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.

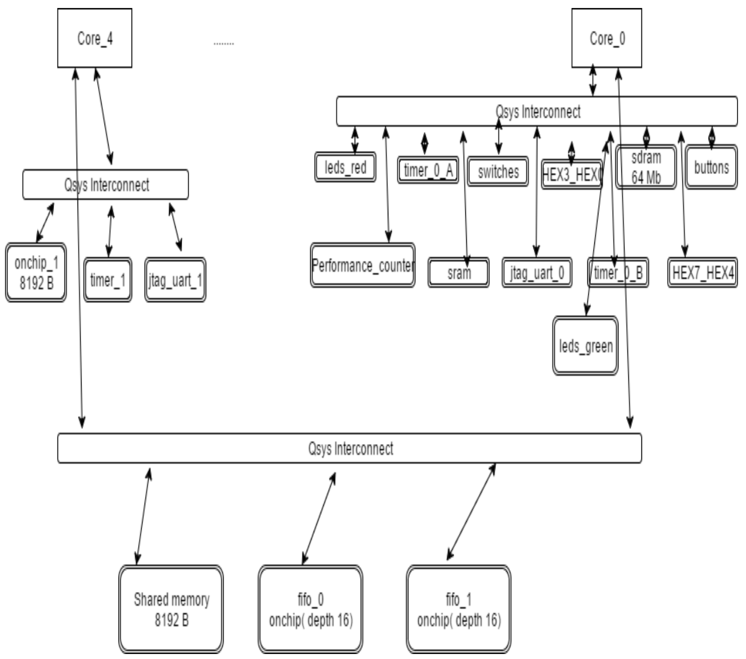
On the other hand, it is up to the designer to implement safe protocols for the communication between cores. Furthermore, this architecture can be inefficient if the user’s applications running on the others cores are not using fully the core, making the cores idle.

This architecture is adapted for embedded systems applications.

The cores and the peripherals are connecting through the Qsys interconnect. Qsys is a high-bandwith 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. In our application, we use one mutex which controls write and read access to the shared memory. If one process wants to write/read to the shared memory it has to get the same mutex to get access to shared memory. As an example when cpu0 owns the mutex and another cpu will write/read to/from shared memory it has to check if the mutex is in used or not.
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 where 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 -Os 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

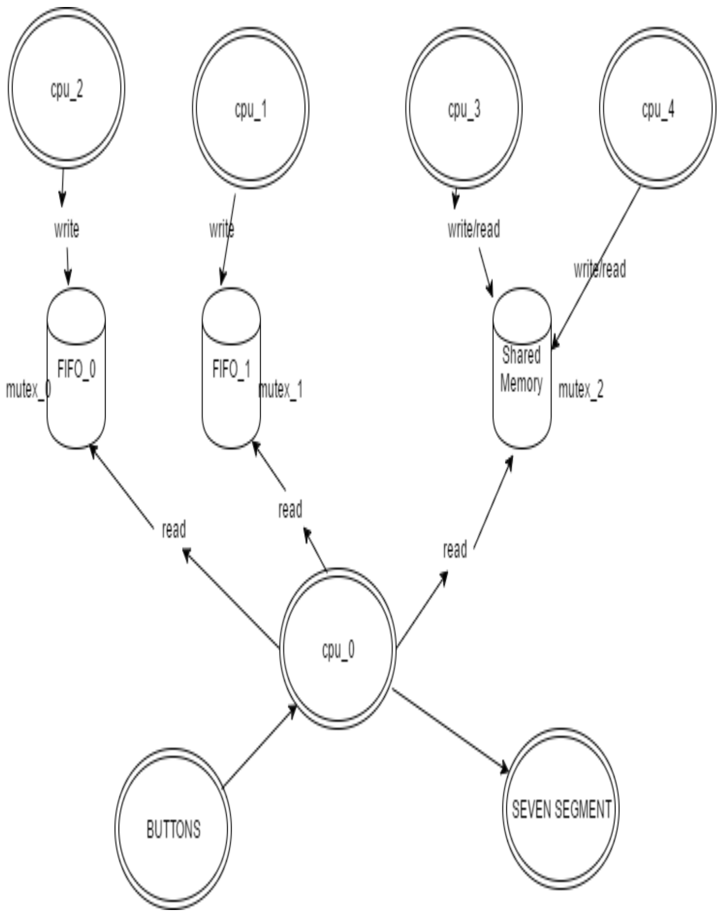
hal.enable_reduced_device_drivers

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
```


cpu_2
cpu_1
cpu_3
cpu_4
write
write
write/read
write/read
FIFO_0
mutex_0
FIFO_1
mutex_1
Shared Memory
mutex_2
read
read
read
cpu_0
BUTTONS
SEVEN SEGMENT