Multiprocessor Communications

Shruti Kuber, Hansell Baran – 2nd Lab 1st Session February 21, 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. Multiprocessor Application

A DSP software application was developed using three different configurations: Single core using a Real Time Operating System (RTOS), Single core without RTOS and a multicore without RTOS. For each implementation, the DSP application does the following:

- Reads a 24-bit RGB image from memory.
- Converts the 24-bit RGB image to 8-bit Grayscale
- Resizes the 8-bit Grayscale image to 50%
- Applies an Edge Detection/Sobel filter to the scaled down image
- Loads the final processed image to memory
- Shows the image using ASCII art in the terminal

Additionally, the application has two modes that can be chosen before generating the `.elf` file: *debug* and *performance* mode.

In debug mode, the application is able to process images of different sizes. After processing each image, the terminal will show the execution time and the resulting image. A 2 second delay was included after processing each image to allow the visualization of each image. This must be done because the *xterm* Linux terminal does not have scrolling functionality. Under Windows, the Nios II Command Shell has scrolling capability and therefore, the delay can be omitted.

The performance mode will only process 32 x 32 pixels images and will show the total execution time after a specific number of images (set before generating the `.elf` file) have been processed. The program cycles through a group of 10 different images to simulate a continuous input stream.

It is important to mention that the images that are going to be processed are character arrays found in the file `images.h` ; this file is added/included in `cpu_0`'s code. In this file, there are two different images, each one with the following sizes: 24 x 24, 32 x 22, 32 x 32, 40 x 28 and 40 x 40. In order to improve, facilitate and diagnose the multicore implementation, 8 new images were added to the "image stream".

Figure 2 shows the memory mapping and the multiprocessor implementation using a high level data flow diagram.

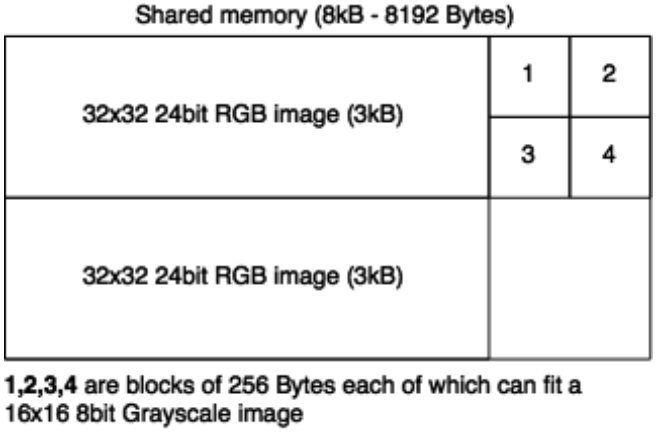

Figure 2. Multiprocessor memory mapping

The chosen shared memory mapping allows `cpu_0` to make 2 images available for the other CPUs at all times. Furthermore, each "slave" CPU (`cpu_1` through `cpu_4`) will be able to put a processed image in the shared memory. This allows data processing parallelism since each CPU is able to process one image without depending on another CPU (the only dependence lies on how fast can cpu_0 keep the image buffer full so that each CPU will wait the least amount of time to read a new image). Another approach was to divide the image into four parts each of which was to be processed by a different CPU. This approach could lead to better throughput but the code and synchronization complexity would increase considerable. In the chosen implementation, each "slave" CPU performs all required operations on a

complete image. Figure 3 shows the data flow diagram for the application.

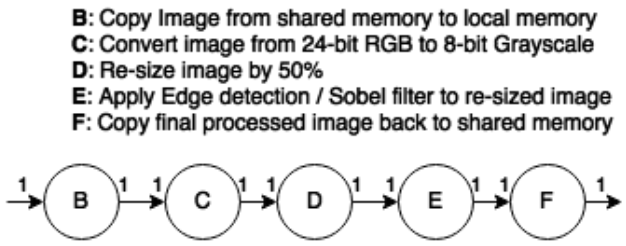

Figure 3. Data flow diagram

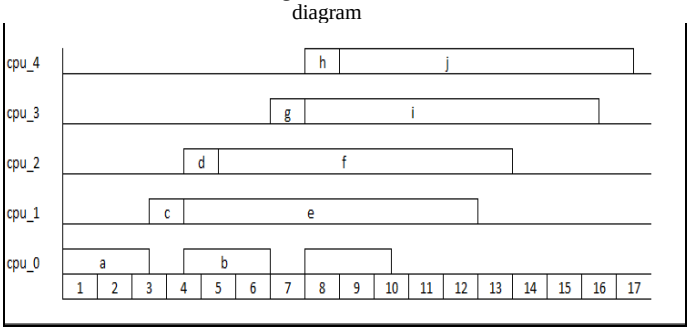

Figure 4. Timing diagram

a – Images 1 and 2 written
b - images 3 and 4 written
c - image 1 read
d – Image 2 read
e - image 1 processed
f - image 2 processed
g - image 3 read
h - image 4 read
i - image 3 processed
j - image 4 processed

Assumption- Total time take to process 420 images was divided into 1000 time units for demonstration. This 1 time unit is approximately equal to 686 micro seconds.

III. Results

The following table shows the memory footprint and the throughput for each implementation.

TABLE I. MEMORY FOOTPRINT AND THROUGHPUT

	Single Core (RTOS)	***Single Core (Bare-Metal)***	***Multicore***
Throughput [s^{-1}]	1.214	1.381	1.5
SRAM (bytes)	117667	14335	15935
OnChip CPU 1 (bytes)	-	-	7952
OnChip CPU 2 (bytes)	-	-	7952
OnChip CPU 3 (bytes)	-	-	7952
OnChip CPU 4 (bytes)	-	-	7952
OnChip (Shared) (bytes)	0	0	7186
Total Memory (bytes)	117667	14965	54929

It is worth mentioning that the SRAM values do not include the `images.h` character arrays. The SRAM values come from the total `.elf` size statistic and by subtracting the given and added sample images (Single core with RTOS: 175516 – 33249 – 24600; Single core without RTOS: 72184 – 33249 – 24600; Multicore without RTOS: 73784 – 33249 – 24600)

Another important aspect to highlight is the values for OnChip (Shared). The application required the CPUs to read from memory. In the single core implementations, since the `images.h` file is included in cpu_0's code, the images are already in memory and there is no need to use the shared memory. Similarly, after the images are processed, the array holding the resulting image is also already in local memory and there is no need to write the images in the shared memory. However, the application has the ability to read and write images to shared memory for all implementations.

A. Optimizations

The performance of the application was significantly increased by performing basic optimizations like replacing multiplications with shifts, arithmetic equivalencies like 0.3125*x = (5/16)*x = (1/4)*x + (1/16)*x = x >> 2 + x >> 4, algorithm/mathematical formula approximations[4], variable re-utilization, etc. Examining figure 3, each node has a specific function; the following optimizations were made:

- **B & F**: Instead of copying the image byte-by-byte, we copy the image using multiple bytes or blocks.
- **C**: Instead of using the given floating point formula: 0.3125Red + 0.5625Green+0.125Blue we used arithmetic equivalences for the floating point values (e.g, 0.3125 = 5/16 = 1/4 + 1/16; 0.5625 = 9/16 = 1/2 + 1/16; 0.125 = 1/8)
- **D**: Instead of using the average of four neighboring pixels, only the pixels in odd numbered rows and columns were taken directly without any averaging function. This will result in an image were edges are more jagged, however, for this application, image quality is of no concern.
- **E**: Rather than an optimization, the application used the following approximation: $G = \sqrt{(Gx2 + Gy2)} \approx |Gx| + |Gy|$. This approximation reduces both code size and execution time and as before, the final image quality is of no concern.

Table II shows the memory footprint and throughput for the optimized application implemented in the three configurations. Table III shows

TABLE II. OPTIMIZED MEMORY FOOTPRINT AND THROUGHPUT

	Single Core (RTOS)	***Single Core (Bare-Metal)***	***Multicore***
Throughput [s^{-1}]	-	138	612
SRAM (bytes)	-	12879	12559
OnChip CPU 1 (bytes)	-	-	3796
OnChip CPU 2 (bytes)	-	-	3796
OnChip CPU 3 (bytes)	-	-	3796
OnChip CPU 4 (bytes)	-	-	3796
OnChip (Shared) (bytes)	-	-	7186
Total Memory (bytes)	-	12879	34929

TABLE III. PERFORMANCE COUNTER - SINGLE CORE BARE METAL

	%	***Time (usec)***	***Time (Clocks)***	***Occurrences***
Execution Time	99	3026199	151309973	1
RGB->Grayscale	60	1816659	90832980	420
Resize	7	230974	11548740	420
Filter	32	977726	48886320	420

REFERENCES

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

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

[3] 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

[4] http://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm