IL2212 Embedded Software
Lab 1 Report
Session: Friday Afternoon

Yuefeng Wu
Embedded Systems
EIT Digital Master School
Personal Number: 921116-6039
Email: yuefeng@kth.se

Renzhi Xing
Embedded Systems
KTH, Royal Institute of Technology
Personal Number: 931011-2025
Email: renzhi@kth.se

***Abstract*—This report is a laboratory record of Lab 1 in IL2212 Embedded Software VT2016. In this report, we presented the features and limitations of the multiprocessor and a diagram illustrating the architecture of the multiprocessor of given system with all peripherals and their interconnections. Additionally, we learned mechanisms for both shared memory and message passing communication through designing and describing the demo program. Finally, we analyzed optimization settings in the compiling and downloading script.**

I. Introduction

This laboratory is named with *Communication in a Bare-Metal Multiprocessor*, which is aimed to guide us in getting a basic handle of communication mechanisms in multiprocessor systems. We implemented the system for laboratory on five NIOS2 soft cores. In the demo program of this laboratory, we used *shared memory, mutexes and FIFO core* to build up communications among five cores. The demo is programed in language C with NIOS2 APIs.

II. Features and Limitations of Multiprocessors

In this section, we will give a brief introduction to features of multiprocessors and a basic analysis of their limitations.
According to some on-line dictionaries, a multiprocessor system is a computer system having two or more processing units (multiple processors) each sharing main memory and peripherals, in order to simultaneously process programs[1]. A multiprocessing system can contain several identical or heterogeneous processors. Identical processors are of the same type, which means the processors can be used interchangeably. In the contrast, the processors of differnt types that cannot be used interchangeably are heterogeneous[2].

The limitations of multiprocessors come from mainly two points.

- **Complexity**
  In multiprocessing systems, a complex mechanism is essential to coordinate processors' utilization of hardware resource and miscellaneous auxiliaries. Additionally, regarding to embedded systems, more complicated dispatchers and schedulers are needed to assign tasks to processors, which is required to be efficient and accurate.
- **Application**
  Not all the applications are capable of multiprocessing, especially the old ones. When this single-processing programs are executed on multiprocessing systems, a waste of hardware resource is inevitable. How to increase the efficiency is still a challenge for multiprocessors.

III. Hardware Architecture

We draw a simplified graph of hardware interconnections in which the purple lines mean "channels for data and control signals". Additionally, we also give a more complex interconnection graph attached at the end of this report which is based on the *nios2_mpsoc.qsys* file and indicates ins and outs. And we will give some brief description of miscellaneous parts of the laboratory system in the subsections.

However, the actual interconnections is more complex. Since the laboratory system is based on Altera's Qsys system, the *Network on Chip, NoC* architecture is also deployed on the system. According to the basic topology of an NoC system. Each endpoint interface in the network, master or slave, is connected to a network interface (NI) component. The network interface captures the transaction or response using the transaction layer protocol, and delivers it to the network as a packet of the appropriate format. The packet network delivers packets to the appropriate packet endpoints, which then pass them to other network interfaces. The network interfaces then terminate the packet and deliver the command or response to the master or slave using the transaction layer protocol[3]. This kind of transmission method is similar with TCP/IP protocol.

A. CPUs

There are five soft cores in the laboratory system, named from cpu_0 to cpu_4. Each core has its own timer(cpu_0 owns two timers), jtag debug module and on-chip RAM.(Except cpu_0 doesn't contain private on-chip memory) Additionally,

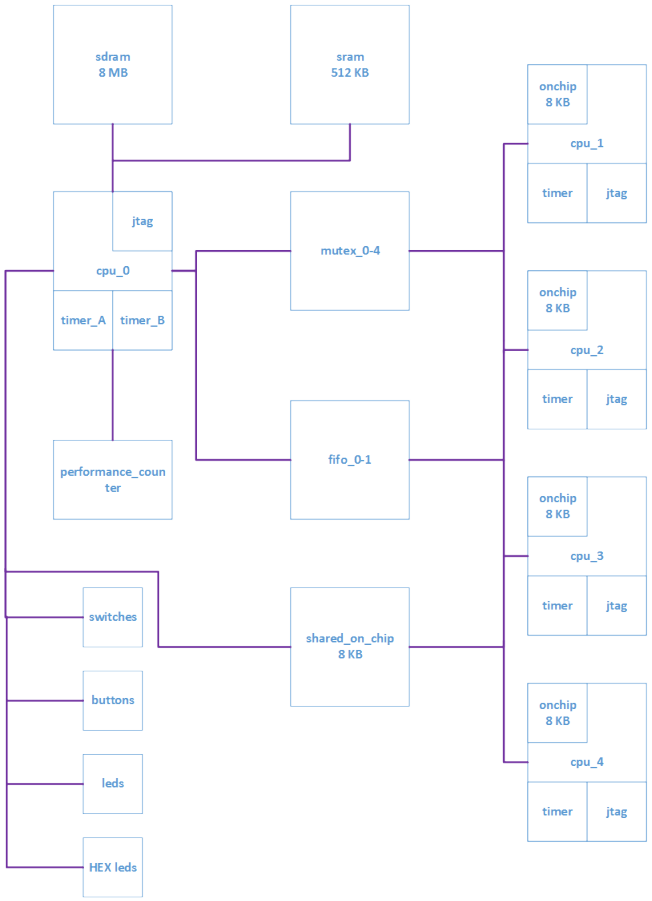

Fig. 1. Simplified interconnection graph.

cpu_0 has its own performance counter to monitor execution time of code sections.

B. Memory

The system has **SDRAM, SRAM and on-chip memory**. The size of SDRAM is **8 MByte**, since the data address is 0x01000000 from to 0x017FFFFF. The size of SRAM is **512 KByte** and the address is 0x00080000 from to 0x000FFFFF. They are both exclusively connected to cpu_0, which means they can only be accessed by cpu_0. Each core except cpu_0 has its own on-chip memory whose size is **8 KByte**. Additionally, there is a shared on-chip memory whose size is **8 KByte** that can be accessed by all the five cores.

C. Peripherals

Eighteen switches, **four** buttons, **eight** LED numeric displays, **nineteen** red LEDs and **eight** green LEDs are connected to cpu_0. They can only be accessed by cpu_0, which can be proved by the differentiation of *system.h* files.

D. Communication between cores

There are three components to implement communication mechanisms: **shared on-chip memory, two FIFO memory cores, four mutexes**. These components are connected to the data interfaces of all the cores. Theses connections can be verified by the *system.h* files.

IV. Communication Mechanisms

As mentioned above, the system provides three hardware components—**shared on-chip memory, FIFO memory cores and mutexes** for inter-core communications.

A. Read and write shared memory

This mechanism is categorized by two method—**safe and unsafe method**.

1) Unsafe method: This is the simplest communication method between cores. There is a specific space in shared memory which can be accessed by address. Once one core writes data to the space and others can read the data through the address. This method comes with a huge potential risk. If the space is not ready to read(unwritten or uninitialized), it may cause fatal errors.

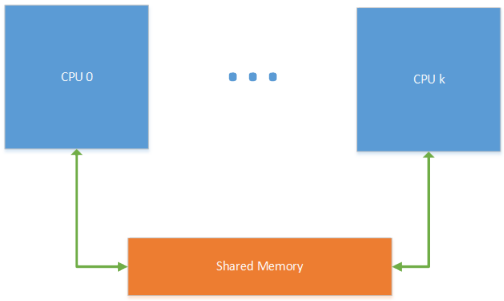

Fig. 2. Shared Memory in Unsafe Method

2) Safe method: To avoid the reading issues, the safe method adds an extra 8-bit space as a **flag**. When a data space is initialized or written, the corresponding flag is set to 1, which means "ready to be read". And after reading the data, the reader is responsible to set the flag to 0, which means "need to be written and can't be read". This method is a trade-off of space and security.

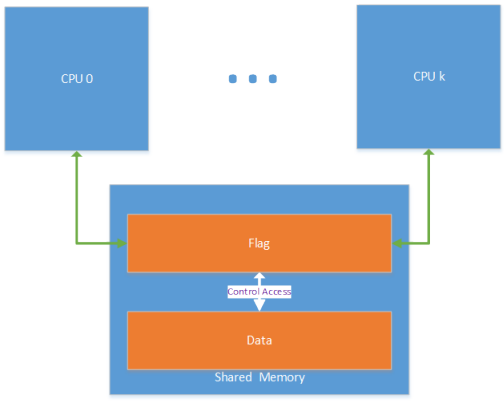

Fig. 3. Shared Memory in Safe Method

B. FIFO memory core

FIFO, the abbreviation for *"first in, first out"*, is a common data structure. The depth of FIFO memory cores in this system is 16, which means that the maximum 16 piece of 8-bit data can be buffered in every core. When the buffer is full,the cores can detect it and stop writing. Analogously, when the buffer is empty, the cores can also stop reading.

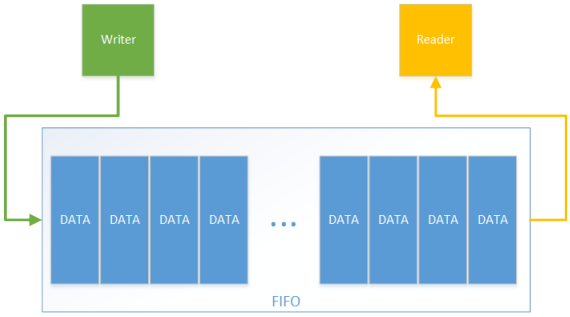

Fig. 4. FIFO

C. Mutex

Mutex exclusion is a mechanism to avoid two or more threads or tasks running into the same critical section, such as accessing the same shared resource or responding to a timer[4]. In our perception, the mutexes in this system are similar with semaphores. Not only in the aspect of function, but also locking mutex provides a blocking function. When a core or thread has locked a mutex, other cores or threads who try to lock the same mutex will be blocked until the mutex is finally unlocked. Additionally, we can also make locking a mutex non-blocking with a trylock function.

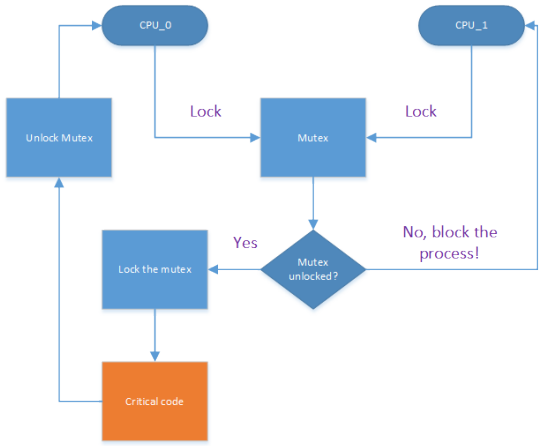

Fig. 5. Mutex

V. Memory Footprint

Memory footprint refers to the amount of main memory that a program uses or references while running. In the script of laboratory, *nios2-elf-size* is included which is a command to show the code size of programs. The output of this command will contain five columns—**text, data, bss, dec, hex**.

- **text:** the size of general code.
- **data:** the size of variables.
- **bss:** the size of uninitialized variables[5].
- **dec:** the total size of the program, displayed in decimal.
- **hex:** the total size of the program, displayed in hexadecimal.

Finally, we put the result of our program in the terminal and the size unit of byte. According to the hardware description

TABLE I
"Memory Footprint of Programs"

CPU	Memory	Text	Data	BSS	DEC	HEX
cpu_0	sram	10448	580	424	11452	2cbc
cpu_1	onchip	1428	96	20	1544	608
cpu_2	onchip	2704	96	20	2820	b04
cpu_3	onchip	2436	96	16	2548	9f4
cpu_4	onchip	1312	96	16	1424	590

VI. Demo Program

As required, we designed a simple demo program to implement three inter-core mechanisms. In the following part, we will describe the five programs in the five cores.

A. CPU 0

Since cpu_0 is connected with relatively more auxiliaries than other cores. We designed more function in cpu_0.

- **Controlling LEDs, buttons and switches**
  In every loop, the program will scan the statuses of buttons and switches. When a newly turned-on switch or a newly pressed-down button is detected, it will light the corresponding LED(s). Additionally, it will also store the status of switches in a variable for further use.
- **Reading data from shared memory**
  In this part, we implement safe method. First it uses FLAG to detect whether the shared memory is readable. If the program will read the data from VALUE and print it out. FLAG is at the start address of shared memory located in 0x00102000 and VALUE is located at the next address. If the part is unreadable, it will print a error message.
- **Displaying information with LED numeric display**
  There are two groups of LED numeric displays—HEX3_HEX0 and HEX7_HEX4. We use HEX3_HEX0 to display the number indicated by switches and HEX7_HEX4 for data read from shared memory.
- **Monitoring execution time**
  We use *PERF_BEGIN* and *PERF_END* to embrace the code section whose execution time we want to measure. And use *perf_print_formatted_report* to print out information we need.

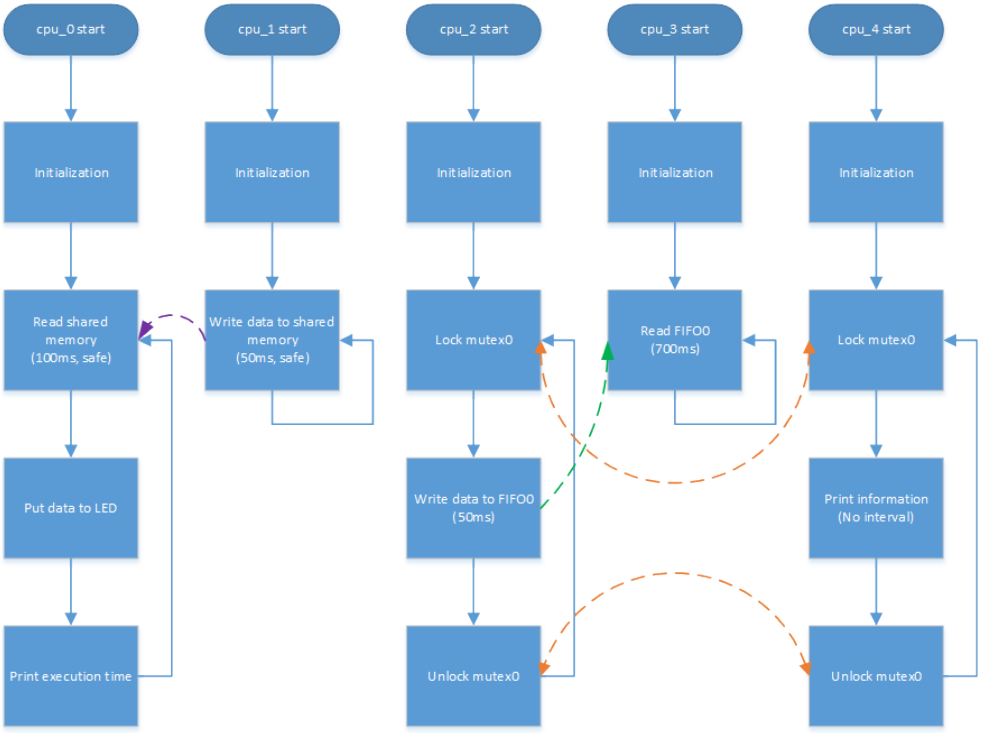

Fig. 6. Program procedure graph for demo program

B. CPU 1

The cpu_1 writes to shared memory addressed at VALUE every 50ms if the content in FLAG is 0. This procure builds up the inter-core connection with cpu_0.

C. CPU 2

After initialization, cpu_2 will lock mutex_0 and then write data to FIFO_0 if it is not full every 50ms. And finally it will unlock mutex_0.

D. CPU 3

Every 700ms, cpu_3 will fetch data from FIFO_0. If it is empty, it will give a error message.

E. CPU 4

At the beginning, cpu_4 will try to lock mutex_0. if it fails, it will be blocked. And after this, it will print a message without delay and unlock mutex_0. The expected result is that the message is printed every 50ms due to the mutex-locking operations of cpu_2.

Since the time is limited, we only designed a simple demo program to demonstrate the three communication mechanisms. We will add more advanced functions in this program.

According to the lab manual, we drew a program procedure graph of this demo. In the figure, the purple arrow is for shared memory; the orange arrows are for mutexes; the green arrow is for FIFO.

VII. Optimizations for Space Usage

Due to the limited resource in DE2 board, some optimizations are turned on to make the resulting program consume less space.

A. Os optimization level

Os level is enabled to optimize space usage (code and data) of resulting program. -Os enables all -O2 optimizations that do not typically increase code size. It also performs further optimizations designed to reduce code size[6].

B. Small C library

Small C library is also enabled with . This reduces code and data footprint at the expense of reduced functionality. Several newlib features are removed such as floating-point support in printf(), stdin input routines, and buffered I/O.

C. Reduced device drivers

Certain drivers are compiled with reduced functionality to reduce code footprint.

D. Lightweight device driver API

Lightweight device driver API is enabled. This reduces code and data footprint by removing the HAL layer that maps device names (e.g. /dev/uart0) to file descriptors. Instead, driver routines are called directly.

E. C++ support

C++ support is disabled to reduce code footprint.

F. Clean exit

Code footprint can be reduced by disabling clean exit.

VIII. Summary

Through this laboratory, we reviewed the NIOS II labs in Embedded Systems course and got a basic of multiprocessor system development. Finishing this lab is a good beginning of the following labs.

References

[1] Defination of multiprocessor. [Online]. Available: http://www.yourdictionary.com/multiprocessor

[2] J. W. Liu, "Real-time systems," 2000.

[3] A. Corporation, "Applying the benefits of network on a chip architecture to fpga system design."

[4] P.-J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with readers and writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667–668, 1971.

[5] A. One, "Smashing the stack for fun and profit," *Phrack magazine*, vol. 7, no. 49, pp. 14–16, 1996.

[6] Optimize options - using the gnu compiler collection (gcc). [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html