into structural level components. At the circuit level, it consists of circuit level components such as transistors, resistors, capacitors. Logic level components consist of elements such as logic gates, registers, and latches. Memories, multiplexors, ALUs, and registers are used to represent microarchitectural level components, while controllers, memories, processors, and busses are used on the system level.

The physical representation maps the design to space or silicon. Examples are polygons, cells, floorplans, chips, and PC boards.

2.2.2 Definition of High-Level Synthesis

Based on the abstraction levels, synthesis can be defined for each level of abstraction, for example, system level synthesis converts system level behavioral specification to a structural one, while microarchitectural synthesis converts an RTL description into a series of interconnected registers, ALUs, and multiplexors. At each level, the corresponding

synthesis tool adds a level of detail and information that can be used by the next lower level synthesis tools.

High-level synthesis generally spans the first two levels of abstraction - it transforms a behavioral system-level description into a microarchitectural structural representation output. Lower level synthesis tools, such as a logic synthesis tools, further transform this output to make it suitable for circuit level synthesis, and so on, until the design is fully specified in silicon. This can be depicted graphically on what is often referred to as a Gajski and Kuhn's Y-Chart [6] in figure 2.1.

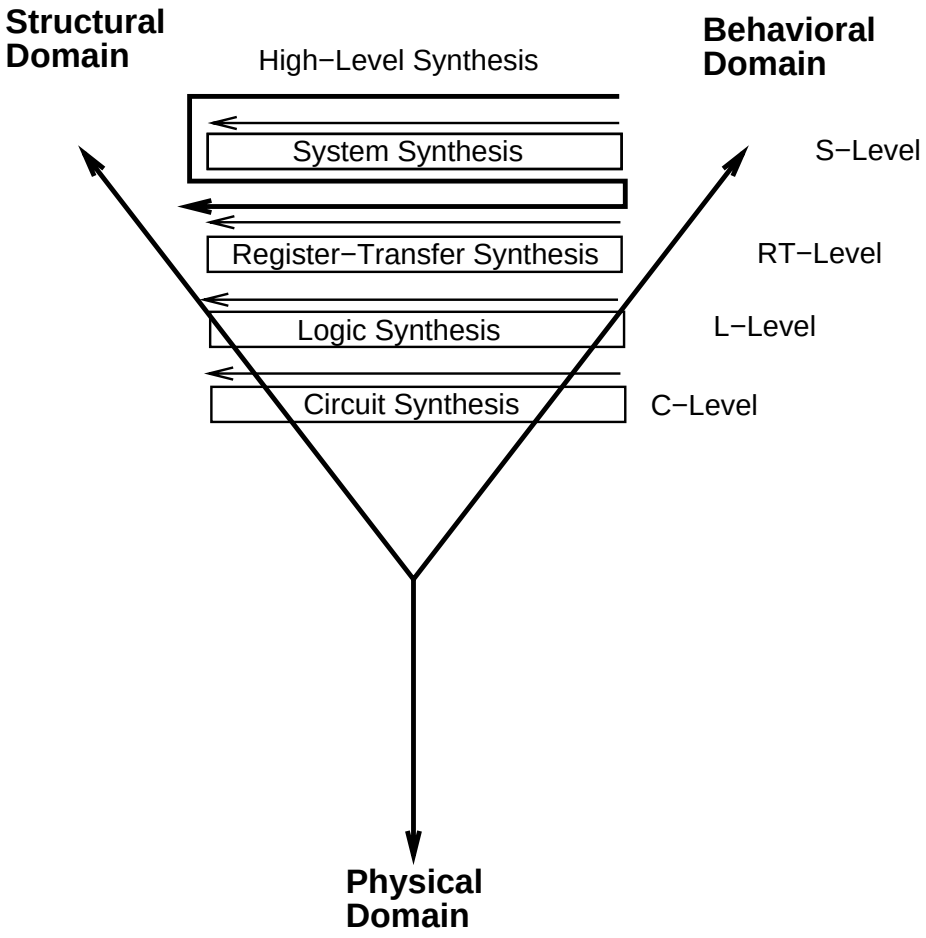


Figure 2.1: Y-Chart

High-level synthesis takes a description in terms of processes communicating via variables. From this, a series of interconnected memories, busses, processors, and controllers are generated, each of which can be described by a register-transfer description. From the register-transfer description, the following two structures are generated: 1) the *datapath*, which consists of storage and functional components used for the processing and transfer of data, and the 2) *controller*, the finite-state machine that controls the operation of the datapath. This structure is collectively known as the finite-state machine with datapath (FSMD).

The steps involved in transforming a high-level behavioral description into the FSMD model are listed below:

1. Compilation - Translation of original code into an intermediate format such as control flow graph (CFG). Control-flow and data-flow dependencies are explicitly obtained from the CFG.

2. Partitioning - The dividing of the design into sub-groups, each of which can be implemented using a FSMD model. Partitioning is performed to satisfy constraints such as minimizing chip size, power dissipation, speed, etc.

3. Scheduling - Specifies the assignment of variables and operations into discrete time-intervals. Control-flow and data-flow dependencies dictate the order of the assignments, amongst other external constraints such as speed and latency requirements.

4. Allocation - Assigns variables and operators to storage and functional units.

2.2.3 Definition of FSMD

This section defines the FSMD. The FSMD model is used to describe a digital design at the register-transfer level. Unlike a simple finite state machine, an FSMD is more powerful in that it may include variables with data types, as well as complex data operations in its actions. The FSMD can be formally defined by a 6-tuple as follows: [10] [30]

$$P =< S, I \cup STAT, O \cup A, f, h >$$

where:

- $S = \{s_0, \ldots, s_n\}$ is a set of states.
- $I = \{i_j\}$ is a set of primary inputs.
- $O = \{o_k\}$ is a set of primary outputs.
- VAR is a set of all storage variables.
- $EXP = \{f(x, y, z, \ldots) : x, y, z, \ldots \in VAR\}$ is a set of expressions.
- $STAT = \{Rel(a, b) : a, b \in EXP\}$ is a set of status signals expressed as a logical relation between two expressions from the set EXP.
- $A = \{x \Leftarrow e : x \in VAR, e \in EXP\}$ is a set of storage assignments.
- f is a *state transition* function that maps a cross product of S and $I \cup STAT$ into S.
- h is the *output function* that maps a cross product of S and $I \cup STAT$ into $O \cup A$ for Mealy models or S into $O \cup A$ for Moore models.

The general structure of the FSMD is depicted in figure 2.2 [10]. It consists of two main units, the datapath and the control. The control unit consists of three main blocks: the state registers to hold the state information, the next-state logic block to generate new state inputs for the state registers, and output logic block to generate the outputs. The control unit controls the datapath via datapath control signals and receives feedback signals from the status bits. Both control and datapath units receive external inputs and can generate external outputs. The datapath external inputs and external outputs tend to be words, while for the controller they are single bits.

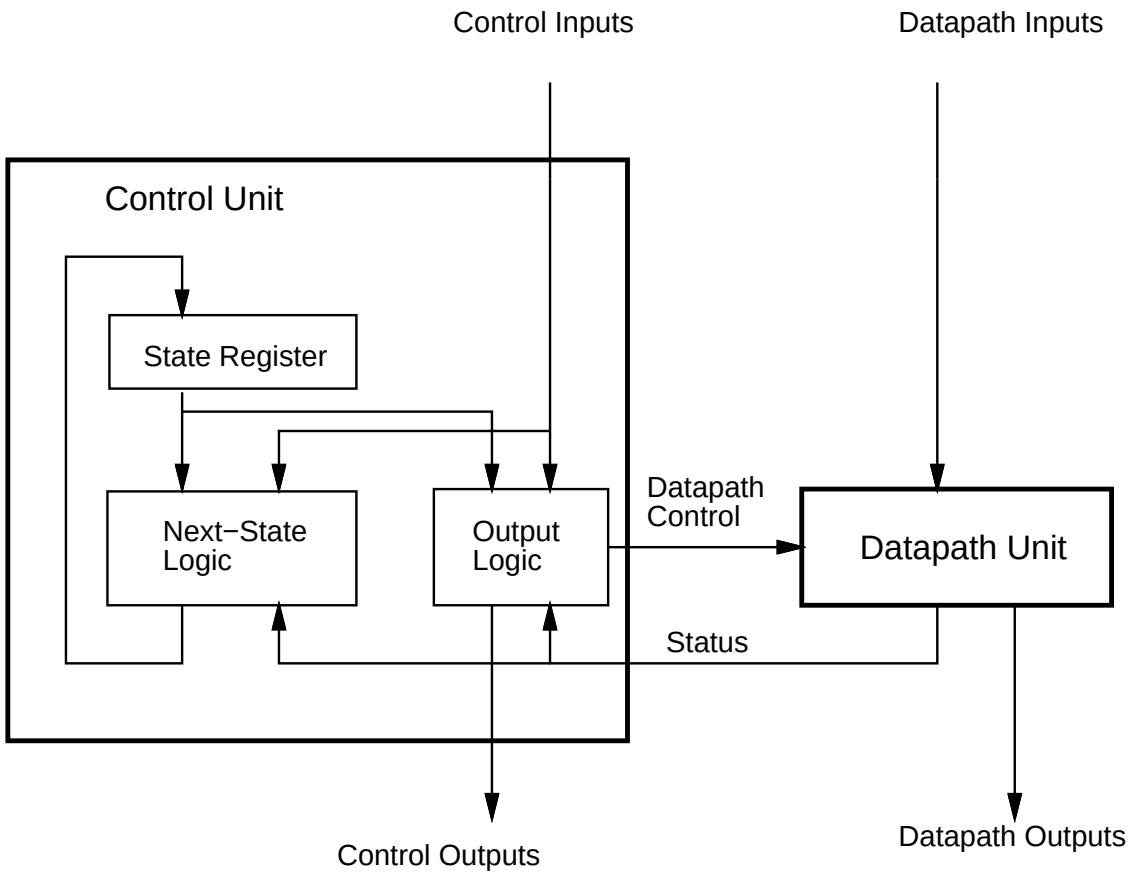


Figure 2.2: FSMD Block Diagram

2.3 Procedure Abstraction

This section reviews previously studied procedure abstraction techniques. A survey of previous techniques shows that procedures are handled in one of two ways: (1) each procedure is treated as a single instance, or (2) each procedure-call is expanded into the calling process [29] - this is also known as *inlining*. Both techniques have merits and demerits. A procedure treated as an instance can represent a basic indivisible computation, thus defining the granularity of functional partitioning [26]. Also, by processing procedures separately, run times and memory requirements can be decreased by an order of magnitude since synthesis tool heuristics are usually non-linear [28] [5]. An instance of a procedure can be used multiple times, hence aiding in design reuse and area reduction, and in some cases improving performance. Disadvantages are the possible overhead in the call and parameter passing mechanisms and the loss of optimizations over the boundaries [4].

Inlining has several advantages. Firstly, inlining maximizes the opportunities for allocation and scheduling tools to generate a concurrent design [26]. Secondly, performance may also be improved since operations in the main body and procedure can be overlapped. However, if a procedure is called numerous times, inlining can greatly increase the number of control steps in the controller thus increasing the controller area significantly. Also, inlining can create a large design for which scheduling and allocation must be performed, which may exceed the time or memory limitations of a high-level synthesis tool [21].

The procedure abstraction technique presented in this thesis is a compromise between the methods described above : procedures are implemented as separate controllers with a shared datapath. A stack mechanism is used to control the flow of procedure calls and to share access to the common datapath. The stack keeps track of the "address" of the calling and called modules. This stacked FSMD model provides a natural solution for the handling of nested and multiple procedure calls - a common occurrence in HLLs - and it

can also be extended to handle recursive calls. The model also produces power efficient designs due to its ability to shut-down controllers that are not in use. Furthermore, having a common datapath increases concurrency by allowing the sharing of resources, and can improve circuit performance.

Previous work has been done to examine the synthesis of descriptions containing procedure and function calls. In [4], procedures are synthesized as independent hardware modules, with the calling mechanism implemented by introducing a wait state in the control unit of the calling module. A similar technique is used in [5] - procedures are implemented as separate modules that share the same clock as the calling module. Recursion is not allowed. Dedicated ports generated for each procedure are used to pass parameters between the calling and called modules. Handshaking signals are used to perform procedure calls and returns. This scheme can have a negative impact on the final design due to high interconnection cost and increased controller complexity if many handshaking signaling schemes are required. Furthermore, since each procedure is implemented as an independent module, the scope for concurrent optimizations is reduced as only local optimizations are possible.

In [21], four different methods of procedure implementation are described and compared: (1) A *fixed-delay macro* method is described in which a procedure of fixed delay is synthesized as an independent macro module and directly instantiated as a datapath component of the calling module. (2) In the *variable-delay macro*, a procedure of variable delay is synthesized and instantiated in the datapath of the calling module. Due to the variable delay of the procedure, a handshake mechanism is required for the communication between the main controller and the variable delay macro. (3) *Inlining* is described in which each call to the procedure is replaced by the body of the procedure, and finally, (4) a *control subroutine* method is described where the controller of the procedure is incorporated with the main controller. Each procedure occupies a portion of the main controller state-table and each call to the procedure transfers control to this portion of

the state table. After the procedure finishes execution, control is transfered to the state following the procedure call. The return state is stored in a special variable, and interestingly enough, the paper mentions the possibility of implementing the variable as a stack for nested calls. This method shares similarities with the SFSMD model in that both use a stack to keep track of the return address and only one datapath unit is used. However, the SFSMD differs in that each procedure is implemented as a *separate* controller, with the ability to shutdown unused controllers for power savings. The control subroutine method, on the other hand, generates a monolithic controller which cannot be power efficient in this way.

In [27], procedures are implemented as separate modules and a common bus is used to transfer address and parameter information between them. A procedure call is initiated by transferring the address of the called procedure, possibly the address of the calling procedure, followed by any parameter data. Called modules are responsible for latching this data. A procedure return is implemented by transmitting the return address (that was latched earlier) and output parameters. Transmission of the calling address by the calling procedure, and return address by the called procedure adds overhead to the controllers compared to the SFSMD model. As will be shown in later chapters, in the SFSMD model, this information is inherent in the stack and is not required to be explicitly transfered by each controller. Furthermore, implementing each procedure separately results in loss of optimizations over the boundaries.

2.4 Memory Abstraction

Dynamic memory allocation is not a supported feature in high-level synthesis. One of the main reasons for this has been the lack of a hardware mechanism to perform dynamic memory allocation. Initial studies on hardware allocation were performed by Puttkamer [20] who used a shift-register based design to implement a buddy-system based allocation. That design was later modified by Chang and Gehringer [15] by using pure combinational logic for speed improvements. However, the design was never implemented or synthesized, so its performance under modern VLSI technology was never investigated.

The memory allocator implemented and studied in this thesis is based on the design proposed by Chang and Gehringer [15] and is described in the next chapter. The next section provides a brief overview of memory allocation techniques.

2.4.1 Memory Allocation Techniques

Memory allocators are used for general purpose *heap* (memory pool) storage, where a program can request a block of memory to store a program object, and free that block at any time [34]. Examples of software implementation of memory allocators are the *malloc* and *free* routines found in the standard C library.

The allocator must keep track of which blocks of memory are free and which are in use. Any blocks of memory that are returned must be made available for reuse. The allocator must balance the execution speed with the minimization of wasted memory. Due to the nature of programs, memory blocks can be freed in any order, thereby creating *holes* within the free memory. The accumulation of holes is known as *fragmentation*, and it can prevent the allocation of memory for larger blocks and is one of the major problems that allocators have to deal with [14].

Different memory allocation techniques have been developed to balance efficient memory usage with execution speed. These techniques are described below:

Sequential First Fit

This allocation technique belongs to a general class of allocation algorithms known as *sequential fits*. In this algorithm, free blocks are connected in a doubly linked list. During allocation, the free block is scanned and the first block that is sufficiently large is returned [8]. If the chosen block is larger than requested, it is split and the unused portion is added back to the free list. If a block that is freed is next to an already free block, the two blocks are coalesced into one big free block.

This technique generally exhibits good memory usage, however, suffers from increased search time due to fragmentation that occurs at the head of the list [34]. This is because search for free blocks must skip past those whose sizes are smaller than requested [14].

Segregated Free Lists

This technique uses an array of free lists, where each free lists holds free blocks of a particular size, usually based on powers of two. Memory requests are serviced by returning a block from a free list of suitable size, and memory is freed by returning the block to its appropriate list.

This is usually a very fast technique since relatively shorter lists have to be traversed compared with the sequential first fit mechanism. However, space is wasted since requests are rounded up to powers of two. This phenomenon is known as *internal fragmentation*. This technique also suffers from severe external fragmentation since blocks of a particular size cannot be used for another.

Buddy-System

This technique is a variation of the segregated free list. It supports a limited amount of splitting and coalescing. This scheme conceptually splits the entire heap area into two large areas, and those areas are further split up into two smaller areas, and so on. This hierarchical division constrains where memory objects are allocated [34]. Essentially,

the permitted sizes are powers of two, such that any block, except the smallest, can be divided into two smaller blocks of permitted sizes [14]. An example of the buddy-system is illustrated in figure 2.3. It shows the allocation of an 8K block of memory, followed by the allocation of a 10K block. Notice that after allocation of the 10K block, 6K of space is wasted due to internal fragmentation.

The performance of the buddy-system is a compromise between the earlier techniques. It is fast like the segregated free list technique, and it can also coalesce memory to save space like the sequential fit mechanism. However, the rounding leads to internal fragmentation.

The hierarchical sub-division of the memory heap into powers of two allows it to be naturally represented by a tree-structure. The tree-structure forms the basis of the hardware memory allocator described later in this thesis.

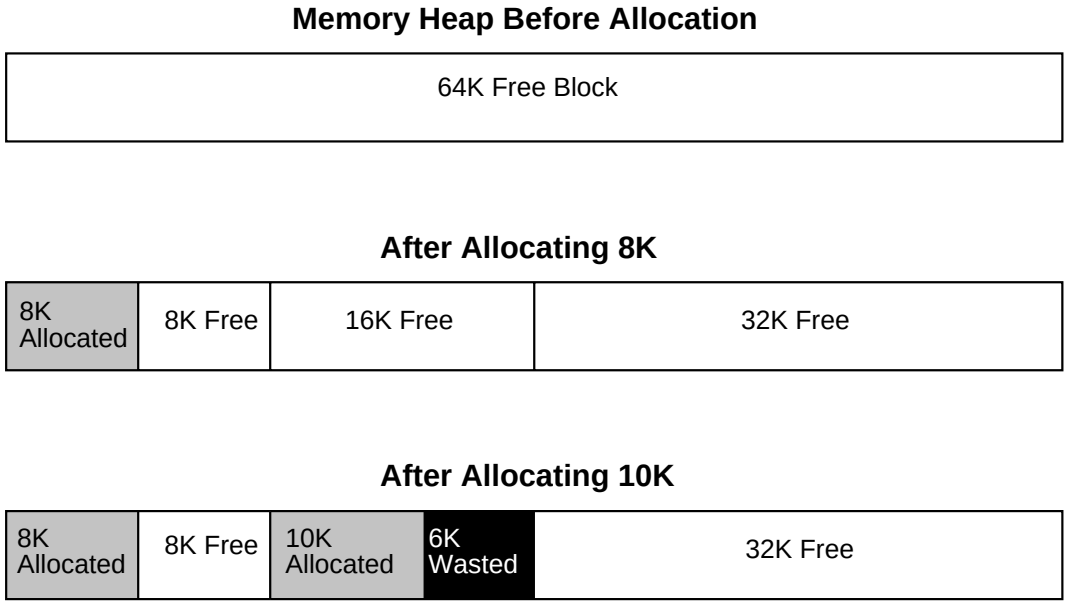


Figure 2.3: Example of a Buddy-System Memory Allocation Technique

2.5 Partitioning for Low Power

Recently, much work has been done in the reduction of power by focusing on higher levels of abstraction. This reduction has been possible by shutting down inactive portions of the circuits.

In [1] a method of *precomputing* is presented in which the output of a combinational logic block in the datapath is precomputed one clock cycle before the output is required. Power can be saved in the succeeding clock cycle by turning off the circuit since its value is already known. [24] presents a *guarded evaluation* technique which tries to determine, on a per clock cycle basis, which parts of a circuit are computing results that will be used, and which are not. The sections that are not needed are the shut off, thus saving power used in all the useless transitions in the part of the circuit.

In [2], a *controller* shutdown technique is presented. The control flow of the specification model is analyzed to detect mutually exclusive sections of the computation, and corresponding interacting FSMD are generated with selectively gated clocks. Only one of the interacting FSMDs is active at any given clock cycle, while all the others are idle and their clock is stopped. This work is extended in [30] by considering both the controller and datapath simultaneously, and shutting down both inactive pairs.

This thesis presents a new controller shutdown technique based on the SFSMD model. It operates at a higher abstraction level than the techniques surveyed earlier by directly targeting loops at the behavioral level. It is a well known fact that due to loops in programs, a small set of computations often account for most of the execution time. By extracting such loop *regions* and implementing them as separate controllers in the SFSMD model, power can be reduced significantly. This is because the controller of each loop kernel is much smaller than the original one implementing the entire process, and only one controller is operating at any give time, while the remaining controllers will be idle [30].

This region based partitioning has some possible advantages over the controller shut-down technique described in [2]. In [2], mutually exclusive controllers are extracted by (i) partitioning of states of the original behavioral description by the construction of mutual exclusiveness relation between basic blocks, and (ii) clustering the blocks of the partition to increase the granularity. Step (ii) is done because it is possible the first step may generate too many components resulting in high power overhead due to the interaction between components. By working at the loop level instead of the basic block level, the region based partitioning scheme automatically achieves a higher level of granularity since there are fewer loops in a program than the set of mutually exclusive basic blocks. Furthermore, loop identification and optimizations are standard tasks handled by a compiler and obtained for"free" by the synthesis tools - additional basic block level analysis performed in (i) is not required.

Chapter 3

The Stacked FSMD Microarchitecture

3.1 Introduction

This chapter describes the stacked FSMD microarchitecture. The first portion of the chapter gives a detailed description of the SFSMD architecture and shows how it can be applied for procedure abstraction. The second part of the chapter extends the SFSMD model to include support for memory abstraction by incorporating a dynamic memory allocation unit.

3.2 The SFSMD Model and Procedure Abstraction

The SFSMD model extends the classic FSMD microarchitecture model to include support for procedure abstraction. Specifically, this model supports sequential procedures (in the sense of a sequential programming languages like C). Sequential procedures have the characteristic that only one of them is active at any given time [5].

In the SFSMD model, procedures are implemented as separate controllers (FSMs) sharing a common datapath unit. The key feature is a special *stack controller* that

controls the interactions between the controllers and also allows the datapath unit to be shared. The structure of the SFSMD model is shown in figure 3.1. All components share the same clock. Note, the figure omits the external input and output signals of the controllers and datapaths for clarification purposes only.

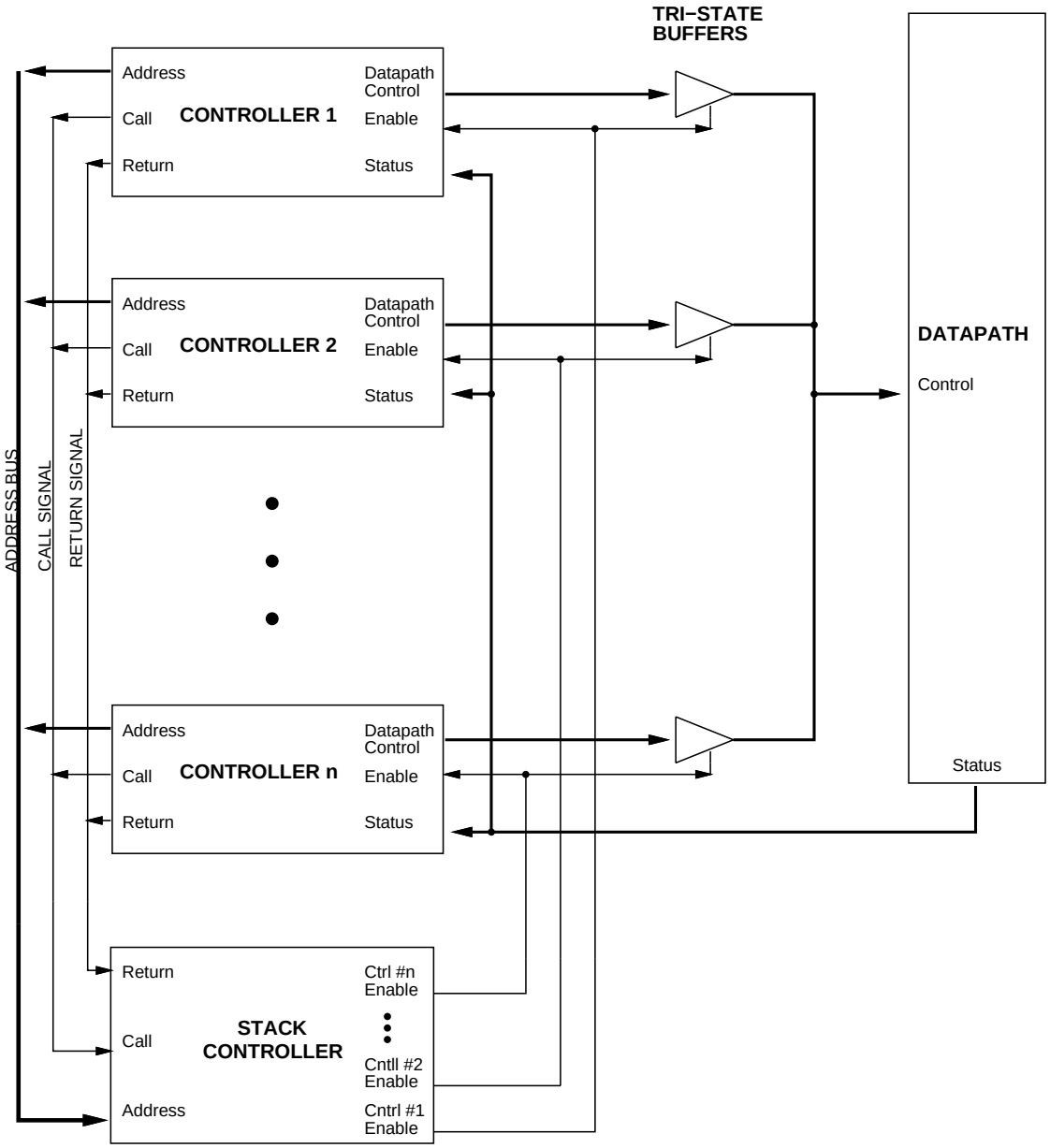


Figure 3.1: Stacked FSMD Model

The stack controller is used to handle procedure calls and returns in an analogous fashion to how stack mechanisms are used for subroutine linkage in microprocessors. The value stored on the top of the stack represents the address of the currently active FSM. This allows the stack controller to activate that particular FSM and halt the rest. Procedure calls are performed by pushing the address of the called FSM onto the stack, and returns are made by popping the stack so that control can be passed back to the caller FSM.

The stack controller controls the activation of the FSMs through the use of enable signals. It decodes the address value at the top of its stack to generate a dedicated enable signal for each FSM. Each enable signal is connected to the *enable* inputs of the state-registers of its corresponding FSM. When the enable signal is asserted, the FSM is able to operate normally, but when the signal is negated, the FSM is halted at the current state since its state-registers are unable to update. Only one enable signal is active at any time due to the sequential nature of procedures.

The enable signals are also used to control access to the datapath unit. Each FSM's datapath control signals are tri-state buffered to the inputs of the datapath unit. The enable signals are used to activate the tri-state buffer for the corresponding FSM so that the datapath components can be accessed. Again, due to the sequential nature of the procedures, only one set of datapath control signal is always active and driving the datapath components. The sharing of the datapath can also be implemented by a multiplexor, in which case encoded values of the enable signals are needed to drive the select inputs of the multiplexor. All controllers have access to the *status* signals of the datapath. Unshared datapath components can be directly controlled by their corresponding FSMs - these connections are not indicated in figure 3.1.

The FSMs transmit data to the Stack Controller via a shared unidirectional bus consisting of an *address bus*, a *call* line, and a *return* line. If the design consists of N FSMs, then *address bus* consists of $\lceil log_2 N \rceil$ lines, and is used to transfer the address

of the called FSM to the stack controller. *Call* is a single line used to indicate a valid address on the *address bus* for a procedure call. *Return* is a single line used to indicate a procedure return. For procedure returns, only the *return* signal is used, the address lines are not driven. Only one FSM controls the bus at a time, with the others providing high-impedance values. For the *call* and *return* signals, external pull-up or downs can be used to provide valid logic levels for the inputs of the stack controller when these signals are not being driven. The stack controller generates N dedicated *enable* signals, one for each FSM and its corresponding datapath control tri-state buffer.

In summary, the SFSMD model adds *address*, *call*, *return*, and *enable* signals to the controller of the FSMD model. The original *datapath control* and *status* signals serve the same purpose as originally described in Section 2.2.3. The *address* and *call* ports are required only on those FSMs that perform function calls, and the *return* port only on those that perform function returns.

Initially when FSM1 is operating, its address is at the top of the stack. When it needs to pass control to FSM2, it places the address of FSM2 on *address bus* and strobes the *call* signal indicating a procedure call. The stack controller pushes this address on to its stack causing FSM2 to activate and FSM1 to halt at its current state. After FSM2 completes its operations, it strobes the *return* signal to indicate a procedure return back to FSM1. This causes the stack controller pops its stack so that the address of FSM1 is back at the top of the stack. This activates FSM1 and it resumes execution. This mechanism can handle both multiple procedure calls and nested calls.

Support for recursion would require a modification: A calling FSM would need to additionally stack the identity of the control-step succeeding a procedure call and the the identity of the control-step in the called procedure. The stack-controller can be used to stack this information or a memory device can be instantiated in the datapath to implement the stack. Obviously, support for recursion makes the design more complicated, but it is nevertheless possible in the SFSMD model.

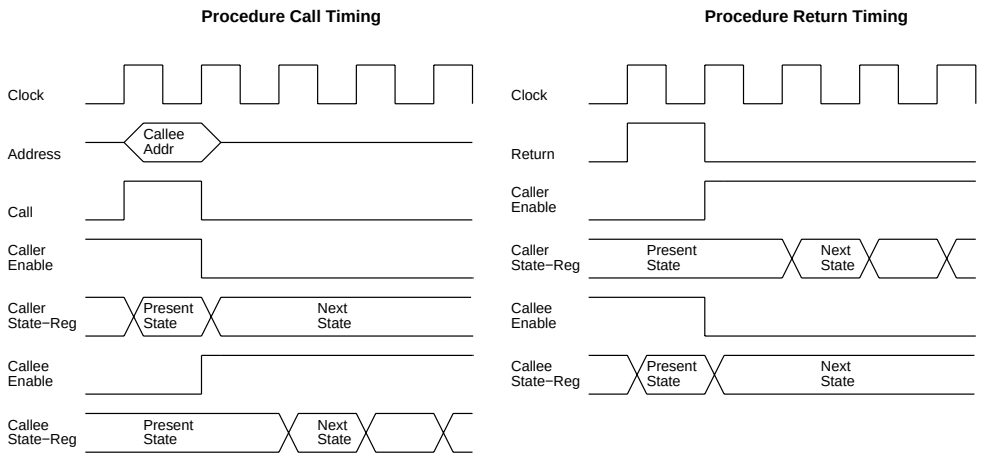


Figure 3.2: Timing Diagram for Procedure Call and Return operations

The timing of the signals involved in procedure calls and returns is indicated in figure 3.2. As indicated by the timing diagram, for procedure calls, the calling FSM halts at a state that immediately succeeds the state in which the *call* strobe was generated, and it resumes operation from that state when control is passed back. This implies that every function call must be followed by a *wait* state into which the FSM enters and idles. Extra states for driving the *address* lines and generating the *call* signal are not required. They can be generated by Moore-assignments in states that immediately precede the inserted *wait* state. This means that a calling procedure incurs the penalty of only one extra state due to the *wait* state.

Similarly, for procedure returns, a *wait* state needs to be inserted immediately after the state that generated the *return* signal. Since like procedures calls, an extra state is not required to drive the *return* signal, a procedure return operation requires only an extra state.

Since procedures may require the passing and returning of parameters, a set of input and output registers can be defined for each procedure. Prior to each procedure call, an additional control step may be required to copy the actual parameters to the input register, and another control step to receive any outputs stored by the procedure. In order to support recursion, the parameters will need to be stacked in memory.

3.2.1 SFSMD Design Example

An example is used to illustrate the SFSMD architecture as shown in figure 3.3. A simple procedural design written in VHDL describing *SumOfCubes* is used. This is similar to a design example used in [21]. The evaluation is performed by implementing *SumOfCubes* as an *inlined* procedure which uses the FSMD model, and comparing it to the SFSMD style in which it is described as a separate controller. Design tradeoffs between the two styles are presented and discussed.

```
entity EXAMPLE is
     port( IN1 : in std_logic_vector (3 downto 0) ;
           IN2 : in std_logic_vector (3 downto 0) ;
           OUT1 : out std_logic_vector (3 downto 0)
         );
end ;

architecture BEHAVIOR of EXAMPLE is
begin

      process
            procedure SumOfCubes(I,J: in std_logic_vector(3 downto 0);
                                 K: out std_logic_vector(3 downto 0)) is
                  variable II, JJ : std_logic_vector(3 downto 0) ;
            begin
                  II := I * I * I ;
                  JJ := J * J * J;
                  K := II + JJ ;
            end SumOfCubes ;

       variables A, B, C, D, E, F, G : std_logic_vector(3 downto 0) ;

      begin

            A := IN1 ; B := IN2 ;
            D := A + B ;
            SumOfCubes(A,B,C) ;
            E := C + 2 ;
            F := D + 2 ;
            SumOfCubes(F,E,G) ;
            OUT1 <= G + 5 ;

      end process ;
end BEHAVIOR ;
```

Figure 3.3: An Introductory Example

FSMD Implementation of *SumOfCubes*

When *inlining*, calls to a procedure are replaced by the body of procedure, resulting in a single FSMD structure. The synthesis results of the example after inlining both procedure calls is shown in figure 3.4. The *status* signals from the datapath unit are not required in this design and so are omitted. The advantages of inlining are apparent, as operations in the main body and the procedure are overlapped resulting in an 8 control step design.

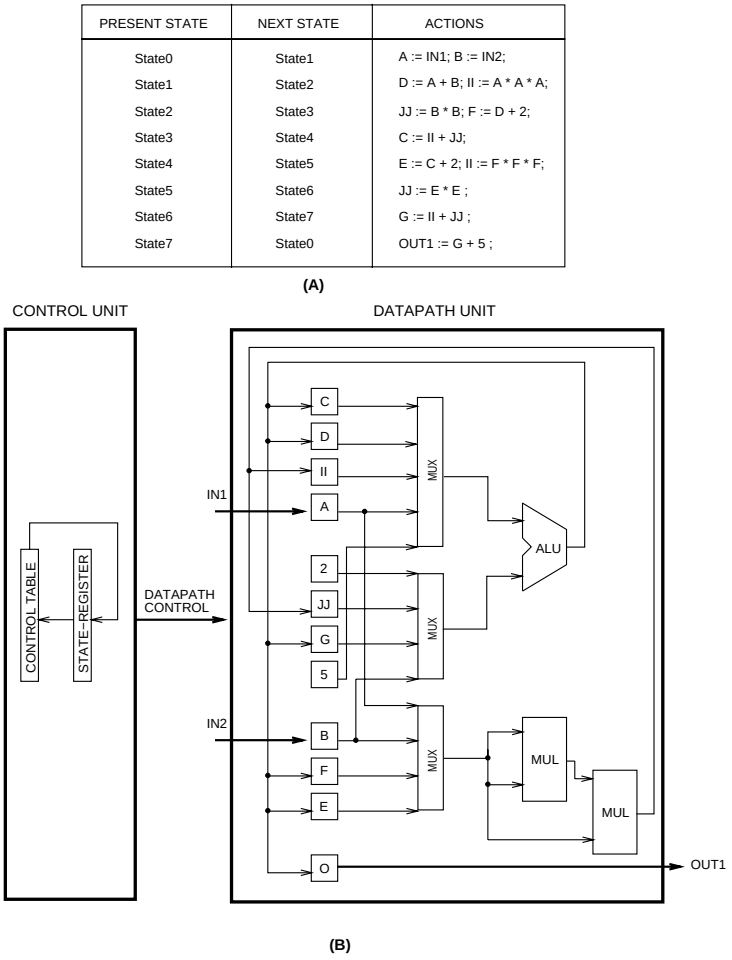


Figure 3.4: Procedure Inlining: (A) State table, (B) Design

We can now discuss the characteristics of the design in terms of resources and the

number of control steps.

Functional Units: Due to the sharing of resource between the main body and the procedure, at least one functional unit of each operation type that exists in either the main body or the procedure is required.

Number Of Registers: Since the procedure is inlined with the main body, in the worst case, the total number of registers required is equal to the number of variables in the procedure plus the number of variables in the main body. This can be reduced if the synthesis tool is able to perform optimizations.

Number Of Control Steps: The number of control steps for procedure inlining can be very high if none of the operations of the procedure can be overlapped with the main block. In this case, the total number of control steps will be the number of control steps in the main block, plus n times the number of control steps in the procedure, where n is the number of times the procedure is called. High values of n could result in a significantly complicated controller.

SFSMD Implementation of *SumOfCubes*

Figure 3.5 shows the implementation of *SumOfCubes* as a separate controller using the SFSMD model. This method can significantly reduce the size of the controller compared with inlining in cases where multiple procedures are called numerous times.

Procedure *SumOfCubes* is called from States 1 and 5 in the main controller FSM. These states are also used to store input parameters into registers P1 and P2. States 3 and 7 are used to read the output data from register R1. Implementation of the design requires a total of 17 clock cycles.

Functional Units: As in the inlining case, the main block and procedures share the functional units, requiring one functional unit for each operation type that exists in either the main body or the procedure.

Number of Registers: An additional register is required for each parameter that

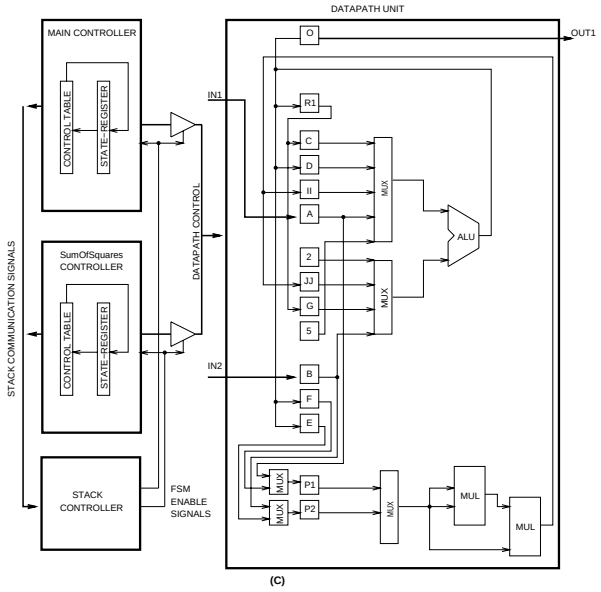
is passed to and from a procedure. Therefore, the total number of registers required is equal to the combined number of variables in the main block and the procedures, plus the number of parameters passed to and from each procedure.

PRESENT STATE	NEXT STATE	ACTIONS
State0	State1	A := IN1t; B := IN2 ;
State1	State2	D := A + B; P1 := A; P2 := B; Drive ADDRESS; Assert CALL;
State2	State3	Wait ;
State3	State4	C := R1 ; F := D + 2;
State4	State5	E := C + 2 ; P1 := F;
State5	State6	P2 := E ; Drive ADDRESS; Assert CALL;
State6	State7	Wait ;
State7	State8	G := R1 ;
State8	State0	OUT1 := G + 5 ;

(A)

PRESENT STATE	NEXT STATE	ACTIONS
State0	State1	II := P1 * P1 * P1;
State1	State2	JJ := P2 * P2 * P2;
State2	State3	R1 := II + JJ ; Assert RETURN;
State3	State0	Wait;

(B)



(C)

Figure 3.5: SFSMD Controller: (A) State table for Main Block (B) State table for Procedure (C) and Design

Control Steps: A procedure call incurs an overhead of two cycles - one for the *wait* state required after a procedure call, and one for the return. Additionally, parameter passing between the main body and the procedure, also requires extra cycles. This overhead, p, is dependent on the procedure call and is: 0 if **no** input or output parameters are used, 1 if **either** input or output parameters are used, of 2 if **both** input and output parameters are used. If there are n calls to a procedure then $n*(p+2)$ steps are required

in addition to the number of steps in the main block and procedures.

3.2.2 Stack Controller

The stack controller is used to handle the flow of procedure calls between the FSMs. A block diagram of the stack controller is shown in Figure 3.6. Control is passed to an FSM by pushing its address value onto the stack of the stack controller. This is accomplished by placing the address value on the address port of the stack controller and raising the *CALL* signal. This causes the address value to be pushed to the top of the stack and the corresponding FSM enable signal to be generated. The *RETURN* signal is used to pop the stack in order to return control back to the calling FSM. Both, pop and push operations take one clock cycle to complete.

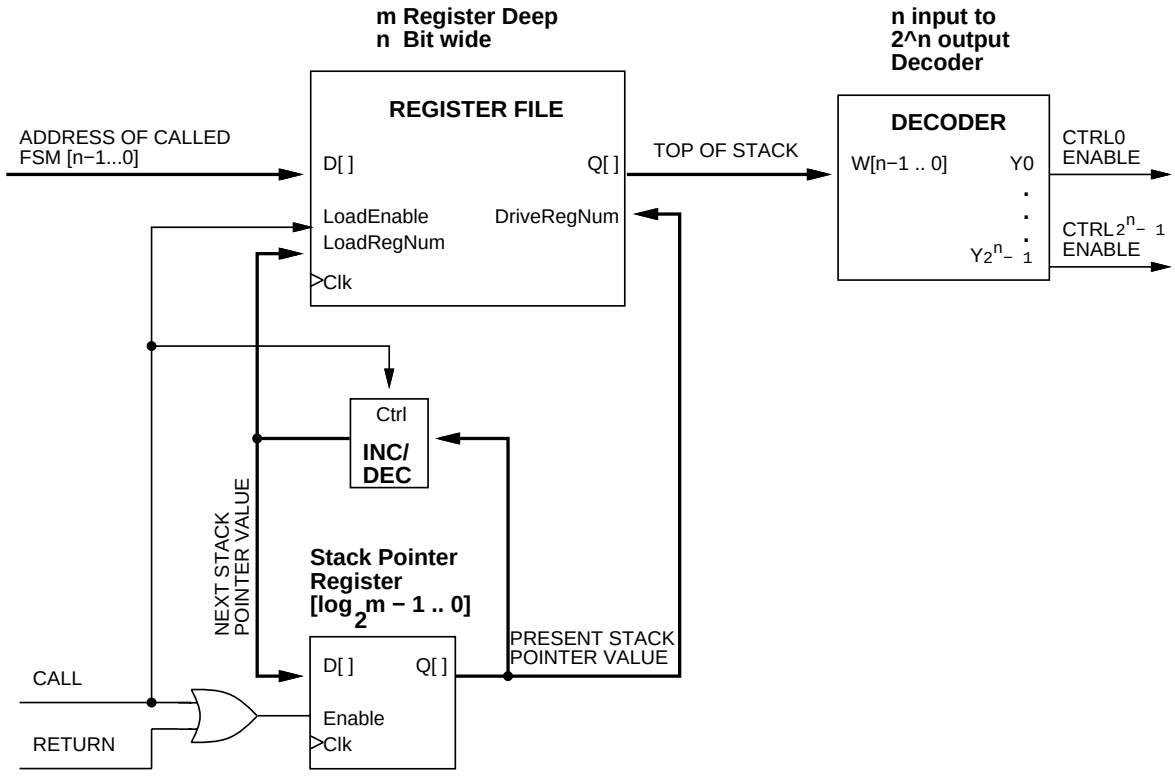


Figure 3.6: Stack Controller Block Diagram

The controller is implemented as a register-file based design where the number of registers in the file determines the size of the stack, and by extension, the maximum allowed depth for nested procedures. For non-recursive calls, the maximum depth of procedure calls can easily be determined during compile time and a stack controller of the appropriate size can be synthesized.

The stack controller consists of three major parts: (1) A *register-file*, that implements the stack, (2) a *stack-pointer register* that keeps track of the top of the stack, (3) and a *decoder*, that generates the enable signals for the FSMs.

The register file is implemented as an array of m registers, each n bits wide, where m corresponds to the maximum stack depth, and n is the number of bits required to encode the address of the FSMs. Data is written to a specific register by placing its address on the *LoadRegNum* input, and asserting the *LoadEnable* input. On the positive-edge of the clock, the data present on the *D[]* lines gets latched into the register. Data from a specific register is available on the *Q[]* lines by placing the address of that register on the *DriveRegNum* input.

The stack-pointer register and its associated components allow the register-file to be controlled as a stack. It holds the address of the register in the register-file that corresponds to the top of the stack. It drives the *DriveRegNum* input of the register-file so that the contents of the top of the stack are available at the output. The stack-pointer is able increment or decrement by utilizing a feedback mechanism that connects its output to its input via a controllable increment/decrement module - this feature is used to implement stack push and stack pop operations. Through an example we see how this is performed.

Assume initially the stack-pointer stores 0, which corresponds to the first register in the register-file. Also assume that first register is storing the number 9. The following sequence of operations are used to push the number 5 onto the stack: First, 5 is placed on the *D[]* inputs of the register file and the *CALL* signal is raised at the start of a rising

clock edge to indicate a push operation. The increment/decrement module increments the stack-pointer value and outputs a 1 which is fed to the *LoadRegNum* input of the register-file, and the *D[]* input of the stack-pointer. On the next rising edge of the clock, the register-file updates the contents of the second register with the value 5, and the stack-pointer updates to 1. The *Q[]* output of the register-file now outputs 5 to reflect the updated stack-pointer value.

For a pop operation, the *CALL* signal is negated and the *RETURN* signal is raised at the start of a rising clock edge. The increment/decrement module now decrements the current stack-pointer value and outputs 0. On the next rising clock-edge the stack-pointer register updates to this value. This causes the register-file to output the contents of the first register which is 9.

This example shows that the *CALL* and *RETURN* strobes must be active for only one clock cycle for correct operation. If these strobes are active for multiple clock-cycles, the stack-pointer will increment or decrement multiple times causing the wrong stack value to be output by the register file.

Lastly, an n input to n^2 output decoder unit is used to generate the enable signals for the FSMs. Only one output is asserted at a time, and each output corresponds to one valuation of the input. The input is connected to the output of the register-file which corresponds to the address of FSM at the top of the stack.

3.3 Dynamic Memory Allocation

A dynamic memory allocation unit can be incorporated into the SFSMD model. In fact, the allocator is general enough to be included in the FSMD architecture as well. During high-level synthesis, variable arrays with dynamic indexing result in memories [4]. At the RTL-level, this corresponds to memory devices instantiated as datapath components. An additional memory device can be dedicated for dynamic memory allocation - based on the size of this memory, an appropriate dynamic memory allocation unit can be synthesized to handle memory allocations for it.

The memory allocation unit can either be used as a datapath component, or be considered a separate microarchitectural component. In the SFSMD model, it would be shared amongst the various controllers in a way similar to how the datapath components are shared. A memory allocation operation would consist of transmitting the desired size of memory to the allocator, and strobing an *Allocate* signal. After a fixed number of cycles, the allocator would respond with a *Success* signal to indicate if the allocation attempt was successful or not, followed by the transmission of the starting address of the block for successful attempts. Memory free operations would involve transmitting the size and address of the memory block to be freed to the allocator, and asserting a *Free* strobe.

3.3.1 Description of Memory Allocator

This section provides a brief description of the dynamic memory allocation unit. It is a buddy-system based design similar to the one described by Chang and Gehringer [15].

The hardware allocator uses a bit-vector to represent memory. Each bit in the bit-vector represents the status of memory located at an address specified by the bit position of that bit. For example, in an eight-bit vector, bit 0 would represent memory at address 0, bit 1 to memory at address 1, and so on. A bit in the bit-vector can represent a memory

block of arbitrary granularity. For example, a bit can represent a byte of memory, or it can represent a word (2 bytes), or a long-word (4-bytes), etc.

A bit value of 0 indicates that it is free, and a bit set to 1 indicates that it is has been allocated. This is illustrated in figure 3.7. A hardware binary-tree is maintained that finds free blocks in the bit-vector using combinational logic.

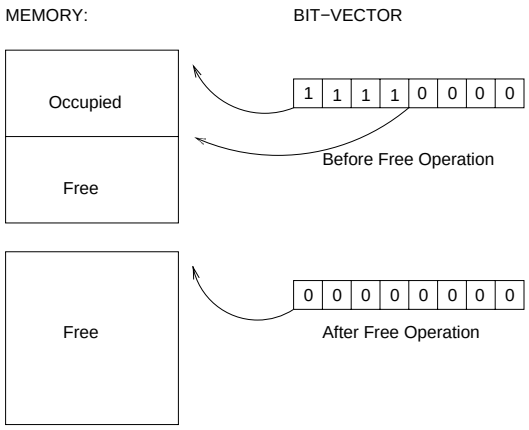


Figure 3.7: Bit-Vector Representation of Memory

The following hardware tasks need to be performed in order to manipulate the bit-vector:

1. Determine from bit-vector if there is a large enough space for allocation.
2. If so, find the starting address of that memory chunk.
3. Flip the corresponding bits in the bit-vector.

An *or*-gate tree is used to perform function (1), an *and*-gate tree used to perform function (2), and *bit-flipper* tree circuitry is used to perform function (3). The important aspect of this implementation is that the circuits are combinational, and therefore fast. Figure 3.8 provides a graphical overview of the memory allocator.

A memory of 2^N blocks requires a memory-allocator with trees that have $2^N - 1$ nodes. This indicates that the circuit size is directly proportional to the memory size.

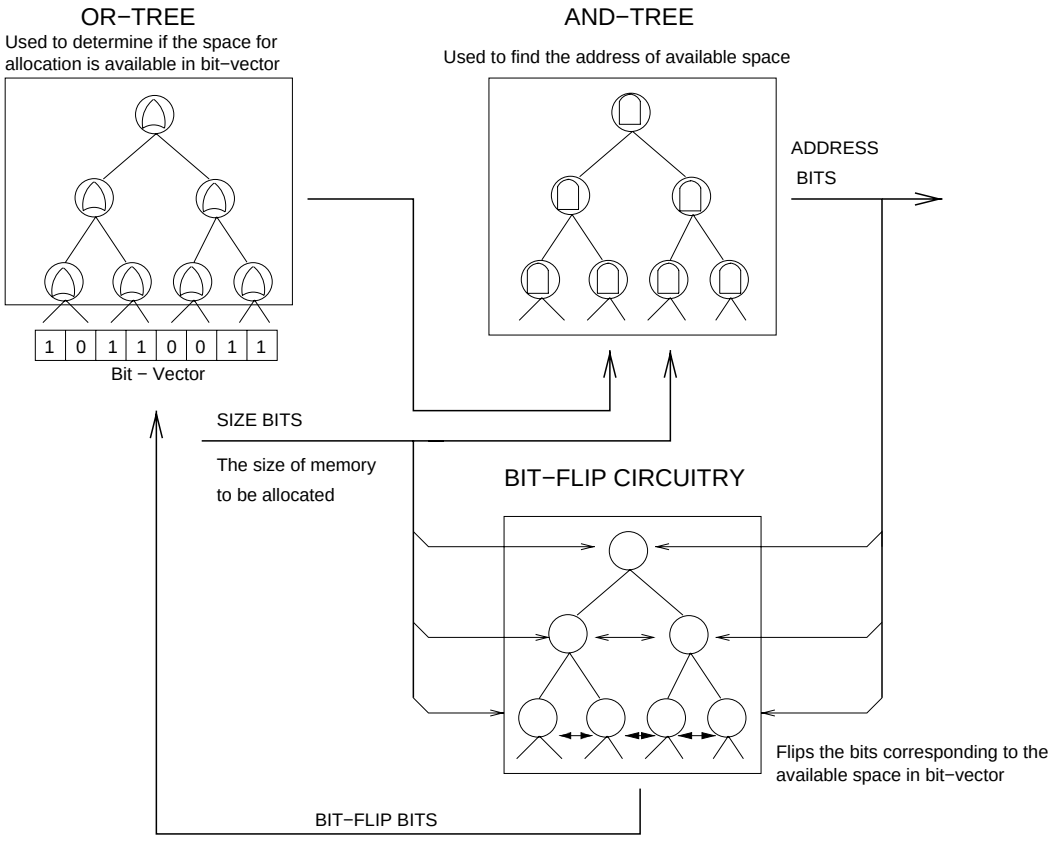


Figure 3.8: Memory Allocator Block Diagram

For allocations, the requested block size is presented on the *Size* bits of the allocator. The allocator returns the starting address of the free memory space that is available in the in the bit-vector. It also generates the bits that are used to mark the the corresponding bits in the bit-vector as occupied. Status signals are used to indicate to the user as to the successful completion of the operation.

Free operations are carried out by presenting the allocator with the size and address of the block of memory to be freed. The bit-flip circuitry generates the bits to mark the corresponding bits in the bit-vector as available.

OR tree

The OR-tree is used to determine if a free block is available in order to accommodate an allocation request. It consists of a complete binary tree "built on top" of the bit-vector [15], where the nodes of the tree consist of simple OR-gates. The leaves of the tree are the bits of the bit vector as indicated in figure 3.8. For a bit-vector with 2^n bits, a tree with $2^n - 1$ nodes is required. The bit-vector values are propagated up through the OR-tree, and the output value of each node provides the *availability* information of a memory chunk of size 2^{n-l}, were l is the depth-level of the node in the tree (the head node is at level 0). This information is used by the AND-tree to generate the address location of the memory block in the bit-vector.

AND tree

The AND-tree uses the availability information from the OR-tree along with with the *Size* information provided by user to find the address of the block in the bit-vector. The nodes of the tree are constructed out of AND gates. The node information from each OR-gate is combined with the *Size* bits and is fed into the AND-tree. This information is propagated up towards the root in such a way that the location of the corresponding block of memory can be found by a non-backtracking search [15] of the nodes. For a bit-vector with 2^n number of bits, a tree with $2^n - 1$ nodes is required.

Bit-Flipper Circuitry

The bit-flipper circuitry is used to update the allocation status of the bits in the bit-vector. For allocation, the corresponding bits must be marked as occupied (i.e, set to 1), and for free operations, they must be cleared. The inputs to the bit-flipper are the starting address of the bits that need to be flipped (provided by the AND tree), and the number of bits to flip (the *Size* bits). Unlike the previous trees, the bit-flipper propagates signals from the root of the tree to the leaves [15]. The bit values available at the leaves

are used to update the status of the bit-vector. Each node in the tree consists of identical combinational logic that allows information from the parent node to be broadcasted to its children. For a bit-vector with 2^n bits, a tree with $2^n - 1$ nodes is required.

The allocator implementation details and results can be found in Chapter 5.

Chapter 4

Region Based Partitioning

4.1 Introduction

This chapter describes a behavioral partitioning scheme for power reduction. It is divided into three major sections: The first section describes the partitioning methodology. The second section describes the clock-gating technique used to implement power-efficient SFSMD controllers, and finally, the last section provides a design example that is used to evaluate the effectiveness of the partitioning scheme.

4.2 Partitioning Methodology

Due to the prevalence of loops in a program, most of the execution time is spent computing a small number of operations. By extracting such loops and implementing them as separate controllers in the SFSMD model, significant power savings can be achieved. This is because the controller for each loop is smaller than the single controller implementing the entire system, and since only one controller is running at any given time, the remaining ones can be deactivated, thus saving power [30].

We propose a partitioning scheme that operates at the behavioral level prior to synthesis. The partitioner redefines the procedure boundaries for the original specification

by first *exlining* loops. Exlining can be defined as the inverse of inlining in which a sequence of statements are replaced by procedure calls [25]. Each exlined loop is then implemented as a separate controller in the SFSMD model. Figure 4.1 illustrates this process.

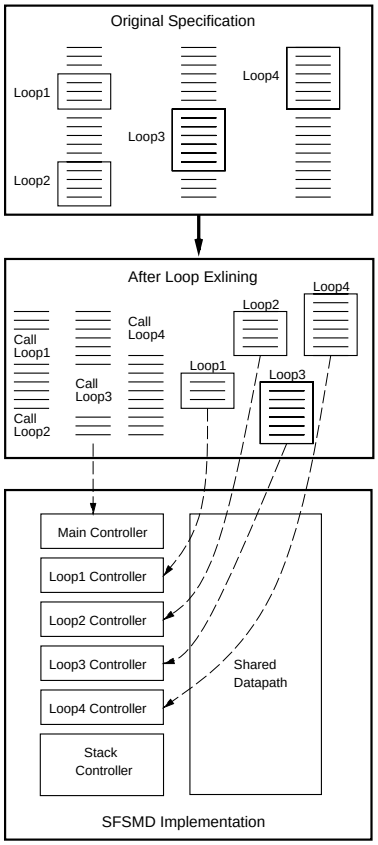


Figure 4.1: Region Based Partitioning

If the loops account for a major portion of the overall execution time, significant power savings can be achieved since overall switching activity is reduced due to the localized activities of each smaller individual controller for the loops [30]. The ability to localize the controller activity is inherent in the SFSMD model where each inactive controller can be disabled by stopping its clock.

The partitioning scheme can be generalized by defining it as a transformation of the original design via a series of exlining *and* inlining operations. Inlining helps improve

parallel-level optimizations by incorporating acyclic instructions from existing procedures into the main body. Inlining also exposes loops within the procedures. The result is a design where the main body is composed entirely of either, a series of acyclic instructions, or procedure calls to exlined loops. This is illustrated in figure 4.2.

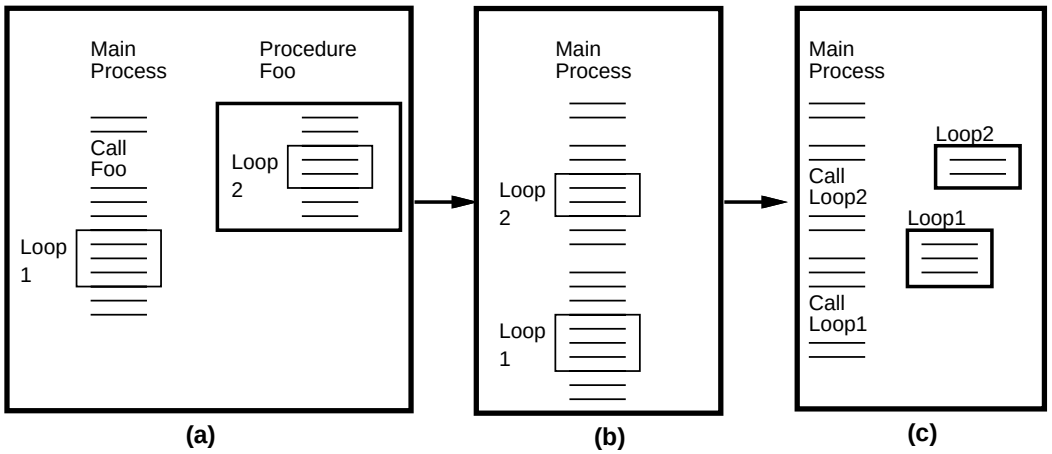


Figure 4.2: Inlining and Exlining Transformations. (a) Original Specification, (b) After inlining *foo*, (c) After loop exlining.

The extraction of loops introduces extra power overhead due to inter-procedural communication between the controllers and due to loss of control-step optimizations across procedure boundaries. Communication overhead occurs due to extra states [1] that need to be inserted to implement procedure calls and returns, switching activities involved in the stack-controller, and the activities of tri-state buffers used for sharing the datapath. This means that loop exlining cannot be done indiscriminantly since high-power overhead loops can result in a design with increased power requirements. What is required is a method for selecting a subset of loops for exlining, such that the reduction of of power far outweighs the power increase due to communication.

[1] Due to the shared datapath, extra states are not required for parameter passing since all controllers have access to the same variables.

4.2.1 Partitioning Power-Index

The loops from a behavioral specification can be extracted for SFSMD implementation in any number of ways as indicated in figure 4.3. If there are n loops at the root-level, then there are 2^n possible ways of partitioning the code - nested loops further increase this value.

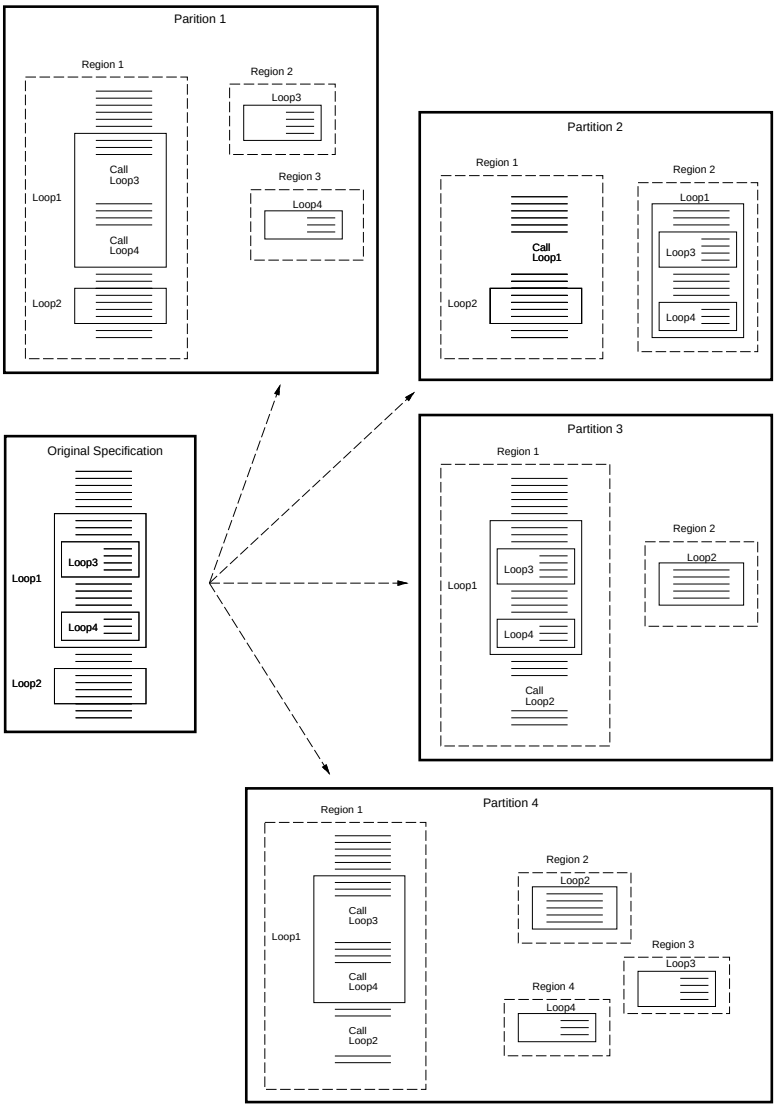


Figure 4.3: Some possible ways of partitioning code

The partitions can be represented by a tree structure as indicated in figure 4.4. Each node in the tree, except for the root, corresponds to a loop region. A child of a node

corresponds to a nested loop within the node. The parent of a node corresponds either to an outer loop or the main process in which the node resides. The root of the tree always corresponds to the main process. Each edge can be weighted with the accumulated number of iterations of the parent node. This represents the total number of calls made to its child.

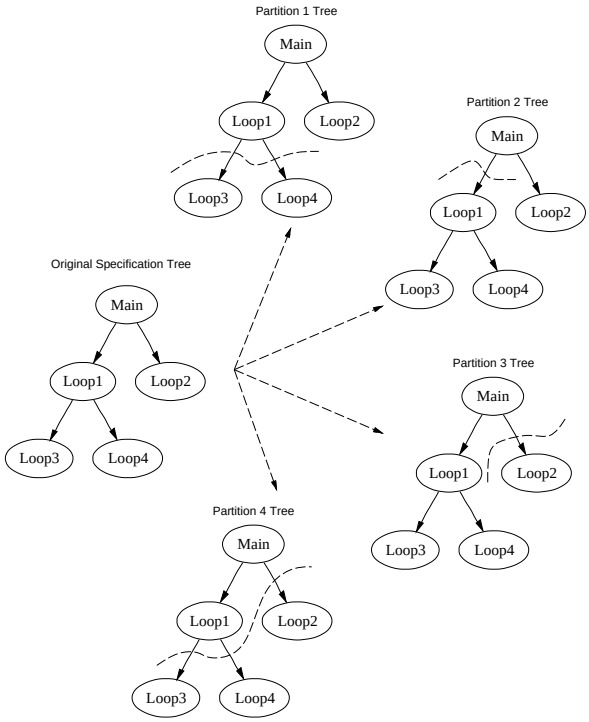


Figure 4.4: Tree Representation of Partitioning

A partition can be simply considered as a "cut" across the nodes of the tree. The cut can be made in any direction - sideways, vertically, diagonally, etc, so there are exponential number of partitions possible. The cuts shown in figure 4.4 correspond to the partitions depicted in figure 4.3. The *regions* of a partition are formed by splitting the tree at points where the cut intersects the edges of the nodes. At each split point, a call instruction has to be added in the parent loop for each child. The total number of calls made to each child is the same as the weight of the edge that was cut.

Two main observations can be made from the partitioning tree: 1) The size of the

controller of a child loop is always smaller than, or equal to the size to the controller of its parent, and 2) the number of times a child loop executes is always greater than, or equal to the number of times its parent executes. These relations occur because the child loops are embedded inside the parent. This means that as we traverse down the tree starting from the root, the controller size of each node *decreases* monotonically while the edge weights *increases* monotonically. Therefore, the leaves of the trees consist of nodes with smallest controllers, but they are also the most often called nodes.

A method is required for selecting the most power efficient partition. Intuitively, since overall switching activity is reduced to the localized switching activities of the individual controllers, the partition with the lowest power would be one in which every node is considered as a separate region. However, this type of splitting would incur the highest communication overhead and could result in increased power consumption. For designs in which the majority of execution time is spent in the inner-most loops, a promising partition would be a horizontal cut at the lowest depth, as indicated by partition 4 in figure 4.4. This partition would result in minimal power consumption as long as the communication overhead was low. The best partition optimally balances the number of extracted regions with communication overhead. A good partitioning algorithm would be able to find the optimal partition by having to examine only a fraction of all the possible partitions. It may be possible to apply the monotonic qualities of the tree to develop such an algorithm. This, however, is considered future work material and is not investigated in this thesis.

Assuming a set of candidate partitions exist, the problem then is of selecting one that maximizes power reduction. Formally, a partition is defined as a collection of k regions R.

$P(S) = \{R_1, R_2, \ldots, R_k\}$ such that $R_i \cap R_j = \varnothing$ for $i \neq j$, and $\bigcup_{i=1}^{k} R_i = S$.

Where R_i consists of the set of basic-blocks corresponding to an exlined loop or main process.

For each partition P_j, a power index is defined as:

$$\mathscr{P}_j = \sum_{i=1}^{k} |States(R_i)| \cdot Cycles(R_i) + K \cdot Calls(R_i) \tag{4.1}$$

$$\text{where } R_i \in P_j$$

$|States(R_i)|$ is the number of control-steps used in the FSM of the controller for region i. It is used to represent the relative "power complexity" of the region. Generating control-steps requires that scheduling be performed on the basic-blocks of the region. Scheduling is typically performed at a later stage during high-level synthesis, and is strongly intertwined with with the allocation phase. However, fast scheduling can be performed at the behavioral level which does not require allocation. For example, "as soon as possible" (ASAP) scheduling can be used to get a quick estimate of the number of control-steps. It does not take resource constraints into consideration (assumes unlimited resources) and follows a simple rule that an assignment to a variable can execute only after the values of its operands have been computed. $Cycles(R_i)$ is the accumulated number of control-cycles spent in region i and can be determined by simulating the original specification. $Calls(R_i)$ represents the number of calls made to region i, and K is a constant for controlling the relative weight of the contribution. $Calls(R_i)$ can be computed by simply simulating the original specification and measuring the frequency of transitions to the region.

Since the regions are active only one at a time, the index expresses the total power of a partition as a sum of the "energy" contributions of the individual regions. The first term represents the energy expended by the controller of the region, and the second term adds the energy for communication. The partition with the lowest power index should be chosen for SFSMD implementation.

4.3 Power Reduction through Clock Gating

The clock gating technique is used to provide a power-efficient implementation of register banks that are disabled in the SFSMD controllers.

Controllers in the SFSMD model are deactivated through the use of *enable* signals. The *enable* signal is connected to the enable input of the control-state registers of the corresponding controller. When the *enable* signal is unasserted, the registers are disabled. Without clock gating, the enable circuitry for the register-banks is implemented by a feedback loop via a multiplexor as shown in figure 4.5 (a).

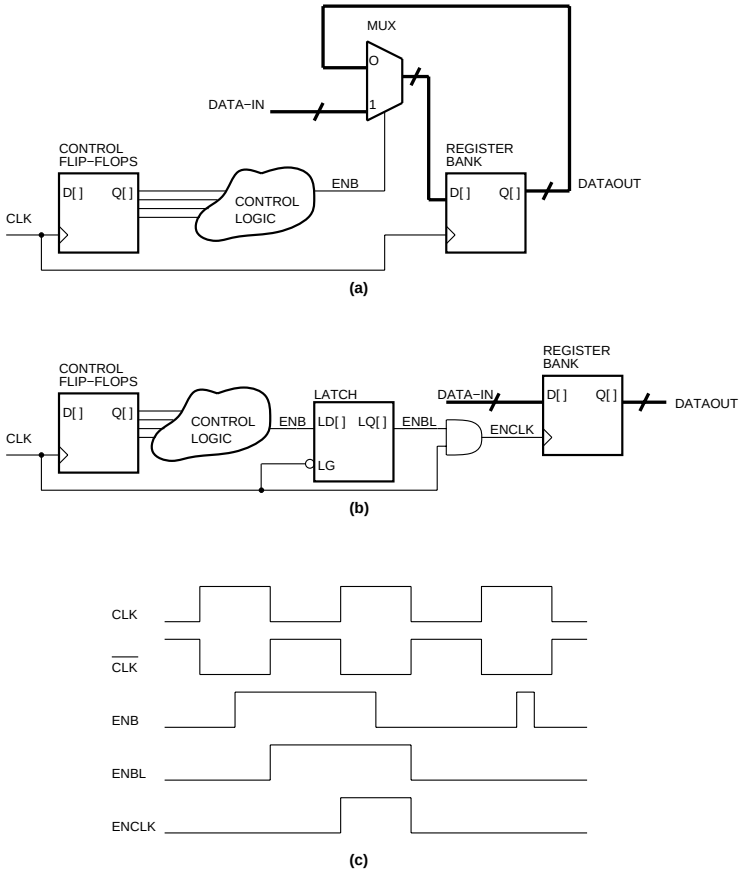


Figure 4.5: Clock-Gating (a) Regular circuitry for disabling registers, (b) Using clock-gating, (c) Timing diagram for clock-gating

When registers maintain the same value for successive clock signals, unnecessary power is expended because the capacitance on the clock net and clock pins of the registers

are still being driven [23]. Furthermore, even though the outputs of the registers do not change, power is still expended due to internal switching within the registers.

Clock-gating handles the enabling of registers by directly controlling the clock of the register banks. It eliminates the need for reloading the same value in the register through multiple clock cycles. Power is saved by eliminating the unnecessary activity associated with reloading register banks and minimizing switching activity on the clock nets and pins.

As shown in figure 4.5 (b), clock-gating eliminates the feedback net and multiplexor by inserting an AND gate in the clock net of the register along with a latch. The waveforms of the signals are shown in figure 4.5 (c). The clock input to the register-bank, *ENCLK* is gated through the AND gate. The gating is controlled by the *ENBL* signal which is derived from the original *ENB* signal. Positive-edge transitions on the *ENCLK* allow the register-bank to be triggered.

The latch prevents glitches on the *ENB* signal from propagating to the register's clock pin. Without the latch, during the logic 1 state of the *CLK* pulse, any glitches on the *ENB* line could propagate and corrupt the register clock signal. The latch eliminates this possibility by blocking signal changes when the clock is at logic state 1 [23].

For designs that have large multibit-registers, such as state-registers of controllers and the register-file of the stack-controller, clock-gating can save power and reduce the number of gates in the design. However, for smaller register-banks, the overhead of adding logic to the clock tree might not compare favorably to the power saved by eliminating a feedback nets and multiplexors [23].

4.4 Design Example

A *blackjack gaming machine* [22] design example is used to illustrate the concept of the region based partitioning. The blackjack machine is first implemented using a regular

FSMD model, in which a control unit and a datapath unit is used. Baseline power and area measurements are taken for this implementation. Then, a loop region in the controller is extracted from the main body resulting in two separate controllers (plus the datapath). This configuration is implemented using the SFSMD model. Power and area measurements for this implementation are compared with the FSMD measurements.

4.4.1 Blackjack Controller

Blackjack is a popular card game played at casinos. It uses the standard 52 card deck, in which the suits of the cards, i.e, spades, clubs, hearts, and diamonds, have no significance and are ignored. The value of the card is important, where the face cards, i.e, Jacks, Queens, and Kings, all have the value 10 and Aces have the value of 1 or 11, depending on what the player chooses.

In blackjack, the highest total value of cards that can be held is 21. The goal of the game is to beat the dealer, who has no other objective than to follow the rules of the casino, which are to hold on hands of 17 or higher, and to draw another card on hands of 16 or less.

If a player's card value is over 21 or less than that of the dealer, he has to declare a bust and he loses. A player can improve his hand by requesting another card from he dealer. This is called drawing. He can hold if he is satisfied with his current hand.

The state-diagram of the blackjack machine is shown in figure 4.6. It is similar to a design described in the book *HDL Chip Design* [22].

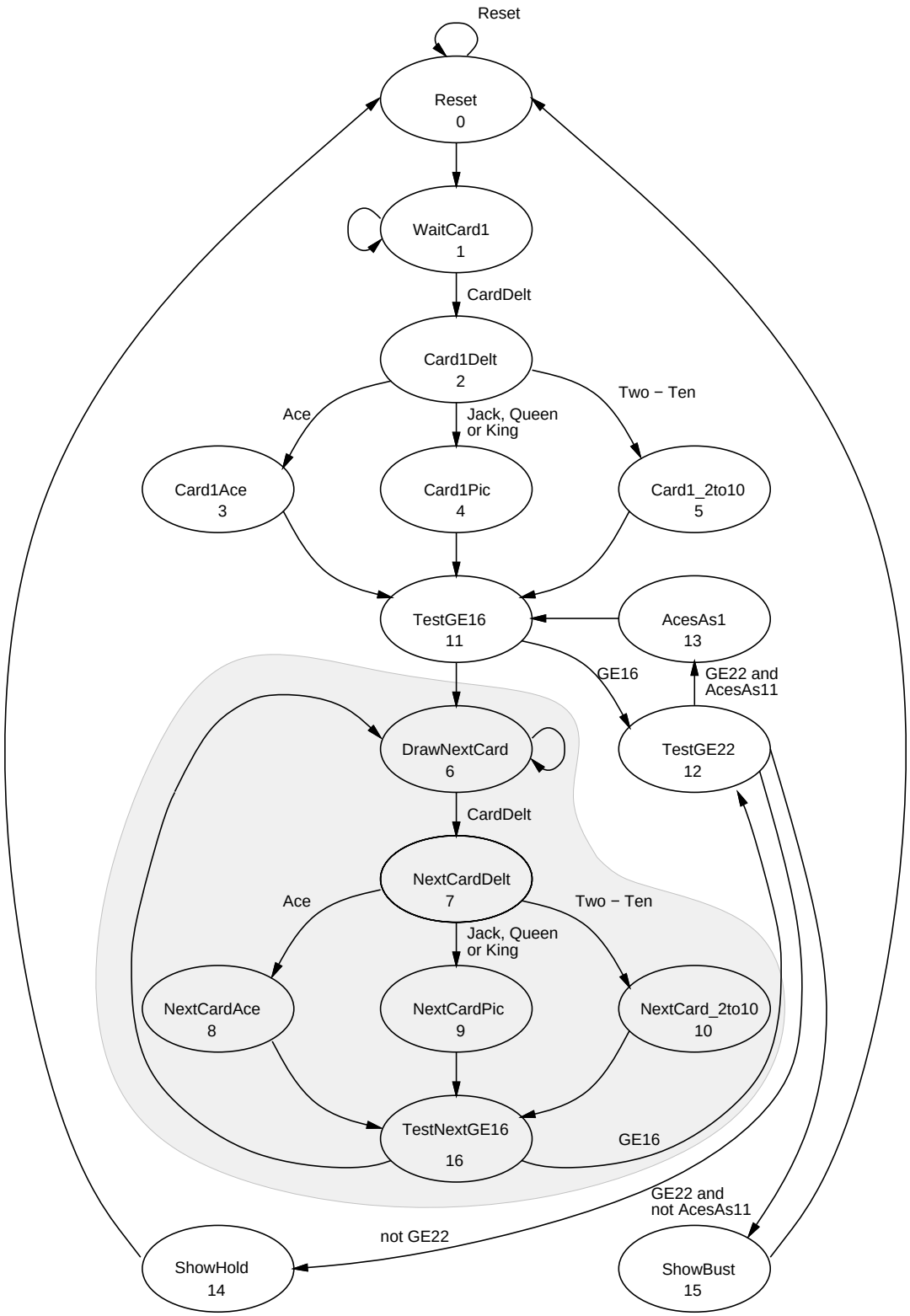


Figure 4.6: Blackjack Controller State-Diagram

The state-diagram defines the controller portion of the FSMD model. The datapath circuit is not shown. For region based partitioning, we need to extract a loop portion for separate implementation that can potentially result in power savings. The highlighted region in the figure is selected. It enters into the *DrawNextCard* state in which the next card is drawn. If the updated total card-value is less than 16, control is passed back to *DrawNextCard*, else it exits out of the region into state *TestGE22*. The loop is formed by the back edge from state *TestNextGE16* to state *DrawNextCard*.

Intuitively, if this region were to be extracted and implemented as a separate controller, maximum power-savings would result if low valued cards were drawn. This is because low valued cards would ensure that most of the execution time is spent within the smaller controller. On the other hand, drawing a sequence of Aces could have an adverse effect on power due to increased interactions between the main controller and the extracted controller.

State-diagrams for the two controllers generated after loop region extraction are shown in figures 4.8 and 4.7. The controllers can be implemented using the SFSMD model which allows the sharing of a common datapath. The datapath is identical to the one used for the FSMD implementation. State *Wait After Call* has been added to the main controller to handle calls to the extracted controller, and state *Wait After Return* has been added to the extracted controller to handle returns.

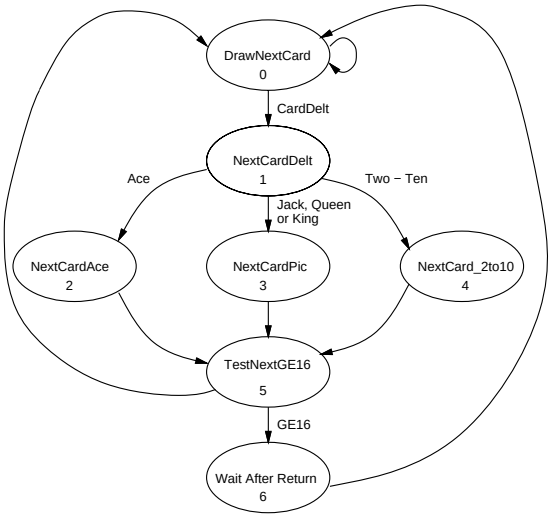


Figure 4.7: Blackjack State-diagram of Loop Region After Extraction

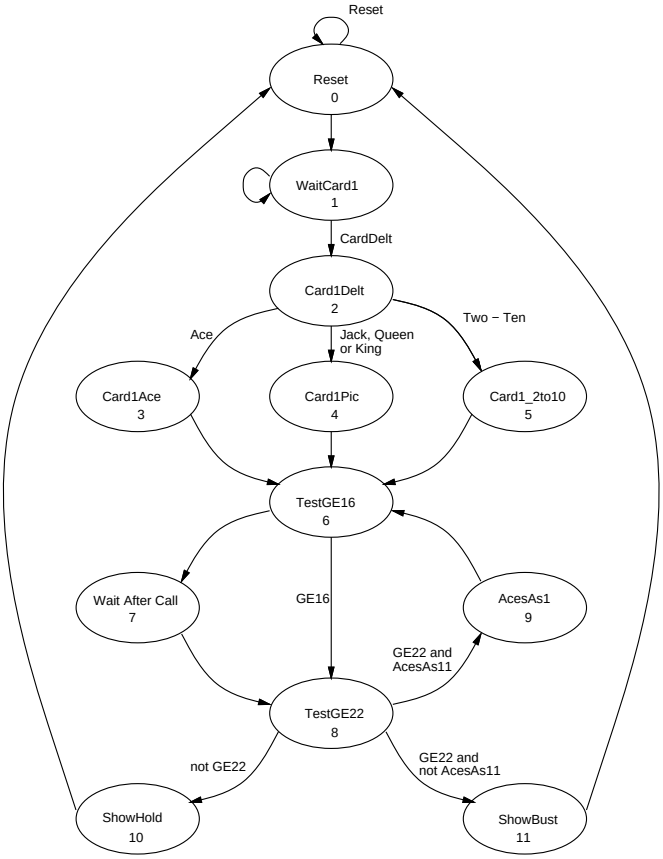


Figure 4.8: Blackjack Sate-diagram of Main Controller Region After Extraction

4.4.2 Implementation Methodology

The design was implemented as follows:

1. For the FSMD implementation, the state-diagram from figure 4.6 was used to describe the controller unit with a corresponding datapath.

   The partitioned design was implemented using an SFSMD model.

   Separate controllers specified by state-diagrams in figure 4.8 and figure 4.7 were described and the appropriate control signals for SFSMD operation were added. Additional components required for SFSMD implementation such as the stack controller and tri-state buffers were also described. The same datapath unit from the FSMD implementation was used. All designs were described in VHDL.

2. The designs were simulated to verify correct operation.

3. The designs were synthesized into gates using the *Design-Compiler* tool of Synopsys[2]. The designs were synthesized using two methods: in the first method, the disabling of register-banks was handled in the standard way, i.e, a multiplexor was used to feed-back outputs to hold the current value. In the second method, clock-gating was used to control the disabling of registers. In the FSMD model, clock-gating is only relevant for datapath registers, since the controller registers are never disabled. In the SFSMD model, clock-gating effects registers in the datapath unit, the partitioned controllers, and the stack controller.

4. The synthesized designs were re-simulated to verify correct functionality, and to capture the switching activity of the gates.

5. Data from switching activity captured during gate-level simulation was used by the *Power Compiler* tool of Synopsys to report power. In addition, area measurements of the design were also made.

[2]The *wcells* CMOSP35 0.35 micron library provided by the Canadian Micro-Electronic Corporation (CMC) was used.

4.4.3 Results of Design Example

The results of the Blackjack controller are tabulated in table 4.1. The power figures are based on switching activity captured by simulating the drawing of low value cards.

As per the results, appreciable power savings due to partitioning occurred only when clock gating was used to control the disabling of registers. Furthermore, clock gating also decreased the area overhead compared with the non-gated implementation. In this example, the SFSMD implementation achieves a maximum 11.6% improvement in power over the FSMD approach, with an area overhead of 26.8%, and execution time overhead of 5.7 %.

Clock Gating Used	**FSMD Implementation**		**SFSMD Implementation**			
	Power (μ Watt)	**Area**	**Power (μ Watt)**	**Area**	**% Power Reduction**	**% Area Overhead**
No	248.9	22447	240.2	29322	3.5%	30.7%
Yes	217.7	21907	192.5	27784	11.6%	26.8%

Table 4.1: Power and Area Results for Blackjack Game Machine

The area break down of the FSMD and SFSMD implementations in terms of internal components is shown in tables 4.2 and 4.3 respectively.

Clock Gating Used	**Component**	**Area**	**% of Total Area**
No	Controller	10004	44.6%
	Datapath	12443	55.4%
	Total	**22447**	**100%**
Yes	Controller	10004	45.7%
	Datapath	11903	54.3%
	Total	**21907**	**100%**

Table 4.2: Area Break-Down For FSMD Implementation

Clock Gating Used	Component	Area	% of Total Area
No	Main Controller	7766	26.5%
	Loop Controller	4887	16.7%
	Stack Controller	2850	9.7%
	Tri-State Buffers	12443	4.7%
	Datapath	12443	42.4%
	Total	**29322**	**100%**
Yes	Main Controller	7116	25.6%
	Loop Controller	4539	16.3%
	Stack Controller	2850	10.2%
	Tri-State Buffers	1376	4.9%
	Datapath	11903	42.8%
	Total	**27784**	**100%**

Table 4.3: Area Break-Down For SFSMD Implementation

Table 4.4 shows results in which the power figures are based on the switching activity captured by simulating the drawing of Aces. As expected, due to the increased communication between the extracted loop and the main body, the SFSMD implementation is not able to reduce power.

This highlights the importance of profiling the application prior to loop extraction, since clues may be revealed regarding the amount of inter-controller switching activity. This information can aid in the decision to extract loops.

Clock Gating Used	FSMD Implementation	SFSMD Implementation	
	Power (μ Watt)	Power (μ Watt)	% Power Reduction
No	251.5	271.1	-1.1%
Yes	217.1	216.3	0.4%

Table 4.4: Power Results Based on High Controller Inter-Communication

Dramatic power reduction through region partitioning can be demonstrated by modifying the blackjack controller. Let us suppose the blackjack controller described in figure 4.6 is changed so that while its waiting for cards to be dealt in states *WaitCard1* and *DrawNextCard*, it generates output pulses. These output pulses could be used to indicate to an external source that the controller is waiting for a card to be dealt. The generation of pulses would require additional states to be added, into which *WaitCard1* and *DrawNextCard* toggle in and out. This modification forms a loop which can be extracted for implementation as a separate controller. Also, assume the blackjack controller is playing with a human dealer, whose response to card requests is several orders of magnitude slower than the controller. Therefore, most of the time is spent toggling in the loops described.

The results of SFSMD implementation based on the extraction of the loop described is presented in table 4.5. Power figures are based on the switching-activity captured by simulating the slow servicing of cards. A power reduction of over 53% is obtained for this design.

Clock Gating Used	**FSMD Implementation**		**SFSMD Implementation**			
	Power (μ Watt)	**Area**	**Power (μ Watt)**	**Area**	**% Power Reduction**	**% Area Overhead**
No	208.5	22355	160.9	29046	22.8%	29.9%
Yes	208.3	22809	97.0	27868	53.4%	22.2%

Table 4.5: Dramatic Power Savings by Region Partitioning

In this example, significant gains in power reductions are made because compared with the original controller, the extracted loop is tiny, and most of the time is spent running this loop.

Chapter 5

Experiments and Results

5.1 Introduction

This chapter describes region based partitioning experiments performed on loop-intensive *C* benchmark kernels. The results are used to evaluate the effectiveness of applying this technique to substantial designs described in HLLs. The experiments are also used to determine how well the values of the power-index equation correlate with the actual power of the partitions. Additionally, the synthesis results of the dynamic memory allocation core are also studied. The quality of the core is evaluated by studying how well it scales with modern process technologies.

5.2 Region Partitioning Experiments

The region based partitioning was applied to various *C* benchmark kernels in order to evaluate its impact on power consumption. The benchmarks used were based on the Livermore kernels [32]. These kernels were chosen because their loop-intensive nature made them appropriate for loop extraction, and also because they could be synthesized in reasonable amounts of time. Minor modifications were made to some of the kernels in order to make them suitable for the experiments. These modifications included adding

initialization code to the beginning of the benchmarks in order to initialize variables and arrays, and in some cases, removing a level of loop nesting in order to reduce synthesis time.

Ideally, an automated tool would have been used to partition the kernels and to generate the appropriate SFSMD structure for synthesis. However, due to the lack of such automation, generating the complete SFSMD structure for each partition of a kernel was infeasible. In the *blackjack controller* example of chapter 4, a full manual implementation of the SFSMD model was possible because of the manageable size of the design and because only one partition was investigated. The *C* kernels, however, result in much larger designs with hundreds of control-states, and multiple partitions have to be considered. Since power measurements of complete SFSMD implementations were not possible, indirect measurements were made by summing up the power contributions of the component regions that comprise a partition. As will be explained later, this is a reasonable approach since in the SFSMD model, only one region controller is active at any time. Furthermore, since the partitioning technique is used for reducing the controller power, only controller power figures are compared - the datapath power is not considered since it is the same for both FSMD and SFSMD implementations (both use identical datapaths). Therefore, these experiments compare the reduction in *controller* power of region based partitioned designs over unpartitioned ones.

The experimental methodology is summarized below:

1. Each *C* kernel was first profiled to determine the potential loop regions and the number of calls made to each region.

2. The kernel was then compiled into VHDL using the *eXcite* synthesis tool from Y-Explorations Inc. It generates an FSMD structure with a separate controller and datapath. It also reports design statistics such as the total number of states in the controller.

3. The VHDL design was simulated in order to verify correct functionality and to capture switching activity. Additionally, the total number of cycles required to complete the simulation and the number of cycles spent in each region were recorded.

4. The controller portion of the VHDL design was synthesized into gates using the *Design Compiler* tool from Synopsys. Data from switching activity captured during simulations was used by the *Power Compiler* tool of Synopsys to report power. Area measurements of the design were also recorded. These measurements formed the base-line results for the unpartitioned design.

5. Each kernel was then manually split up into various partitions. This was done by studying the kernels, and manually extracting the region portions of the code into separate *C* files.

6. The different regions corresponding to each partition were compiled into VHDL.

7. The controller portion of each of the region was synthesized into gates using *Design compiler*. Power and area measurements of the region were made.

8. The total effective controller power for each partition was calculated by summing up the energy contributions of each region, and dividing that by the total execution time. The area of each partition was calculated by summing the areas of the component regions. These power and area figures were compared against the unpartitioned design.

The *wcells* CMOSP35 0.35 micron library provided by the Canadian Micro-Electronic Corporation (CMC) was used by *Design compiler* to synthesize the design.

5.2.1 Partitioning Methodology

As discussed earlier, the number of partitions of a program is exponential in the number of loops present. Since we were manually extracting and processing the partitions, the

search space had to be decreased due to time constraints. This was done by limiting the number of partitions of a kernel to the maximum loop depth in the specification. For example, in a design with three root-level loops, only two partitions would be considered : The first partition would consist of only one *region* - the original specification without exlining. The second partition would be composed of four *regions* consisting of three exlined loops along with the main procedure which calls the loops.

In general, if the maximum nesting-level of a loop in a specification was m (where root level loops are considered to be at level 1), the number of partitions considered were $m + 1$, where each partition would exline nested loops at the corresponding level. Figure 5.1 shows an example of three partitions resulting from a specification with a level 2 nested loop. Partition 0 always corresponds to the unpartitioned design. In order to reduce communication overhead, the parents of nested loops were not exlined as the loop depth was traversed - this is indicated in Partition 2 of the figure.

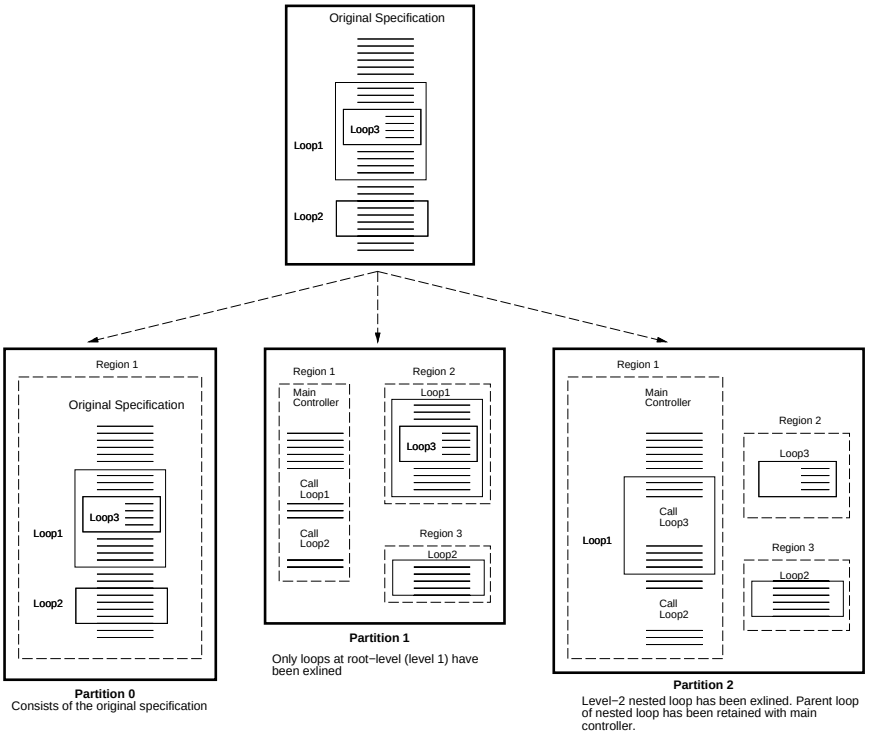


Figure 5.1: Partitions Considered

The partitioning strategy can also be visualized by using the tree-representation that was described in the previous chapter. The partitions are formed by making horizontal cuts across the tree as each depth level is traversed as indicated in figure 5.2. Due to the tree-structure, horizontal cuts allow the widest range of region-granularity to be exercised in the shortest number of steps. Therefore, for each kernel, power figures are available for partitions with regions ranging from the highest granularity (unpartitioned) all the way to very low (the deepest level nested loop have been exlined).

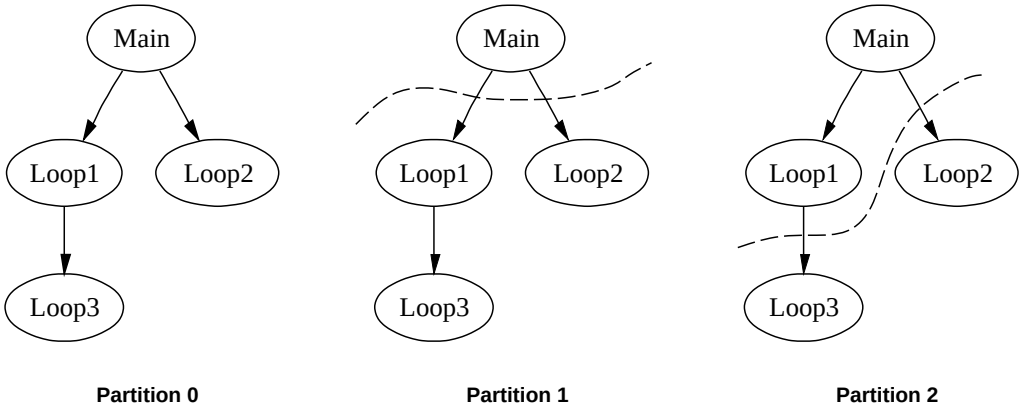


Figure 5.2: Tree-Representation of Partitions Considered

This strategy is effective in finding low power partitions for designs in which most of the execution time is spent in the inner most loops. This is because the horizontal cuts insure that the deepest nested level loops will be ultimately extracted and implemented as separate controllers. Since these controllers represent regions of the finest granularity, their power consumption will be very low.

5.2.2 Measuring Power Of The Partitions

The power of a partitioned design was calculated by summing the energy contributions of the individual regions and dividing it by the total time of the simulation. The equation used to calculate the power for each partition is shown:

$$Power = \frac{1}{Cycles_{Tot}} \sum_{i=1}^{k} Cycles_{Ri} \cdot Pwr_{Ri} + Pwr_{Stack} \cdot Calls_{Ri} \tag{5.1}$$

where Ri is a region of the partition

$Cycles_{Tot}$: This is the total number of cycles used to complete the simulation of the unpartitioned design. Cycle overhead for regions calls and returns were added for the different partitions.

$Cycles_{Ri}$: This is the accumulated number of cycles spent executing region i. This was obtained from simulations of the unpartitioned design. Cycle overhead for region calls and returns were added for the different partitions.

Pwr_{Ri}: This is the intrinsic power of the controller associated with region i. This value was obtained by using *Power Analyzer* to report power for the synthesized controller of the region. *Power Analyzer* calculates power based on the switching activity of the nets in the design. If the primary inputs of the design are not annotated, is assumes a certain toggle rate and switching probability in order to propagate switching activity to the unannotated nets. This is reasonable for data bus lines, however, inappropriate for FSMs, since in FSMs, certain input signals such as *Reset*, tend to stay idle for long periods of time. The power accuracy can be improved by annotating the primary inputs of the design with switching activity captured during simulation. *Power analyzer* then uses internal zero-delay simulation to propagate switching activity through the unannotated nets. For highest power accuracy, all the gates in the design can be annotated with switching activity captured during simulation.

In these experiments, since only the unpartitioned design could be fully simulated, it was not possible to fully annotate all the gates of the partitioned controllers. Instead, only the primary inputs of the partitioned controllers were annotated from the switching activity captured during simulations. The accuracy of this method was confirmed to be

within 6% of the full gate annotation method. This was done by comparing the power results of the full gate annotation with the primary input annotation for the unpartitioned designs.

Pwr_{Stack}: This is the intrinsic power of the Stack controller. It was obtained by synthesizing a Stack controller and measuring its power with *Power Analyzer*.

$Calls_{Ri}$: This is the number of calls made to region i. This was obtained by profiling the kernels.

The value in the summation represents the total energy contributed by each region i. The first term represents the energy expended exclusively by the region's controller, and the second term adds the energy overhead of calling the region. Adding up the energy contributions of all the regions and dividing it by the total number of cycles gives the effective power of the partition. Notice that for the unpartitioned design, this equation is simply equal to the power of the unpartitioned controller. The equation assumes that while one controller is active, the energy contributions from all other controllers are negligible. This is reasonable since in the SFSMD model, the power consumption of all inactive controllers are reduced through clock-gating. This can be even further reduced by employing the well known technique of operand isolation: since power can be consumed due to the propagation of switching activities on its inputs (via changes on *status* and external input lines), additional isolation logic (AND or OR gates) can be inserted along with the *enable* signal to hold the inputs stable whenever the controller is not being used. The energy expenditure of the SFSMD tri-state buffers is not considered in this equation because their power consumption would normally be very low.

5.2.3 Power Results

The power results for partition levels 1, 2, and 3 are shown in tables 5.1, 5.2, and 5.3, respectively. The first column lists the benchmark kernels used. The increase in execution time for each partition over the unpartitioned implementation is shown in the second

column. The next two columns list the power and energy values of the partitions, and the last two columns show the power and energy decrease over the unpartitioned design. Partition 1 consists of designs in which root level loops have been extracted for separate implementation. Of the twenty-three benchmarks[1], only thirteen had nested loops, and were hence, able to be partitioned into level 2. Of these, five kernels had another level of loop nesting, and were able to be partitioned into level 3. In this partition, the deepest nested loops were extracted and implemented as separate controllers.

Benchmark	**Partition 1**				
	Time Increase (%)	Power ($mWatt$)	Energy ($nJoule$)	Power Decrease (%)	Energy Decrease (%)
LL1_int	4.6	1.1	10.3	13.2	9.2
LL2_int	0.7	2.7	154.7	12.0	11.4
LL3_int	7.0	1.2	7.1	19.2	13.6
LL4_int	1.7	2.0	47.7	14.6	13.1
LL5_int	3.4	1.2	13.9	16.5	13.6
LL6_int	1.1	1.9	69.9	12.3	11.4
LL7_int	2.1	2.0	38.4	33.6	32.2
LL8_int	1.8	6.9	511.6	20.0	18.6
LL9_int	5.0	2.8	47.0	19.6	15.6
LL10_int	1.6	3.3	85.0	31.4	30.3
LL11_int	6.3	1.0	6.4	31.9	27.6
LL12_int	6.5	1.0	6.4	33.4	29.1
LL13_int	1.1	2.7	104.8	32.9	32.1
LL14_int	0.8	1.5	85.0	42.7	42.2
LL15_int	1.3	2.3	73.0	48.7	48.1
LL16_int	6.0	2.5	26.2	56.7	54.1
LL18_int	0.3	6.5	1028.0	13.0	12.8
LL19_int	1.5	1.9	49.4	14.2	12.9
LL20_int	0.3	3.1	469.9	16.0	15.8
LL21_int	0.9	2.0	99.1	24.5	23.8
LL22_int	1.7	1.1	25.9	45.3	44.4
LL23_int	1.7	3.4	82.7	20.6	19.3
LL24_int	5.0	1.5	12.5	29.9	26.4
Average	2.7	2.4	132.8	26.2	24.2

Table 5.1: Controller Power Results for Partition Level 1

The reduction in power consumption across all partitions ranged between 12.0 % and 67.7 %, with a corresponding reduction in energy ranging between 11.4 % and 67.1 %.

[1]LL17_int was not synthesized since the entire kernel consists of just one loop and so partitioning cannot be applied.

Benchmark	**Partition 2**				
	Time Increase (%)	Power ($mWatt$)	Energy ($nJoule$)	Power Decrease (%)	Energy Decrease (%)
LL2_int	1.4	2.4	138.4	21.8	20.7
LL4_int	4.8	1.9	457.5	20.5	16.7
LL6_int	1.1	1.6	57.8	27.5	26.7
LL8_int	2.6	6.3	471.4	26.9	25.0
LL14_int	3.8	1.7	102.6	32.8	30.3
LL15_int	2.6	2.1	66.4	53.3	52.7
LL18_int	1.8	2.4	387.8	67.7	67.1
LL19_int	5.4	1.6	44.6	25.4	21.4
LL20_int	0.5	2.8	426.6	24.0	23.6
LL21_int	1.3	1.9	94.9	28.0	27.1
LL22_int	2.5	1.0	24.1	49.5	48.2
LL23_int	2.1	3.3	79.4	24.2	22.6
LL24_int	7.5	1.3	10.9	40.4	36.0
Average	2.9	2.3	150.0	34.0	32.1

Table 5.2: Controller Power Results for Partition Level 2

Benchmark	**Partition 3**				
	Time Increase (%)	Power ($mWatt$)	Energy ($nJoule$)	Power Decrease (%)	Energy Decrease (%)
LL2_int	3.9	1.5	87.9	51.5	49.7
LL6_int	5.0	1.4	51.9	37.3	34.2
LL15_int	2.5	2.1	67.5	53.1	51.9
LL21_int	1.7	1.7	84.4	36.2	35.1
LL23_int	2.5	3.0	72.9	30.6	28.9
Average	3.1	1.9	72.9	41.8	40.0

Table 5.3: Controller Power Results for Partition Level 3

The average power reduction for partition levels 1, 2, and 3, were 26.2 %, 34.0 %, and 41.8 %, respectively, while the energy reduction for the partitions were 24.2 %, 32.1 %, and 40.0 %. Due to the low cycle-time overhead for *call* and *return* operations, the energy reduction for each benchmark was very close to its power reduction (averaging within 2 %). As per the results, power reduction improves with increased partitioning levels. This makes sense because the majority of execution time for the kernels was spent in the inner-most loops. Since these loops represent the finest-grained regions, they have the smallest controllers and consequently consume the least amount of energy.

Furthermore, the power contribution of the stack controller was found to be very small, so the communication overhead for the higher partition were insignificant.

Power reduction plots in figures 5.3, 5.4, and 5.5 illustrate the effects of the different partitioning levels on power consumption. By comparing the power reductions between partitions 1 and 2, we see that for all benchmarks, except *LL14_int*, partition 2 always out-performs partition 1. *LL14_int* produces different results because for that benchmark the power of the root-level loop region (partition 1) is slightly lower than the power of the nested loop regions (partition 2). This occurred because a larger design got synthesized for the nested loop region and so it consumed more power. For benchmark *LL18_int*, partition 2 shows a dramatic power reduction over partition 1. This is because the nested loops in partition 2 are much smaller and consume less power than the root level loop.

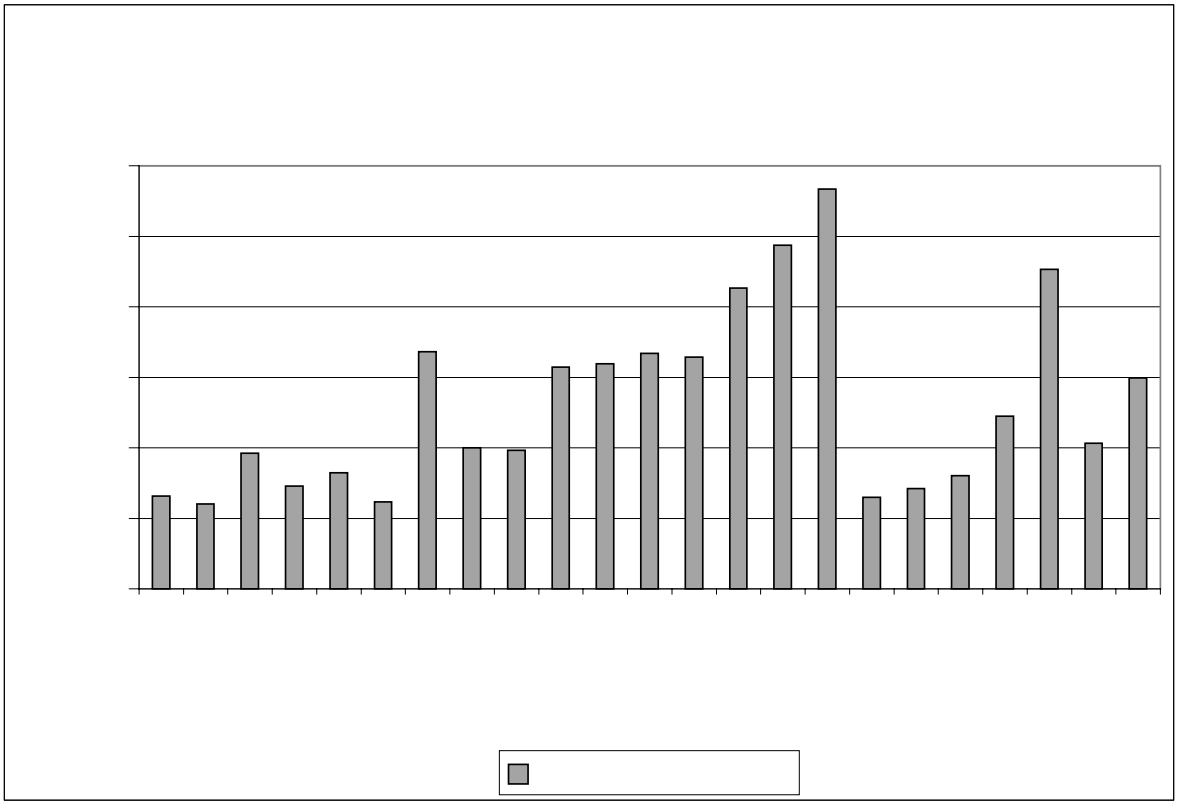


Figure 5.3: Power Reduction for Partition Level 1

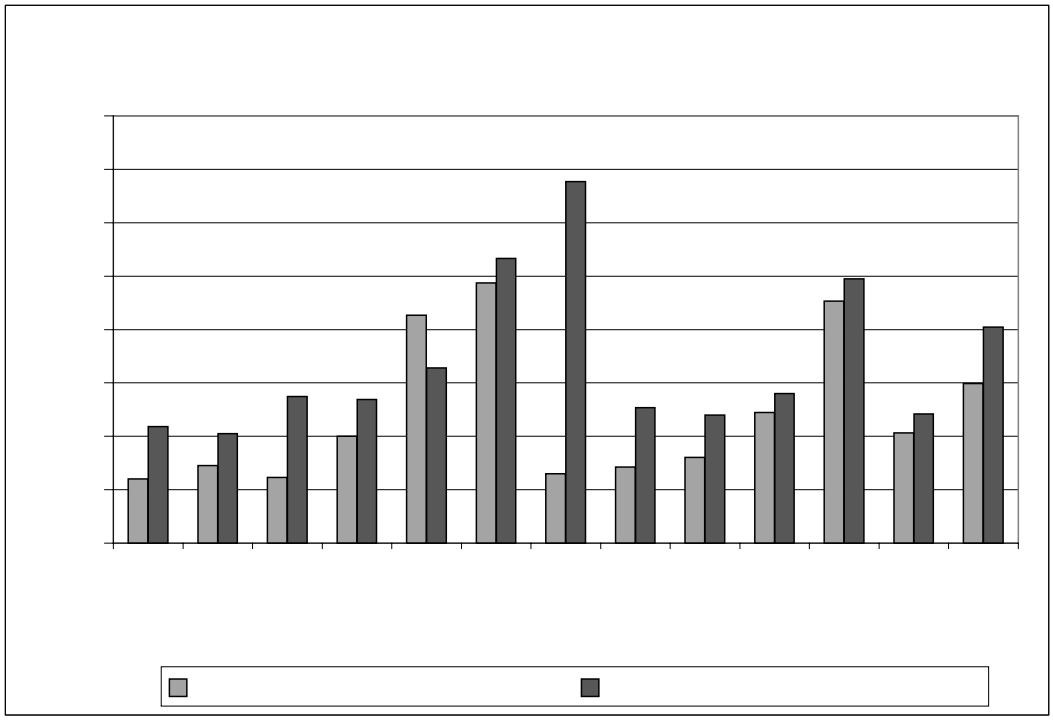


Figure 5.4: Power Reduction for Partition Levels 1 and 2

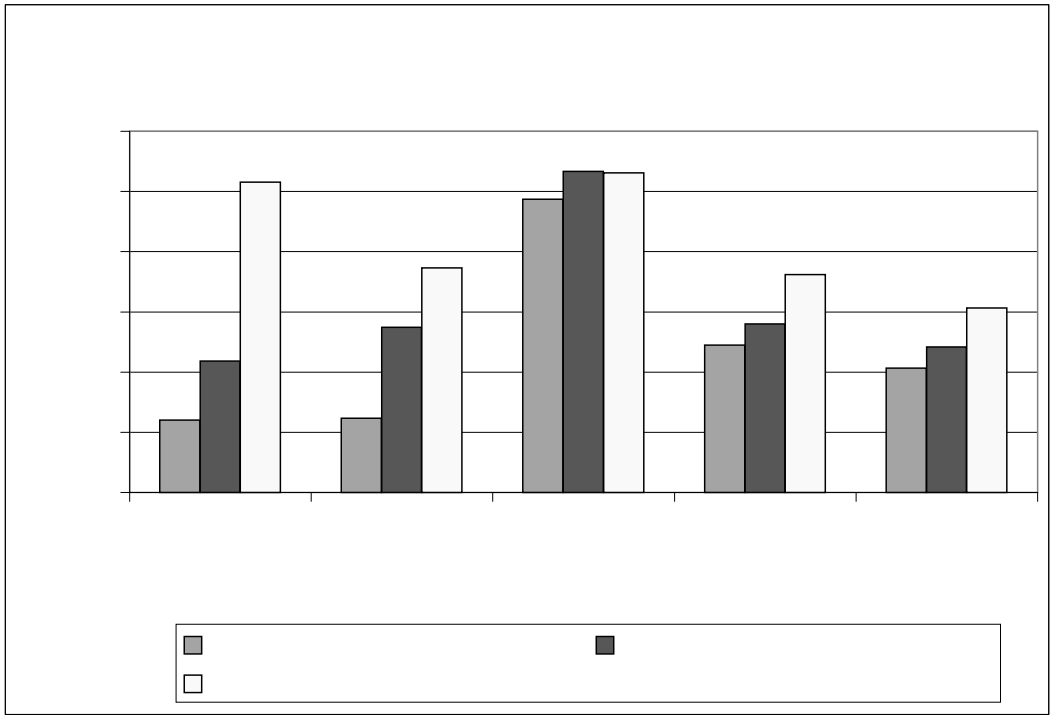


Figure 5.5: Power Reduction for Partition Levels 1, 2 and 3

As per figure 5.5, partition level 3 out-performs partition level 2 for majority of the benchmarks. Once exception is benchmark *LL15_int*, in which partition level 2 results are slightly better than level 3. This is a situation in which the power figure reported for a larger circuit (partition 2) is lower than the power reported for a smaller one (partition 3). This anomaly is generated because the power values happen to be outside the noise-margins of the power measurement technique.

The fidelity of the power-index that was introduced in chapter 4 is also evaluated. We define the fidelity as a measure of how well the calculated power index value of a partition correlates with its actual power. This is done by comparing the power plots of the partitions against the power-index plots. Fidelity is checked by confirming that for each benchmark, the ordering of the relative increase or decrease of the partitions in the index matches the ordering of the partitions in the power plots. These are indicated in figures 5.6, 5.7, 5.8, 5.9, 5.10, and 5.11. The value of 0.74 was used for constant K in the power-index equation (4.1). This value was determined by using the following formula:

$$K = \frac{Pwr_{Stack}}{< Pwr_{Region} >} \cdot < |States(Region)| >$$

Pwr_{Stack} is the power of the stack-controller, $< Pwr_{Region} >$ is the average power of the considered regions, and $< |States(Region)| >$ is the average number of ASAP scheduled control-steps in the FSMs of the considered regions. The resulting value represents the "power-complexity" of the stack-controller as an equivalent number of control-steps which is consistent with the equation 4.1. For these benchmarks, the first two terms were obtained from the power measurements of the stack controller and the various regions. However, in order to apply the power-index equation at the behavioral level, these values must be known a priori to synthesis. They can be obtained by synthesizing a set of sample circuits and making the appropriate power measurements.

In the majority of the cases, the results of the power-index matched the partitioned

power results. An exception was *LL15_int*, for which the power index choose the wrong partition due to the anomaly mentioned earlier for this particular benchmark. Also, for benchmarks *LL14_int* and *LL18_int*, the power-index ordering for partitions 1 and 2 did not match the power results. Higher values for K were also tested for the power-index equation with no significant reduction in fidelity. This was due to the low power consumption of the stack-controller and the relatively few number of calls that were made.

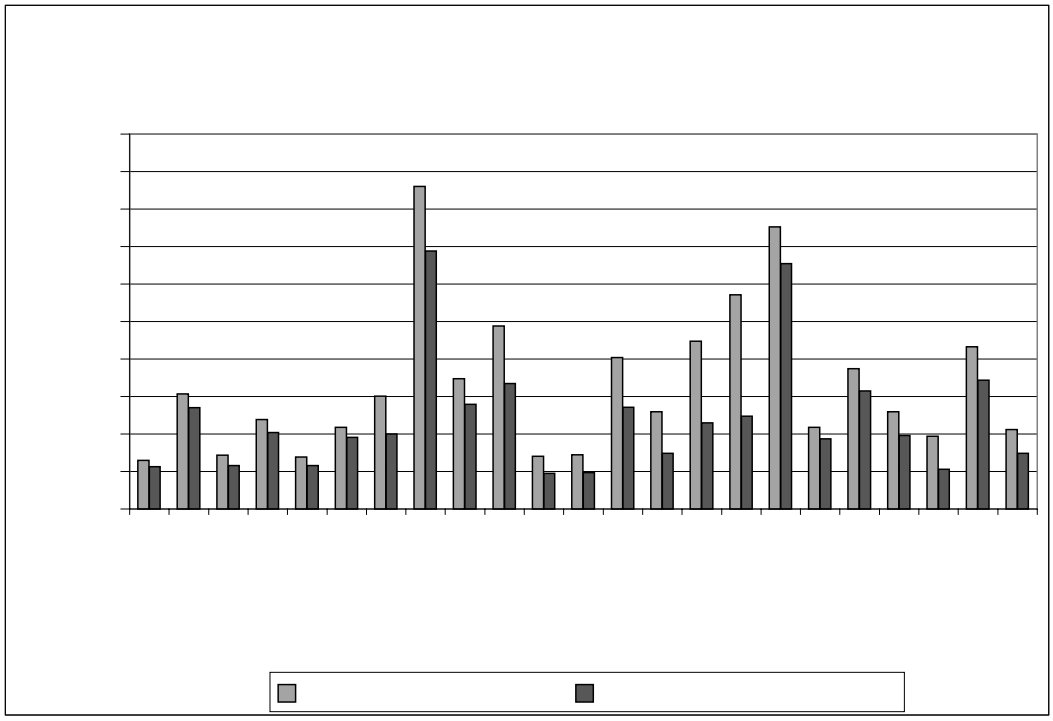


Figure 5.6: Power for Partition Level 1

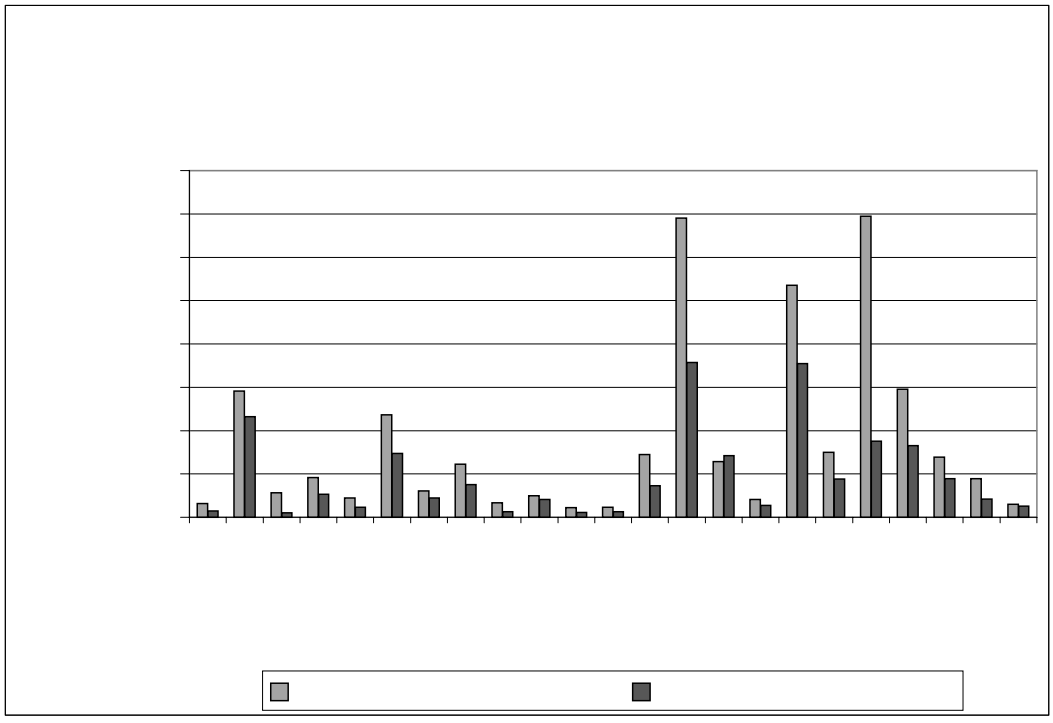


Figure 5.7: Power-Index for Partition Level 1

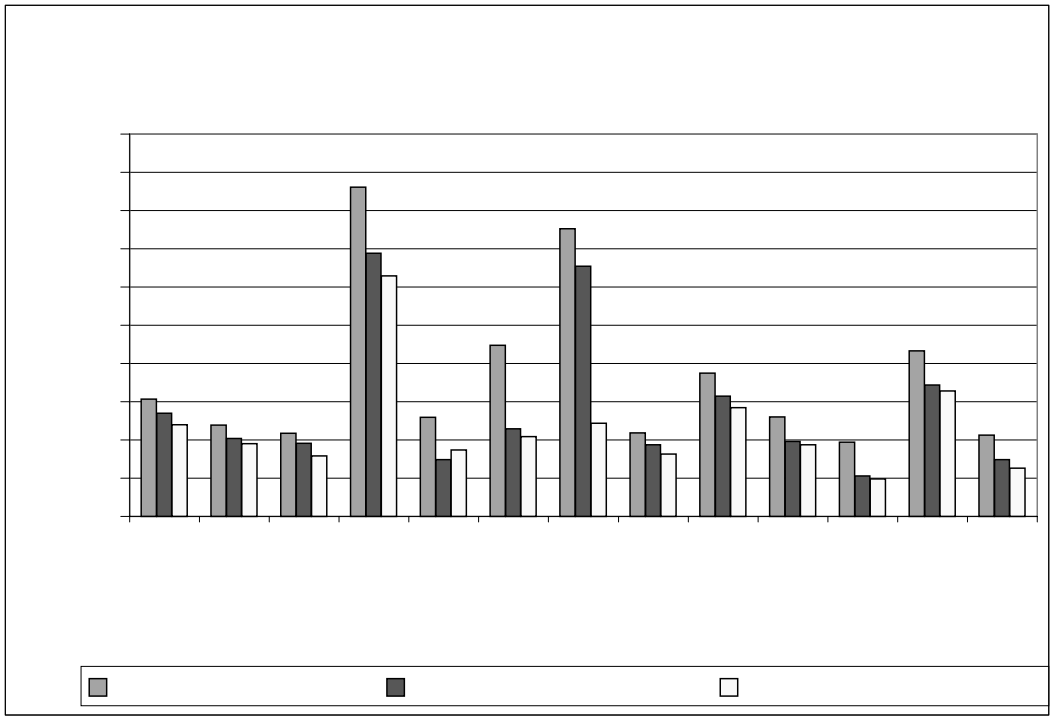


Figure 5.8: Power for Partition Levels 1 and 2

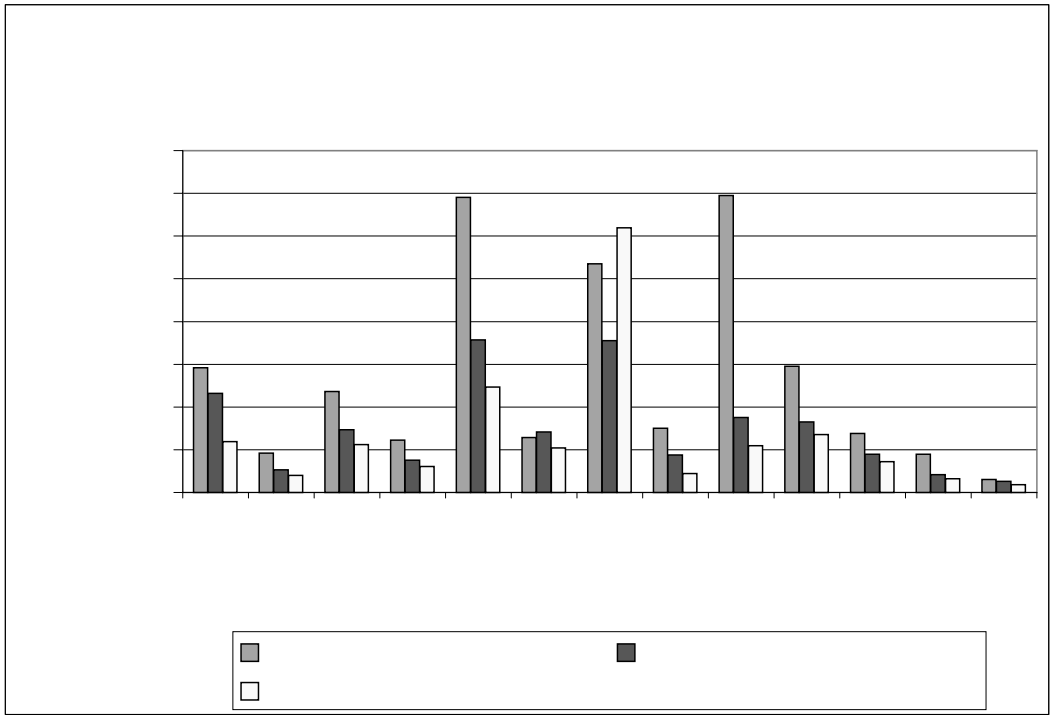


Figure 5.9: Power-Index for Partition Levels 1 and 2

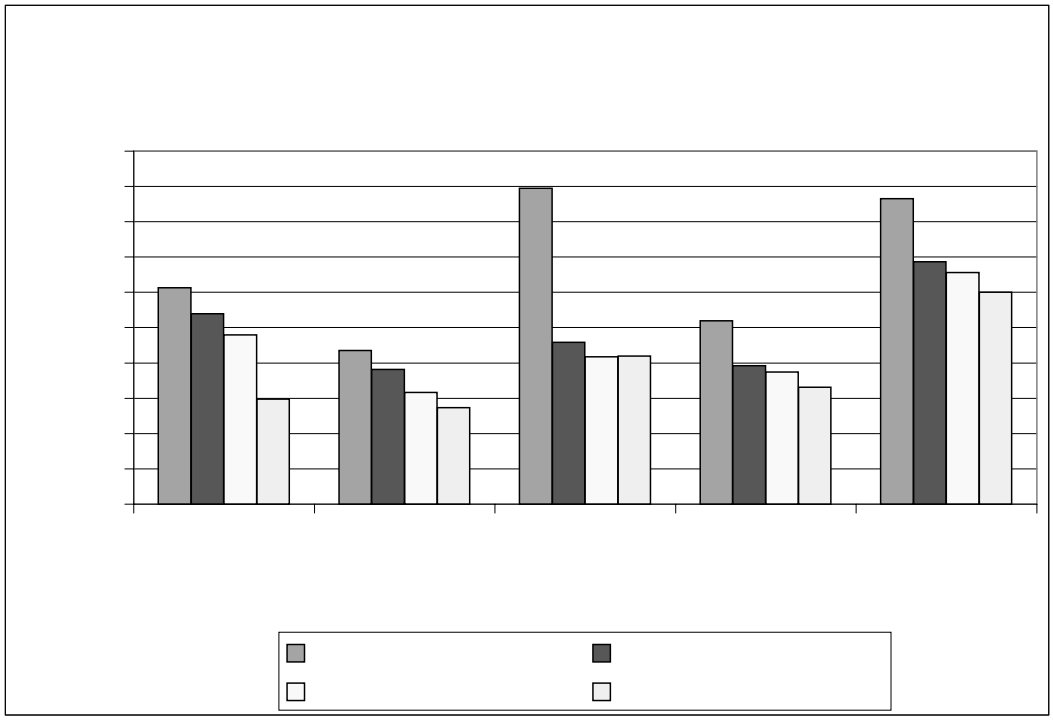


Figure 5.10: Power for Partition Levels 1,2 and 3

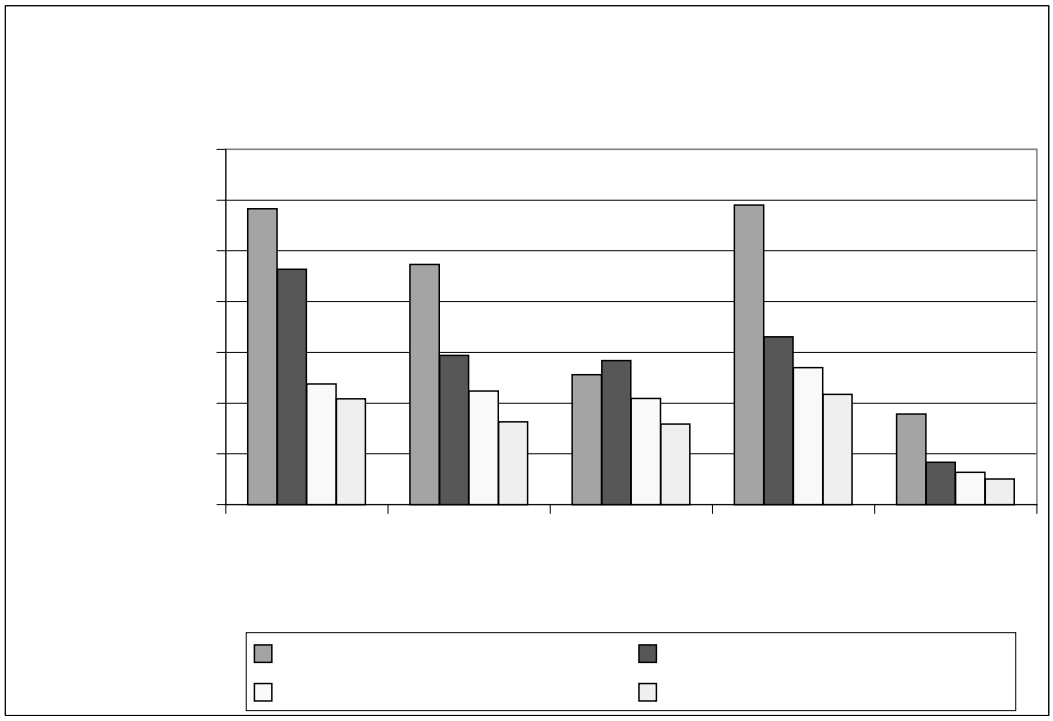


Figure 5.11: Power-Index for Partition Levels 1,2 and 3

5.2.4 Area Results

The area results of the partitioned controllers are listed in table 5.4. The average area increase for partitions 1, 2, and 3 over the unpartitioned design are 5.96%, 6.70%, and 3.96%, respectively.

Benchmark	**Unpartitioned**	**Partition 1**		**Partition 2**		**Partition 3**	
	Area	Area	Area Increase (%)	Area	Area Increase (%)	Area	Area Increase (%)
LL1_int	142356	196773	38.23	-	-	-	-
LL2_int	330894	352402	6.5	137088	4.92	320350	-3.19
LL3_int	112583	141155	25.38	-	-	-	-
LL4_int	268111	291021	8.54	291295	8.65	-	-
LL5_int	138117	162227	17.46	-	-	-	-
LL6_int	249298	286685	15.00	290745	16.63	275813	10.63
LL7_int	316115	423938	34.11	-	-	-	-
LL8_int	1421889	1355959	-4.62	1295958	-8.86	-	-
LL9_int	427026	481570	12.77	-	-	-	-
LL10_int	603090	617211	2.34	-	-	-	-
LL11_int	127339	128141	0.63	-	-	-	-
LL12_int	126425	127538	0.88	-	-	-	-
LL13_int	539627	553208	2.52	-	-	-	-
LL14_int	426624	367535	-13.85	435819	2.16	-	-
LL15_int	715232	554828	-22.43	570866	-20.18	556350	-22.21
LL16_int	671323	524760	-21.83	-	-	-	-
LL18_int	1366023	1414442	3.54	1310792	-4.04	-	-
LL19_int	259396	273544	5.45	376897	45.30	-	-
LL20_int	506722	493998	-2.51	502666	-0.80	-	-
LL21_int	316029	393438	24.49	410840	30.00	402585	27.38
LL22_int	199219	183992	-7.64	186619	-6.32	-	-
LL23_int	602398	648684	7.68	665963	10.55	645557	7.16
LL24_int	172040	179850	4.54	187663	9.08	-	-
Average Area Increase:		5.96 %		6.70 %		3.96 %	

Table 5.4: Controller Area Results of Partitioning

The area overhead plots for the different partitions have been included below:

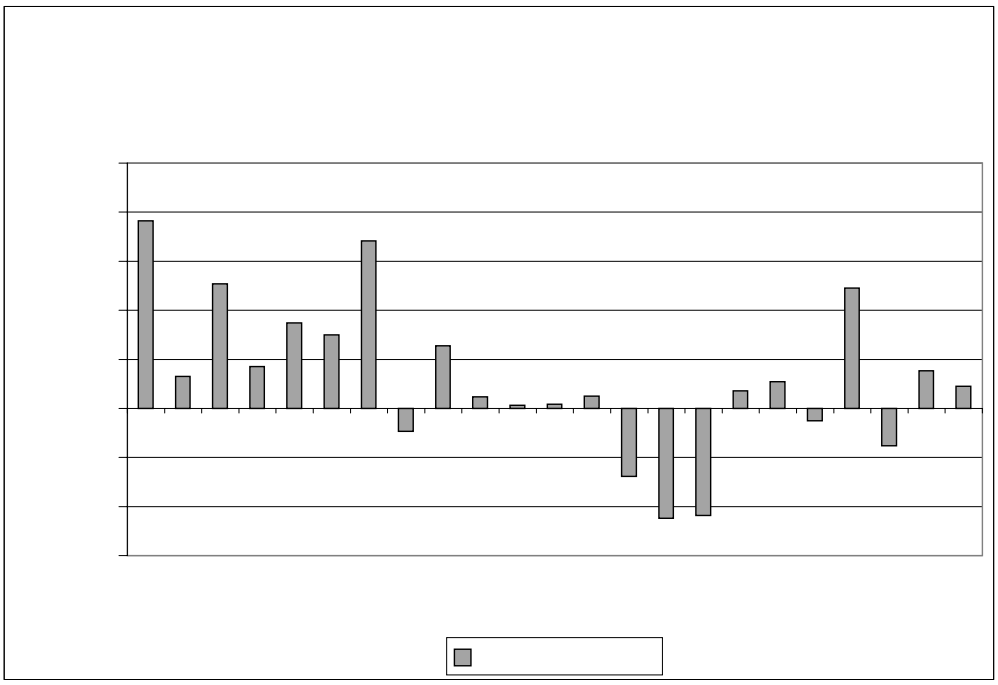


Figure 5.12: Area Overhead for Partition Level 1

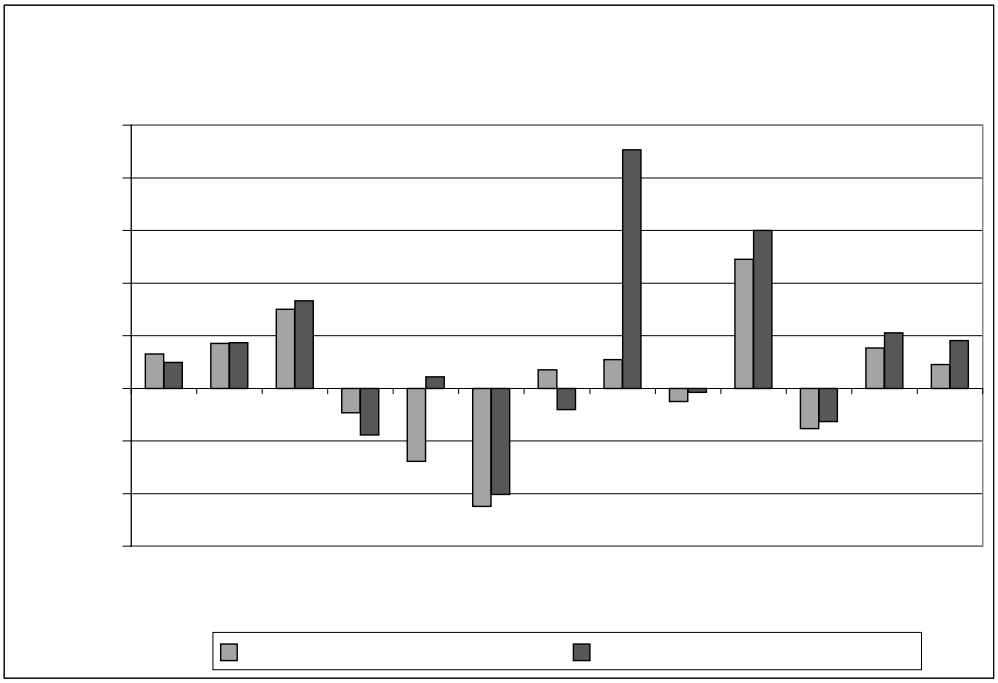


Figure 5.13: Area Overhead for Partition Levels 1 and 2

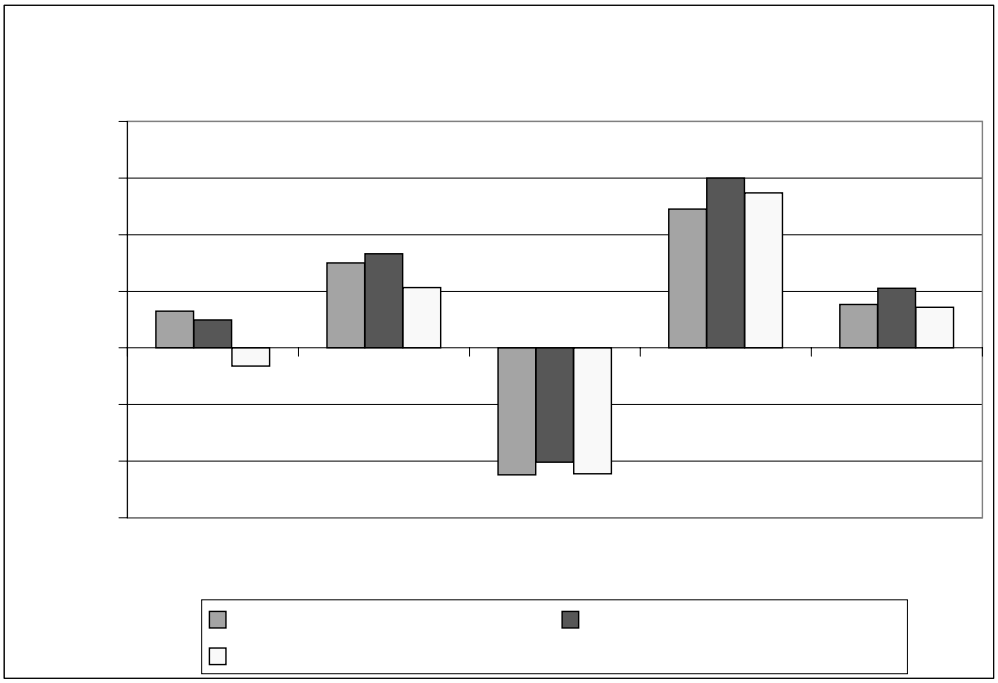


Figure 5.14: Area Overhead for Partition Levels 1, 2 and 3

Interestingly enough, partitions for benchmarks *LL8_int, LL14_int, LL15_int, LL16_int, LL20_int,* and *LL22_int*, result in area decrease over the unpartitioned design. One possible explanation for this phenomenon is that since the synthesis tool is dealing with smaller designs, it can to a better job in terms of resource sharing due to the smaller search space it has to deal with.

5.3 Memory Allocator Implementation

The suitability of the memory allocator core as a microarchitectural component is evaluated by studying its synthesis results. The goal is to determine the scalability of the core as a function of the memory heap size it works with. The implementation details of the core are provided below:

1. A parameterized memory allocator was described in VHDL. It was coded in a hierarchical fashion in order to facilitate modularity and bottom-up compilation.

The size [2] of the memory allocator was defined as a parameter in the top-level design using the VHDL *generic* statement. The top-level design passed this value to the lower designs, which passed it further to the lower designs, and so on. Based on the size, VHDL *generate* statements were used to replicate and connect basic circuits in order to generate the complete memory allocator.

2. The VHDL code was compiled using the *Design Compiler* tool from Synopsys. The compiled code was simulated to verify correct functionality.

3. The VHDL code was compiled using various compile strategies and with different allocator size implementations. Reports were generated indicating area, speed, and compilation times.

4. The allocator was also prototyped and tested an an Altera FLEX 10K70 Field Programmable Gate Array (FPGA).

5.3.1 Design Synthesis

Allocators of different bit-vector sizes were compiled using a top-down and a bottom-up approach. For each compile method, the total design area, critical path delay, and compile times were recorded.

In the top-down approach the entire design was read-in and multiple instances of any design reference were resolved by flattening the top-level design (removing all hierarchy). For each bit-vector size, the design was compiled using two separate optimization constraints: 1) Area only - The design was optimized for smallest size, and 2) Speed and Area - The design was optimized for smallest size *and* maximum speed.

For the bottom-up compile, the sub designs were first compiled independently. Then the top-level design and any compiled sub-designs not already in memory were read in.

[2] This is the bit-vector size of the allocator which corresponds to the heap size the memory allocator works with.

Constraints were set to the top-level design and it was linked. In interests of time, the bottom-up approach was compiled only with the *speed and area* constraint.

The bottom-up compilation partitioned the code in two different ways: In the first method (*low-level partition compile*), only the lowest level entities of the VHDL code were optimized. The higher level designs simply linked the optimized lower-level code as is. This resulted in the final synthesized design, which maintained an identical structure to the original VHDL description. This had the effect of reducing the compile times considerably for larger designs at the expense of higher critical delays and area. In the second method, (*high-level partition* compile), carefully chosen intermediate levels of hierarchy were flattened (merged) and synthesized. The merging process increased the scope for synthesis optimizations and produced better results.

The *wcells* CMOSP35 0.35 micron library provided by the Canadian Micro-Electronic Corporation (CMC) was used to compile the design. All compilations were performed on a Sun Ultra 5/10 workstation with an UltraSPARC-IIi 360 MHz CPU and 320 Mbytes of RAM.

5.3.2 Synthesis Results

Area Results

The area results for the top-down and bottom-up compilation strategies are shown in figure 5.15.

All four curves are linear (the data-sets are in fact linear even though they are plotted on a log-scale) which indicates that the area of the memory allocator scales directly with the bit-vector (memory) size.

As expected, the top-down (area optimization) compilation provides the lowest area usage, while the bottom-up (low level partition) compilation provides the highest area usage. The remaining curves are in-between. The linear relationship allows a rough

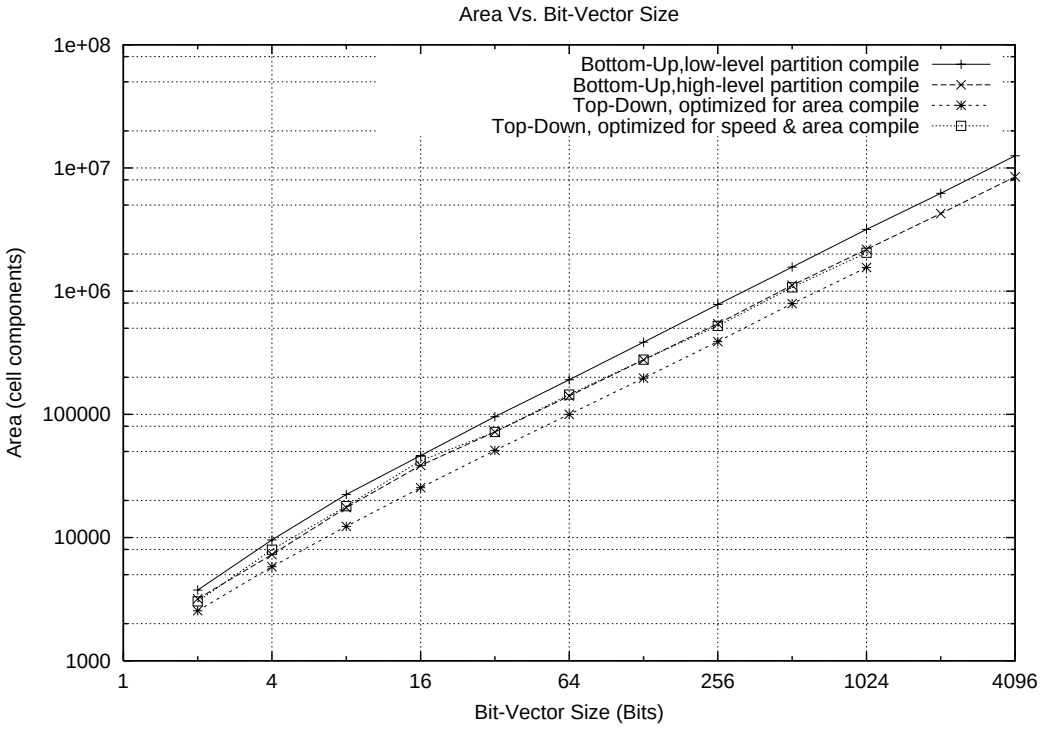


Figure 5.15: Area Vs. Bit-Vector Size

estimate of approximately 80 transistors/bit for the bottom-up (high-level partition) compile. This allows us to estimate the transistor count for larger designs. As an example, a 64 Kbit memory allocator would require:

$$65536\ bits \cdot 80\ transistors/bit = 5,242,880\ transistors.$$

As reference, this is comparable to the number of transistors in the original Intel Pentium processor. Larger memory allocators would not be feasible as an microarchitectural component since they would begin to dominate the chip area.

Critical Path Delay Results

The critical path delay results for the top-down and bottom up compilation strategies are shown in figure 5.16. The critical path delay represents the time it takes to complete a memory allocation or free operation. All path delays, except for the bottom-up

(low level partition) exhibit a logarithmic trend. As expected, the top-down (speed and area) compile has the lowest propagation delay, while the bottom-up (low level partition) compile has the highest. The propagation delay of the bottom-up (high level partition) compile is in between the two extremes.

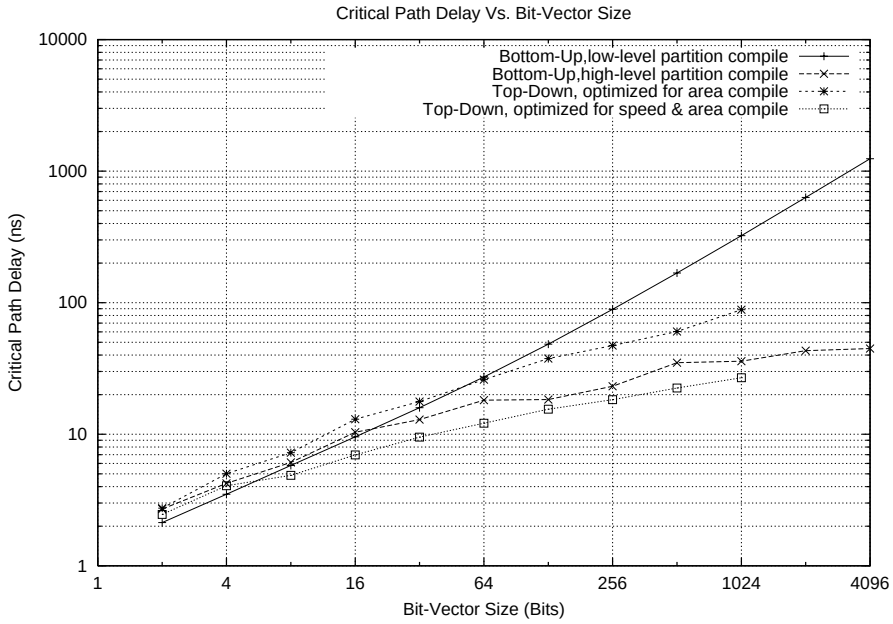


Figure 5.16: Critical Path Delay Vs. Bit-Vector Size

Compile Time Results

The compile time results for the top-down and bottom-up compilation strategy are shown in figure 5.17

As expected, the bottom-up compilation strategies have shorter compile times for larger designs than the top-down strategies.

As per the graph, the compile time for the 4096 bit-vector for the bottom-up (high-level partition) was 1607 minutes, or 26.8 hours [3] In contrast, the *low-level* partition method took only 214 minutes to compile, or 3.6 hours. Clearly, this indicates that a substantial allocator can only be compiled by paying careful attention to partitioning.

[3]Identical compilation on a SunBlade 1000 Model 2750 server with 2.5 GB RAM, and 2 Sun UltraSpARC-III 750Mhz CPUs took only 6.9 hours to complete.

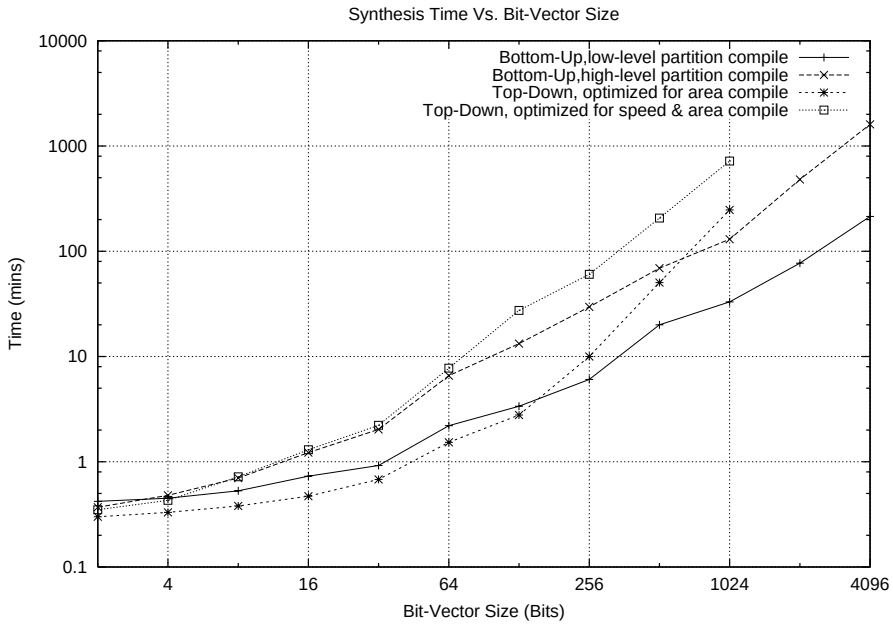


Figure 5.17: Compile Time Vs. Bit-Vector Size

FPGA Implementation

The allocator was prototyped and tested on an FPGA. The Ultragizmo board from the University of Toronto was used as the implementation environment. The Ultragizmo board is a printed circuit board (PCB) developed and used by the University of Toronto as a teaching tool for its course curriculum.

The Ultragizmo board contains a Motorola MC68306 integrated processor, an Altera 10K70 FPGA and supporting hardware. The 10K70 FPGA contains the equivalent of 70,000 standard gates and 18432 bits of user programmable RAM.

A 256 bit-vector memory allocator was compiled for the FPGA using Altera's MAX-PLUS2 FPGA synthesis and layout tool. The device utilization of the FPGA was 44%, with a critical path delay was around 300 ns.

A wrapper circuitry was added to the memory allocator (all within the FPGA) so that it could be accessed by the MC68306 processor. From the processor's point of view, the allocator was a memory-mapped device which was accessed by reading/writing to registers at specific addresses. *malloc* and *free* routines were written in 68000 assembly so

that applications running on the MC68306 processor could utilize the memory allocator. A test-bench file was generated which specified a number of memory allocation and free operations with the expected results. This test-bench was run using the hardware memory allocator and the results were compared to verify proper operation .

Chapter 6

Conclusion and Future Work

6.1 Conclusions

This thesis offers two main contributions. First, it introduced microarchitectural changes that enables high-level synthesis to better cope with designs described in high-level languages. These changes included extending the FSMD model into a stacked model (SFSMD) to support procedure abstraction, and incorporating a dynamic memory allocation unit to support memory abstraction. In the SFSMD model, each procedure was implemented as a separate controller with a common datapath shared amongst them. A stack-controller was used to handle the controller calls and it allowed the datapath to be shared. A buddy-system based dynamic memory allocator core was also described and studied. The second contribution of the thesis was the introduction of a behavioral partitioning technique for power reduction. It operated on the basis of implementing loop kernels as separate controller in the SFSMD model. Power savings were achieved because the controller of each loop was smaller than the one large controller implementing the entire system, and only one controller was running at any given time. Also, a partitioning index was defined that could be used to estimate and compare the power of different partitioning styles prior to synthesis.

Based on this study, we draw four main conclusions. First, the SFSMD model provides a good basis for procedure abstraction. Secondly, the memory allocator core is shown to be a viable solution for the implementation of dynamic memory allocation in high-level synthesis. Since the core grows linearly with the memory heap size, a heap size of 64K, which is sufficient for most on-chip designs, is considered as the upper limit for the core. Allocators for larger heaps would be infeasible as microarchitectural components due to high-area usage. The third conclusion is that region based partitioning is an effective technique for the reduction of controller power consumption. The final conclusion is that due to the strong correlation between the power-index values and the actual partition power, the power-index equation can be used to effectively guide the partitioning decisions of a high-level partitioning tool.

6.2 Future Work

There is a lot of scope for future work in this area. An important step would be the integration of the SFSMD model into a synthesis engine. This would automate the transformation of procedural descriptions into an SFSMD model, thereby enabling more comprehensive studies of this model to be performed. Estimators could be developed to directly compute the design area, speed, and power of this implementation style, providing the designers with the capability to decide how the design should be implemented. Similarly, a tool can be developed to perform loop region based partitioning. Such a tool would partition the code into a power optimal configuration before forwarding it to the synthesis engine for SFSMD implementation.

Bibliography

[1] M. Alidina, J. Monteiro, S. Devadas, and A. Ghosh. Precomputation-Based Sequential Logic Optimization for Low Power. *Proceedings of the International Conference on Computer Design*, pages 74–81, October 1994.

[2] L. Benini and G. De Micheli. Synthesis of Low-Power Selectively-Clocked Systems from High-Level Specification. *ACM Transactions on Design Automation of Electronic Systems*, 5, No. 3:311–321, July 2000.

[3] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic With VHDL Design*. McGraw Hill, 2000.

[4] R. Camposano, L. Saunders, and R. Tabet. VHDL as input for high level synthesis. *IEEE Design and Test of computers*, pages 43–49, March 1991.

[5] R. Camposano and J. van Eijndhoven. Partitioning a Design in Structural Synthesis. *Proceedings of the International Conference on Computer Design*, 1987.

[6] Editor D. Gajski. *Silicon Compilers*. Addison-Wesley, 1987.

[7] S. Devadas and Sharad Malik. A Survey of Optimization Techniques Targeting Low power VLSI Circuits. *Proceedings of the Design Automation Conference*, pages 242–247, 1995.

[8] D.Grunwald, B. Zorn, and R. Henderson. Improving the Cache Locality of Memory Allocation. *SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993.

[9] E. Frey. ESIM: A Functional-Level Simulation Tool. *Proceedings of the International Conference on Computer Aided Design*, pages 48–50, 1984.

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

[11] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[12] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, 2000.

[13] Y Explorations Inc. *eXCite User Guide, Rev. 3*. 2003.

[14] K. Jasrotia and J. Zhu. Hardware Implementation of a Memory Allocator. *EUROMICRO Symposium on Digital System Design*, pages 355–358, September 2002.

[15] J.Change and E. Gehringer. A high-performance memory allocator for object-oriented systems. *IEEE Trans. Computers*, pages 357–366, 1996.

[16] G. Lakshminarayan, A. Raghunathan, K.S. Khouri, N.K. Jha, and S. Dey. Common-Case Computation: A High-Level Technique for Power and Performance Optimization. *Proceedings of the Design Automation Conference*, June 1999.

[17] E. Macii, M. Pedram, and F. Somenzi. High-Level Power Modeling, Estimation, and Optimzation. *Proceedings of the Design Automation Conference*, pages 31–38, 1997.

[18] G. De Micheli. *Synthesis And Optimization of Digital Circuits*. McGraw Hill, 1994.

[19] G. De Micheli and D. Ku. *High-Level Synthesis of ASICs under Timing and Synchronization Constraints*. Kluwer Academic Publishers, 1992.

[20] E. Puttkamer. A simple hardware buddy system memory allocator. *IEEE Trans. Computers*, pages 953–957, October 1975.

[21] L. Ramachandran, S. Narayan, F. Vahid, and D. Gajski. Synthesis of Functions and Procedures in Behavioral VHDL. *Proceedings of the European Design Automation Conference*, 1993.

[22] D. Smith. *HDL Chip Design*. Doone Publications, 1996.

[23] Synopsys. *Power Compiler Reference Manual, Version 2000.05*. May 2000.

[24] V. Tiwari, S. Malik, and P. Ashar. Guarded Evaluation: Pushing Power Management to Logic Synthesis Design. *International Symposium on Low Power Design*, 1995.

[25] F. Vahid. Procedure Exlining: A New System-Level Specification Transformation. *European Design Automation Conference*, pages 508–513, September 1995.

[26] F. Vahid. Procedure Exlining: A Transformation for Improved System and Behavioral Synthesis. *International Symposium on System Synthesis*, pages 84–89, September 1995.

[27] F. Vahid. I/O and Performance Tradeoffs with the FuctionBus during Multi-FPGA Partitioning. *International Symposium on FPGAs*, pages 27–34, February 1997.

[28] F. Vahid. A Three-Step Approach to the functional Paritioning of Large Behavioral Processes. *International Symposium on System Synthesis*, pages 152–157, December 1998.

[29] F. Vahid. Procedure Cloning: A Transformation for Improved System-Level Functional Partitioning. *ACM Transactions on Design Automation of Electronic Systems*, 4, No.1:70–96, January 1999.

[30] F. Vahid, E. Hwang, and Y. Hsu. FSMD Functional Partitioning for Low Power. *Design Automation and Test In Europe*, pages 22–28, March 1999.

[31] R. Waxman. Hardware Design Languages for Computer Design and Test. *IEEE Design and Test for Computers*, 19 no 4, April 1986.

[32] LiverMore Benchmark WebPage. *http://parallel.ru/ftp/benchmarks/livermore/livermorec.c.*

[33] SystemC WebPage. *http://www.systemc.org/.*

[34] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Proc Int'l Workshop on Memory Management*, pages 953–957, September 1995.

[35] B. Zorn. The measured cost of conservative garbage collection. *Software-Practice and Experience*, pages 733–756, July 1993.