Multiprocessor Communications

Shruti Kuber, Hansell Baran – Lab Session February 7, Afternoon 13-17

***Abstract*—Inter-process communication is crucial in a multiprocessor environment, especially when there are shared resources. We explore a five-processor hardware architecture with a shared on-chip memory implemented on an Altera's Field Programmable Gate Array. We also analyze the platform and tools used to design and create this multiprocessor system. Finally, we develop an application/software that demonstrates the hardware's inter-process communication capabilities and propose possible optimizations for the platform, tools and hardware architecture.**

***Keywords—Multiprocessors, IL2212, Embedded Software, Inter-process communication (IPC), Altera, DE2, Quartus, optimizations, FPGA, mutex, message passing*.**

I. Introduction

Inter-process communication (IPC) "is a method of communication by exchanging data/information among multiple threads or multiple process" [1]. In a system with multiple processors (CPUs) and shared resources, access to shared resources must be controlled to avoid data corruption due to race conditions (which happen when two or more CPUs try to access and use a specific shared resource at the same time). Mutual exclusion, which allows only one process to accesses the shared resource at any given time, is used to prevent race conditions and it is enforced through the use of locks (commonly spinlocks), which typically are atomic operations.

II. Laboratory Environment

As part of KTH's Embedded Software IL2212 course Laboratory, we are provided with an already designed and built hardware architecture based on the following platform and tools: *Altera's DE2 Development Board*, *Quartus II*, *Nios II "Soft" Processor* and *Nios II Software Build Tools / IDE*

The course staff provides qsys, sopcinfo (System On a Programmable Chip system information/description), jdi (JTAG debugging information), sof (SRAM Object File), example C code files (`.c`) as well as automated linux shell scripts (`.sh`) to generate all the necessary BSP (Board Support Packages) and ELF (Executable and Linkable Format) files.

A. Code Optimizations

The script *run_hello_mpsoc.sh*, specifically the section where the BSPs for cpu_1 through cpu_4 are generated has the following flags:

```
--set hal.make.bsp_cflags_debug -g \
--set hal.make.bsp_cflags_optimization -Os \
--set hal.enable_small_c_library 1 \
--set hal.enable_reduced_device_drivers 1 \
--set hal.enable_lightweight_device_driver_api 1 \
--set hal.enable_sopc_sysid_check 1 \
--set hal.max_file_descriptors 4 \
--default_sections_mapping onchip_$i \
--set hal.sys_clk_timer none \
--set hal.timestamp_timer none \
--set hal.enable_exit false \
--set hal.enable_c_plus_plus false \
--set hal.enable_clean_exit false \
--set hal.enable_sim_optimize false
```

It is worth noting that the last 6 lines/settings are NOT present in the command used to generate the BSP for cpu_0. These flags will be briefly described but the results of using them will be seen and discussed further in the results section. Detailed information on all available commands and flags can be found in [6]. Most of the flags present in the shell script provide code size optimization; however, after examining [6] we can make the following observations:

- The `hal.disable_startup_thread_sync` setting could be enabled to reduce code footprint if the program has NO initialized global or static variables and does NOT use `alt_malloc()|free()|calloc()` functions.
- The settings `hal.enable_small_stack` & `hal_exclude_default_exception` could be enabled if the program does NOT make use of exceptions and has an execution stack of less than 384 Bytes.
- Set `stdin`, `stdout` and `stderr` to `/dev/null` ONLY if the program does NOT make use of them (`hal.stderr|stdin|stdout`).
- Further code footprint reduction can be achieved by using UNIX-Style File I/O instead of ANSI C functions, and in case they were necessary, they could be emulated with UNIX-style functions.
- Using the Minimal Character-Mode API would also reduce code footprint (use `alt_printf()`, `alt_putchar()`, `alt_putstr()`, `alt_getchar()`)
- Use the free-standing environment provided by `alt_main()` to eliminate unused device drivers and initialize ONLY the necessary and used ones; thus, reduce the code footprint.
- The stack size `hal.thread_stack_size` can be reduced instead of leaving the default value of "…¾ of the size of the memory region…" [6]

B. Hardware architecture

After analyzing the qsys file provided, we can derive the system diagram shown in figure 1.

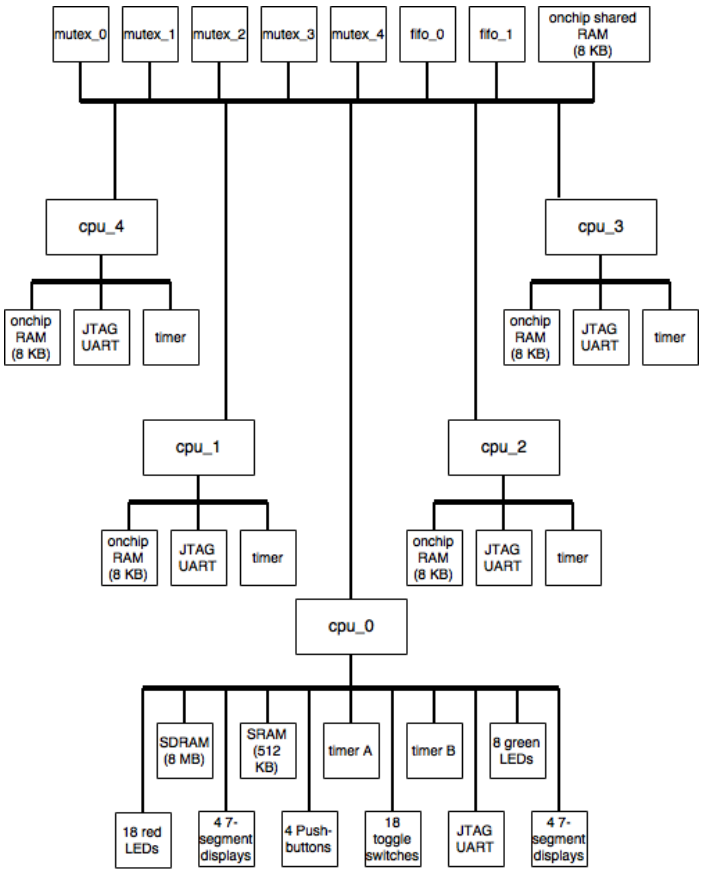

Figure 1. System block diagram for the provided hardware architecture

For simplicity in the diagram, the connection between each processor and the mutex, fifo and shared memory is drawn as a bus; rather, each mutex, fifo and the shared RAM has a direct connection to each of the five CPUs. A more accurate representation of the interconnection network can be found in the altera documentation [7].

The following components are worth describing:

- **5 Nios II/e processors**. They run from a 50 MHz clock common to all components.
- **1 SDRAM**. Has an 8 MB size and is accessible ONLY by cpu_0.
- **1 SRAM**. Has a 512 KB size and is accessible ONLY by cpu_0. Moreover, it is used by cpu_0 as main memory (code, data, stack, etc.).
- **2 FIFO Cores**. Could be used as part of a message passing mechanism.
- **5 MUTEX Cores**. Provided to coordinate access to shared resources. These cores provide "…hardware-based atomic test-and-set operations…" [7]
- **1 On-Chip Shared RAM**. Has a size of 8 KB. Since all CPUs have access to this memory, controlled access is required to ensure data integrity.

Peripherals are considered shared when multiple processors can access them. Shared peripherals can be a very powerful feature of multiprocessor systems, but care must be taken when deciding which system peripherals are shared, and how the different CPUs cooperate regarding their use. Sharing peripherals in multiprocessor systems presents some difficult challenges and is generally considered to lead to inefficient system designs. Altera recommends that you restrict all peripherals (except memory and mutex) to be accessible by only one processor in the system. If other CPUs require use of the peripheral, it is better to use a hardware FIFO, or a mutex-protected message buffer, to communicate with the single CPU connected to that peripheral. That single CPU acts as a server for the other processor clients of that peripheral. When building any system, especially a multiprocessor system, it is advisable only to make connections between processors and peripherals that require direct communication.

The given design reveals that all 5 CPUs share one on-chip memory and 2 FIFO's. To control access to these resources, hardware mutexes were also provided. Other peripherals like LED's, 7 segment displays, buttons, toggle switches, SRAM and SDRAM are connected only to `cpu_0`. Such an implementation increases the design efficiency.

C. *Message passing mechanisms*

One way to safely pass messages between CPUs is using the FIFO and Mutex Cores. With the help of mutexes, a process will try to lock a mutex and ONLY after the mutex is successfully locked and owned, the process will be able to manipulate the shared resource (memory or FIFO).

Altera provides other mechanisms to pass messages between processors such as the Mailbox Core which "…contains mutexes to ensure that only one processor modifies the mailbox contents at a time." [7], however, it will not be analyzed any further since it is not implemented in the given architecture.

It is worth mentioning that using different hardware components, optimizations could be made to lower the FPGA Logic Elements (LEs) utilization, reduce latency / complexity and gain execution speed. The use of Avalon Memory-Mapped Pipeline Bridges could not only reduce system interconnection complexity and latency but could also reduce the FPGA's total power consumption. Also, depending on the application, it could be possible to eliminate all mutexes but one.

Similarly, the use of mailboxes instead of mutexes alone could prove to be a better implementation; again, this will depend on the application.

III. Multiprocessor Application

A multiprocessor software application was developed to demonstrate inter-process communication using the provided hardware architecture.

The application is a ticket booking system where there are 8 vacant seats and 4 customers. Each customer enters its user id and can book 2 seats. After booking the 2 seats, the customer logs out and the server displays which customer booked which seat. The server is `cpu_0`, which displays the booked seats and user id on the 7-seg display and LEDs respectively. The seats

are addresses in the shared memory. Logging out is updating a flag in the shared memory and logging in is writing a value to the FIFO. Mutexes protect access to the shared memory and the FIFO.

Finally, the hardware architecture provides a performance counter component, which was used to measure the execution time when `cpu_0` initializes the `shared_onchip` memory. In addition, we measure the total execution time of the application (after all seats have been taken). Figure 2 shows a diagram of the application.

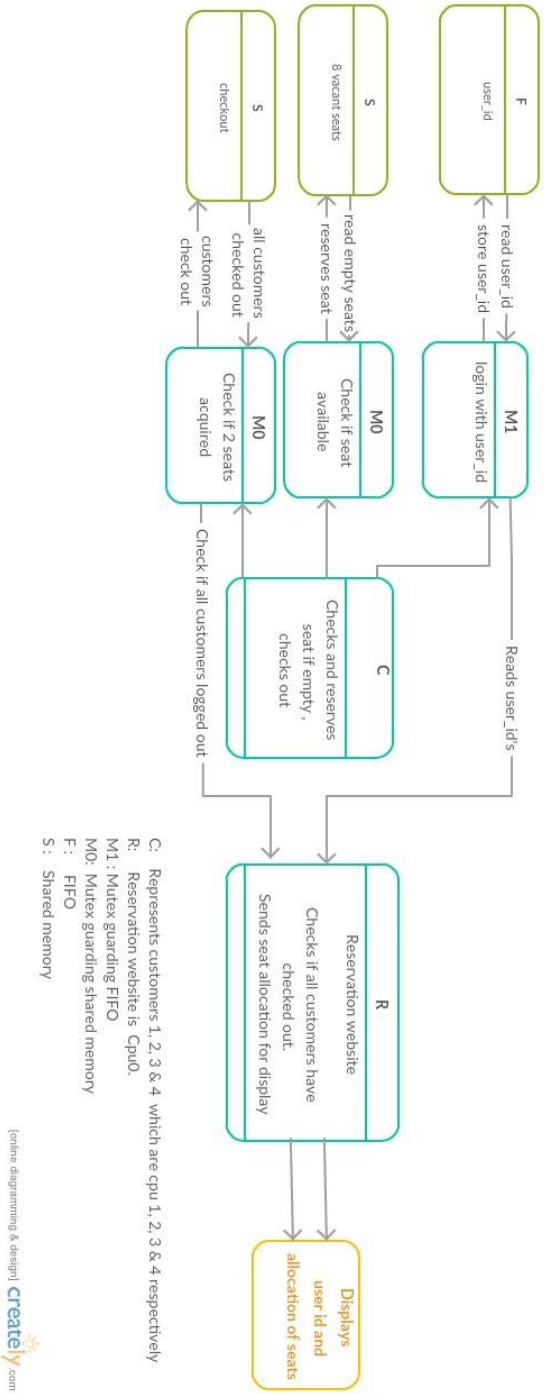

Figure 2. Application diagram.

IV. Results

A. Code Optimizations

The following table shows the `.elf` file statistics shown on the terminal window after executing the given shell script `run_hello_mpsoc.sh` under different code and script modifications.

TABLE I. ELF File Size Comparison

Modifications	cpu_0	cpu_1	cpu_2	cpu_3	cpu_4
given code/script	4876	2160	2160	2160	1352
Min. Char. API [b]	4876	1736	1736	1736	1700
Same flags [c]	1723	1352	1352	1352	1352

[a] File size given in bytes. *(nios2-elf-size)*

[b] *Minimal Character API. cpu_4 used alt_printf instead of alt_putstr*

[c] *delay_asm.s and "from" were removed*

While writing the multiprocessor application, code optimization was kept in mind, therefore, a comparison table would be pointless; however, for informational purposes, the following table shows the final `.elf` file sizes obtained.

TABLE II. ELF File Size Comparison

	cpu_0.c	***cpu_1.c***	***cpu_2.c***	***cpu_3.c***	***cpu_4.c***
`Final .elf` [b]	10780	2352	2364	2384	2904
without perf counter	4244	2352	2364	2384	2904

[d] File size given in bytes. *(nios2-elf-size)*

B. Performance Measurements

The following table shows the execution times measured using the performance measurement hardware:

TABLE III. Performance Counter Measurement

	Execution time (µs)
Section 1 - `shared_onchip` initialization	147
Section 2 – total execution time	2964530

After analyzing how the `.elf` files get downloaded to the development board (sequentially from cpu_0 to cpu_4) and how the program actually works (it waits until all CPUs have "reserved two seats", we can see that Section 2's execution time will vary depending on how fast all these `.elf` files are downloaded to the development board. The faster the download, the lower the execution time.

References

[1] TechNotif's Basic Guide of Interprocess Communication and Pipes. http://technotif.com/basic-guide-interprocess-communication-pipes/

[2] Altera's Development and Educational Boards. https://www.altera.com/support/training/university/boards.html

[3] Terasic's Altera DE2 Development Board Product. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=53&No=30

[4] Introduction to the Altera Qsys System Integration Tool. For Quartus II 12.1. October 2012.

ftp://ftp.altera.com/up/pub/Altera_Material/12.1/Tutorials/Introduction_to_the_Altera_Qsys_Tool.pdf

[5] Altera Nios II Processor. https://www.altera.com/products/processors/overview.html

[6] Nios II Classic Software Developer's Handbook. NII5V2. May 2015. https://www.altera.com/en_US/pdfs/literature/hb/nios2/n2sw_nii5v2.pdf

[7] Quartus II Handbook Version 9.1. QII5V1-9.1.1. November 2009 https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/qts/archives/quartusii_handbook_9.1.2.pdf