Extending the Thread Programming Model Across Hybrid FPGA/CPU Architectures

Dissertation Defense
by
Razali Jidin

Advisor: Dr. David Andrews

Information Technology and Telecommunications Center (ITTC)
University of Kansas

April 15, 2005

Thank you

- Committee members
  - Dr. David Andrews
  - Dr. Douglas Niehaus
  - Dr. Perry Alexander
  - Dr. Jerry James
  - Dr. Carl E. Locke Jr.
- Research members
  - Wesley Peck
  - Jason Agron, Ed Komp, Mitchel Trope, Mike Finley, Jorge Ortiz, Swetha Rao, …….

Presentation Outline

- Problem Statement & Motivation
- Background – Previous works
- Research Objectives
- Hybrid Synchronization Mechanisms
- Hardware Thread
- Performance Results
- Evaluation of Hybrid Thread - Image Processing
- Conclusion & Future Works

Contributions

- Status: Completed HW/SW co-design of multithreading programming model, stable and running.
- Publications:
  1. Andrews, D.L., Niehaus, D., Jidin, R., Finley, M., Peck, W., Frisbee, M., Ortiz, J. Komp, E. Ashenden, P., Programming Model for Hybrid FPGA/CPU Computational Components: A Missing Link, IEEE Micro, July/Aug 2004
  2. R.Jidin, D.L. Andrews, W. Peck, D. Chirpich, K. Stout, J. Gauch, Evaluation of the Hybrid Thread Multithreading Programming Model using Image Processing Transform, RAW 2005, April 4-5, 2005, Denver Colorado
  3. R. Jidin, D.L. Andews, D. Niehaus, W. Peck, Fast Synchronization Primitives for Hybrid CPU/FPGA Multithreading, IEEE RTSS, WIP 2004, Dec 5-8, Lisbon Portugal
  4. R.Jidin, D.L.Andrews, D. Niehaus, Implementing Multithreaded System Support for Hybrid Computational Components, ERSA 2004, June 21-24, Las Vegas
  5. Andrews, D., Niehaus,D., Jidin, R., Implementing the Thread Programming Model on Hybrid FPGA/CPU Computational Components, WEPA, International Symposium on Computer Architecture, Feb 2004, Madrid, Spain
- Further impacts:

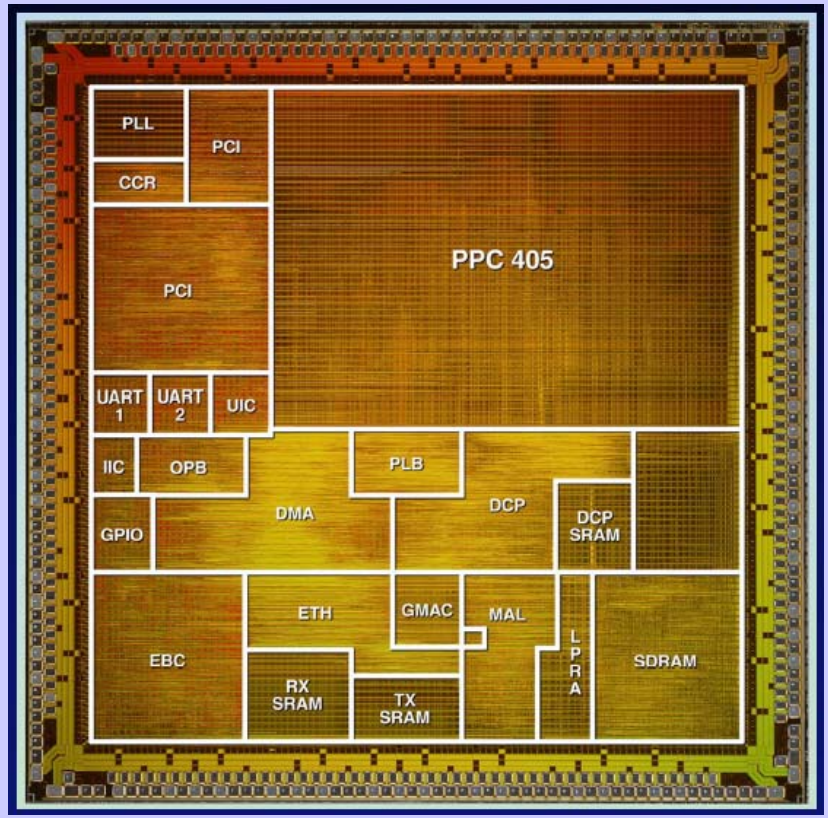
  Inquiries received from universities world wide and Cray Research

Problem Statement

- FPGAs serve as computing platforms ?
  - History: serve as prototyping and glue logics devices
  - Becoming more denser and complex devices
  - Hybrid devices: embedded CPUs + other resources
  - Require better tools to handle new complexities
- Current FPGA programming practices
  - Require hardware architecture knowledge – not familiar to the software engineers
  - Have to deal with timing issues, propagation delays, fan-out, etc.
  - Hardware and software components interaction using low level communication mechanisms

Motivation: Hybrid CPU/FPGA Architectures

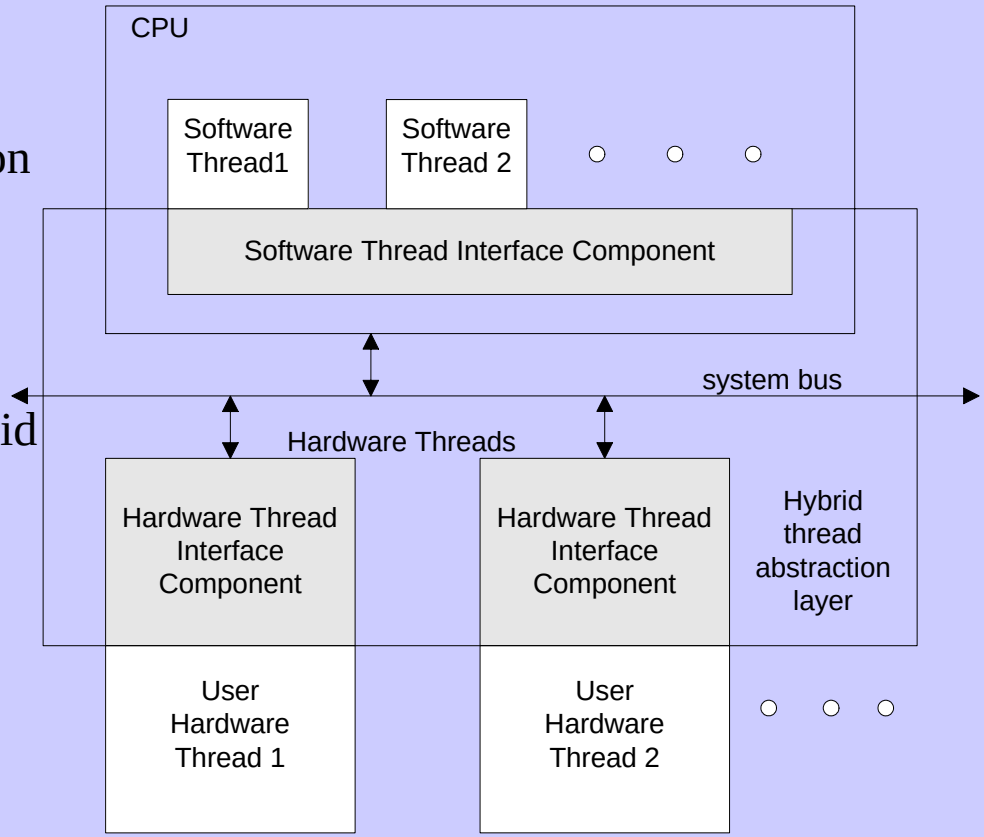
- Embedded PPC 405 CPU + Sea of free FPGA Gates (CLB's) + …
- BRAM provides efficient storage to save system "states".
- System components provided as libraries or soft IPs:
  - System buses (PLB, OPB)
  - Interrupt controllers, UARTs
- Migration of system services from CPU into FPGA can provide new capabilities to meet system timing performance. New services are provided in the form of soft IPs



Source: IBM T.J Watson Research Center

Motivation: Higher Abstraction Level

- Need to use high level of abstraction to increase productivity
  - Focus on applications not on hardware details.
  - Reduce the gap between between HW and SW designs, to enable programmer access to hybrid devices.
- Hybrid Thread Abstraction Layer
  - abstract out hardware architectures such as buses structure, low level peripheral protocols, CPU/FPGA components, etc.



Previous Works

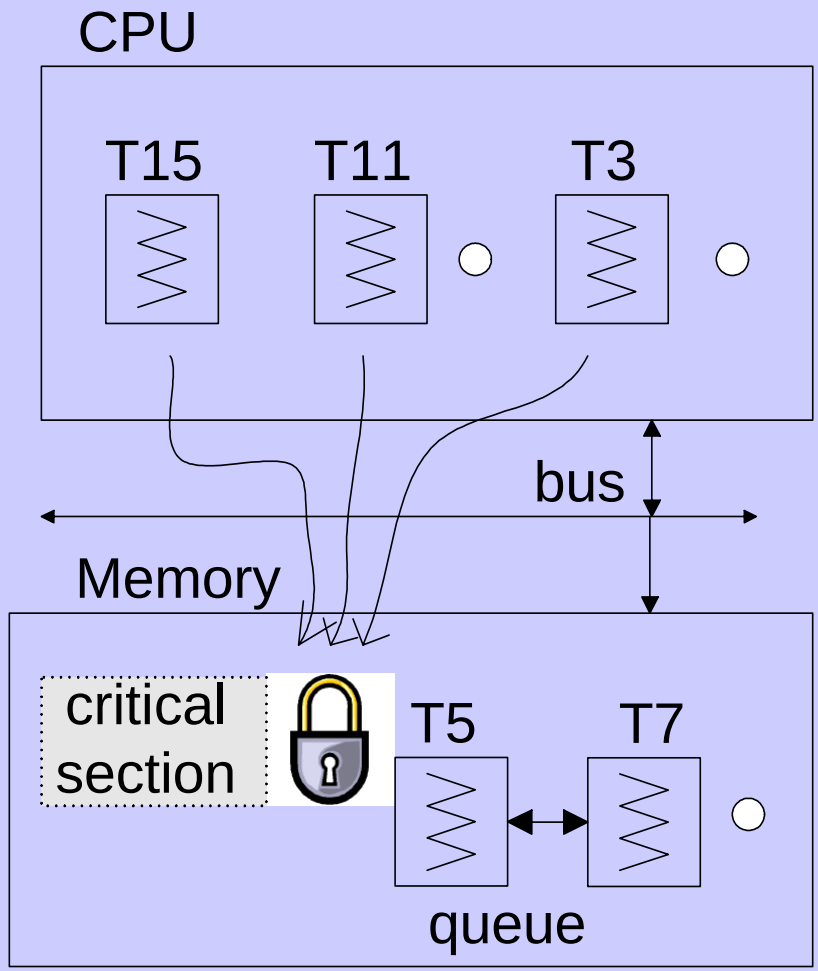
- Research efforts to bring High Level Languages (HLL) into hardware domain
- Streams-C [Los Alamos]
  - Supplements C with annotations for assigning resources on FPGA
  - Suitable for systolic based computations, compiler based on SUIF
  - Hardware/software communication using FIFO based streams
  - Programming productivity versus device area size
- Handel C [Oxford]
  - Subset of C + additional constructs for specifying FPGA circuits
  - Compile Handel C programs into synchronous machines
  - Hardware/software interactions using low level communication mechanisms
- System level [System C, Rosetta]
  - Attempt to remove hardware/software boundaries
  - High level integration between hardware & software components?

Research Objectives

- Goal: Create an environment where programmers can express system computations using familiar parallel thread model
  - standard thread semantics across CPU/FPGA boundaries
  - threads represented such that they can exist on both components
  - enable threads to share data with synchronization mechanisms
- Issues of interest:
  - FPGA based Thread Control and Context:
    - initiating, terminating, synchronizing threads
    - computational models (threads over FSM's)
    - new definition of thread context
  - Synchronization Mechanisms for CPU/FPGA based Threads
    - Semaphore, lock (mutex) or condition variables
  - API and Operating System (OS) Support
    - User Application Program Application (API) Library Functions
    - System services adaptation and migration
      - Ex. thread scheduling

Current Thread Programming Model (TPM)

- An application can be broken into executable units called threads (a sequence of instruction).
- Threads execute concurrently on CPUs (M threads map to N CPUs)
- Threads interleave on a single CPU to create an illusion of concurrency
- Accesses to shared data are serialized with the aid of synchronization mechanisms.



Current Synchronization Mechanisms

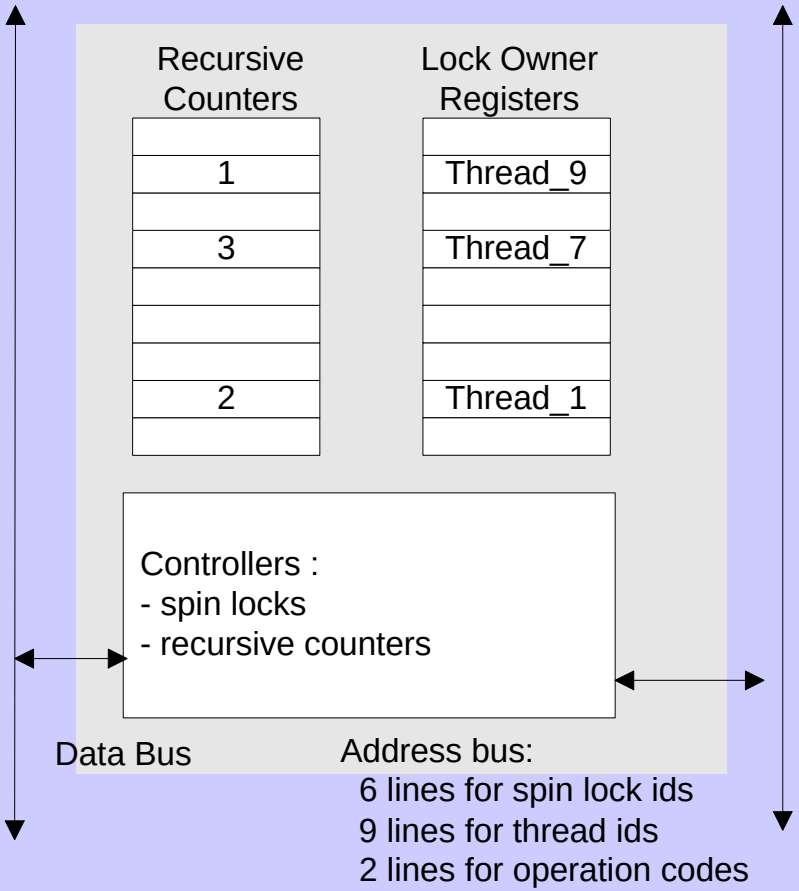
- Current synchronization mechanisms
  - Depend on atomic operations provided by HW or CPU
  - SW: CPU instructions Test and Set
    - Set variable to one, return old value to indicate prior set
  - HW: Snoopy cache on multiprocessors
- Challenges
  - Current methods do not extend well to the HW based Thread
  - Do not want to increase overhead on CPU
- New methods
  - FPGAs provide new capabilities to create more efficient mechanisms to support semaphores
  - No special instruction, no modification to processor core
  - New FPGA based synchronization mechanism provided as IP cores

Achieving Atomic Operations with FPGA

- Atomic transaction controller on FPGA
  - Read acknowledgement is delayed
  - Hardware operation completes within this delay
  - Use lower order address lines to encode necessary information such thread ID and lock ID
  - Controller returns status & grant to the application program interface (API) request on data bus
- Issues on cost of FPGA resources when the number of synchronization variables in a system is large
  - Implement all the synchronization variables within a single entity.
  - Use a single controller to manage multiple synchronization variables.
  - Use on chip block memory (BRAM) instead of LUT to save the state of each individual variables
  - Example our multiple (64) spin locks core

Multiple Spin Locks Core

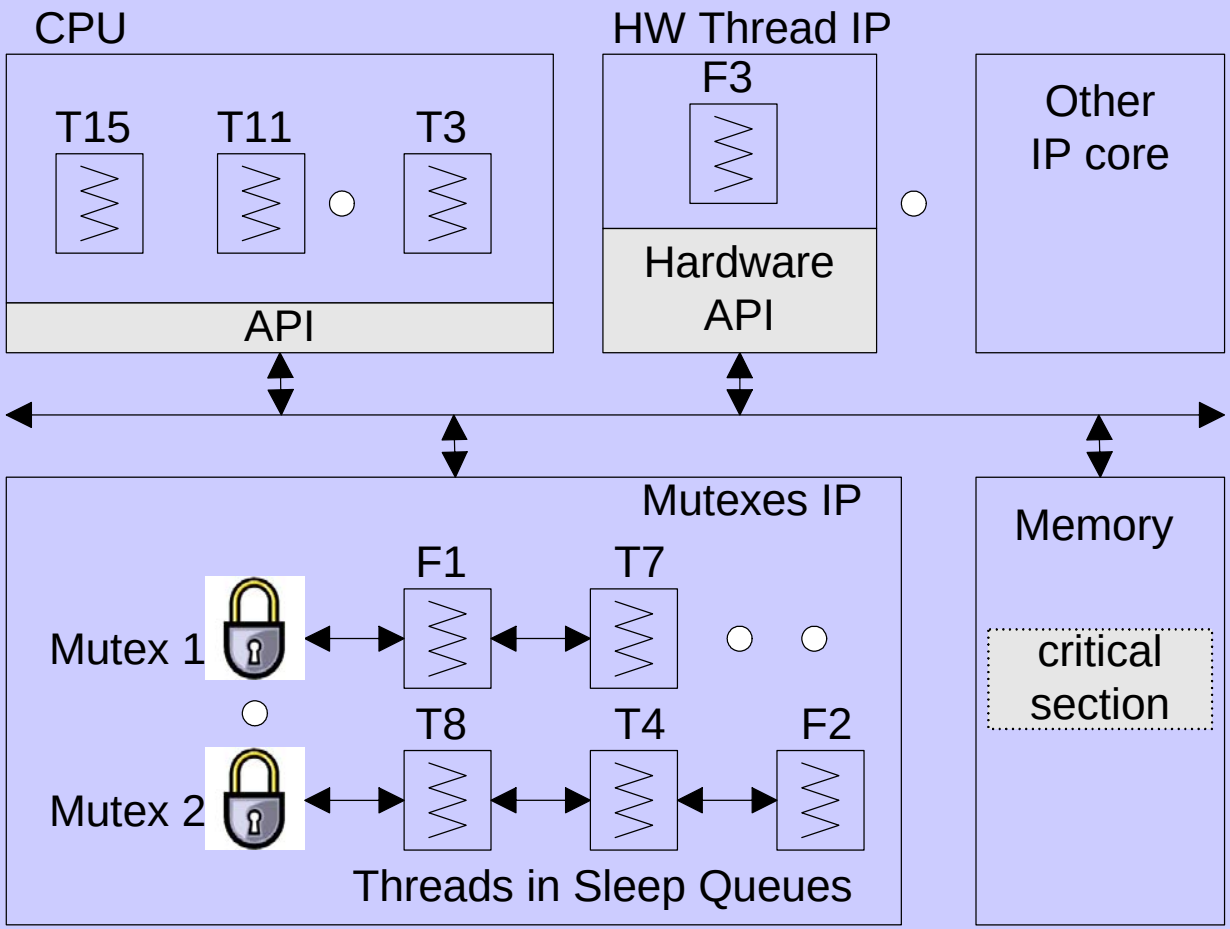
- APIs
  - Spin_lock
  - Spin_unlock
- Lock BRAM
  - 64 Recursive counters
  - 64 Lock Owner register
- Controllers
  - Common controllers for multiple locks
  - Access to Lock BRAM
  - Atomic read transaction
  - Recursive error
  - Reset all locks



Blocking Type Synchronization

- Spin vs. blocking type synchronization
  - Blocking reduces bus activities and does not tie CPU
  - Blocking requires queues to hold the sleeping threads
- Mapping of synchronization variables to sleep queues
  - Provides a separate queue for each blocking semaphore is costly when many semaphore variables are needed on a system
- Global Queue
  - Creates multiple semaphores with a single global queue
  - Efficient queuing operation but not at the expense of hardware resources
- Wakeup mechanism & delivery of unblocked threads
  - De-queue operation of unblocked threads
  - Delivery of unblocked threads either to the scheduler queue or individual hardware threads (bus master capability)

Hybrid Thread System

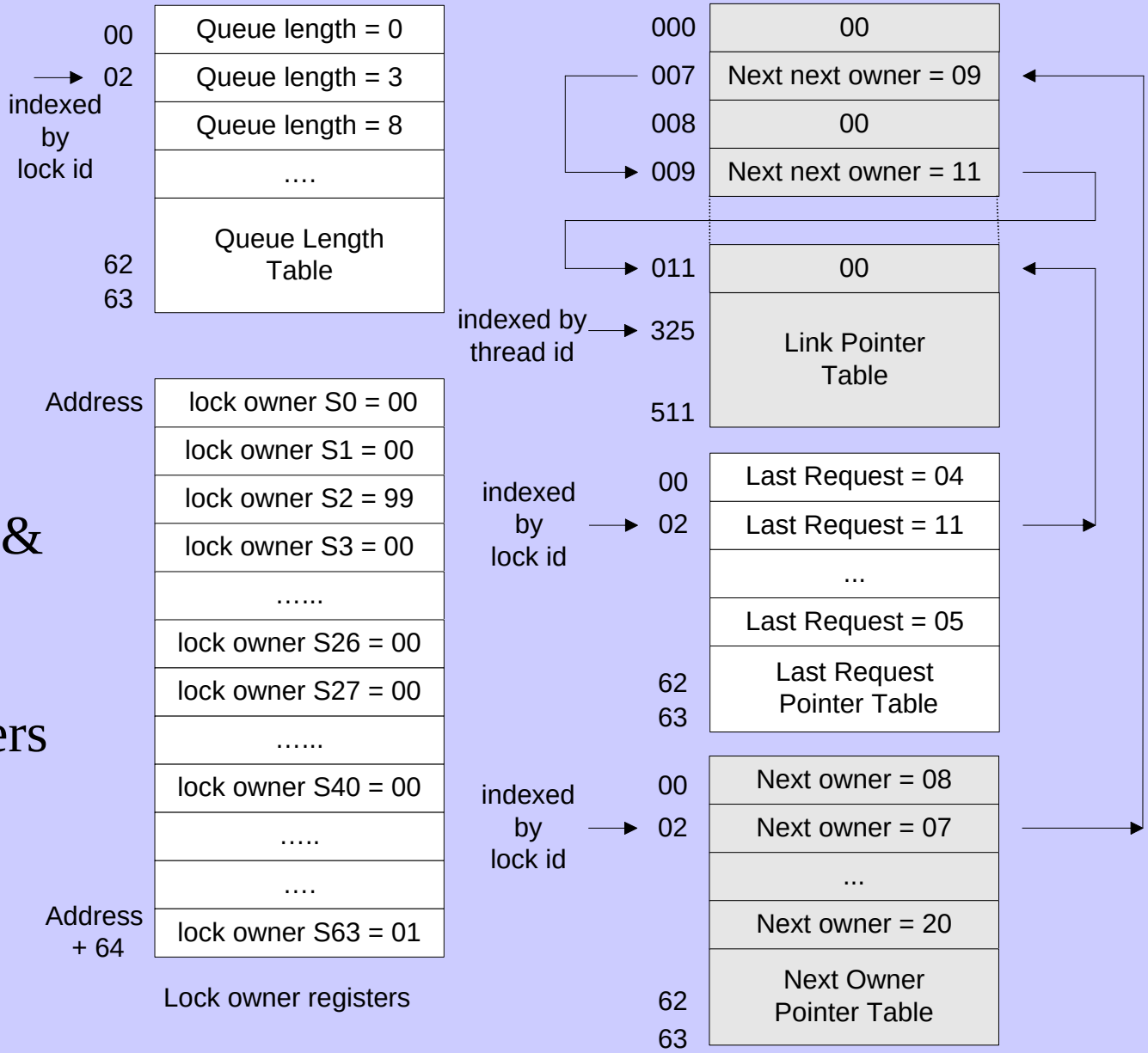


- Moves Mutexes + queues + wake-up into FPGA from memory
- Provides synchronization services to FPGA & CPU threads

Blocking Synchronization Core Design

- Global Queue
  - Conceptually configured as multiple sub-queue associated with different semaphores
  - Combined lengths of all sub-queues will not be greater than the number of total threads in the system as a blocked thread cannot make another request
  - For efficient operation, the global queue is divided into four tables:
    - Queue Length Table contains an array of queue lengths
    - Next owner Pointer Table contains an array of lock next owners
    - Last Request Pointer Table contains an array of last requesters
    - Next Next Owner Table contains link pointers

Global Queue & Lock Owner Registers



Multiple Recursive Mutexes Core

- Provide exclusive accesses to shared data & allow threads to block
- Operations: *mutex_lock* (recursive), *unlock* and *trylock*

mutex_lock( )

if thread ID = OWNER

  lock selected mutex

  cnt = cnt + 1

else

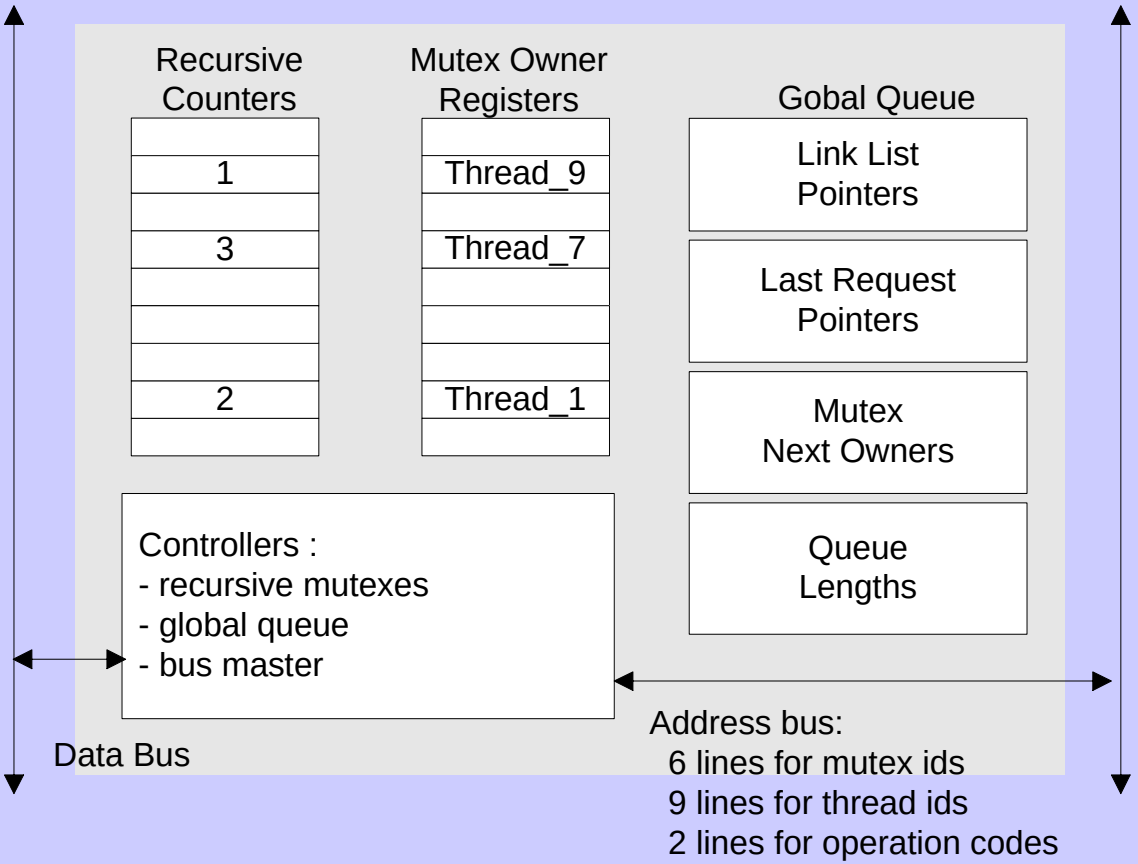
  queue thread ID

mutex_unlock( )

cnt = cnt – 1

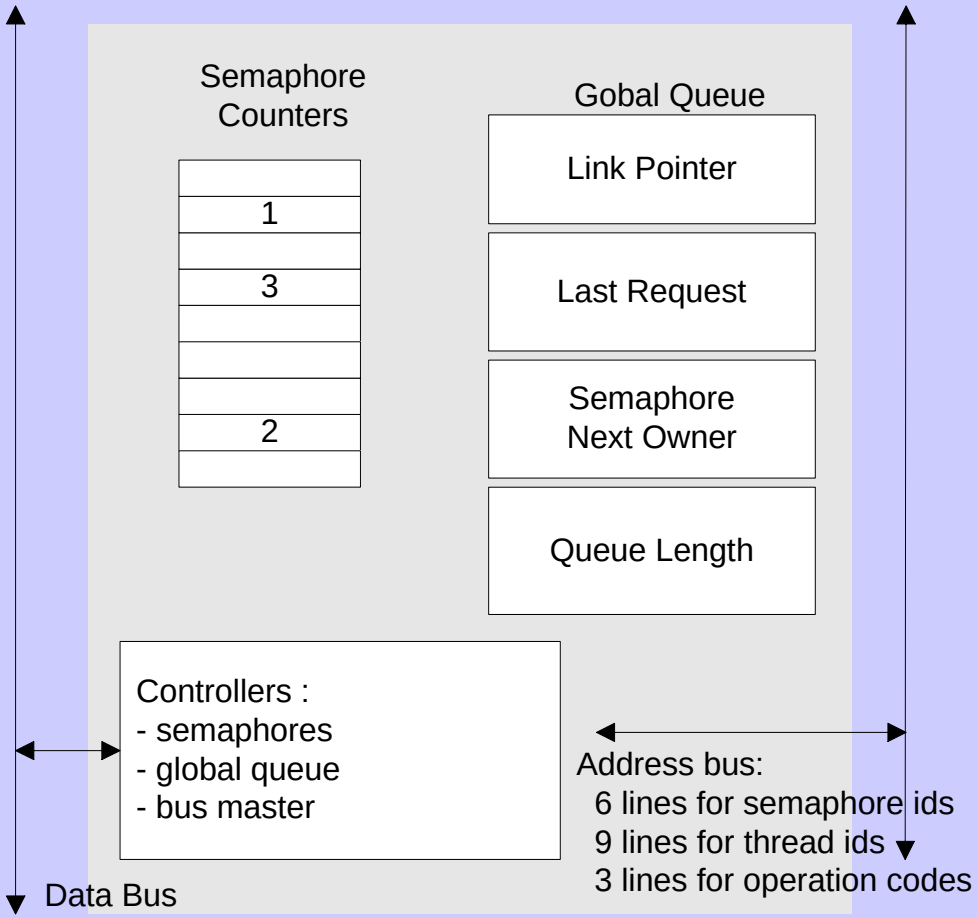
releases the mutex

when its cnt reaches 0



Multiple Semaphores Core

- *sem_wait(sm)*
  - if C ≥ 1 then C = C - 1
  - else queues thread ID
- *sem_post(sm)*
  - if blocked thread, dequeues
  - else C = C + 1
- *sem_trywait(sm)*
  - non blocking



A Condition Variable

- Implements sleep/wakeup semantics using condition variables
- Useful for event notification
- Associated with a predicate which is protected by a mutex or spin lock
- Wakeup one or all sleeping threads
- Up to 3 or more mutexes are typically required:
  - one for the predicate
  - one for the sleep queue (or CV list)
  - one or more for the scheduler queue (context_switch)
- New approach requires one mutex (predicate)

Condition Variable APIs

```
void wait (cv *c, mutex *m)
{
  lock (&c->qlistlock);
  add thread to queue
  unlock (&c->qlistlock);
  unlock (m); //release mutex
  context_switch ( );
  /* when wakes-up */
  lock (m); //acquire mutex
  return;
}
```

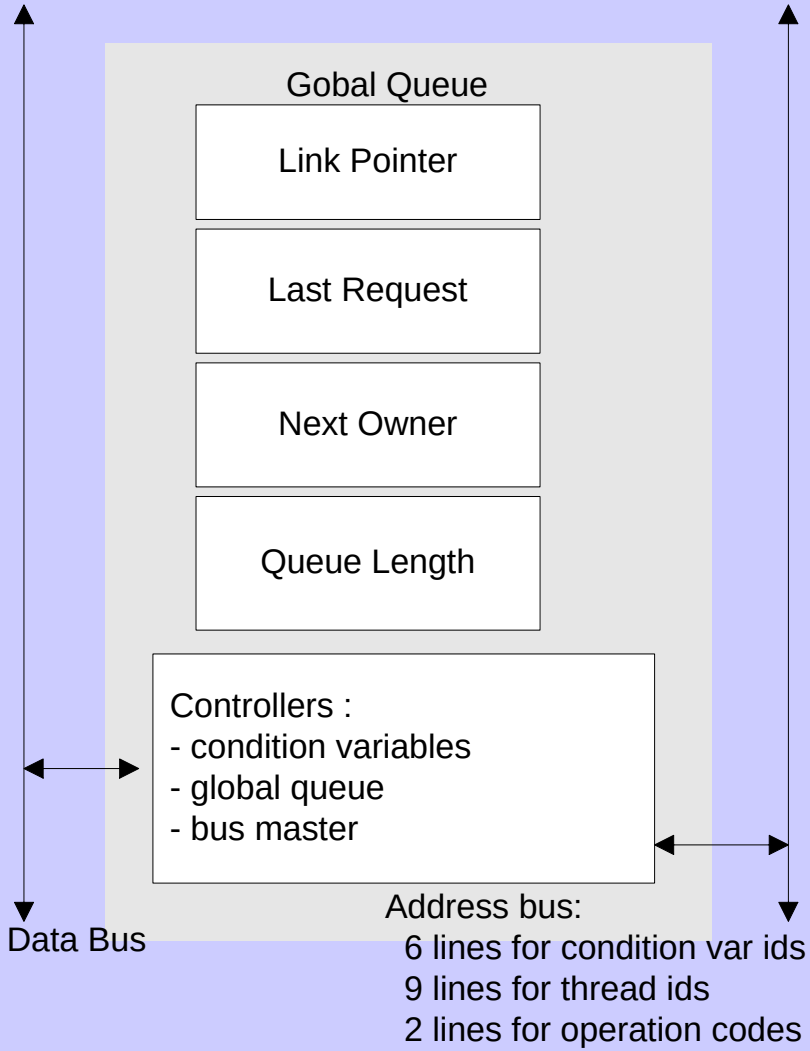
```
void signal (cv *c)
{  lock (&c->qlistlock);
    remove a thread from list
    unlock (&c->qlistlock);
    if thread, make runnable;
    return; }
```

```
void broadcast (cv *c)
{ lock (&c->qlistlock);
   while (qlist is nonempty) {
     remove a thread
     make it runnable}
  unlock (&c->qlistlock);
  return;}
```

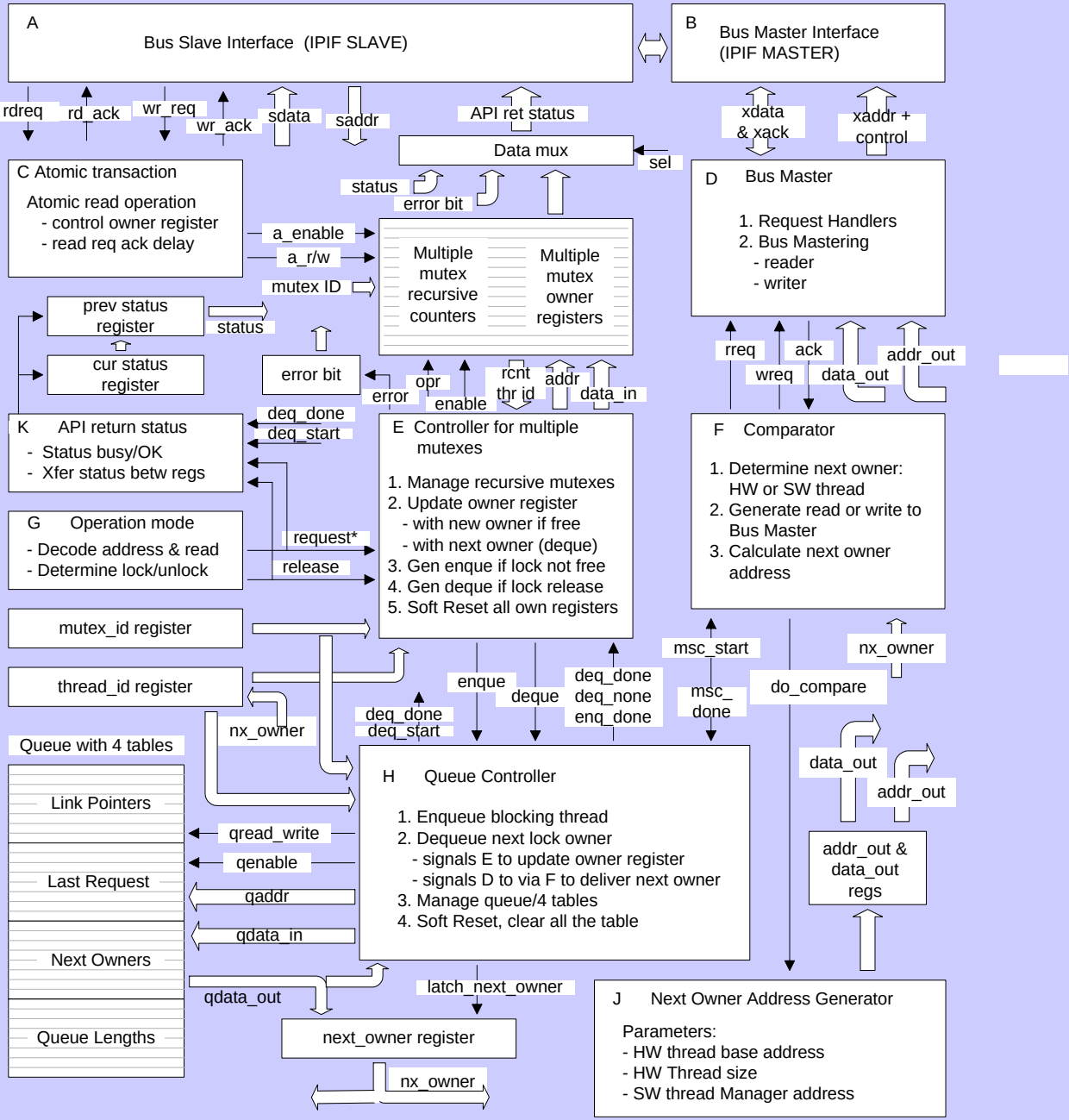
Source: VAHALIA, UNIX Internals

Multiple Condition Variables Core

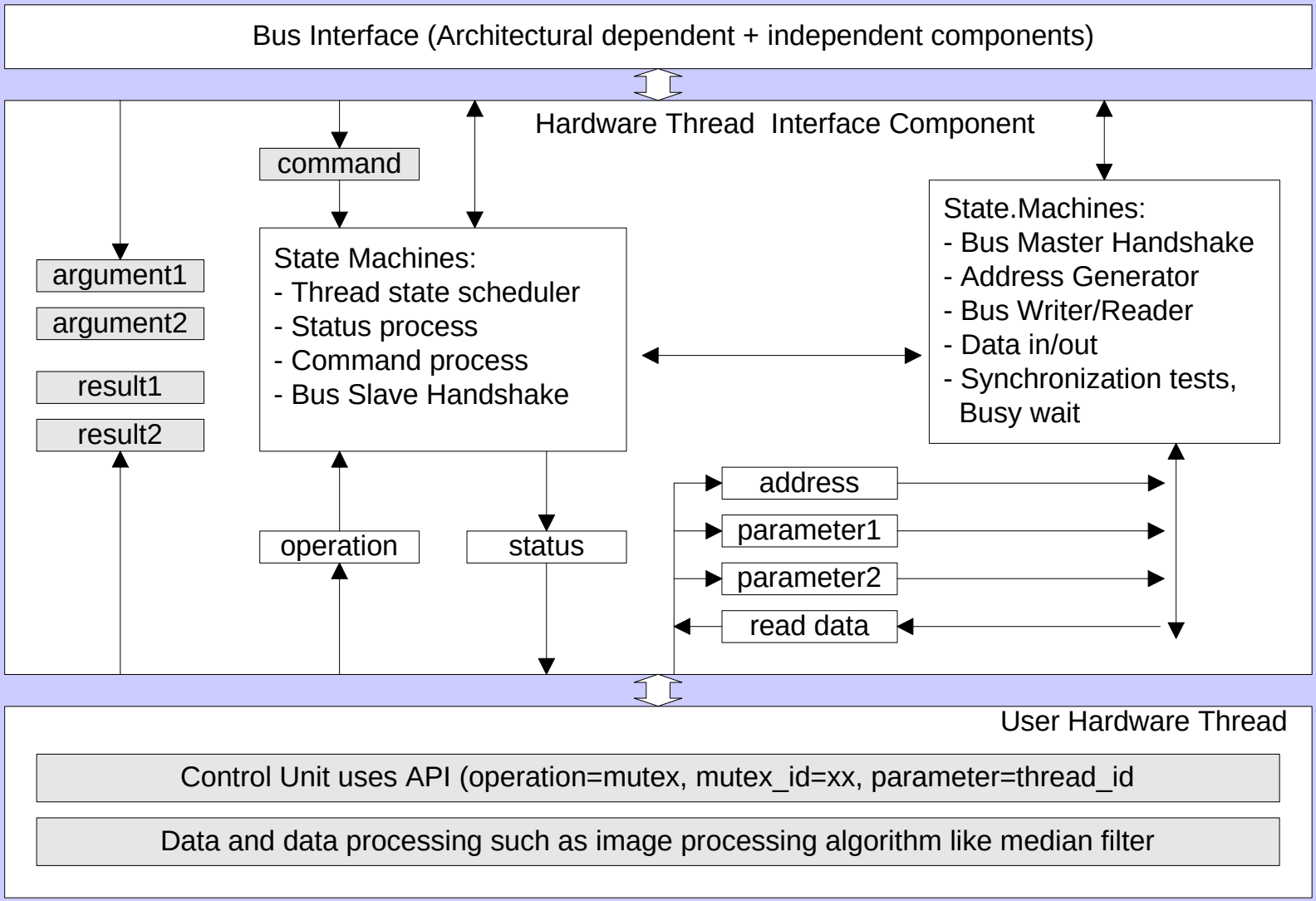
- *cond_wait(cv, mutex)*
  - Queuing of thread Ids
- *cond_signal(cv)*
  - De-queuing of a thread ID
- *cond_broadcast(cv)*
  - De-queuing & delivery of all blocked threads
  - Return busy status to new requests if delivery is not complete yet.

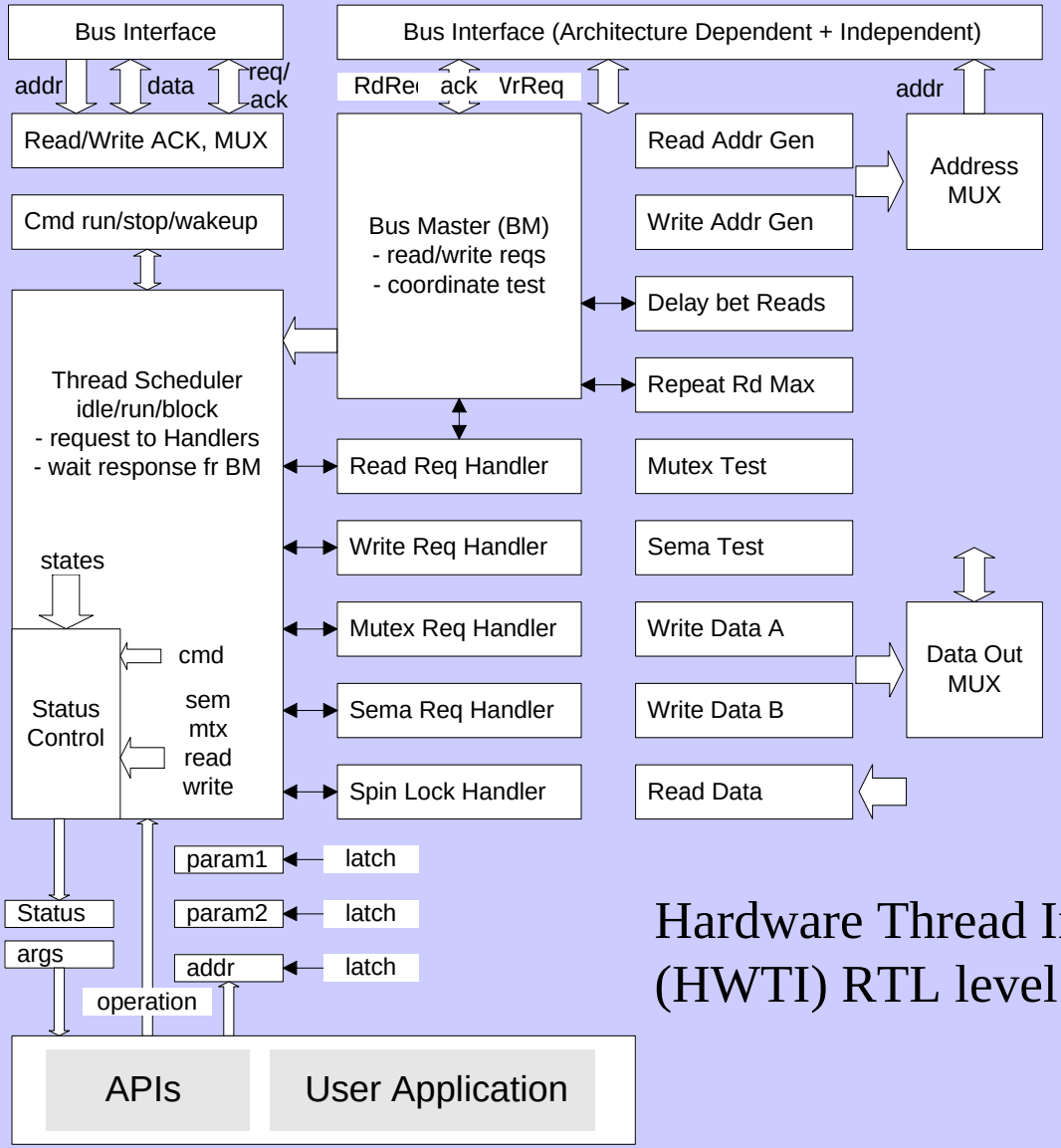


Multiple Mutex Core RTL level description



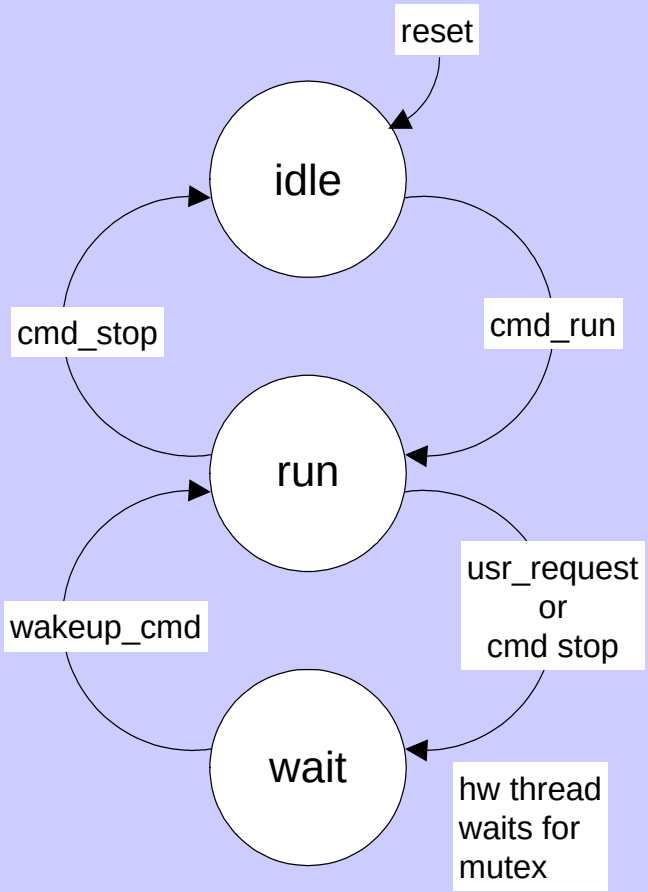
Hardware Thread Architecture





Hardware Thread Interface Core (HWTI) RTL level description.

Hardware Thread States (Contexts)



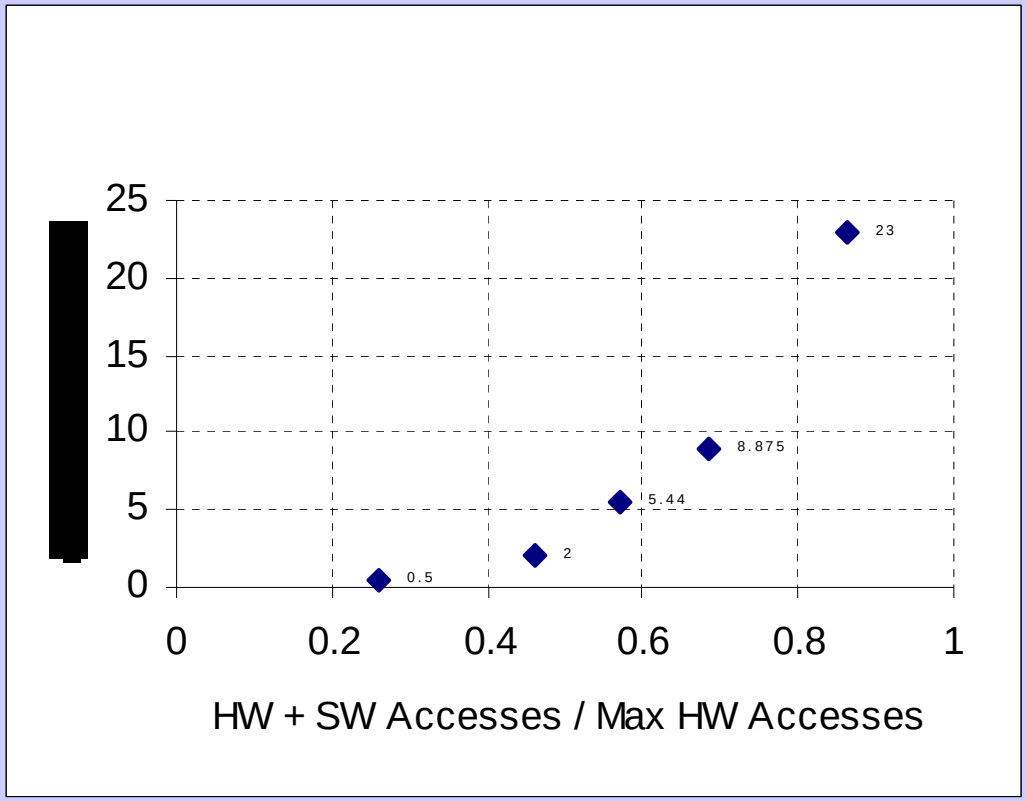
1. Moves to RUN if receives cmd_run
2. Moves to WAIT while in the process of obtaining mutex or semaphore
3. Moves to RUN state if mutex is obtained.
4. If mutex is not available, block waits in WAIT state until wake-up command is received from the mutex core
5. Thread state visible via status register
6. User computation decides when it is appropriate to check status register and control it’s own operation

Hardware Thread APIs

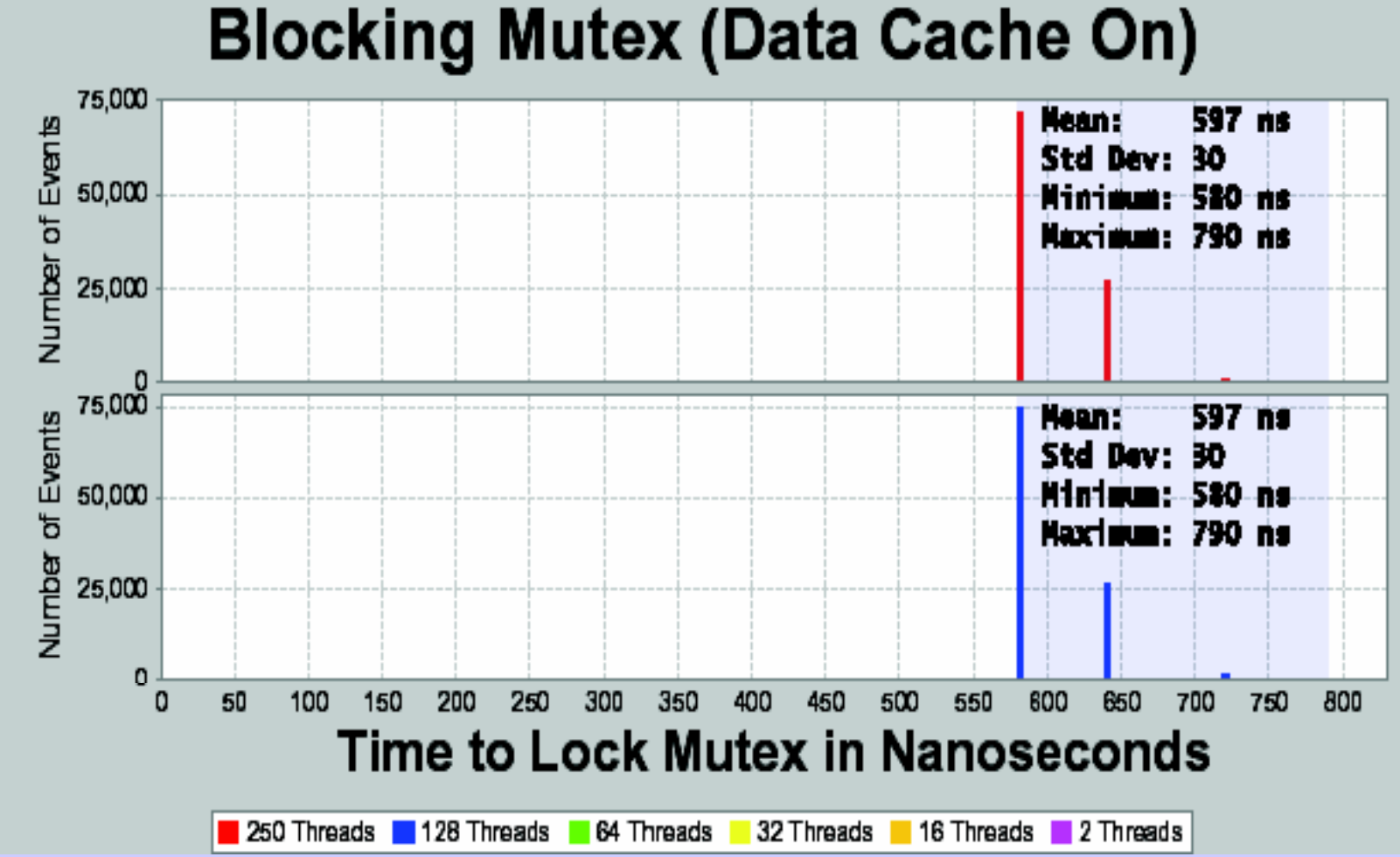
- HW_Thread_Create API on CPU
  - CPU loads arguments to registers
  - CPU writes “code” into command register to start/stop
- HW APIs on HW Thread
  - Synchronization APIs
    - Mutex: blocking lock, unlock
    - Semaphore: wait, post
    - Spin lock: lock, unlock
  - Memory read/write accesses APIs
  - APIs write operation codes into the operation register, and status register provides feedback to the user

HW/SW Threads Spin Lock Access Ratios

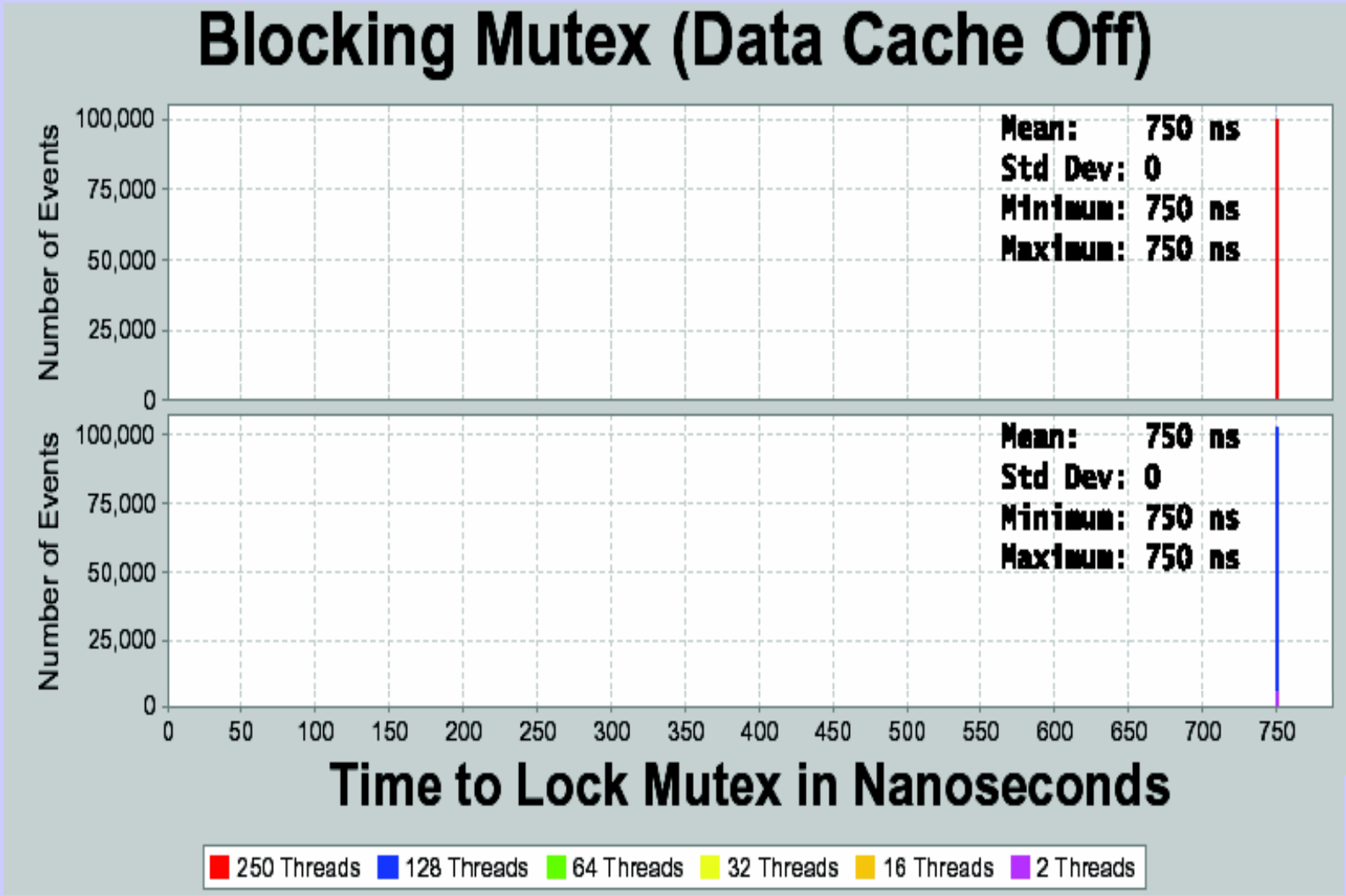
- Baseline performance HW and SW thread run individually to own and release a spin lock, hw faster by a 6:1 ratio.
- Allow both Hardware/Software Hybrid Threads to compete:



Timing Performance



Timing Performance



Synchronization Hardware Cost

Synchronization type	Total slices for 64 synchronization variable	Number of slices per synchronization variable
Spin Lock	123	1.9
Mutex	189	3
Semaphore	229	3.6
Condition Variable	137	2.1

Hardware Resources for 64 MUTEXES (excluding bus interface)

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

Synchronization Access Time

Synchronization APIs	internal operation (clk cycles)	bus transaction after internal operation start (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

Hybrid Threads: Image Processing

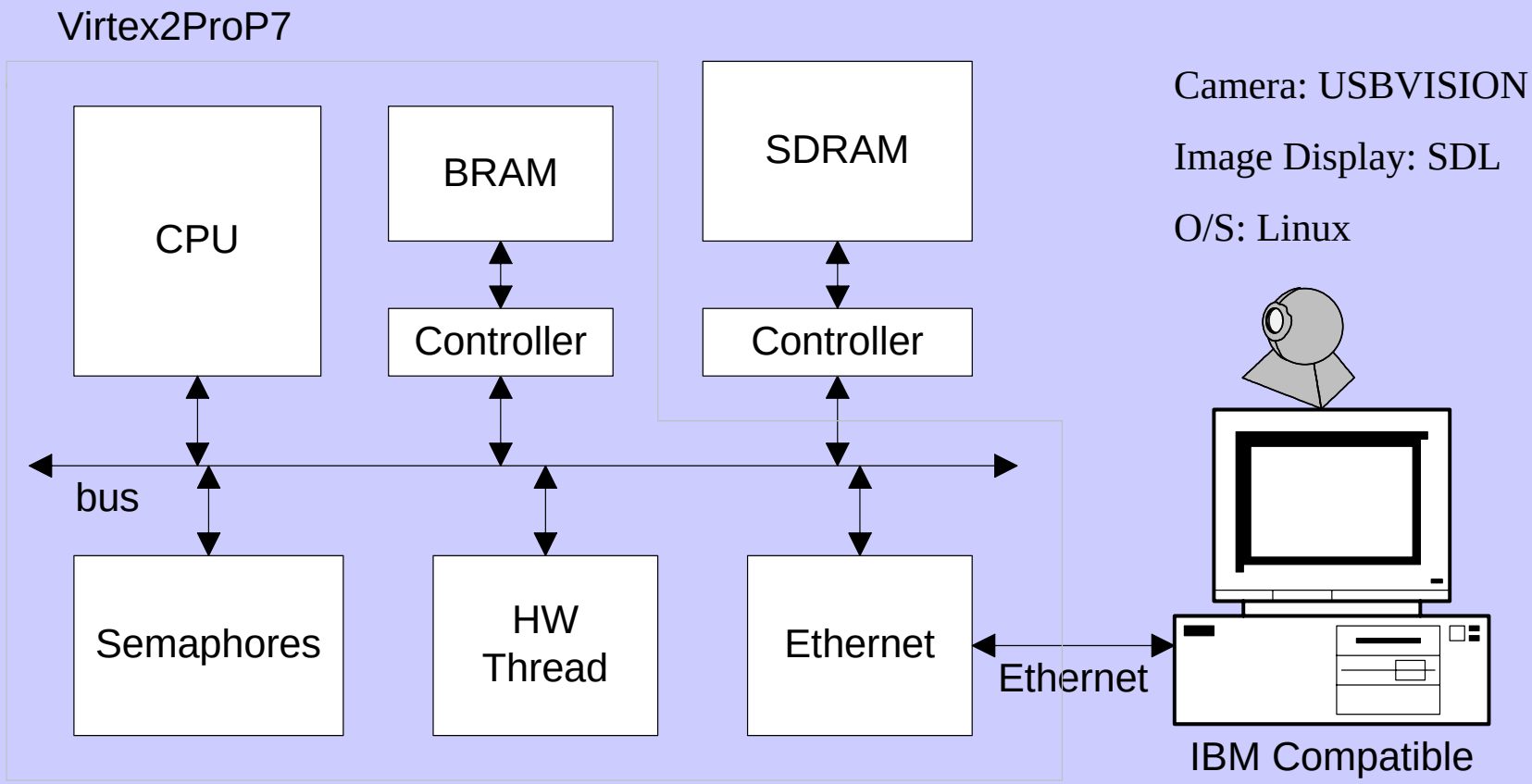


Image Processing Flow Diagram

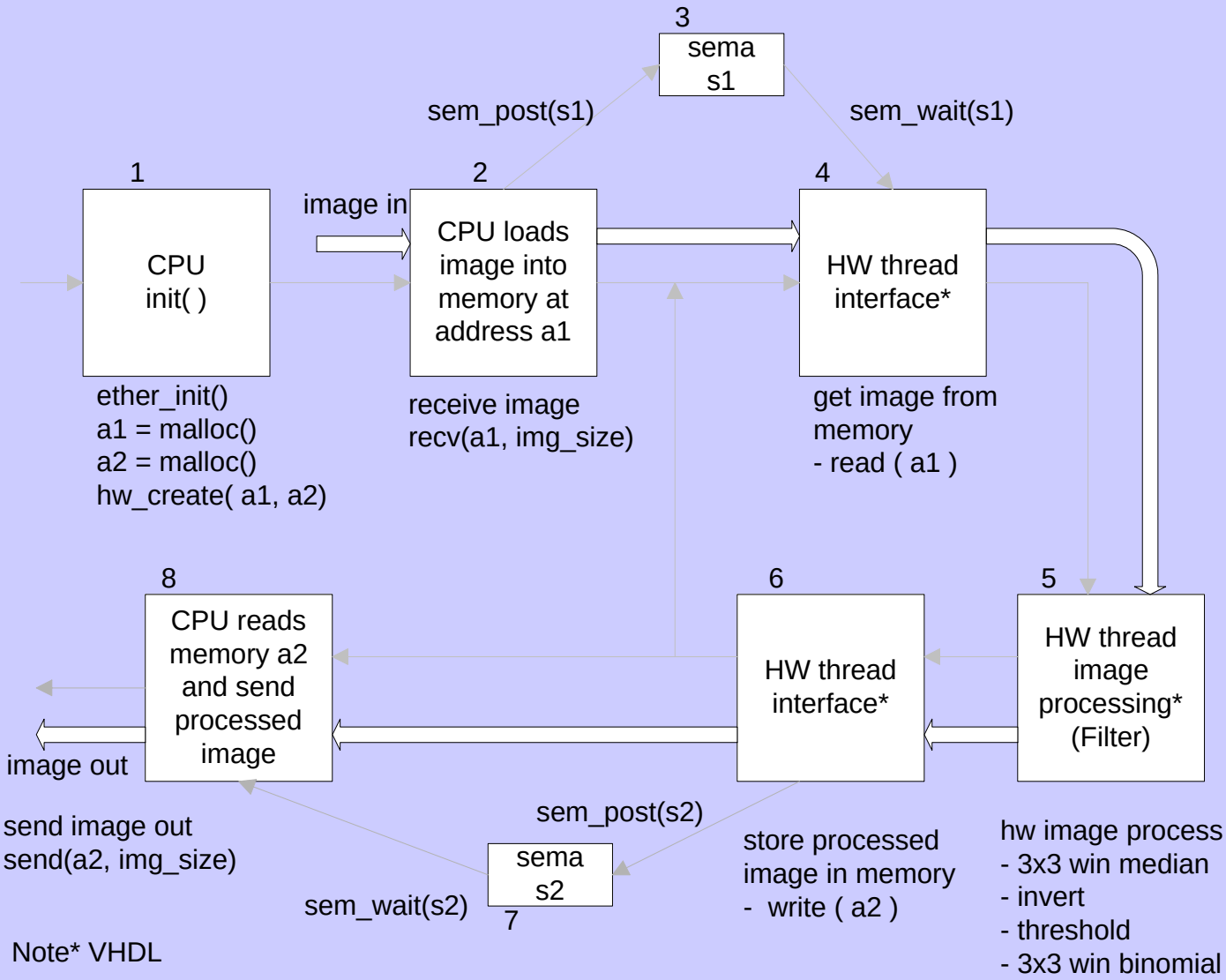
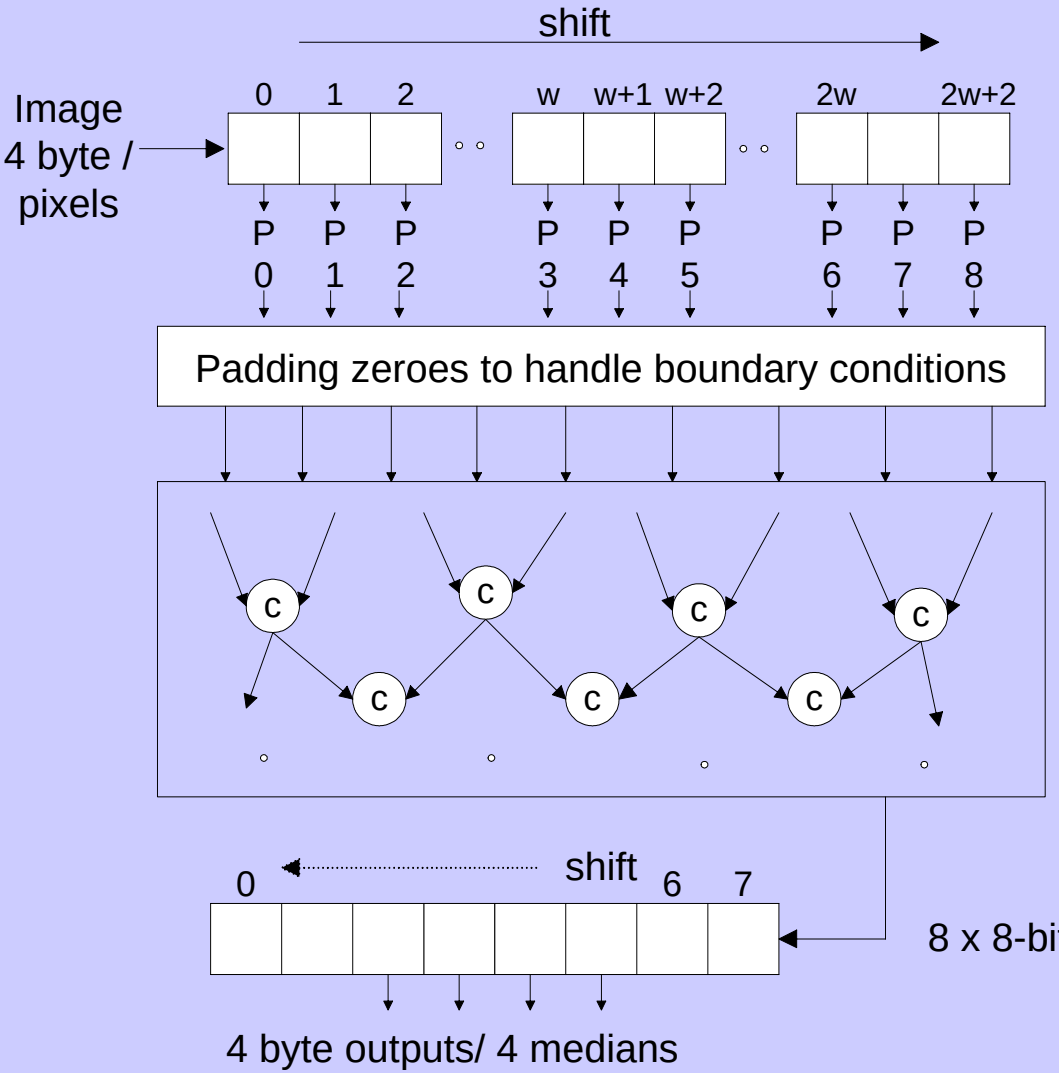


Image Transform Example: SW + HW Components

```
PART OF SOFTWARE (CPU):
addr1 = malloc(image_size) //raw image ptr
addr2 = malloc(image size) //proc image ptr
//Hardware thread create API
hw_thread_create(addr1, addr2, function )
while (1) {
 //Get image from Ethernet
    receive(src, addr1, img_size)
//Let hw thread know image data is available
    sem_post( &sema1 );
//Wait for hw thread finish processing
    sem_wait( &sema2 );
//Send processed image
    send(dest, addr2, img_size); }
```

```
PART HARDWARE (FPGA):
If command == run
{
 SW:   sem_wait( &sema1 )
 RD:   read data
       processing wait
        write data
       if count != image_size
          RD:
       else
          SP:
 SP:   sem_post ( &sema2)
       branch SW:
}
```

Frame buffer & Parallel Median Filter



Frame Buffer
Size (2W+3) * 8 bits
Output: 3x3 window or 9 pixels
Image size: W * H * N

Boundary condition:
top left, top side, top right,
right side, etc

Pipelined Median filter
9 stages 8 bit comparators,

Calculate median of 9 pixels

8 x 8-bit shift register

HW vs.SW Image Processing

- Image frame size 240 x 320 x 8 bits
- FPGA & CPU clocked at 100 MHz
- For median transform, FPGA can process 100 frames/sec, speed-up about 40x, consistence with [12, 27]
- Execution time dominated by communication

Image Algorithms	HW Image Processing	SW Image Processing Cache OFF	SW Image Processing Cache ON
Threshold	9.05 ms	140.7 ms	19.7 ms
Negate	9.05 ms	133.9 ms	17.5 ms
Median	11.2 ms	2573 ms	477 ms
Binomial	10.6 ms	1084 ms	320 ms

Conclusion & Future Works

- Extend thread programming model across CPU/FPGA
- Our synchronizations cores provides services similar to POSIX thread.
  - Test program uses our CVs and mutex produced similar result when port it to desktop running with Pthread.
  - Semaphores used in the image transform evaluations.
- Effective synchronization mechanism, improve system performance & reduce memory requirements.
- Improve programming productivity, while at the same time providing the benefit of customized hardware from within a familiar software programming model
- Hardware thread can be used as a base to implement other computations into hardware.
- High level language compiler that can translate applications into hybrid hardware and software components

Thank You!