

# Embedded Software

## Lab 1

Tanoh Henry Gertrude

Farbod Haselzadeh

### I. INTRODUCTION

This report summarizes our work concerning the laboratory 1 of the Embedded Software course. The lab1 consists of understanding the multiprocessor architecture (cores, peripherals, interconnection between cores and peripherals) and developing a demo application showcasing the communication between the cores and the handling of I/O peripherals.

#### A. The multiprocessor

The multiprocessor is composed of five cores. The chosen architecture is such that it has a main core\_0 that has access to the primary memory and all the I/O peripherals and the cores\_1, 2, 3, 4 has no access to the I/O peripherals and have their own memory (on-chip memory). The core\_0 communicate with the peripherals with a master/slave procedure.

The cores can communicate with each other through the shared memory and the message passing.

This architecture is called Asymmetric multiprocessing.

This architecture is suited for embedded systems applications because it allows the designer to develop specific tasks on a single CPU, and to ease the implementation of the coding. This architecture also proves to be faster because of no delay due to "hand-shaking" between cores and flexible because of the possibility of running multiple Operating Systems on the different cores.

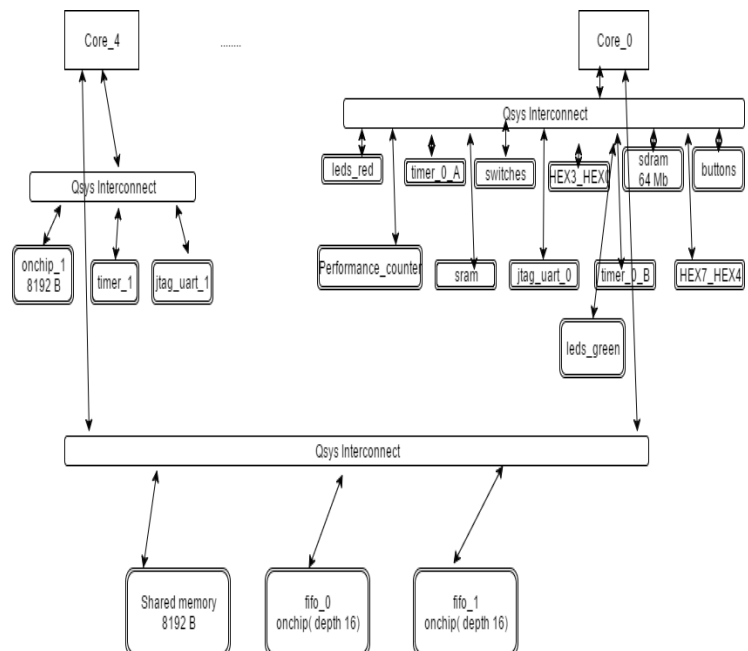
On the other hand, it is up to the designer to implement safe protocols for the communication between cores. Furthermore, this architecture can be inefficient if the user's applications running on the others cores are not using fully the core, making the cores idle.

This architecture is adapted for embedded systems applications.

The cores and the peripherals are connecting through the Qsys interconnect. Qsys is a high-bandwidth structure that allows to connect different components of different data widths or clocks domains. The interfaces are mapped to Avalon Memory Mapped Master/Slave.

#### B. Architecture Diagram

The Architect is the same for cores 1-3 as core 4.



#### C. Demo application

For a safer shared memory and message passing process, we use mutexes to prevent deadlocks and data corruption.

The demo application displays a synchronization process between the four cpus. Cpu\_0 communicate with cpu\_1 and cpu\_2, with cpu\_3 and cpu\_4 by shared memory. Cpu\_3 and cpu\_4 communicate also with each other by shared memory. When a key of the board is pressed, the cpu\_0 read the data matching the key and a cpu id and display it on the seven segment.

#### D. Performance Counter

The performance counter report shows that how long time it is taken to run through a section of code and even the amount of clocks.

Sections that were measured by performance counter was

Reading and writing from/to Fifo and Shared Memory .

Section	Time(usec)	Time(clocks)
Write FIFO 4 data	23	1157
Read FIFO 4 data	18	949
Write Shared Memory	0	4
Read Shared Memory(0/512]	0/11	0/185
Polling Button	3	150

#### II. COST

In our application, we write and read by group of four because we display the data on the seven segment. We decided to measure write/read four data to fifo, write 4 data to shared memory, and read 0 and 512 data to shared memory.

Writing to shared memory in our application doesn't take appreciable time (0) while writing to Fifo takes longer time.

If we were to meet a throughput constraint we would use shared memory more than the Fifo but we still aware that shared memory requires a more complicated design.

#### E. Footprint of the code on each cpu

##### cpu\_0

Statistics

text	data	bss	dec	hex	filename
11344	580	436	12360	3048	lab1_0.elf

##### cpu\_1

Statistics

text	data	bss	dec	hex	filename
5020	328	16	5364	14f4	lab1_1.elf

##### cpu\_2

Statistics

text	data	bss	dec	hex	filename
5020	328	16	5364	14f4	lab1_2.elf

##### cpu\_3

Statistics

text	data	bss	dec	hex	filename
4672	328	20	5020	139c	lab1_3.elf

##### cpu\_4

Statistics

text	data	bss	dec	hex	filename
4492	328	20	4840	12e8	lab1_4.elf

#### F. Code Optimization

We can reduce

By applying size optimizations for code size reduction

--set hal.make.bsp\_cflags\_optimization -O0 ;

Enable Compiler Optimizations

To enable compiler optimizations, use the -O3 compiler optimization level for the nios2-elf-gcc compiler. You can specify this command-line option through a BSP setting. With this option turned on, the Nios II compiler compiles code with the maximum optimization available, for both size and speed.

We can use several BSP settings to reduce footprint.

Some BSP setting is listed below:

hal.enable\_lightweight\_device\_driver\_api

hal.enable\_clean\_exit

hal.enable\_sim\_optimize

hal.enable\_reduced\_device\_drivers

After adding BSP settings in to the shell script we got some optimization. Below is the footprint of cpu 0 that shows the optimization

##### Cpu\_0

Statistics

text	data	bss	dec	hex	filename
11228	580	296	12104	2f48	lab1_0.elf

