Communication and Synchronization in Multithreaded Reconfigurable Computing Systems

Enno Lübbers and Marco Platzner
Computer Engineering Group
University of Paderborn

{enno.luebbers, platzner}@upb.de
Design of CPU/FPGA Systems

- hardware accelerators typically integrated as slave coprocessors
- hardware/software boundary explicit
- tedious to program
- portability issues

- software application
- hardware accelerator
- CPU
- FPGA
Design of CPU/FPGA Systems

- hardware accelerators typically integrated as slave coprocessors
- hardware/software boundary explicit
- tedious to program
- portability issues

![Diagram showing the design of CPU/FPGA systems with software application, hardware accelerator, CPU, and FPGA connected by a bus.]
Multithreaded Programming

application
Multithreaded Programming
Multithreaded Programming

operating system

thread

thread

thread

thread
Multithreaded Programming

- software thread
- software thread
- hardware thread
- hardware thread

operating system

CPU

FPGA
Multithreaded Programming

- Software thread
- Software thread
- Hardware thread
- Hardware thread

- OS kernel
- OS interface
- OS interface
- CPU
- FPGA
Multithreaded Programming

- Software thread
- Software thread
- Hardware thread
- Hardware thread

- OS kernel
- OS interface
- OS interface

- CPU
- FPGA
Multithreaded Programming

- communication and synchronization

- software thread
- software thread
- hardware thread
- hardware thread

- OS kernel
- OS interface
- OS interface
- CPU
- FPGA
Multithreaded Programming

- communication and synchronization
  - high-level (between the threads themselves)
Multithreaded Programming

- communication and synchronization
  - high-level (between the threads themselves)
  - low-level (between hardware threads and operating system)
Outline

- motivation

- high-level communication and synchronization

- low-level communication and synchronization

- performance & overheads

- conclusion & outlook
Programming Model

- Applications are divided into threads
- Threads communicate via operating system objects
  - Semaphores
  - Mailboxes
  - Shared memory
  - ...

.Thread_A

.MBOX_IN1

.MBOX_IN2

.MBOX_DATA

.shared memory

.THREAD_A

.THREAD_B

.THREAD_C

.MBOX_OUT
Programming Model

- applications are divided into threads
- threads communicate via operating system objects
  - semaphores
  - mailboxes
  - shared memory
  - ...

examples for API functions used by threads

<table>
<thead>
<tr>
<th>software (POSIX, C)</th>
<th>hardware (ReconOS, VHDL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sem_post()</td>
<td>reconos_sem_post()</td>
</tr>
<tr>
<td>pthread_mutex_lock()</td>
<td>reconos_mutex_lock()</td>
</tr>
<tr>
<td>mq_send()</td>
<td>reconos_mbox_put()</td>
</tr>
<tr>
<td>value = *ptr</td>
<td>reconos_read()</td>
</tr>
<tr>
<td>pthread_exit()</td>
<td>reconos_thread_exit()</td>
</tr>
</tbody>
</table>

E.Lübbers & M.Platzner, University of Paderborn
ERSA '08
High-Level Synchronization in ReconOS

- **semaphores**
  - general mechanism to synchronize execution
  - blocking wait() operation, non-blocking post() operation
  - supported by both hardware and software threads
High-Level Synchronization in ReconOS

- **semaphores**
  - general mechanism to synchronize execution
  - blocking `wait()` operation, non-blocking `post()` operation
  - supported by **both** hardware and software threads

- **mutexes**
  - specific mechanism to protect critical sections (e.g. read-modify-write to shared memory)
  - a thread can only release a mutex it “owns” (has previously locked)
  - supported by **both** hardware and software threads
High-Level Synchronization in ReconOS

- **semaphores**
  - general mechanism to synchronize execution
  - blocking wait() operation, non-blocking post() operation
  - supported by both hardware and software threads

- **mutexes**
  - specific mechanism to protect critical sections (e.g. read-modify-write to shared memory)
  - a thread can only release a mutex it “owns” (has previously locked)
  - supported by both hardware and software threads

- **thread termination**
  - a thread can block until another thread exits
  - currently, only software threads can join(), but all threads can exit()
High-Level Communication in ReconOS

- shared memory
  - all threads have direct access to the entire memory space
  - accesses need to be synchronized using semaphores or mutexes
  - dedicated hardware support for burst transfers
High-Level Communication in ReconOS

- **shared memory**
  - all threads have direct access to the entire memory space
  - accesses need to be synchronized using semaphores or mutexes
  - dedicated hardware support for burst transfers

- **message queues**
  - can block if queue is empty / full
  - combined communication and synchronization primitive
  - supported by both hardware and software threads
  - dedicated hardware FIFOs for hardware threads
Outline

- motivation
- high-level communication and synchronization
- low-level communication and synchronization
- performance & overheads
- conclusion & outlook
Hardware Architecture

- developed on Xilinx Virtex-II Pro, Virtex-4 FX FPGAs
- based on CoreConnect bus topology
- OS kernel is eCos for PowerPC ported to Virtex

Diagram:
- CPU
- OS kernel
- I/O controller
- memory controller
- external memory
- system buses (PLB, DCR)
- interrupt controller
Hardware Architecture

- developed on Xilinx Virtex-II Pro, Virtex-4 FX FPGAs
- based on CoreConnect bus topology
- OS kernel is eCOS for PowerPC ported to Virtex
Hardware Architecture

- developed on Xilinx Virtex-II Pro, Virtex-4 FX FPGAs
- based on CoreConnect bus topology
- OS kernel is eCos for PowerPC ported to Virtex
Hardware Architecture

- developed on Xilinx Virtex-II Pro, Virtex-4 FX FPGAs
- based on CoreConnect bus topology
- OS kernel is eCos for PowerPC ported to Virtex
Hardware Thread

transitions occur only when OS interface is ready

user logic

local RAM

OS Interface

hardware thread

OS synchronization state machine

clk
reset

i_osif
o_osif

IDLE
READ
POST
RUN
WRITE

/ sem_wait
(C_SEM_A)

done = '1'/
run <= '1'

/ shm_read()

done = '0'/
run <= '0'

/ sem_post
(C_SEM_B)

/ shm_write()

run
done

i_osif
o_osif

E.Lübbers & M.Platzner, University of Paderborn
ERSA '08
Hardware Thread

- a hardware thread consists of two parts
Hardware Thread

- A hardware thread consists of two parts
  - An OS synchronization state machine
    - Synchronizes thread with operating system calls
    - Serializes access to OS objects via the OS interface
    - Can be blocked by the OS interface

![Hardware Thread Diagram](image-url)
a hardware thread consists of two parts

- an OS synchronization state machine
  - synchronizes thread with operating system calls
  - serializes access to OS objects via the OS interface
  - can be blocked by the OS interface

- parallel “user processes”
  - communicate with OS synchronization state machine
  - can directly access local memory blocks
  - are not necessarily blocked
ReconOS API for Hardware Threads

- VHDL function library
- used similar to software API
- may only be used inside OS synchronization state machine

```vhdl
osif_fsm: process(clk, reset)
begin
  if (reset = '1') then
    state <= IDLE;
    run <= '0';
    reconos_reset(o_osif, i_osif);
  elsif rising_edge(clk) then
    reconos_begin(o_osif, i_osif);
    if reconos_ready(i_osif) then
      case state is
        when IDLE =>
          reconos_sem_wait(o_osif, i_osif, C_SEM_A);
          state <= READ;
        when READ =>
          reconos_shm_read_burst(o_osif, i_osif, local_address, global_address);
          state <= RUN;
        when RUN =>
          run <= '1';
          if done = '1' then
            run <= '0';
            state <= WRITE;
          end if;
        when WRITE =>
          reconos_shm_write_burst(o_osif, i_osif, local_address, global_address);
          state <= POST;
        when POST =>
          reconos_sem_post(o_osif, i_osif, C_SEM_B);
          state <= IDLE;
        when others => null;
      end case;
    end if;
  end if;
end process;
```

**Diagram:**
- OS synchronization state machine
- Hardware thread
- Transitions occur only when OS interface is ready

**Components:**
- OS Interface
- Clk
- Reset
- Hardware thread
- User logic
- Local RAM
- IDLE
- READ
- POST
- WRITE
- Run
- Done

**Functions:**
- `reconos_reset`
- `reconos_begin`
- `reconos_sem_wait`
- `reconos_shm_read_burst`
- `reconos_shm_write_burst`
- `reconos_sem_post`
OS Interface

64 bit PLB (memory bus)

- PLB slave attachment
- bus master controller

32 Bit DCR (control bus)

- DCR slave attachment
- FIFO manager

command decoder

OS interface
to interrupt controller

local RAM

user logic

hardware thread
to/from other threads

E.Lübbers & M.Platzner, University of Paderborn ERSA ‘08
OS Interface

- processes requests from hardware thread
  - handles blocking and resuming of hardware thread
OS Interface

- processes requests from hardware thread
  - handles blocking and resuming of hardware thread
- relays OS object interactions to CPU
  - DCR interface with bus-addressable registers
  - dedicated interrupt
OS Interface

- processes requests from hardware thread
  - handles blocking and resuming of hardware thread

- relays OS object interactions to CPU
  - DCR interface with bus-addressable registers
  - dedicated interrupt

- executes memory accesses
  - PLB master interface
  - direct access to entire system’s address space (memory and peripherals)
OS Interface

- processes requests from hardware thread
  - handles blocking and resuming of hardware thread

- relays OS object interactions to CPU
  - DCR interface with bus-addressable registers
  - dedicated interrupt

- executes memory accesses
  - PLB master interface
  - direct access to entire system’s address space (memory and peripherals)

- dedicated FIFO channels
  - provide high-throughput hardware support for message passing
Delegate Threads
Delegate Threads

- basic mechanism
  - a delegate thread in software is associated with every hardware thread
  - the delegate thread calls the OS kernel on behalf of the hardware thread
  - all kernel responses are relayed back to the hardware thread
Delegate Threads

- **basic mechanism**
  - a delegate thread in software is associated with every hardware thread
  - the delegate thread calls the OS kernel on behalf of the hardware thread
  - all kernel responses are relayed back to the hardware thread

- **advantages**
  - no modification of the kernel required
  - extremely flexible
  - transparent to kernel and other threads
Delegate Threads

- basic mechanism
  - a delegate thread in software is associated with every hardware thread
  - the delegate thread calls the OS kernel on behalf of the hardware thread
  - all kernel responses are relayed back to the hardware thread

- advantages
  - no modification of the kernel required
  - extremely flexible
  - transparent to kernel and other threads

- drawbacks
  - increased overhead due to interrupt processing and context switch
Hardware Support for Message Passing
Hardware Support for Message Passing

- direct HW thread-to-thread communication can be inefficient
Hardware Support for Message Passing

- direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
direct HW thread-to-thread communication can be inefficient
- shared memory: external memory access + bus arbitration
direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
  - message passing: OS overhead (interrupt processing in CPU)
Hardware Support for Message Passing

- Direct HW thread-to-thread communication can be inefficient
  - Shared memory: external memory access + bus arbitration
  - Message passing: OS overhead (interrupt processing in CPU)
Hardware Support for Message Passing

- direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
  - message passing: OS overhead (interrupt processing in CPU)
- direct connection of HW threads via buffered point-to-point links
- transparently supported by existing message queue API
- message routing dependent on current thread location (planned)
Hardware Support for Message Passing

- direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
  - message passing: OS overhead (interrupt processing in CPU)
- direct connection of HW threads via buffered point-to-point links
- transparently supported by existing message queue API
- message routing dependent on current thread location (planned)
Hardware Support for Message Passing

- direct HW thread-to-thread communication can be inefficient
  - shared memory: external memory access + bus arbitration
  - message passing: OS overhead (interrupt processing in CPU)
- direct connection of HW threads via buffered point-to-point links
- transparently supported by existing message queue API
- message routing dependent on current thread location (planned)
Outline

- motivation
- high-level communication and synchronization
- low-level communication and synchronization
- performance & overheads
- conclusion & outlook
Synchronization Overheads
Synchronization Overheads

- synthetic hardware and software threads
  - semaphore and mutex processing time
    (post → wait / unlock → lock)
Synchronization Overheads

- synthetic hardware and software threads
  - semaphore and mutex processing time
    (post → wait / unlock → lock)
- OS calls involving hardware exhibit higher latencies
- limited impact on system performance
  - logic resources mainly used for heavy data-parallel processing
  - less synchronization-intensive control dominated code

<table>
<thead>
<tr>
<th>Configuration</th>
<th>with data cache</th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>semaphore</td>
<td>mutex</td>
</tr>
<tr>
<td>SW → SW</td>
<td>3.39</td>
<td>4.53</td>
</tr>
<tr>
<td>SW → HW</td>
<td>4.71</td>
<td>6.29</td>
</tr>
<tr>
<td>HW → SW</td>
<td>10.74</td>
<td>14.49</td>
</tr>
<tr>
<td>HW → HW</td>
<td>11.90</td>
<td>14.60</td>
</tr>
</tbody>
</table>

All values are given in $\mu$s
## Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74 μs 170.80 MB/s</td>
<td>46.41 μs 168.34 MB/s</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54 μs 192.71 MB/s</td>
<td>40.55 μs 192.66 MB/s</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51 μs 58.96 MB/s</td>
<td>625.00 μs 12.50 MB/s</td>
</tr>
<tr>
<td>HW→HW (mbox read)</td>
<td>61.42 μs 127.20 MB/s</td>
<td>61.42 μs 127.20 MB/s</td>
</tr>
<tr>
<td>HW→HW (mbox write)</td>
<td>61.45 μs 127.14 MB/s</td>
<td>61.45 μs 127.14 MB/s</td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500 μs 0.13 MB/s</td>
<td>3740000 μs 0.02 MB/s</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510 μs 0.13 MB/s</td>
<td>3740000 μs 0.02 MB/s</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data.

---

**Diagram:**

- CPU
- other peripherals
- memory controller
- DRAM
- system buses (PLB/DCR)
- OS interface
- FIFO
- hw thread

---

E.Lübbers & M.Platzner, University of Paderborn  
ERSA ‘08
## Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th></th>
<th>without data cache</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>μs</td>
<td>MB/s</td>
<td>μs</td>
<td>MB/s</td>
</tr>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74</td>
<td>170.80</td>
<td>46.41</td>
<td>168.34</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54</td>
<td>192.71</td>
<td>40.55</td>
<td>192.66</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51</td>
<td>58.96</td>
<td>625.00</td>
<td>12.50</td>
</tr>
<tr>
<td>HW→HW (mbox read)</td>
<td>61.42</td>
<td>127.20</td>
<td>61.42</td>
<td>127.20</td>
</tr>
<tr>
<td>HW→HW (mbox write)</td>
<td>61.45</td>
<td>127.14</td>
<td>61.45</td>
<td>127.14</td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500</td>
<td>0.13</td>
<td>374000</td>
<td>0.02</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510</td>
<td>0.13</td>
<td>374000</td>
<td>0.02</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data.
## Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$s</td>
<td>MB/s</td>
</tr>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74</td>
<td>170.80</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54</td>
<td>192.71</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51</td>
<td>58.96</td>
</tr>
<tr>
<td>HW→HW (mbox read)</td>
<td>61.42</td>
<td>127.20</td>
</tr>
<tr>
<td>HW→HW (mbox write)</td>
<td>61.45</td>
<td>127.14</td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500</td>
<td>0.13</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510</td>
<td>0.13</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data.

---

**System Buses (PLB/DCR):**

- CPU
- Other peripherals
- Memory controller
- DRAM
- System buses (PLB/DCR)

**OS Interface:**

- FIFO
- hw thread

**Software Mailboxes:**

- hw thread
### Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>μs</td>
<td>MB/s</td>
</tr>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74</td>
<td>170.80</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54</td>
<td>192.71</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51</td>
<td>58.96</td>
</tr>
<tr>
<td><strong>HW→HW (mbox read)</strong></td>
<td><strong>61.42</strong></td>
<td><strong>127.20</strong></td>
</tr>
<tr>
<td><strong>HW→HW (mbox write)</strong></td>
<td><strong>61.45</strong></td>
<td><strong>127.14</strong></td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500</td>
<td>0.13</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510</td>
<td>0.13</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data

![System Diagram](https://via.placeholder.com/150)

- hardware FIFOs
- software mailboxes

---

E.Lübers & M.Platzner, University of Paderborn

ERSA '08
## Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mu$s</td>
<td>MB/s</td>
</tr>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74</td>
<td>170.80</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54</td>
<td>192.71</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51</td>
<td>58.96</td>
</tr>
<tr>
<td>HW→HW (mbox read)</td>
<td>61.42</td>
<td>127.20</td>
</tr>
<tr>
<td>HW→HW (mbox write)</td>
<td>61.45</td>
<td>127.14</td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500</td>
<td>0.13</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510</td>
<td>0.13</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data

---

**Diagram:**

![System diagram](diagram.png)

- **CPU**: Central Processing Unit
- **other peripherals**: Additional hardware components
- **memory controller**: Interface to DRAM
- **DRAM**: Dynamic Random Access Memory
- **system buses (PLB/DCR)**: Interconnect busses
- **OS interface**: Operating system interface
- **FIFO**: First-In, First-Out buffer
- **hw thread**: Hardware thread

---

E.Lübbers & M.Platzner, University of Paderborn **ERSA '08**
Communication Performance

<table>
<thead>
<tr>
<th>Operation</th>
<th>with data cache</th>
<th></th>
<th>without data cache</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>μs</td>
<td>MB/s</td>
<td>μs</td>
</tr>
<tr>
<td>MEM→HW (burst read)</td>
<td>45.74</td>
<td>170.80</td>
<td>46.41</td>
</tr>
<tr>
<td>HW→MEM (burst write)</td>
<td>40.54</td>
<td>192.71</td>
<td>40.55</td>
</tr>
<tr>
<td>MEM→SW→MEM (memcpy)</td>
<td>132.51</td>
<td>58.96</td>
<td>625.00</td>
</tr>
<tr>
<td>HW→HW (mbox read)</td>
<td>61.42</td>
<td>127.20</td>
<td>61.42</td>
</tr>
<tr>
<td>HW→HW (mbox write)</td>
<td>61.45</td>
<td>127.14</td>
<td>61.45</td>
</tr>
<tr>
<td>SW→HW (mbox read)</td>
<td>58500</td>
<td>0.13</td>
<td>374000</td>
</tr>
<tr>
<td>HW→SW (mbox write)</td>
<td>58510</td>
<td>0.13</td>
<td>374000</td>
</tr>
</tbody>
</table>

All operations were run for 8 kBytes of data

- comparison between memory access and FIFO transfers
  - FIFOs are faster for HW thread to HW thread communications (+40%)
  - no additional load on memory system or CPU
  - improves thread-parallelism
Conclusion & Outlook
Conclusion & Outlook

- common set of high-level communication and synchronization objects for hard- and software unifies programming model
- existing operating systems can be extended with mechanisms for low-level communication and synchronization between hardware threads and kernel
- acceptable performance in benchmarks and larger case studies
Conclusion & Outlook

- common set of high-level communication and synchronization objects for hard- and software unifies programming model
- existing operating systems can be extended with mechanisms for low-level communication and synchronization between hardware threads and kernel
- acceptable performance in benchmarks and larger case studies

future work
- extension of hardware FIFOs to allow direct access from software threads
- fusion of shared memory and message-passing interfaces to hardware threads
Thank you

www.reconos.de
OS Overheads (Area)

- command decoder: 23%
- bus slave registers: 12%
- fifo manager: 5%
- PLB IPIF: 59%

- total OSIF slice count: 1213 slices
  - most of this taken up by PLB IPIF logic
## Supported OS Calls

- **Semaphores (counting and binary)**
  - reconos_semaphore_post()
  - reconos_semaphore_wait()

- **Mutexes**
  - reconos_mutex_lock()
  - reconos_mutex_trylock()
  - reconos_mutex_unlock()
  - reconos_mutex_release()

- **Condition Variables**
  - reconos_cond_wait()
  - reconos_cond_signal()
  - reconos_cond_broadcast()

- **Mailboxes**
  - reconos_mbox_get()
  - reconos_mbox_tryget()
  - reconos_mbox_put()
  - reconos_mbox_tryput()

- **Memory access**
  - reconos_read()
  - reconos_write()
  - reconos_read_burst()
  - reconos_write_burst()
ReconOS Software API (POSIX)

- standard POSIX thread creation
- ReconOS hardware thread creation

```c
mqd_t my_mbox;
sem_t my_sem;

pthread_t thread;
pthread_attr_t thread_attr;
...

pthread_attr_init(&thread_attr);

pthread_create(
    &thread,       // thread object
    &thread_attr,  // attributes
    thread_entry,  // entry point
    ( void * ) data // entry data
);

mqd_t my_mbox;
sem_t my_sem;
reconos_res_t thread_resources[2] = {
    { &my_mbox, POSIX_MQD_T },
    { &my_sem, POSIX_SEM_T } }
);

rthread thread;
pthread_attr_t thread_swattr;
rthread_attr_t thread_hwattr;
...

pthread_attr_init(&thread_swattr);
rthread_attr_init(&thread_hwattr);
rthread_attr_setslotnum(&thread_hwattr, 0);
rthread_attr_setresources(&thread_hwattr,
                            thread_resources, 2);

rthread_create(
    &thread,       // thread object
    &thread_swattr, // software attributes
    &thread_hwattr, // hardware attributes
    ( void * ) data // entry data
);
```

- standard POSIX thread creation
- ReconOS hardware thread creation
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Multi-Cycle Commands

- transfer of multiple parameters and return values with a single VHDL call
- distributes execution of an FSM state across multiple clock cycles
Toolchain

- software thread (C)
- ReconOS repository
- hardware thread (VHDL)
Toolchain

- software threads are written in C
  - using the eCos software API

- hardware threads are written in VHDL
  - using the ReconOS VHDL API

- architecture generation
  - automatically inserts OS interfaces and hardware threads into Xilinx EDK platform templates
  - configures and builds static eCos library
Toolchain

- software threads are written in C
  - using the eCos software API

- hardware threads are written in VHDL
  - using the ReconOS VHDL API

- architecture generation
  - automatically inserts OS interfaces and hardware threads into Xilinx EDK platform templates
  - configures and builds static eCos library
Toolchain

- software threads are written in C
  - using the eCos software API

- hardware threads are written in VHDL
  - using the ReconOS VHDL API

- architecture generation
  - automatically inserts OS interfaces and hardware threads into Xilinx EDK platform templates
  - configures and builds static eCos library

- eCos extensions
  - hardware thread object encapsulating delegate thread and OS interface “driver”
  - profiling support to track the state of the hardware threads' OS synchronization state machines
Case Study - Image Processing Filter

- **three threads**
  - capture image from Ethernet
  - apply LaPlacian filter
  - display image on VGA monitor

- **threads communicate through shared memory**
  - image resolution: 320x240 pixels, 8 bit greyscale
  - image data organized into blocks (e.g. 40 lines = 1 block)
  - a block is protected by two semaphores
    - “ready” semaphore: data can be safely written into this block
    - “new” semaphore: new data is available in this block
Case Study - Image Processing Filter

- **three threads**
  - capture image from Ethernet
  - apply LaPlacian filter
  - display image on VGA monitor

- **threads communicate through shared memory**
  - image resolution: 320x240 pixels, 8 bit greyscale
  - image data organized into blocks (e.g. 40 lines = 1 block)
  - a block is protected by two semaphores
    - “ready” semaphore: data can be safely written into this block
    - “new” semaphore: new data is available in this block
Case Study - Results

![Bar chart showing results for different window sizes and configurations.](chart.png)

- **frames/s**
- **window size**

- **SW-SW-SW**
- **SW-HW-SW**
- **SW-HW-HW**
- **SW-HW-HW double buffered**

Values for different configurations:
- 3 frames/s: SW-SW-SW (16.2), SW-HW-SW (19.0), SW-HW-HW (23.5), SW-HW-HW double buffered (25.5)
- 5 frames/s: SW-SW-SW (8.5), SW-HW-SW (18.6), SW-HW-HW (23.2), SW-HW-HW double buffered (25.4)
Case Study - Results