IL2212 Embedded Software Lab Project

Assignment 3

Jon Ander Lezeta, Matteo Mainenti

<lezeta@kth.se, mainenti@kth.se >

March 6, 2019

Contents

Kungliga Tekniska Högskolan (KTH) , Stockholm, Sweden
IL2212 Embedded Software Lab Project

Assignment 3

Jon Ander Lezeta, Matteo Mainenti

March 6, 2019

1 Introduction

This is the report for the Laboratory Project in Embedded Software IL2212 course. The project aims to show the student how to use formal methodologies for system model design and finally implementing the initial design into an application that runs on different hardware architectures. The used methodology for formal system design is ForSyDe. ForSyDe is a methodology for modeling and designing heterogeneous and cyber-physical systems. The objective of ForSyDe is to make sure that System design starts at a high abstraction level, focusing on pure functionality without being worried about the implementation. This methodology uses formal refinement techniques for bridging the transition from design and implementation [1].

As said before, this lab helps the student understand how system modeling works and how it can finally be implemented into an application. The student must understand how to build the bridge from the initial design into a final implementation using system modeling architectures. The lab will be carried out in pairs of two. This does not mean that the work should be split in two, but instead, both students should be aware of everything done through the Project. The project should be presented following three milestones along the project. These milestones are not mandatory but recommended. Missing the deadlines does not have any repercussion, but signals a clear warning to the project group.

This project consists of implementing a data-flow application in different ways into a multi-core processor, implemented on an FPGA. The final goal is to implement an image tracking algorithm, that tracks a moving "circle" pattern in several image frames. In the end, the system should output the same image removing the identified pattern. This will be implemented on a given custom multiprocessor hardware hosted on a DE2 FPGA. In total the system will be implemented in three manners, single core using the Real-Time Operating System MicroC/OS-II, single core in bare-metal and finally multicore in bare-metal as well.

2 Modelling

This section will give an overview of the model used to implement the application into the final Hardware. A Synchronous Data Flow graph SDF is being included as well as the description of the proposed algorithm. On top of that, a description of each subsystem or the designed functions will be described in terms of patterns, data-size, complexity, and data/time dependencies.

2.1 Overview

The model being described in this subsection was given to the student through a Haskell program written using the ForSyDe Methodology for Formal System Design developed in KTH. This Model represents a concurrent data flow application on which an image tracking algorithm is designed. The objective of this model is to track a given pattern (circle) in a series of images for the purpose of future processing. This model outputs the coordinates where the pattern has been found or detected.

2.2 SDF graph

The model being described in this subsection was given to the student through a Haskell program written using the ForSyDe Methodology for Formal System Design developed in KTH. This Model represents a concurrent data flow application on which an image tracking algorithm is designed. The objective of this model is to track a given pattern (circle) in a series of images for the purpose of future processing. This model outputs the coordinates where the pattern has been found or detected. The input image is introduced to the model formatted in the "PPM" or "Netpbm" format. The output of the model is given using (x,y) coordinates that indicate what is the center of the pattern detected inside the image. The coordinates (x,y) represent the horizontal and vertical values of the pixels within the image.

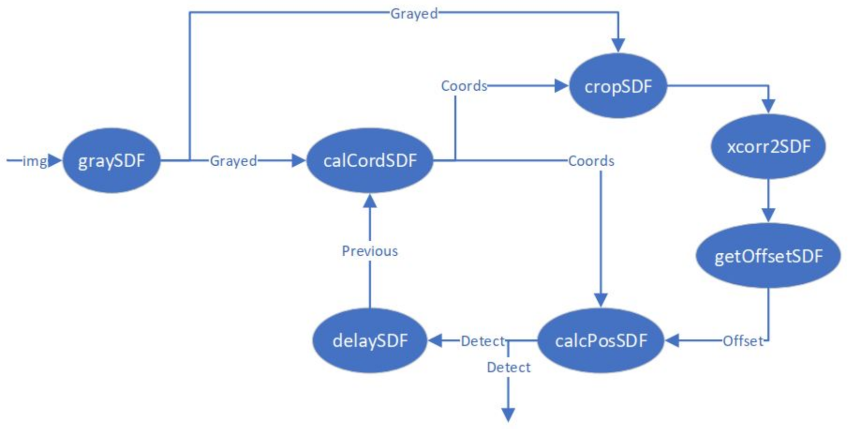


Figure 1: SDF graph of the ForSyDe model

2.3 Description of algorithm

This section describes what each subsystem of the SDF graph in Figure 2.2 does in terms of the intrinsic functionality, input/output data and dependencies. The general input for the model is the image that will be analyzed in "PPM" or "Netpbm" format. The input matrix is formed by several images from the same size that they represent the movement of a pattern if all images are visualized sequentially. Each line of the matrix represents a separate image.

2.3.1 graySDF

This function takes an input matrix with integer values. As specified in section 2.2, the input matrix is basically a vector formed by vectors corresponding to the *PPM* format. The *PPM* format is formed by a header element and a body. The header contains three values, the *width*, the *height* and the maximum value for the pixels in the body. The body contains all the *r, g,* and *b* values for each pixel from 0 to *maxVal*. Thus, the input matrix can be represented as in M_{in}, and the vector in PPM format representing an image by V_{rgb}.

$$V_{rgb} = \begin{pmatrix} width & height & maxVal & r & g & b & r & g & b & \cdots & r & g & b \end{pmatrix}$$

$$M_{in} = \begin{pmatrix} r & g & b & r & g & b & \cdots & r & g & b \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ r & g & b & r & g & b & \cdots & r & g & b \end{pmatrix}$$

As said before, each group of *r, g, b* represent the colour of each pixel in the image. This function *graySDF* takes the input pixels and converts them into the gray scale. The gray value of an *r, g, b* pixel can calculated by Equation 1.

$$f(r,g,b) = r * 0,3125 + g * 0,5625 + b * 0,125 \tag{1}$$

As a consequence, the width of the matrix gets divided by three. Thus, the output matrix can be represented as matrix $M_{grayscale}$, with double floating point values. The value of g goes from 0 to $maxVal$ which is 255 in the example given.

$$M_{grayscale} = \begin{pmatrix} width & height & maxVal & gray & gray & \cdots & gray \\ width1 & height1 & maxVal1 & gray & gray & \cdots & gray \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ widthn & heightn & maxValn & gray & gray & \cdots & gray \end{pmatrix}$$

2.3.2 calCordSDF

This function calculates the coordinates that will be analyzed based on some input coordinates coming from delaySDF indicating what values were previously checked. This function will only be executed once data is ready to be consumed from *graySDF* and *delaySDF*. Data from *graySDF* is given in the form of a double floating point matrix, and data from *delaySDF* is given in the form of 2D coordinates (*x,y*), being horizontal and vertical values of the previously analyzed coordinates respectively. The output of the function *calCordSDF* is given in the same format (*x,y*), which will be used by the function *cropSDF* in the section 2.3.3.

2.3.3 cropSDF

This function requires two separates inputs, the cropping (*x,y*) coordinates and the image matrix in the grayscale $M_{grayscale}$. This function *cropSDF* will never be executed until data is ready to be consumed from both *graySDF* and *calCordSDF*. Once data is available and ready to be consumed, this function *copSDF* takes the matrix from *graySDF* and the coordinates in format (*x,y*) coming from *cropSDF*, where *x* and *y* are the horizontal and vertical values of the pixel in the image. The coordinates indicate the reference that is used by the function and crop the image/matrix coming from *graySDF*. The size of the sub-matrix generated by the crop is predefined by [*cropSIZE, cropSIZE*] and is given by equation 2 where *dSPAN = 15*. This sub-matrix is the output data being used later by *xcorr2SDF*.

$$cropSIZE = dSPAN * 2 + 1 \tag{2}$$

2.3.4 xcorr2SDF

The function *xcorr2SDF* takes the cropped matrix or the sub-matrix coming from *cropSDF* that has the dimension of [*cropSIZE, cropSIZE*] in terms of column and line numbers. The objective of this application is to fins a predefined pattern inside a given function. This pattern is included inside this function, which will be used for analyzing the cropped matrix coming from *cropSDF*. The dimension of the pattern is [5,5] in this application. The shape of the image/pattern to be found is represented by '*1*' and the leftover or not interesting values are represented by '*0*'. Firstly the size of the pattern matrix is checked by the *xcorr2SDF* function. Once the size of the pattern is known, all the possible stencil sub-matrices are calculated forming another matrix. Now, the data is in form of a 4 Dimension Matrix. As the size of the cropped matrix is [*cropSIZE,cropSIZE*], the 4D matrix will be formed by [27,27] stencil matrices. Since both the pattern and the calculated individual stencil sub-matrices have the same size or dimension, a pair-wise multiplication is performed . The output of the binary operation will be a matrix of the same size as seen in $Msub_{xcorr2}$. This happens because all the stencil sub-matrices and the pattern matrix have the same dimension. All the *g* values in the matrix $Msub_{xcorr2}$, represent every gray value from 0 to 255.

$$Msub_{xcorr2} = \begin{pmatrix} 0 & 0 & \mathbf{g} & 0 & 0 \\ 0 & \mathbf{g} & 0 & \mathbf{g} & 0 \\ \mathbf{g} & 0 & 0 & 0 & \mathbf{g} \\ 0 & \mathbf{g} & 0 & \mathbf{g} & 0 \\ 0 & 0 & \mathbf{g} & 0 & 0 \end{pmatrix}$$

Finally, a reduce action is applied in all the sub-matrices, where all the values are summed up. The output of this function will be a 2 dimension matrix where result of the reduce action of each $Msub_{xcorr2}$ will be represented. The final or output matrix is represented by M_{xcorr2}.

$$M_{xcorr2} = \begin{pmatrix} M_{xcorr2-0x0} & M_{xcorr2-0x1} & \cdots & M_{xcorr2-0x26} \\ \vdots & \vdots & \vdots & \vdots \\ M_{xcorr2-26x0} & M_{xcorr2-26x1} & \cdots & M_{xcorr2-26x26} \end{pmatrix}$$

2.3.5 getOffsetSDF

This function takes the output matrix coming from *xcorr2SDF* where sum of the elements of each individual $Msub_{xcorr2}$ is represented in M_{xcorr2}. The function *getOffsetSDF* is executed once data coming from M_{xcorr2} is ready to be consumed. The maximum value of a pixel in the gray scale is of 255 (white) as stated in section 2.3.1. Thus, this function searches for the highest value of M_{xcorr2}, and returns the position of that element in (x,y) format.The pattern that is being analyzed is a white diamond. Thus, the pattern to be detected inside the sub-matrix will be the one with the highest value in M_{xcorr2}. This function just returns the coordinates of the maximum value of M_{xcorr2}, representing the coordinates of the pattern inside the cropped matrix in *cropSDF*.

2.3.6 calcPosSDF

The objective of this function *calcPosSDF* is very straightforward. Once data coming from *getOffsetSDF* and *calCoordSDF* is ready to consume, this function is executed. The function *getOffsetdSDF* provides the coordinates of the pattern inside the cropped matrix/image. However, the final goal is to identify where inside the input matrix will be the pattern located. This function uses data coming to *getOffsetSDF* and *calCoordSDF* to identify the final position. The function *calCoordSDF* selects the coordinates to crop the initial image. Thus, we have all the required information for locating the pattern in the initial matrix. This function sums both coordinates in the format (x,y) and adds an offset of $(2,2)$ to point exactly to the center of the identified pattern. The output will also be in the format of (x,y).

2.3.7 delaySDF

This function *delaySDF* provides the initial values that the function *calCordSDF* requires to be fired. After the initial values are consumed, the output generated by *calcPosSDF* will be directly bridged, so the loop will be closed and the algorithm can continue working correctly.

3 Hardware (MPSoC)

This laboratory requires to use a personalized Hardware using Nios II soft core processors implemented into an Altera/Intel FPGA. During the course IL2206 *Embedded Systems*, a proper design of a single-core Nios II Soft-core processor and its required peripheral was required. In this course IL2212 *Embedded Software*, the Hardware is given, and it should be programmed accordingly. As an overview, the application to be implemented will be running on a Multi-Processor System-on-Chip *MPSoC*. This system should be implemented on the Altera FPGA provided at the beginning of the lab.

3.1 Interconnection

The analysis of the MPSoC Hardware has been done using the Qsys Platform Designer system integration tool using the Intel Quartus software. This tool helps you see all the internal Hardware components and their interconnection. Since the Hardware was provided, this tool has been used for checking the interconnection between CPU-s and peripherals and understand precisely the provided HW architecture.

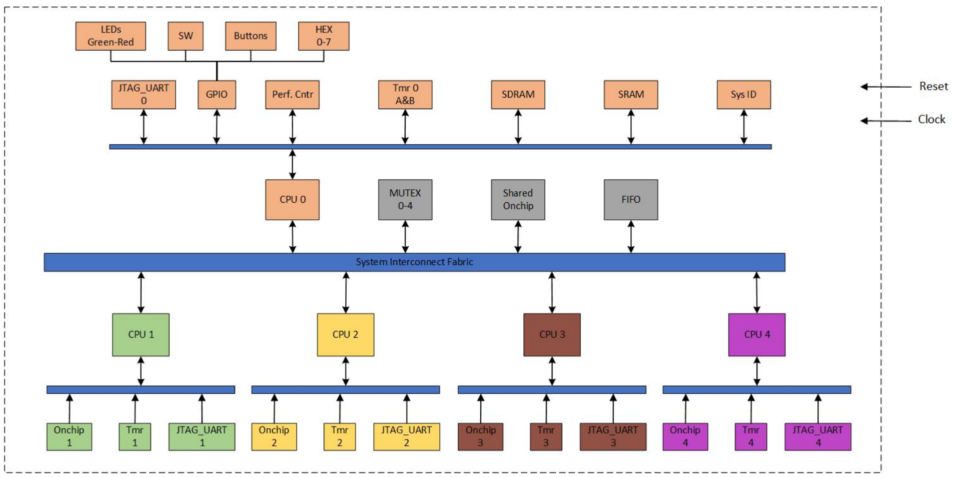


Figure 2: Graph of the Hardware Architecture MPSoC

Figure 2 represents the graph of the HW Architecture being implemented into the Altera FPGA. As it can be seen in the image, The main interconnection is the *System Interconnect Fabric*, where all the main peripherals and CPU-s are connected. The *Clock* and the *Reset* are displayed as separate input. However, it is good to mention that all the peripherals share the Clock and Reset signals. The main CPU is the *CPU 0*, which has all the main peripherals that can be seen in the following list.

- JTAG UART 0
- General Purpose Input/Output GPIO
- Performance Counter
- Timer 0 (A and B)
- SDRAM (8MB)
- SRAM (65KB)
- System ID

However, some other peripherals are required to be directly connected to the System Interconnect Fabric, so that all the separate CPUs can have access to it. This peripherals usually remain accessible to everyone because they share resources/characteristics that can be used by any core in the Hardware implementation. The following list contains all the peripherals connected to the main Interconnection Fabric.

- MUTEX 0-4
- Shared Onchip (8KB)
- FIFO

The implemented Hardware Architecture has in total five cores. As specified before, the core *CPU 0* is considered to be the main core. Instead, *CPU 1, CPU 2, CPU 3* and *CPU 4* are considered as secondary cores. All the units have the same architecture, as in the following list:

- On-chip Memory (8KB)
- Timer
- JTAG UART

It is important to specify that all the used *CPU*-s (*CPU0*,*CPU1*,*CPU2*,*CPU3*,*CPU4*) are the economic version of the Altera Nios II Core with 32 bits.

3.2 Avalon Switch Fabric

The Avalon Switch Fabric is the *Network on Chip NoC* that is implemented in this *MPSoC* Hardware. This Interconnection provides a number of significant advantages over regular buses or interconnections. The implementation, in this case, is performed by the Platform Designer, Qsys, allowing the designer to create high-performance systems easily, without deep and advanced knowledge on *NoC* Interconnects. On top of that, this platform generates an optimized FPGA *NoC* interconnect based on the requirements of the applications. *NoC* does not treat the interconnection as a single point to point connection, instead, it treats the interconnect as a protocol with its different functions. Each Master or Slave in the architecture is connected to a Network Interface *NI* that captures the transaction and delivers the appropriate packets which are routed towards its destination. The main benefit of *NoC* is that implements independent implementation and optimization layers, customization, multiple topologies, simplified development, interoperability, and scalability. The used Interconnection Fabric permits data read/writes of 32 bits, which needs to be taken into account for granting maximum data transfer speed and exploit full benefits of the fabric itself [2].

4 Implementations

As said before, the *MPSoC* Hardware architecture has been initially provided, meaning that not all the included and explained peripherals will be used in the Software application. In this specific application, the following peripherals are being used. The following subsections explain which ones are being used for both, the Single-core and Multi-core implementation. These assignments are divided into different implementations of the initial design, which will be explained in the following subsections.

4.1 Single core (bare-metal)

The first implementation of the initial design model has been made using a Single-core Hardware architecture in Bare-metal. This means that there is no operating system or scheduler to execute different tasks. The first implemented application replicates the initial design, having one function per actor of the initial model design. The functional explanation for each actor has been previously explained in 2. As said, a function has been developed for each actor in Figure 1. However, we consider that this is not the best implementation.

There are other ways of developing the application always preserving the semantics of Figure 1. These improvements will later be explained in the document. When it comes to HW architecture, a single core implementation simplifies the complexity of the system initially. A single core implementation requires just one computing unit or CPU, thus, CPU 1-4 and their respective peripherals (On-chip, Tmr, and JTAG UART) are not used. On top of that, the FIFO queuing peripheral and the MUTEX is neither used, as there is no resource that has to be protected against shared resource troubles. The memory used for storing the images and variables is the *SRAM*, thus, the *SDRAM* is not used. In this implementation, none of the *GPIO* ports are used. The *Timer* is used by other peripherals, and the *Performance Counter* for measuring the performance of the whole implementation.

Single core (Bare Metal)	
Throughput [s^-1]	SRAM [bytes]
31	9

Table 1: Single core (Bare Metal) results

Function	Execution time (s)
GRAYScale	0.0127
Coords	0.00001
Crop	0.0015
xCORR2	0.01403
POSMAX	0.0033
OffstetPos	0.0012

Table 2: Execution time of functions in single core (Bare Metal)

4.2 Single core (RTOS)

This implementation shares the Hardware architecture of the Bare-Metal implementation. However, this application uses a Real-Time Operating System *RTOS* for scheduling all tasks and preserve the correctness of the semantics on the *SDF* graph in Figure 1. The initial implementation has been done by translating each actor of the initial program into one task. Thus, each task will preserve the individual functionality of every single actor. However, in order to preserve the overall functionality of the system, the scheduling of each task has to be performed in an adequate way. There are two ways of synchronizing this task. The first one would be using mailboxes or message queues. However, in this case, we implemented a synchronization using semaphores, so we ensure that each task is executed in the chosen order, and thus preserve the semantics of the initial model. The following table shows the result of the performance of the system.

4.3 MPSoC (bare-metal)

The last and third implementation has been the *MPSoC* Multicore implementation in Bare-Metal. The software application developed in this case has been different from the ones

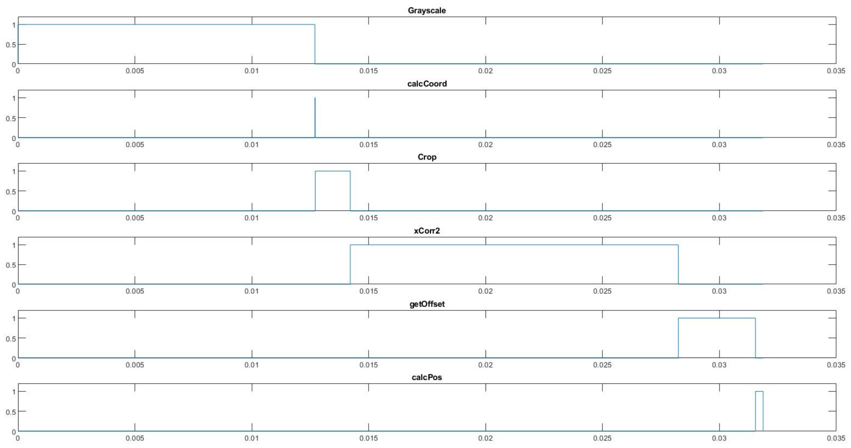


Figure 3: Bare Metal Scheduling graph

Single core (RTOS)	
Throughput [s^-1]	SRAM [bytes]
25	74900

Table 3: Single core (RTOS) results

applied in the Single core. In this case, the *CPU0* acts as a "master" over the "slaves" *CPU1, CPU2, CPU3 and CPU4.* The master is in charge of cropping the input picture in the *SRAM* and storing it in the Shared-Memory. All the slaves will take one-fourth part of the cropped and gray-scale the image int the Shared-memory. All the "slave"-s *CPU1, CPU2, CPU3, and CPU4* will perform the same operations over the sub-cropped image. Each one will apply the Grayscale, xcorr2, and calcPos functions. Then, the algorithm will return the coordinates of the most similar to the pattern picture and a pattern matching sum value. These values will be stored in the Shared-memory with the respective coordinates. The highest of these values will be referring to the pattern to be found. Finally, the "master" *CPU0* will compare all the values coming from each "slave" *CPU* and output the coordinates

Function	Execution time (s)
GRAYScale	0.018
Coords	0.00001
Crop	0.00195
XCORR2	0.01707
POSMAX	0.00229
OffstetPos	0.00063

Table 4: Execution time of functions in single core (RTOS)

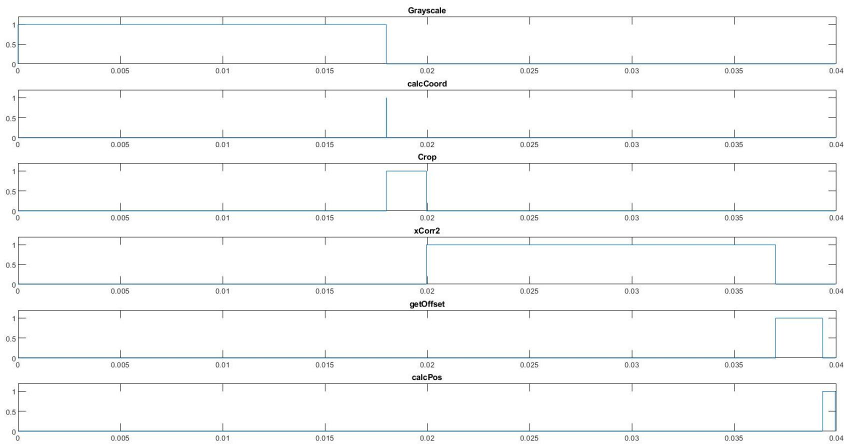


Figure 4: RTOS Scheduling graph

of the detected pattern in the image. Even though this algorithm has no actor/function direct relation, the semantics of the whole implementation are the same as in the given model. This means that the correctness is ensured from the design to the implementation, which was one of the main goals of this Laboratory session.

Multi-core (MPSoC)							
Throughput (s^-1)	SRAM [bytes]	OnChip CPU 1 [bytes]	OnChip CPU 2 [bytes]	OnChip CPU 3 [bytes]	OnChip CPU 4 [bytes]	OnChip Shared [bytes]	Total Memory [bytes]
33	11264	5680	5680	5680	5680	3096	37080

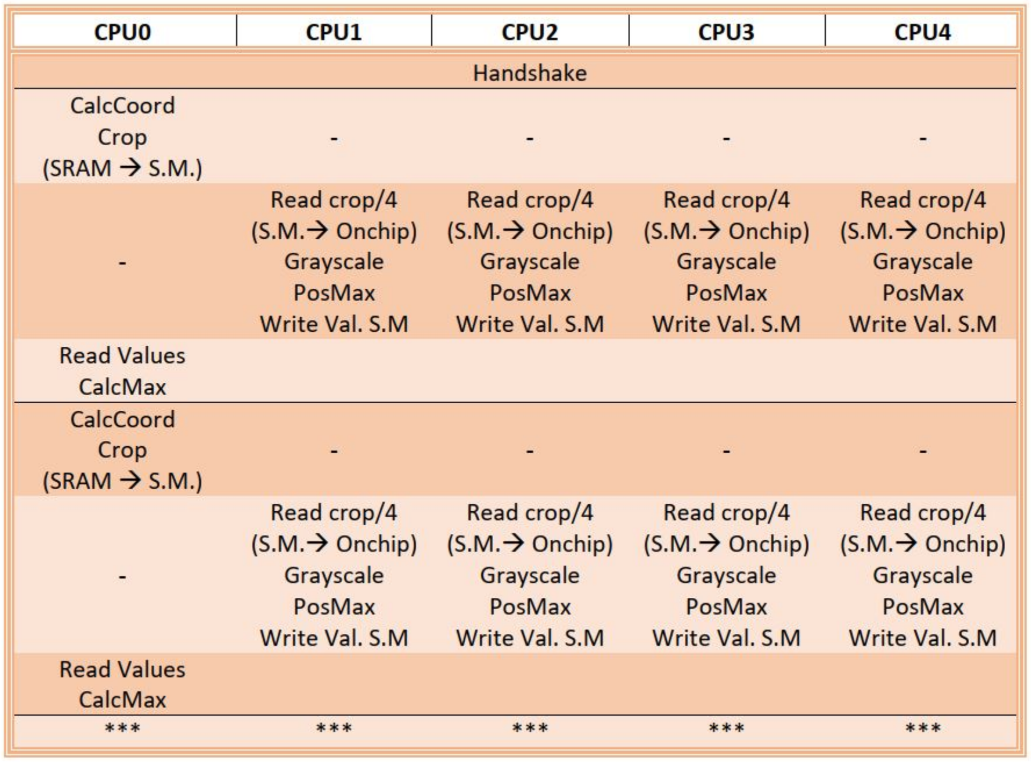
Table 5: Multi-core (MPSoC) results

4.4 Optimizations

4.4.1 Bit-shift operation

Floating operations can be really costly especially in non-specialized Hardware applications. In our case, we don't have a floating point unit to make these float operations by Hardware. That is why these operations were carried out purely by software, resulting in really slow performance. Thus, we had to look for alternatives for this drawback. The issued floating point operation is being executed in the *GRAYScale* function, where the instruction in 1. The multiplications with the floating numbers make the program execute slowly. As an alternative, bit-shifting has been used. We know that a 0.5 can be represented as in Function 3. Thus, the following function 4 was written representing the same floating point function but using bit-shift operations.

$$num * 0.5 = num >> 1 \tag{3}$$



CPU0	CPU1	CPU2	CPU3	CPU4
		Handshake		
CalcCoord Crop (SRAM → S.M.)	-	-	-	-
-	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M
Read Values CalcMax				
CalcCoord Crop (SRAM → S.M.)	-	-	-	-
-	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M	Read crop/4 (S.M.→ Onchip) Grayscale PosMax Write Val. S.M
Read Values CalcMax				
***	***	***	***	***

Figure 5: MPSoC Scheduling graph 1

$$grayVal = (r >> 2 + r >> 4) + (g >> 1 + g >> 4) + (b >> 3) \tag{4}$$

4.4.2 Burst read/write

The implemented Avalon Switch Fabric permits data read/write of 32 bits or 4 bytes. However, since initially, we were read and writing char-wise, one fourth (one-byte data read/write) of the bus capabilities were being used. Instead, if we use 32 bit data types for read/write operations, we could transmit 4 number over the same instruction. This optimization improved all the data transmissions from SRAM, Shared Memory and Onchip memories. The result is four times faster read and write operations.

4.4.3 HW communication

Initially we were using flags for communication between different CPU-s. We initially detected that this implementation is really slow. This happens because one CPU is constantly accessing the Shared Memory to check the flag values and know when to continue the execution. This CPU will be constantly accessing the Shared memory, while the others need to read/write values from the same resource as well. Thus, the program execution is really slow. However, using all the HW mutex-es for blocking the execution of each CPU can ensure that the share resources are accesed in a proper way. The result is that there is no communication time losses and you can exploit full potential of your algorithm.

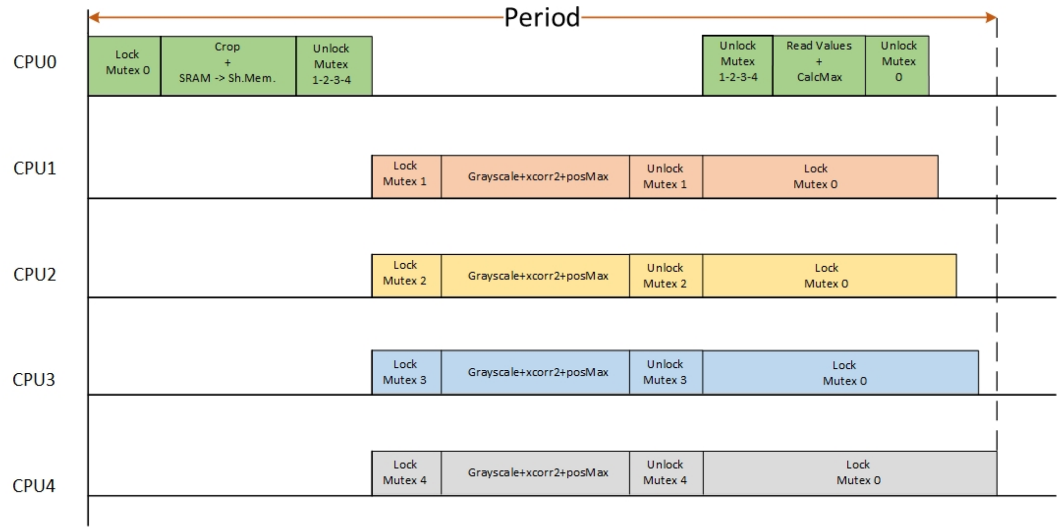


Figure 6: MPSoC Scheduling graph 2

4.4.4 Crop and Grayscale merge

This application just analyzes the cropped part which is a subsection of the input image. The first algorithm was applying the grayscale function to every value in the input image matrix. However, not all the gray-scaled points were being used by the future function, just the cropped subsection is analyzed. The cropped image is one-fourth of the input image, meaning that we will just gray-scale the cropped part. Thus, the ”new” grayscale function takes one-fourth time of the original grayscale function.

4.4.5 Results after optimization

Finally, all the improvements explained through this chapter have been implemented just in the Multi-core implementation due to lack of time. We saw that before optimization the Multi-core was the fastest implementation so we had to make a decision on which one to optimize, being the *MPSoC*.

After all the optimizations where implemented, we finally got a throughput of 357 images/second. Initially we just implemented the bitshift, datatype and the crop optimizations, but we realised we were not having the expected improvements. After checking the execution times of each CPU, we realised that in our case the communication based on flags was not the most optimal solution. That is why we had to switch to pure Mutex communication. After solving our bottleneck, we got the solution that can be seen in the following table 6.

References

[1] ForSyDe:A Methodology for Formal System Design,
`https://forsyde.github.io/`

Multi-core (MPSoC)							
Throughput (s^-1)	SRAM [bytes]	OnChip CPU 1 [bytes]	OnChip CPU 2 [bytes]	OnChip CPU 3 [bytes]	OnChip CPU 4 [bytes]	OnChip Shared [bytes]	Total Memory [bytes]
357	11264	5680	5680	5680	5680	3096	37080

Table 6: Multi-core (MPSoC) results, after optimization

[2] Applying the Benefits of Network ona Chip Architecture to FPGA System Design
`Intel FPGA White Paper`