Lab 5: Multi-Core Synchronization Device Design

National Chiao Tung University
Chun-Jen Tsai
5/21/2018

Lab Description

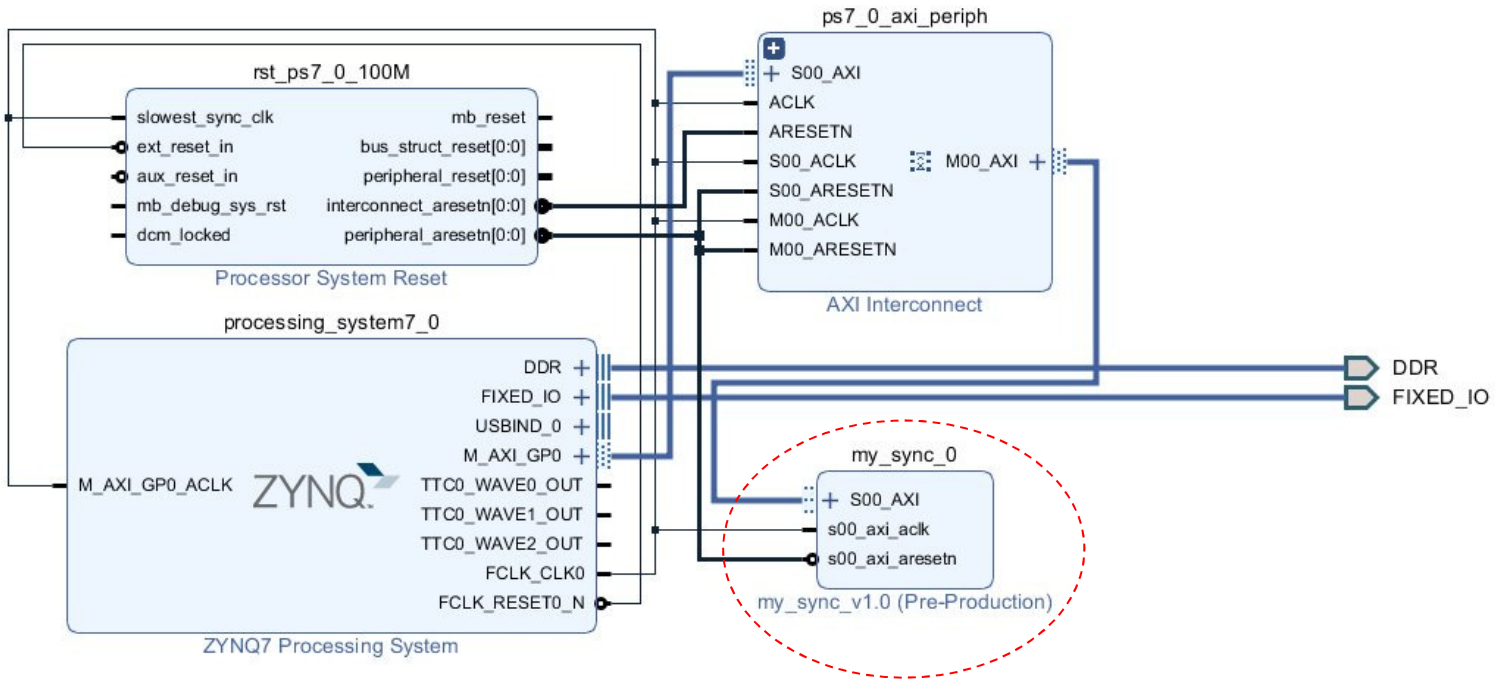
- In this lab, you will design a hardware mutex device for the synchronizations between two Cortex A9's
  - A set of asymmetric multi-processor (AMP) applications will be provided to you as an example
  - You must design a HW IP that contains a mutex device to synchronize the operations of the two processor cores
- Demo to your TA on 6/6

Zynq-7000 Boot Process

- At power-up, each ARM core in the Zynq SoC executes a piece of boot code hardwired inside the chip
- CPU 0 boot code:
  - Configures the system
  - Loads the boot image (FSBL or the user code) from the boot device (platform flash, SD card, or JTAG) into the memory
  - Branches to the boot image for execution
- CPU 1 boot code:
  - Executes the ARM wait-for-event (WFE) instruction
  - If a set-event (SEV) instruction is executed by CPU 0, CPU 1 will jump to the address stored in `$FFFFFFF0`

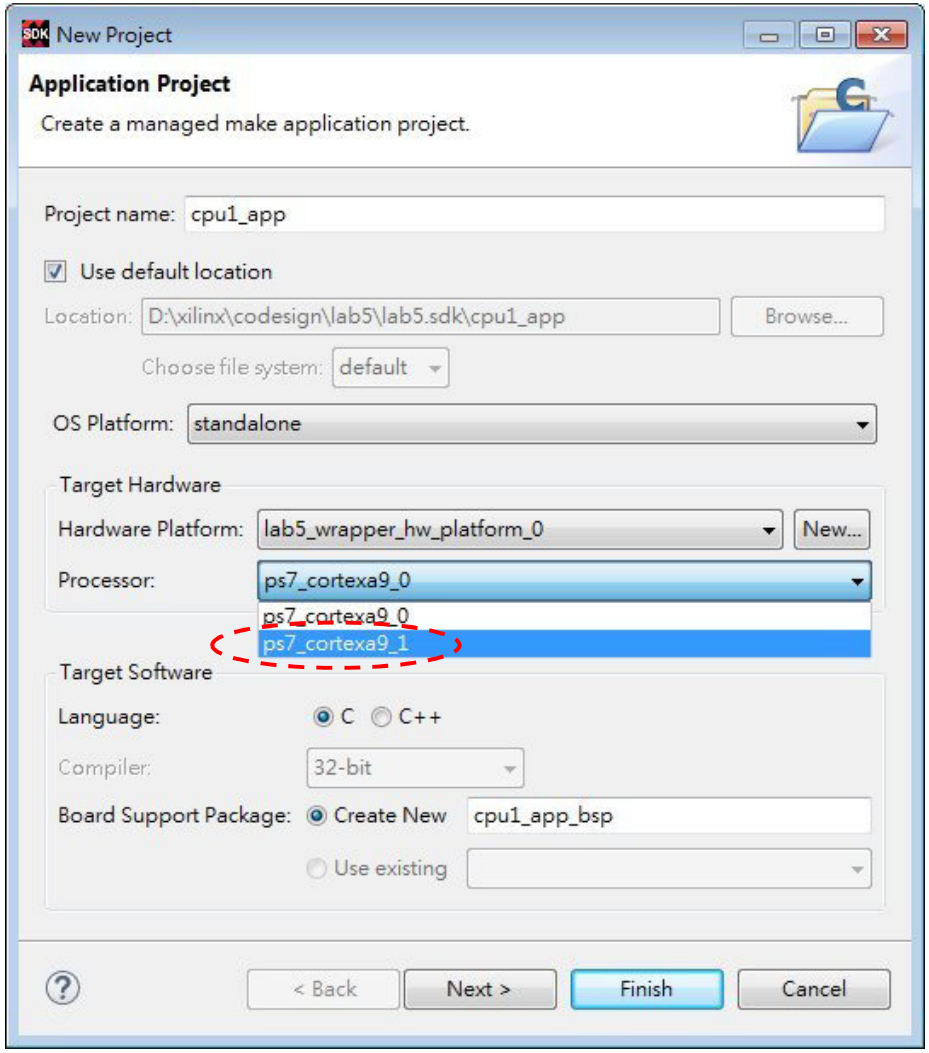
Creation of the HW Platform for Lab5

- For Lab5, you should first create a HW platform with a AXI4-Lite slave user IP, `my_sync`, with four registers



Utilization of Both ARM Cores in Zynq

- You must create two application projects, `cpu0_app` & `cpu1_app`
  - You can specify which CPU to use for each application
- To run both apps on both cores concurrently, you must modify the linker scripts and the BSP build flags



Linker Scripts for CPU 0 and CPU 1

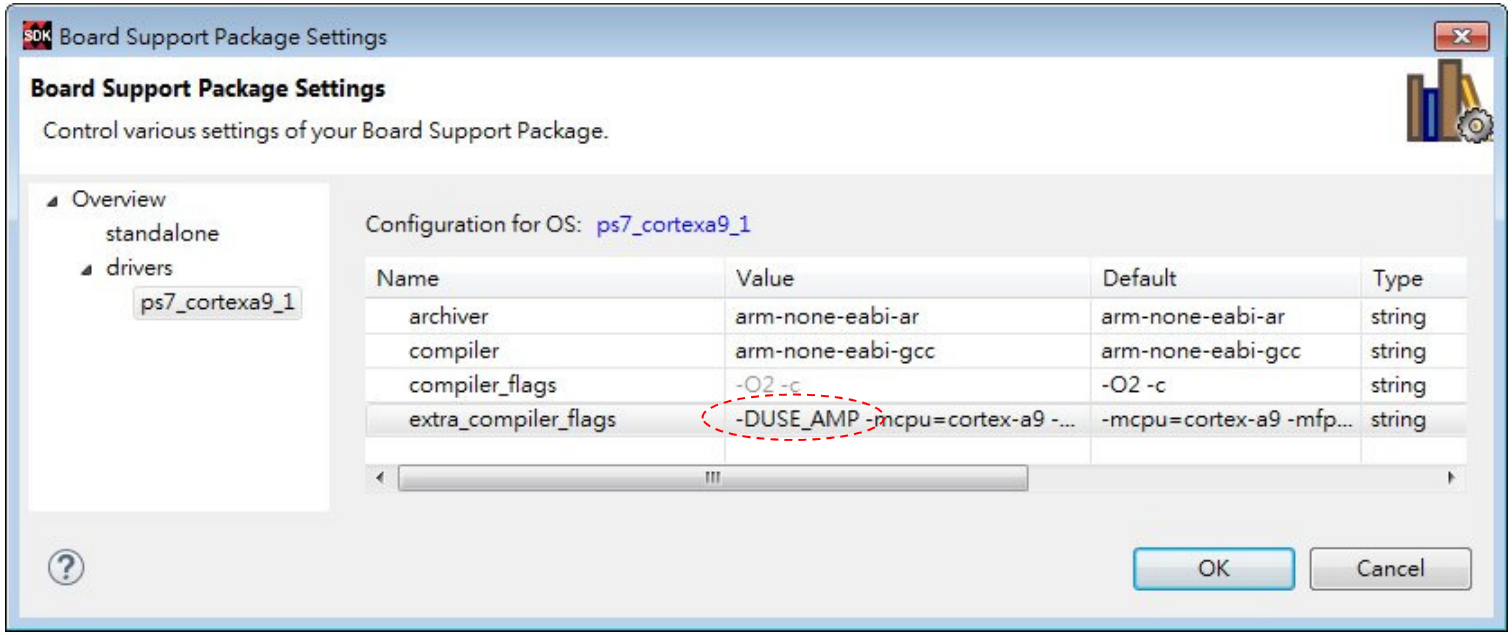
- The linker scripts for cpu0 and cpu1 applications must be modified so that their runtime code, stack, and heap memory will not collide into each others:

CPU 0 lscript.ld	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x00100000	0x10000000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

CPU 1 lscript.ld	Base Address	Size
ps7_ddr_0_S_AXI_BASEADDR	0x10100000	0x10000000
ps7_qspi_linear_0_S_AXI_BASEADDR	0xFC000000	0x1000000
ps7_ram_0_S_AXI_BASEADDR	0x00000000	0x00030000
ps7_ram_1_S_AXI_BASEADDR	0xFFFF0000	0x0000FE00

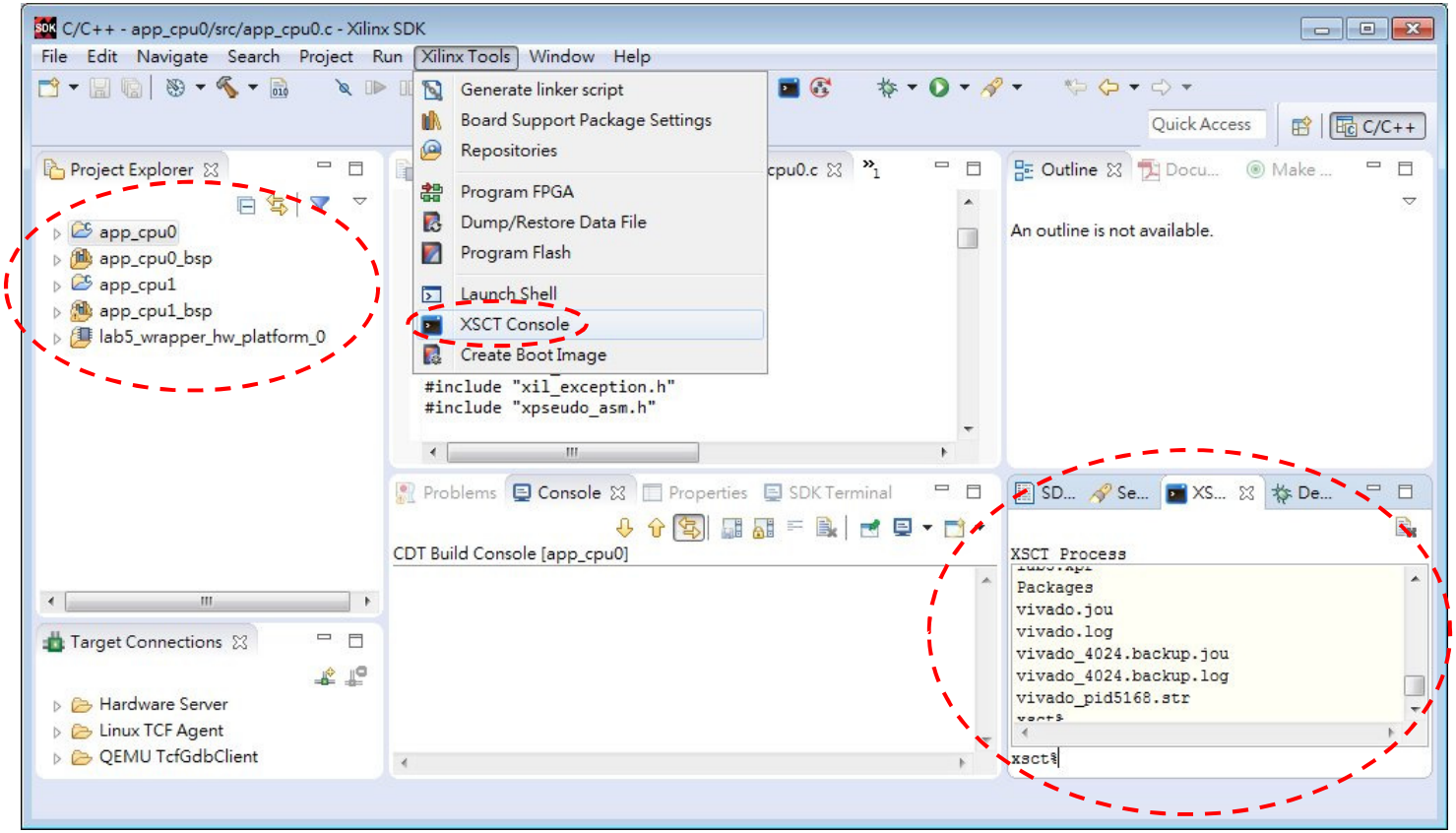
Add CPU 1 BSP Compilation Flag

- You should add the flag `-DUSE_AMP=1` to the BSP settings of CPU 1 application:
  - Assume that CPU 0 will run `cpu0_app` first
  - This way, CPU 1 will not initialize the Zynq PS7 twice!



Running Applications on Both CPUs

- It is possible to run the two programs from the GUI using “Run As”, but you have more control if you use the Xilinx Shell Command Tool (XSCT)



Control Two Cores Using XSCT

- XSCT gives you better control of the two cores:

```
XSCT % pwd
XSCT % cd <YOUR_PATH>/lab5
XSCT % connect
attempting to launch hw_server
                . . .
XSCT % fpga lab5.runs/impl_1/lab5_wrapper.bit
                . . .
100%     3MB   1.6MB/s  00:02
XSCT %
XSCT % source lab5.sdk/lab5_wrapper_hw_platform_0/ps7_init.tcl
XSCT % targets  → Check available targets
 1  APU
     2  ARM Cortex-A9 MPCore #0 (Running)
     3  ARM Cortex-A9 MPCore #1 (Running)
 4  xc7z020
XSCT % target 2  → Set command target core
XSCT % ps7_init
XSCT % ps7_post_config  } Initialize Zynq PS7 and the DDR memory
XSCT %
```

For more command explanations, see Ref [1].

Execute Programs on Both Cores

- If you have two applications `app_cpu1` and `app_cpu0` to be executed on each core, use the commands:

```
XSCT% target 2
XSCT% dow lab5.sdk/app_cpu0/Debug/app_cpu0.elf
XSCT% con

XSCT% target 3
XSCT% dow lab5.sdk/app_cpu1/Debug/app_cpu1.elf
XSCT% con
```

Compiling the application in Debug mode makes it easier to see the shared data corruption!

- To see the "printf()" outputs of the programs:
  - For XSCT execution, you must run a standalone console program such as TeraTerm to see the output
  - For GUI execution, only one of the STDIO's can be set to a COM port, the other application should not set the STDIO
  - The outputs may get corrupted without proper protection

Reset Processor Cores

- If you want to reset the processor cores for another round of execution, you can use the following XSCT commands:

```
XSCT% target 2
XSCT% rst -processor
XSCT% target 3
XSCT% rst -processor
XSCT%
```

After the reset commands, you can reload the applications into the memory for another execution

Mutex

- A mutex is a variable to indicate the two states “locked” and “unlocked” of a shared resource:

```
int mutex;

lock_mutex(mutex);
Critical Section code that uses the shared resource.
unlock_mutex(mutex);
```

Execution blocked here if the mutex is locked by other threads.

- A mutex variable can be implemented using either software approach or hardware approach

Mutex Implementation

- Software mutex implementation techniques
  - Software algorithms (e.g. the Peterson's algorithm)
    - Drawback: time-consuming
  - Atomic test-and-set or exclusive load-store instructions
    - Drawback: either lock bus cycles or not suitable for HW-SW synchronization
- Hardware mutex approach
  - A HW mutex is a special memory device that "conditionally" accepts write requests
  - Suitable for synchronization between HW-SW
  - Low-cost
  - Drawback: limited number of mutexes

Software Mutex Algorithm

- Peterson's algorithm† guarantees exclusive accesses to a shared resource among n threads (running on n cores) without special assembly instructions
- A two-thread version is as follows:

CPU 0

```
/* trying protocol for T1 */
Q1 = true; /* request to enter */
TURN = 1;  /* who's turn to wait */
wait until not Q2 or TURN == 2;
Critical Section;
/* exit protocol for T1 */
Q1 = false;
```

CPU 1

```
/* trying protocol for T2 */
Q2 = true; /* request to enter   */
TURN = 2;  /* who's turn to wait */
wait until not Q1 or TURN == 1;
Critical Section;
/* exit protocol for T2 */
Q2 = false;
```

† G. L. Peterson, "Myth about the Mutual Exclusion Problem," *Information Processing Letters*, **12**, no 3, June 30, 1981.

Test-and-Set Atomic Operations

- For synchronization, a thread must execute the following code before entering a critical section:

```
int mutex; /* '0' means unlocked, '1' means locked */

while (test_and_set(mutex) == 1) /* busy waiting */
Critical Section code that uses the shared resource.
mutex = 0;
```

- A 'SWAP' instruction can be used to implement test-and-set, but it increases interrupt latency and reduces multi-core performance

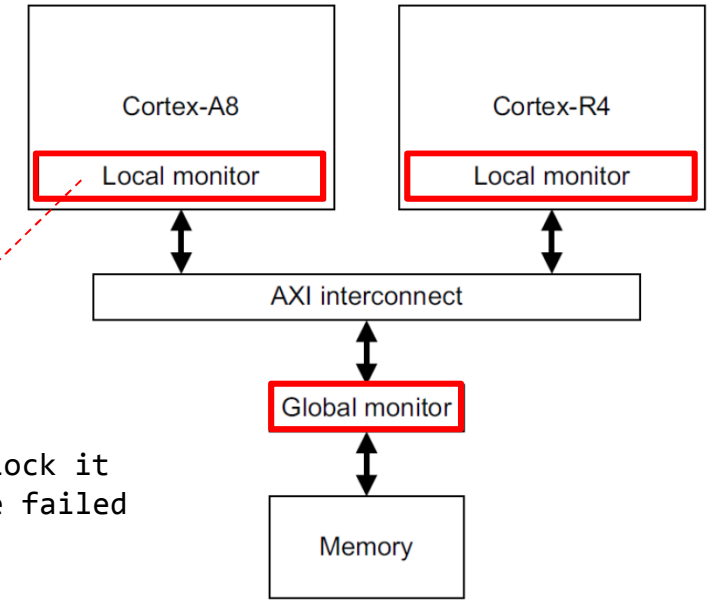
```
int test_and_set(int mutex)
{
    temp = mutex;
    mutex = 1;
    return temp;
}
```

The first two lines cannot be interrupted during execution!

Exclusive Load/Store Instructions

- New multicore processors have complex instructions (e.g., LDREX/STREX, memory barriers) designed for mutex implementation

```
locked EQU 1
unlocked EQU 0

; lock_mutex() function
lock_mutex PROC
    LDR     r1, =locked
1   LDREX   r2, [r0]      ; r0 points to the mutex
    CMP     r2, r1        ; Test if mutex is locked
    BEQ     %f2           ; If locked - wait
    STREXNE r2, r1, [r0] ; Not locked, attempt to lock it
    CMPNE   r2, #1        ; Check if Store-Exclusive failed
    BEQ     %b1           ; Failed - retry from 1
                          ; Lock acquired
    DMB                   ; Set memory barrier check point
    BX lr                 ; Return to the caller function

2 ; Take appropriate action while waiting for mutex to become unlocked
    WAIT_FOR_UPDATE       ; CPU may enter low-power state here
    B %b1                 ; Retry from 1
ENDP
```

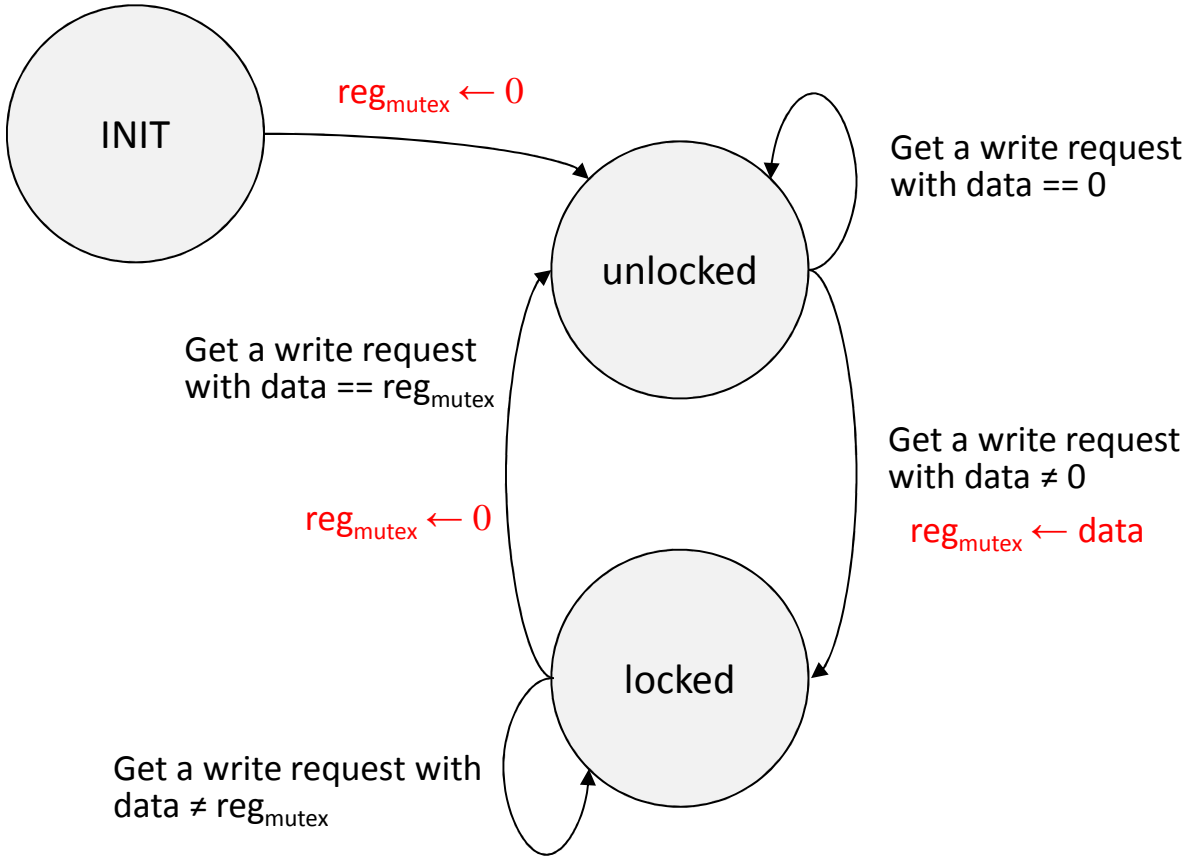


Hardware Mutex Design

- A hardware mutex is a register controlled by an arbiter
  - Each mutex register stores an ID of the thread (HW or SW) that currently owns the mutex
  - It has two types of “write” operations, lock and unlock
  - Only one of the simultaneous lock requests will be granted (by any arbitration rule)
- Writing an ID to a mutex issues a lock operation
  - If the write operation is successful, the mutex is locked
  - Otherwise, someone else has locked the mutex before you
  - If a mutex is in locked state, any write operations with a different ID would simply be ignored
- Writing the owner ID to the mutex (usually by the current owner) will reset the mutex to zero and releases the lock

Controller of the Mutex

- The controller of the HW mutex:



SW Interface of Mutex Lock/Unlock

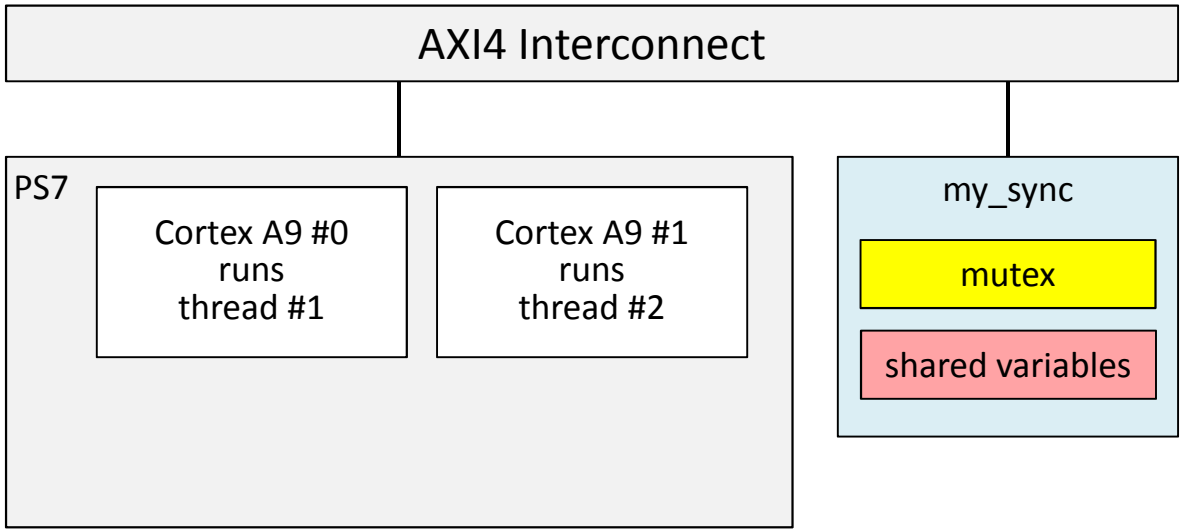
- The functions for lock/unlock a mutex are as follows:

```
volatile int *hardware_mutex =  (int *) XPAR_MY_SYNC_0_S00_AXI_BASEADDR;

void lock_mutex(int thread_id)
{
    do {
        *hardware_mutex = thread_id;
    } while (*hardware_mutex != thread_id)  /* busy waiting */;
}

void unlock_mutex(int thread_id)
{
  *hardware_mutex = thread_id; /* set mutex to 0 */
}
```

This operation is similar to a technique called “write 1 to reset”!
(no “read” operation is involved in order to clear a single bit)

Lab 5 System Overview (1/2)

- In Lab 5, you must design a mutex device IP, called `my_sync`, to synchronize the two ARM cores
  - The IP contains a hardware mutex register, which allows you to protect a shared resource



Lab 5 System Overview (2/2)

- Both software threads for CPU 0 & CPU 1 will increase the shared counter cooperatively and independently from 0 to 1,000,000
- If the shared counter is not protected by a mutex, the total number of counts increased by both threads may not be 1,000,000 exactly

```
int shared_counter;

void main_task()
{
    int local_counter = 0;

    while (shared_counter < 1,000,000)
    {
        shared_counter++;
        local_counter++;
    }
}
```

The Specification of the `my_sync` IP

- The IP `my_sync` you design shall have four registers
  - `hardware_mutex (slv_reg0)`:
    the hardware mutex register used to protect a shared resource
  - `shared_counter (slv_reg1)`:
    a shared variable for the applications
  - `start_trigger (slv_reg2)`:
    a register used to signal the execution of the app main loop
  - `cpu1_local_ctr (slv_reg3)`:
    a local variable used only by CPU1 during execution;
    CPU 0 will read its value at the end to check data corruption
- On E3, the two application programs `cpu0_app.c` and `cpu1_app.c` will be provided.

Overview of the Sample Applications

- CPU 0 main task

```
void cpu0_main()
{
  int thread_id;
  int *local_counter = &cpu0_local_ctr;

  srand((int) &local_counter);
  thread_id = rand();

          . . .

  while (*start_trigger == 0) /* waiting */

  while (! done) {
    lock_mutex(thread_id);

    done = (*shared_counter >= LIMIT);
    if (! done) {
      (*shared_counter)++;
      (*local_counter)++;
    }
    unlock_mutex(thread_id);
  }

  return 0;
}
```

CPU 1 main task

```
int cpu1_main()
{
  int thread_id;
  int *local_counter = &cpu1_local_ctr;

  srand((int) &local_counter);
  thread_id = rand();

       . . .

 *start_trigger = 1;

  while (! done) {
    lock_mutex(thread_id);

    done = (*shared_counter >= LIMIT);
    if (! done) {
      (*shared_counter)++;
      (*local_counter)++;
    }
    unlock_mutex(thread_id);
  }

  return 0;
}
```

Sample Output

- Without a proper mutex device, the shared counter will be corrupted:

```
CPU0 & CPU1 are about to tick the shared counter at [0x43c00004].
CPU0 waiting for CPU1 to start ...
At the end, the shared counter = 1000001
Local CPU0 counter = 5000003
Local CPU1 counter = 4999999
CPU0 counter + CPU1 counter != Shared counter, the counter is corrupted.
```

- In lab5, you must implement the hardware IP `my_sync` properly so that the output is correct.

References

- Xilinx, *Software Command-Line Tool (XSCT) Reference Guide*, UG1208, June 8, 2016.
- J. McDougall, *Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors*, XAPP 1079, Jan. 24, 2014.