Building Multi-Processor FPGA Systems

*Hands-on Tutorial to Using FPGAs and Linux*

Chris Martin
Member Technical Staff Embedded Applications
Agenda

Introduction
Problem: How to Integrate Multi-Processor Subsystems
Why…
  – Why would you do this?
  – Why use FPGAs?
Lab 1: Getting Started - Booting Linux and Boot-strapping NIOS
Building Hardware: FPGA Hardware Tools & Build Flow
Break (10 minutes)
Lab 2: Inter-Processor Communication and Shared Peripherals
Building/Debugging NIOS Software: Software Tools & Build Flow
Lab 3: Locking and Tetris
Building/Debugging ARM Software: Software Tools & Build Flow
References
Q&A – All through out.
The Problem – Integrating Multi-Processor Subsystems

- Given a system with multiple processor subsystems, these architecture decisions must be considered:
  - Inter-processor communication
  - Partitioning/sharing Peripherals (locking required)
  - Bandwidth & Latency Requirements
Why Do We Need to Integrate Multi-Processor Subsystems?

- May have inherited processor subsystem from another development team or 3rd party
  - Risk Mitigation by reducing change
- Fulfill Latency and Bandwidth Requirements
  - Real-time Considerations
  - If main processor not Real-Time enabled, can add a real-time processor subsystem
- Design partition / Sandboxing
  - Break the system into smaller subsystems to service task
  - Smaller task can be designed easily
- Leverage Software Resources
  - Sometimes problem is resolved in less time by Processor/Software rather than Hardware design
  - Sequencers, State-machines
Why do we want to integrate with FPGA? (or rather, HOW can FPGAs help?)

- Huge number of processor subsystems can be implemented
- Bandwidth & Latency can be tailored
  - Addresses Real-time aspects of System Solution
  - FPGA logic has flexible interconnect
  - Trade Data width with clock frequency with latency
- Experimentation
  - Allows you to experiment changing microprocessor subsystem hardware designs
  - Altera FPGA under-the-hood
  - However: Generic Linux interfaces used and can be applied in any Linux system.

Simple Multiprocessor System

- ARM
- Mailbox
- NIOS
- Shared Peripheral
- N Peripheral

- And, why is Altera involved with Embedded Linux…
Why is Altera Involved with Embedded Linux?

More than 50% of FPGA designs include an embedded processor, and growing.

Many embedded designs using Linux

Open-source re-use.

- Altera Linux Development Team actively contributes to Linux Kernel

Source: Gartner September 2010
SoCKit Board Architecture Overview

- **Lab focus**
  - UART
  - DDR3
  - LEDs
  - Buttons
SoC/FPGA Hardware Architecture Overview

- **ARM-to-FPGA Bridges**
  - Data Width configurable

- **FPGA**
  - 42K Logic Macros
  - Using no more than 14%
Lab 1: Getting Started
Booting Linux and Boot-strapping NIOS

Topics Covered:
- Configuring FPGA from SD/MMC and U-Boot
- Booting Linux on ARM Cortex-A9
- Configuring Device Tree
- Resetting and Booting NIOS Processor
- Building and compiling simple Linux Application

Key Example Code Provided:
- C code for downloading NIOS code and resetting NIOS from ARM
- Using U-boot to set ARM peripheral security bits

Full step-by-step instructions are included in lab manual.
Lab 1: Hardware Design Overview

NIOS Subsystem
- 1 NIOS Gen 2 processor
- 64k combined instruction/data RAM (On-Chip RAM)
- GPIO peripheral

ARM Subsystem
- 2 Cortex-A9 (only using 1)
- DDR3 External Memory
- SD/MMC Peripheral
- UART Peripheral
### Lab1: Programmer View - Processor Address Maps

<table>
<thead>
<tr>
<th>NIOS</th>
<th>ARM Cortex-A9</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address Base</strong></td>
<td><strong>Peripheral</strong></td>
</tr>
<tr>
<td>0xFFC0_2000</td>
<td>ARM UART</td>
</tr>
<tr>
<td>0x0003_0000</td>
<td>GPIO (LEDs)</td>
</tr>
<tr>
<td>0x0002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0x0000_0000</td>
<td>On-chip RAM</td>
</tr>
<tr>
<td><strong>Address Base</strong></td>
<td><strong>Peripheral</strong></td>
</tr>
<tr>
<td>0xFFC0_2000</td>
<td>UART</td>
</tr>
<tr>
<td>0xC003_0000</td>
<td>GPIO (LEDs)</td>
</tr>
<tr>
<td>0xC002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0xC000_0000</td>
<td>On-chip RAM</td>
</tr>
</tbody>
</table>
# Lab 1: Peripheral Registers

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Address Offset</th>
<th>Access</th>
<th>Bit Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sys ID</td>
<td>0x0</td>
<td>RO</td>
<td>[31:0] – System ID. Lab Default = 0x00001ab1</td>
</tr>
<tr>
<td>UART</td>
<td>0x14</td>
<td>RO</td>
<td>Line Status Register [5] – TX FIFO Empty [0] – Data Ready (RX FIFO not-Empty)</td>
</tr>
<tr>
<td>UART</td>
<td>0x30</td>
<td>R/W</td>
<td>Shadow Receive Buffer Register [7:0] – RX character from serial input</td>
</tr>
<tr>
<td>UART</td>
<td>0x34</td>
<td>R/W</td>
<td>Shadow Transmit Register [7:0] – TX character to serial output</td>
</tr>
</tbody>
</table>
Lab 1: Processor Resets Via Standard Linux GPIO Interface

- NIOS resets connected to GPIO
- GPIO driver uses /sys/class/gpio interface

```c
int main(int argc, char** argv)
{
    int fd, gpio=168;
    char buf[MAX_BUF];

    /* Export: echo ### > /sys/class/gpio/export */
    fd = open("/sys/class/gpio/export", O_WRONLY);
    sprintf(buf, "%d", gpio);
    write(fd, buf, strlen(buf));
    close(fd);

    /* Set direction to Out: */
    /* echo "out" > /sys/class/gpio/gpio###/direction */
    sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);
    fd = open(buf, O_WRONLY);
    write(fd, "out", 3); /* write(fd, "in", 2); */
    close(fd);

    /* Set GPIO Output High or Low */
    /* echo 1 > /sys/class/gpio/gpio###/value */
    sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);
    fd = open(buf, O_WRONLY);
    write(fd, "1", 1); /* write(fd, "0", 1); */
    close(fd);

    /* Unexport: echo ### > /sys/class/gpio/unexport */
    fd = open("/sys/class/gpio/unexport", O_WRONLY);
    sprintf(buf, "%d", gpio);
    write(fd, buf, strlen(buf));
    close(fd);
}
```
Lab 1: Loading External Processor Code Via Standard Linux shared memory (mmap)

- NIOS RAM address accessed via mmap()
- Can be shared with other processes
- R/W during load
- Read-only protection after load

```c
/* Map Physical address of NIOS RAM to virtual address segment with Read/Write Access */
fd = open("/dev/mem", O_RDWR);
load_address = mmap(NULL, 0x10000,
    PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0xc0000000);

/* Set size of code to load */
load_size = sizeof(nios_code)/sizeof(nios_code[0]);

/* Load NIOS Code */
for(i=0; i < load_size ;i++)
{
    *(load_address+i) = nios_code[i];
}

/* Set load address segment to Read-Only */
mprotect(load_address, 0x10000, PROT_READ);

/* Un-map load address segment */
munmap(load_address, 0x10000);
```
Post-Lab 1 Additional Topics

Hardware Design Flow and FPGA Boot with U-boot and SD/MMC
Building Hardware: Qsys (Hardware System Design Tool) User Interface

Connections between cores

Interfaces Exported In/out of system
Hardware and Software Work Flow Overview

**Inputs:**
- Hardware Design (Qsys or RTL or Both)

**Outputs (to load on boot media):**
- Preloader and U-boot Images
- FPGA Programmation File: Raw Binary Format (RBF)
- Device Tree Blob
**SDCARD Layout**

- **Partition 1: FAT**
  - Uboot scripts
  - FPGA HW Designs (RBF)
  - Device Tree Blobs
  - zImage
  - Lab material

- **Partition 2: EXT3** – Rootfs

- **Partition 3: Raw**
  - Uboot/preloader

- **Partition 4: EXT3** – Kernel src
## Updating SD Cards

<table>
<thead>
<tr>
<th>File</th>
<th>Update Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>zImage</td>
<td>Mount DOS SD card partition 1 and replace file with new one:</td>
</tr>
<tr>
<td></td>
<td>$ sudo mkdir sdcard</td>
</tr>
<tr>
<td>soc_system.rbf</td>
<td>$ sudo cp &lt;file_name&gt; sdcard/</td>
</tr>
<tr>
<td>soc_system.dtb</td>
<td>$ sudo umount sdcard</td>
</tr>
<tr>
<td>u-boot.scr</td>
<td>$ sudo dd if=preloader-mkpimage.bin of=/dev/sdx3 bs=64k seek=0</td>
</tr>
<tr>
<td>preloader-mkpimage.bin</td>
<td>$ sudo dd if=preloader-mkpimage.bin of=/dev/sdx3 bs=64k seek=0</td>
</tr>
<tr>
<td>u-boot-socfpga_cyclone5.img</td>
<td>$ sudo dd if=u-boot-socfpga_cyclone5.img of=/dev/sdx3 bs=64k seek=4</td>
</tr>
<tr>
<td>root filesystem</td>
<td>$ sudo dd if=altera-gsrd-image-socfpga_cyclone5.ext3 of=/dev/sdx2</td>
</tr>
</tbody>
</table>

- More info found on Rocketboards.org

- Automated Python Script to build SD Cards:
  - `make_sdimage.py`
Lab 2: Mailboxes
NIOS/ARM Communication

Topics Covered:
- Altera Mailbox Hardware IP

Key Example Code Provided:
- C code for sending/receiving messages via hardware Mailbox IP
  - NIOS & ARM C Code
  - Simple message protocol
  - Simple Command parser

Full step-by-step instructions are included in lab manual.
- User to add second NIOS processor mailbox control.
Lab 2: Hardware Design Overview

NIOS 0 & 1 Subsystems
- NIOS Gen 2 processor
- 64k combined instruction/data RAM
- GPIO (4 out, LED)
- GPIO (2 in, Buttons)
- Mailbox

ARM Subsystem
- 2 Cortex-A9 (only using 1)
- DDR3 External Memory
- SD/MMC Peripheral
- UART Peripheral
# Lab 2: Programmer View - Processor Address Maps

## NIOS 0 & 1

<table>
<thead>
<tr>
<th>Address Base</th>
<th>Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0_2000</td>
<td>ARM UART</td>
</tr>
<tr>
<td>0x0007_8000</td>
<td>Mailbox (from ARM)</td>
</tr>
<tr>
<td>0x0007_0000</td>
<td>Mailbox (to ARM)</td>
</tr>
<tr>
<td>0x0005_0000</td>
<td>GPIO (In Buttons)</td>
</tr>
<tr>
<td>0x0003_0000</td>
<td>GPIO (Out LEDs)</td>
</tr>
<tr>
<td>0x0002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0x0000_0000</td>
<td>On-chip RAM</td>
</tr>
</tbody>
</table>

## ARM Cortex-A9

<table>
<thead>
<tr>
<th>Address Base</th>
<th>Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFC0_2000</td>
<td>UART</td>
</tr>
<tr>
<td>0x0007_8000</td>
<td>Mailbox (to NIOS 1)</td>
</tr>
<tr>
<td>0x0007_0000</td>
<td>Mailbox (from NIOS 1)</td>
</tr>
<tr>
<td>0x0006_8000</td>
<td>Mailbox (to NIOS 0)</td>
</tr>
<tr>
<td>0x0006_0000</td>
<td>Mailbox (from NIOS 0)</td>
</tr>
<tr>
<td>0xC003_0000</td>
<td>GPIO (LEDs)</td>
</tr>
<tr>
<td>0xC002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0xC001_0000</td>
<td>NIOS 1 RAM</td>
</tr>
<tr>
<td>0xC000_0000</td>
<td>NIOS 0 RAM</td>
</tr>
</tbody>
</table>
## Lab 2: Additional Peripheral (Mailbox) Registers

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Address Offset</th>
<th>Access</th>
<th>Bit Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mailbox</td>
<td>0x0</td>
<td>R/W</td>
<td>[31:0] – RX/TX Data</td>
</tr>
<tr>
<td>Mailbox</td>
<td>0x8</td>
<td>R/W</td>
<td>[1] – RX Message Queue Has Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[0] – TX Message Queue Empty</td>
</tr>
</tbody>
</table>
Key Multi-Processor System Design Points

- **Startup/Shutdown**
  - Processor
  - Peripheral
  - Covered in Lab 1.

- **Communication between processors**
  - What is the physical link?
  - What is the protocol & messaging method?
  - Message Bandwidth & Latency
  - Covered in Lab 2

- **Partitioning peripherals**
  - Declare dedicated peripherals – only connected/controlled by one processor
  - Declare shared peripherals – Connected/controlled by multiple processors
  - Decide Upon Locking Mechanism
  - Covered in Lab 3
LAB 2: Designing a Simple Message Protocol

- **Design Decisions:**
  - Short Length: A single 32-bit word
  - Human Readable
  - Message transactions are closed-loop. Includes ACK/NACK

- **Format:**
  - Message Length: Four Bytes
  - **First Byte** is ASCII character denoting message type.
  - **Second Byte** is ASCII char from 0-9 denoting processor number.
  - **Third Byte** is ASCII char from 0-9 denoting message data.
  - **Fourth Byte** is always null character ‘\0’ to terminate string (human readable).

- **Message Types:**
  - “G00”: Give Access to UART (Push)
  - “A00”: ACK
  - “N00”: NACK

- **Can be Extended:**
  - “L00”: LED Set/Ready
  - “B00”: Button Pressed
  - “R00”: Request UART Access (Pull)

---

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Byte 1</th>
<th>Byte 2</th>
<th>Byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘L’</td>
<td>‘0’</td>
<td>‘0’</td>
<td>‘\0’</td>
</tr>
<tr>
<td>‘A’</td>
<td>‘0’</td>
<td>‘0’</td>
<td>‘\0’</td>
</tr>
</tbody>
</table>

![Cortex-A9](Cortex-A9.png) ![NIOS 0](NIOS 0.png)
Lab 2: Inter-Processor Communication with Mailbox HW Via Standard Linux Shared Memory (mmap)

- Wait for Mailbox Hardware message empty flag
- Send message (4 bytes)
- Disable ARM/Linux Access to UART
- Wait for RX message received flag
- Re-enable ARM/Linux UART Access

```c
/* Map Physical address of Mailbox to virtual address segment with Read/Write Access */
fd = open("/dev/mem", O_RDWR);
mbox0_address = mmap(NULL, 0x10000, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0xff260000);
<snip>
/* Waiting for Message Queue to empty */
while(((volatile int*)(mbox0_address+0x2000+2) & 1) != 0 ) {} 
/* Send Granted/Go message to NIOS */
send_message = "G00";
*(mbox0_address+0x2000) = *(int *)send_message;

/* Disable ARM/Linux Access to UART (be careful here)*/
config.c_cflag &= ~CREAD;
if(tcsetattr(fd, TCSAFLUSH, &config) < 0) { }

/* Wait for Received Message */
while(((volatile int*)(mbox0_address+2) & 2) == 0 ) {} 

/* Re-enable UART Access */
config.c_cflag |= CREAD;
tcsetattr(fd, TCSAFLUSH, &config);
/* Read Received Message */
printf(" - Message Received. DATA = '%s'.\n", (char*)(mbox0_address));
```
Post-Lab 2 Additional Topic

Using Eclipse to Debug: NIOS Software Build Tools
Nios II SBT for Eclipse key features:
- New project wizards and software templates
- Compiler for C and C++ (GNU)
- Source navigator, editor, and debugger
- Eclipse project-based tools
- Download code to hardware
Lab 3: Putting It All Together – Tetris!
Combining Locking and Communication

Topics Covered:
- Linux Mutex

Key Example Code Provided:
- C code showcasing using Mutexes for locking shared peripheral access
- C code for multiple processor subsystem bringup and shutdown

Full step-by-step instructions are included in lab manual.
- User to add code for second NIOS processor bringup, shutdown and locking/control.
Lab 3: Hardware Design Overview *(Same As Lab 2)*

**NIOS 0 & 1 Subsystems**
- NIOS Gen 2 processor
- 64k combined instruction/data RAM
- GPIO (4 out, LED)
- GPIO (2 in, Buttons)
- Mailbox

**ARM Subsystem**
- 2 Cortex-A9 (only using 1)
- DDR3 External Memory
- SD/MMC Peripheral
- UART Peripheral
### Lab 3: Programmer View - Processor Address Maps

#### NIOS 0 & 1

<table>
<thead>
<tr>
<th>Address Base</th>
<th>Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0_2000</td>
<td>ARM UART</td>
</tr>
<tr>
<td>0x0007_8000</td>
<td>Mailbox (from ARM)</td>
</tr>
<tr>
<td>0x0007_0000</td>
<td>Mailbox (to ARM)</td>
</tr>
<tr>
<td>0x0005_0000</td>
<td>GPIO (In Buttons)</td>
</tr>
<tr>
<td>0x0003_0000</td>
<td>GPIO (Out LEDs)</td>
</tr>
<tr>
<td>0x0002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0x0000_0000</td>
<td>On-chip RAM</td>
</tr>
</tbody>
</table>

#### ARM Cortex-A9

<table>
<thead>
<tr>
<th>Address Base</th>
<th>Peripheral</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFC0_2000</td>
<td>UART</td>
</tr>
<tr>
<td>0x0007_8000</td>
<td>Mailbox (to NIOS 1)</td>
</tr>
<tr>
<td>0x0007_0000</td>
<td>Mailbox (from NIOS 1)</td>
</tr>
<tr>
<td>0x0006_8000</td>
<td>Mailbox (to NIOS 0)</td>
</tr>
<tr>
<td>0x0006_0000</td>
<td>Mailbox (from NIOS 0)</td>
</tr>
<tr>
<td>0xC003_0000</td>
<td>GPIO (LEDs)</td>
</tr>
<tr>
<td>0xC002_0000</td>
<td>System ID</td>
</tr>
<tr>
<td>0xC001_0000</td>
<td>NIOS 1 RAM</td>
</tr>
<tr>
<td>0xC000_0000</td>
<td>NIOS 0 RAM</td>
</tr>
</tbody>
</table>
Available Linux Locking/Synchronization Mechanisms

Need to share peripherals
- Choose a Locking Mechanism

Available in Linux
- Mutex <- Chosen for this Lab
- Completions
- Spinlocks
- Semaphores
- Read-copy-update (decent for multiple readers, single writer)
- Seqlocks (decent for multiple readers, single writer)

Available for Linux
- MCAPI - openmcapi.org
Tetris Message Protocol – Extended from Lab 2

**NIOS Control Flow:**
- Wait for button press
- Send Button press message
- Wait for ACK (Free to write to LED GPIO)
- Write to LED GPIO
- Send LED ready msg
- Wait for ACK

**ARM Control Flow:**
- Wait for button press message
- Lock LED GPIO Peripheral
- Send ACK (Free to write to LED GPIO)
- Wait for LED ready msg
- Send ACK
- Read LED value
- Release Lock/Mutex
Lab 3: Locking Hardware Peripheral Access Via Linux Mutex

- In this example, LED GPIO is accessed by multiple processors
- Wrap LED critical section (LED status reads) with:
  - `pthread_mutex_lock()`
  - `pthread_mutex_unlock()`
- Also need Mutex init/destroy:
  - `pthread_mutex_init()`
  - `pthread_mutex_destroy()`

```c
pthread_mutex_t lock;

/* Initialize Mutex */
err = pthread_mutex_init(&lock, NULL);

/* Create 2 Threads */
i=0;
while(i < 1)
{
    err = pthread_create(&tid[i], NULL, &nios_buttons_get, &nios_num[i]);
    i++;
}

/* Critical Section */
pthread_mutex_lock(&lock);
/* Critical Section */
pthread_mutex_unlock(&lock);

/* Wait for threads to complete */
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);

/* Destroy/remove lock */
pthread_mutex_destroy(&lock);
```
Post Lab 3 Additional Topic

Altera SoC Embedded Design Suite
Altera Software Development Tools

- Eclipse
  - For ARM Cortex-A9 (ARM Development Studio 5 – Altera Edition)
  - For NIOS
- Pre-loader/U-Boot Generator
- Device Tree Generator
- Bare-metal Libraries
- Compilers
  - GCC (for ARM and NIOS)
  - ARMCC (for ARM with license)
- Linux Specific
  - Kernel Sources
System Development Flow

FPGA Design Flow

- **Design**
  - Quartus II design software
  - Qsys system integration tool
  - Standard RTL flow
  - Altera and partner IP

- **Simulate**
  - ModelSim, VCS, NCSim, etc.
  - AMBA-AXI and Avalon bus functional models (BFMs)

- **Debug**
  - SignalTap™ II logic analyzer
  - System Console

- **Release**
  - Quartus II Programmer
  - In-system Update

Software Design Flow

- **Design**
  - Eclipse
  - GNU toolchain
  - OS/BSP: Linux, VxWorks
  - Hardware Libraries
  - Design Examples

- **Simulate**
  - ModelSim, VCS, NCSim, etc.
  - AMBA-AXI and Avalon bus functional models (BFMs)

- **Debug**
  - SignalTap™ II logic analyzer
  - System Console

- **Release**
  - Flash Programmer
Inside the Golden System Reference Design

- Complete system example design with Linux software support
- Target Boards:
  - Altera SoC Development Kits
  - Arrow SoC Development Kits
  - Macnica SoC Development Kits
- Hardware Design:
  - Simple custom logic design in FPGA
  - All source code and Quartus II / Qsys design files for reference
- Software Design:
  - Includes Linux Kernel and Application Source code
  - Includes all compiled binaries
References
Altera References

System Design Tutorials:
- http://www.alterawiki.com/wiki/Designing_with_AXI_for_Altera_SoC_ARM_Devices_Workshop_Lab_-_Creating_Your_AXI3_Component
- Designing_with_AXI_for_Altera_SoC_ARM_Devices_Workshop_Lab
- Simple_HPS_to_FPGA_Comunication_for_Altera_SoC_ARM_Devices_Workshop

Multiprocessor NIOS-only Tutorial:

Quartus Handbook:

Qsys:
- System Design with Qsys (PDF) section in the Handbook
- Qsys Tutorial: Step-by-step procedures and design example files to create and verify a system in Qsys
- Qsys 2-day instructor-led class: System Integration with Qsys
- Qsys webcasts and demonstration videos

SoC Embedded Design Suite User Guide:

© 2015 Altera Corporation—Public
Related Articles

- **Performance Analysis of Inter-Processor Communication Methods**

- **Communicating Efficiently between QorIQ Cores in Medical Applications**

- **Linux Inter-Process Communication**
  - [http://www.tldp.org/LDP/TLK/ipc/ipc.html](http://www.tldp.org/LDP/TLK/ipc/ipc.html)

- **Linux locking mechanisms (from ARM)**

- **OpenMCAPi**
  - [https://bitbucket.org/hollisb/openmcapi/wiki/Home](https://bitbucket.org/hollisb/openmcapi/wiki/Home)

- **Mutex Examples**
  - [http://www.thegeekstuff.com/2012/05/c-mutex-examples/](http://www.thegeekstuff.com/2012/05/c-mutex-examples/)
Thank You

- Full Tutorial Resources Online

- Includes:
  - Source code
  - Hardware source
  - Hardware Quartus Projects
  - Software Eclipse Projects