Extending the Thread Programming Model Across CPU and FPGA Hybrid Architectures

by

Razali Jidin

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Dr David Andrews, Chairperson

Dr Douglas Niehaus

Dr Perry Alexander

Dr Jerry James

Dr Carl E Locke Jr.

Date Submitted:

ABSTRACT

Field-programmable gate arrays (FPGA's) have come a long way from the days when they served primarily as glue logic and prototyping devices. Today's FPGA's have matured to the level where they can host a significant number of programmable gates and CPU cores to create complete System on Chip (SoC) hybrid CPU+FPGA devices. These hybrid chips promise the potential of providing a unified platform for seamless implementation of hardware and software co-designed components. Realizing the potential of these new hybrid chips requires a new high-level programming model, with capabilities that support a far more integrated view of the CPU and the FPGA components than is achievable with current methods. Adopting a generalized programming model can lead to programming productivity improvement, while at the same time providing the benefit of customized hardware from within a familiar software programming.

Achieving abstract programming capabilities across the FPGA/CPU boundary requires adaptation of a high-level programming model that abstracts the FPGA and CPU components, bus structure, memory, and low-level peripheral protocol into a transparent computational platform [2]. This thesis presents research on extending the multithreaded programming model across the CPU/FPGA boundary. Our objective was to create an environment to support concurrent executing hybrid threads distributed flexibly across CPU and FPGA assets.

To support this generalized model across the FPGA, we have developed a Hardware Thread Interface (HWTI) that encapsulates mechanisms to support synchronization for FPGA based threads. The HWTI enables custom threads within the FPGA to be created, accessed, and synchronized with all other system threads through library API's. Additionally, the HWTI is capable of managing "thread state", accessing data across the system bus, and executing independently without the need to use CPU.

Current multithreaded programming models use synchronization mechanisms such as semaphores to enforce mutual exclusion on shared resources. Semaphores depend on atomic operations provided through the CPU assembler instruction set. In multiprocessor systems, atomic operations are achieved by combinations of processor condition instructions integrated within memory coherency protocol of snooping data caches. Since these current mechanisms do not extend well to FPGA based threads, we have developed new semaphore mechanisms that are processor family independent. We achieve a much simpler solution and faster mechanisms (8 clock cycles or less) for achieving semaphore semantics with new atomic operations implemented within the FPGA. These new FPGA based semaphores provide synchronization for hardware, software and combinations of hardware/software threads. We also migrate sleep queues and wake-up capabilities that are normally associated with each semaphore into the FPGA. The wake-up mechanism has the ability to deliver unblocked threads either to the CPU or FPGA. The queue and wake-up operation do not incur any system software overhead.

As the total number of semaphore required in a system may be large, implementing separate queues for each semaphore can require significant FPGA resources. We address the resource utilization issue by creating a single controller and a global queue for all the semaphores without sacrificing performance. We solve the performance issue with hardware and queuing algorithm solutions. The semaphores are provided in the form of intellectual property (IP) cores. We have implemented recursive mutexes, recursive spin lock and condition variable cores in addition to the semaphore core. These cores provide synchronization services similar to the POSIX thread library.

Toward the end of this thesis, we present an application study of our hybrid multithreaded model. We have implemented several image-processing functions in both hardware and software, but from within the common multithreaded programming model on a XILINX V2P7 FPGA. This example demonstrates hardware and software threads executing concurrently using standard multithreaded synchronization primitives transforming real-time images captured by a camera and displayed on a workstation.

In loving memory of those departed during my tenure with this degree:

Mother (Year 2004)

Grand Father (Year 2002)

Auntie (Year 2004)

Step Father-in-law (Year 2005)

Acknowledgements

I would like to express my sincere appreciation and gratitude to everyone who made this thesis possible. First and foremost, I would to thank my adviser Dr. David Andrews for his guidance, encouragement and patience throughout the tenure of this research. From him, I hope that I have learned enough to conduct research, to write and publish papers.

I am grateful to the members of my dissertation committee, Dr. Douglas Niehaus, Dr. Jerry James, Dr. Perry Alexander and Dr. Carl Locke for their precious time and advice. A special thank to Dr. Carl Locke for introducing me to KU and his for encouragement. I appreciate Dr. Niehaus's guidance on semaphore and thread especially during early stages of this research.

Testing work conducted would not have been possible without the assistance of Wesley Peck. I appreciate all the help and discussion from Wesley Peck, Jason Agron, Ed Komp, Mitchell Trope, Mike Finley, Sweetha Rao and Jorge Ortiz.

To my wife Norlida, thank you for her ceaseless support, love and patience throughout the entire period of this study. This thesis is dedicated to my four children Nurul Farzana, Nurul Adeela, Naeem and Nurul Irdeena, for without them I would not fulfill my long ambition of seeking this degree. Their curiosity and aspiration helped me push through this very challenging period of my life. They are adapting well to living in Kansas, enjoy learning another culture, and love attending Hillcrest Elementary School.

My gratitude goes to both my departed parents Sa'amah and M. Jidin who always emphasized that knowledge is important and for giving me the opportunity to learn.

Last, but not the least, my thanks to God, without His Mercy and Grace I would not be here today.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

Field Programmable Gate Arrays (FPGA's) have matured significantly from their origins as simple programmable logic devices (PLDs) used as substitutes for SSI combinational logic chips. Over the last three decades, FPGA's have grown from simple glue logic components, through moderate prototyping platforms and more recently, as complete systems on chip (SoC) components. Today's modern FPGAs now commonly share portions of their silicon die area with a variety of diffused IP, such as multipliers, bulk RAM, and processor cores. The rapid increase in fabrication technology has spurred increases in system developers desires to build more complex systems with these fully capable commodity parts. Unfortunately, the increase in fabrication capabilities has not been matched with a corresponding increase in software tools and methods for exploiting the full potential of these components. The common methods in use for circuit design are based on old hardware description languages (HDL's) that were developed two decades ago for describing low level SSI, MSI, and VLSI components. These languages were adopted for circuit design and replaced schematic capture for application specific integrated circuits (ASICs) in the 1980's. Although they are still in use today, they present drawbacks for specifying the complex subsystems and circuits associated with modern FPGA's. First, they are not a particularly efficient languages for driving modern automatic synthesis tools. This is a result of their lack of abstraction in their data types and ambiguity in certain behavior constructs that do no translate well to logic gates, wires, and finite state machines. Even though they allow structural specification, they do not contain constructs that reflect the organization of modern FPGA building blocks and interconnect networks. New low level languages such as JHDL [51] have been proposed to address this problem. However, both artifact HDL's and the newer JHDL still present problems for dealing with the complexity of modern FPGAs. This approach requires specifying individual circuits in terms of bits, combinational logic circuits, flip-flops, latches, and I/O drivers. Designing at this low level of abstraction is simply becoming impractical for efficiently exploiting the complexity of modern FPGA's. An additional concern is that this approach requires hardware design skills that are not possessed by the majority of system programmers and software engineers.

New research is now addressing this problem by investigating new approaches to circuit generation from traditional high-level software languages. Specifying circuit behavior from a high level language helps in dealing with the complexity of modern FPGA's as well as enabling programmers to access the potential of the reconfigurable fabric [19, 21, 48]. Although an important step, enabling circuit specification from a higher level syntax only addresses part of the problem. The missing component is advancements in not just programming languages, but also programming models. A programming model specifies computational components such as tasks, threads, and processes, as well as their interactions such as semaphores and inter-process communications (IPC). Programming models also provide the definition of standard component interfaces and abstract data types. Thus, a programming model provides the developer with an abstraction that alleviates knowledge of low level platform details and allows the expression of the application in a more appropriate form.

1.1 OBJECTIVE

The goal of this research was to develop such an abstract capability by bringing both hardware and software computations under the familiar multithreaded programming model. This approach provides the following advantages. First, extending a software programming capability over the hybrid device enables programmers to gain access to the potential of the reconfigurable logic. This is significant, as our current methods of programming the FPGA require hardware design skills not familiar to software engineers and programmers. Second, the use of higher level abstractions and languages decreases development time and costs. Bringing in higher levels of abstractions is also important as the complexity of these components are already much greater than can be efficiently handled with low level methods. Third, the system services that have been developed to support the programming model provide increased capabilities for time critical applications that could not be achieved through classical software approaches. This enables new levels of precise control over critical real time applications.

We chose the multithreaded framework as our model as it represents the type of concurrency typically found in a range of embedded applications, and is familiar to a wide range of system programmers. This is evidence from wide acceptance of POSIX multi-threaded programming. Additionally this model permits other computational models to be composed on top. As a simple example the data streams model is easily implemented as a set of threads that linearly synchronize.

The successful realization of a concurrent hybrid system requires uniform concurrency mechanisms for both CPU based software threads as well as FPGA based hardware threads. The different concurrency control primitives defined by POSIX include mutexes, semaphores and condition variables. Each of these synchronization primitives serves different purposes such as mutual exclusion, event waiting and controlling countable resources. Blocking primitives require sleep queues and wake-up mechanisms. Another category of concurrency control is referred to as spin primitive. Spin is useful to serve blocking primitive such as condition variables and for multi-processor environment. In our case spin is helpful to synchronize concurrent execution of FPGA hardware threads and CPU based threads. Implementing all these mechanisms in FPGA either partly or otherwise will depend on feasibility, performance enhancement, resources versus performance trade-off, and other aspects of hardware software co-design. For example implementing an efficient sleep queue may be costly in terms of FPGA resources if the size of the supporting circuits must scale with the number of blocking primitives.

Developing a uniform computational model requires reconciling the different underlying computational models of the CPU and FPGA. Whereas a CPU has a program counter, stack, and register set, the FPGA has no cycle-by-cycle instruction stream, stack, and temporary register set. Additionally, with current FPGA technology, system developers must synthesize and map the data paths and operations that represent the computations of the thread into FPGA before runtime. These differences require new mechanisms to represent FPGA based hardware computation to the thread abstraction, and to support interactions among threads across the CPU/FPGA boundary. Although it seems that the lack of an existing computation model is a detriment, on the contrary it is an asset as it presents an opportunity to create more efficient mechanisms. The implementation of threads in an FPGA must maximize the advantages of using the FPGA platform, while preserving the common multithreaded programming model. At the same time, the FPGA thread implementation must not use methods that degrade the efficiency of synchronization across the CPU/FPGA boundary. Neither should the FPGA methods create any form of unfairness between the FPGA and CPU threads.

On processors, an operating system provides services including general resource management to the software threads. However, on the FPGA, the hardware threads do not have direct access to these system services, and it is not efficient for the hardware threads to cross the boundary to seek services from the operating system running on the CPU. There are several approaches to this problem. First, new services hardware services can be created specifically to serve the hardware threads. Second, base services can be created in the hardware that make use of software threads as proxies to seek services from the operating system. Still yet, existing software services can be migrated into the hardware to serve both hardware and software threads. Although more complex, this last approach can enable new levels of performance for software threads as well as providing non-intrusive services to the hardware threads.

1.2 APPROACH

The objective of this research was to create a programming and execution environment that strongly integrated the hybrid CPU/FPGA based computational components under familiar multi-threaded programming paradigm. Our successful outcomes include an adaptable hardware and software co-design and execution environment within which a given application can be realized as a set computation threads with many possible mappings on both the CPU and FPGA processing assets. Our approach was to bring both the FPGA based thread and CPU based components of an application under the umbrella of the POSIX thread model. This first required enabling the FPGA to support atomic operations equivalent to classic processor load-linked store conditional instructions. Providing atomic operations is fundamental for supporting synchronization operations, which in turn are used for thread synchronization. Following the creation of new synchronization primitives, the first of two co-design efforts focused on developing appropriate low-level services across and between the CPU/FGPA assets within the operating system. The key issue in this co-design effort was to address how to decompose system services and migrate them into the FPGA. The second co-design effort was to create the application level API's callable by both hardware and software threads. This required the creation of small wrappers for software threads, and a new abstraction interface structure for hardware threads. The abstraction interface encapsulated the lower level platform specific details within procedures that had a similar API interface with their software API counterparts.

1.2.1 Enabling Atomic Operations

Management of shared resources is fundamental to the successful implementation of a concurrent programming model. Accesses to common resources by the concurrent executing threads in a shared-memory system are serialized by synchronization mechanisms such as semaphores, or simple binary locks. On general-purpose processors, synchronization implementations are based on the atomic operation such as test-and-set or swap instructions. For example, the PowerPC755 has *lwarx* and *stwcx* instructions with RSRV signal. In multiprocessor systems, atomic operations are achieved by combinations of processor condition instructions integrated within a memory coherency protocol of snooping data caches. Since these current mechanisms do not extend well to FPGA based threads, new methods have to be developed. In addition several issues must be considered:

- The location of each synchronization variable and its associated services either on the FPGA or processor and system memory.
- The number of synchronization variables in a given system can be changed without the need to redesign the hardware.
- Portability: current System-on-Chip (SOC) has the capability to host multiple processor architectures including digital signal processing (DSP) and the need to support possible heterogeneous applications.

An efficient atomic operation based on normal write/read pair can be easily implemented within FPGA to support synchronization primitives. These synchronizations will be CPU family independent since their accesses utilize standard input/output operations. Implementing the atomic operation within FPGA offers an additional advantage, as multiple bus cycle and wait state associated with typical memory (DRAM) will be avoided.

To minimize time to market, system designers are now implementing the intellectual property (IP) base design in scheming their hardware system. Adopting this approach permits modularity, allowing FPGA based synchronization or other types of IP to be added to the system without the need to redesign the hardware. However, the need to scale the IP core to the number of concurrency primitives is also an issue to be addressed, as their number can change at run time and varies from one application to another.

As multiple threads running on processors and FPGA can access the synchronization variables, the appropriate solution requires the synchronization IP cores to be attached to the system bus. An internal bus may be added, to allow FPGA threads to obtain the synchronization variables without crossing the system bus, to reduce the system bus traffic. However an internal bus can be unfair to the CPU based threads. Independent of the internal bus existence, both the FPGA threads and the synchronization IP must have bus interfaces. In addition FPGA threads must have the ability to initiate and arbitrate the system bus to access the synchronization IP. It must have facilities to arbitrate the system bus, generating address and relevant bus handshaking and control signals.

These basic atomic structures within the FPGA will provide basic building blocks toward implementing the higher order concurrency control primitives such as semaphore, mutual exclusion and condition variables.

1.2.2 Co-design and Hardware Implemented O/S Services

High-level integration of the CPU and FPGA provides opportunities to discover new ways of implementing operating systems especially for embedded systems. Traditionally, an operating system is a collection of system software that provides hardware abstraction to the application layer, while the multiprocessing programming is the enabling technology permitting hardware sharing. Current FPGA devices are not only allowing us to migrate the concurrent control mechanisms from the CPU into the hardware, but other system software components as well, especially toward achieving processor workload reduction and system response variability improvement.

In current parallel programming implementation, the synchronization mechanisms provided can be categorized into two types – busy wait or blocking type. The blocking type mechanisms, such as semaphores, require sleep queues to place blocked threads when there are access contentions. These sleep queues are traditionally implemented in the system memory and each semaphore has its associated sleep queue. Thus, it is natural to migrate the sleep queue into the FPGA, as well. In addition, blocking semaphore must have a facility to wake-up the blocked threads. As the awakened threads can be FPGA threads or CPU threads, the blocking semaphore should have the capability to deliver them either to the CPU or FPGA. As the number of blocking semaphores in a given system can be large, the issue of mapping the sleep queue to hardware resources needs to be addressed. The queue and its wake-up mechanism should expend FPGA resources optimally and without sacrificing performance. In addition the state of the sleep queues have to be protected between the start of unblocking process until the delivery of all the unblocked threads, as there are possibilities of new requests.

Delivery of awakened CPU threads to the scheduler queue requires the generation of exception to the processor. The scheduler then may need to run the scheduling decision depending on the scheduling algorithm being used. Obviously not all insertion of unblocked threads cause swapping of threads to run on the processor. Independent of scheduling algorithm, delivery of unblocked threads to the scheduler queue can cause unnecessary exception processing overheads. Obviously migrating the scheduler and key time services into the FPGA can eliminate context switching associated with the unblocking operation. However, the scheduler queue relocation to the FPGA affects the handling mechanism of software threads. Thus, it is a natural extension to migrate the software thread management into the FPGA, as well. Further, whether the system functionality is implemented in hardware or software, it should be abstracted from the user application. User applications do not need to know the location of services provided.

The above approach defines new partition for operating system services across hardware/software boundaries. On the hardware side, FPGA threads cannot request system services using the same mechanism as CPU based threads, such as traps to the operating system. As conventional operating services cannot support the hardware threads, new services have to be invented to enable hardware threads to have abilities to emulate the software threads. These new services will abstract the low-level hardware architecture details from the user application. The degree of abstraction required depends on the programming language chosen by the user and the requirement of application. The operating system is not required to conceal all the hardware instead it should abstract only the non-customized components. To manage these new services and cope unclear boundary between hardware and software, we propose to classify the operating system components for the hybrid CPU/FPGA devices into four categories:

1. Software implemented conventional operating system (soft O/S services)
2. Hardware implemented conventional operating system (hard O/S services)
3. Hardware implemented to service both hardware and software threads (hybrid O/S services)
4. Hardware operating system (hardware services or hardware functions)

The second category, which we refer to as hard O/S, is to serve CPU based software threads while the fourth category is for FGPA based hardware computation. An example of hard services includes the software thread scheduler mentioned above. The hardware functions include the facilities to enables FPGA thread to access memory and concurrency control mechanisms. Access to system memory enables FPGA threads share data with the CPU based software threads. These hardware functions will not cost much hardware resources and, for performance consideration, there are best implemented within individual FPGA threads. These basic hardware functions are sufficient to enable FPGA threads to perform tasks within the multithreading environment. The need to transfer the data to the peripherals can be done with the help of the first category operating service component that we refer to as soft O/S. The FPGA threads however must first transfer the data into the heap section of main memory. Alternatively, the support for peripheral interfaces can be implemented within the FPGA. For example, hardware TCP/IP stack can be implemented into the FPGA. These new hardware services can be categorized as hybrid services as it could serve both software and the hardware threads. This attractive solution not only enables FPGA threads communicate directly with peripherals, but will also reduce memory utilization and free CPU from similar processing task. Concurrent control mechanisms mentioned previously can be categorized as hybrid services mentioned above, and they will be useful to control sharing of the new hardware implemented services. The scope of research in this thesis however is not to implement all the mentioned services but rather to identify new hardware and software partition and to create necessary enabling mechanisms that extend multithreaded programming model into the FPGA.

In implementing this programming model, we have two options: a simple micro-kernel or desktop type operating system. As the microkernel is designed to tame the complexity of the desktop operating system, it offers small footprint and good timeliness behavior for the embedded system. The micro-kernel is also an attractive choice when considering less effort is required to adapt it, and to add certain system services into the hardware. However, it lacks much system services and development tools. Sufficient system service is essential, especially displaying outputs graphically during the demonstration phase.

1.2.3 Application Level CPU and FPGA Co-design

The acceptance of the multithreading programming model has led to the definition of a platform independent operating standard (POSIX) that supports application level multi-threaded programming. We will use the same approach as the POSIX multi-threaded library to provide application program interface (API) to create our own library. We will approach the API iteratively, first implementing basic FPGA-thread creation and then proceed to hybrid concurrency control. Essentially, for the synchronization variables, at least two API are needed. One is to acquire and the other is to release the variables. Acquiring APIs may cause blocking operations. The blocking operation on the CPU requires switch of context and change of states. The release may cause unblocking operation. On the CPU, the unblocking operation will change the thread state and put it back into the scheduler queue. We will further concentrate on the concurrency control mechanism, add more APIs as we adding more features.

FPGA Thread Control

Current FPGA technology requires that logic primitives to be programmed prior to execution. The algorithm that has been specified for a thread must be synthesized and loaded into the FPGA prior to the run-time. In effect, the configured logic primitive and interconnections that form data path and control unit for the hardware computational component must be pre-loaded without allowing the FPGA thread to execute. When the thread is loaded into the FPGA, the thread will default to the idle state. It remains in this state unless it receives a start command.

To start or suspend an FPGA thread requires creation of a “controller” that will manage state transition similar to the software thread. Useful states that can represent the operation of a thread includes idle, run, and wait. Similar to the software threads, a thread goes to sleep when it is blocked from getting a semaphore. The thread transitions back to the run state when it is unblocked. Therefore, a given state controller must have an interface in the form of memory-mapped command register to accept control word from the CPU or semaphores. To start a thread, an application program interface on the CPU writes a start command to the register. To unblock a thread, semaphores write wake-up control word to the register. Additionally, the argument registers can be added within the controller to provide options for a given thread to choose different execution paths.

In addition to the duty of managing state transition, the controller provides other services as well. The essential services include synchronizations and input/output operations that effectively enable data exchange across the system bus with other threads. The controller performs these services in response to requests from the user on the hardware side. The controller interprets a user request, performs the intended service, for example, obtaining a semaphore. To abstract the controller low-level details, the hardware implemented application program interfaces (APIs) are provided. The user makes use of these APIs when requesting services from the controller.

1.2.4 Experimental Platform and Evaluation

We have chosen as our first experimental platform a commercially available development board that contains a hybrid CPU/FPGA chip; the Xilinx Virtex-II Pro. The Virtex-II Pro chip contains a Power PC 405 core embedded within an FPGA fabric that contains over 11,000 CLB’s. This level of CLB integration provides a sufficient number of devices for our experimental hardware co-design requirements. We also have access to a significant library of VHDL descriptions of IP cores which can be used a base and modified for our work. This provides us with a true SoC target from which to proceed.

For evaluation of our hybrid threads, we have chosen to implement several image processing algorithms. This evaluation will demonstrate hardware and software threads executing concurrently, sharing data by means of new hybrid semaphores, transforming real-time images captured by a camera and displayed on a workstation.

1.3 CONTRIBUTION OF THIS THESIS

The research presented in this thesis is part of KU Hybrid Thread Project. This thesis makes the following contributions in the development of a framework for extending multithread programming model across CPU/FPGA architectures:

1) Definition of FPGA based hardware thread context and control.
2) Design, implementation and testing of FPGA based threads.
3) Design, implementation, and testing of hardware implemented APIs for synchronization services and memory accesses.
4) Design, implementation, and testing of FPGA based recursive spin lock core.
5) Design, implementation, and testing of FPGA based recursive mutex core.
6) Design, implementation, and testing of FPGA based counting semaphore core.
7) Design, implementation, and testing of FPGA based condition variables core.
8) Initial design and contribution to hardware based software thread management.
9) Implementation of synchronization primitive APIs for testing.
10) Integration and synchronization testing of CPU-based threads and FPGA-based threads.
11) Introduction to new partition of hardware and software services.
12) Evaluation of the hybrid multi-threads model using image transformations.

1.4 OUTLINE

The remainder of this thesis is organized as follows. The next chapter presents background information on FPGA architectures, and related work in hardware compilation. Chapter three presents an introduction to the POSIX multithreading programming model. Chapter 4 describes the hybrid thread abstraction layer, context and control for hardware threads, and the VHDL application interface. Chapter 5 starts with a general description of classic atomic operations, followed by descriptions of our evolutionary prototypes for synchronization primitives. This chapter then describes the design and implementation of global queues and controllers to control multiple instances of synchronization variables with efficient hardware costs but not at the expense of execution times. The integration of all core modules, testing procedures, performance tests and summary of results are covered in Chapter 6. In Chapter 7, we present an evaluation of our hybrid thread model with image processing applications. This thesis concludes with future research direction in Chapter 8.

2 BACKGROUND

2.1 FIELD PROGRAMMABLE GATE ARRAYS TECHNOLOGY

2.1.1 Introduction

The Field Programmable Gate Array (FPGA) was first introduced in the mid-1980s, and is in the class of programmable devices that provide the benefits of custom hardware, but avoiding the initial cost, fabrication time delay, and inherent risk of conventional masked application specific integrated circuits (ASIC). The primary advantage of an FPGA over an ASIC is that it can be re-programmed an unlimited numbers of times to implement a wide variety of customized digital systems. In the early days, FPGAs were used as testing and prototyping devices. As fabrication technology improves, the use of FPGAs have widened to include "glue logic" to replace multiple discrete chips with a single components, custom accelerators in digital signal processing applications, and as general purpose high performance co-processors. Figure 2-1 shows the maturation of FPGA's happened in the last ten years. Within this period, FPGA's have significantly gained in density (200-fold) and speed (20 times faster) but their prices continued to decrease (300 times) [40].

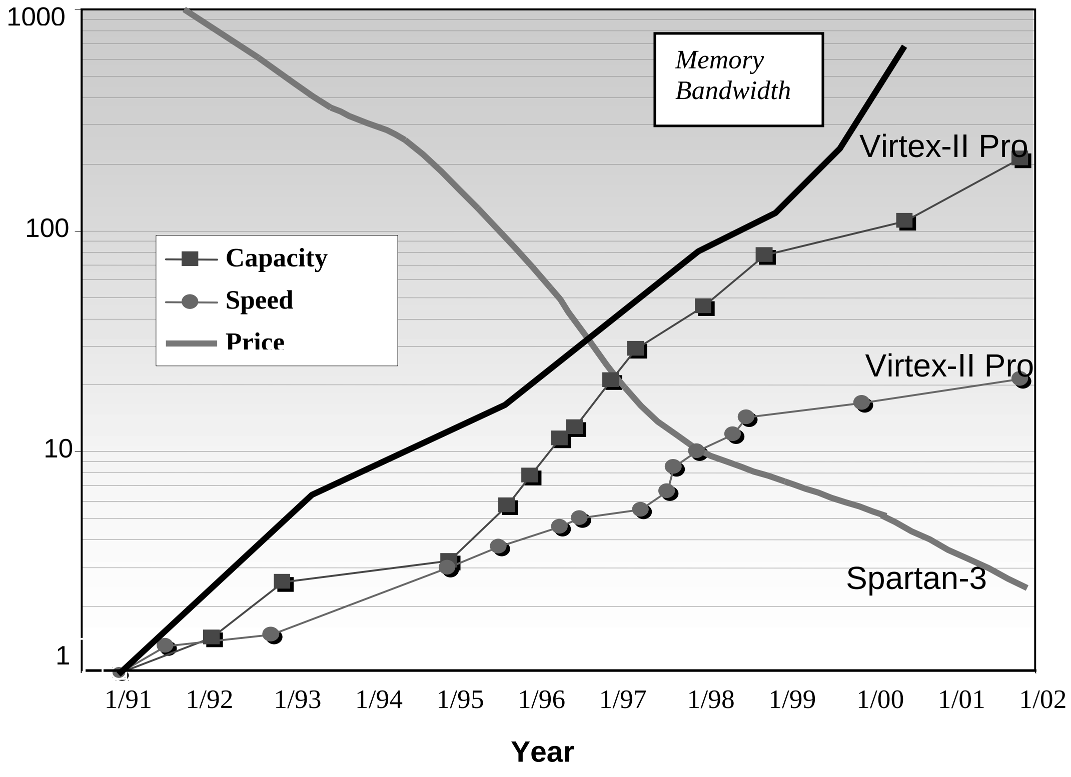


Figure 2-1 History of Xilinx FPGA in the Last Ten Years [40]

Presently, FPGAs have matured to a stage where they can host a complete embedded system, including a CPU, support components and other complex application specific functions. For example, a XILINX VIRTEX XC4VFX140 device now contains 2 RISC processors, 10Megabits memory, and 140K logic cells, with an operation frequency as high as 500Mhz [61].

The internal structure of a Xilinx FPGA is shown in Figure 2-2. This device that consists of a matrix of configurable logic cells (CLBs), with a grid of interconnecting routing lines and switches between them. Input/output blocks (IOB) exist around the perimeter to interface the internal interconnect lines and external package pins. The specific implementation and capabilities of a CLB varies with the manufacturer of the device. Xilinx CLBs are comprised of combinational logic and storage cells. The storage cells can be used as look up tables for realizing Boolean logic equations or as storage devices.

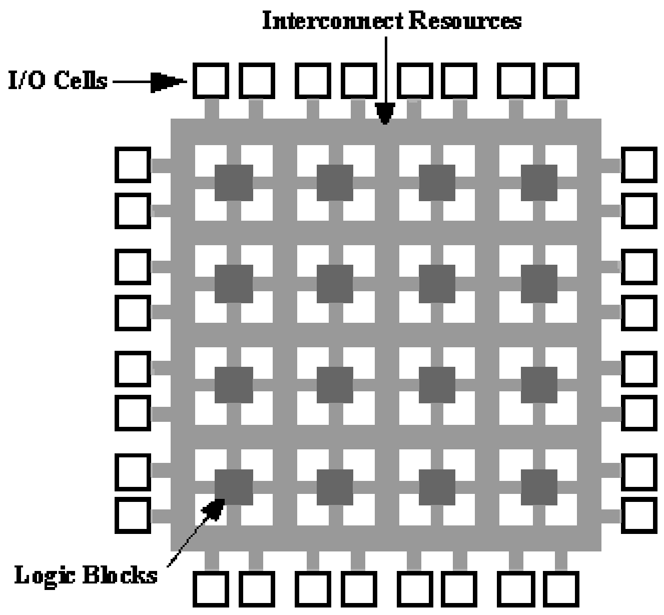


Figure 2-2 Programmable Logic Array [34]

The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLB and IOB onto appropriate networks. Programming or customizing an FPGA includes configuring the logic cells to implement Boolean functions and connecting the switches in the interconnect lines to both route variables between functions and also to compose local functions into more complex functions.

2.1.2 Configurable Logic Block (CLB)

A Xilinx CLB (Configurable Logic Block) element is shown in Figure 2-3. Essentially, each CLB contains a pair of single bit flip-flops and two independent look-up table function generators. These function generators are configurable either as four input lookup tables (LUT), two bits shift registers or two bits distributed RAMs. Each look-up table takes four bits of inputs from the routing network and generates a one-bit output. By filling in the look-up table with appropriate bits, any four-bit logic function can be implemented. The table output can optionally be latched by a flip-flop before being send back to the routing network to other logic blocks.

Configurable Logic Blocks implement most of the logic in an FPGA. Four inputs is a good size for a look-up table as suggested by various studies, trading utility (complexity of a block) against utilization (what fraction ends up in use) [34]. The symmetry of the CLB architecture is also important as it facilitates the placement and routing of a given intended function. In addition to the lookup tables, other related logic block resources such as dedicated carry chain circuits are included to facilitate and speed-up common user intended logical operations.

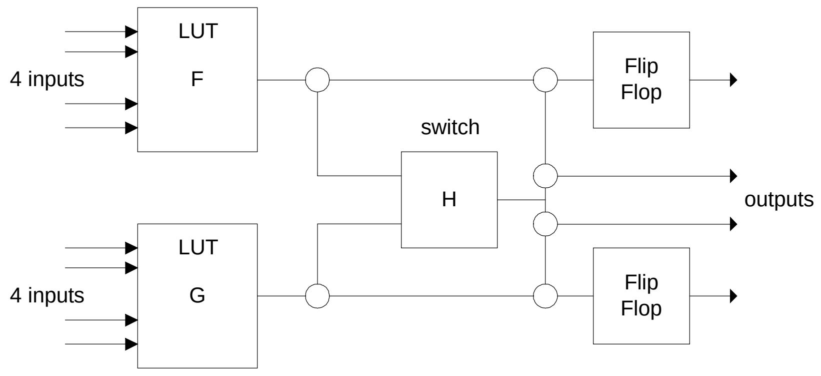


Figure 2-3 Configurable Logic Block [61]

In addition to CLB’s, current generation FPGAs include additional diffused hardware resources typically required for embedded systems. For example the Xilinx XC4FX140 (90 nm CMOS technology, 500 MHz) features dedicated digital signal processing 18-bits multipliers and accumulators (MAC), dual port memory block RAM (BRAM), digitally control clock manager (DCM), 32-bits 5-stage pipeline PowerPC RISC CPU, Ethernet MAC, fast input/output transceiver (fast I/O transmit/receive), significant number of input/output pins (I/O), shown in table 2-1 [61].

Device	CLB		Other Resources						
	Logic cells	Distributed RAM (KB)	DSP resource	18KB Blocks RAM	DCM	CPUs	Ethernet MAC	Fast I/O Tx/Rx	I/O Pins
XC4VFX140	142,128	987	192	552	20	2	4	24	896
XC4VFX100	94,896	659	160	376	12	2	4	20	788
XC2VPro30	30,816	428	136	136	8	2	-	8	644
XC2VPro7	11,088	154	44	44	4	1	-	8	396

Table 2-1 Current Generation of FPGAs

2.1.3 Sample of current generation FPGA

The Virtex II Pro FPGA device family from Xilinx was chosen as our experimental platform based on the availability of diffused and soft IP. For example a XC2VP125 FPGA includes two PPC405 processor cores with up to 44,096 CLBs. Other resources include 18Kbit block RAMs (BRAMs), 18bitx18bit multipliers and digitally control routing resources. The block RAMs are extremely useful to store temporary data and are used throughout our design to hold state information. The processors operate at clock speeds up to 300 MHz [60]. The FPGA logic can be clocked up to 400Mhz, however the final operation speed will naturally depend on the critical path of the implemented Boolean circuits.

Xilinx also provides a library of intellectual property (IP) cores. An IP core is a pre-made logic block that can be implemented on FPGA or ASIC. As essential elements of design reuse, IP cores are part of the growing electronic design automation (EDA) standard components. IP cores fall into one of three categories: hard, firm and soft cores. Hard cores are physical manifestations of the IP diffused into the silicon circuitry. Soft cores are provided as a list of the logic gates as an HDL module. The soft core IP provided by Xilinx includes serial ports, Ethernet controllers, processor busses (PLB), peripheral busses (OPB), bus arbiters, memory controllers and the micro-blaze processor [61].

CPU/FPGA Hybrid

Specific to our studies, the Virtex II Pro V7 has an IBM Power PC 405 RISC CPU hard core embedded in the FPGA fabric logic. This high level of integration between a CPU and an FPGA (CPU embedded within the FPGA) allows significant flexibility to attach peripherals or other IPs to CPU. A wide range of peripheral IP can be implemented out of the FPGA logic fabric, and accessible from the CPU via the standard processor local bus (PLB) or on-chip peripheral bus (OPB). The details of these busses are described later. This configuration allows users to create their own IP and connected it to the CPU via one of the busses. In fact almost the whole computer system that used to be on a printed circuit board can now be implemented within this single FPGA chip (except the main memory chips).

Processor resources

The Power PC 405 hard core is based on IBM-Motorola PowerPC RISC processor architecture. The Power PC 405 architecture is optimized for embedded systems applications (low power). It implements a subset of the PPC32 instruction set with additional extensions. An application binary interface (ABI) provided by IBM serves as an interface for compiled programs to system software [52]. The embedded Application binary (EABI) is derived from the PowerPC ABI supplement to the UNIX System V ABI. The ABI differs from the supplement with the goal of reducing memory usage and optimizing execution speed. The EABI describes conventions for register usage, parameter passing, stack organization, small data area, object file and executable file format.

Several low level details of the Power PC 405 architecture should be mentioned. First, the Power PC 405 architecture does not have a push/pop instruction for manipulating the stack. Instead, the architecture treats the stack pointer register as a general-purpose register that can be manipulated using standard load/store register to memory instructions. Second, the PPC 405 does not have expose internal signals necessary to lock the bus for synchronization operation (semaphores). Although there are reserved instructions for synchronization operations useful for synchronizing multiple processors configured in a shared memory processor (SMP) configuration, successful concurrency control among multiple processors requires additional external mechanisms. This is a critical issue for realizing hybrid threads in our system and necessitated the creation of more efficient hardware based synchronization primitives.

Core Connect Buses

Xilinx provides the standard IBM Core Connect [50] bus as soft IP to connect peripheral IP cores to the processor. Core Connect provides three levels of hierarchical buses: processor local bus (PLB), on-chip peripheral bus (OPB) and device control bus (DCR). The processor local bus (PLB) is used to connect processor cores to the system main memory and other high-speed devices. The OPB bus is dedicated for connecting slower on-chip peripheral devices indirectly to the CPU. The OPB bus supports variable size data transfers and as well as flexible arbitration protocols. Both the PLB and OPB busses have their own bus arbiters, and the two busses are interconnected by at least one bridge.

Xilinx provides a convenient bus attachment interface layer for each of the three buses in the form of soft core IP. The attachment, called the IPIF, allows peripheral IPs or other cores to connect to either of the buses. The IPIF is decomposed into two layers to allow easy migration of peripheral or IP cores for each of the different system buses. The first layer provides an interface facility (including set of standard signals) to be used between the IP core and the IPIF. The second layer is a bus specific portion, and interfaces the IPIF to one of the buses. To move an IP core from one bus to another requires only substitution of the second layer.

The IPIF provides two different types of attachment to an IP core: a slave and a master attachment. With the master attachment, user cores have ability to initiate bus transactions. Bus arbitration logic is included within the master attachment. However it is the user core's responsibility to re-arbitrate or abort the bus and switch the data bus from slave mode.

2.2 PROGRAMMING OF FPGA

In current practice, hardware descriptive languages (HDL) are widely used to implement applications on the FPGAs. However, using HDL requires knowledge of hardware details such as timing issues, propagation delay and signal fan-out. New techniques are emerging that attempt to raise the level of abstraction required to program FPGA's. With these techniques, the designers are no longer required to possess low-level hardware knowledge when implementing their applications onto FPGA's. In addition, researchers are seeking solutions to remove the boundary between hardware and software components. Widely use FPGA programming languages are discussed in the following sections.

2.2.1 Hardware Descriptive Languages

VHDL [60] and VERILOG [59] are the two most widely used hardware descriptive languages (HDLs) in use today for specifying digital systems. HDL syntax and semantics includes explicit notations for expressing time and concurrency, two primary attributes of hardware. VHDL, which stands for Very High Speed Integrated Circuit Hardware Descriptive Language, was initially developed by Department of Defense for documentation and design exchange, and later adopted as standard for hardware language by the IEEE [18]. VHDL, which is based on a discrete event concurrency model, contains language elements that are capable of supporting behavioral, dataflow and structural models.

In VHDL, the primary hardware abstractions are *entities*. They are used to identify and represent digital systems. An entity interfaces to the external world with well-defined input and output ports. The function to be performed by a digital system is specified inside the *architecture* definition within an entity. The function can be described either behaviorally or structurally or in combinations of both. Very basic building block entities are specified behaviorally. Then these basic entities can be structurally connected to form a larger entity. For example the predefined Xilinx block RAM entity can be wired to a controller entity to form a memory subsystem entity. Interconnection of multiple entities adds up propagation delays, thus care must be taken to ensure that the delay for the critical path of implemented circuit does not exceed the system clock period. The functionality of a large entity can be described using combinations of dataflow, structure and behavior specifications.

VHDL supports a two-level behavioral hierarchy. At the first level, specification can be decomposed into sets of concurrent processes. At the second level, sequential execution can be specified within a process. Additionally, to support notion of time, VHDL has *signals*, which differ from variables in that their values are defined over time. These signals can be activated either asynchronously or synchronously. These signals are employed as communication mechanism between concurrent processes.

Synchronization in VHDL can be implemented in two ways, either using the process *sensitivity li*st or *wait* statements. A sensitivity lists provides initiating events for evaluating a process. As such, the sensitivity list must consist of all events or signals that can trigger a reevaluation of the process. To define synchronous processes the clock should be the only signal in the process's sensitivity list. Although convenient for simulating processes, the sensitivity list alone is insufficient for actual implementation of the circuit in hardware. To support implementations, VHDL requires implementing conditional statements within the process body instead of the sensitivity list.

Synthesis

After creating and simulating a digital design, then the circuit must be synthesized for actual implementation. Synthesize is the process of translating a design to a gate level representation which can be mapped to the hardware resources within the FPGA. The synthesize tool takes the logic design, target technology features of the FPGA, and constraints specified by the user, and generates a net-list of gate-level representations. Synthesize processes also typically involve development of logic design in terms of library components, and optimization on area and gate delay (the synthesizer is not aware of wire delay).

Implementation

The synthesizer outputs a netlist description of the design. The netlist is a standard format that can then be mapped onto the physical logic elements and interconnection networks. This involves three steps: the mapping of logic to physical elements, placement of resulting elements, and routing of interconnect between the elements. The output of the physical mapping is a bit-stream file. In the case of SRAM based FPGAs, the bit-stream programming tool generates the physical implementation in the form of CLBs, IOBs, BRAMs, other FPGA resources and interconnections between them.

Download

Download include clearing configuration memory, loading the bit-stream (configuration data) into the configuration SRAM, and activating logic via a startup process. For non-volatile configurations, the bit-stream can be stored in either EPROM or EEPROM. The configuration data represents values stored in SRAM cells: CLB implement logic with SRAM-truth tables, SRAM-control multiplexers and routing that makes use of pass transistor SRAM switches (making/breaking the connection wire segments).

Disadvantages of HDL

Hardware descriptive languages such as VHDL offer the advantages of expressiveness in term of temporal and fine grain parallelism. As such implementing applications in HDL requires understanding of hardware details including clock cycles, hardware architectures and signal fan in/out, thus entails considerable efforts. There have been substantial research interests investigating on extending the high level languages to bring them into the hardware domain. Three of the more common efforts are Streams-C [21], Handel-C [48] and SystemC [54], which are presented in the next sections.

2.2.2 Streams-C

Introduction

Streams-C [21] was developed for systolic type processing at Los Alamos Laboratories, and extends the C programming language with capabilities for supporting reconfigurable logic hardware. The objective of the Streams-C project was to bring a new high-level language capability into the FPGA design environment. The research effort attempted to free application developers from the low level details of gate-level design, enabling application programs for both the FPGA and CPU to be written in a high-level language. Streams-C supplements the C language with a set of additional annotations and callable function libraries. The annotations are used to declare and assign hardware resources on the FPGA. The resources include processes, streams, and signals. The libraries provide communication facilities between different processes based on low-level handshake signals for hardware process synchronization.

The Streams-C environment includes a multi-pass compiler, and hardware and software libraries targeted for stream based applications. The characteristic of stream-based computing can be described as having a high data flow rate, fixed size, small stream payload, and repetitive computation on the data stream [54]. As such, it is an appropriate tool for implementing image or video processing type algorithms on FPGAs. A pre-processor converts the annotations and macro calls (SC_MACRO) into PRAGMAS, and passes them to the compiler. For hardware processes, the compiler generates a Register-Transfer-Level representation (or VHDL) targeting multiple FPGAs on the Annapolis Microsystems Wild Force [58] board. For software processes, a multithreaded software program is generated.

An application study of Streams-C has demonstrated that the circuit's areas of compiler-generated design are 1.37 to 4 times larger than that designed by VHDL. But the time spent to implement the applications with Streams-C is favorably short. Streams-C study group claimed that applications written in Streams-C could be completed five to ten times faster than designs that implemented using VHDL [18].

Hardware (Synthesis Compiler Library)

An application in Stream-C can be implemented as a collection of processes that communicate using streams and signals. Processes can run either in software (CPU/host computer) or on the hardware (FPGA).

The hardware process module has two main components, a data-path component (VHDL process) and an instruction sequencer. The data-path component is decomposed into a data-path entity and a pipeline control entity. The data path entity can be broken into instruction decoder and data-path circuits. The instruction sequencer is a state machine that sequences the instruction set of the process module. A process may have interface ports for stream, signal, external memory and block RAM. Processes communicate by means of stream modules or signals.

The stream modules are FIFO (first in first out) based synchronous communication channels between processes [21]. Each channel has a different data width to match the stream payload. It is parameterized with respect to data register width and FIFO depth. The data width ranges from 16-bit or 32-bit to 64-bit, specific to the size of the stream payload. Examples of stream modules are StreamFifoWrite (software process to hardware process FIFO), StrmFifoRead (hardware to software FIFO) and StreamIntraRead (hardware process to another hardware process). The stream module uses signals to indicate it is ready to receive or output data. An example of process and stream declarations in Streams-C is given in Figure 2-4.

```
/// PROCESS_X controller
/// INPUT frame_input               // input stream
/// OUTPUT frame_output             // output stream
/// PROCESS_X_BODY
SC_FLAG(tag)                        // stream element with one-bit flag
SC_REG(frame_word, 32);             // 32-bits stream port
SC_STREAM_OPEN(frame_input);        // stream operation open
SC_STREAM_READ(frame_input, frame_word, tag);
///PROCESS_X_END
```

Figure 2-4   An Example of Streams-C Process Declaration

Streams-C Language Construct

The Streams-C language consists of a small set of annotations and library functions callable from a conventional C program. The annotations are used to declare and to assign resources on the FPGA to these following objects: processes, streams, and signals. The libraries provide low-level hardware stream communication facilities and synchronizations between processes. An example of a stream-oriented computation is depicted in Figure 2-5 [55].

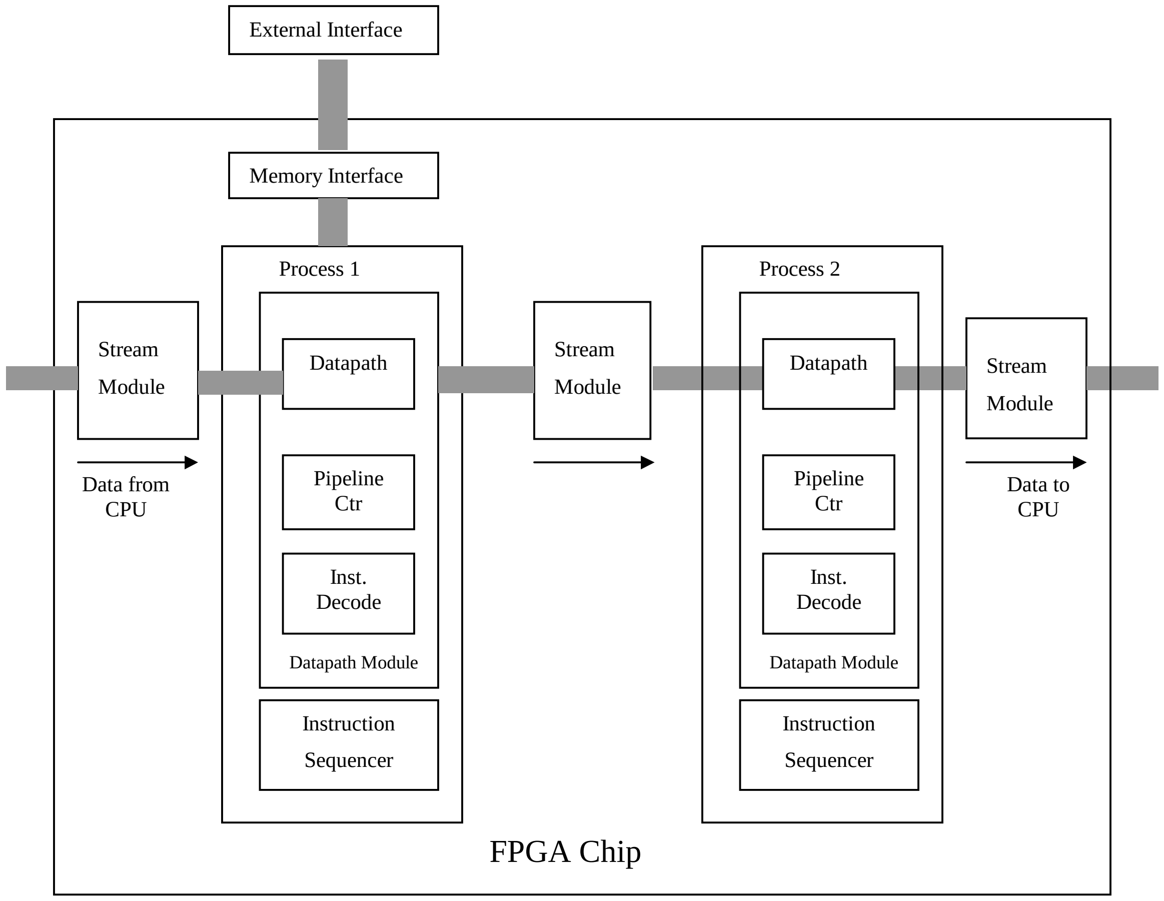


Figure 2-5 Streams-C Hardware Processes

The hardware process 1 (on the left) receives stream of data (or images) from a software process running on the CPU via the PCI bus. Then hardware processes 1 and 2 manipulate the stream, and the result is returned back to another software process (on the right). In this example, processes communicate and synchronize via the low-level hardware stream modules. The figure also shows a memory interface to enable hardware process 1 to access the external memory. Each hardware process has instruction sequencer and data-path modules. Stream data is processed in the data-path module while the sequencer is the activity coordinator.

Streams-C Compiler

The Streams-C compiler [18] as depicted in Figure 2-6 is based on the multi-pass (Stanford University Intermediate) SUIF infrastructure compiler [56].

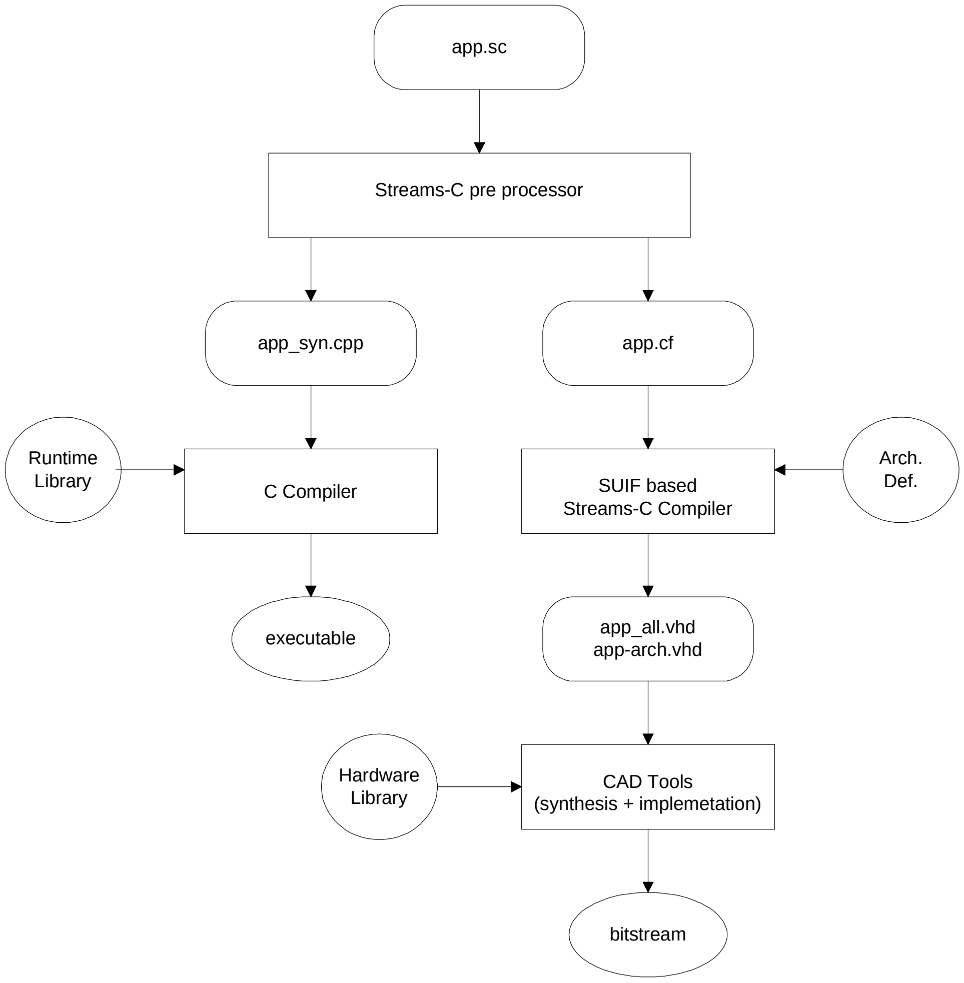


Figure 2-6 Organization of the Streams-C Compiler [18]

It translates the C program (the FPGA processes part) into Register-Transfer-Level (RTL) or VHDL and is capable of generating pipelined stream computations. Hardware processes are written in a subset of C, and compiled into data-path modules on the FPGA. Features of the Streams-C compiler include semantic validation of processes, streams, pipelining, state machine generation for sequencing and stream communication libraries.

2.2.3 HANDEL-C

Introduction

Handel-Cs approach to bringing high-level languages into the hardware design domain, shares commonalities with Streams-C. Like Streams-C, it adopts C like syntax that can be directly compiled into synchronous digital hardware. The Handel-C language consists of subset of the C programming language and additional "low-level" augmentations for describing parallel operations and specific hardware components [48].

Being not a hardware descriptive language, its compiler does not produce optimized hardware circuits. It is however focused on fast prototyping and optimizing at the algorithmic level. Program execution in Handel-C by default follows a sequential path, rather than maximizing concurrency. Although programs execute sequentially, Handel C supports the *par* construct (parallel), to enable a process to spawn multiple sub-processes (branches). All sub-processes within the parallel construct will be executed concurrently, and execution flow rejoins when all the sub-processes complete. Any sub-processes that complete early must wait for all other processes to complete.

Handel-C Computation Model

Handel-C is based on the Communication Sequential Processes (CSP) model [49], and extends the C language to overcome concurrency deficiencies of the basic language. Handel-C allows programs to be specified as set of concurrent processes, using constructs that simplify the specification of communication and synchronization between these processes. Communication between concurrent processes can be achieved by means of message passing, with named non-queue communication channels. A process block must wait until the other process is ready to send or receive data over the channel.

The Handel-C compiler [49] was designed to hide the low gate-level details such as propagation delays, clock skews, and pipeline lengths. Handel-C augments the high level language with the capability to express the notion of time. The notion of time is simplified into two specifications; time advances in a computation in units of one clock cycle, and variable assignments require exactly one clock cycle. Thus it allows only the design of synchronous digital circuits.

In contrast to HDL, which supports the specification of low-level concurrency, Handel-C adheres to the sequential flow of the governing C program. Each assignment in the source program executes in exactly one clock cycle. An application can be broken down into sets of sequential units of computations called branches. Parallel branches communicate over a named non-buffered blocking communication channel. One branch has to wait (block) another branch for sending and receiving of data over.

Language Construct

Handel-C basically consists of a subset of the C language extended with additional constructs such as *par* to exploit hardware parallelism, *delay* for timing, *ram* for built hardware component and others. A list of the Handel-C language constructs is given in the table 2-2. Programs in Handel-C by default are made up of sequential constructs. However designers can take advantage of hardware parallelism (using par construct) for parallel processing.

Constructs	Descriptions
Par	Parallel execution
Delay	One clock delay
Chan	Channels for communications
?	Reads from channel
!	Write to channel
prialt	Select first active channel
seq	Sequential execution
signal	Hold value for one clock cycle
interface	External connection
Width(…)	Determine number of bits
ram/rom	Memory devices

Table 2-2 Example of Handel-C Language Constructs

Unlike conventional C which variable size cannot be reduced to less than 8 bits width, hardware-optimized constructs such *bit-width* can be used to size variables or constants to as small as one bit width, when declaring a simple flag, allowing efficient use of hardware resources. The channel construct is to support CSP synchronous channel point-to-point communication. Other constructs include special data path variables (variables mapped to registers), logical, bit manipulation, arithmetic, relational operators, delay construct, assignment and flow control. The delay instruction takes one cycle.

Handel-C Program Examples:

a) Declaration syntax extended for bit-width (int n x);

```
int 4 x, y;            // define variable x, y as 4 bits variable
unsigned int 2 z ;     // define variable z of type integer, size is 2 bits
```

b) Sequential Expression

```
{
   x  =  1;    //  assignment statement execute sequentially
   y  =  2;    //  requires two clock cycles
}
```

c) Parallel Expression

```
par {
   x = 5;      // assignment statement run in parallel
   y = 2;      // statements within par take one clock cycle
 }
```

d) Synchronization between parallel branches:

An example of two concurrent processes (two parallel branches) communicate by via a channel is given in Figure 2-7:

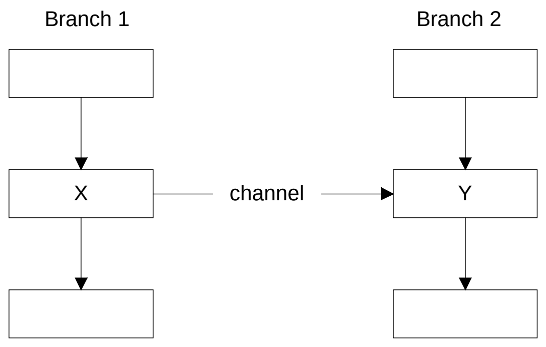


Figure 2-7   Two Parallel Processes

The communication between the two branches can be achieved by the following constructs:

channel ? variable	- to read a value from a channel and assigns it to variable
channel ! expression	- writes a value resulting from expression evaluation to a channel

In each case the writer or reader is made to wait if there isn't a reader or writer at the other end of the channel (branch X has to wait for Branch 2 to reach state Y, if it reaches state X earlier).

2.2.4 SYSTEM C

Introduction

Recently, multiple components including CPU, digital signal processors, memories, busses, interrupt controllers, busses, and embedded software can be implemented within a single chip called system-on-chip (SOC). To manage complexity of these SOC and to reduce design time, designers are focusing on raising the design abstraction to system level design environments. System level design environments enable designers to deal with hardware and software design tasks simultaneously [44]. These tasks include modeling, partitioning, verification and synthesis of a complete system. Current approaches of providing system level design tools include [23]:

- Reusing existing hardware languages, adapting and recreating new methodologies. For example System VERILOG adapting VERILOG to include creation and verification of abstract architectural level model.
- Extending high-level languages with hardware design capabilities. Examples of these efforts include Spec C, Handel C and System C.
- Creating new languages like Rosetta.

System C extends the C++ language with hardware system descriptions targeting SOC devices. It allows hardware modeling with explicit concurrent processes and communication channels. System C supports multi-level communication semantics to enables system input/output protocols with different level of communications abstraction. A port is an abstraction used to describe communication interfaces at different levels of abstraction including data transaction level and bus cycle level [45, 46].

System C Language

System C language includes constructs such as processes, modules, channels, interfaces and events. A system may be modeled as a collection of modules that contain processes, ports, channels, and even other modules. As modules can be instantiated within other modules, structural design hierarchies can easily be built. A channel is an object that serves as a container for communication and synchronization. A channel implements one or more interfaces. An interface is simply a collection of access methods or function definitions within a channel. Therefore the interface itself does not provide the implementation. A process accesses a channel's interface via a port on the module. Ports and signals enable communication of data between modules, and all ports and signals are declared by the user to have a specific data type. An event is a low-level synchronization primitive that can be used to construct other forms of synchronization. Channels, interfaces and events enable designers to model a wide range of communication and synchronization that can be found in the system design. Features of System C class library include [46]:

Modules:

Modules are considered as container classes (like C++) or fundamental building blocks. They are hierarchical entities that other modules or process can be defined within them. Modules and processes communicate by means of functional interfaces.

Processes:

Processes define the behavior of a particular module and provide methods for expressing concurrency. Processes can be hardware or software. Processes can be stand-alone entities or can be contained within modules. Process abstractions include asynchronous blocks and synchronous blocks. Processes communicate through signals. Explicit clocks are used to order events and synchronize processes.

Signals:

Signals can be resolved or unresolved types. Resolved signals have one driver while unresolved signal can have more than one driver. Clocks are considered as special signals. Multiple clocks with arbitrary phase relationship are also supported. Mechanisms such as waiting on clock edges events, signal transition, and watching for event like reset event are included to support reactivity modeling.

Rich set of signal type (data type):

System C has rich set of signal types to support different design domain and abstraction level. The abstraction level ranges from high-level functional model to low register-transfer level. Signal types include single bit, bit vectors, fixed precision type (especially for simulations or digital signal processing), four-states logic, arbitrary precision integer value, and floating point [43].

2.4.5 Summary

In summary, advancements have been made in bringing high-level languages into the domain of hardware design and toward seamless integration of hardware and software components. However the research efforts described above are lacking especially in terms of hardware and software integration. For example, in the case of Stream-C, the communication between hardware and software components is achieved by using low-level streams and signals. Moreover Stream-C is designed to handle systolic-based computation only. Handel C research effort was mainly focus on raising the abstraction level to program the FPGA. Language constructs such as “interface” and “ROM/RAM” in Handel C still require some knowledge of hardware details. The synchronization of software and hardware components in Handel C environment is achieved by means of low-level communication mechanisms. These efforts do not abstract away the boundary between hardware and software components. Therefore new approaches are required, including adopting system level and programming model methodologies to resolve these issues. Programming models, specifically the multithreading programming model is discussed in the next chapter.

3 MULTITHREAD PROGRAMMING

3.1 INTRODUCTION

It is standard for operating systems today to support multiple processes in order to achieve better resource utilization and processor throughput. The multithread programming model evolved as a light multiprocessing model where each thread has it's own execution path, but all threads share the same address space. On single CPU machines, this allows a thread to block on a resource and allows other threads within the same program to continue execution. The thread scheduler achieves this capability by interleaving processing resources between multiple threads, thus giving the illusion of concurrency on a single processor. Performance improvements can be gained on a single processor system as it allows slow input/output device operations to overlap with computations on a processor.

3.2 THREAD

A thread is an abstraction that represents an instruction stream that is able to execute independent of all other threads. A thread possesses its own stack, register set, execution priority and program counter as summarized below and shown in Figure 3-1. Additionally, each thread is also assigned a unique identification code (ID)

- Program counter (current execution sequence).
- Stack pointer
- Stack frame
- State (other registers value beside stack pointer and program counter)

In addition to its own private execution context, all threads within a process share the process resources such program code, heap storage, static storage, open files, socket descriptors, other communications ports, and environment variables equally.

In a single-processor system, only a single thread of execution is running at a given time. The CPU quickly switches back and forth between several threads to create an illusion of concurrency.

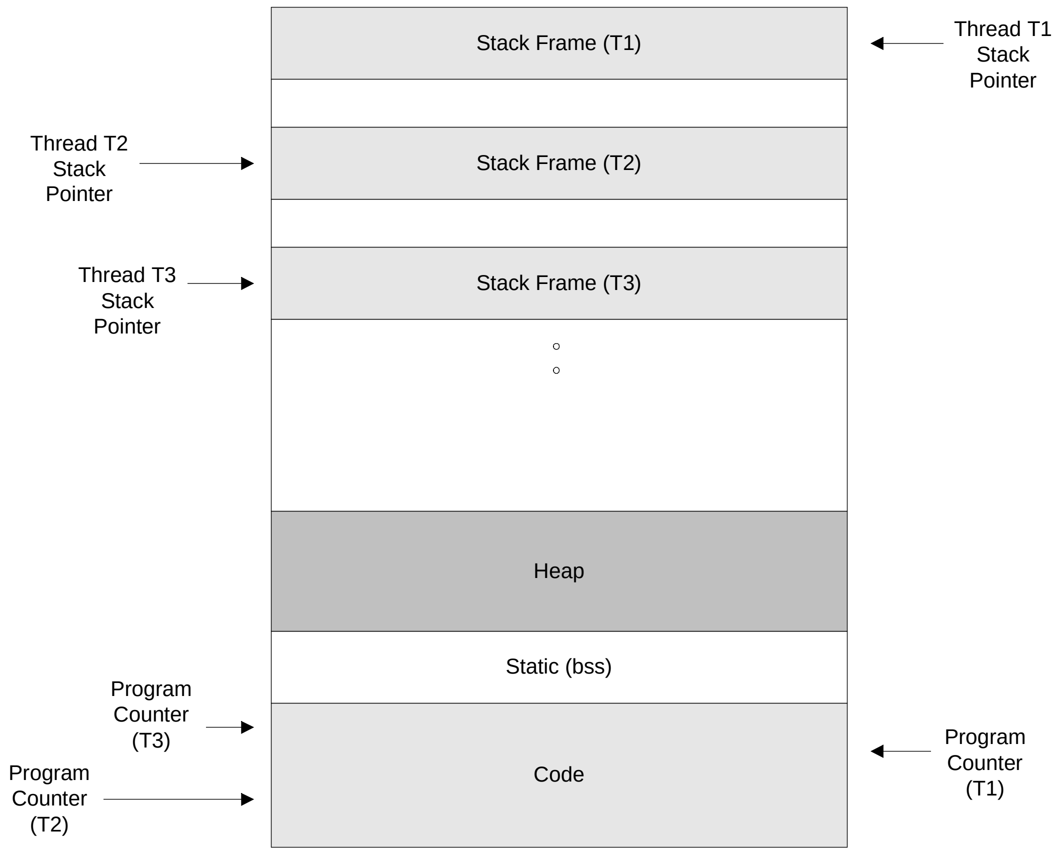


Figure 3-1 Threads within a Process

This means that a single-processor system supports logical concurrency, not physical concurrency. On multiprocessor systems, several threads do in fact execute in parallel. Thus physical concurrency is achieved. The important characteristic of multithreading is that it creates logical concurrency between executing threads that can also be implemented using physical concurrency, option based on the platform configuration.

Thread context switching is much cheaper than the context switching required between processes. To switch threads, only the execution context is needed, so minimal kernel services are required. This leads to at least two approaches for thread scheduling:

- *User-level threads* are scheduled independent of the kernel using a thread library. To the kernel, the multiple threads appear as a single-threaded process. The advantage of this approach is that switching between threads is fast as mode switching is not needed and fairly portable. The disadvantage of this approach is it does require additional code to be written in assembly. For example context switches and certain parts of code must be able to execute atomic instructions. This type of thread complicates its implementation, as all I/O must be handled in a non-blocking manner.

- *Kernel-level threads* are scheduled in the kernel together with threads of other processes. This approach has the advantage that multiple threads can be assigned to multiple processors. The drawback of this approach is that two mode switches are required when scheduling different threads.

The advantages of multithreading can be summarized as follows:

- Better utilization of the CPU in the presence of slower I/O devices, by switching out threads that are waiting on I/O devices, and switching in a thread that is ready to run.
- Concurrency can be used to provide multiple simultaneous services to users. Users perceive improved application responsiveness, if dedicated threads are used to serve different services such as displaying outputs or reading inputs.
- The use of threads increases code visibility and makes code extension simple as it provides more appropriate structures for programs to interact with the environment, control multiple activities, and handle multiple events.
- Some applications are inherently concurrent in nature. For example a database server may listen for numerous client requests, service concurrently active or data ready connections. Scientific calculations that compute terms in an array, each term independent of the others can be broken into multiple threads.
- Multithreading provides benefits to a large job when it can be divided into smaller jobs and distributed amongst multiple processors for greater efficiency. Threads also can help to deliver scalable multiprocessor systems.

Thread concurrency can introduce race conditions when multiple threads attempt access to shared data without proper coordination. Race conditions are introduced by non-deterministic execution sequences from input or output completion, signals, and the preemptive action of a scheduler.

Accesses to the shared resources are serialized and controlled with the aid of concurrency control, or synchronization mechanisms. Proper use of synchronization mechanisms guarantees the elimination of these race phenomena. The standard synchronization mechanisms in use in multithreaded programming are discussed in the following section.

3.3 SYNCHRONIZATION MECHANISMS

Management of shared resources is fundamental to the successful implementation of concurrent programming model. Accesses to the shared resources by the concurrently executing threads must be serialized to avoid programming errors or undesired inconsistent results. Processes or threads that share access to these resources must execute in a mutually exclusive manner. These shared resources are also known exclusive resources because they must be accessed by one thread at a time.

Accesses to these exclusive resources are usually coordinated explicitly by programmers, using concurrency control mechanisms such as locks, mutual exclusion (mutex), semaphores and condition variables. Semaphores are useful for controlling countable resources, while condition variables are employed for event waiting. These synchronization mechanisms are enabling mechanisms that elevate the concurrent programming to a higher level than individual processor instructions, permitting segments of programs to execute in apparent indivisible operations with no interleaving.

These sequences of statements that must execute in a mutually exclusive manner are typically referred to critical sections [33]. There are a number of requirements that need to be satisfied when processes or threads execute within critical sections to ensure fairness and symmetric progression [33, 39].

Mutual exclusion:

- Only one process is in the critical section at one time.

Progress:

- Progress in absence of contention. If no process is executing in the critical section, a process that wishes to enter a critical section will get in. This ensures that if one process dies, the others are not blocked.
- Live-lock freedom: process must not loop forever while in critical section.
- Deadlock freedom: if more than one process want to enter a critical section, at least one of them must succeed.

Bounded waiting:

- Getting fair chances to access to a critical section and no starvation: if a process wishes to enter a critical section it will eventually succeed. No thread or process is postponed indefinitely.

For a long critical section, threads that fail to acquire an unavailable synchronization variable should be put to sleep instead of wasting processor resources busy waiting. Each semaphore can have an associated queue in which to place the sleeping threads. When a semaphore is released, a wake-up mechanism transfers a thread from the semaphore wait queue onto the ready to run queue. Such synchronization mechanisms that support sleeping threads are referred to as blocking synchronization primitives. Although blocking synchronization primitives are advantageous, there are many scenarios in which polling the synchronization variable is more desirable. As an example, in a multiprocessor system it is more efficient to busy wait for rather than block when the rescheduling overhead is more expensive than short spinning times, and when bus contention is low. This kind of synchronization is called spin type synchronization.

Lock

The simplest type of synchronization mechanism is a mutual exclusion lock or more commonly referred to as a lock. A lock is essentially a binary variable that has two states: locked or unlocked. It is normally used around a critical section to ensure mutual exclusion or to obtain exclusive access to a shared resource. Only one thread can own the lock at a time. While a thread holds the lock, all other threads are prevented from opening the lock until relinquished by the owner thread. Thus locks protect critical sections from being executed simultaneously by multiple threads.

Spin Lock

A characteristic of a spin lock is that a thread ties up a CPU while attempting to unsuccessfully gain access to a critical section. Conversely a spin lock can be efficient when the amount of wait time for the lock is smaller than the time required to perform a context switch. Thus it is essential that spin locks execute for only extremely short durations. In particular, they must not be held across blocking operations. Depending on system requirements it may also desirable to disable interrupts on the current processor prior to acquiring a spin lock. The main advantage of using the spin lock is that its operation is inexpensive when the probability of lock contention is low. When there is no contention on the lock, the cost of both acquiring and releasing the lock typically amounts to few CPU cycles only. Thus, they are ideal to protect data structures that need to be accessed briefly or when the critical section is short. They are also normally used to protect higher order synchronization mechanisms.

Blocking Lock/Mutex

Depending on the length of the critical section, a thread may need to hold a lock for long duration. For such situations, it is more efficient for the threads that wish to own the lock go to sleep instead of wasting processor precious cycles, busy waiting for the lock to be available. Going to sleep involves inserting the requesting thread id into a sleep queue and calling the system scheduler to perform a switch context (changing its state to block, sleep on this resource, and relinquish the processor to another thread). When the current lock owner exits the critical section, it releases the lock, which generates a wake-up signal to the scheduler. If there is at least on thread in the queue, the wake-up mechanism will de-queue one or all the threads from the sleep queue, change their state to ready and transfer them to the scheduler queue. The next mutex owner can then be decided according to the scheduling algorithm.

A mutex has a flag to represent the usage state and a queue to hold blocked threads. A locked mutex may contain zero or more threads waiting in its queue. When the mutex is not locked, the queue is empty. When the mutex is unlocked and while its queue is not empty, one of the blocked threads will be removed from the queue and transferred to the ready to run queue. The following are application program interface (API) provided for POSIX mutex:

pthread_mutex_init(mutex) –	to initialize a mutex variable.
pthread_mutex_lock(mutex) –	to acquire a lock or mutex before accessing a critical section. The calling thread blocks if the mutex is not available.
pthread_mutex_trylock(mutex) –	to test whether a mutex is locked without cause the calling thread to block. Therefore, a thread can do other work instead of blocking if mutex is already locked.
pthread_mutex_unlock(mutex) –	to release a mutex and unblock a sleep thread if there is one in the mutex queue.

Semaphore

A semaphore is a synchronization mechanism normally used for controlling access to a countable shared resource. Each semaphore has a counter that can be used to synchronize multiple threads and a sleep queue to hold blocked threads. The counter can be incremented to any positive value or can be decremented to a non-negative value. Two atomic operations are used to change the value of the counter – wait and post operations. The wait operation decreases the value of the counter by one. If the value is already zero, the wait operation causes the calling thread to block until the value of the semaphore becomes positive. When the semaphore's value becomes positive, it is decremented by one and the wait operation completes. Essentially when the value of the counter is zero, any wait operation will cause threads to be blocked and queued into the sleeping queue and the counter value remain unchanged at zero. A post operation increases the semaphore counter value by one when the queue is empty. If the sleep queue is not empty, a post operation causes one of the threads in the queue to be unblocked, and the counter remains zero. The unblocked thread will be transferred to the scheduler queue.

sem_wait(semaphore)

- Decrements the semaphore counter value or the calling thread blocks if its current value is zero.

sem_post(semaphore)

- Increments the semaphore counter value if queue is empty or wakes-up at least one waiting thread and counter value remains zero.

Condition variables

Waiting in the sleep queue implies blocking until some event occurs. A condition variable waits atomically on an arbitrary predicate, which makes it a convenient mechanism for blocking threads on combination of events. A condition variable itself does not contain the actual condition to test, instead it is a variable that allows threads to block safely (on it) when the condition is not true. It has an associated lock that protects the condition to be tested. It is supported with three atomic operations: *waiting*, *signaling* and *broadcast*. These operations allow threads to block and wake-up within the context of the lock. To prevent lost wakeups, the lock is passed as an interlock when a thread blocks on the condition. Thus a condition variable supplements mutex lock by allowing threads to block and await signals from other threads when a condition is not true. When the running threads change the predicate, a condition variable wakes one or all the blocked threads. The awaken thread will attempt to obtain the lock before testing the condition. The following are application program interfaces (APIs) provided by POSIX for condition variables:

pthread_cond_wait(condition variable, mutex lock)

- Causes the calling thread to block on the condition variable and release its mutex lock.

pthread_cond_signal(cond)

- Awakens one thread waiting on condition variable.

pthread_cond_init(cond)

- To initialize a condition variable.

pthread_cond_broadcast(cond)

- Wakes up all threads waiting on a condition variable. These awakened threads contend for the mutex lock. If more threads are waiting, one is selected in a manner consistent with scheduling algorithm.

Atomic operation

All synchronization mechanisms rely on hardware to provide atomic operations. An atomic operation is an operation that, once started, completes in a logically indivisible way (i.e. without any other related instruction interleaved) [33]. Many systems provide an atomic Test-And-Set instruction or an atomic Swap instruction. Test-And-Set sets a memory location and returns its old value. If the return value is one, the lock is already own by another thread. Swap has two arguments and swaps the values of its arguments atomically. The Test-And-Set must executed atomically, even on multiprocessor systems. If Test-And-Set instructions are attempted simultaneously by multiple CPUs, they must be executed sequentially.

3.4 THREAD SCHEDULING

Figure 3-2 shows the possible states a thread may assume during its life. Typically, in a single processor environment, a scheduler manages the sharing of the CPU by switching the threads context in and out at periodic intervals. Many algorithms exist for determining when and how to select a thread for scheduling. By far, the simplest scheduling algorithm is the first come first served (FIFO) algorithm. In this approach, a thread that is running maintains the CPU until it relinquishes the CPU via blocking or termination. All other threads are then scheduled in the order that they were added to the ready to run queue. In the simple FIFO algorithm, a currently running thread cannot be pre-empted thus potentially achieving poor aggregate system performance. In contrast to the non-preemptive FIFO algorithm, preemptive scheduling algorithms allow the currently running thread to be taken off the CPU and replaced by a different thread. Preemption can be implemented based on time slicing, or priority assignments to the threads. For time slicing, a hardware timer normally generates a periodic interrupt to the scheduler to perform a forced scheduling decision. This type of time sliced periodic scheduler allows other threads of the same priority to gain a slice of time on the CPU. This approach is referred to as a round-robin scheduler. Additionally threads can be assigned priority levels, and a thread with a higher priority that has been moved from a blocked queue to the ready to run queue can immediately cause a preemptive scheduling decision. Thus, the thread with highest priority level will always be running on the CPU.

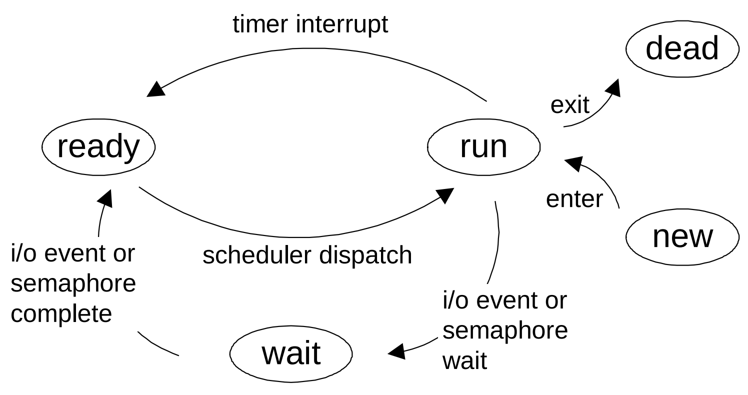


Figure 3-2 Thread States

While a thread is in the *run* state, it may transit to the *wait* state when it fails to gain a resource needed and is blocked. The resource includes a synchronization variable or data from a peripheral. The following are events that can cause a thread to change its state and results in context switching:

1. Synchronization – A thread that fails to gain a synchronization variable will change its state to *blocked* or *wait*, places itself in a waiting queue (waiting queue associated to the requested synchronization variable), and then calls the thread scheduler to allow another thread to run.
2. Preemption – Preemption occurs if a running thread does something that causes a higher priority thread to become *runnable*. The actions that causing this to happen include releasing a lock, changing the priority level of a *runnable* thread upward, lowering its (active thread) priority downward.
3. Yielding – The scheduler will dispatch another ready thread, if the active thread voluntarily yields and there is at least one thread in the scheduler queue. Otherwise an idle thread will run on the CPU.

Threads wait in sleep queues while in their *wait* state. A wake-up mechanism will change their state to *ready* and put them back into the scheduler queue when the requested synchronizations or the data from the peripherals become available. Description of each of these states is described in table 1.

States	**Descriptions**
Ready	Ready to run, but waiting for a processor or CPU.
Run	Currently executing on a processor. At least one is running with a maximum equals to number of processors.
Blocked or wait	Waiting for a resource other than the processor to become available. The resource is a synchronization or data from peripheral device.
Terminated or dead	Completes its execution but not yet detached or joined.

Table 3-1 Thread State Descriptions

3.5 CONTEXT SWITCHING AND QUEUES:

Thread Queues:

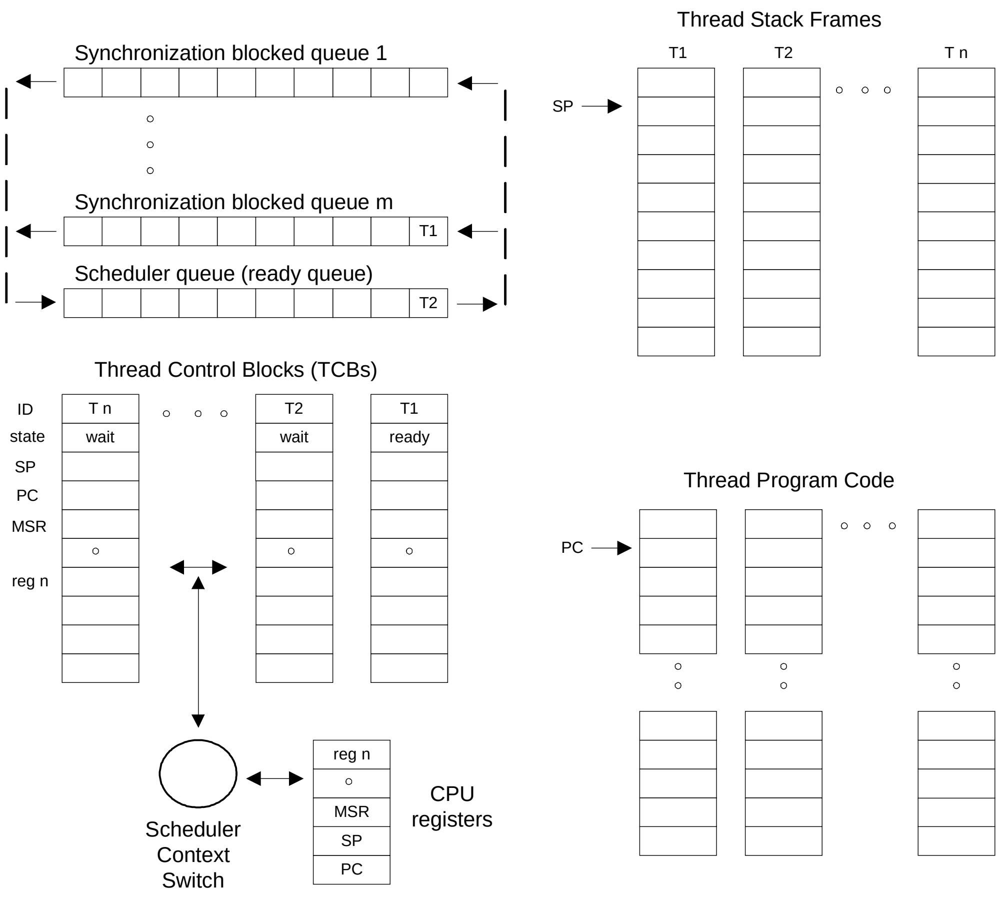
- Conceptually threads migrate between the various queues during their lifetime.
- The queues are actually used hold thread ID or pointers to thread control blocks (TCBs).
- Scheduler ready queue: ready queue points to the TCB (Thread control Block) of the threads ready to execute on the CPU.
- Synchronization blocked queue: it is for threads wait or block on a specific synchronization variable.
- Device blocked queue: one blocked queue per device, and is used to hold the TCB pointers of threads blocked waiting for an I/O operation (on that device) to complete.
- When a thread is switch out at a timer interrupt, it is still in the *ready to run* state, so its TCB pointer stays on the ready queue.
- When a thread is switched out because it is blocked on a semaphore operation, its thread ID is moved to the semaphore blocked queue.
- When a thread is switched out because it is blocked on an I/O operation, its TCB pointer is moved to the blocked queue of the device.
- An example of threads execution states on a CPU with various queues, thread stacks, and thread control blocks (TCB) is shown in Figure 3-3.

Except for certain operations and dependent on the scheduling algorithm, a thread scheduler normally invokes context switch procedure when the periodic scheduling timer expires or the thread blocks. The following are examples of events that can cause context switching and the corresponding sequences of operations that occur during the associated context switching operations.

Periodic Timer Interrupt:

a. Thread executing
b. Timer Interrupt occurs
c. Program counter changes to the vector of timer interrupt handler, and current thread state is saved
d. Interrupt Service Routine (ISR) runs
    i. Disables interrupt,
    ii. Checks if the current thread has run long enough

e. If YES post software (SW) trap
f. Enables interrupt
g. Returns from ISR
h. Check if SW Trap posted?
    i. If NO: Restores thread state
    ii. If YES: Performs context switch



1. TCBs and queues are in the static (bss) memory section.
2. Stack frames are in the stack memory section
3. Thread program code in the text memory section
4. PC is program counter
5. SP is stack pointer
6. MSR is machine state or condition code register

Figure 3-3 Thread Execution Representation and Supporting Structures

Blocking I/O call:

a. Thread executing
b. System Call I/O
c. SW Trap handler runs in kernel. Saves the current thread state
d. Kernel code (OS) runs the I/O call
e. I/O operation starts (I/O driver)
f. Updates thread state to WAITING
g. Adds thread ID to the Wait Queue of the requested I/O device
h. Performs Context Switch
i. I/O done (I/O interrupt)
j. Wakes-up waiting thread, moves it from the Wait Queue to the Scheduler Queue

Blocking semaphore call:

a. Thread executing
b. Thread calls Semaphore API
c. Software Trap handler runs in kernel. Saves the current thread state.
d. Kernel code (OS) executes the semaphore call
e. If block:
    i. Updates TCB thread state to WAIT
    ii. Adds thread ID to Semaphore Wait Queue
    iii. Calls scheduler to perform context switch to allow another thread to runs
f. The thread that currently owns the semaphore performs release call
g. Software Trap handler runs in kernel
h. Kernel code (OS) executes the release call
i. The semaphore is available now
    i. Wakeup at least one thread waiting in the Wait Queue
    ii. Move thread ID from Wait Queue to Scheduler Queue

3.6 THREAD SCHEDULING POLICIES

Threads that are queued and waiting on a synchronization variable may be unblocked using various policies. These policies define the semantics when a synchronization variable is released and there is more than one thread waiting to acquire the resource. Essentially a scheduling policy defines which waiting thread shall acquire the synchronization when the current owner releases it.

With a FIFO scheduling policy, threads waiting for the lock will be granted the lock in a first come first served order. This can help prevent a high priority thread from starving lower priority threads that are also waiting on the synchronization variables.

With a priority driven scheduling policy, the thread with the highest priority can acquire a synchronization variable even though there may be low priority threads waiting in the synchronization queue. This can lead to a starvation phenomenon, which implies low-priority threads may never acquire a synchronization variable especially when there is high contention for the variable and always at least one high-priority thread waiting for the same variable. When there are multiple threads with the same priority level waiting for a synchronization variable, one of the other scheduling priorities will determine which thread shall acquire the lock.

Conversely, situations can occur when a low priority thread owns a lock on which a higher priority thread is blocked. If the lower priority thread is itself blocked on a different lock that must be released by yet another higher priority thread, then the lower priority thread may never get scheduled to release the lock to the blocked higher priority thread. Although a symptom of bad usage of locks, this situation, termed priority inversion, can and does occur. Most operating systems address this issue by allowing the priority of the thread that owns the lock to be raised to at least the priority of the highest priority thread blocked on the lock. In this fashion, the currently running thread will eventually get access to the CPU and relinquish the lock.

3.7 DEADLOCK, STARVATION AND PRIORITY FAILURE

Deadlock can occur when two or more threads are each blocked, waiting for conditions to occur that only the other ones can cause. Since each is waiting on the other, neither will be able to continue. A deadlock can happen when a thread needs to acquire multiple locks. For example thread T1 holds resource R1 and tries to acquire resource R2. At the same time, thread T2 is holding R2 and trying to acquire R1. Neither thread can make progress.

Starvation is the situation in which a thread is prevented from making sufficient progress in its work during a given time interval because other threads own the resource it requires. This can easily occur when a high priority thread prevents a low priority thread from running on the CPU, or one thread that always win over another when acquiring a lock are examples of starvations.

Priority inversion is a scenario that occurs when a high priority thread attempts to acquire a lock that is held by a lower priority thread. This causes the execution of the high priority thread to be blocked until the low priority thread has released the lock, effectively inverting the relative priorities of the two threads. If other threads with medium level priorities attempt to run in the interim, they will take precedence over both threads. The delayed execution of the high priority thread (after the low priority thread release the lock) normally goes unnoticed and causes no harm. However on some occasions, priority inversions can cause problems especially in a real time system. If the high priority thread is deprived of a resource long enough, it may lead to a system malfunction or triggering of corrective measure such as watchdog timer resetting the whole system. The priority inversion can also causes threads to execute in such sequence that the required work is not performed in time to be useful anymore. POSIX defines the two standard mechanisms to avoid priority inversion: priority inheritance and priority ceiling protocols. Priority inheritance allows a low priority thread inherits priority of a high priority thread, thus preventing medium priority threads from preempting the low priority thread. Priority ceiling is a procedure that assigns the thread that possesses a lock with high or ceiling priority. This works well as long as other threads do not possess priority levels higher than the ceiling level priority.

3.8 POSIX THREAD LIBRARY:

Thread Management

Pthreads contains a runtime library to manage threads in a transparent way to the users. The package includes calls for thread management, scheduling and synchronization. The thread management APIs are given below:

int pthread_create(thread_t id, void *(*start_function) (int), int argumnet, int priority)

- Create a thread to execute a specific function

void pthread_exit(void *value_ptr)

- Causes the calling thread to terminate without causing entire process to exit

int pthread_join(thread_t id, void **value_ptr)

- Causes the calling thread to wait for the specified thread (thread id) to exit

pthread_self( ) - return caller’s identity or thread ID int pthread_yield( )

- Threads can voluntarily release CPU to let other threads run by calling thread_yield.

Threads can be dynamically created and terminated during the execution of a program. However the total of number threads is subject to the resource limitations of each given system. For example the number of threads can be limited by the scheduler queue size. Threads are created dynamically with the *thread_create* API. The *thread_create* reserves and initializes a thread control table and adds a thread ID into the scheduler queue. The *start_function* is the name of a function or routine that the thread calls when it begins execution. The *start_function* takes a single parameter specified by the argument. The start routine returns a pointer (pointer of type void), which later to be used for an exit status by the *thread_join*.

Threads exit in two ways. First, by returning from the thread function (implicit exit). For this implicit exit, the return value from the function is passed back to the parent thread as the return value. Alternatively, a thread can explicitly exit by calling thread *thread_exit*. The argument to the *thread_exit* is the thread return value. The *value_ptr* parameter value is available to its parent *thread_join*.

A parent thread *uses thread_join* to wait for all its children to terminate, before it can exit itself. This will avoid de-allocating of data structures that its children may still require. The *thread_join* API takes two arguments, the thread ID of thread to wait for and a pointer to a void* variable that will receive the finished threads' return value.

4 HYBRID THREAD

4.1 INTRODUCTION

General programming models form the definition of software components and governing interactions between the components [14]. Achieving abstract programming capabilities across the FPGA/CPU boundary requires adaptation of a high-level programming model that abstracts the FPGA and CPU components, bus structure, memory, and low-level peripheral protocol into a transparent computational platform [3]. The KU hybrid threads project has chosen a multithreaded framework as our model, supporting concurrent hybrid threads distributed flexibly across the systems CPU and FPGA assets. Within our model, all threads adhere to the policies of accepted shared memory synchronization protocols for exchanging data and synchronizing control. To support this generalized model across the FPGA, we have developed a Hardware Thread Interface that encapsulates mechanisms to support synchronization for FPGA based threads. Under this unified model, application programmers perform procedure calls from within both VHDL and C to access high level synchronization primitives between threads running across both hardware and software computations. The set of Hardware Thread Interface components as well as a standard software interface component form our system-level hybrid thread abstraction layer as shown in Figure 4-1.

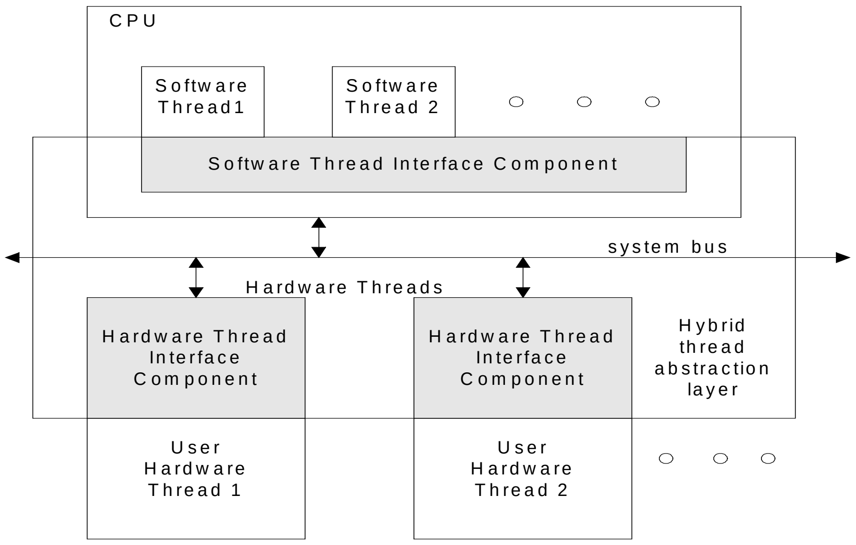

CPU
Software Thread1
Software Thread 2
Software Thread Interface Component
system bus
Hardware Threads
Hardware Thread Interface Component
Hardware Thread Interface Component
Hybrid thread abstraction layer
User Hardware Thread 1
User Hardware Thread 2

Figure 4-1 Hybrid Thread Abstraction Layer

4.2 HYBRID THREAD ABSTRACTION LAYER

Concurrent execution of FPGA based threads and CPU based threads as given in Figure 4-2 illustrates the low level services provided within the hybrid thread abstraction layer. As shown in Figure 4-2, thread F3, an FPGA based thread and T11, CPU based thread, can communicate and synchronize with thread concurrency control operations. The concurrency control mechanisms include blocking and queue facilities enable threads to block other threads. Threads T8, T4, F2 and T9 enter blocked states when they attempt to acquire MUTEX N, already locked by another thread. CPU threads T2, T5, T12 are in a *runnable* state waiting to be scheduled on CPU. Within the abstraction layer the following services are supported:

- Fundamental atomic operations that form the basis of higher order concurrency protocols. These "atomic operations" are CPU family independent, and are able to handle multiprocessor environments.
- Uniform concurrency mechanisms that able to enforce mutual exclusion on shared resources accessed by both CPU based software threads and FPGA based hardware threads. These control primitives are enabling mechanisms that elevate concurrency higher than low-level processor instructions.
- Blocking synchronization mechanisms that include queues and wake-up services for concurrency control functions. The queuing operation and wake-up services operate autonomously and require no overhead from the operating system running on the CPU. In addition, the wake-up facility includes delivery of wake-up threads to the scheduler queue. The performance of the queuing and wake-up facilities is independent of the number of blocked threads in the queues.
- Synchronization mechanisms that enable for mutual exclusion, event waiting, controlling countable shared resources and arbitrary blocking conditions in order to support assorted applications. Also synchronization mechanisms that avoid lost wake-up and "thundering herd" problems.
- A hardware thread support layer that acts as an interpreter for user computations to request synchronization and access the system memory. Within this layer, a scheduler has also been created to manage hardware threads with no overhead required on the CPU. Thread management operations are also provided to software programs

- Application programming interfaces (APIs) are presented to the threads on the CPU to access the new concurrency control mechanisms. The APIs have two layers: the high level layer which provides an interface for application programs, and a low level layer that provides an interface for the operating system.

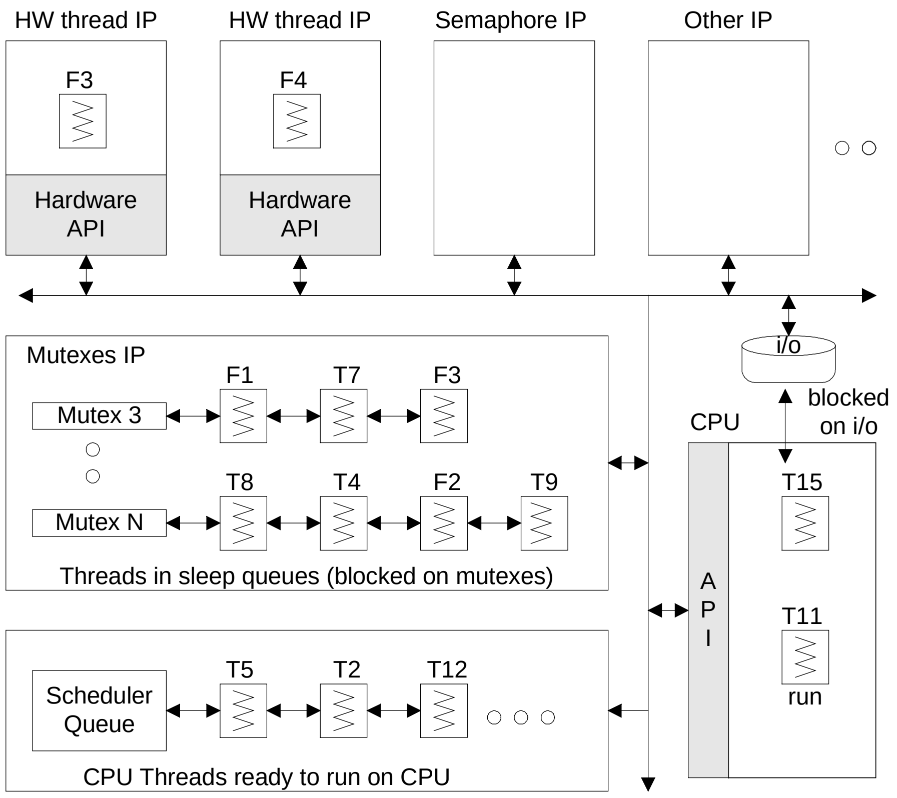


Figure 4-2 Hybrid Threads System

The Hardware Thread Interface (HTI) component is provided as a library for inclusion within user-defined hardware threads. The hybrid thread abstraction layer implements all interactions between source hardware threads and other system components through the command and status register pair. This capability is particularly useful for debugging and is used during runtime to interact with other system components, such as our semaphore mechanisms for thread blocking and wake-up. During debug, the status register is accessible from user programs allowing developers to monitor and control the execution state of a hardware thread.

4.3 HARDWARE THREAD

4.3.1 Introduction

Some applications naturally decompose into as collection of distinct tasks that can be executed concurrently. Traditionally these concurrent tasks are implemented as processes or threads executing either on a single or multiple processors. A thread can be thought as independently executable sequence of instructions. For software threads, these instructions are stored in the system memory and are available to be executed by multiple invocations of a common thread.
To extend the multithreaded programming model across the processor/FPGA boundary, we must first understand the operational semantics of a hardware thread. First, unlike software threads, a hardware threads do not share a common code base. Instead, each hardware thread is a unique physical parallel hardware compute engine built within the configurable logic fabric. The execution sequence of the thread is based on a finite state machine and not modeled on the von Neumann stored program architecture. As hardware threads are independent executable entities, we can capitalize on the potential of the FPGA to support true physical concurrency. This view exposes the FPGA to wider pools of system designers. Other incentives includes:

- Reduce development times in implementing applications in FPGA
- Offering new insights and techniques toward a more unified platform of hardware and software co-design and presenting a more integrated view of processor and FPGA components.
- Provides faster and more deterministic system response times through the co-designed system services.

To promote portability and platform independence of the user computation from the underlying platform, we provide the Hardware Thread Interface (HWTI) shown in Figure 4-3.

To enable these hardware-implemented computations, new services such as synchronization, thread management and other operational support must be created. In addition, other services such input and output infrastructure are necessary to enable user computations to share data with other threads across system bus. The collection of these new services and necessary support hardware forms the hardware thread interface. The support hardware includes register sets, architecture dependent and independent system bus interface logic, and hardware controllers. The services are packaged in the form of a standard application program interface (API). User computations use these API's to request services from the interface. The interface responds to requests and provides feedback to the user computation in form of a return status.

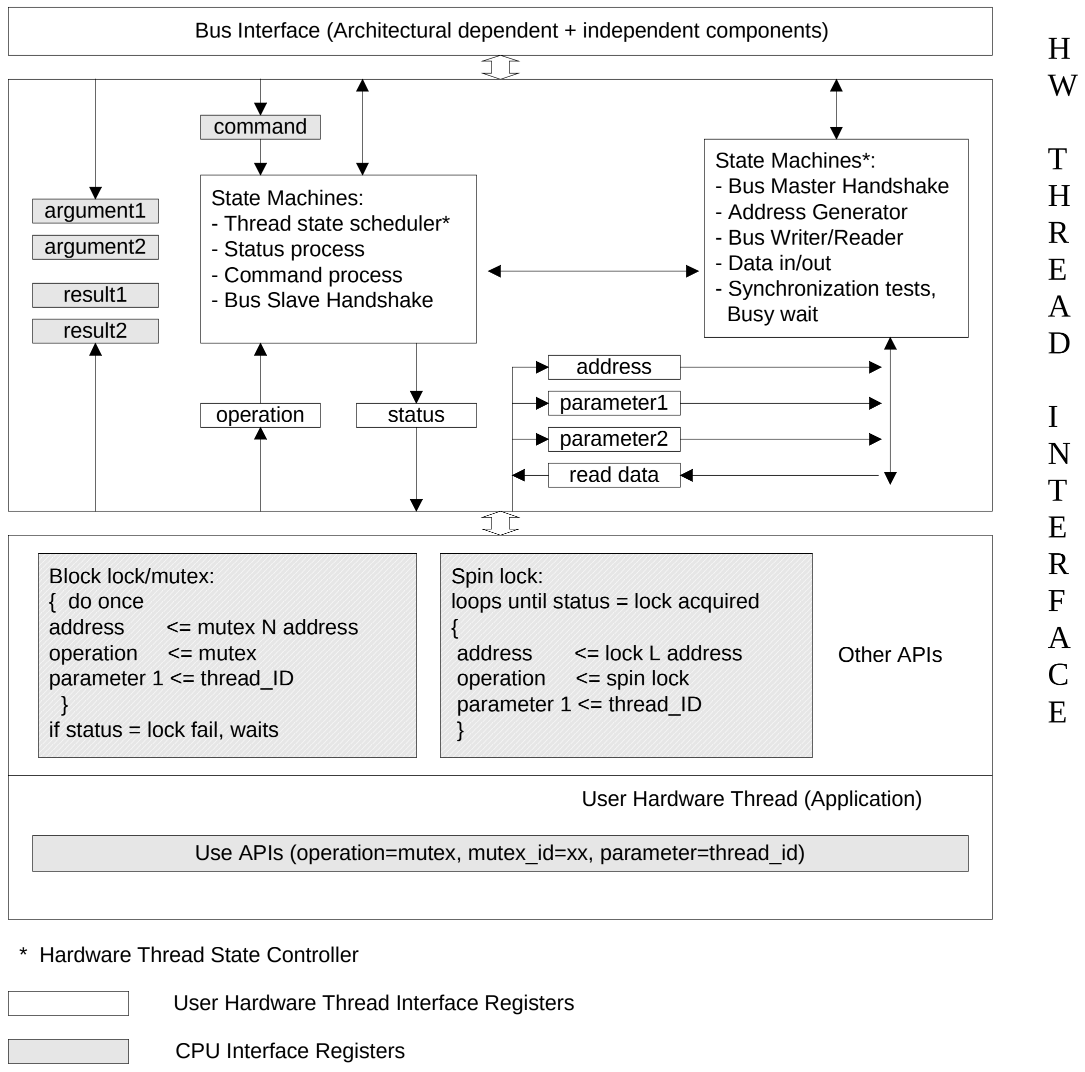


Figure 4-3 Hardware Thread Interface

At this time both the user-specified and the interface components are synthesized and loaded into the FPGA before run time. However, this interface has been designed to serve as the common interface for dynamic reconfiguration of the FPGA.

4.3.2 Hardware Thread Interface

The Hardware thread interface (HWTI) is placed between the system bus and the user thread. The HTI acts as interpreter that translates user thread requests to sequences of actions and provides feedback when the actions are complete. In the opposite direction, it responds to commands issued by the CPU and other entities on the system bus. One new requirement for supporting hardware threads when compared to existing approaches is the ability of a thread to write in addition to read from memory mapped locations. The bus interface supports both bus slave and master modes. As a bus master, bus transfers can be requested on behalf of the user thread.

To serve user computations efficiently, we divide the Hardware Thread Interface component into three subcomponents according to their distinct functions: a CPU interface, a hardware-thread state controller, and a user hardware-computation interface. The user interface defines a set of platform independent functions implemented within the library API's. The API uses set of interface registers to request services from the thread state controller. The thread state controller accepts the user requests, performs the necessary operations and returns both the status and the requested information. The command register defined within the HWTI CPU interface enables the system CPU, or any other bus controller, to interact with the user hardware thread.

HWTI - CPU Interface

The CPU interface contains the bus interface, the command/status register pair, argument and result registers. These registers collectively enable the CPU and other system bus entities to control the execution of the hardware thread. These registers are memory mapped components instantiated with the hardware thread HWTI.

This interface provides an application program running on the CPU to start or stop a hardware thread's execution. Writing a command into the command register is analogous to the software thread create and exit operations. During run time, this facility enables other system bus entities such as semaphores to wake up a sleeping thread. This capability is also particularly useful for debugging purposes. A CPU based application thread may also inspect the status register to determine the state of the hardware thread's current execution.

We provide an application program interface (API) library function similar to current software thread packages for thread creation and deletion. In software based multithreading packages, the scheduler does not guarantee immediate execution of a newly created thread. Instead, the API contains a call to the scheduler that may or may not immediately run the new thread. In contrast to this approach, a hardware thread is an independent computational component and will be executed immediately upon creation. The API to create the hardware thread is shown in Figure 4-4. To initiates hardware thread execution, the API writes of a unique op_code into the thread's command register.

```
hw_thread_addr    =  hw_thread_base + (thread_id * 256);
#define cmd_reg_offset    0x05;
#define arg1_reg_offset    0x07;
#define arg2_reg_offset    0x07;
#define cmd_run              0x03;
        hw_thread_start(thread_id, arg1, arg 2) {
                *(hw_thread_addr + arg1_reg_offset) = arg1;
                *(hw_thread_addr + arg2_reg_offset) = arg2;
                *(hw_thread_addr + cmd_reg_offset) = cmd_run;
                }
```

Figure 4-4   Pseudo Code for Hardware Thread Create API

HWTI - Thread State Controller

Our approach for controlling hardware threads is to provide the state machine template shown in Figure 4-5. Threads that have not yet started or have terminated are in the idle state. Threads that are currently in the run state enter the wait state when a thread requests, or continues to be blocked on, a semaphore. Threads request a semaphore by executing our library API's discussed in Chapter 5. Our API's implement a semaphore request by writing the address of the semaphore into an address register, and writing a unique op_code into the operation register. When the request is made, the controlling state machine transitions the thread from the running to the wait state. The status of this, or any operation, is available to the thread in the status register. As long as a thread remains in the wait state, no further execution of the user thread is allowed. A thread in the run state releases a semaphore by executing a release semaphore library API. The implementation of the release API is similar to the request API, using semaphore address and operation registers.

Our state machine supports both spin lock and blocking semaphores for hardware threads. For hardware threads task switching on a blocking semaphore is not required, as the thread does not contain shared resources that should be rescheduled. Instead, the dedicated hardware thread simply idles in the same fashion as it would with a spin lock. The hardware thread interface component however does perform different processing for spinning and blocking semaphores.

For the spinning semaphore, the hardware thread interface transitions the thread state into the wait state, issues a single request for the semaphore across the bus, and returns the thread to running when the status of the request is returned into the status register.

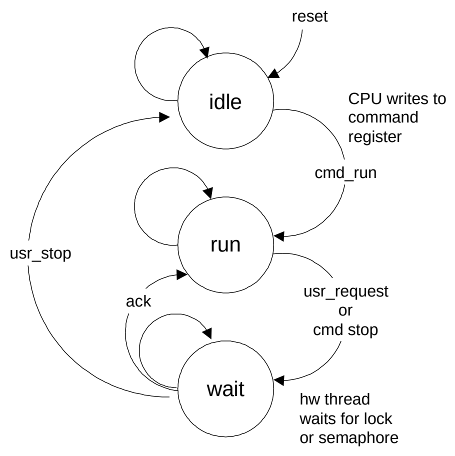


Figure 4-5 Hardware Thread States

In contrast, for a blocking semaphore the hardware thread interface transitions the thread to the wait state, issues the request for the blocking semaphore and leaves the thread in the wait state if the semaphore is in use. Upon a grant or release the state machine will then transition the thread back into the run state. The semaphore IP's discussed in Chapter 5 details the differences in the semaphore logic for the two types of semaphores. The thread scheduler manages the transitioning between states during the lifetime of the thread. The states are visible via the status register to all other threads. User computations are treated as autonomous threads and can be stopped by the controller at selected points of execution. This implies that a user computation may decide when it is appropriate to check the status register and control it's own operation. This allows independent hardware threads to be stopped by a controlling CPU based thread to load new operating information or configuration.

4.4 HARDWARE THREAD INTERFACE ARCHITECTURE

The diagram of hardware thread interface RTL representation is given in Figure 4-6. The hardware thread interface can be decomposed into the following hardware components:

- Architecture dependent bus interface
- Architecture independent bus interface
- Thread Scheduler
- Bus master controller
- Register processes

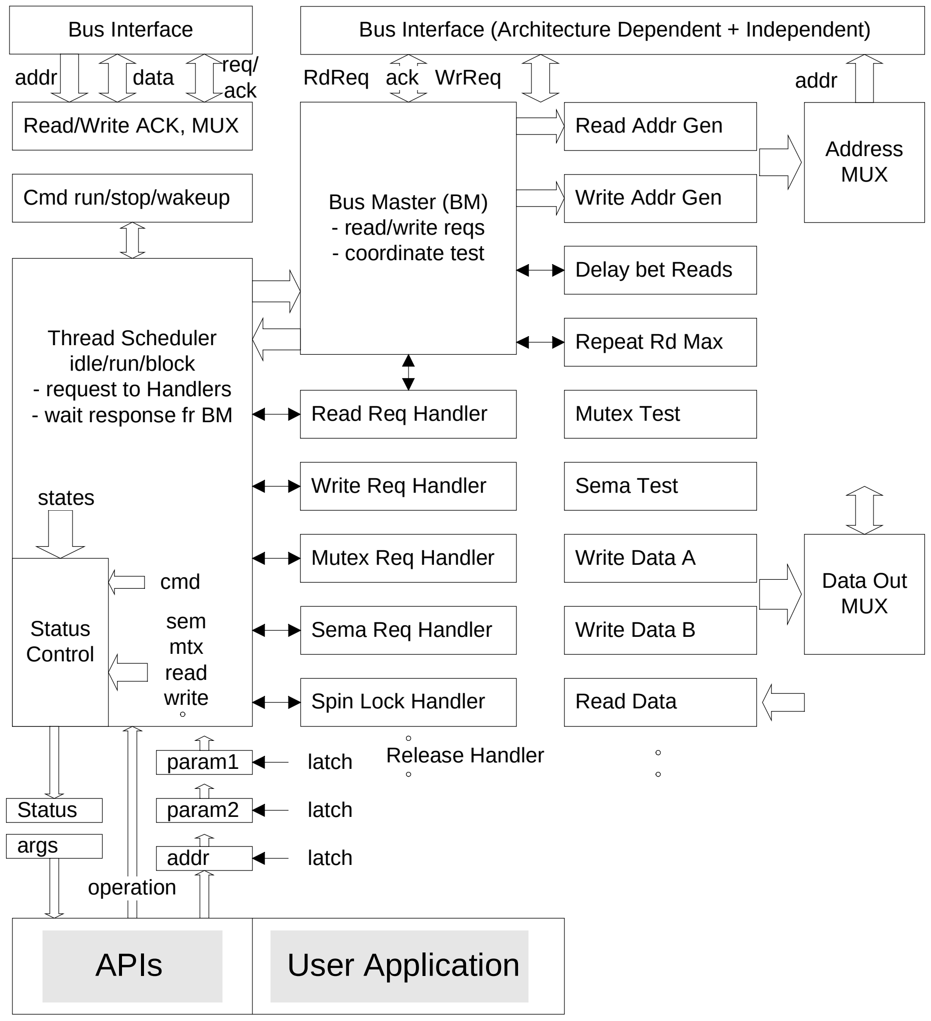


Figure 4-6 Hardware Thread Hardware RTL representation

The system bus components that can instantiated within the hardware thread interface is: processor local bus (PLB) or peripheral or off-processor bus (OPB). These buses support facilities that include communication exchange protocols and bus arbitration. A standard set of signal interface (IPIF) is used on the IP core side to enable attachment of different IP modules to either bus. This IPIF interface facilitates migration of core modules including the hardware thread from one system bus to another.

The kind of services that IPIF provides include address decoding, data buffering, interface registers, and peripheral interrupts. The details technical specification these system busses can be found in IBM and XILINX datasheets [50,61].

Instead of creating a new interface, we utilize the IPIF as a part of our hardware thread interface to serve as a layer between the system bus and architecture independent bus interface component. In implementing the architecture independent interface we select bus slave and bus master services common in most microprocessor-based system. Adopting this approach enables users to replace only the architecture dependent interface portion when porting hardware threads to other hardware platforms.

The collection of interface, thread scheduler, bus mastering and register processes can be grouped into two distinct functions: a control unit to manage user thread execution and data path for data flow in and out of the user thread. The control unit is broken into several hierarchical processes or state machines as shown in Figure 4-7. The hierarchical state machines are identified as A, B and C as shown in the Figure 4-7. Each hierarchy state machine is further decomposed into several concurrent state machines or processes as indicated in the diagram. Hierarchical state machine A is made up of thread scheduler state machine (A1), status register process (A2), and so on. B1 represents group of request handler processes and B2 is the bus mastering state machine that provides input/output access services across the system bus and synchronization ownership tests. The interface procedure (API) in hierarchy process U generates appropriate address and operation codes for the thread scheduler upon a request from the user program. For a semaphore request, the API calculates the semaphore variable address from the semaphore base address and the semaphore ID.

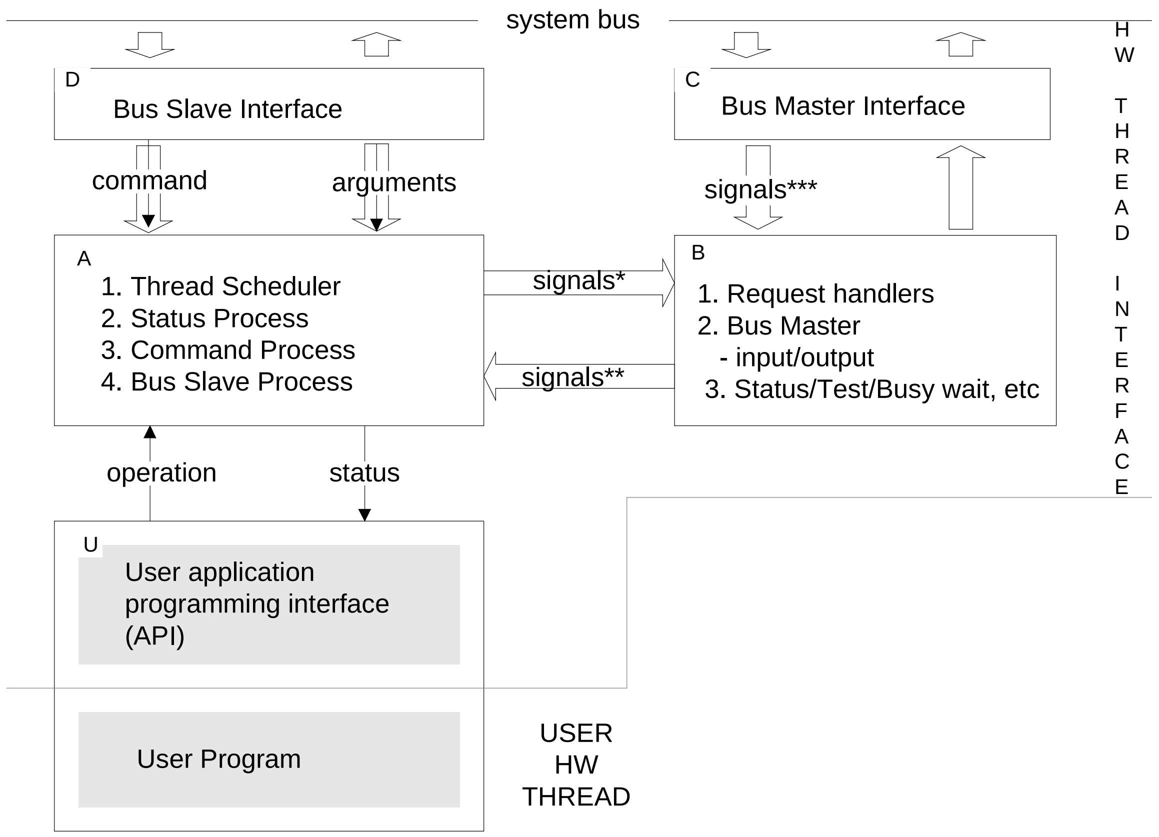


Figure 4-7 Hardware Thread Hierarchical Processes

The API then writes an appropriate code into the operation register. The operation code can be one of the different types of synchronizations requests or data input/output transactions. The specific hardware procedures available to a user thread are discussed in next section. The user program then waits for status register to be updated by the scheduler. The scheduler then transitions from the run state to one of the waiting states determined by the operation register content. The state machine of the thread scheduler is given in Figure 4-8. As shown in the state machine diagram in the Figure 4-8, there are five different possible execution paths the scheduler must follow depending on the request made by the user program. The execution path can be for a spin lock, block lock, semaphore wait, read or write transaction. The write transaction is not only for sending data out but also for spin un-lock, mutex un-lock or semaphore post operation. It also needs to switch the bus from bus slave to the bus master, since CPU read and bus master write transaction uses the same data lines. When the scheduler moves from the run state to the wait state, it issues a signal to latch the address and the parameter registers (the user interface registers).

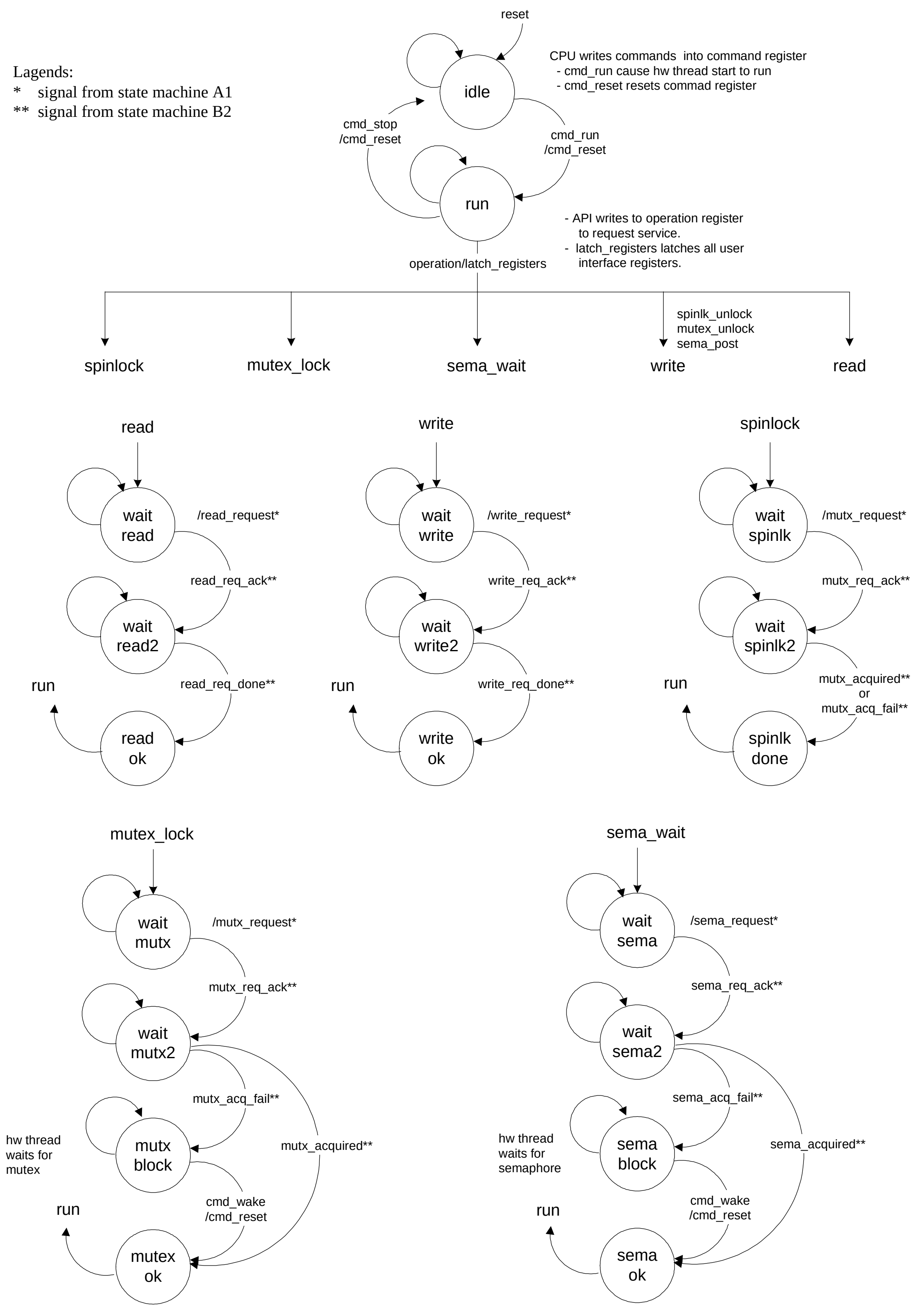


Figure 4-8 Thread Scheduler State Machine

Then it generates a request signal to the request handler state machine B1, which in turn causes bus master B2 to perform the intended operation on behalf of the scheduler.

The state machines for different request handlers are given in Figure 4-9. The task of the request handlers is to deliver the scheduler request only when the bus master is in the idle state. This approach permits concurrent processes and ensures that the request will not be missed. Then it sends an acknowledgement to the scheduler to indicate the request is in progress and for the scheduler to de-assert it’s request and move to another waiting state. Request handler B1 not only ensures the request will not be missed but also prevents the bus master from repeating a requested task when it returns back to idle state once it has completed its duty. The state machine diagram for the bus-master controller is given in Figure 4-9. Depending on the type of responses it receives from the bus master, the scheduler returns to the run state or goes into blocking state. The scheduler remains in the block state until a synchronization core sends a wake-up command to it via the command register. Upon receiving the wake-up command, the scheduler resets the command register to the init state.

The responsibility of the bus-master controller is to accept different bus transaction requests from one the request handlers, and generate read or write request signals to the bus master interface. If a request is for a semaphore acquisition then it also must perform a synchronization ownership test. As shown in Figure 4-9, there are four possible paths the bus master will follow depending on which request handler it receives signal from. If the request is for a semaphore, the bus master carries out a read operation to fetch the requested semaphore variable, and proceeds to perform synchronization test to determine whether it has ownership of the requested semaphore.

For the spin lock and binary blocking lock operations, the read data is compared with the thread ID. If the read data is the same as the thread ID, the lock is obtained. For the semaphore wait operation the read data is compared with zero. If they are equal the semaphore is exhausted. It then sends an appropriate acknowledgement (semaphore fail or success) to the scheduler.

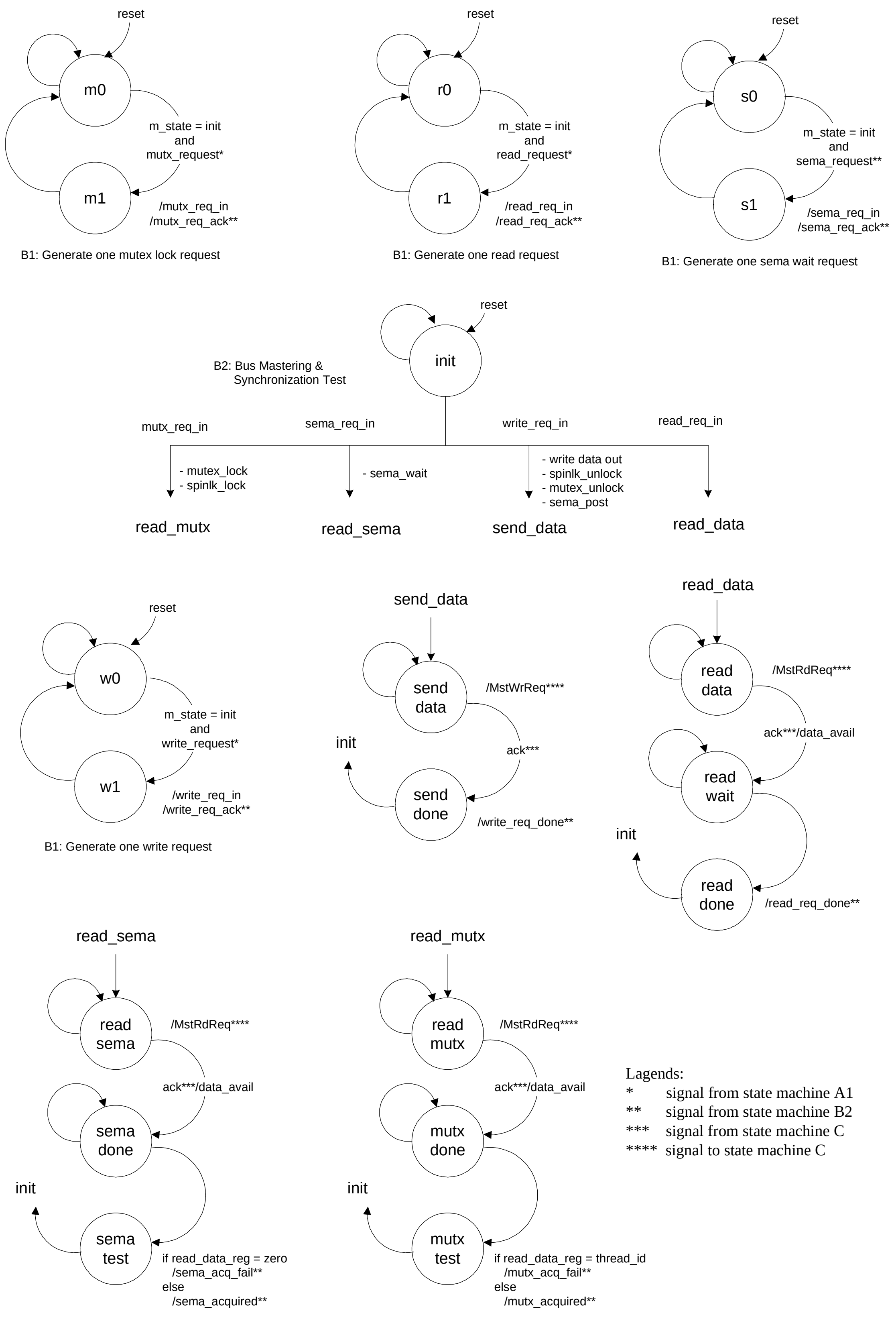

reset
m0
m_state = init and mutx_request*
m1
/mutx_req_in /mutx_req_ack**
B1: Generate one mutex lock request
reset
r0
m_state = init and read_request*
r1
/read_req_in /read_req_ack**
B1: Generate one read request
reset
s0
m_state = init and sema_request**
s1
/sema_req_in /sema_req_ack**
B1: Generate one sema wait request
reset
init
B2: Bus Mastering & Synchronization Test
mutx_req_in
sema_req_in
write_req_in
read_req_in
- mutex_lock
- spinlk_lock
- sema_wait
- write data out
- spinlk_unlock
- mutex_unlock
- sema_post
read_mutx
read_sema
send_data
read_data
reset
w0
m_state = init and write_request*
w1
/write_req_in /write_req_ack**
B1: Generate one write request
send_data
send data
/MstWrReq****
ack***
send done
/write_req_done**
init
read_data
read data
/MstRdReq****
ack***/data_avail
read wait
read done
/read_req_done**
init
read_sema
read sema
/MstRdReq****
ack***/data_avail
sema done
sema test
init
if read_data_reg = zero /sema_acq_fail** else /sema_acquired**
read_mutx
read mutx
/MstRdReq****
ack***/data_avail
mutx done
mutx test
init
if read_data_reg = thread_id /mutx_acq_fail** else /mutx_acquired**
Lagends:
* signal from state machine A1
** signal from state machine B2
*** signal from state machine C
**** signal to state machine C

Figure 4-9 Bus Master State Machine

As mentioned in the previous section, the status register is necessary to represent the execution state of the hardware thread and the status of user request. Thus some of the registers bits are used to represent the state of hardware, while other bits are used to indicate the status of a user request. An additional requirement is that the status must be in a format that is meaningful to the user program rather in terms of low-level hardware details. Thus a simple technique of copying the hardware thread states into the status register will prove to be inadequate and not user friendly. We take an approach that a single process (A2) is dedicated to update the status register with all required information provided as inputs to this process as shown in Figure 4-10. The status register bits however do not represent every possibility of input combinations, but instead reflect useful states and conditions meaningful to the user thread.

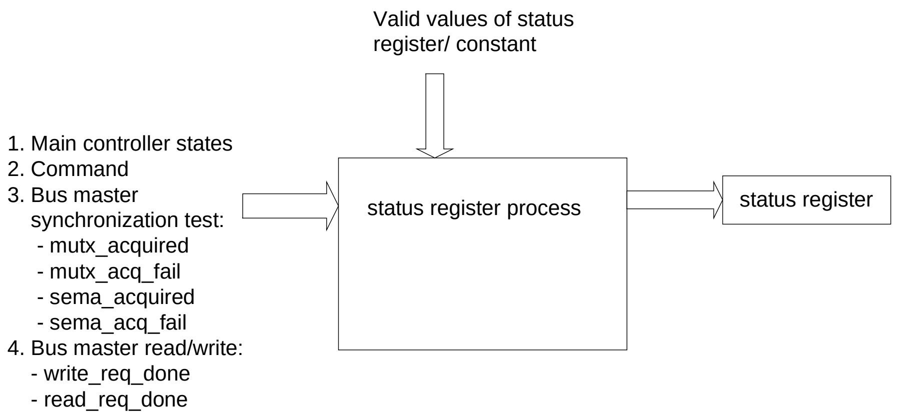


Figure 4-10   Status Register Process

As mentioned previously, the command register is provided to control the operation of the hardware thread from the system bus. Process A3 is responsible for updating the command register and deliver commands to the thread scheduler. This process changes this register state in response to system resets, CPU commands, or acknowledgements from the thread scheduler. The valid states of this register are INIT, IDLE, RUN, and WAKE-UP. It goes into the initial state following a system reset. Either the CPU or the semaphore IP core writing a command codes into this register causes it’s a state change into a RUN, IDLE or WAKE-UP state. The thread scheduler detects this register state change, and responds by sending an acknowledgement back to the process. The process in turn resets this register back to initial state.

4.5 HARDWARE THREAD USER INTERFACE

A combination of register sets and hardware-implemented procedures represent this sub-component. This interface provides hardware threads with standard procedures for communicating with other entities or cores such as semaphores across the system bus. The set of registers in this interface are the operation, status, address, parameters, and read data registers. The operation and status register pair serves as the interface for coordinating between the user thread control unit and the thread scheduler. In addition to the control unit, address, read data, and parameter registers are provided as data paths for exchanging user data across the bus. These registers also provide storage and buffering of temporary data for the user computation. Temporary data includes data brought in from memory, and data coming from user hardware thread going out to other cores connected to the bus.

The set of functions or hardware procedures callable by the user thread are provided in form of standardized program interface. The programming tasks that to be performed by the user are significantly reduced by means of this standardized program interface (API).

Pseudo API	Opcode	Return Code
Read_data( )	READ	READ_OK
Write_data( )	WRITE	WRITE_OK
Sem_post( )	WRITE	SEM_POST_OK
Sem_wait( )	SEM_WAIT	SEM_WAIT_OK
Mutex_lock	MUTEX_LOCK	MUTEX_LOCK_OK
Mutex_Unlock	WRITE	MUTEX_UNLOCK_OK
Spin_lock( )	SPIN_LOCK	SPIN_LOCK_OK
Spin unlock( )	WRITE	SPIN_UNLOCK_OK

Table 4-1 Hardware Thread Application Program Interfaces (APIs)

The control protocol for all API's includes writing a unique code into the operation register and reading the status register. The set of operations provided by the API, the operation code and the return code expected from the status register is shown in table 4-1. The implementation details of the application interface are described later in section 4.5.2

4.5.1 Interface Registers

The functionality of each register in the hardware thread interface are summarized below:

a) Command register

- It is a CPU writable register, for starting/stopping execution of the hardware thread. If the thread is already in the run state, the user has to decide when to check the command register in order to stop at safe or allowable states.
- Semaphore controller writes wake-up code into this register to unblock the hardware thread from the blocking condition when a synchronization variable becomes available.

b) Status register

- This register represents the current execution state of the hardware thread and the status of the user request. Therefore, it is updated by the thread controller and is accessible from the user programs. The user programs or computations wait or proceed to a next sequence depending the status return by this register.

c) Argument registers

- These registers are used to hold arguments. A program running on a CPU or a hardware thread writes arguments such as addresses of shared data or other information to these registers before commanding a hardware thread to start execution. For example a software program writes pointers into these register after memory allocation operation.
- User program read these register either to determine the algorithm it has to execute or to obtain addresses of shared data in memory it can access to. For example user program may execute different image processing algorithm depending on values written into one of these registers.

d) Result registers

- User program has option to use these registers to store temporary result of computation and write it out to memory. In addition, programs in other entities may read these registers to get the results. User program can signal other threads that its computation has completed by using synchronization primitives.

e) Operation register

- User program writes operation codes to this register to request different services from the thread controller or scheduler. User program can request for synchronization or memory accesses. User program then read status register to determine it's next step. If the user program requests for a semaphore and no more semaphore available, the thread goes to blocking state and the state is returned by the status register.

f) Address register

- API writes an address of memory or synchronization variable that it needs to communicate with. To write or read data, user program needs to provide actual address of data to be sent out to or to read from. However to request for a semaphore, user program needs to specify the semaphore ID instead of the semaphore address. The API calculates the semaphore address from the semaphore ID and semaphore base address, and writes appropriate semaphore addresses into this register.

g) Parameter/output registers

- These registers used by the user program API for multiple purposes. It writes data that need to be transferred to memory into one of them. In addition it can be use to hold other parameters such as thread id and the number of semaphore to request.

h) Input register

- These registers provide temporary storage for read data. User program API reads this register after the controller has fetched data from the memory or device connected on the system bus. User program knows that data is already available in this register when status register returns success status on its read request.

4.5.2 Hardware Thread Application Program Interfaces

The following are partial portion of application program interfaces (APIs) provided within the hardware thread interface to allow user thread to request for different kind of synchronization operations and memory accesses. Synchronization APIs make use of base addresses of synchronization cores that are passed as generics to calculate location of each variable. An example how to use one of these APIs is as follows: sem_wait(sem ID).

```
procedure write_data
     ( signal  address       :  out  std_logic_vector(0 to addrbus_width);
       signal  addr          :   in    std_logic_vector(0 to addrbus_width);
       signal  opcode        :  out   std_logic_vector(0 to 3);
       signal  param         :  out std_logic_vector(0 to databus_width);
       signal   data         :   in   std_logic_vector(0 to databus_width)) is
   begin
       address <= addr;
       opcode <= WRITE;
       param  <= data;
   end procedure write_data;

procedure read_data
     ( signal address        :  out  std_logic_vector(0 to addrbus_width);
       signal addr           :  in    std_logic_vector(0 to  addrbus_width);
       signal  opcode        :  out   std_logic_vector(0 to 3) ) is
     begin
       address <= addr;
       opcode <= READ;
   end procedure read_data;

procedure sem_wait
     ( signal address        :  out  std_logic_vector(0 to addrbus_width);
       constant sema_id   :  in    std_logic_vector(0 to 5);
       signal     opcode     :  out   std_logic_vector(0 to 3) ) is
   begin
       address <= sem_baseaddr(0 to 12) & sema_wait & hw_thread_id & sema_id & “00”;
       opcode <= SEM_WAIT;
   end procedure sem_wait;

procedure sem_post
     ( signal address         :  out  std_logic_vector(0 to addrbus_width);
       constant sema_id    :  in    std_logic_vector(0 to 5);
```

```
      signal     opcode      :  out   std_logic_vector(0 to 3);
      signal     param        :   out  std_logic_vector(0 to databus_width) ) is
   begin
      address <= sema_baseaddr(0 to 12) & sema_post & hw_thread_id & sema_id & “00”;
      opcode <= WRITE;
      param  <= x”00000” & “000” & hw_thread_id;
   end procedure sema_post;

procedure spinlock_lock
     ( signal address              :  out  std_logic_vector(0 to addrbus_width);
      constant spinlock_id    :  in    std_logic_vector(0 to 5);
      signal     opcode           :  out   std_logic_vector(0 to 3) ) is
   begin
      address <= spinlock_baseaddr(0 to 12) & sp_lock & hw_thread_id & spinlock_id & “00”;
      opcode <= SPIN_LOCK;
   end procedure spinlock_lock;

procedure spinlock_unlock
     ( signal address              :  out  std_logic_vector(0 to addrbus_width);
      constant spinlock_id    :  in    std_logic_vector(0 to 5);
      signal     opcode           :  out   std_logic_vector(0 to 3);
      signal     param            :   out  std_logic_vector(0 to databus_width) ) is
   begin
      address <= spinlock_baseaddr(0 to 12) & sp_unlock & hw_thread_id & spinlock_id & “00”;
      opcode <= WRITE;
      param  <= x”00000” & “000” & hw_thread_id;
   end procedure spinlock_unlock;
```

5 HYBRID THREAD SYNCHRONIZATION

5.1 INTRODUCTION – ATOMIC OPERATION

Management of shared resources is essential to the implementing a multithreaded programming model. Current multithreaded programming models use synchronization mechanisms such as binary semaphores to enforce mutual exclusion on shared data to avoid concurrent access by multiple threads. Thus, providing atomic operations are fundamental to achieving efficient semaphore semantics. Semaphores are implemented on general purpose CPU's with atomic read and (conditional) write pair operations such as the load linked and store conditional, test and set, and test-swap instructions [39]. For example MIPS R4000 and Digital's Alpha AXP processors use load and store instruction to provide an atomic read-modify-write operation [42]. While semantically correct, these existing mechanisms introduce additional complexity in the system design that is not easily extendable to hardware threads. Instead of replicating these mechanisms, we use the FPGA to implement more efficient mechanisms that are CPU family independent, and require no additional control logic to interface into the system memory coherence protocol for hardware threads. As such, our new mechanisms are easily portable across shared and distributed memory multiprocessor configurations and are also available to CPU/FPGA systems with only software threads.

During the course of this research project, we developed several new methods of achieving atomic operation to implement semaphores in FPGA. Our first attempt was to mimic the classic standard atomic write and read operation pair. We created memory mapped request and/or owner registers and a simple control structure within the FPGA that conditionally accepted or denied the request. The sequence of operations executed by a thread requesting a lock was to write its thread ID into the request register followed by a read operation of an owner register to see if the request had been accepted. The order of these operations was reversed from the classic read first, followed by conditionally write to request and check the success of obtaining the semaphore. We created a second method based on this first attempt that reduced the two instruction pair into a single atomic read bus transaction. This single operation approach was possible by using the address lines to encode the requesting thread ID during a normal bus read operation. The control structure within the semaphore IP then conditionally accepted or denied the request during the single read bus operation.

This approach was then extended to support blocking semaphores that are common for situations where shared resources must be locked for long durations. To enable blocking semaphores, queues are required to hold the thread id's of the sleeping threads. Wake-up mechanisms are also required that select a blocked thread id or multiple id's to be transferred from the blocked queue onto the schedulers ready to run queue. In multi-processor environments, however, spin locks are still appropriate synchronization primitives as their overhead is comparatively lower than a blocking semaphore. This is method of choice when application programs will busy wait for less time than required to perform context switching. Obviously for a single processor environment, spin locks can cause deadlock since the lock owner has no chance to release the lock while the other thread spins. This phenomenon can be avoided if the owner is not preempted or the owner has to release the lock before it goes to sleep.

We have support and have implemented both type spin and blocking locks within the FPGA. In addition to extending the multithreaded programming synchronization capabilities to hardware threads, these new synchronization primitives are also significantly lower overhead for threads running on processors. We achieved lower overhead by:

- Migrating operating system functionality associated with processing semaphore requests and maintaining the semaphore into the FPGA.
- Creating the capability to port other operating system services into the FPGA to further reduce the overhead of operating system services that improve system response times and provide more deterministic delays.

In our first prototypes, we implemented each semaphore with dedicated FPGA resources. As the number of semaphores in most system can be large, this approach did not scale well within FPGA's with limited resources. To minimize the resources requirements, we then created a single control mechanism that managed multiple instances of the semaphores within a single entity. In addition, the semaphore entity was designed to exploit our single read bus transaction approach to achieving atomic operations. This reduced the application program interface (API) to a single read instruction, instead of a write followed by read instruction pair, to request and check the success of obtaining the semaphore. We present the complete evolution of our implementations in order, starting with a simple single semaphore and proceeding through our optimizations leading to our final implementation of our multiple spinning and blocking semaphores that use a single control entity and are accessed with a single atomic read operation.

5.2 SPIN LOCK PROTOTYPE

The block diagram for a spin lock is shown in Figure 5-1. The API pseudo code for accessing the binary spin lock is given in Figure 5-2.

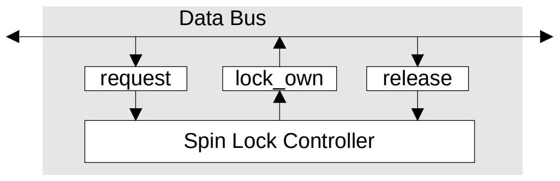


Figure 5-1 One Spin Lock

```
spin_lock_request(&sema, thread_id) {
  grant = 0;
  while(!grant) {
    thread_id → request_reg
    if (lock_owner == thread_id)
      grant = 1;
   else  delay( );  }  }

spin_lock_release(&sema, thread_id) {
  thread_id → release     }
```

Figure 5-2 Spin Lock Pseudo Code API

The semantics for accessing a lock for both hardware and software threads are identical. API's for both hardware and software threads and are made available as library routines to the system developer. To request the semaphore, the API first writes the thread_id into the request register. After the thread_id has been written, the API then reads the lock owner register (lock_own) and compares it with its own thread_id. To release the semaphore, the thread writes its thread_id into the release register. When a thread_id is written into the request register and the semaphore is free, the state machine control logic implemented within the semaphore IP updates the lock owner register. If the semaphore is currently locked, then the control logic performs no update. After the first access, the lock is only freed when a thread writes into the release register.

5.3 SPIN COUNTING SEMAPHORE PROTOTYPE

The block diagram of a spinning counting semaphore is shown in Figure 5-3. The user API pseudo code for accessing this structure is shown in Figure 5-4. The max_cnt register is initialized to the maximum number of resources initially available. The thread first gains access to the counting semaphore request registers by accessing the binary spin lock. The binary spin lock protects the next two instructions that first write the requested number of resources and then reads back a status. A requesting thread writes its request for a number of resources into the req_num register. The semaphore logic then checks to see if sufficient resources are available and appropriately sets the grant register. If insufficient resources are available, then a Boolean value of 0 remains in the grant register. If sufficient resources are available, then the Boolean value 1 is written into the grant register. In either event, the thread reads the result of the request from the grant register. The control logic resets the grant register upon read. The grant flag will stay valid for the request and will not change until the requestor performs the reading of the grant flag. No other requester can cause recalculation to occur as the request/grant check pair are protected by the spin lock. Even if these two sequential operations are interrupted by a valid release request, the grant flag will not be affected and will continue to reflect the value of the count when the requesting task entered into the spin lock. A thread can release any number of resources by writing into the rel_num register. Accessing the spin lock is not required for the resource release, as its operation does not affect the grant register.

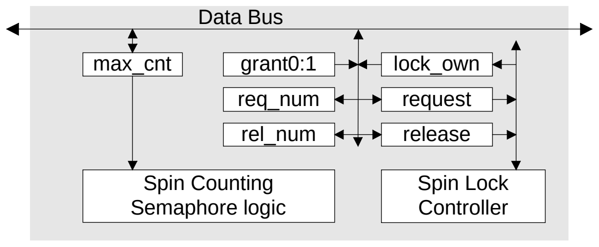


Figure 5-3   Spin Counting Semaphore

```
spin_sema_request(&sema, thread_id, value) {
    grant = 0;
    while (!grant) {
          spin_lock_request( &sema, thread_id)
              value  →  req_num
              grant  ←  grant0:1
          spin_lock_release(&sema)
         if (!grant) delay( );  }  //  end while
     }

  spin_sema_release(&sema, value) {
     value → rel_num
     }
```

Figure 5-4   Spin Counting Semaphore API

5.4 MULTIPLE SPIN LOCKS IP

The block diagram for achieving multiple spin locks using a single controller is shown in Figure 5-5. This single entity provides control for sixty-four spin locks. We use the address lines to encode both the semaphore ID and thread ID during a normal bus read operation. A single control structure within the semaphore IP then conditionally accepts or denies the request during a single read bus operation.

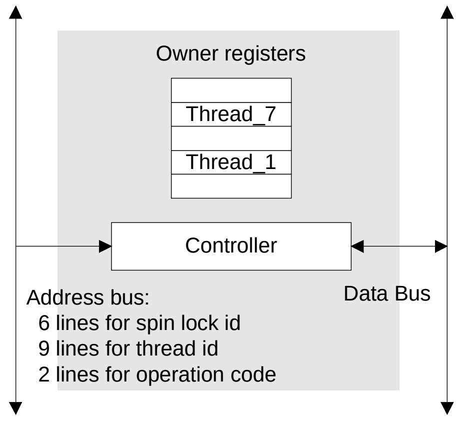


Figure 5-5   Multiple Spin Locks IP

We migrated the multiple owner registers of our previous design into the on chip BRAM thus eliminating the FPGA gates that were used to implement the individual owner registers. This resulted in significant savings of the FPGA's CLBs. The semaphore ID that is encoded within the address is directly decoded to select the semaphore in the BRAM.

To request a semaphore, the API issues a read to an address formed by encoding the semaphore ID and thread ID as the least significant bytes of the base address. In response to this read operation, the spin lock controller decodes the address lines and extracts both the semaphore and thread ID's. The extracted thread ID is then compared with the thread ID stored in the owner register to determine if the semaphore is empty or currently in used. If the owner register is free, it will be updated with the requested thread ID. If the lock is currently locked, then the control logic performs no update. After the check is performed, the controller places the appropriate thread ID from the current owner thread id onto the data bus and terminates the bus cycle. The controller takes eight cycles to complete the request. To release the semaphore, the API writes its thread ID to an appropriate address. The controller state machine then decodes the address lines and updates the selected owner register to non-owner status.

5.4.1 Multiple Spin Lock Hardware Architecture

Our final architecture for multiple spinning binary semaphores consists of: 1) interface and error registers 2) lock owner registers, recursive counters (and its controller) 3) operation mode controllers 4) atomic transaction controller 5) soft reset circuits. Figure 5-6 shows the RTL level description of this architecture. The figure does not include the reset circuit for clarity.

1. Interface and error registers:
    - Lock ID register
    - Thread ID register
    - Error status register (recursive overflow)
    - Data Multiplexer (API return value: error bit and lock owner)

2. Owner registers and its controller
    - 64 Owner registers implemented within BRAM (Lock BRAM)
    - 64 Recursive lock counters (implemented within Lock BRAM also).
    - Lock BRAM access controller

3. Operation mode controller

4. Atomic transaction component

5. Soft reset counters

   Address counter used to reset all the owner registers and recursive counters

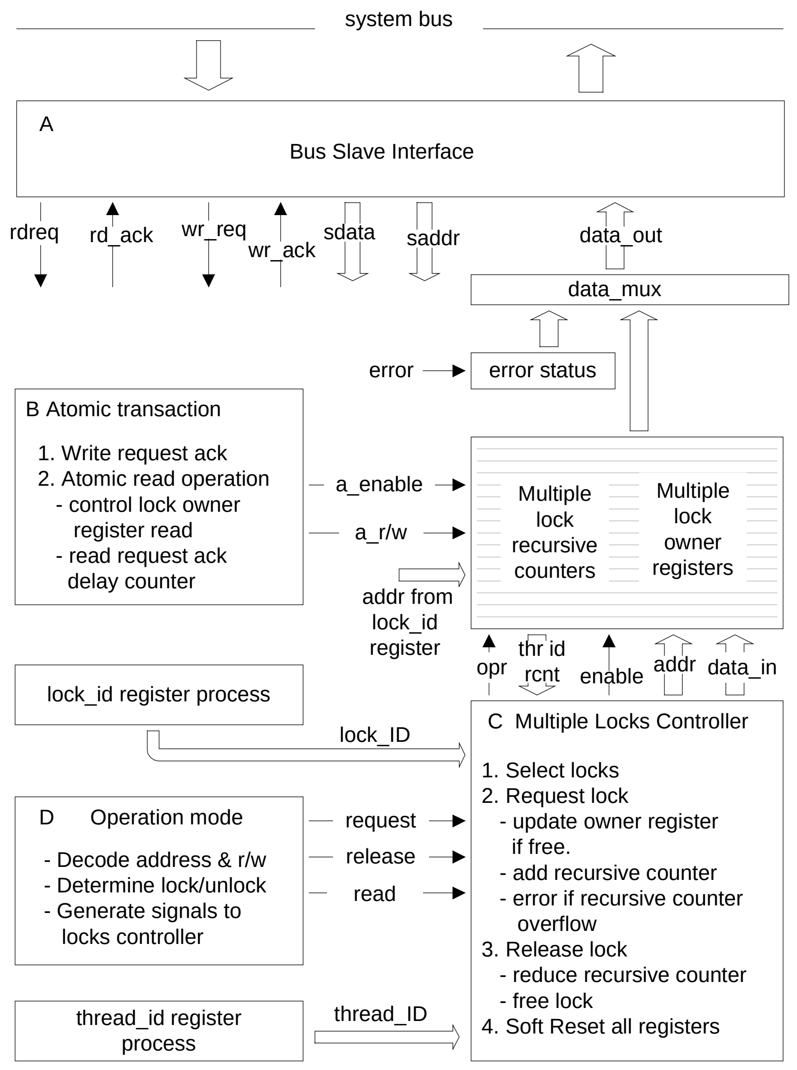


Figure 5-6 Multiple Spin Locks Hardware Architecture

Lock ID register

This register holds a requested lock ID decoded from the address bus. Our application program interface (API) uses address lines A24:A29 to encode the lock ID. The number of address lines used for encoding the id is set during design and based on the number of system locks set by the developer. For this example, sixty-four spin locks are supported within this single entity. The address lines are latched into this register when either the read request or the write request signal goes high. The BRAM Access Controller then uses this register as an index to select one of the sixty-four owner registers within the BRAM.

Thread ID register

The thread ID register latches and holds the thread ID encoded on address lines A23:A14. These nine lines support 512 active threads (256 software threads and 256 hardware threads). The content of this register is then compared with the contents of the appropriate owner register. This register is also used as a transit place to hold a thread ID, before transferring it to one of the owner registers. In addition, this register also encodes a NO OWNER value when a lock is released to free a selected owner register.

Lock Owner registers (Lock BRAM)

The Lock owner registers hold the thread IDs of the threads that currently own the locks.

Recursive counters

The recursive counter register holds the number of grants for an owner thread to support recursive lock semantics. We currently implement 64 recursive counters, one for each of our sixty-four spin locks. To implement the counters, we divided each 16 bit BRAM entry into two columns: one that holds the 9 bit thread id for the current owner, and the second to hold an associated 6 bit recursive counter.

Lock BRAM Access Controller (lock & recursive counter controller)

The Lock BRAM access controller updates the lock owner registers and recursive counters. The state machine for this controller is shown in Figure 5-7. This state machine waits for the Operation Mode controller to issues run signals. When a lock request signal is received, the controller starts by reading the selected lock BRAM location and determines if the requested lock is free. If free (rcnt = 0), the controller writes the requesting thread id (thr id) into the owner register. If the lock is not free, the controller then compares the requesting thread ID with the owner thread ID read from the lock BRAM. If identical, the controller then checks if the recursive counter is already at maximum value. If the counter value (recur cnt) has not reached it's maximum value, the recursive counter is incremented and stored back into the BRAM. If the recursive counter is already at maximum, an ERROR signal is asserted within the error status register. If however the two thread IDs compared do not match, the controller immediately returns to the init state. When the Operation Mode controller issues a lock release signal, the controller reads the recursive count and if the read value is not one, it decrements the recursive count. If the read value is one, it first sets the owner register with a NO OWNER value before decrementing the recursive counter to zero.

Operation Mode Controller

This controller serves as an interface between the application program and the spin lock IP. The operating mode controller decodes the application program interface request and generates the appropriate control signals for the other controllers. The supported op_codes that can be issued by our API's are given in table 5-1. This controller uses read and write request signals, and two address lines to decode requests. Based on the request, the controller then outputs the following signals: lock request, lock release, and lock (owner register) init. This controller works concurrently with the Atomic Transaction controller to process lock request transactions. (Atomic Transaction controller will be described later). This controller also generates signals that initiate processing of the lock BRAM Access Controller.

Spin lock (recursive lock/unlock)

Write Request	Read Request	A13	A14	Operations	Error & Status
0	1	0	0	Read a spin lock (owner register)	N/A
0	1	0	1	Spin_lock()	Recursive overflow
0	1	1	1	Spin_unlock( )	N/A

Table 5-1 Operations Request by the Application Interface

Atomic Transaction Component

This component consists of two processes. The first process acknowledges the system buses write operation, which occurs when a lock is released. The acknowledgement is asserted on the next cycle, immediately after the write request occurs. The second process acknowledges an atomic read request, which occurs when a spin lock is requested. An atomic read process is initiated when the read request line goes high. However it does not immediately terminate the bus read request cycle with an acknowledgement. Instead, the controller waits until the selected owner register update (by the Lock BRAM access controller). As shown in Figure 5-6, the atomic transaction controller outputs two signals - enable (a_enable) and read acknowledge (rd_ack). These signals are asserted at six and eight cycles respectively, after the read request to the owner register occurs. The enable signal causes the owner register referenced by the lock ID to be read. The read acknowledgement signal to bus request occurs two cycles after the owner register has been accessed. The two cycle delay is required to allow the owner register's data output to stabilize on the system bus. Updating the owner register update time from the initial read request occurs in six cycles.

Data output multiplexer (API return value)

The result of each request is the returned of the owner thread ID and the error status as shown in Table 5-2. Both the error status and/or thread ID are input to the data multiplexer prior to being driven on the data bus.

Spin Lock APIs	Return values: bits (0 to 31)			Descriptions
	Recursive Error (4 bit)	Not used Zeroes (19 bits)	Lock Owner Thread ID (9 bits)	
spin_owner			thread ID	read lock owner
spin_unlock				write operation
spin_lock	0		thread ID	success
spin_lock	0		other thr ID	lock not avail
spin_lock	1		thread ID	recursive cnt overflow

Table 5-2 Spin Lock API return values

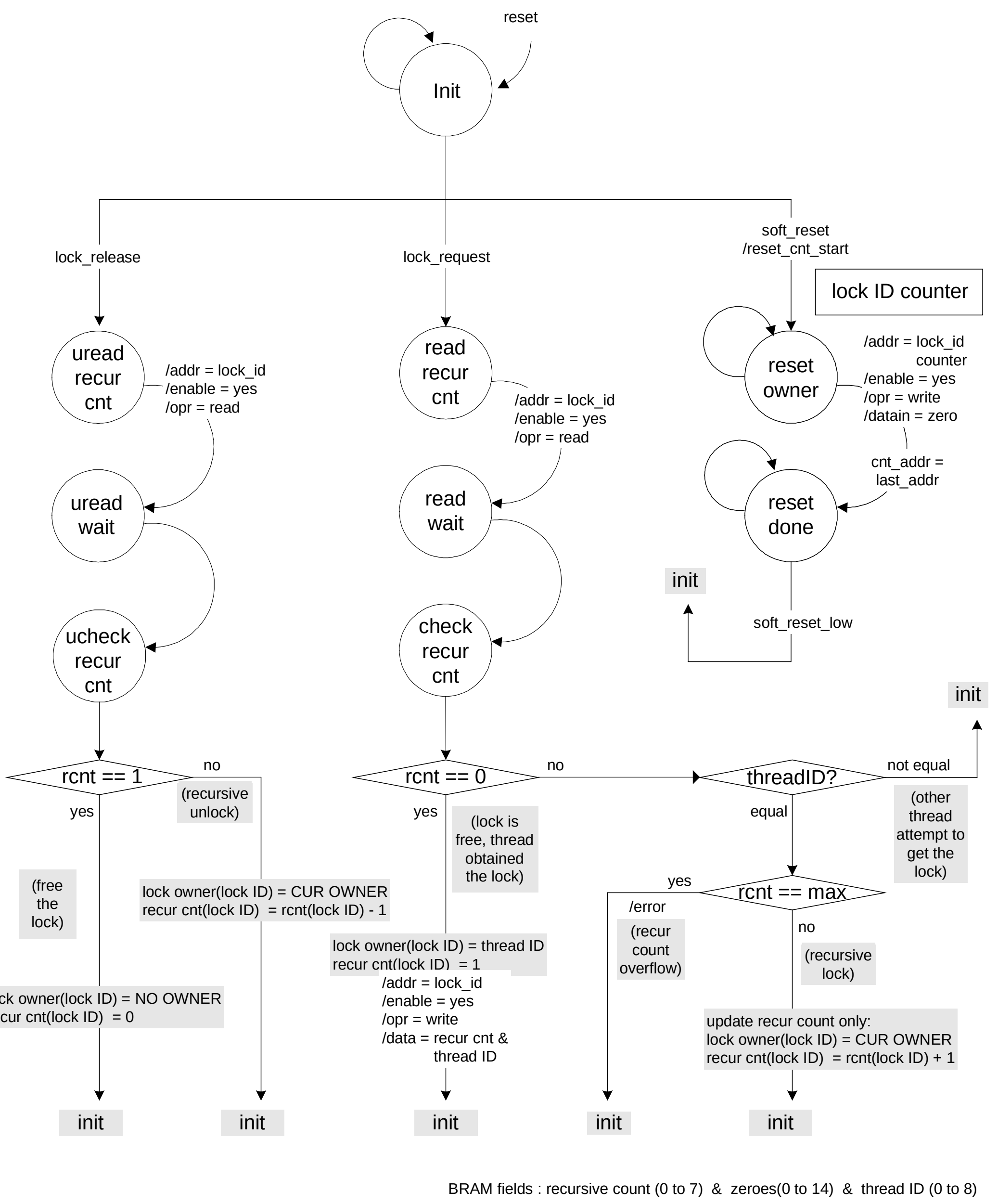


Figure 5-7 Multiple Spin Lock BRAM Access Controller State Diagram

5.4.2 Resources Analysis

Tables 5-3 and 5-4 show a comparison between the resources required to implement one spin lock from our first prototype against the resources required to implement 512 spin locks from our final design. Even though the multiple spin lock IP supports 512 spin locks, it only requires a slight increase in the total number of slices, flip-flops, and LUTs compared to a single spin lock prototype. This was a result of implementing the 512 lock owner registers and 512 recursive counters within a BRAM instead of using LUT based registers and having a single controller handle all requests.

Spin lock	# used	#total	% used
Slices	72	4928	1.46
Flip-flops	96	9856	0.97
4 inputs LUTs	71	9856	0.72
BRAMs	0	44	0.00

Table 5-3 One spin lock prototype

Spin lock	# used	#total	% used
Slices	104	4928	2.11
Flip-flops	157	9856	1.59
4 inputs LUTs	82	9856	0.83
BRAMs	1	44	2.27

Table 5-4 512 multiple spin locks implementation

5.5 BLOCKING SYNCHRONIZATIONS

Blocking synchronization allows threads that are not granted access to the semaphore to be queued and suspended, thus enabling more efficient usage of the computing resources and decreasing congestion on the system bus. For our prototype, the basic blocking control mechanism includes queue structures associated with each blocking semaphore to hold the thread ids of suspended threads or next lock owner threads.

Releasing a blocking lock can generate a wakeup event that needs to be interfaced to the operating system scheduler as well as the independent hardware threads. Thus blocking synchronization mechanism must have the capability to deliver the next lock owner or wakeup signal to the responsible subsystem. For a hardware thread to receive wakeup signal, an additional supporting interface and control infrastructure is required. For software threads, an interrupt to the processor is required to place the “unblocked” thread id back into the scheduler ready to run queue. The scheduler then may need to run a scheduling decision if a preemptive, priority based scheduling policy is in use. Obviously, not all insertions of a new thread id into the ready to run queue would result in a swapping of the currently running thread with the new unblocked thread. Independent of the outcomes of any scheduling decision, routing interrupts from external semaphores into the scheduler on the CPU will result in additional overhead processing and jitter. Obviously, migrating the scheduler and key system time services into the FPGA can eliminate the overhead and jitter of the interrupt processing and context switching associated with the unblocking operation. Instead of directly interrupting the processor to deliver the “next lock owner” to the scheduler, the blocking semaphore communicates directly with a new hardware module, the software thread manager, which then interfaces with a hardware resident scheduler. A detailed description of the thread manager including its design and implementation can be found in [61]. Interfaces and communication protocols between blocking semaphores and the hardware based thread manager will be discussed in Section 5.8.1.

Independent of the actual location of the scheduler, blocking synchronization mechanisms need to deliver the thread IDs of unblocked software threads into the scheduler queue, and issue additional wake-up commands directly to hardware threads. For testing purposes, we implemented the additional interface structure shown in Figure 5-8 between the CPU and the individual blocking synchronizations. This structure simplifies the interrupt logic needed between the blocking synchronization components and the CPU, and reduces the unnecessary interrupt overhead. This structure interrupts the CPU via the interrupt controller module.

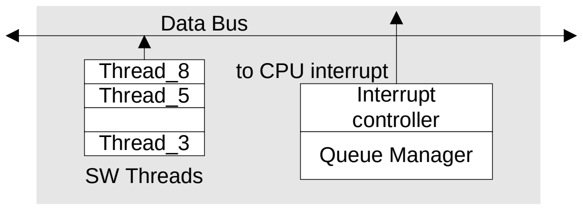


Figure 5-8   Blocking Synchronization CPU Interface

5.6 MUTEX PROTOTYPE (A BINARY BLOCKING LOCK)

The block diagram of our first single MUTEX prototype (prototype blocking binary lock) and its API are shown in Figure 5-9 and Figure 5-10 respectively.

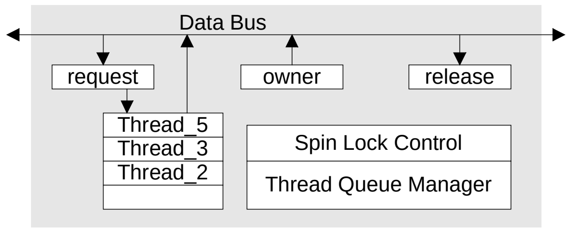


Figure 5-9   Single MUTEX (a Blocking Binary Lock)

The blocking semaphore API writes a thread id into the request register and then follows up with checking the owner register similar to the binary spin lock. If the thread did not receive the lock, then the API puts the thread to sleep in the case of a software thread, or into the wait state in the case of a hardware thread. Once awakened, the thread will read the owner register.

```
blk_lock_request(&sema, thread_id) {
  grant = 0;
  while(!grant) {
      thread_id  → rqst_reg
```

```
        if(lock_owner == thread_id)
            grant = 1;
         else sleep( )  //context switch  }
    }

  blk_lock_release(&sema, thread_id) {
      thread_id  → release    }
```

Figure 5-10 Blocking Binary Lock API

The lock is released by writing the thread_id into the release register similarly to releasing a spin lock. Unlike a spin lock, the control logic must now queue a requested thread_id if the lock is currently owned by another thread. The depth of the request queue for our prototype is a system parameter set at design time.

5.7 MULTIPLE BLOCKING SYNCHRONIZATIONS

In traditional operating systems, a sleep or wait queue is associated with each blocking semaphore. A thread goes to sleep by queuing itself into the appropriate queue. The first prototype of a blocking mechanism (described previously) included similar queue structures associated with each blocking semaphore to hold the thread ids of suspended threads. As the total number of synchronization variables in a system may be quite large, implementing separate queues for each semaphore required significant FPGA resources. We addressed this resource utilization issue by creating a single entity to control queuing and wakeup operations for multiple blocking locks.

Additionally, this approach created a single global queue for all blocking semaphores. The global queue size need only be equal to the total number of threads in the system. Conceptually, this global queue is configured as multiple sub-queues associated with different semaphores. However, the combined lengths of all sub-queues need not be greater than the total number of threads in the system as sleeping threads cannot make additional requests for other semaphores. Releasing a blocking semaphore triggers de-queuing of the semaphore's next owner. To manage the global queue efficiently, we created a single waiting queue that is divided into four tables - Queue Length, Next Owner Pointer**,** Last Request Pointer and Next Next-Owner.

The Queue Length Table maintains the length of each semaphores queue, and is accessed by indexing into the global queue table using the semaphore ID. The Last Request Table contains either a thread ID or pointer to the Next-next owner table. This table is also indexed by semaphore ID. The table is used to point to the last semaphore request.

The Next Owner Table contains the next owner thread IDs, which is used to index into the Next-Next-Owner Table. When a semaphore is released, this pointer is used to provide fast lookup of the next semaphore owner. After the next owner has been accessed the location is updated with the new next owner by reading the next-next owner table. In summary, the Next-Next owner entry serves as the head of a linked list of all blocked threads for a given semaphore.

5.8 MULTIPLE MUTEXES IP

The block diagram for a multiple block lock (MUTEX) IP is shown in Figure 5-11. This single entity carries sixty-four block locks (64 owner registers in BRAM). In addition to the lock owner registers, other structures include the global wait queue used to hold the thread ID's of blocked threads.

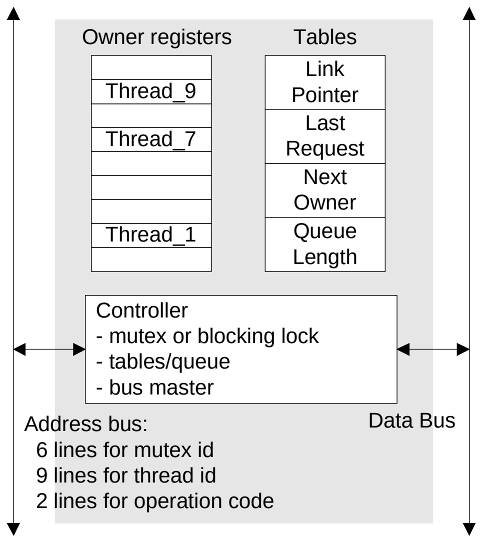


Figure 5-11 Multiple Blocking Locks IP Core

Each sub-queue within the global queue has its own queue length that varies from zero to maximum number of thread in the system. The controller responds in a similar fashion to the controller for a spin lock. However, if the owner is not free, the appropriate queue length will be updated and the requested thread ID will be queued. The controller takes eight cycles to complete the request.

To release the semaphore, the API writes its thread ID to an appropriate address. The controller will decode the address lines and update the selected owner register to non-owner status if the queue is empty. If the queue length of selected semaphore ID is not zero, the queue length will be decremented and the next owner pointer will be read to de-queue a next owner thread ID, and the owner register will be updated with the next owner.

5.8.1 Multiple MUTEXES Hardware Architecture

Our final blocking lock (mutexes) architecture consists of: 1) interface and status registers 2) lock owner registers, recursive counters (and its controller) 3) blocked threads global queue 4) global queue controller 4) other sub-controllers 5) soft reset circuits. Figure 5-12 shows the hardware components of the blocking binary semaphore (mutexes). The figure however does not include the reset circuit.

1. Interface and status registers:
    - Single Mutex ID register
    - Single Thread ID register
    - Busy status register
    - Error status register
    - Output MUX (API return status)

2. Owner registers, recursive counters and its controller
    - Up to 512 mutex owner registers implemented within BRAM (Mutex BRAM)
    - Up to 512 recursive counters implemented within BRAM (within the Mutex BRAM)
    - Mutex BRAM access controller (Multiple Mutexes controller)

3. Queue and its controller
   - Queue implemented within BRAM (Queue BRAM)
   - Queue Controller
   - Next Mutex Owner register (unblocked thread register)

4. Other Controllers:
   - Operation Mode controller
   - Atomic Transaction controller
   - HW/SW Comparator and Next Owner Address Generator
   - Bus Master

5. Soft reset circuit
   - Part of global queue and lock BRAM controllers: to reset the recursive counters, lock owner register and global queue.
   - Counter generates addresses used to reset all the owner registers and recursive counters
   - Counter generates addresses used to reset all the global queue cells

Mutex ID (Lock ID) register

This register latches the mutex ID encoded in address lines A24:A29. The address lines are latched into this register when the read request signal goes high. This register is used as an index by the Mutex BRAM Access Controller to access one of the sixty-four lock owner registers, and as an index to access tables in the next owner queue.

Thread ID register

The Thread ID register is used to hold thread ID to be compared with a lock owner register. This register is additionally used as transit storage for a thread ID, before it moves either into the global queue or an owner register. The Queue Controller uses this register as an index to access the Link Pointer table in the queue. It holds NO OWNER default value when a lock is released and no new owner available in the queue.

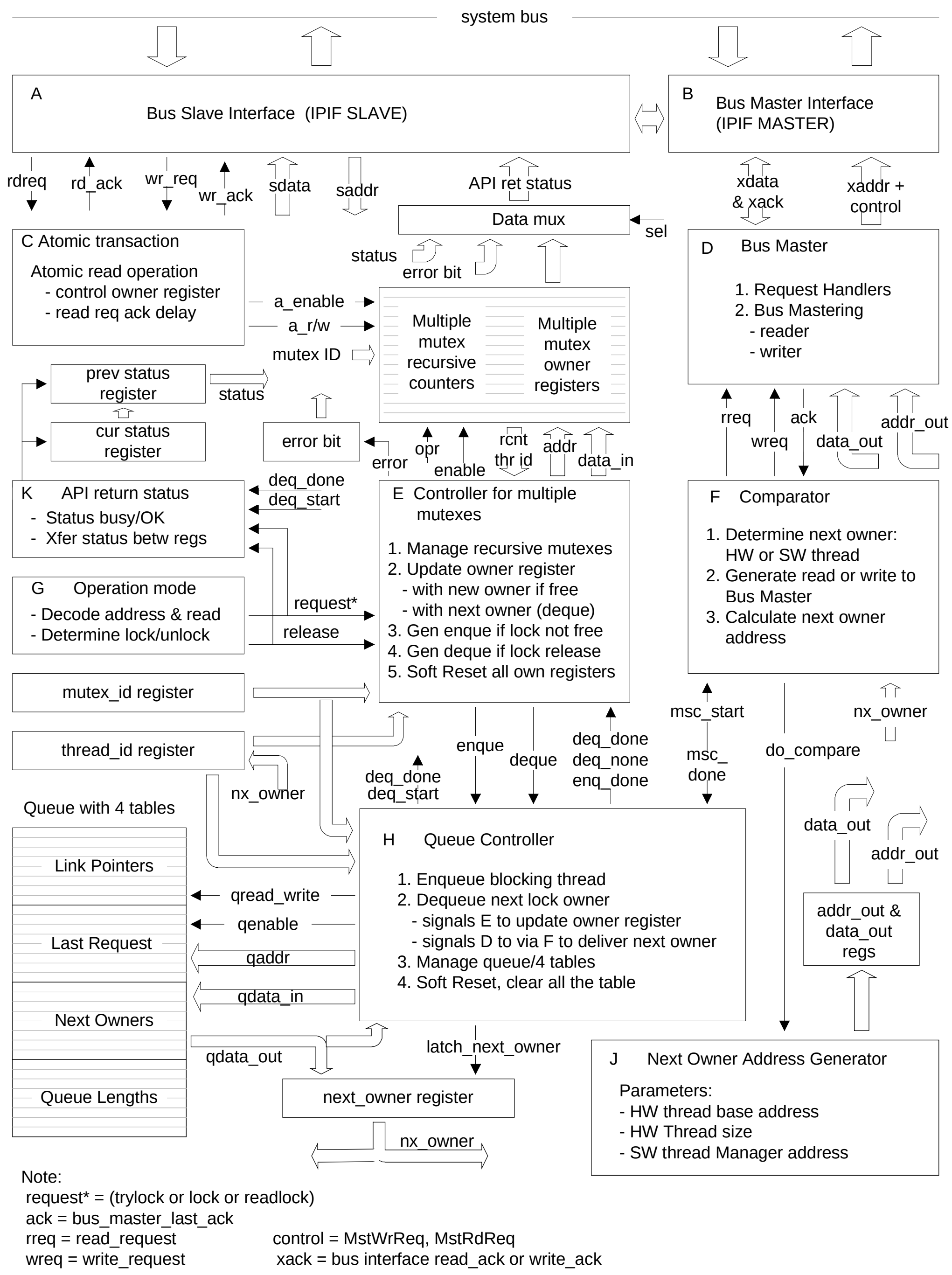


Figure 5-12 Multiple Mutexes IP Hardware Architecture

Operation Mode Controller

This controller serves as an interface between the application program and semaphore hardware. It decodes the application program interface request and generates appropriate signals to other controllers. The allowable operations that can be requested by the application program and control signals output by this controller are given in table 5-5. It uses read request and two address lines to decode an application request and outputs: lock request, lock release, and try lock signals. This controller works concurrently with the Atomic Transaction controller to service lock request transactions. (Atomic Transaction will be described later). The generated outputs initiate the Lock BRAM Access Controller operation, which in turn will cause other controllers to start operations.

Write Request	Read Request	A13	A14	Operations	Error & Status
0	1	0	0	Read a mutex (owner register)	N/A
0	1	0	1	Mutex_unlock(), release a mutex, dequeuing a next owner if there is one in the queue.	Busy
0	1	1	0	Mutex_lock(), request for a mutex, enqueuing the calling thread if the requested mutex is not free.	Recursive overflow, Busy
0	1	1	1	Mutex_trylock, request for a mutex, but do not enqueue the calling thread if the mutex is not free.	Fail/Succeed, Busy

Table 5-5 Mutex Application Interface Requests

Atomic Transaction Controller

For a description of the Atomic Transaction Controller see section 5.4.1.

Busy Status Mechanism and Error Status registers

Busy status serves as a busy indicator to the API request when the Queue controller has not completed delivery of next owner (unblocked thread) thread id to the thread manager. Even though the de-queue operation takes at most nine cycles, the delivery operation across the bus may be delayed by other system bus activities. The busy status mechanism includes two registers: Previous Status and Current Status. The status in the Previous Status register along with error status and other information as shown in table 5-6 is returned to the API request. The error status register provides an error indicator (the error bit is set) when the number of recursive lock requests surpasses the maximum value of the recursive counter.

The start and completion of an unblocked thread delivery event causes the Current Status register to change state to “busy status” and “non-busy status” respectively. An API request (lock, unlock, or try lock request) causes transfer of status from the Current Status register to the Previous Status register. Therefore an API request that occurs between the start and completion of the delivery will receive a “busy status”. When an API receives a busy status, it must retry its request. As the Lock BAM Access controller will not return to its initial state if the delivery of unblocked thread is not completed, any new API request will not cause any new operation. Since the new API request will not cause any operation and not affect the busy status (if the unblocked thread is not delivered yet), the integrity of all internal core operations is maintained.

Data output multiplex (API return value)

Hardware returns busy status, recursive error status and thread ID as shown in Table 5-6 to response to the API request (return value read by the API). The thread ID is read from one of the owner registers chosen by the atomic transaction controller. The status and thread ID are transferred to the data bus through a data multiplexer.

Mutex APIs	Return Values: bits (0 to 31)					Descriptions
	Busy status (4 bits)	Recursive Error (1 bit)	Not used Zeroes (11 bits)	Not used Zeroes (8 bits)	Mutex Owner Thread ID (8 bits)	
mutex_owner						current mutex owner
mutex_unlock	1010					not busy or success
mutex_unlock	1110					busy
mutex_lock	1010	0			thread ID	not busy, get mutex
mutex_lock	1010	1			thread ID	not busy, recursive error
mutex_trylock	1010				thread ID	not busy, get mutex
mutex_trylock	1010				other thr ID	not busy, mutex not avail
mutex_trylock	1110					busy

Table 5-6 Mutex API Return Values

Mutex BRAM Access Controller (Recursive Mutex Controller)

The MUTEX BRAM access controller has several responsibilities including updating the lock owner register, modifying the recursive lock counters, and activating queue (en-queue or de-queue) operations when necessary. The state machine of this controller is given in Figure 5-13. This controller waits for the Operation Mode controller to issue an initiate processing signal. When it receives a lock acquisition signal, it starts by reading the selected lock BRAM location to determine whether the requested lock is free. If it is free (rcnt = 0), the controller updates the owner register with the requester thread ID. If the lock is not free, the controller compares the requester thread ID with the thread ID (thr id) it just read from the lock BRAM. It they are the same, the controller then checks if the recursive counter is already at maximum value. If the counter value is not maximum, it increments the recursive counter and save the new counter value (recur_cnt) into the BRAM. If however the two thread IDs are not the same, it raises a signal to command the Queue Controller to en-queue the requester thread ID.

When the Operation Mode controller issues a lock release signal, this controller makes comparison whether the recursive count is equal to one. If the counter is not equal to one, the recursive count is decremented and the new value is saved into the lock BRAM. If the recursive counter is equal one, the controller will issue the dequeue signal to the Queue Controller. It then waits for a response from the Queue Controller. If the Queue Controller response indicates that there is no next owner in the queue, it will update the lock BRAM with NO_OWNER and reset recursive count to zero. If the Queue Controller responds that there is a next owner, it will issue a signal to busy status mechanism, and update the owner register with a new owner thread ID and set the recursive count to one. The busy status signal is to flag that it is busy and waits for Bus Master to deliver the de-queued next owner to the final destination (either scheduler queue or hardware thread). Upon received of delivery completion acknowledgement signal from the Bus Master, it resets the Current Status register to NOT_BUSY status and returns to init state.

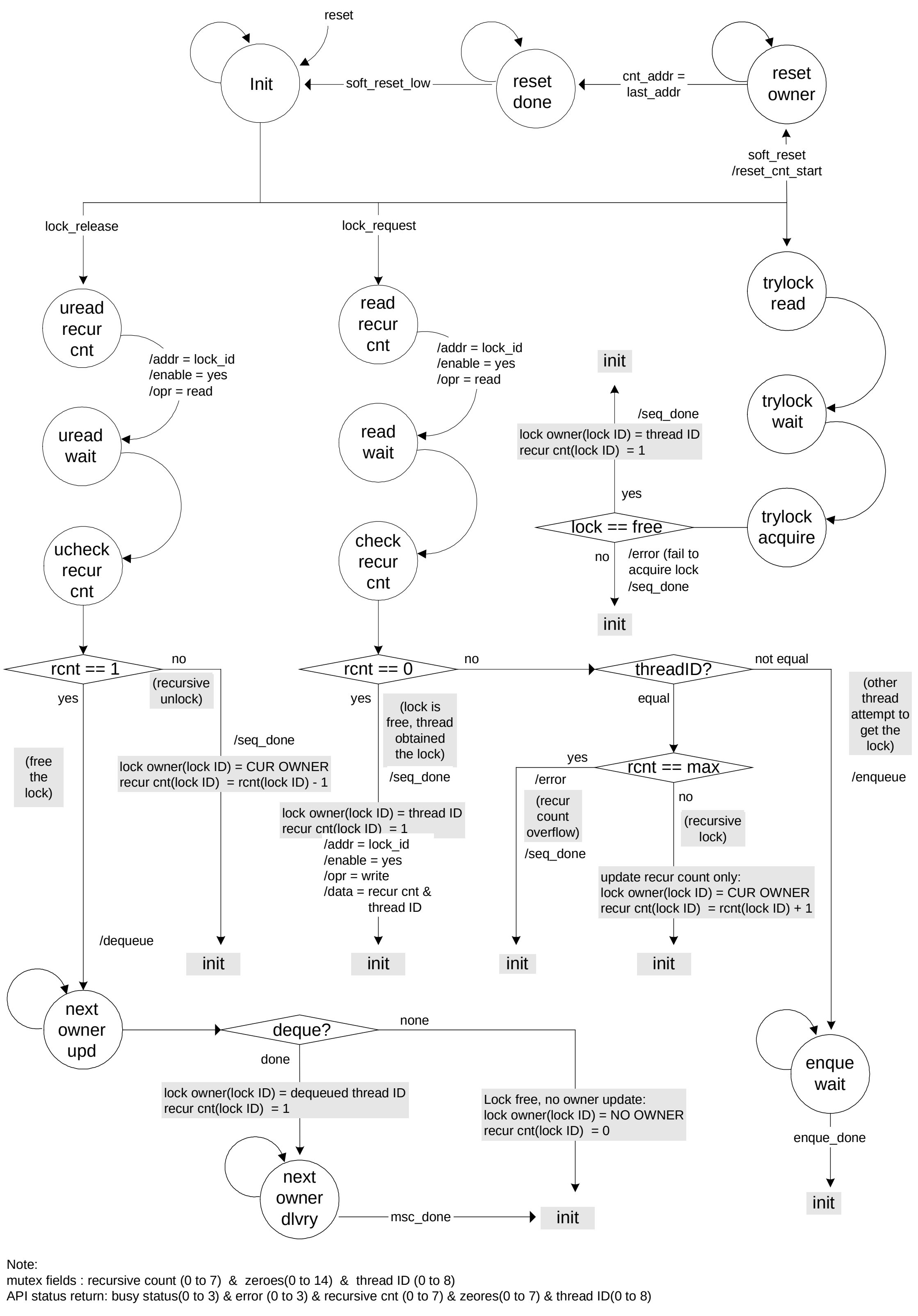

reset
Init
soft_reset_low
reset done
cnt_addr = last_addr
reset owner
soft_reset /reset_cnt_start
lock_release
lock_request
uread recur cnt
/addr = lock_id /enable = yes /opr = read
uread wait
ucheck recur cnt
read recur cnt
/addr = lock_id /enable = yes /opr = read
read wait
check recur cnt
trylock read
trylock wait
trylock acquire
init
/seq_done
lock owner(lock ID) = thread ID
recur cnt(lock ID) = 1
yes
lock == free
no
/error (fail to acquire lock /seq_done
init
rcnt == 1
no
(recursive unlock)
yes
/seq_done
(free the lock)
lock owner(lock ID) = CUR OWNER
recur cnt(lock ID) = rcnt(lock ID) - 1
rcnt == 0
no
yes
(lock is free, thread obtained the lock)
/seq_done
lock owner(lock ID) = thread ID
recur cnt(lock ID) = 1
/addr = lock_id /enable = yes /opr = write /data = recur cnt & thread ID
threadID?
not equal
equal
(other thread attempt to get the lock)
/enqueue
yes
rcnt == max
/error
(recur count overflow)
/seq_done
no
(recursive lock)
update recur count only:
lock owner(lock ID) = CUR OWNER
recur cnt(lock ID) = rcnt(lock ID) + 1
/dequeue
init
init
init
init
next owner upd
deque?
none
done
lock owner(lock ID) = dequeued thread ID
recur cnt(lock ID) = 1
Lock free, no owner update:
lock owner(lock ID) = NO OWNER
recur cnt(lock ID) = 0
next owner dlvry
msc_done
init
enque wait
enque_done
init
Note:
mutex fields : recursive count (0 to 7) & zeroes(0 to 14) & thread ID (0 to 8)
API status return: busy status(0 to 3) & error (0 to 3) & recursive cnt (0 to 7) & zeores(0 to 7) & thread ID(0 to 8)

Figure 5-13 Mutex BRAM Access Controller

Global Queue

The global queue is designed to hold up to 512 thread IDs blocked on any of the sixty-four MUTEXES. The queue is divided into the four tables shown in Figure 5-14, and is implemented within the BRAM (Queue BRAM).

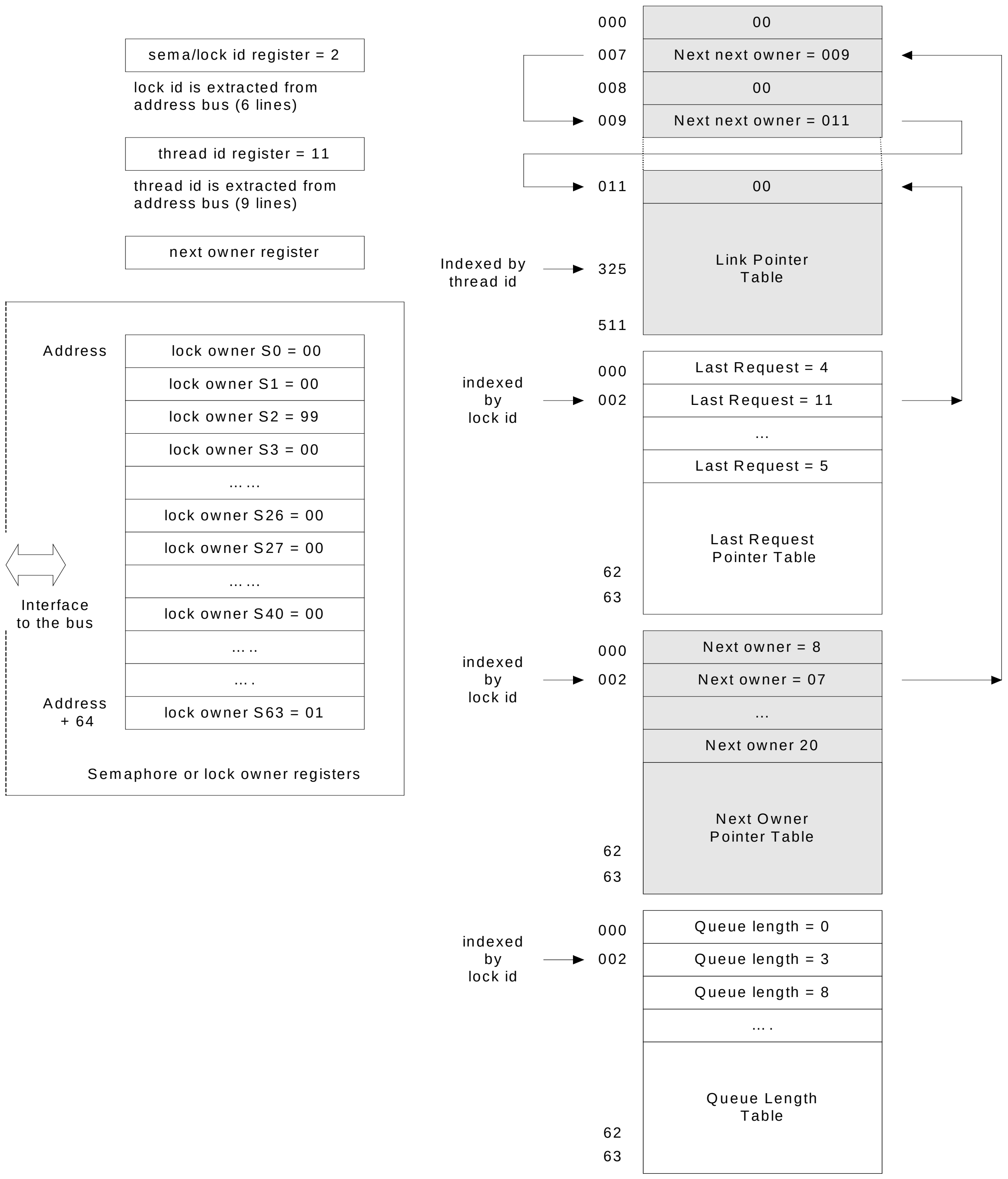


Figure 5-14 Global Blocking Queue and Lock Owner Structures

The individual tables implemented are the Queue Length, Last Request, Next Owner and Link Pointer tables. Except for the Link Pointer, all other tables are indexed by the mutex ID (lock ID or semaphore ID). The Link Pointer table is indexed by the thread ID register. An example of the operations performed on the global queue for a given mutex is shown in Figure 5-15. In this figure, the Next Owner Pointer (one of the Next Owner table cells) contains a next owner thread (thread ID). The Last Request Pointer (a table cell) contains the thread id that has made the latest mutex request, and it is used to update the Link Pointer table. For a given mutex, the Link Pointer table provides a link list of all its next owners as shown at the bottom part of the figure.

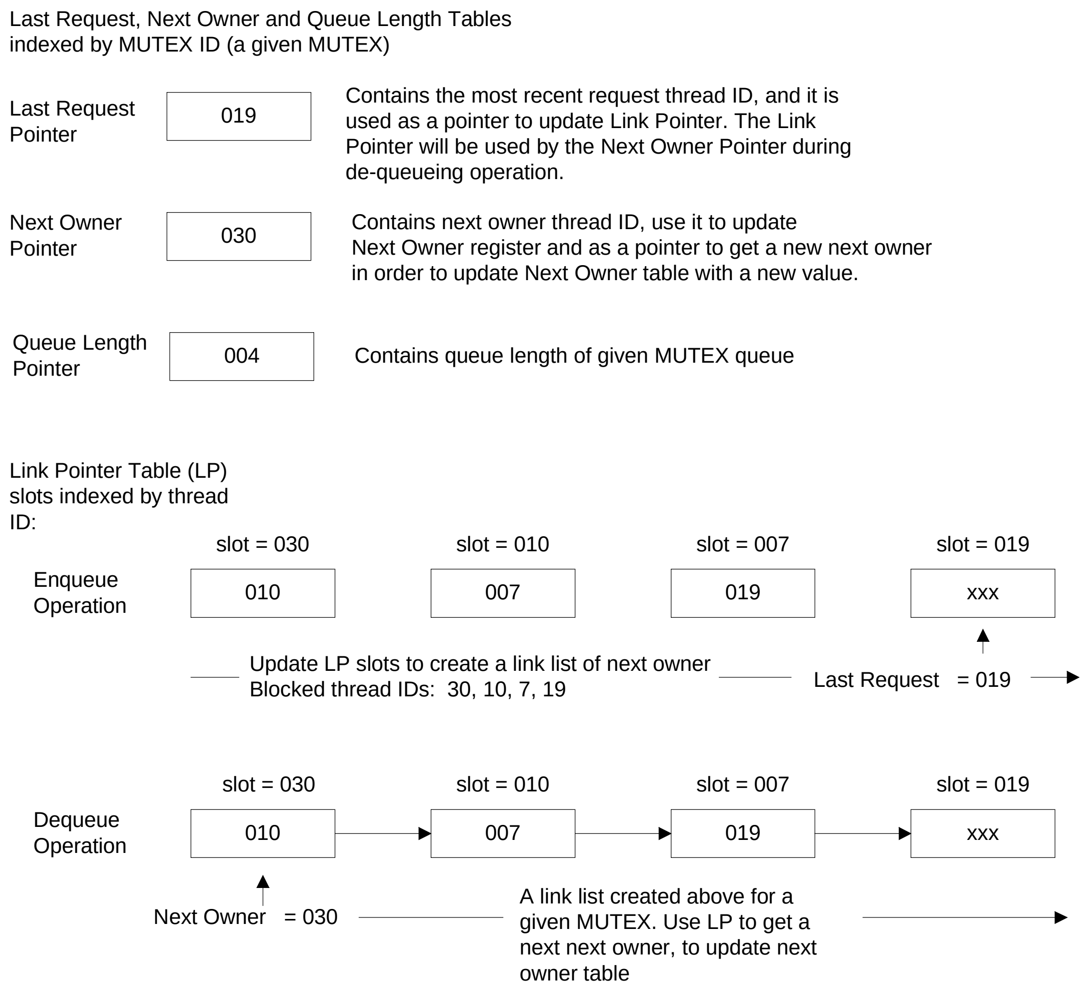

Last Request, Next Owner and Queue Length Tables indexed by MUTEX ID (a given MUTEX)
Last Request Pointer
019
Contains the most recent request thread ID, and it is used as a pointer to update Link Pointer. The Link Pointer will be used by the Next Owner Pointer during de-queueing operation.
Next Owner Pointer
030
Contains next owner thread ID, use it to update Next Owner register and as a pointer to get a new next owner in order to update Next Owner table with a new value.
Queue Length Pointer
004
Contains queue length of given MUTEX queue
Link Pointer Table (LP) slots indexed by thread ID:
Enqueue Operation
slot = 030
010
slot = 010
007
slot = 007
019
slot = 019
xxx
Update LP slots to create a link list of next owner Blocked thread IDs: 30, 10, 7, 19
Last Request = 019
Dequeue Operation
slot = 030
010
slot = 010
007
slot = 007
019
slot = 019
xxx
Next Owner = 030
A link list created above for a given MUTEX. Use LP to get a next next owner, to update next owner table

Figure 5-15 Global Queue Operation

Next Owner Register

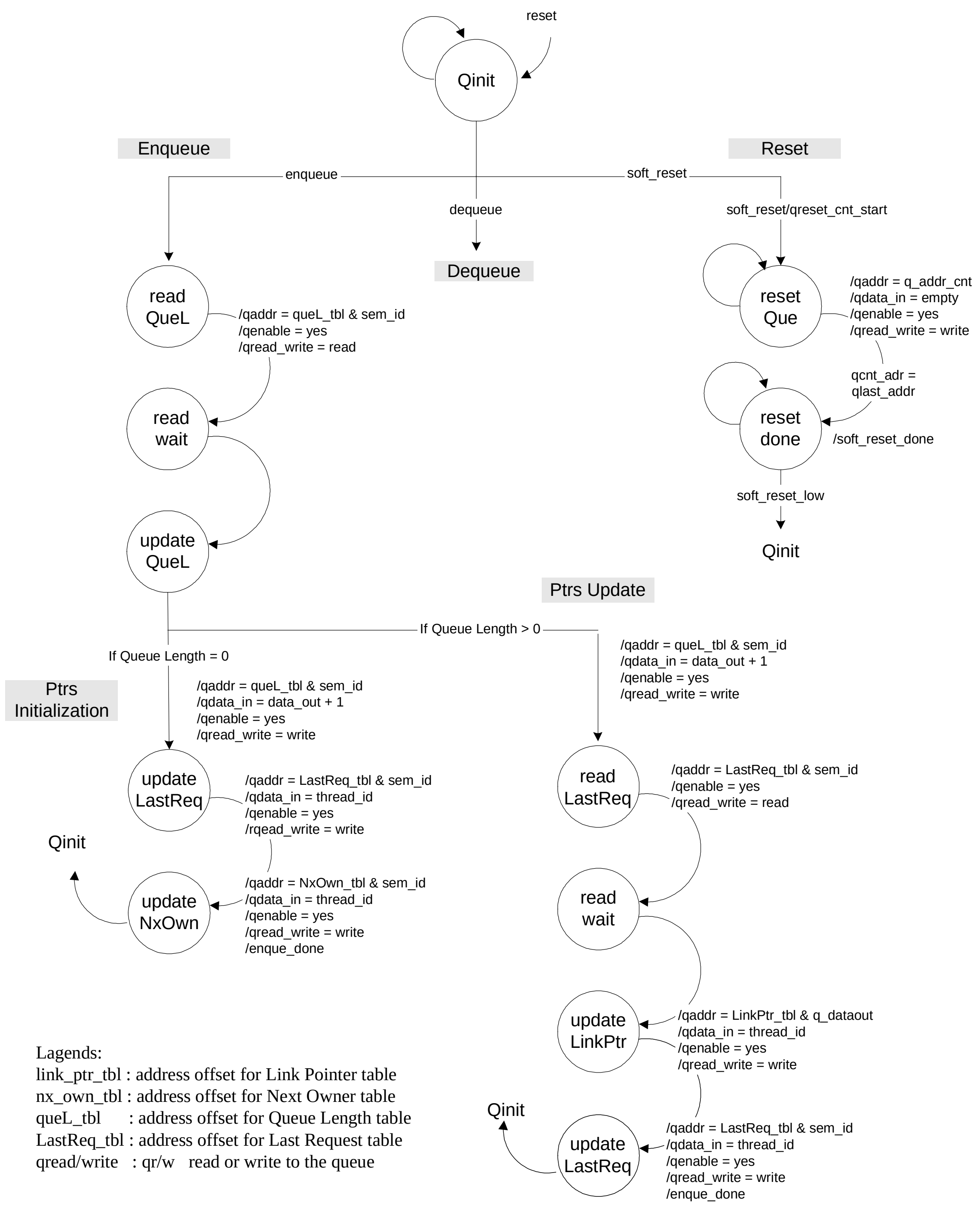
The Next owner register saves the next owner thread ID de-queued from the global queue by the Queue Controller. This enables the Queue Controller to continue managing the queue and at the same time allows other controllers to start initiating processing delivery of the next lock owner.

Queue controller

The Queue Controller is responsible for three distinct tasks: queuing and dequeing a locks next owner, and initializing all queue tables. The state machine diagrams for Queue Controller operations (queue and removal elements from the Queue BRAM) are given in Figure 5-16 and 5-17. It follows three different paths responding to three different input signals - ENQUE, DEQUEUE and SOFT RESET. If the SOFT RESET is active, it transitions to the reset state. While in transition, it outputs a signal to initialize the BRAM address counter. The BRAM address counter generates BRAM cell addresses. In the reset state, this controller initializes all the BRAM cells or locations. After clearing all the queue BRAM locations, it moves into the *QueResetDone* state, and asserts the *Sem_Rst_Done* signal. It remains in this state and continuously asserts the *Sem_Rst_Done* until the soft reset signal is de-asserted.

A Request for an owned lock initiates an en-queuing operation. An en-queuing operation starts by reading the Length Queue pointed by the Lock ID register. If the queue length is zero, the controller increases the queue length by one. Next it uses lock ID as an index to access both the Last Request Pointer and Next Owner Pointer, and initializes both pointers with current requester thread ID.

If the queue length is non-zero, the controller must perform several additional tasks. First it updates the queue length. Then it reads Last Request to get the index of Link Pointer and writes the current requester thread ID into the Link Pointer table. It uses the index it has just retrieved as a pointer into the Link Pointer table. The Link Pointers serve as a series of link lists of all the next lock owners for a given lock or semaphore. The link pointers are also used later by the Next Owner Pointer to find a new next lock owner. Finally it updates the Last Request with current requester thread ID.



Lagends:

link_ptr_tbl	: address offset for Link Pointer table
nx_own_tbl	: address offset for Next Owner table
queL_tbl	: address offset for Queue Length table
LastReq_tbl	: address offset for Last Request table
qread/write	: qr/w read or write to the queue

Figure 5-16 Queue Controller Enqueue State Machine

Releasing a lock causes the de-queuing to begin. The state machine for the de-queue operation is given in Figure 5-17. As shown in the state machine diagram, the de-queue operation has three execution paths depending on the length of the next owner in the queue. Thus all de-queue operations start with first checking the queue length. The queue length is retrieved from the Queue Length table by using the lock ID as an index.

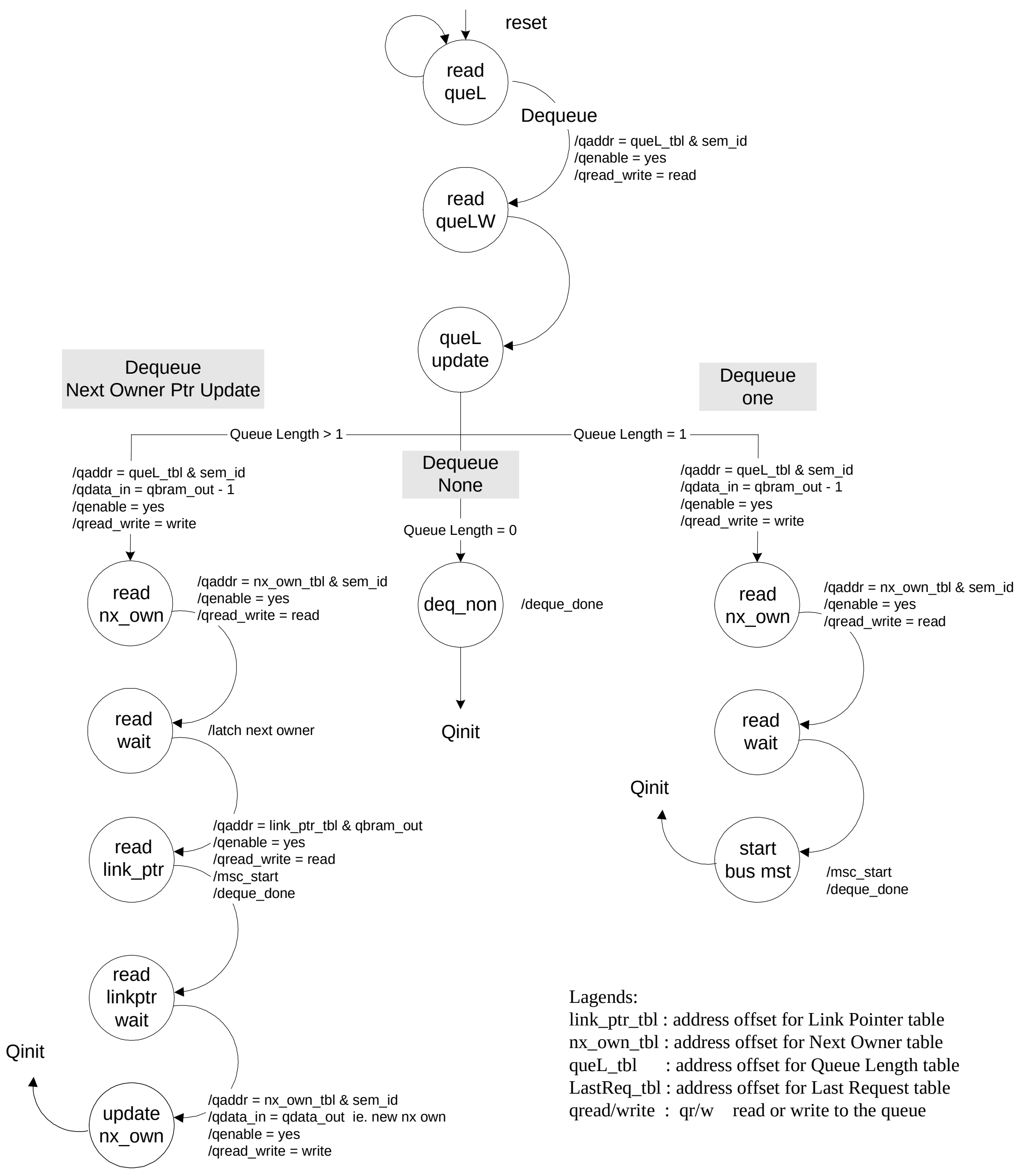
If the queue length is zero, which means no next lock owner, the de-queue operation ends here. It then notifies the Lock BRAM Access Controller by raising the DEQUE_NONE signal. The Lock BRAM Access Controller then frees the lock by writing NO_OWNER value into the owner register.

If the queue length is one, controller proceeds to reduce the queue length by one. Then it de-queues the next owner from the Next Owner table into the Next Owner register and signals the Hardware/Software Comparator (including Next Owner Address Generator) to start its task, which in turn will signals the Bus Master. The Bus Master delivers the next owner (unblocked thread) to either the Software Thread Manager or hardware thread.

If the queue length is more than one, the controller needs to execute several more steps in addition to mentioned above. First it updates the Queue Length table with a new length. Next it de-queues the next owner, and transfers it to the Next Owner register, and signals the Hardware/Software Comparator. It also needs to update the Next Owner Table with a new next owner. To get a new next lock owner, it uses the next owner that it just retrieved as an index to read the Link Pointer table. Then it updates the Next Owner table with the new next owner it just obtained from the Link Pointer table. The Next Owner table update and the next owner delivery (notification) in actual run concurrently, as shown in the state machine diagram.

HW/SW Comparator & Next Owner Address Generator

This controller includes a comparator, a process to calculate delivery destination (of next owner or unblocked thread), a state machine, and a pair of register to place the unblocked thread address and its wake-up code: ADDR_OUT and DATA_OUT.



Lagends:
link_ptr_tbl : address offset for Link Pointer table
nx_own_tbl : address offset for Next Owner table
queL_tbl : address offset for Queue Length table
LastReq_tbl : address offset for Last Request table
qread/write : qr/w read or write to the queue

Figure 5-17 Queue Controller De-queue Operation

Upon receiving a signal from the Queue controller, this controller (as shown in Figure 5-18) determines whether the next owner (unblocked thread in the next owner register) is a hardware or software thread. The Software Thread Manager requires a read transaction in order to put the unblocked thread back into the scheduler queue. However, hardware threads require synchronization IP to write wake-up codes into command register in order to unblock.

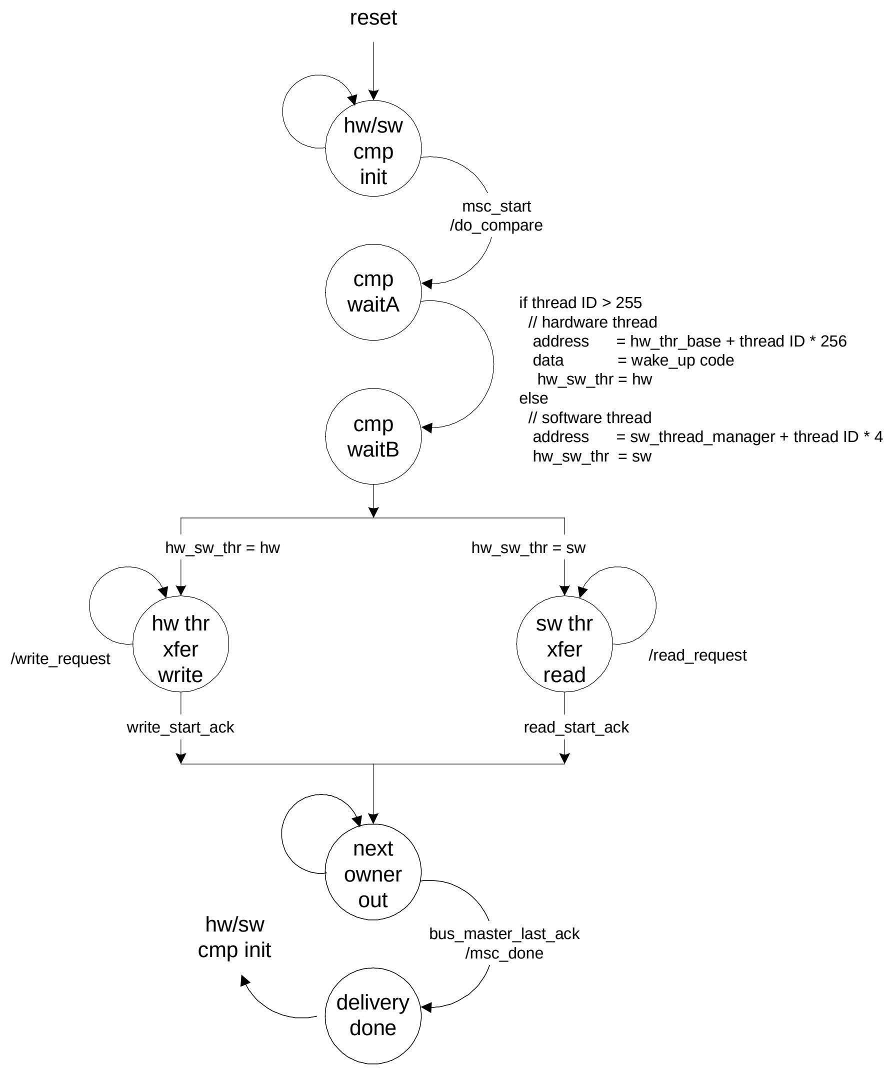


Figure 5-18 HW/SW Comparator & Next Owner Address Generator

In the case of a software thread, an address is calculated by adding the Software Thread Manager base address with the thread ID multiplied by four. For hardware threads, the address of a command register is calculated by adding the command register address offset, the base address for hardware threads location in system memory map and the product of thread ID with hardware thread size. In addition, a hardware thread requires a wake-up code be placed into the DATA_OUT register. The base addresses of both hardware base address and the software manager as well as the hardware thread size are passed as generics during system set-up. The controller then asserts either a read or a write request to one of the request handlers of the Bus Master, and wait for acknowledgement.

When the controller receives the acknowledgment from the Bus Master, it de-asserts its request and proceeds to the wait state. It waits in this state until it receives delivery acknowledgement (LAST_ACK) from the Bus Master. Upon receiving this acknowledgment, it issues DEQ_DONE (MSC_DONE) and returns to the initial state. The DEQ_DONE is required to signal the Lock BRAM Access controller and busy status process that the delivery of the unblocked thread has completed.

Bus Master

Unlike spin locks, blocking locks must master the bus in order to en-queue blocked threads to either the thread manager or a specific hardware thread command register. The Bus Master hardware includes Bus Master controller and a pair of request handlers. Since the Hardware/Software Comparator makes available the destination address and data registers, this module does not provide multiplexers and registers for both the data and address buses. The responsibility of the bus-mastering controller is to accept different bus transaction requests from one the request handlers, generates read or write request signals to the Bus Master interface either to read in data or write wakeup code to hardware threads.

As shown in Figure 5-19, there are two possible paths the Bus Master follows depending on which request handler it receives signals from. The Bus Master performs read operation to the software Thread Manager to deliver next lock owner to the scheduler queue. Otherwise the Bus Master writes the wake-up code to the appropriate hardware thread. The state machine asserts read/write request, with the address bus and data bus connected to the ADDR_OUT and DATA_OUT registers respectively, and waits for acknowledgment from the bus interface. When it receives acknowledgement, it de-asserts its request, issues LAST_ACK signal to the Hardware/Software Comparator and return to init state.

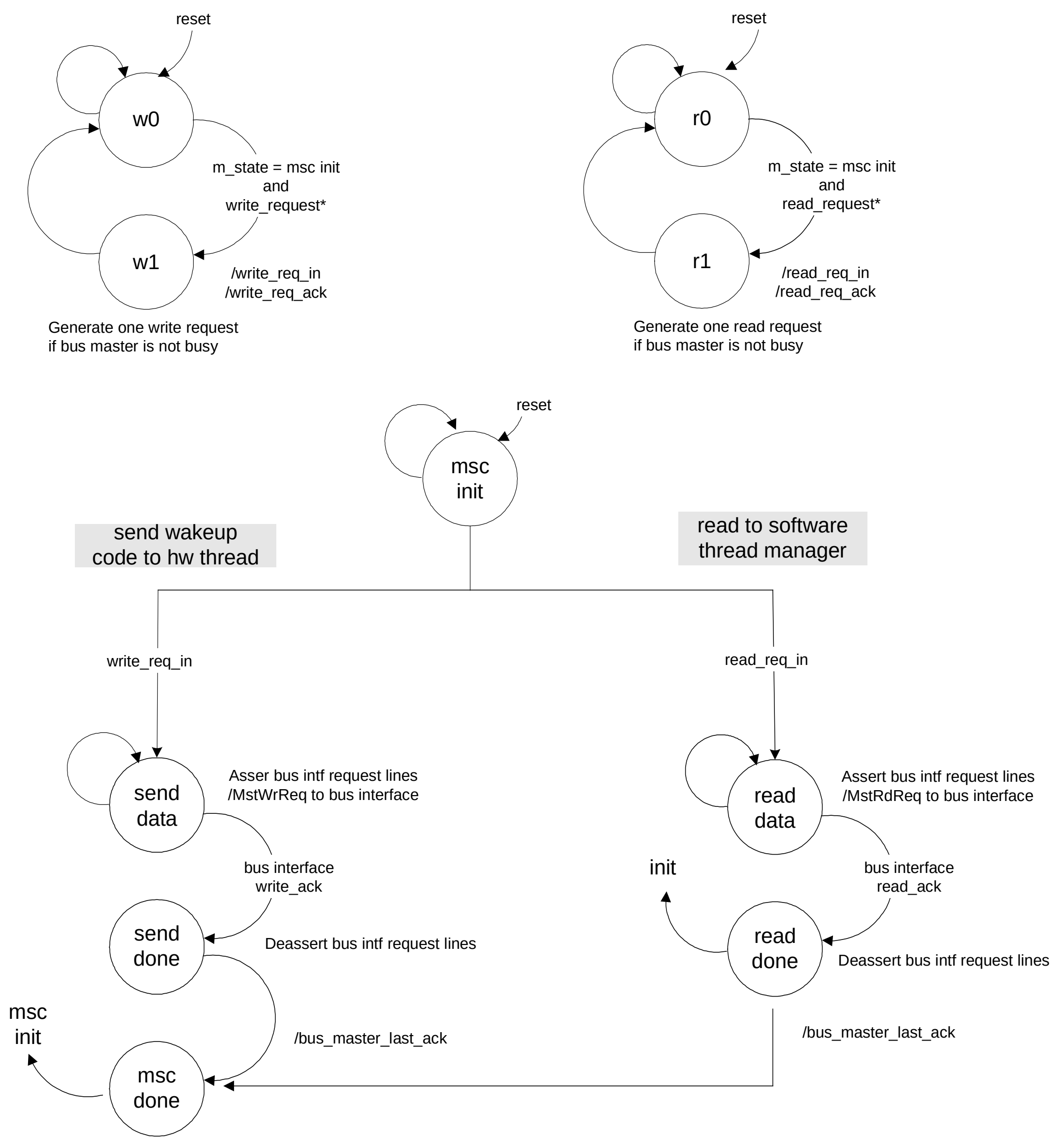


Figure 5-19   Bus Master State Machine

5.8.2 Resource Utilization

Tables 5-7 and 5-8 show the resource comparisons for our blocking lock implementations. It is interesting to observe that our new design, which fully supports 512 blocking locks now requires less slices, flip-flops, and LUT's that the original design for a single blocking lock. Our new approach does require an additional BRAM.

Block lock	# used	#total	% used
Slices	584	4928	11.12
Flip-flops	572	9856	5.80
4 inputs LUTs	808	9856	8.20
BRAMs	1	44	2.27

Table 5-7 Hardware Resources for a Prototype MUTEX

Block lock	# used	#total	% used
Slices	357	4928	7.24
Flip-flops	381	9856	3.87
4 inputs LUTs	548	9856	5.56
BRAMs	2	44	4.56

Table 5-8 Hardware Resources for Multiple (512) MUTEXES

5.9 BLOCKING COUNTING SEMAPHORE PROTOTYPE

The block diagram of a blocking counting semaphore is shown in Figure 5-20. The API pseudo code for a blocking counting semaphore is shown in Figure 5-21.

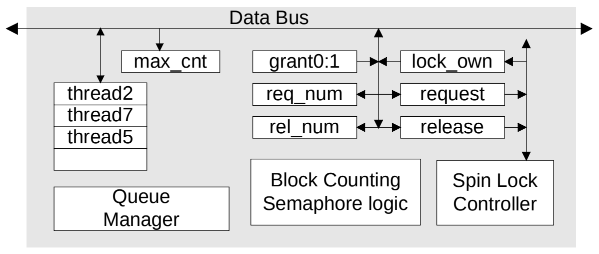


Figure 5-20 Blocking Counting Semaphore (Prototype)

With the blocking counting semaphores if the resources are available, the thread continues to run. However, if insufficient resources are available, then the thread will be queued and suspended similarly to a blocking binary lock. The semaphore IP will issue all queued thread_ids for rescheduling when resources are released. Just as in the spin semaphore protocol, writing a value into the request number register (req_num) causes a Boolean value to be set in the grant register. If insufficient resources are available, then the thread_id needs to be queued before the spin lock is released.

```
blk_sema_request(&sema, thread_id, value){
  grant =0;
  while (!grant) {
    spin_lock_request(&sema, thread_id )
    value → rqst_num
    grant ← grant0:1
    if (grant)
       spin_lock_release(thread_id )
    else {
       queue thread_id/&command_reg
       spin_lock_release( )
       sleep( )  //context switch  } } }
```

Figure 5-21 Blocking Counting Semaphore API

5.10 MULTIPLE BLOCKING COUNTING SEMAPHORES IP

The block diagram for our final multiple blocking counting semaphores is shown in Figure 5-22. This entity carries sixty-four counting semaphores (64 counters in BRAM). In addition to the counters, there is a wait queue to hold the thread ID's of blocked threads. This single queue is designed to queue all threads blocked on all of the sixty-four semaphores. To request for a semaphore, the *sem_wait( )* API issues a read to an address formed by encoding the semaphore ID and thread ID as the least significant bytes of the base address. In response to this read operation, the controller decodes the address line and extracts both the semaphore and thread ID's. The controller reads the counter pointed by the extracted semaphore ID, places the read value on data bus, terminates the bus cycle, and performs additional operations depending on the value of the counters. If the counter value is non-zero, the controller decrements the counter by one, otherwise extracted thread id is en-queued. If the returned value is zero, the API puts the thread to sleep. The API pseudo code for blocking counting semaphore is shown in Figure 5-23.

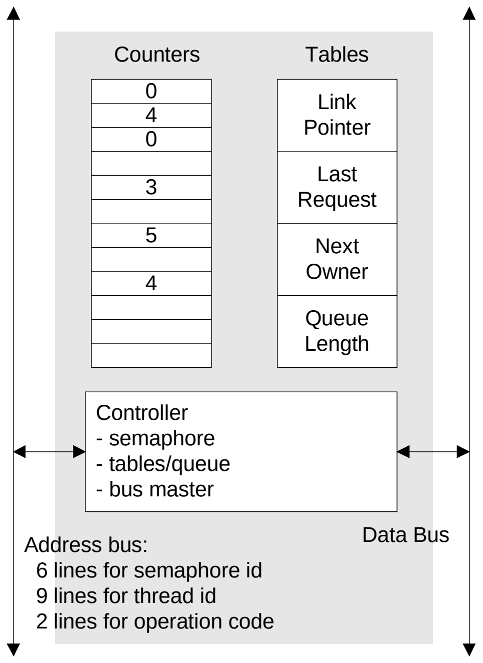


Figure 5-22 Blocking Counting Semaphore

```
sem_wait(sema_id, thread_id) {
    address <= encode sema_id , thread_id;
     if location(address) == zero
         sleep( );
     else
        continue;
}

sem_post(&sema, thread_id) {
    address <= encode sema_id , thread_id;
    thread_id → location(address);
}
```

Figure 5-23 Blocking Counting Semaphore API

To release the semaphore, the *sem_post( )* API write its thread ID to an appropriate address. The controller state machine then decodes the address lines and reads the selected counter. If the selected counter is non-zero, it will incremented by one. Otherwise if the counter is zero, the controller proceeds checking the queue length of the selected semaphore. If the queue length is zero, the counter will be incremented. If the queue is not zero, the next semaphore owner will be de-queued and the counter will not be updated.

5.10.1 Multiple Counting Semaphore Hardware Architecture

Our final multiple counting semaphore IP hardware architecture consists of: 1) interface and status registers 2) semaphore counters and its controller 3) global queue 4) global queue controller 4) other controllers 5) soft reset circuits. Figure 5-24 shows the hardware components of the semaphore. The Figure 5-24 however does not include the reset circuit.

Semaphore counters and its controller

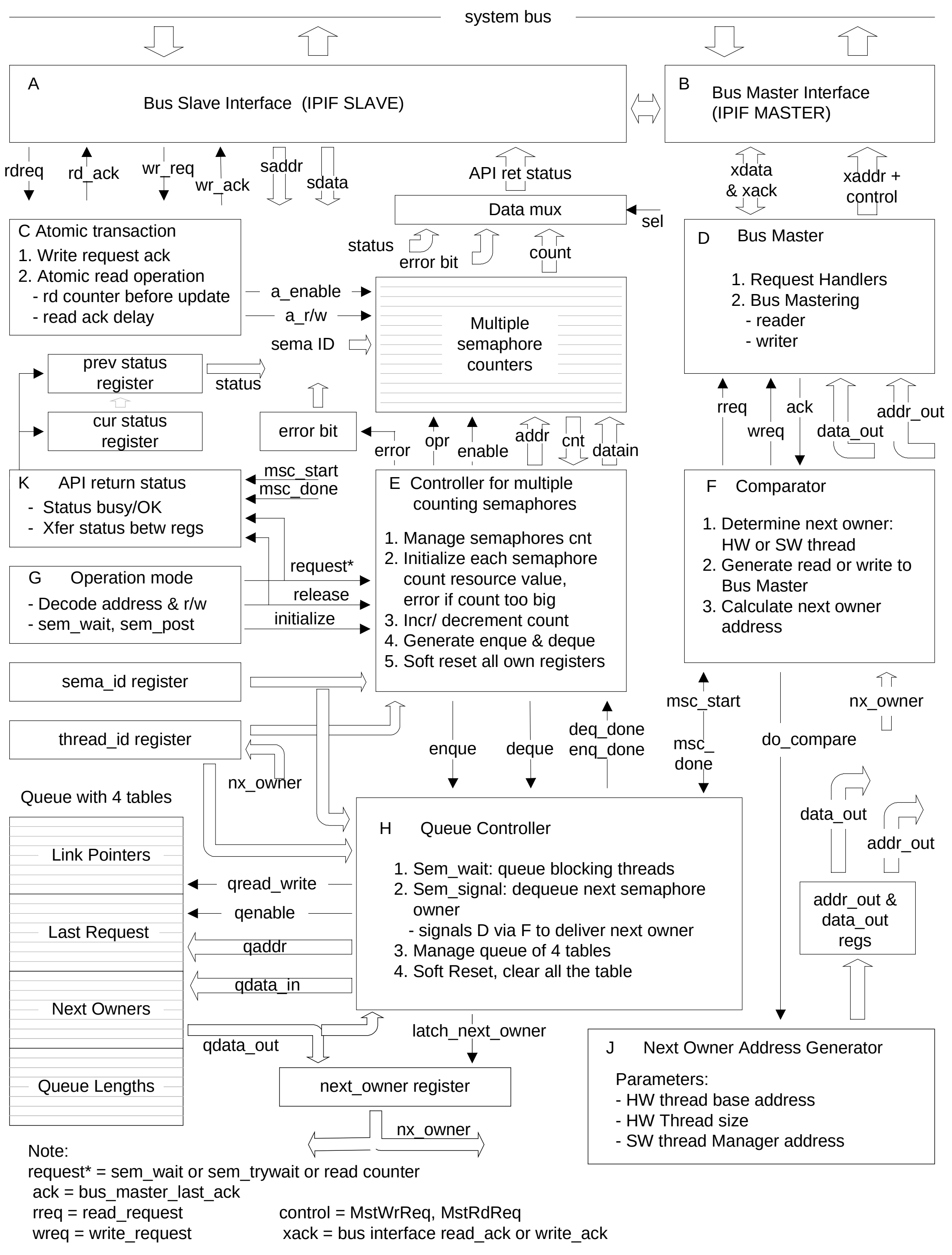
- Semaphore counters implemented within BRAM (SEMA BRAM)
- SEMA BRAM access controller

Interface and status registers:

- Semaphore ID register
- Thread ID register
- Busy status register
- Error status register
- Output MUX (API return status)

Global queue and its controller

- Global queue implemented within BRAM (Queue BRAM)
- Queue Controller
- Next Owner register (next semaphore owner or unblocked thread)



Note:
request* = sem_wait or sem_trywait or read counter
ack = bus_master_last_ack
rreq = read_request
wreq = write_request
control = MstWrReq, MstRdReq
xack = bus interface read_ack or write_ack

Figure 5-24 Counting Semaphore Hardware Architecture

Other Controllers:

- Operation mode controller
- Atomic transaction controller
- Comparator/Next Owner Address Generator
- Bus master

Soft reset mechanism

- To initialize global queue and semaphore counters
- Semaphore address counter generates addresses of all the semaphores that to be reset.
- Queue address counter generates addresses of all global queue location to be reset.

Semaphore ID register
See description in binary blocking semaphores or mutexes (Section 5.8.1)

Thread ID register
See description in binary blocking semaphores or mutexes (Section 5.8.1)

Busy status and error status registers
The error status register provides error indicator (the error bit is set) when the initialization value is greater the maximum value of the semaphore counter. Similar to the MUTEX busy status, a status mechanism serves as a busy indicator to the API requests when the queue controller has not completed its delivery of next owner or unblocked thread from previous API request. The busy status mechanism includes two registers: Previous Status and Current Status. The start and completion of thread delivery events cause the current status register to change to "busy status" and "non-busy" status respectively. Every API operation including wait, post and try wait causes transfer of status from the current status register to the previous status register. The status in the Previous Status register is returned to the API. When an API receives busy status, it should retry the request. API requests that occur while the current transaction is yet to complete will cause no harm. If the current transaction is busy, SEMA BRAM access controller will not return to initial state, thus any new API requests will not cause any new operation.

Operation Mode Controller
This controller serves as an interface between the application program and semaphore hardware. It decodes the application program interface request and generates appropriate signals to other

controllers. The type of operation requested by the application program and control signals output by this controller is given in table 5-9. It uses read request, write request and two address lines to decode application requests and generates appropriate signals: SEM_GETVALUE, SEM_POST, SEM_WAIT, SEM_TRYWAIT, SEM_INIT. This controller works concurrently with the atomic transaction controller in response to semaphore wait API request. (Atomic transaction controller will be described later). The generated signals trigger the SEMA BRAM Access Controller and Queue Controller to start to execute, which in turn cause other controllers to start their tasks.

Write Request	Read Request	A12	A13	A14	Operations	Error & Status
0	1	0	0	0	Sem_getvalue( ), read a semaphore counter	N/A
0	1	0	0	1	Sem_post( ), add count, unblocked thread if there is at least one in the queue	Busy
0	1	0	1	0	Sem_wait( ), decrement count, block if counter is 0	Busy
0	1	0	1	1	Sem_init status*	Error (Count>max), Busy
0	1	1	0	0	Sem_trywait(), request a semaphore, thread will not block if not successful	Success/Fail, Busy
1	0	0	1	1	Sema_init( )* initialize a semaphore counter	N/A

Table 5-9 Semaphore Application Interface Requests

Semaphore Counters (SEMA BRAM)

This semaphore IP entity supports sixty-four 8-bit non-negative counters implemented in the block RAM (BRAM). The current implementation can support up to 512 semaphore counters within this BRAM (we named it SEMA BRAM). The read and write operation on the SEMA

BRAM is controlled by an access controller (SEMA BRAM access controller) that will be described later.

Atomic transaction controller

This component consists of several processes. The first process is to acknowledge the system bus write operation. This process asserts acknowledgement on the next cycle after the write operation occurs. The second process is to acknowledge atomic read operation when a hardware thread or a software thread requests for a semaphore. This atomic read process is initiated when the read request line goes high. However it does not perform immediate acknowledgement but waits for the referenced semaphore counter output become stable, which can occur before the counter value is updated. As shown in Figure 5-24, the atomic read process issues two signals; enable and read acknowledge (a_enable and rd_ack) that goes high at four and six cycles respectively after the *sem_wait* request occurs. (Enable is asserted just before semaphore counter update, to wait for busy status register output to become stable, as semaphore count and busy status are returned to the API). Enable signal causes counters referenced by the semaphore ID to be read, output its content on the data bus. The controller then issues acknowledgement for the system bus to latch the output (when the counter data output is stable). The nonzero value (counter value) returned to the API indicates that the requested semaphore resource is available. This controller does not modify the counter value, which is the responsibility of SEMA BRAM access controller.

Output MUX (API return value)

All API requests are returned through this output multiplexer. Different statuses including busy status, error status, and semaphore availability returned to the API are shown in table 5-10.

SEMA BRAM access controller

Two controllers manage accesses to the SEMA BRAM (multiple semaphore counters): the atomic transaction controller and SEMA BRAM access controller. The atomic transaction controller reads a semaphore counter pointed to by the semaphore ID register, and puts this value on the bus in response to *sem_wait( )* API. The transfer of the counter contents to the data bus must be completed before the alteration of the value for a current *sem_wait()* operation. A more detailed description is given in the atomic transaction controller.

The SEMA BRAM access controller is the main controller for accessing the SEMA BRAM. The operation of the state machine shown in Figure 5-25 is described below for each different API request. This controller updates the semaphore counters in response to the operation mode controller signals.

Semaphore APIs	Return values: bits (0 to 31)					Descriptions
	Busy status (4 bits)	Overflow error (1 bit)	Not used Zeroes (11 bits)	Not used Zeroes (8 bits)	Sema Counter (8 bits)	
sem_cnt_read	1010				count	read semaphore counter
sem-post	1010					not busy or success
sem_post	1110					busy
sem_wait	1010				zero	not busy, no semaphore
sem_wait	1010				nonzero	not busy, get semaphore
sem_wait	1110					busy
sem_trywait	1010				zero	not busy, no semaphore
sem_trywait	1010				nonzero	not busy, get semaphore
sem_trywait	1110					busy
sem_init	1010	0				init success
sem_init	1010	1				init error

Table 5-10 Semaphore API Return Values

When the operation mode controller issues a *sema_post* (lock release) signal, this controller performs several steps depending on the value of the counters and the number of thread blocked on the referenced semaphore:

1. Generate de-queue signals to the Queue controller to allow concurrent operations.. The Queue controller outputs the queue length or number of blocked thread in the queue. The queue length is pre-fetched at this step to be available at the start of step 3.
2. Check if the referenced counter value (cnt) is equal to zero. If the counter is nonzero, the counter will be incremented by one, the new counter value is saved and controller returns to init state.
3. If the counter is zero, further steps are required. It checks the queue length provided by the Queue controller.

4. If the queue length is zero implying no blocked threads are in the queue, then the counter will be incremented by one, the new counter value saved. The controller returns to init state.
5. If the queue length is nonzero, the controller proceeds to a wait state, and waits until the Bus Master controller on behalf of the Queue controller delivers the unblocked thread either to the software thread manager or hardware thread (further details is given in Queue controller). In this case counter remains zero. When it receives the MSC_DONE signal from Bus Master, it returns to the init state.
6. It does not return to init state until the delivery of unblocked thread is completed, thus any new request from the API will not cause any operation. This is to ensure that API gets busy status consistent with the SEMA BRAM access controller action. In other word it should return to the init state at the same time as the busy status register changes to NON_BUSY status.

Sem-wait operation:

The *sem_wait* operation depends on the value of the counters:

1. When the *sem_wait* is initiated, this controller reads counter and checks its value
2. If the counter is nonzero, it will be decremented by one, new value is saved and controller returns to init state.
3. If the counter is zero, the en-queue operation is initiated. The controller asserts the en-queue signal to command the Queue controller to en-queue the requester thread ID.
4. The controller then waits for the Queue controller to complete its operation before returning to the init state.

Sem-trywait operation*:*

The *sem_trywait* operation is similar to the *sem_wait* operation except requester thread will be not queued (thread will not block) if it cannot gain the requested semaphore: The controller state machine executes the following sequences:

1. The controller reads the referenced semaphore counter and checks the value
2. If the counter is nonzero, it will be decremented by one and the new value saved. Controller returns to init state.
3. If the counter is zero, perform no further action and return to the init state.

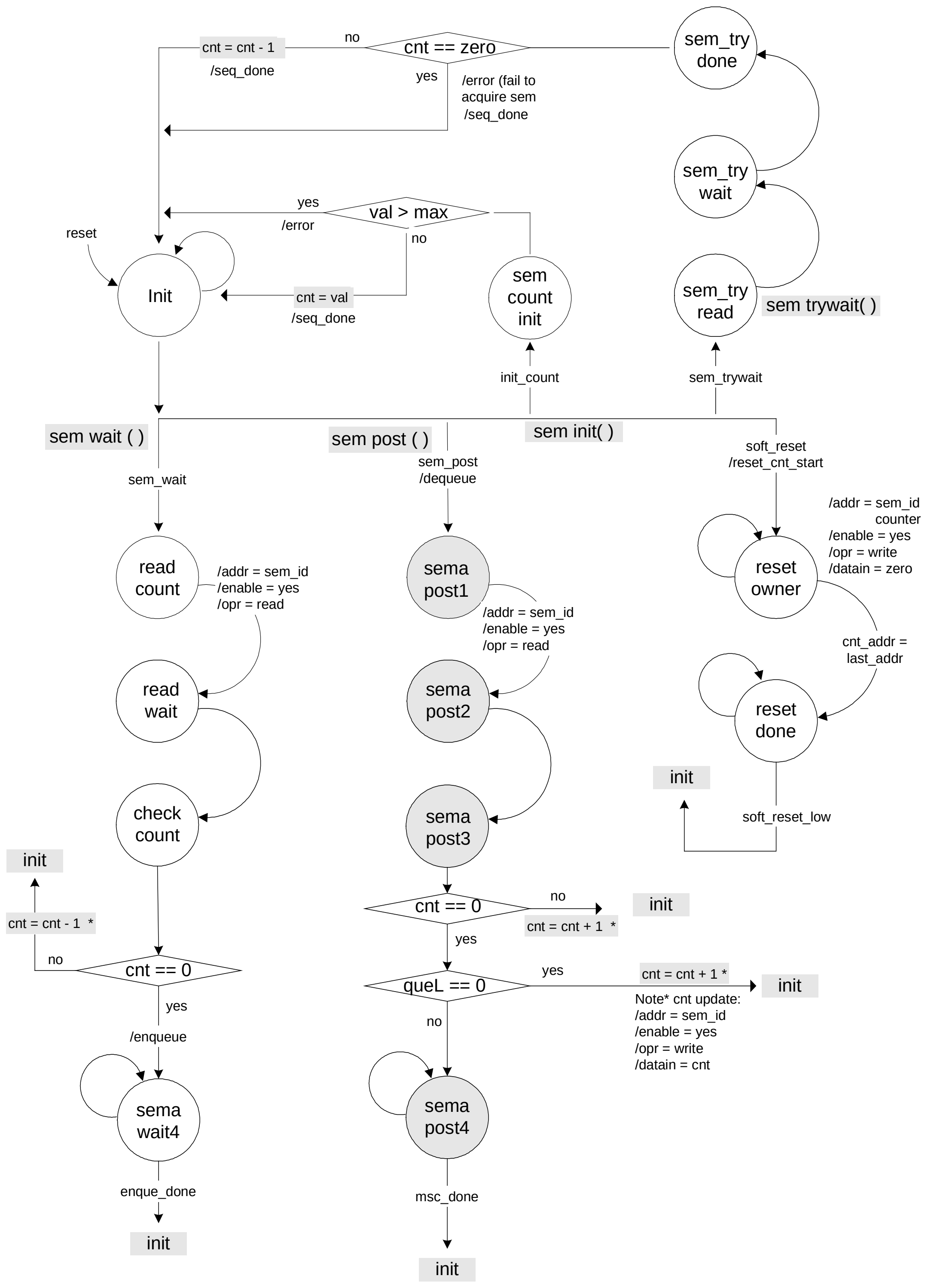


Figure 5-25 Counting Semaphore BRAM Access Controller

Sem_init operation:

The semaphore counter value initialization procedure is made-up of a pair of write and read operations. First the API writes the initial value, followed by a read operation. If the read operation returns NO_ERROR status, the initialization was successful. The controller takes the following steps in response to the initialization procedure:

1. In response to the write, the controller checks if the value is greater than the maximum value.
2. If the value (val) if greater than maximum value, overflow error bit is set, and the error is returned to the read operation.
3. If the value is within the allowable range, the counter is updated with the new value, followed by initialization of the queue length (of the referenced semaphore) to zero (by the Queue controller). NO_ERROR status is returned to the read operation.

Global Queue

The queue is divided into the four previously described tables implemented within the BRAM (Queue BRAM). The tables are semaphore queue length, last request, next semaphore owner and link pointer. Except for the Link Pointer table, all other tables are accessed with the semaphore ID as an index. The Link Pointer table is indexed by the thread ID. The global queue can support up to 512 blocked threads.

Next Owner Register

The Next owner register saves the thread id of the next owner de-queued from the global queue by the Queue Controller. This enables the Queue Controller to continue managing the queue and at the same time allows the global queue controller to simultaneously deliver the next semaphore owner.

Global Queue Controller

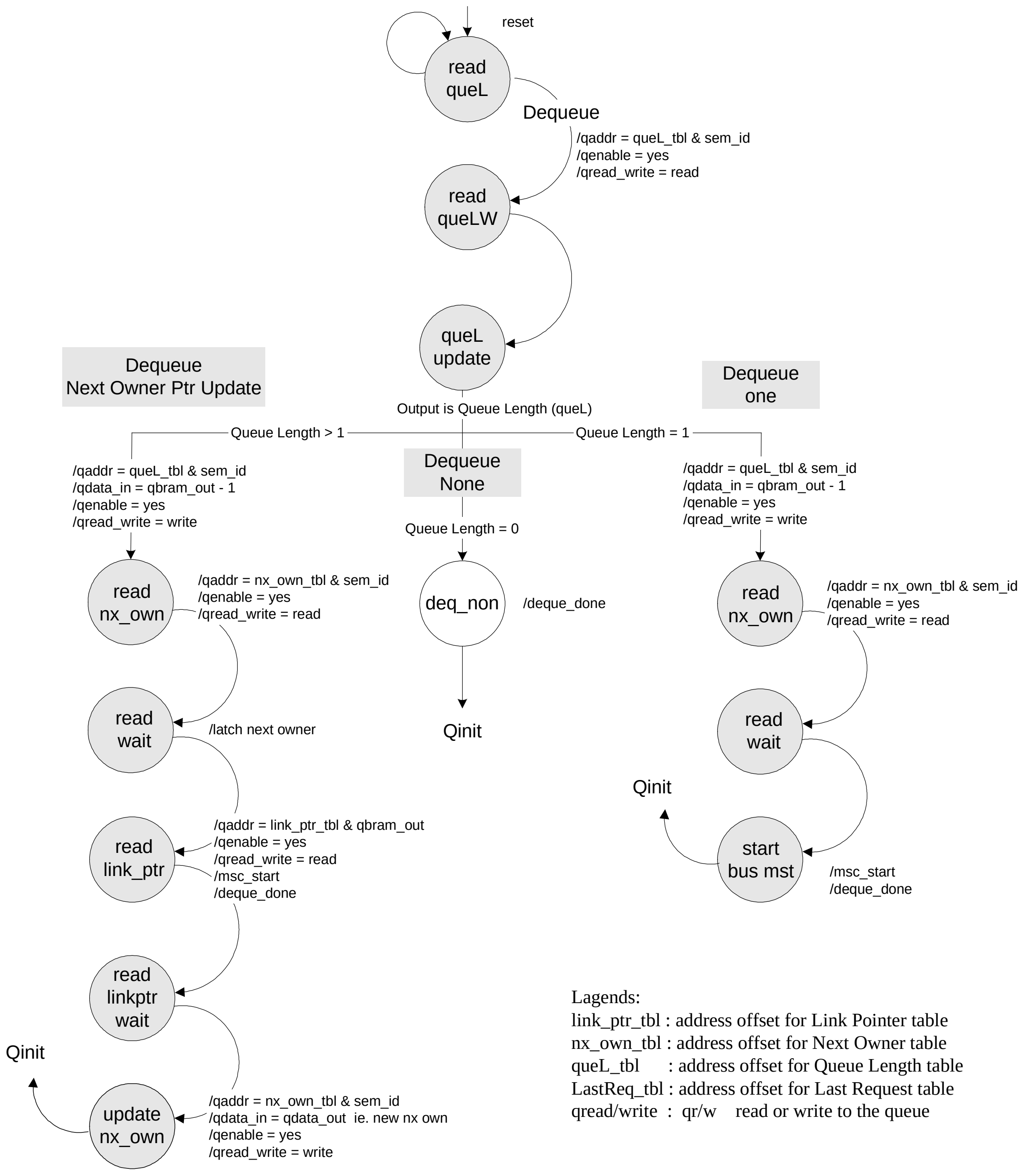
The Queue Controller is responsible for three distinct tasks: queuing the semaphore next owners, de-queuing the next owner and initialize all the queue tables. It follows three different paths responding to three different input signals - ENQUEUE, DEQUEUE and SOFT RESET. If the SOFT RESET is activated, the controller transitions to the reset state. While in transition, it outputs a signal to initialize the BRAM address reset counter. In the reset state, this controller initializes all the QUEUE BRAM locations similar to reset operation described in the MUTEX section.

Requesting semaphore zero (*sem_wait*) initiates an en-queuing operation. Upon receiving the ENQUEUE signal from the SEMA BRAM access controller, the en-queuing operation reads the Length Queue table indexed by the semaphore ID register. If the queue length is zero, the controller increases the queue length by one. Next it uses the semaphore ID as an index to access both the Last Request Pointer and Next Owner Pointer, and initializes both pointers with the current requester thread ID. If the queue length is non-zero, the controller performs the following additional tasks. First it updates the queue length. Next it reads Last Request to get the index of Link Pointer. Then it writes the current requester thread ID into the Link Pointer table. It uses the index it has just retrieved as a pointer to the location in the Link Pointer table. The Link Pointers serve as a link list for all next owners for a given semaphore, which will be used later by the Next Owner Pointer to find a new next semaphore owner. Finally it updates the Last Request with current requester thread ID.

A semaphore post causes the de-queuing to run concurrently with SEMA BRAM access controller. The state machine for the de-queue operation is given in Figure 5-26, and the shaded states run concurrently with SEMA BRAM access controller. As shown in the state machine diagram, the de-queue operation has three execution paths depending on the length of the next owner in the queue. Thus all de-queue operations start with checking the queue length first. The queue length is retrieved from the Queue Length table by using the semaphore ID as an index. If the queue length is zero, which means no next semaphore owner, the de-queue operation ends here. It then notifies the SEMA BRAM access controller by raising the DEQUE_NONE signal.

If the queue length is one, controller proceeds to reduce the queue length by one. Then it de-queues the next owner from the Next Owner table into the Next Owner register and signals the HW/SW Comparator/Next Owner Address Generator to start its task, which in turn signals the Bus Master. The Bus Master starts to deliver the next owner.

If the queue length is more than one, the controller needs to execute several more steps in addition to mentioned above. First it updates the Queue Length table with a new length. Next it de-queues the next owner, and transfers it to the Next Owner register, and signals the Comparator/Next Owner Address Generator. Then it needs to update the Next Owner Table with a new owner. To get a new next semaphore owner, it uses the next owner that it just retrieved as an index to read the Link Pointer table. Then it updates the Next Owner table with the new next owner it just obtained. The Next Owner table update and the delivery of next owner run concurrently, as shown in the state machine diagram.



Lagends:
link_ptr_tbl : address offset for Link Pointer table
nx_own_tbl : address offset for Next Owner table
queL_tbl : address offset for Queue Length table
LastReq_tbl : address offset for Last Request table
qread/write : qr/w read or write to the queue

Figure 5-26 Dequeue State Machine

When the de-queue operation starts, the controller issues a BUSMSC_START signal to flag that it is busy and waits for the Bus Master signal to deliver the de-queued next owner to the final destination (either scheduler queue or hardware thread). As there are possibilities of new API requests from other processors or hardware threads between the acknowledgement of the current API request and delivery of an unblocked owner, a busy status is necessary. The BUSMSC_START (msc_start) signal causes the current status register to change to a busy status. Upon receiving a completion delivery acknowledgement signal from the Bus Master, it issues MSC_DONE signal to reset the busy status to NOT_BUSY and returns to its init state.

The Bus Master and Next Owner Address Generator operations are similar to the ones described in the MUTEX section (Section 5.8.1).

5.11 CONDITION VARIABLES

Condition variables enable threads to block and synchronize for arbitrary conditions. Condition variables typically support the wakeup of one or all blocked threads when the blocking condition is met. Thus condition variables prevent threads from wasting processor time waiting for certain conditions to change**.** The condition variable is usually used in conjunction with a lock and a predicate (typically a Boolean variable). The lock is needed to protect the predicate since it is normally associated with shared resources.

Consider for example, a shared queue where client threads queue their job request and worker threads remove the requests and perform the requested tasks. The shared resource in this case is the queue and the predicate is an empty queue. The shared resource has to be protected by a lock, (either blocking or spin lock) since any code that uses it is part of a critical section. An additional variable (condition variable) is needed to provide a safe mechanism for worker threads to block on predicates involving the shared resource rather than busy waiting. The worker threads need to block when the predicate is true. Worker threads go to sleep with the condition variable (CV) by calling *cond_wait(CV)* when the predicate is true. The predicate changes when the client threads deposit their job requests. The client threads then use the condition variable to signal (wake-up) a worker thread. Whenever a client thread adds a request into the queue, it signals (calls *cond_signal(CV)*) on the condition variable that a change has taken place. This signal wakes up a blocking thread, which then reevaluates the predicate.

When a waiting thread is signaled, it must acquire the lock first before evaluating the predicate. If the predicate is false, the thread should release the lock and block again. The lock must be released before the thread blocks to allow other threads to gain access to the lock and change the protected shared resource. The release of the lock and the blocking must be atomic so that another thread does not change the queue status between these two operations (queuing requests can occur between the lock release and thread block events). The *cond_wait* function takes the lock as an argument and atomically releases the lock and blocks the calling thread. Since the signal only means that the variable may have changed and not that the predicate is now true, the unblocked thread must retest the predicate each time it is signaled.

The predicate itself is not part of condition variable. It must be evaluated by calling the routine each time before *cond_wait( )* is called. The following are the steps should be followed when using a condition variable to synchronize on an arbitrary predicate or condition [40]:

Waiting on a condition variable
Acquire a lock or mutex say M1 (M1 protect predicate)
Evaluates the predicate
If the predicate is false, call *cond_wait(&cv, &M1)* and go to step 2 when it returns.*
If the predicate is true, perform some work
Release mutex M1

Signaling on a condition variable
Acquire mutex M1
Change the predicate
Call *cond_signal(&cv)* to signal the condition variable
Release mutex M1

*The *cond_wait( )* atomically releases the lock and blocks the calling thread in order to avoid the lost wake-up problem. Thus, the lock is released explicitly if the predicate is true and released implicitly within the *cond_wait( )* predicate is false. When a thread waiting on a condition variable is unblocked, it reacquires the lock automatically as part of the unblocking process.

Sample implementation of the condition variable functions [42]:

```
/* The user program has to acquire a mutex say mtx before testing the predicate  */
/* If the predicate fails, call this function */
/* If predicate is success, perform some work and release the mutex */
void cond_wait( condition *cv, lock_t  *mtx)
{
     spinlock_lock (&cv->queueLock);  /* protect condition variable queue */
     add self to the queue;
     spinlock_unlock (&c->queueLock);
     mutex_unlock (mtx);      /* release mutex that protects the predicate before blocking  */
     context_switch();        /* perform context switch  */
     /* When wake-up from sleep, the signal has occurred  */
     mutex_lock(mtx);         /* acquire the mutex (predicate protection) again  */
     return;
 }

void cond_signal (condition *cv)
/* Wake up one thread waiting on this condition  */
{
     spinlock_lock (&cv->queueLock);      /* protect condition variable queue */
     de-queue one thread from linked list, if it is nonempty;
     spinlock_unlock (&cv->queueLock);
     if a thread was removed from the list, make it runnable;
     return;
  }

void cond_broadcast (condition *cv)
/*Wake up all threads waiting on this condition  */
{
     spinlock_lock (&cv->queueLock);     /* protect condition variable queue */
     while  (queue or linked list is nonempty)    {
               dequeue all threads from linked list;
               transfer them to scheduler queue;
     }
     spinlock_unlock (&cv->queueLock);
  }
```

5.11.1 Hardware Implementation of Condition Variables:

The block diagram for a multiple condition variable is shown in Figure 5-27. This single entity provides control for sixty-four condition variables. The essential components include a global waiting queue, atomic transaction controller and bus master. The global waiting queue is used to hold the threads (thread IDs) waiting on one of the sixty-four condition variables. This single global queue is sized to queue up to 512 threads blocked on sixty-four condition variables. The condition variable IP expects the application interface (API) to encode a condition variable ID and a thread ID within the address during each normal read operation. A single control structure within the condition variable IP then performs the necessary steps within each single read bus operation. If the controller cannot complete its operation within a given read operation, it asserts its busy status. Busy controller does not take any actions on new API requests except returns the busy status. The API must then retry the read until it receives a non-busy status from the hardware.

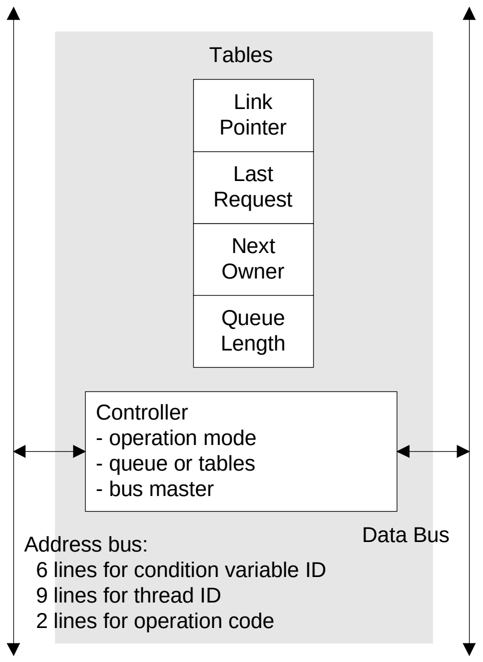


Figure 5-27 Multiple Condition Variables Core

The cond_wait API is given in Figure 5-28. In response to this read operation, the controller decodes the address lines and extracts both the condition variable and thread ID. The controller transfers the busy status register to the data bus, and terminates the bus cycle. It then may continue to perform additional operations depending busy status. If the busy status is not set (not busy), the controller queues the extracted thread id into the global queue, otherwise it performs no additional operation. If the returned value is not busy (success), the API then can proceed to release the predicate spin lock and continues to perform a context switch (sleep). If the return value is busy (fail), the API continues to perform read operations until it gets the free status. The API pseudo code for condition wait is shown in below:

```
/* The user program has to acquire a mutex say mtx before testing the predicate  */
/* If the predicate fails, call this function */
/* If predicate is success, perform some work and release the mutex */

void cond_wait( cv_id, mtx)
/* Queue thread waiting on this condition  */
{
    address = encode cv_id, thread_id
    status = fail
    while( status == fail)  {
       status = *address    /* perform read on the busy status register */
       wait ( )
     }
    mutex_unlock(mtx)     /* release mutex that protects predicate  before blocking  */
    context_switch( )

    /* When wakes-up from sleep, the event has occurred  */
    mutex_lock(mtx)       /* reacquire mutex  */

    return

  }
```

Figure 5-28 Cond_wait API

To signal a condition variable, the *cond_signal( )* API (as shown in Figure 5-29) performs a read with an address formed by encoding the condition variable ID as the least significant bytes of the base address. The controller state machine then decodes the address lines to extract the operation request and condition variable. The controller places the busy status on the data bus in response to the request. If the controller is still busy performing a previous request, no further action on the new request will take place. If controller is free, it proceeds checking the referenced condition variable queue length. If the queue length is zero, controller goes no further and returns to init state. If the queue is not zero, it removes the next condition variable owner from the queue. Then it turns on busy status and proceeds to deliver the unblocked thread (thread ID) to the scheduler queue or hardware thread. When the delivery is complete, it updates the busy status register to not busy.

```
void cond_signal (cv_id)
/* Wake up one thread waiting on this condition  */
{
     address = encode cv_id, thread_id
     status = fail
     while( status == fail)  {
         status = *address   /* perform read busy status register */
         wait ( )
       }
    return
}
```

Figure 5-29   Cond_Signal API

To broadcast a condition variable, the *cond_broadcast( )* API performs a read from an address formed by encoding the condition variable ID as the least significant bits of the base address and two address lines for operation request. The *cond_broadcast( )* API is shown in Figure 5-30. The controller state machine then decodes the address lines to extract the operation request and condition variable. The controller places the busy status on the data bus in response to the request. If the controller is still busy performing a previous request, no further action occurs on the new request. If the controller is free, it proceeds checking the referenced condition variable queue length. If the queue length is zero, the controller terminates. If the queue is not zero, the controller removes all the blocked thread id’s for the condition variable. It then asserts a busy status and proceeds to deliver the first unblocked thread (thread ID) to the scheduler queue or hardware thread. When the delivery of all of the unblocked threads is complete, it updates the busy status register to not busy. The system bus can be locked while delivering all the unblocked threads to the scheduler. However locking the system bus for long durations will affect system response. Instead the busy status is used to indicate the state of the controller.

```
void cond_broadcast (cv_id)
/* Wake up all the thread waiting on this condition  */
{
    address = encode cond_var_id, thread_id
    status = fail
    while( status == fail)   {
        status = *address  /* perform read on busy status */
        wait ( );
      }
    return
 }
```

Figure 5-30   Cond_Signal API

5.11.2   Condition Variable Hardware Architecture

The condition variable IP architecture consists of: 1) interface and status registers 2) a global queue to hold thread (thread ID) block on the condition variables 3) global queue controller 4) other controllers 5) soft reset circuits. Figure 5-31 shows the hardware components of multiple condition variables IP core. The figure however does not include the reset circuit.

Interface and status registers

- Condition variable ID register
- Thread ID register
- Busy status register
- Output MUX (API return status)

Global Queue and its controller

- Global Queue implemented in BRAM (Queue BRAM)
- Queue Controller

- Next Owner register (next condition variable owner)
- Queue Length register

Other Controllers:

- Operation mode
- Atomic transaction controller
- HW/SW Comparator/Next Owner Address Generator
- Bus Master

Soft reset component

- To initialize condition variable global queue
- Address counter (BRAM address reset counter) generates address of all global queue cells that to be reset.

Condition variable ID register

This register holds the condition variable ID referenced by a thread. A user program interface (API) uses several address lines to encode a specific condition variable ID that it needs to access. For the sixty-four condition variables implemented within our system address lines A24 to A29 are used to encode the condition variable ID. These lines are latched into this register when the read request goes high and this IP is selected. This register is also used for other purposes including as an index to access queue length, next owner and next owner tables of the global queue.

Thread ID register

See description in the blocking mutex section (Section 5.8.1).

Busy Status Mechanism

Similar to MUTEX busy status mechanism, a condition variable core has two registers: Previous Status and Current status to serve as a busy indicator to the API when the queue controller has not completed its delivery of unblocked threads (condition variable next owners).

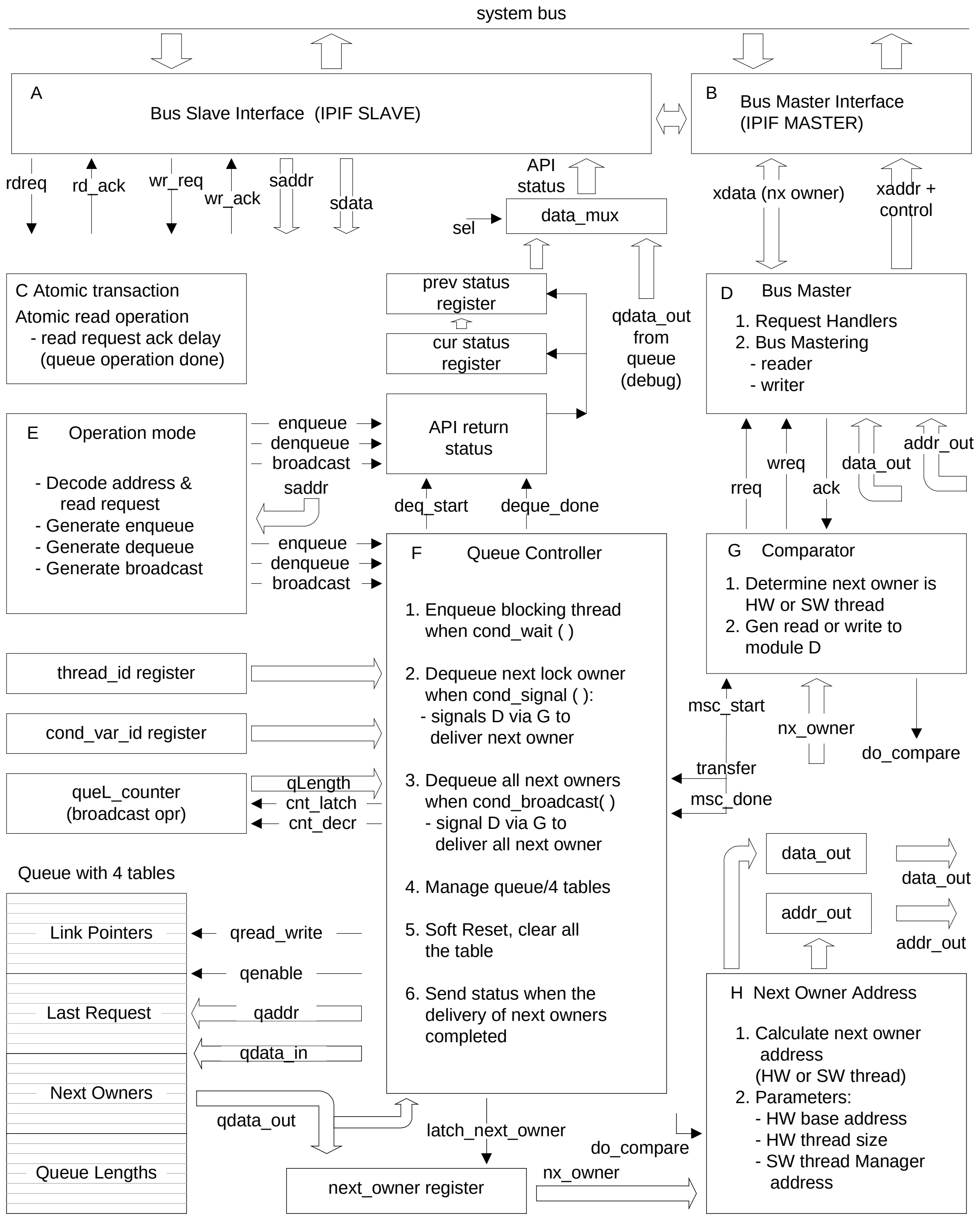


Figure 5-31 Condition Variable Hardware Architecture

Even though the de-queue operation takes at most nine cycle, the delivery operation across the bus may be delayed by other system bus activities, especially when more than one unblocked thread needs to be transferred to the software manager for the broadcast mode. The content of the Previous Status register together with next owner thread ID is returned in response to the API requests. The API has the option either to check all the information or only the busy status returned by the hardware. The hardware returns “non-busy “status, if it processes an API request. Otherwise the API should retry its request.

The Current Status changes to “busy status” when the delivery of unblocked thread starts. It is set back to “non-busy status” when the delivery completes. Every API operation, either waiting or signal or broadcast causes transfer of status from the Current Status register to the Previous Status register. However, the API request will not cause the Current Status register to change its status. Its status can only be changed by the current transaction of unblocked thread delivery that supposed to be completed.

Operation Mode Controller

This controller serves as an interface between the application program and condition variable hardware. It decodes the application program interface request and generates appropriate signals to other controllers. The type of operation requested by the application program and control signals output by this controller is given in table 5-11. It uses read request and two lines of address to decode the API requests and asserts one of these signals: COND_WAIT, COND_SIGNAL or COND_BROADCAST. The signals cause the Queue Controller to start its queuing tasks. The controller issues de-queue signal to the Queue Controller in response to either signal or broadcast API requests.

Write Request	Read Request	A13	A14	Operations	Error & Status
0	1	0	0	Read condition variable next owner (debug)	N/A
0	1	0	1	Cond_signal( ), enqueue a calling thread	Busy
0	1	1	0	Cond_wait( ), dequeue one next owner	Busy
0	1	1	1	Cond_broadcast( ), dequeue all next owner	Busy

Table 5-11 Condition Variable Application Interfaces

Output MUX (API return value)

To indicate that the API request has been processed, the controller returns the busy or not busy status along with other information as shown in Table 5-12. For a cond_wait, the next owner thread ID is returned in addition to the status if controller is not busy.

Condition Variable APIs	Return Values: bits (0 to 31)			Descriptions
	Busy status (4 bits)	Not used Zeroes (19 bits)	Next Owner thread ID (9 bits)	
cond_owner				next owner
cond_signal	1010			not busy or success
cond_signal	1110			busy
cond_wait	1010			not busy or success
cond_wait	1010			busy
cond_broadcast	1010			not busy or success
cond_broadcast	1110			busy

Table 5-12 Condition Variable API Return Codes

Atomic Transaction Controller

This duty of this controller is to acknowledge atomic read operation when a hardware thread or a software thread requests for a condition variable. This atomic read process is initiated when the read request line goes high. It does not perform immediate acknowledgement but waits for the queue operation completed. It however does not wait for completion the delivery of the next owner (in the case of broadcast or signal operation).

Global Queue and Queue Controller

The queue is divided into four tables, and is implemented within the BRAM (Queue BRAM). The tables are queue length, last request, next owner and link pointer. Except for the Link Pointer, all other tables can be access by using the condition variable ID as a reference. Link Pointer table is indexed by the thread ID. The Queue Controller responsible for three distinct tasks: queuing the condition variable next owners, de-queuing the next owner and initialize all the queue tables. The controller follows four different paths responding to different input signals: enqueue, dequeue, broadcast and soft reset. If the soft reset is active, it transitions to the reset state. While in transition, it output a signal to initialize the BRAM address counter. In the reset state, this controller initializes all the BRAM locations similar to the reset operation described in the MUTEX section.

Cond_wait initiates an en-queuing operation. An en-queuing operation starts by reading the Length Queue pointed by the condition variable ID register. If the queue length is zero, the controller continues to increment the queue length by one. Next it uses condition variable ID as an index to access both the Last Request Pointer and Next Owner Pointer, and initializes both pointers with current requester thread ID. If the queue length is non-zero, the controller has to perform several additional tasks. First it updates the queue length. Next it reads Last Request to get the index of Link Pointer. Then it writes the current requester thread ID into the Link Pointer table. It uses the index it has just retrieved as a pointer to a location in the Link Pointer table. The Link Pointers serve as a link list of all the next lock owners for a given condition variable, that will be used later by the Next Owner Pointer to find a new next condition variable owner. Finally it updates the Last Request with current requester thread ID.

Cond_signal initiates the de-queuing operation. The state machine for the de-queue operation is given in Figure 5-32. As shown in the state machine diagram, the de-queue operation has three execution paths depending on the length of the next owner in the queue. Thus all de-queue operations start with checking the queue length. The queue length is retrieved from the Queue Length table by using the condition variable ID as an index. If the queue length is zero, which means no next owner, the de-queue operation ends. If the queue length is one, controller proceeds to reduce the queue length by one. It then raises the DEQ_START signal to flag busy status. Then it de-queues the next owner from the Next Owner table into the Next Owner register and signals the HW/SW Comparator & Next Owner Address Generator to start its task, which in turn signals the Bus Master. The controller then waits for Bus Master to complete its task. Once the Bus Master has successfully delivered the next owner, it raises the MSC_DONE signal to inform the Queue Controller. The Queue Controller then return to init state and at the same time asserts the DEQ_DONE to reset the busy status. If the queue length is more than one, the controller needs to execute several more steps in addition to mentioned above. First it updates the Queue Length table with a new length. Next it de-queues the next owner, flags busy status (DEQ_START), transfers the next owner thread to the Next Owner register, and signals the HW/SW Comparator (including Next Owner Address Generator).

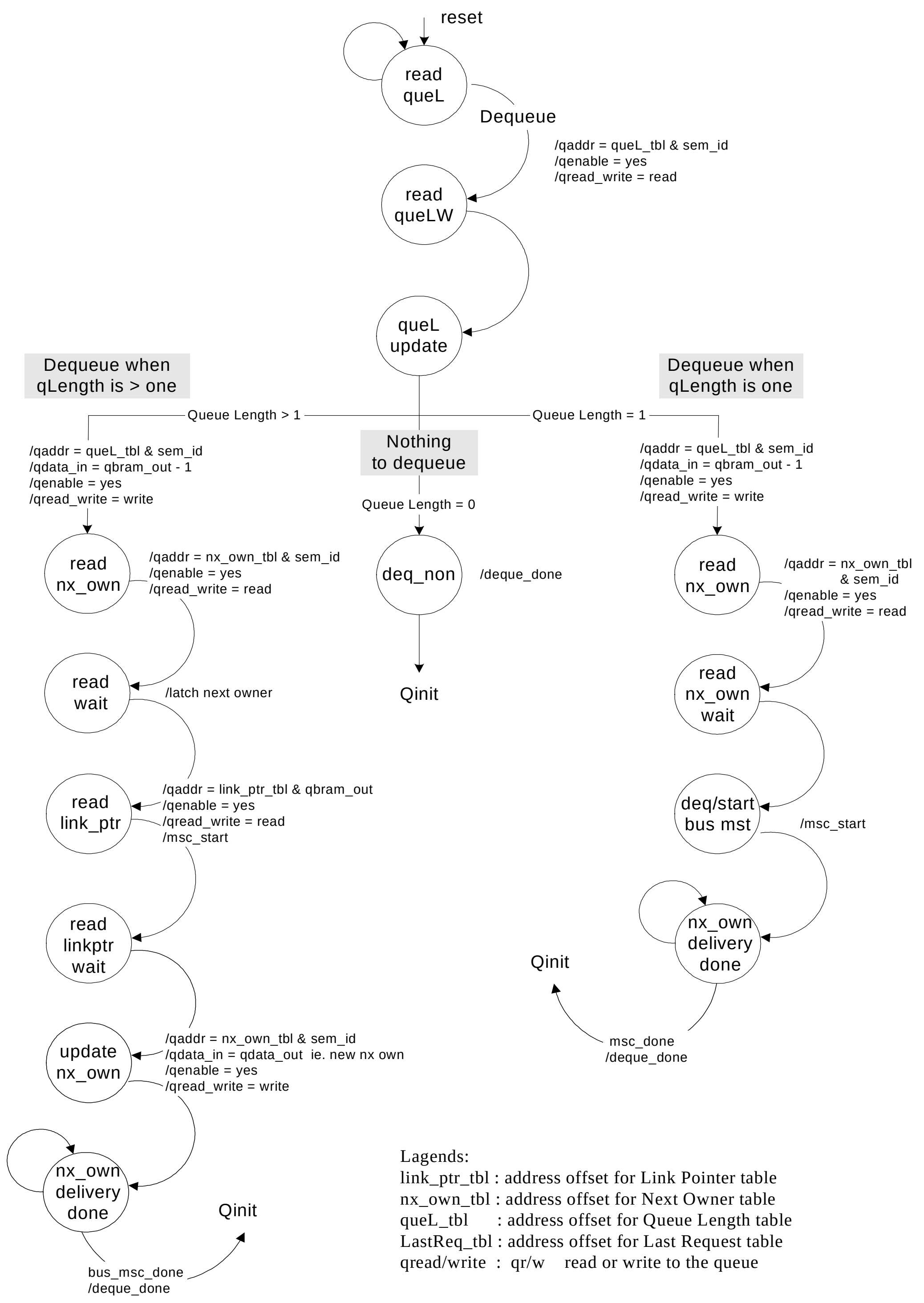


Figure 5-32 Cond_Signal State Machine

Then it needs to update the Next Owner Table with a new owner. To get a new next condition variable owner, it uses the next owner that it just retrieved as an index to reads the Link Pointer table. Then it updates the Next Owner table with the new next owner it just obtained. The Next Owner table update and the next owner delivery by the Bus Master run concurrently, as shown in the state machine diagram. When the Bus Master completes its delivery task, it raises the MSC_DONE signal to inform the Queue Controller. The Queue Controller returns to init state and at the same time issues the DEQ_DONE to reset the busy status.

Cond_broadcast initiates de-queuing all of the threads that blocked on condition variable referenced by the condition variable ID register. The state machine for the broadcast operation is given in Figure 5-33. As shown in the state machine diagram, the broadcast operation has three de-queuing execution paths depending on the length of the next owner in the queue. Thus all broadcast operation starts with checking the queue length first. The queue length is retrieved from the Queue Length table by using the condition variable ID as an index. If the queue length is zero, which means no next owner, the de-queue operation ends here. If the queue length is one, controller updates the queue length to zero. It then de-queues the next owner from the Next Owner table into the Next Owner register, and asserts DEQ_START signal to flag busy status. It then issue MSC_START signal to the HW/SW Comparator (including the Next Owner Address Generator) to start its task, which in turn signals the Bus Master to deliver the next owner.

If the queue length is more than one, the controller issues signal (CNT_LATCH) to latch the queue length into a register (queL_counter register). It then updates queue length of referenced condition variable with zero. The controller then loops to de-queue all the blocked threads using the queue length register as a counter. The controller issues CNT_DECR signal to decrement the counter each time it loops the dequeue operation. The HW/SW Comparator shown in Figure 5-34 asserts the TRANSFER signal once the dequeued thread is already transferred from the Next Owner register to Bus Master ADDR_OUT register to signal the Queue Controller that it is safe to de-queue another thread into the Next Owner register, thus allowing de-queue operation to run in parallel with delivery operation. The shaded states in broadcast state machine and HW/SW Comparator state machine diagrams are to show that the de-queue operation and delivery of last de-queued thread execute in parallel.

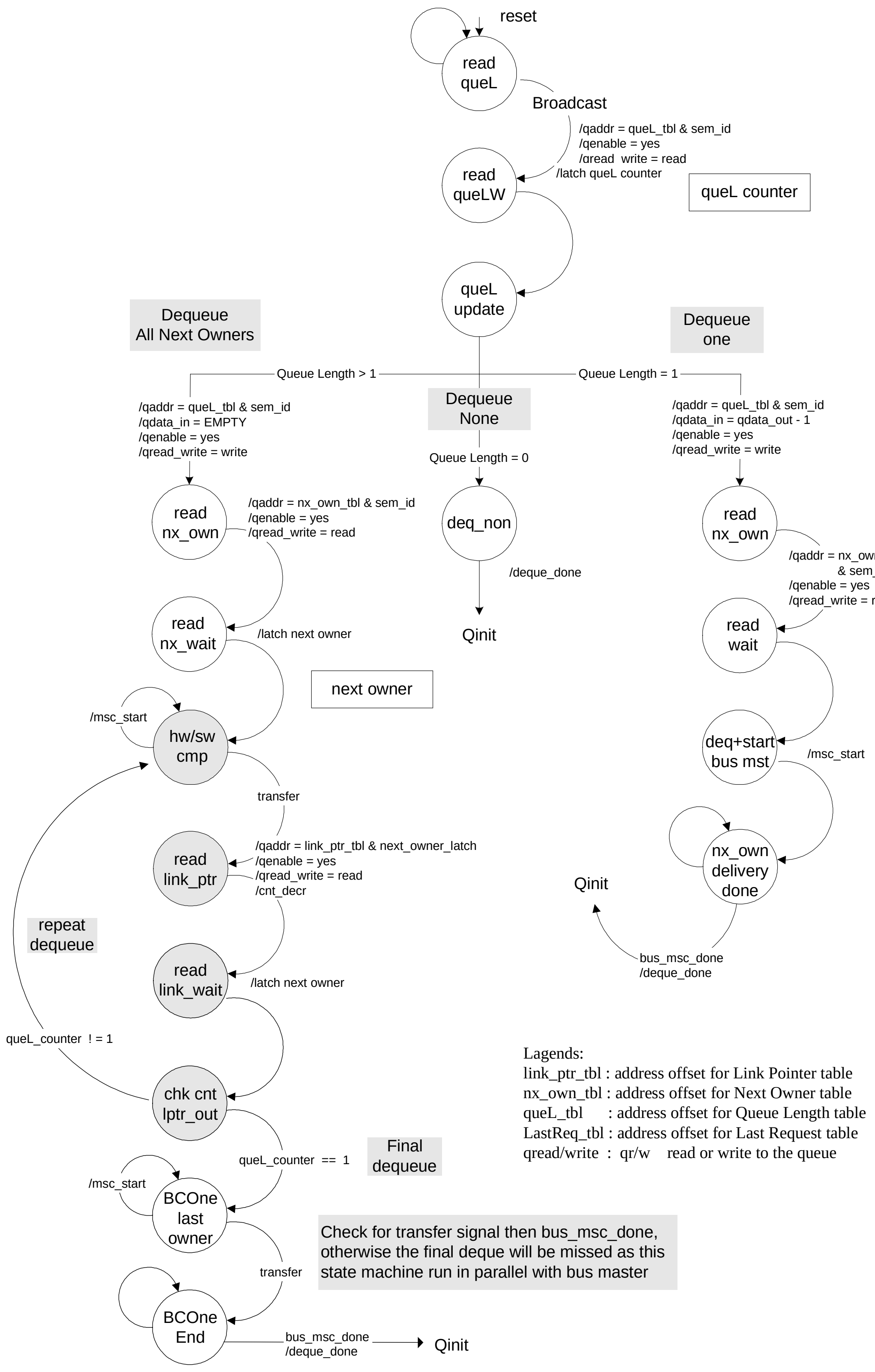


Figure 5-33 Cond_Broadcast State Machine

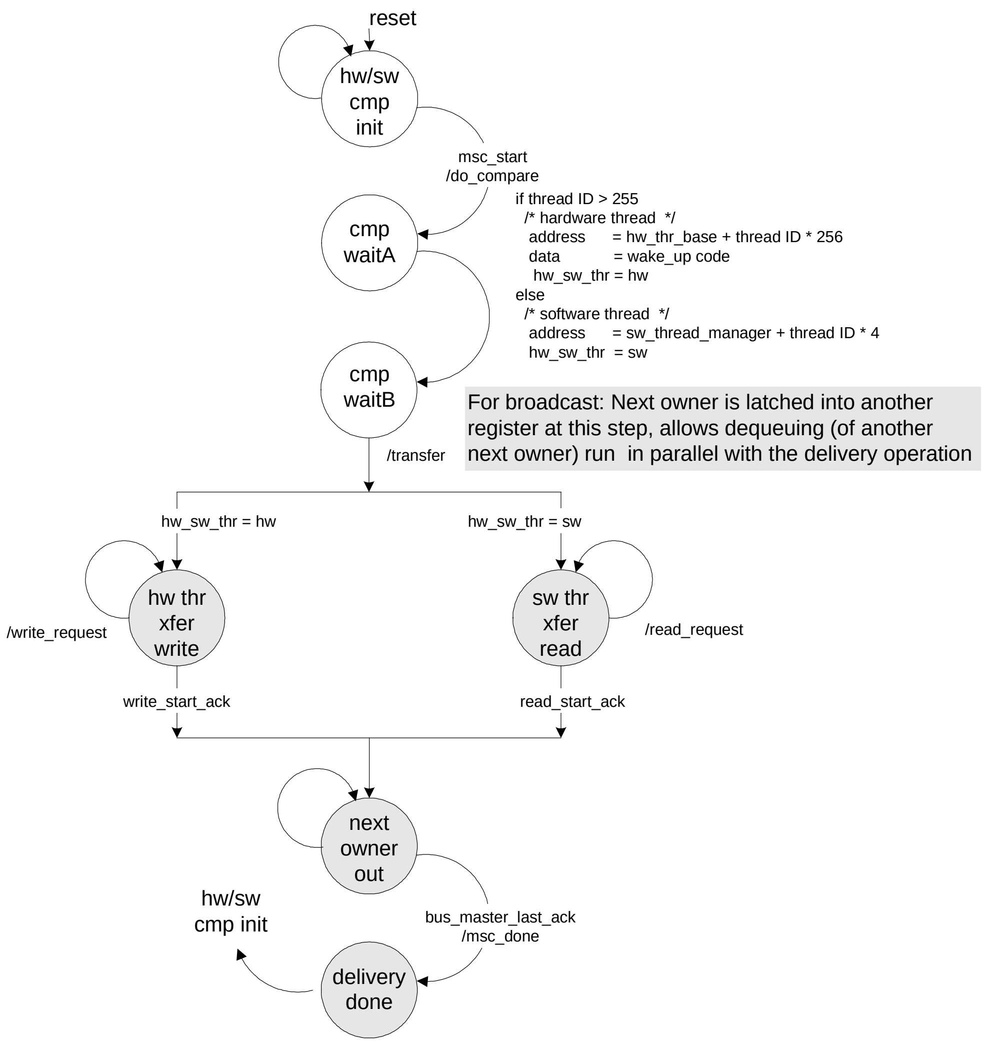


Figure 5-34 HW/SW Comparator & Next Owner Address Generator

Once the last thread is successfully delivered, the controller asserts DEQ_DONE to reset the busy status register (current status register) to NOT_BUSY state and returns to init state. The Bus Master and HW/SW Comparator operations are similar to those described in the MUTEX section (Section 5.8.1) except for broadcast operation described above.

6 HYBRID SYSTEM CORES INTEGRATION AND TEST

6.1 INTRODUCTION

This chapter describes the testing performed to verify the functionality and performance of all synchronization cores, as well as hardware thread cores. Figure 6-1 shows the block diagram, along with address ranges, of the specific cores included in our test system. At this point it is worth outlining how address ranges are determined and provided to the hardware cores that need to associate an address with a thread. This is important as we currently assign a thread id value to a hardware thread based on the address offset of its command register from the starting base address of the hardware thread address range, as address ranges for cores must be assigned during the initial system design. To allow hardware threads to access the synchronization cores, the base addresses of the synchronization cores are passed as VHDL generic parameters during instantiation of the hardware thread cores.

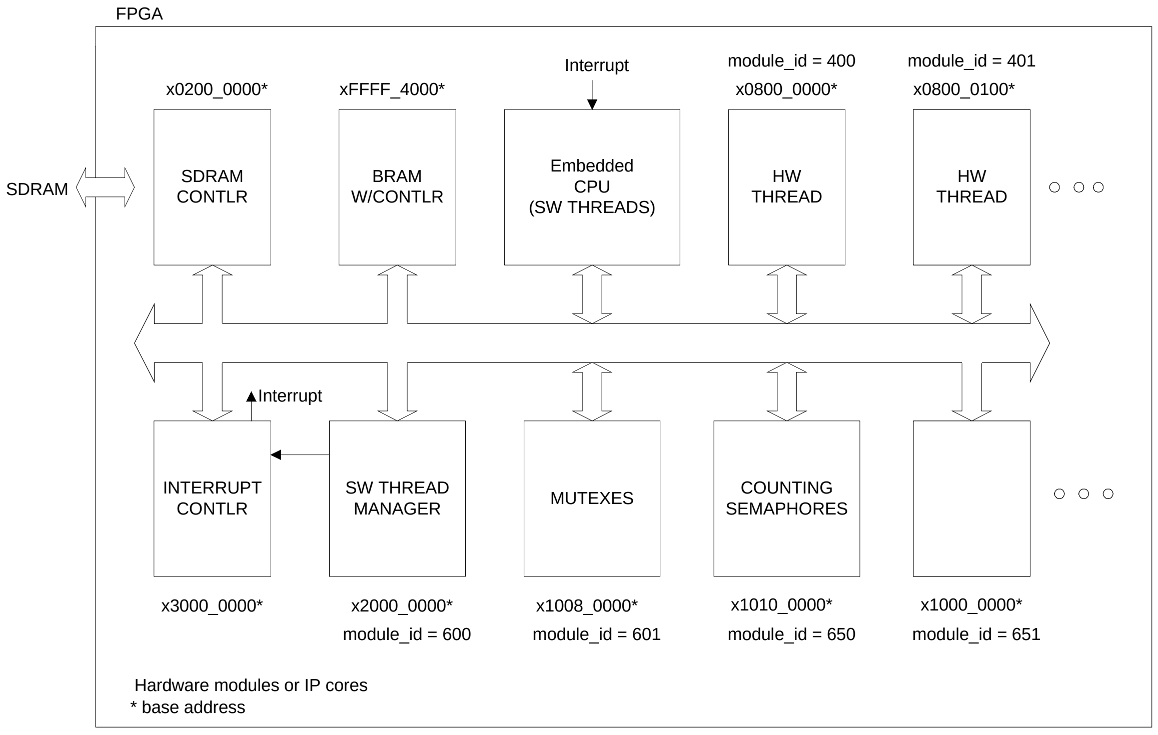


Figure 6-1 Single FPGA Chip with Embedded CPU and Other Cores

For the blocking synchronization cores such as MUTEXES, semaphores and condition variables, the start address of the hardware thread cores and the software thread manager are also passed as VHDL generic parameters. The starting addresses enable the synchronization cores to transfer unblocked threads to appropriate destinations. If the unblocked thread is a hardware thread, the synchronization cores will write a wake-up command code to the hardware thread command register. Within each synchronization core, the addresses of a hardware threads command register is calculated by adding the start address of the first hardware thread core with the product of thread ID and the size of the hardware thread interface component. Figure 6-2 shows an example memory map that includes two hardware threads. In the memory map, the start address of the first hardware thread interface component is set to 0x0800_0000 and passed as a generic parameter during the instantiation of the synchronization cores. To improve system performance, the memory map is arranged such that the SDRAM can be cached. An example of calculation performed by the semaphore core to deliver the unblocked thread (or next semaphore owner) to appropriate destination address is as follows:

```
Semaphore Core:
If the unblocked thread is a software thread, it will be delivered to the scheduler ready queue in
the Software Thread Manager core:

If unblocked thread is a HW Thread:
 Say the unblocked thread is hardware thread number 2 (Thread ID 262)
 Destination address:
   = HW Thread Start Address + (HW Thread ID x HW Thread Size) + Command Register Offset
   = 0x0800_0000 + ( 2 * 0x100 ) + 0x5

If unblocked thread is SW Thread:
 Say the software thread number is 4 (Thread ID 4)
 Destination address:
  = SW Thread Manager + Add Register Offset + 0x4 << 2
  = SW Thread Manager + Add Register Offset + 16
  = 0x3000_0000 +  0x100 + 0x10

For the hardware thread core (HW Thread) the address encoding process to acquire or release a
semaphore can be summarized as follows:
HW Thread Core:
Say HW Thread 5 (Thread ID 260) makes a request for semaphore 3
Encoded address generated by the HW Thread:
 = Operation Code + Semaphore Base Address + (Thread ID << 8 ) + (Semaphore ID << 2 )
 = Operation Code + 0x1010_0000 + ( 0x105 << 8 ) + ( 0x3 << 2 )
 = 0x20000 + 0x1010_0000 + 0x10500 + 0xC     ( sem_post operation )
 = 0x40000 + 0x1010_0000 + 0x10500 + 0xC     ( sem_wait operation )
```

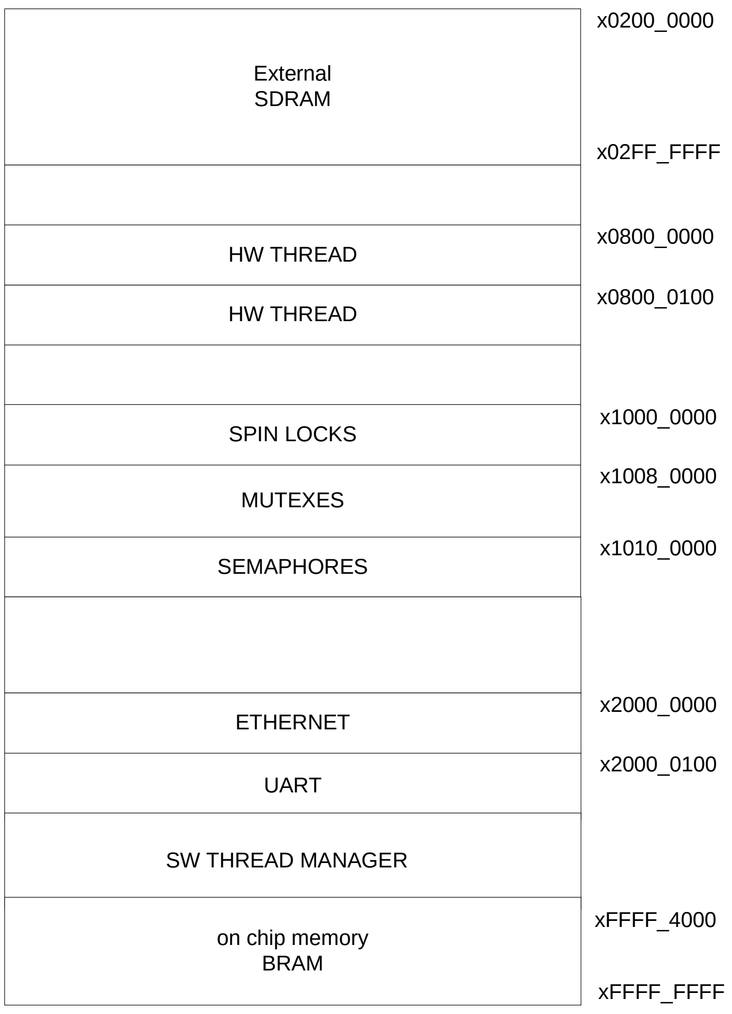


Figure 6-2 An Example of Memory Map of a Hybrid Thread System

6.2 INDIVIDUAL CORE FUNCTIONAL TESTS

We have performed a variety of stress tests to validate the functionality of all synchronization cores under various scenarios of concurrently executing software and hardware thread loads. The tests include semantic verification of hybrid threads competing for locks, queuing blocked threads, and associated unblocking operations. The unblocking tests involved invocation of interrupts and deliveries of unblocked threads to the CPU scheduler queue.

Functional Test Set-up:

Hardware required for these tests include each synchronization core, timer, interrupt module, reset module, and CPU. The test program on the CPU include timer interrupt handler and reset interrupt handler. The timer interrupt handler contained a counter that incremented at every interrupt. At different counter values different testing tasks are assigned, for example requesting or releasing several semaphore variables. The different sequence of tests performed for each core are summarized as follows:

Spin locks:

1. Acquire free spin locks from hardware and software threads.
2. Release “owned” spin locks.
3. Try to acquire “owned” locks will not cause ownership change.
4. Acquire same spin locks recursively.
5. Release same spin locks recursively.
6. Core soft reset to initialize recursive counters and lock owner registers.
7. Repeat above tests after soft reset.

MUTEXES:

1. Acquire free MUTEXES from hardware and software threads.
2. Release of MUTEXES.
3. Acquire the same MUTEXES recursively causing recursive counter to be incremented.
4. Release of the same MUTEXES recursively causing the counter to be decremented.
5. Acquiring of “owned” MUTEXES causing en-queue of software and hardware calling threads
6. Repeated acquisitions of “owned” MUTEXES causing queue sizes to grow accordingly. Capacity testing of maximum number of threads
7. Release of MUTEXES that have blocked threads in queue, causing de-queuing of blocked threads. If the unblocked threads are hardware-based, wake-up command codes will be delivered to the hardware thread command registers. If the unblocked threads are CPU based threads, read operations to appropriate locations of Software Thread Manager will be performed.
8. Repeated release of MUTEXES cause de-queuing of threads in appropriate order.
9. Core soft reset to initialize MUTEX owner registers, recursive counters and global queue.
10. Repeats tests 1 to 8 after performed the reset.

Semaphores:

1. Semaphore wait operation on zero semaphore resource (counter is zero) causes the calling thread to be queued and the counter remains zero.
2. Semaphore wait performed when the semaphore counter is not zero causes the counter to be incremented by one.
3. Semaphore post when the semaphore counter is not zero causes the semaphore counter decremented by one.
4. Semaphore post when the semaphore counter is zero causes a thread to be removed from the semaphore queue (if there is one in the queue). The de-queued thread will be delivered to either the Software Thread Manager or Hardware threads.
5. Consecutive semaphore waiting when the semaphore counter is zero causes queuing all the calling threads and counter remains zero. Followed by consecutive semaphore release operations that will cause de-queuing of threads and counter remains zero and incremented when no more thread in the queue.
6. Proper initialization of semaphore counters, core soft reset operations.
7. Repeat tests 1 to 6 after performed the core soft reset.

Condition variables:

1. Condition wait operations cause the calling threads to be queued
2. Condition signal causes a thread in the queue to be removed (if there is one in the queue)
3. Condition broadcast causes all the blocking threads to be de-queued and delivered to appropriate destinations.
4. Consecutive wait calls cause queuing of the calling threads. Consecutive signal calls cause removal of threads from the queue.
5. Proper execution of its operation to response to different sequences of wait, signal and broadcast calls.
6. Perform core soft reset and repeat test 1 to 5 after soft reset.

In addition, each synchronization cores is subjected to regression tests of a system with 250 software threads that generate more than 100,000 events in each test. The test scenarios are summarized as follows:

Mutex & Spin Lock Cores:

This test involved 250 software threads competing to acquire a mutex. The first thread that attempts to lock the mutex owns the lock, and blocks the other 249 threads (no blocking and queuing in the case of spin lock). The test starts with the main thread creates 250 children and then performs thread_join on all its children. Each created child loops attempting to acquire the lock, followed by yield, unlock and yield again. Mutex unlocking by the current owner causes the next thread in the queue to wake-up and own the mutex. Observe that blocked threads cannot print their thread ID and make any new lock request.

Semaphore Core:

In this test, the main thread creates 125 consumer threads and 125 producer threads. The main thread then performs join to suspend itself and wait for its children to exit. Each created consumer performs wait say on semaphore S1 and then yields. Since S1 is initially at zero, all consumer threads go to sleep into the S1 queue. Each created producer thread loops to perform post on semaphore S1, and then yields. Each semaphore post operation awakes one consumer thread. Active consumer thread prints its thread ID to indicate it is now running.

Condition Variable core:

- The main thread creates a lock or mutex, a variable, two condition variables (CV1 and CV2), 125 worker threads + 125 dispatch threads, and then performs join on all its children. The variable is to represent the number of jobs available in a bounded buffer. The lock is used to protect the variable. The condition variable, CV1 enables worker threads to sleep when the buffer is empty. The other condition variable, CV2 is employed to block dispatcher threads when the number of job reaches a certain number (arbitrary number that normally matches the size of the job buffer say ten).
- Each created worker thread loops to acquire the lock and check the buffer. If the buffer is empty, it goes to sleep by calling condition wait on CV1. When a worker thread awakens, it checks the buffer, removes a job from the buffer if it is not empty, performs condition signal on CV2, releases the lock, and yields.
- Each created dispatcher thread loops to acquire the lock and checks the number of jobs available in the buffer. If the number of jobs in the buffer is ten, the dispatch thread goes to sleep by calling condition wait on CV2. If the number of job is less than ten, it adds one job into the buffer, performs signal on condition variable CV1, releases the lock, and yields. An awakened dispatcher thread checks the number of job in the buffer, adds a job

if the number of job is less than ten and then performs condition signal on CV1. Then it releases the lock and yields.

Hardware Thread Cores:

Tests on the hardware threads can be divided into functional tests and performance tests. The functional tests include proper working of the hardware thread controller to access multiple synchronization variables and memory locations. The synchronization variables and memory are accessed by means of procedures or APIs. The functional tests can be summarized as follows:

1. Acquire spin locks, MUTEXES, semaphore.
2. Release of spin locks, MUTEXES, semaphores.
3. Memory accesses either read or writes.
4. Blocking operations when fail to gain synchronization variables. Unblocking operation when semaphores write wake-up codes.
5. Competition with other hardware and software threads to gain synchronization variables.

6.3 PERFORMANCE EVALUATIONS

The performance tests on the cores include performance evaluations of hardware threads against software threads competing for synchronization resources. Each hardware thread and software thread runs individually to establish base line performances as shown in table 6-1. Both the hardware thread and CPU are clocked at 100 MHz. The result indicates that hardware thread is about six times faster than the software thread in acquiring a spin lock.

Time in seconds	Lock Access Count by HW Thread (HW)	Lock Access Count by SW Thread (SW)	(HW)/ (SW)
6	7456497	1183568	6.30
12	14927918	2369504	6.30
18	22399315	3555435	6.30
24	29870751	4741374	6.30
30	37342162	5927308	6.30
36	44813614	7113248	6.30
42	52285041	8299185	6.30
48	59756442	9485118	6.30
54	67227884	10671057	6.30
60	74699301	11856992	6.30
66	82170757	13042933	6.30

Table 6-1 Baseline HW Thread vs. SW Thread

Then both threads competed to acquire a spin lock. The hardware thread access is delayed gradually to study the effects of competition. The effect of competition is shown in Figure 6-3.
As indicated in the graph, when the hardware thread is not delayed, hardware thread dominates the lock access and the ratio can be as high as twenty (Hardware thread gaining lock twenty times more than that of software thread).

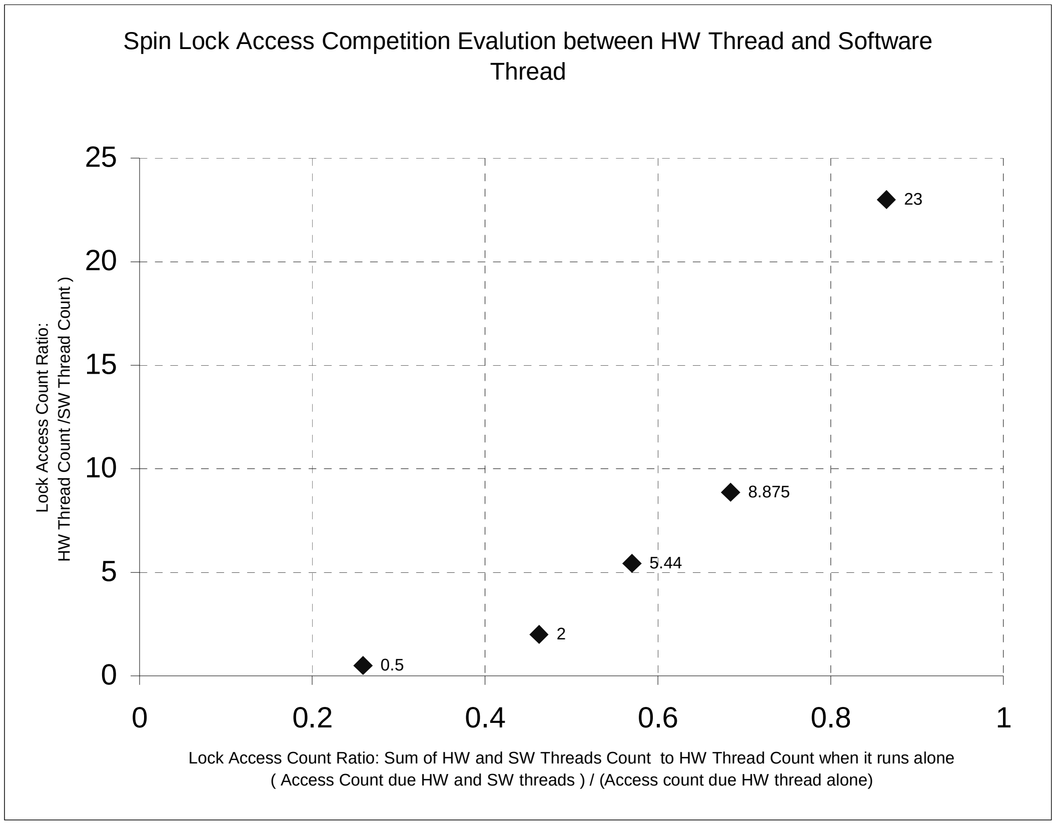


Figure 6-3 Hardware Thread vs. Software Thread

We have conducted performance tests on our mutex or lock cores with various loads of software threads running concurrently on PPC405 CPU. The mutex cores are clocked at 100 MHz, the maximum clock speed for our FPGA logic and the CPU is operated at 300 MHz. The number of lock acquisitions for each load of threads in the system is about 100,000 events. As depicted in Figure 6-4 and Figure 6-5, depending on the mode of CPU cache, the average time required for requesting a lock is about 75 or 59 clock cycles respectively. With the CPU cache turned on, the lock access times are mostly at 580ns but can be as high as 790ns when the cache miss occur.

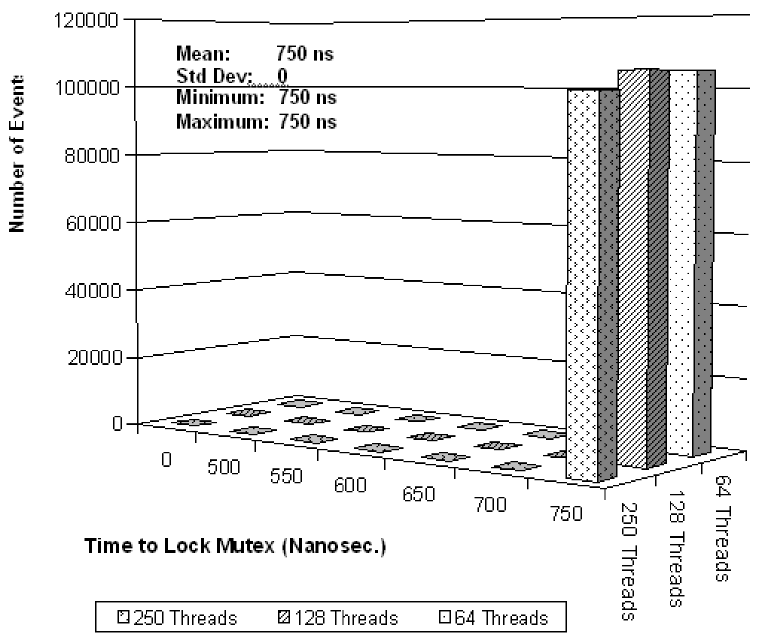


Figure 6-4 Mutex Access Speed (CPU Data Cache Off)

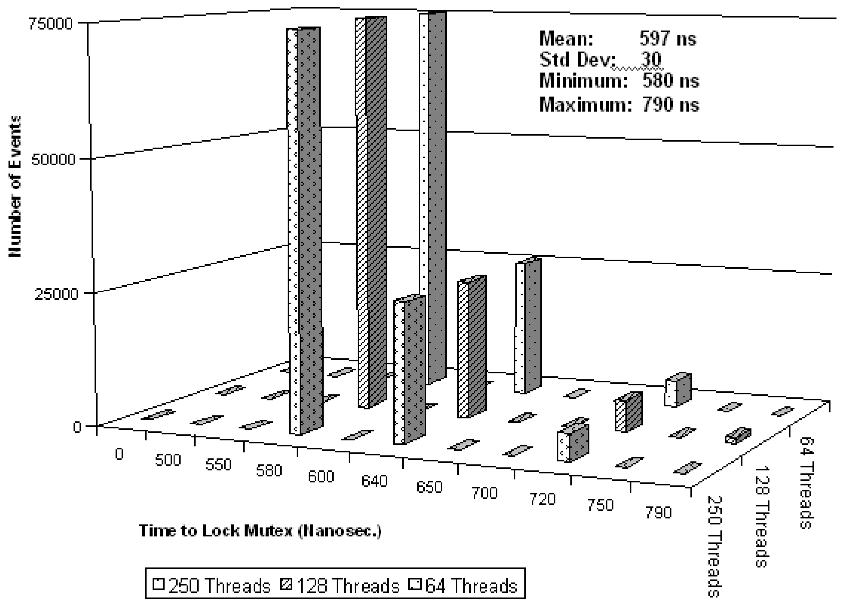


Figure 6-5 Mutex Access Speed (CPU Data Cache On)

The access times for different synchronization API operations are given in Table 6-2. The total clock cycles for each operation is defined as the time taken when the internal operation within the core starts and excludes the time required to issue a request from either the CPU or the Hardware threads. The issue request time for these tests is excluded in order to eliminate the time difference that exists between a CPU and Hardware thread performing bus requests. The bus transaction is either for acknowledgment or for the bus master within the core to perform a bus operation either to read the Software Thread Manager to deliver an unblocked thread or write wake-up command to a hardware thread. If we define synchronization latency as the time to acquire a synchronization variable in absence of contention and synchronization delay, then this time can be measure as the time between request and acquisition acknowledge. For a MUTEX variable, the latency and delay are 11 and 23 clock cycles respectively (measured from the moment the core receives a request and generates an acknowledgement).

Synchronization APIs	Internal Operation (clk cycles)	Bus Transaction after the Internal Operation starts (clk cycles)*	Total Clock Cycles
spin_lock	8	3	11
spin_unlock	8	3	11
mutex_lock	8	3	11
mutex_trylock	8	3	11
mutex_unlock	13	10	23
sem_post	9	10	19
sem_wait	6	3	9
sem_trywait	6	3	9
sem_init	3	3	6
sem_read	6	3	9
cond_signal	11	10	21
cond_wait	10	3	13
cond_broadcast	6n	10n	16n

Table 6-2 Cores Access Speed

6.4 CORES HARDWARE RESOURCES:

This section summarizes the hardware cost of implementing our synchronization cores on a XILINX VIRTEX V2P7. The V2Pro7 resources include 4928 slices and 44 blocks of distributed RAM (BRAM). The hardware resource to implement hardware thread interface which includes the thread state controller, synchronization and bus master components is about 3 percent of total slices available on V2P7. The different type of resource needed to implement one hardware thread interface is given on Table 6-3.

Resources Types	Resources Used	Total Resources on chip	% Used
Slices	128	4928	3
Flip-flop	153	9856	2
4 -input LUT	205	9856	2
BRAMs	0	44	0

Table 6-3 Hardware cost for hardware thread interface

The cost of FPGA hardware to implement sixty-four recursive spin locks core is about 2.5 of total hardware resource available on FPGA as shown in Table 6-4. The single BRAM is used as 64 spin lock owner registers and 64 recursive counters.

Resources Types	Resources Used	Total Resources on chip	% Used
Slices	123	4928	2.5
Flip-flop	80	9856	0.8
4 -input LUT	215	9856	2.2
BRAMs	1	44	2.3

Table 6-4 Hardware cost for 64 Spin Locks (excluding bus interface)

The hardware resource required to implement sixty-four MUTEXES core is given in Table 6-5. One BRAM is used as a queue to hold up to five hundreds and twelve sleeping threads (hardware or software threads). The second BRAM is used as MUTEX owner registers and recursive counters. The resource also includes the controller to de-queue and deliver the wake-up threads either to the scheduler queue or hardware threads.

Resources Types	Resources Used	Total Resources on chip	% Used
Slices	189	4928	3.8
Flip-flop	134	9856	1.4
4 -input LUT	328	9856	3.3
BRAMs	2	44	4.5

Table 6-5 Hardware Cost for 64 MUTEXES (excluding bus interface)

The FPGA hardware resource to implement sixty-four semaphores is given in Table 6-6. The semaphore entity supports *sem_wait*, *sem_trywait*, *sem_post*, and *sem_count_init* operations similar to POSIX API. The semaphore queue is sized to hold up to five hundreds and twelve sleeping threads (hardware or software threads). The resource also includes the controller to de-queue and deliver the wake-up threads either to the scheduler queue or hardware threads.

Resources Types	Resources Used	Total resources on chip	% Used
Slices	229	4928	4.6
Flip-flop	186	9856	1.9
4 -input LUT	414	9856	4.2
BRAMs	2	44	4.5

Table 6-6 Hardware Cost for 64 Semaphores (excluding bus interface)

The cost of FPGA hardware to implement sixty-four condition variables (CVs) is given in Table 6-7. The CV has a queue that is sized to hold up to five hundreds and twelve sleeping threads (hardware or software threads).

Resources Types	# Used	# total on chip	% used
Slices	137	4928	2.8
Flip-flop	136	9856	1.4
4 -input LUT	231	9856	2.3
BRAMs	1	44	2.3

Table 6-7 Hardware Cost for 64 CVs (excluding bus interface)

Table 6-8 summarizes hardware cost to implement different types of synchronization. Table 6-8 also indicates the number of slices needed to implement one synchronization variable of each type. For example one spin lock variable requires only 1.9 slices only, while one semaphore variable costs about 3.6 slices and .07 percent of BRAM. As the capacity of BRAM is not fully utilized in the current design, each synchronization core can be expanded to support up to 512 variables (total 2048 synchronization variables) without additional cost except three more address lines are needed. Table 6-9 shows hardware cost in term of slices needed to implement one synchronization variables for each type.

Synchronization Type	Total Slice for 64 synchronization variables	Number of Slices for each synchronization variable
Spin lock	123	1.9
Mutex	189	3.0
Semaphore	229	3.6
Condition Variable	137	2.1

Table 6-8 Hardware Cost for 256 Synchronization Variables (excluding bus interface)

Synchronization Type	Total slices for 512 synchronization variables	Number of slices per synchronization variable
Spin lock	123	0.2
Mutex	189	0.4
Semaphore	229	0.4
Condition variable	137	0.3

Table 6-9 Hardware Cost for 2048 Synchronization Variables (excluding bus interface)

7 HYBRID THREAD APPLICATION STUDY

7.1 INTRODUCTION

This chapter presents an application study of our hybrid multithreaded model. We have implemented several image-processing functions in both hardware and software from within our common multithreaded programming model on a XILINX V2P7 FPGA. The transforms were first implemented as software threads that communicated using our synchronization primitives running on the PPC 405 processor core. Then the software threads that performed the transforms were recoded in VDHL and implemented within the FPGA still using our programming model and synchronization primitives. This example demonstrated hardware and software threads executing concurrently using standard multithreaded synchronization primitives. The application threads transformed real-time images that were first captured by a camera connected to our host workstation, and then the results were displayed back on the workstation. All communications between the V2P7 and the host workstation were across Ethernet. In both the software and hardware implemented transform test cases, a communications Ethernet driver thread ran in software. The driver thread communicated with the transform threads using our synchronization primitives. Both hardware and software threads synchronized their access to shared data through our standard API's. Our hardware thread application interface enables application developers to write applications in VHDL without going into the details of the system bus architecture. Thus, our current hybrid thread programming model can reduce development time, and opens the door for software engineers to access the reconfigurable logic through a familiar POSIX like generalized software multi threaded programming model.

7.2 IMAGE TRANSFORMATION

Filtering is an example of transformation process that can be applied to images. Filtering may remove noise, enhance details, or blur image features depending on the selected transform algorithm. Examples of filter used for smoothing in the spatial domain include median filters, binomial average filters, and Gauss kernel filters. A spatial filter replaces each pixel within an image with a new value that is produced by a function with inputs coming from itself and its neighbors.

The spatial filter algorithm defines contributing neighbors by a mask. Figure 7-1 shows an example of a 3x3 mask kernel for a binomial average filter. The values in the mask are the weights (w_i) applied to each pixel (p_i) in the average when the mask is centered on the pixel being transformed.

$$\frac{1}{16} \times \begin{array}{|c|c|c|} \hline 1 & 2 & 1 \\ \hline 2 & 4 & 2 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$$

$$p_{i,j} = \frac{1}{16}\begin{bmatrix} w_{i-1,j-1} * p_{i-1,j-1} + w_{i,j-1} * p_{i,j-1} + w_{i+1,j-1} * p_{i+1,j-1} \\ + w_{i-1,j} * p_{i-1,j} + w_{i,j} * p_{i,j} + w_{i+1,j} * p_{i+1,j} + \\ w_{i-1,j+1} * p_{i-1,j+1} + w_{i,j+1} * p_{i,j+1} + w_{i+1,j+1} * p_{i+1,j+1} \end{bmatrix}$$

Figure 7-1 Binomial 3x3 Mask Kernel

Binomial filters generally reduce noise in an image by replacing each pixel with the binomial weighted average of itself and neighboring pixel values. Figure 8-1 shows a 3x3 binomial kernel example represented by 1/16 [1 2 1 2 4 2 1 2 1]. The Binomial filter is an example of a general linear filter, as the function is the weighted average of the pixels in the mask. In contrast, a median filter is a non-linear filter, as the median cannot be obtained from a linear combination of the pixels under the mask.

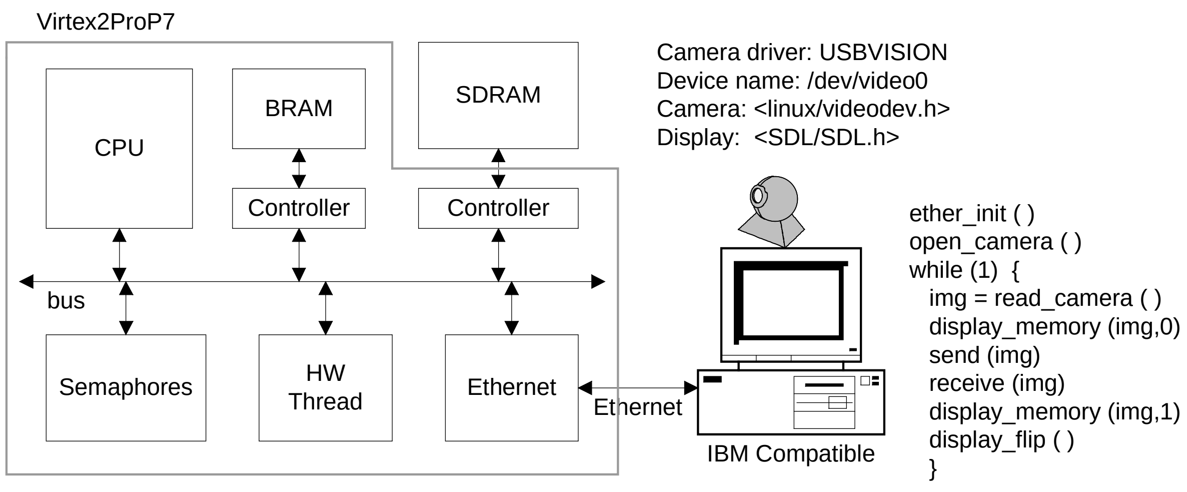
7.3 EXPERIMENT SET-UP

We developed the following experimental setup to verify our multithreaded models capability to support concurrent execution of both hardware and software threads communicating and synchronizing using our standard shared memory synchronization protocols. We implemented several simple image transforms using the experimental set-up illustrated in Figure 7-2. A camera attached to a PC running LINUX was used to capture real time pictures of moving objects. The image frames were then transferred from the PC to V2P7 FPGA board via a dedicated Ethernet link. After each frame had been processed on the V2P7 board, the modified image was then sent back to the PC. Both the original and modified images were then displayed on the PC real-time.

A software thread was created on the embedded PPC405 CPU to receive frame data from the Ethernet and place it on the heap (in SDRAM). We used two counting semaphores to synchronize the CPU resident software and the FPGA resident hardware threads. Semaphore S1 synchronized accesses to shared image data not yet processed, while semaphore S2 serialized access to processed images.

An initialization software thread first ran on the PPC 405. The C program listing for the initialization routine is given in Figure 7-3. The initialization routine performed two memory allocations on the heap to get two pointers for storing image data, initialized the Ethernet link and called a hardware thread create API. The hardware thread create API provided the two address pointers to the hardware thread through the hardware thread interface’s argument registers. The hardware thread create also resulted in the transition of the hardware thread’s controlling state machine from the idle state to the run state.

The software thread then waited for image to arrive from the Ethernet link. When a new image was available, the software thread transferred the received image into the SDRAM location pointed to by the first pointer. The software thread then performed a semaphore post on S1 to communicate to the hardware thread that the image to be processed was available on the heap. The software thread then executed a wait on semaphore S2.



Camera driver: USBVISION
Device name: /dev/video0
Camera: <linux/videodev.h>
Display: <SDL/SDL.h>

```
ether_init ( )
open_camera ( )
while (1)  {
   img = read_camera ( )
   display_memory (img,0)
   send (img)
   receive (img)
   display_memory (img,1)
   display_flip ( )
   }
```

Figure 7-2 Hybrid Thread Image Processing

```
{
//Initialize Ethernet link
ether_init( );

//Raw image data pointer
address1 = malloc(image_size)
// Processed image data pointer
address2 = malloc(image size)

img->in   = address1;
img->out = address2;

//Hardware thread create API
hw_thread_create(address1, address2, algorithm )

while (1) {
   // Get image from ethernet
          receive(img->in, img_size)
    // Let hw thread know image data is available
         sem_post( S1 );
    //Wait for hw thread finish processing
        sem_wait( S2 );
   // Send processed image
        send(destination, img->out, img_size);
    }
 }
```

Figure 7-3   Software Thread on CPU (part of C program)

Upon receipt of the semaphore post (S1) operation from the software, the semaphore IP wrote a wake-up command to the blocked hardware thread (a write to the hardware thread command register). The hardware thread awoke, read the image from the heap, performed the image processing specified within the thread, and wrote the processed image back into heap at the location pointed to by the second pointer variable. When the hardware thread finished processing all the image pixels, it performed a post on semaphore S2 to signal the completion of the image frame to the software thread. The posting on semaphore S2 caused the sleeping software thread to wake and initiate the transfer of the processed image frame back to the workstation for display.

This section describes the organization of the hardware user thread used during the experiment. The hardware thread user component consists of – a control unit and a filter data path. The control unit synchronizes with the software thread and accesses the external global memory through the hardware thread interface. Addresses for the global data contained in the external SDRAM are generated by the control unit. For several of our simple image processing functions such as gray scale inversion or threshold, the control unit is represented by the following five states: 1) semaphore wait, 2) read, 3) waiting for process to complete, 4) write and 5) semaphore post. The pseudo code for these five states is shown in Figure 7-4. The VHDL code for this hardware thread pseudo code that use API is shown in Figure 7-5.

```
if command == run                        // hardware thread in run mode
 {
  SW:   sem_wait(S1)                     // perform semaphore wait operation
        pixel_count = 0                  // initialize number of pixel to be processed
  RD:   read data (address1)             // get image from main memory
  WT:   process wait                     // wait for data path to transform image
  WR:   write data(address2)             // store processed image to main memory
        address1 = address1 + 4          // memory address to read image pixel
        address2 = address2 + 4          // memory address to write processed image to
        pixel_count = pixel_count + 4    // four bytes transferred for each memory access
        if (pixel_count == image_size)   // image_size = image_width * image_height
           go to  RD:
  SP:   sem_post (S2)                    // perform semaphore post operation
        branch SW:
  }
```

Figure 7-4   Hardware Thread Control Pseudo Code

```
-- ** Control Unit to Invert Image ** --
when sema_wait =>                          -- semaphore wait
   semw1: sem_wait(usr_base_addr, S1, usr_operation);
   if  thread_status = SEMA_WAIT_OK then
        reset_count    <= ‘1’;             -- to reset pixel_count
       next state        <= inv_read;
   end if;

when inv_read =>                           -- read pixels from memory
   inv_rd: read_data(usr_base_addr, buf_in_pointer, usr_operation);
   if  thread_status = READ_OK then
        start_process  <= START;           -- signal to start perform transform task
        next state        <= inv_filter;
  end if;

when inv_filter =>
   --if img_process = DONE then            -- receive signal that transform completed
        next_state <= inv_write;
   --end if;

when inv_write =>                          -- write processed pixel (invert_out) to memory
   inv_wr: write_data(usr_base_addr, buf_out_pointer, usr_parameter, invert_out);
   if  thread_status = WRITE_OK then
       update_count <= ‘1’;                -- to increment pixel_count
       next state        <= inv_done;
  end if;

when inv_done =>                           -- last pixel to be processed in an image frame
if pixel_count = image_size then
   next_state <= sema_post
else
   next_state <= inv_read
end if;
```

```
when sema_post =>
   semp2: sem_post(usr_base_addr, S2, usr_operation);    -- semaphore post operation
   if  thread_status = SEM_POST_OK then
       next state        <= sem_wait;
   end if;
```

Figure 7-5   Hardware Thread Control Unit (VHDL Code)

The data path consists of the logic for transforming data to the desired output. For the gray scale inversion and threshold filters, the data paths are given in Figure 7-6 and Figure 7-8 respectively. For performance comparisons, we have also implemented our targeted image algorithms in C. Part of the code for both filters running on software is shown above in Figure 7-7 and Figure 7-9 respectively.

```
-- ** Invert Image Data Path Component ** --
-- Counts number of pixel within an image frame
-- buf_in_pointer is memory address from where the pixel to be read from
-- buf_out_pointer is memory address to where the processed image to be written
-- Both addresses initialize with values obtained from usr_argument registers.
-- Software writes initial addresses into usr_argument registers during hw_thread creation API.
counter: process( clock ) is
begin
   if  clock = ready then
       if reset = yes or start = yes  then
            pixel_count <=  zero;
            buf_in_pointer   <= unsigned(usr_argument1);
            buf_out_pointer <= unsigned(usr_argument2);
       elsif update_count = yes then
            pixel_count <=  pixel_count + 4              // 4 byte or 4 pixels for each I/O operation
            buf_in_pointer   <= buf_in_pointer + 4;
            buf_out_pointer <= buf_out_pointer + 4;
       end if;
    end if;
end process counter;

--Inverts image pixels. The pixel is read from memory and stored in register usr_ret_data
--Inverted pixel is stored in invert_out register before transferred to memory
--N is image depth, intensity of each pixel, if N = 8 then 255 is white and 0 is black
Invert_Filter: process( clock ) is
begin
    if clock = ready then
        if reset = yes then
            invert_out <= zero;
        elsif start_process = START then    -- start process, 4 pixels at a time
            invert_out(0 to N-1)          <=  max_value – usr_ret_data(0 to N-1);
            invert_out(N to N*2-1)     <= max_value – usr_ret_data(N to N*2-1);
            invert_out(N*2 to N*3-1)  <= max_value – usr_ret_data(N*2 to N*3-1);
            invert_out(N*3 to N*4-1)  <= max_value – usr_ret_data(N*3 to N*4-1);
         --img_process <= DONE;               -- process end
        end if;
     end if;
  end process Inverse_Filter;
```

Figure 7-6 Invert Image Data Path (VHDL code)

```
// Invert Filter "C" code:
// Invert all pixels within a frame of size height*width
int max_value = 0xFF;                // image depth is 8 bit, max value is 255
for (y = 0; y < height; y++ {
      for (x = 0; x < width; x++ ) {
          inv[y*width+x] = max_value – img[y*width +x] ;
       }
}
```

Figure 7-7   Invert Image "C" Code

```
-- ** Threshold Image Data Path Component** --
-- Function to perform threshold
-- N is pixel depth, bits/pixel
function thresholding(pixel : std_logic_vector(0 to N-1))
      return std_logic_vector(0 to N-1) is
      variable min_value  : std_logic_vector(0 to N-1) := x”00”;
      variable max_value  : std_logic_vector(0 to N-1) := x”FF”;
      variable threshold    : std_logic_vector(0 to N-1) := x”2E”;    -- setting threshold value
begin
      if pixel >= then threshold then
            return max_value;
      else
             return min_value;
end thresholding;

-- Threshold the image pixels
Threshold_proc: process( clock ) is
begin
    if clock = ready then
        if reset = yes then
            threshold_out <= zero;
        elsif start_process = ‘1’ then           // process 4 pixels at a time
            threshold_out(0 to N-1)         <= thresholding(usr_ret_data(0 to N-1));
            threshold_out(N to N*2-1)    <= thresholding(usr_ret_data(N to N*2-1));
            threshold_out(N*2 to N*3-1) <= thresholding(usr_ret_data(N*2 to N*3-1));
            threshold_out(N*3 to N*4-1) <= thresholding(usr_ret_data(N*3 to N*4-1));
        end if;
     end if;
  end process Threshold_proc;
```

Figure 7-8   Threshold Filter Data Path (VHDL code)

```
// “C” code for threshold filter:
//  Perform threshold operation on all pixel within an image of size=width*height
int max_value = 0xFF;
int min_value  = 0x00;
int threshold    = 0x2E;
for (y = 0; y < height; y++ {
      for (x = 0; x < width; x++ ) {
         if img[y*width+x] > threshold
             thr[y*width+x] = max_value;
         else
             thr[y*width+x] = min_value;
       }
}
```

Figure 7-9   Threshold Filter in “C” Language

7.4 MEDIAN & BINOMIAL FILTERS

Median filters reduce “salt and pepper” noise within an image by replacing each pixel with the median of the neighboring pixel values. A 3x3 median filter replaces each referenced pixel with the median obtained from its eight neighbors and itself. Implementing the binomial and median filters in hardware required several modules. The required modules include a frame buffer module, boundary condition module and the filters itself. The frame buffer provides temporary storage for the current referenced pixel and its eight neighbors. The frame buffer can be sized to reduce the number of memory reads required to process a given pixel. The optimal size of this frame buffer is (2*W + 3) * N, where W is the width and N is the depth of an image frame. When a new pixel is read from the main memory, all other pixels in the frame buffer shift to their next positions before the new one loads into position zero. For a 3*3 windowed filter, the frame buffer is implemented such that it outputs pixels at positions 0,1,2,w, w+1,w+2, 2w, 2w+1, and 2w+2 where the referenced pixel is at position w+1 and other outputs are its neighbors.

The boundary condition module is needed to provide neighbor values that are out of the image frame when the referenced pixels are at the edges of the frame. This module is a state machine that transitions either by counting the number of pixels passed through the frame buffer, or calculating position of a pixel in each column of an image frame. The out of frame neighbor pixel values can be zero or be set to the same value as the referenced pixel. For two-dimensional images, the number of states is ten including the init state. Obviously, no output is produced during the init state. Even though both the frame buffer and the boundary module are capable of producing a new pixel value every clock cycle, new pixel values are not made available at every clock cycle as transferring pixel values between main memory and frame buffer takes twenty clock cycles per four bytes.

For median filters, there are at least two common techniques available to be implemented into the hardware. The first technique requires nineteen compare and sort operations, while the second pre-sort technique requires only thirteen compare and sort operations. For both techniques, the basic building block is an N bit compare and swap, where N is the pixel depth. To operate at high frequencies, pipelining is used between stages. An example implementation of a multi-stage median filter produces nine sorted pixels during the final stage. However since the median is the only output of interest, FPGA resources can be saved if the number of sorters is reduced at the few last stages, where the median will be at column 4 (column 0 to 8, sorting pixel P1 to pixel P8, where column 0 is supposed to produce smallest byte pixel).

For the binomial average, the pixel outputs from the boundary condition module at position P1, P3, P5, P7 are shifted left once, while P4 requires left shift operation twice. After all the shift operations complete, all shifted pixels are summed. The sum output is then shifted right four times as to the divide it by sixteen.

Median Filter Data Path

For the median filter algorithm, the data path is given in Figure 7-10, and consists of a frame buffer, a module to handle boundary conditions, and nine 8-bit comparators that produces the median of the nine pixels. The median filter operates at 100 MHz, and is capable of producing a new pixel value every clock cycle.

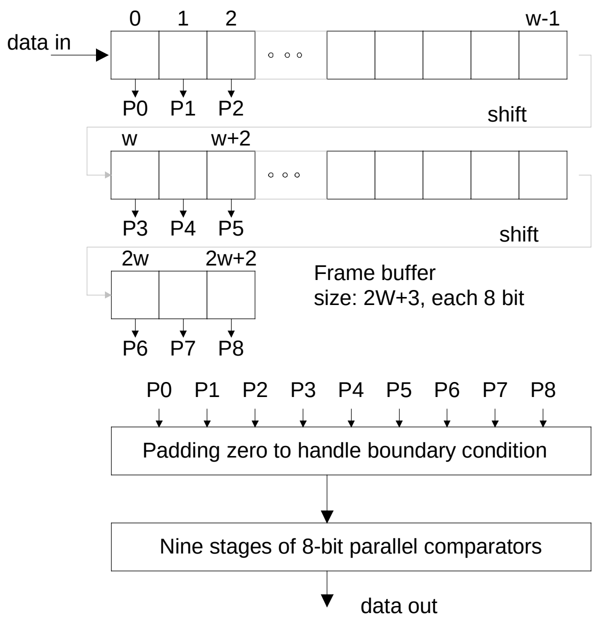

0
1
2
w-1
data in
P0 P1 P2
shift
w
w+2
P3 P4 P5
shift
2w
2w+2
Frame buffer
size: 2W+3, each 8 bit
P6 P7 P8
P0 P1 P2 P3 P4 P5 P6 P7 P8
Padding zero to handle boundary condition
Nine stages of 8-bit parallel comparators
data out

Figure 7-10 Median Filter Data Path (VHDL)

For performance comparisons, we implemented both Median and Binomial filters in software, coded in “C” language. Part of “C” code for the median filter running on CPU is shown in Figure 7-11.

```
//Median Filters
img = (int*) malloc( width * height * sizeof(int) );          // original image
med = (int*) malloc(width * height * sizeof(int) );           // processed image (median)
k      = (int*) malloc (9 * sizeof(int) );                     // temporary result

for (y = 0; y < height; y++ {
  for (x = 0; x < width; x++ ) {
    kc = 0;
     for(dx=max(0,x-1); dx<=min(width-1,x+1); dx++) {
       for(dy=max(0,y-1); dy<=min(height-1,y+1); dy++) {
            k[kc++] = img[dy*width + dx];
         }
       }
sort(k, kc);         // outer loop sort until (kc+1)/2
med[y*width+x] = k[kc/2]
```

```
    }
}
```

Figure 7-11   Main Part of Median Filter (C Code)

```
// Binomial Filter "C" Code:
int dx, dy, x, y;
int idx, total_data, total_mask;
int mask[9] = [1,2,1,2,4,2,1,2,1];                              //binomial mask kernel
int* bin;
img = (int*) malloc( width * height * sizeof(int) );          // raw image
bin  = (int*) malloc(width * height * sizeof(int) );          // processed image (binomial)

for (y = 0; y < height; y++ {
      for (x = 0; x < width; x++ ) {
          idx = 0;
          total_data  = 0;
          total_mask = 0;
          for(dx=max(0,x-1); dx<=min(width-1,x+1); dx++ {
                for(dy=max(0,y-1); dy<=min(height-1,y+1); dy++) {
                     total_data += img[dy*width + dx] * mask[idx];
                     total_mask += mask[idx];
                     idx++;
                }
           }
          bin[y*width+x] = ( total_data / total_mask );
       }
}
```

Figure 7-12   Main Part of Binomial Filter (C Code)

7.5 RESULTS

The hardware cost to implement the four image transforms is given in Table 7-1. The cost includes the hardware thread interface component. The hardware thread interface contributes to about one fourth of total slices needed to implement the four image processing algorithms.

Resources types	Resources used	Total resources on chip	% used
Slices	1429	4928	29
Flip-flop	1205	9856	12
4 -input LUT	2590	9856	26
BRAMs	0	44	0

Table 7-1 Hardware Thread Resources

This experiment demonstrates the utility of our approach in providing programmers access to the potential of the FPGA through a familiar programming model. The performance results given in Table 8-2 shows a comparison between the software and hardware implemented filters. The results clearly show that the median filter implemented on hardware can handle about 90 frames per second based on an image size of 320 x 240 x 8 bits per frame. For comparison we operate the embedded CPU and reconfigurable logic at 100 MHz, the clock limit of our current XILINX FPGA. For the software implementations, we ran separate experiments with the processor memory cache both on and off to study the effect of memory access on the CPU execution time. Further the four different transforms also indicated that CPU spends more time to perform the image processing operations as compared to memory references, particularly for the median filter. Comparison of execution times between the hardware based and software based transform implementations also reveal that the hardware thread enables us to exploit the nature of the reconfigurable hardware architectures ability to parallelize and pipeline tasks.

The four different hardware based transforms demonstrate that the execution times are dominated by the image data transfer from/to the system memory. This can be deduced from the same size of images processed by different transforms and their slight variation of execution times (9.05ms to 11.2 ms). Even though the latency of each hardware transform algorithm is small, ranging from three to eleven clock cycles, and capable of producing average of nine pixel every clock cycle in the case of binomial and median transforms, the communication costs far overweigh the image processing times as indicated by the results in the table.

Image algorithms	HW image processing	SW image processing cache off	SW image processing cache on
Threshold	9.05 ms	140.7ms	19.7 ms
Negate	9 .05ms	133.9ms	17.5ms
Median	11.2 ms	2573ms	477ms
Binomial	10.6 ms	1282ms	320 ms

Table 7-2 Image Transforms Execution Times

8 CONCLUSION AND FUTURE WORK

8.1 CONCLUSION

This thesis presents research on the extension of the multithreaded programming model across general processor and reconfigurable hardware architectures. The goal of this research is to create a programming environment able to support concurrent execution of both FPGA based hardware and CPU based software threads. Our approach allows an application to be broken into multiple threads that can be implemented into the hardware and software to take advantage of both domains within a familiar programming model. We have delivered a successful running hybrid thread system with hardware and software components synchronized using our new FPGA based synchronization mechanisms as the final outcome of this research. In addition this work provides initial framework in migrating other system services into the hardware especially to improve system performance variability. We believe that adopting this generalized programming model can lead to productivity improvement as it enables hardware and software components to share resources and synchronize at higher level communication protocol.

To extend this programming model, we have created new synchronization mechanisms and hardware thread interfaces (HWTI) within the abstraction layer described in Chapter 4. Hardware thread interface is provided to elevate the hardware computation to a higher level of representation equivalent to the software thread. The synchronization mechanisms implemented within the FPGA control mutual exclusion and countable resources to avoid concurrent access to the shared resources by multiple threads. In addition, these mechanisms provide facilities for hardware and software threads to safely sleep and wake-up. The HWTI and all the synchronization mechanisms are provided in the form of hardware libraries or IP cores (intellectual property cores). Both the CPU and FPGA threads perform their synchronizations by means of application programming interfaces (API's), similar to POSIX thread API's, to relieve users from low level system software and hardware.

As the number of synchronization variables required in a system can be large, we have created a single controller to manage multiple synchronization variables of the same type. We also migrate the sleep queues that are normally associated with each blocking synchronization variable into the FPGAs. Adopting this approach will reduce system memory requirement, free CPU from queuing tasks and improve overall system performance. Instead of implementing one queue for each synchronization variable, we create a global queue for all the synchronization variables to minimize utilization of FPGA resources. We achieve efficient global queue operation with algorithm solution. Essentially multiple variables share common controllers and one global queue within a single entity is able to reduce the hardware utilization significantly without sacrificing performance. This approach is able to lower the hardware cost to implement each synchronization variable. The hardware cost per variable for semaphores, mutexes and condition variables are 3.6 slices (0.07% of our FPGA candidate), 3 slices (0.06%) and 2.1 slices (0.04%) respectively. These new synchronization mechanisms provide fast synchronization for hardware, software and combinations of hardware/software threads. In terms of performance, a complete blocking semaphore operation on CPU, including the request and checking granting of semaphore, and subsequent queuing of a blocked thread is performed in 58 clock cycles (580nsecs on our 100 MHz system). Not only are these new mechanisms fast but they are processor family independent too. These synchronization mechanisms have capabilities similar to the POSIX thread library. The atomic operation that usually achieved by combinations of processor condition instructions integrated within memory coherency protocol of snooping data caches is encapsulated within these synchronization mechanisms. We have achieved much a simpler solution and faster mechanism for achieving semaphore semantic within 8 or less clock cycles.

We present a new enabling method in the form of HWTI core library for custom hardware computation to participate in a highly integrated manner within the general software concurrent programming model. Our approach differs from other research efforts in that the computation within the hardware is an entity that is able to execute independently, accessing data on its own and exchange data with the software components according to shared data protocol. We have created HWTI as a supporting layer that act as a system service for user computation to request for synchronization and access to system memory. In other word, HWTI enables custom threads within the FPGA to be created, accessed, and synchronized with all other system threads through library APIs. Within this layer, supporting mechanisms such as controllers to maintain the "state" of a hardware thread, provide execution control over the thread, and a set of hardware functions serving as programming interfaces (APIs) are included. The supporting mechanisms enable hardware computations to behave as independent entities that synchronize and suspend their execution similar to that of software based threads. In terms of hardware resource, each HWTI requires about 128 slices or 3% of total slices available on our FPGA example (XILINX V2P7).

The HWTI also promotes portability by encapsulating the platform specific signal within generic API’s callable by the user code.

We have performed regression and functional tests on our mutex, spin lock, counting semaphore and condition variable cores. Each regression test is loaded with 250 threads generating more than 100,000 events. In each functional test, each core is subjected to a series combination of command sequences to validate its operation including requests made by the hardware thread. Similarly, we have rigorously tested the hardware thread operation in accessing synchronization variables and system memory. For example, we have tested a hardware thread that ran individually to acquire a lock. It subsequently competed with the software thread over a period that generated more than 100,000 acquisitions. The test also includes verification of blocking and unblocking operation of hardware threads. In our tests, the hardware thread is about six times faster in acquiring a synchronization variable and accessing memory compared to our embedded CPU.

High-level integration of CPU and FPGA provides new opportunities to redefine the partition between hardware and software boundaries. Migration of synchronization mechanisms to extend the multi-thread programming model provides an exploratory step in altering these boundaries. Other system components can be migrated from the CPU into the hardware, in particular to reduce processor workload and help improve system response variability. This approach enables improvement of overall system performance that could not be achieved through traditional system software approaches alone. This research provides initial steps to migrate the software thread manager, thread scheduler and system timing services into the FPGA.

Our application study demonstrates hardware and software threads executing concurrently, synchronizing by means of our hybrid semaphore, transforming real-time images captured by a camera and displayed on a workstation. This evaluation study verifies that our hybrid thread model can be implemented within the reconfigurable hardware. The demonstration also provides insights into how best to decompose an application into multiple hybrid threads. For example sequential properties can be gathered within one thread or by using several binary blocking semaphores. This approach enables us to harness FPGA superiority in performing repetitive tasks and at the same time allows general-purpose processors to execute irregular code within the familiar multithreading program model. The result from our experiments shows that speed-up of 42 times can be achieved when hardware thread based median filtering is evaluated against the software implemented ones. In our evaluation study, hardware thread "creation" and hardware-software synchronization are achieved by means of application program interfaces (APIs). Similarly, the image processing within the hardware thread performs access to the system memory with APIs and without the need to use CPU. Thus this evaluation study has demonstrated that adopting our approach can lead to programming productivity improvement.

8.2 FUTURE WORK

The hardware thread provides an excellent base to migrate other computation components for example Java garbage collectors into the hardware especially to improve program execution times. Adopting hardware thread and the synchronization mechanism cores can relieve users from hardware/software synchronization and data exchange issues when implementing their applications. As the hardware thread and concurrency mechanisms are provided in the form of verified IP cores, users can instantiated as many hardware threads or other provided cores to suit their application without the need to rewrite them.

At present, users have to use hardware descriptive languages when implementing their applications into hardware threads. In the future, we wish to seek a suitable compiler that can translate an application written in a high-level language into both hardware and software components. For example, we are searching a compiler that can translate a high-level language program to hardware descriptive language such as VHDL.

The FPGA device that we currently use in our experiment has one embedded CPU and limited hardware resources. We are porting our hardware cores into a bigger FPGA device that has two embedded CPUs. We wish to test our synchronization mechanisms in truly multiprocessors environments.

BIBLIOGRAPHY

1. Alexander, P. and Kong, C., Rosetta: “Semantic Support for Model Centered Systems Level Design”, IEEE Computer, November 2001

2. Anderson, T., “The performance of spin lock alternatives for shared memory multiprocessors,” IEEE Transaction on Parallel and Distributed Systems, vol. 1, no. 1, pp. 6-16, January 1990.

3. Andrews, D.L., Niehaus, D., Ashenden, P. " Programming Models for Hybrid FPGA/CPU Computational Components", IEEE Computer, January 2004

4. Andrews, D.L., Niehaus, D., and Jidin, R., Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components,” Proc. 1st Workshop on Embedded Processor Arch, Proc. 10th Int’l Symp. High Performance Computer Architecture (HPCA 10), Feb 2004.

5. Andrews, D.L., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbie, M., Ortiz, J., Komp, E., Ashenden, P., “Programming Models for Hybrid FPGA-CPU Computation Components – A Missing Link”, IEEE Micro, July/Aug 2004.

6. Andrews, D.L., Niehaus, D., “Architectural Framework for MPP Systems on a Chip”, Third Workshop Massively Parallel Processing (IPDPS), Nice, France 2003

7. Ashenden, P., Designer’s Guide to VHDL, Morgan Kaufmann, 2002

8. Baloron, F., Giusto, P., Jurecska, A., Passerone, C., Sentovich, E., Chiodo, M., Hsieh, H., Lavagno, L., Sangiovanni-Vincentelli, A.L., and Suzuki, K., “Hardware-Software co-design of embedded systems: the POLIS approach”, Kluwer, 1997

9. Böhm, A.P.W., Draper, B., Najjar, W., Hammes, J., Rinker, R., Chawathe, M., and Ross, C., "One-Step Compilation of Image Processing Algorithms to FPGAs," in IEEE Symposium, Field-Configurable Custom Machines (FCCM 2001), Rohnert Park, CA, 2001.

10. Böhm, W., Hammes, J., Draper, B., Chawathe, M., Ross, C., Rinker, R., and Najjar, W., "Mapping a Single Assignment Programming Language to Reconfigurable Systems," The Journal of Supercomputing, vol. 21, pp. 117-130, 2002.

11. Bouganis, C.S., Cheung, P.Y.K, Ng, J., and Bharath, A.A., “A Steerable Complex Wavelet Construction and Its Implementation on FPGA”, Field-Programmable and Applications, 14th International Conference, FPL 2004, Antwerp, Belgium, Aug/Sept 2004.

12. Duncan, A.B., Arnold, J.M., Kleinfelder, W.J., Splash 2: FPGAs in a Custom Computing Machine. IEEE Computer Society Press, 1996

13. Dydel, S. and Bala, P., “Large scale protein sequence alignment using FPGA reprogrammable logic devices”,14th International Conference, FPL 2004, Anterwerp, Belgium, August/September 2004

14. Edward, L., "Whats ahead for Embedded Software?", IEEE Computer, Sept 2000, pp. 18-26

15. Edward, L., “Overview of Ptolemy Project”, Technical Memorandum, UCB/ERL MO 1/11, University of California, March 6, 2001.

16. Engel, F., Heiser, G., Kuz, I., Petters, S.M., Ruocco, S., “Operating Systems on SOCs: A Good Idea? “, 25th IEEE International Real-time Systems Symposium (RTSS 2004), Decemmber 5-8, 2004, Lisbon, Portugal. 2004.

17. Finley, M., Hardware/Software Co-design: Software Thread Manager, MSc thesis, ITTC, University of Kansas, Lawrence, KS, Fall 2004.

18. Frigo, J., Gokhale, M.B., Lavenier, D., “Evaluation of the Streams-C C-to-FPGA Compiler: An Application Perspective, ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays, February 11-13, 2001, Monterey, California, USA, pp 134-140.

19. Gajski, D.D., Vahid, F., Narayan, S., Gong, J., Specification and Design of Embedded Systems, Prentice Hall, 1994.

20. Gokhale, M.B., J. Frigo, J., Stone, J., Parallel C programming of reconfigurable computers: the Streams-C approach. In *HPEC 2000*, September 2000

21. Gokhale, M.B., Stone, J.M., Arnold, J., Kalinowski, M., "Stream-Oriented FPGA Computing in the Streams-C High Level Language," in IEEE Symposium on Field-Programmable Custom Computing Machines, 2000.

22. Gupta, S., Luthra, M., Dutt, N.D., Gupta, R.K., Nicolau, A., " SPARK: A high -Level Synthesis Framework for Applying Parallelizing Compiler Transformations, Proceedings of the International Conference on VLSI Design, January, 2003

23. Habibi, A., Tahar, S., "A Survey on System-On-a-Chip Design Languages", Proc. IEEE 3rd International Workshop on System-on-Chip (IWSOC'03), Calgary, Alberta, Canada, June-July 2003, pp. 212-215, IEEE Computer Society Press

24. Jidin, R., Andrews, D.L., and Niehaus, D., "Implementing Multithreaded system Support for Hybrid FPGA/CPU Computational Components, "Proc. Int'l Conf. on Engineering of Reconfigurable System and Algorithms, CSREA Press, June 2004. pp. 116-122.

25. Jidin, R., Andrews, D.L., Niehaus, D., Peck, W., Komp, E., "Fast Synchronization Primitives for Hybrid CPU/FPGA Multithreading", 25th IEEE International Real-time System Symposium (RTSS2004 WIP), Dec 5-8, 2004, Lisbon, Portugal

26. Jidin, R., Andrews, D., Peck, W., Chirpich, D., Stout, K., Gauch, J., "Evaluation of the Hybrid Multithreading Programming Model using Image Processing Transform", 12th Reconfigurable Architectures Workshop (RAW 2005), April 4-5, 2005, Denver, Colorado, USA

27. King, L.A., Quinn, H., Leeser, M., Galatopoullos, D., Manolakos, E.,"Runtime Execution of Reconfigurable Hardware in a Java Environment" in the Proceedings of

the IEEE International Conference on Computer Design (ICCD-01), 2001, pp. 380-385.

28. Lee, J., Hardware/Software Deadlock Avoidance for Multiprocessor Multi-resource System-on-Chip, PhD thesis, Georgia Institute of Technology, Atlanta, GA, Fall 2004.

29. Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., and Stockwood, J., "Hardware software co-design of embedded reconfigurable architectures," in Design Automation Conf. (DAC), 1999.

30. Loo, S.M., Wells, B.W., Freije, N., Kulick, J., “Handel-C for Rapid Prototyping of VLSI Coprocessors for Real Time System”, 24th IEEE International Conference, Southeastern Symposium on System Theory Conference (SSST2002), Huntsville, Alabama.

31. Micucci, D., Ruocco, S., Tisato, F., and Trentini, A., “ Time Sensitive Architectures: a Reflective Approach”, Proceeding of 7th International Symposium on Object-oriented Real-time distributed Computing (ISORC 2004) IEEE Computer Society Press, May 12-14, 2004, Vienna, Austria

32. National Research Council, Embedded Everywhere, A Research Agenda for Networked Systems of Embedded Computers, National Academy Press, 2001.

33. Robbins, K.A., and Robbins, S., “Practical UNIX Programming, A Guide to Concurrency, Communication, and Multithreading”, Prentice Hall, 1996

34. Rose, J., Gamal, A.E., Sangiovanni-Vincentelli, A., “Architecture of Field-Programmable Gate Arrays,” Proceedings of the IEEE, Vol. 81, No. 7, pp. 1013-1029, July 1993.

35. Self, R.P. M. Fleury, and A. C. Downton, “A design Methodology for Construction of Asynchronous Pipelines with Handel-C”, IEE Proceedings Software, Volume 150(1), pp. 39-47, 2003

36. Shaw, A.C., Real-Time Systems and Software, John Wiley & Sons. Inc., 2001.

37. Singh, H., Lee, M.H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., and Chaves Filho, E.M., "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications," *IEEE Trans. on Computers*, vol. 49(5), pp. 465-481, 2000.

38. Snider, G., B. Shackleford, B., Carter, R.J., "Attacking the Semantic Gap Between Application Programming Languages and Configuration Hardware", International Symposium on Field Programmable Gate Arrays, FPGA'01, Monterey, California, USA, Feb 2001, pp.115-124

39. Vahalia, U., "UNIX Internals, The New Frontiers, Prentice Hall, 1996

40. Vissers, K., Cases Keynote Speech, www.casesconference.org, 2004

41. Waingold, E., Taylor, M., Srikrishna, D., Srakar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarsinghe, S., and Agrawal, A., "Baring it all to Software: Raw Machines," *IEEE Computer*, vol. 30, pp. 86 - 93, September 1997.

42. Edwards, M., and Fozard, B., "Rapid Prototyping of Mixed Hardware and Software Systems", http://www.celoxica.com/technical_library/academic_papers/

43. Connell, J., Johnson, B., Early HW/SW Integration Using System C v2.0, Embedded Systems Conference - San Francisco, http://www.systemc.org/ , 2002

44. Gerlach, J., Rosenstiel, W., System Level Design Using the SystemC Modeling Platform, (SDL 2000), http://www.systemc.org/

45. Pasricha, S., STMicrolectronics, Transaction level modeling of SoC with SystemC 2.0, http://www.systemc.org/, 2002

46. Swan, S., Cadence, An Introduction to System-Level Modeling in SystemC 2.0, http://www.systemc.org/, 2003

47. Cypress Information Resources, http://www.mindbranch.com/, 2004

48. Handel C, Celoxica Inc., www.celoxica.com, 2005

49. Handel-C Language Reference Manual, Version 3,Celoxica Limited, 2004.

50. IBM core connect bus. http://www-03.ibm.com/chips/products/coreconnect/, 2005

51. JHDL 0.3.41, www.jhdl.org/,2005

52. Microcontroller Application IBM Microelectronics, ppcsupp@us.ibm.com, 1998

53. Ocapi Overview IMEC, www.imec.be/ocapi, 2003

54. Open System C Initiative, www.systemc.org, 2005

55. Streams-C sc2 Reference Manual, Los Alamos National Laboratory, http://rcc.lanl.gov/Tools/Streams-C/

56. SUIF Compiler, http://suif.stanford.edu/suif/suif1/, 2004

57. Synplify Pro, Advanced FPGA-Synthesis Solution for Multi-Million Gate Programmable Logic Designs, www.synplicity.com, 2005

58. The StarFire Board, www.annapmicro.com, Annapolis MicroSystems, 2004

59. VERILOG, www.eda.org/sv-cc/, 2004

60. VHDL, www.eda.org/vhdl-200x/, 2003

61. Xilinx. http://www.xilinx.com/, 2005