RYERSON UNIVERSITY
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

4K Ultra-HD Video Noise Reduction System

Muhammad Obaidullah - 500671408
mobaidullah@ryerson.ca

Abdullah Siddiqui - 500390829
abdullah.siddiqui@ryerson.ca

Adeem Mustafa - 500733414
adeem.mustafa@gmail.com

Dr. Lev Kirischian
lkirisch@ee.ryerson.ca

Electrical & Computer Engineering Department, Ryerson University,
Toronto, M5B 2K3, Canada

December 19, 2015

Abstract

4K resolution video recording is becoming the facto standard for professional high definition video. In this paper we design, discuss, and implement a multi-modal application consisting of FIR filter and RGB-to-GrayScale Converter modes operating on 4K resolution 60 fps video capable of either reducing noise or converting RGB to gray-scale.

1 PROJECT SPECIFICATIONS

1.1 FUNCTIONAL SPECIFICATIONS

Total No. of Application Modes	2
Simultaneous Application Modes	No
Application Modes	Noise Reduction Mode & Gray-scale Mode
Color Resolution (R,G,B)	8-bit
RISC/CISC Execution clock cycles for Multiplication	4 c.c.
RISC/CISC Execution clock cycles for Add/Clear...etc.	1 c.c.
Hardware clock cycles for Multiplication	2 c.c.
Hardware clock cycles for Addition	1 c.c.

1.1.1 NOISE REDUCTION MODE

Red Channel output is given by:

$$Y_R(x,y) = \frac{\sum_{i=x-1}^{i=x+1} \sum_{j=y-1}^{j=y+1} P_R(i,j)}{9} \tag{1.1}$$

Green Channel output is given by:

$$Y_G(x,y) = \frac{\sum_{i=x-1}^{i=x+1} \sum_{j=y-1}^{j=y+1} P_G(i,j)}{9} \tag{1.2}$$

Blue Channel output is given by:

$$Y_B(x,y) = \frac{\sum_{i=x-1}^{i=x+1} \sum_{j=y-1}^{j=y+1} P_B(i,j)}{9} \tag{1.3}$$

1.1.2 Gray-scale Mode

Gray Channel output is given by:

$$Y_{Gray}(x,y) = 0.3 \times P_R(x,y) + 0.59 \times P_G(x,y) + 0.11 \times P_B(x,y) \tag{1.4}$$

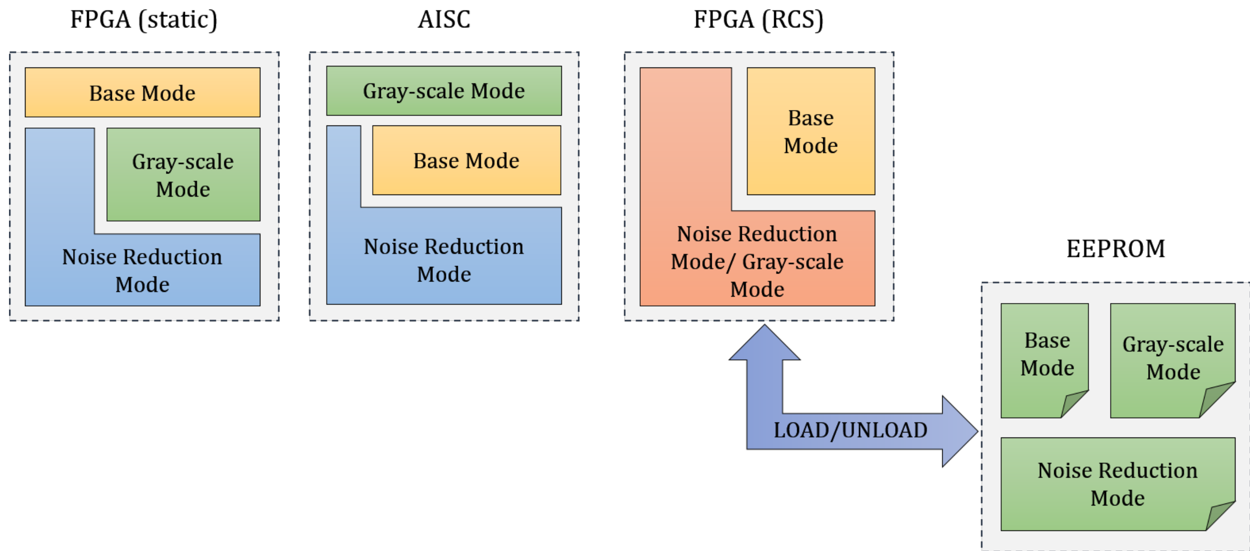


Figure 1.1: Three possible implementations.

1.2 Technical Specifications

Performance	60 fps
Resolution	4K $= 3840 \times 2160 = 8,294,400\ pixels/frame$
Total Application Modes	2 (Noise Reduction Mode & Gray-scale Mode)
Logic Cells (Noise Reduction Mode)	58,150
Logic Cells (Gray-scale Mode)	49,895
Logic Cells (Base Mode)	17,503

Two modes of operation (Noise Reduction Mode & Gray-scale Mode) require base mode logic to operate.
Available FPGA devices in the market:

FPGA Device	Number of Logic cells	Configuration Bits	Approximate cost for 100 units	Approximate cost for 1000 units	Approximate cost for 10,000 units	Max. Operating Frequency (MHz)
Virtex-7 XC7V2000T	19,54,560	447,337,216	$16,500	$14,025	$13,200	741
Kintex-7 XC7K160T	162,240	53,540,576	$354	$300	$283	741
Virtex-6 XC6VLX130T	128,000	43,719,776	$1,465	$1,245	$1,172	1600
Zynq-7000 XC7Z020-1CLG484C	85,000	32,364,512	$895	$761	$716	1000
Artix-7 XC7A15T	16,640	17,536,096	$56	$48	$45	628
Artix-7 XC7A35T	33,280	17,536,096	$88	$74,800	$704,000	628
Artix-7 XC7A50T	52,160	17,536,096	$111	$94	$89	628
Artix-7 XC7A75T	75,520	30,606,304	$168	$143	$134	628
Artix-7 XC7A100T	101,440	30,606,304	$251	$213	$203	628
Artix-7 XC7A200T	215,360	77,845,216	$ 269	$229	$215	628

Figure 1.2: All prices are taken from Digi-Key Electronics Canada. 0% discount for 100 units, 15% discount for 1000 units, and 20% discount for 10,000 unit

2 RISC/CISC Software Implementation

Address	Operation	Operand 1	Operand 2	Result location
0x10000	Load	C_1 in Mem[1F00002]		Store result in Reg. A
0x10002	Load	C_2 in Mem[1F00004]		Store result in Reg. B
0x10004	Load	C_3 in Mem[1F00006]		Store result in Reg. C
0x10006	Load	C_4 in Mem[1F00008]		Store result in Reg. D
0x10008	Load	C_5 in Mem[1F0000A]		Store result in Reg. E
0x1000A	Load	C_6 in Mem[1F0000C]		Store result in Reg. F
0x1000C	Load	C_7 in Mem[1F0000E]		Store result in Reg. G
0x1000E	Load	C_8 in Mem[1F00010]		Store result in Reg. H
0x10010	Load	C_9 in Mem[1F00012]		Store result in Reg. I
0x10012	Load	Operand in X	1F00000	Store result in Reg. X
0x10014	Load	PR_1 in Mem[X] + 0		Store result in Reg. PR1
0x10016	Load	PR_2 in Mem[X] + 1		Store result in Reg. PR2
0x10018	Load	PR_3 in Mem[X] + 2		Store result in Reg. PR3
0x1001A	Load	PR_4 in Mem[X] + 3		Store result in Reg. PR4
0x1001C	Load	PR_5 in Mem[X] + 4		Store result in Reg. PR5
0x1001E	Load	PR_6 in Mem[X] + 5		Store result in Reg. PR6
0x10020	Load	PR_7 in Mem[X] + 6		Store result in Reg. PR7
0x10022	Load	PR_8 in Mem[X] + 7		Store result in Reg. PR8
0x10024	Load	PR_9 in Mem[X] + 8		Store result in Reg. PR9
0x10028	Multiply	Operand in PR_1	Operand in A	Store result in Reg. PR1
0x1002A	Multiply	Operand in PR_2	Operand in B	Store result in Reg. PR2
0x1002C	Multiply	Operand in PR_3	Operand in C	Store result in Reg. PR3
0x1002E	Multiply	Operand in PR_4	Operand in D	Store result in Reg. PR4
0x10030	Multiply	Operand in PR_5	Operand in E	Store result in Reg. PR5

0x10032	Multiply	Operand in PR_6	Operand in F	Store result in Reg. PR6
0x10034	Multiply	Operand in PR_7	Operand in G	Store result in Reg. PR7
0x10036	Multiply	Operand in PR_8	Operand in H	Store result in Reg. PR8
0x10038	Multiply	Operand in PR_9	Operand in I	Store result in Reg. PR9
0x1003A	Add	Operand in PR_1	Operand in PR_2	Store result in Reg. J
0x1003C	Add	Operand in PR_3	Operand in PR_4	Store result in Reg. K
0x1003E	Add	Operand in PR_5	Operand in PR_6	Store result in Reg. L
0x10040	Add	Operand in PR_7	Operand in PR_8	Store result in Reg. M
0x10042	Add	Operand in J	Operand in K	Store result in Reg. N
0x10044	Add	Operand in L	Operand in M	Store result in Reg. O
0x10046	Add	Operand in N	Operand in O	Store result in Reg. P
0x10048	Add	Operand in P	Operand in PR_9	Store result in Reg. YR
0x1004A	Store	Operand in YR		Result in Mem[X] + 9
0x10042	Load	PG_1 in Mem[X] + 10		Store result in Reg. PG1
0x10044	Load	PG_2 in Mem[X] + 11		Store result in Reg. PG2
0x10046	Load	PG_3 in Mem[X] + 12		Store result in Reg. PG3
0x10048	Load	PG_4 in Mem[X] + 13		Store result in Reg. PG4
0x1004A	Load	PG_5 in Mem[X] + 14		Store result in Reg. PG5
0x1004C	Load	PG_6 in Mem[X] + 15		Store result in Reg. PG6
0x1004E	Load	PG_7 in Mem[X] + 16		Store result in Reg. PG7
0x10050	Load	PG_8 in Mem[X] + 17		Store result in Reg. PG8
0x10052	Load	PG_9 in Mem[X] + 18		Store result in Reg. PG9
0x10054	Multiply	Operand in PG_1	Operand in A	Store result in Reg. PG1
0x10056	Multiply	Operand in PG_2	Operand in B	Store result in Reg. PG2
0x10058	Multiply	Operand in PG_3	Operand in C	Store result in Reg. PG3
0x1005A	Multiply	Operand in PG_4	Operand in D	Store result in Reg. PG4
0x1005C	Multiply	Operand in PG_5	Operand in E	Store result in Reg. PG5
0x1005E	Multiply	Operand in PG_6	Operand in F	Store result in Reg. PG6
0x10060	Multiply	Operand in PG_7	Operand in G	Store result in Reg. PG7
0x10062	Multiply	Operand in PG_8	Operand in H	Store result in Reg. PG8
0x10064	Multiply	Operand in PG_9	Operand in I	Store result in Reg. PG9
0x10066	Add	Operand in PG_1	Operand in PG_2	Store result in Reg. J
0x10068	Add	Operand in PG_3	Operand in PG_4	Store result in Reg. K
0x1006A	Add	Operand in PG_5	Operand in PG_6	Store result in Reg. L
0x1006C	Add	Operand in PG_7	Operand in PG_8	Store result in Reg. M
0x1006E	Add	Operand in J	Operand in K	Store result in Reg. N
0x10070	Add	Operand in L	Operand in M	Store result in Reg. O
0x10072	Add	Operand in N	Operand in O	Store result in Reg. P
0x10074	Add	Operand in P	Operand in PG_9	Store result in Reg. YG
0x10076	Store	Operand in YG		Result in Mem[X] + 19
0x10078	Load	PB_1 in Mem[X] + 20		Store result in Reg. PB1
0x1007A	Load	PB_2 in Mem[X] + 21		Store result in Reg. PB2
0x1007C	Load	PB_3 in Mem[X] + 22		Store result in Reg. PB3
0x1007E	Load	PB_4 in Mem[X] + 23		Store result in Reg. PB4
0x10080	Load	PB_5 in Mem[X] + 24		Store result in Reg. PB5
0x10082	Load	PB_6 in Mem[X] + 25		Store result in Reg. PB6
0x10084	Load	PB_7 in Mem[X] + 26		Store result in Reg. PB7
0x10086	Load	PB_8 in Mem[X] + 27		Store result in Reg. PB8
0x10088	Load	PB_9 in Mem[X] + 28		Store result in Reg. PB9
0x1008A	Multiply	Operand in PB_1	Operand in A	Store result in Reg. PB1

0x1008C	Multiply	Operand in PB_2	Operand in B	Store result in Reg. PB2
0x1008E	Multiply	Operand in PB_3	Operand in C	Store result in Reg. PB3
0x10090	Multiply	Operand in PB_4	Operand in D	Store result in Reg. PB4
0x10092	Multiply	Operand in PB_5	Operand in E	Store result in Reg. PB5
0x10094	Multiply	Operand in PB_6	Operand in F	Store result in Reg. PB6
0x10096	Multiply	Operand in PB_7	Operand in G	Store result in Reg. PB7
0x10098	Multiply	Operand in PB_8	Operand in H	Store result in Reg. PB8
0x1009A	Multiply	Operand in PB_9	Operand in I	Store result in Reg. PB9
0x1009C	Add	Operand in PB_1	Operand in PB_2	Store result in Reg. J
0x1009E	Add	Operand in PB_3	Operand in PB_4	Store result in Reg. K
0x100A0	Add	Operand in PB_5	Operand in PB_6	Store result in Reg. L
0x100A2	Add	Operand in PB_7	Operand in PB_8	Store result in Reg. M
0x100A4	Add	Operand in J	Operand in K	Store result in Reg. N
0x100A6	Add	Operand in L	Operand in M	Store result in Reg. O
0x100A8	Add	Operand in N	Operand in O	Store result in Reg. P
0x100AA	Add	Operand in P	Operand in PB_9	Store result in Reg. YB
0x100AC	Store	Operand in YB		Result in Mem[X] + 29
0x100AE	Add	Operand in X	Constant = 30	Store Result in Reg. X
0x100B0	GOTO	Address 0x10014	If X < 8294400	Else GOTO Next

3 Introduction

4K resolution, which is officially referred to as UHD (Ultra-high definition), offers at least 4 times as many pixels as regular HDTV. This leads to greater image clarity and more varied and realistic colors at higher frame rates. A 4K display has a resolution of 3840 pixels (horizontally) $\times$ 2160 pixels (vertically) where the horizontal resolution can go up to 4000 pixels.

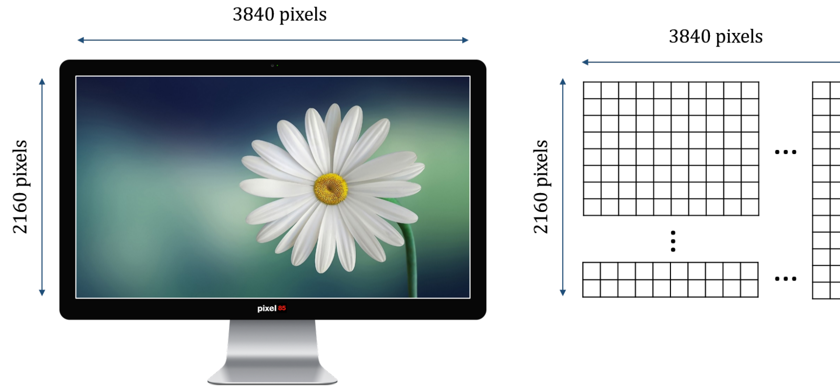


Figure 3.1: Dimensions of 4K resolution frame.

Therefore, total number of pixels in one frame of the 4K video is given by:

$$F_{pixels} = width\ in\ pixels \times height\ in\ pixels$$

$$F_{pixels} = 3840 \times 2160 = 8,294,400\ pixels/frame$$

If each pixel is composed of Red (R), Green (G), and Blue (B), where each channel pixel is 8 bits wide. The total bits for one frame is as follows:

$$F_{bits} = 8,294,400 \times 3 \times 8 = 199,065,600\ bits/frame$$

4 Theory

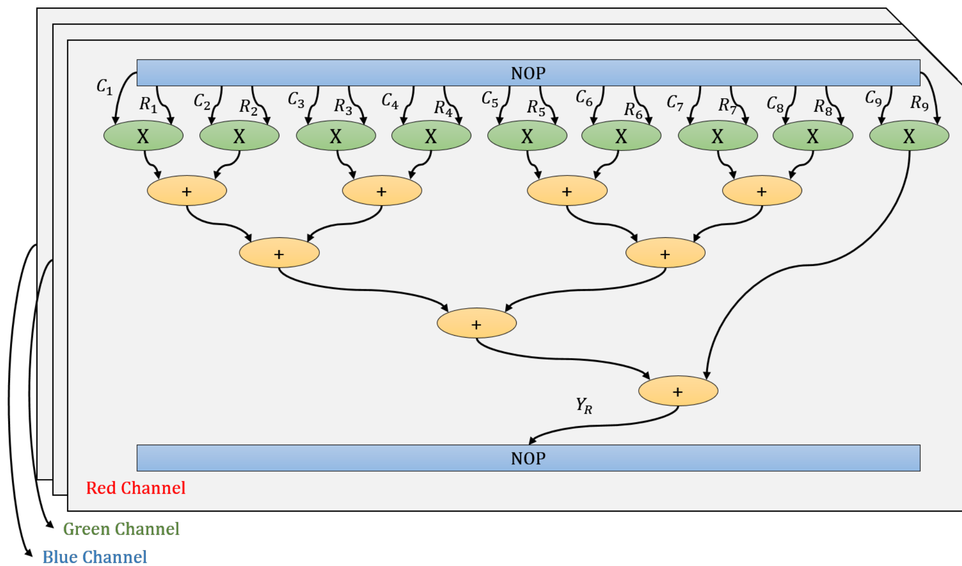


Figure 4.1: The sequencing graph for the red channel has been shown in detail. The sequencing graphs of blue and green channels have not been shown to avoid redundancy.

4.1 Sequencing Graph for FIR Filter

4.2 Sequencing Graph for RGB-Gray-scale converter

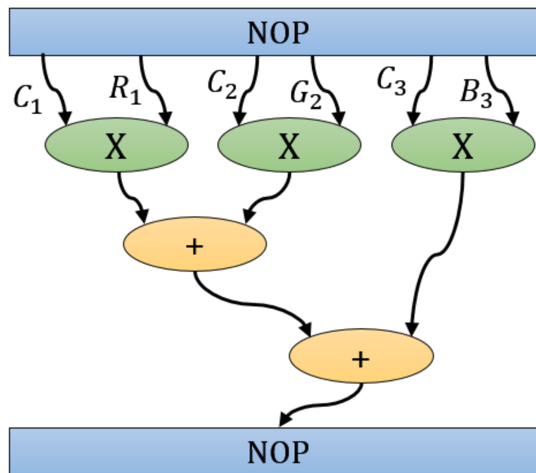


Figure 4.2: The sequencing graph for the RGB-to-Gray-scale converter can be seen in the above figure.

4.3 Implementation on CISC non-pipelined processor

4.3.1 Total number of clock cycles for one pixel calculation

No. of multiply instructions:

$$N_{Multiply} = 27$$

Clock cycles taken by multiply instructions:

$$\tau_{Multiply} = 27 \times 8\ c.c. = 216$$

No. of Add/Store/Clear...etc. instructions:

$$N_{Normal} = 56$$

Clock cycles taken by multiply instructions:

$$\tau_{Normal} = 56 \times 5 \; c.c. = 280$$

Total number of clock cycles for 1 pixel calculation:

$$\tau_{one\ pixel} = 216 + 280 = 496 \; c.c.$$

4.3.2 Total number of clock cycles for one frame calculation

No. of pixels in one frame:

$$F_{pixels} = 3840 \times 2160 = 8,294,400 \; pixels/frame$$

Time taken for calculation of all pixels in the frame:

$$\tau_{frame} = 8,294,400 \times 496 = 4,114,022,400 \; c.c.$$

4.3.3 Required Operating frequency for meeting 60fps specification

Clock signals required for calculating 60 frames in one second:

$$f_{required} = 4,114,022,400 \; c.c. \times 60 frames/sec \approx 246.84 \; GHz$$

The frequency obtained is unreasonably high. Hence, this implementation is not possible.

4.4 Implementation on RISC pipelined processor

4.4.1 Pipeline Diagram

Pipeline diagram given in appendices.

4.4.2 Total number of clock cycles for one pixel calculation

Total number of clock cycles for 1 pixel calculation:

$$\tau_{one\ pixel} = 57c.c. + 2 \times (57 + 2)c.c. + 5c.c. + 5c.c. = 185c.c.$$

4.4.3 Total number of clock cycles for one frame calculation

No. of pixels in one frame:

$$F_{pixels} = 3840 \times 2160 = 8,294,400 \; pixels/frame$$

Time taken for calculation of all pixels in the frame:

$$\tau_{frame} = 8,294,400 \times 185 \approx 1,534,464,000 \; c.c.$$

Frequency required to deliver 60fps:

$$f_{operating} = 60 fps \times 1,534,464,000 \; c.c. \approx 92.07 GHz$$

The value obtained suggests that there will be an extraordinary amount of power expenditure at this operating frequency.

4.5 Implementation on statically configured FPGA

Mode Name	**Total Logic Cells**
Noise Reduction Mode	58,150
Gray-scale Mode	49,895
Base Mode	17,503
Total Logic to fit	125,548

Therefore, the best lowest cost FPGA that fits our design is Artix-7 XC7A200T.

4.5.1 Frequency required on FPGA for non-pipelined implementation

Implementation without data division
Clock cycles required to calculate 1 pixel:

$$\tau_{pixel} = \tau_{multiplication} + \tau_{addition\ stage\ 1} + \tau_{addition\ stage\ 2} = 2c.c. + 2c.c. + 2c.c. = 6c.c.$$

Clock cycles required to calculate 1 frame:

$$\tau_{frame} = \tau_{pixel} \times 8,294,400\ pixels/frame = 49,766,400c.c.$$

$$f_{operating} = 60fps \times 49,766,400c.c. = 2,985,984,000Hz \approx 2.99GHz$$

However, if we divide the 4K resolution into 4 large chunks where each chunk is processed by separate hardware, we will only need 1/4 of $f_{operating}$.
Implementation with data division
Clock cycles required to calculate 1 pixel:

$$\tau_{pixel} = \tau_{multiplication} + \tau_{addition\ stage\ 1} + \tau_{addition\ stage\ 2} = 2c.c. + 2c.c. + 2c.c. = 6c.c.$$

Clock cycles required to calculate 1 frame portion:

$$\tau_{frame} = \tau_{pixel} \times 2,073,600\ pixels/frame\ portion = 12,441,600c.c.$$

$$f_{operating} = 60fps \times 12,441,600c.c. = 746,496,000Hz \approx 746.50MHz$$

4.6 Frequency required on FPGA for pipelined implementation

Implementation without data division
Clock cycles required to calculate 1 pixel:

$$\tau_{pixel} = \tau_{multiplication} + \tau_{addition\ stage\ 1} + \tau_{addition\ stage\ 2} = 2c.c. + 2c.c. + 2c.c. = 6c.c.$$

From the scheduling diagram, we can see that:

$$\tau_{latency} = 6c.c. \tag{4.1}$$

$$\tau_{cycle\ time} = 2c.c. \tag{4.2}$$

Clock cycles required to calculate 1 frame:

$$f_{operating} = \tau_{latency} + \tau_{cycle\ time} \times (8,294,400 - 1)\ pixels/frame = 6 + 2 \times (8,294,399) \approx 995.33MHz$$

However, if we divide the 4K resolution into 4 large chunks where each chunk is processed by separate hardware, we will only need 1/4 of $f_{operating}$.
Implementation with data division
Clock cycles required to calculate 1 pixel:

$$\tau_{pixel} = \tau_{multiplication} + \tau_{addition\ stage\ 1} + \tau_{addition\ stage\ 2} = 2c.c. + 2c.c. + 2c.c. = 6c.c.$$

From the scheduling diagram, we can see that:

$$\tau_{latency} = 6c.c. \tag{4.3}$$

$$\tau_{cycle\ time} = 2c.c. \tag{4.4}$$

Clock cycles required to calculate 1 frame portion:

$$\tau_{frame} = \tau_{latency} + \tau_{cycle\ time} \times (2,073,600 - 1)\ pixels/frame\ portion = 6 + 2 \times (2,073,599) = 4,147,204c.c.$$

$$f_{operating} = 60fps \times 4,147,204c.c. = 248,832,240Hz \approx 248.83MHz$$

4.7 Implementation on dynamically configured FPGA (RCS)

Mode Name	Total Logic Cells
Noise Reduction Mode	$58,150 + 17,503 = 75,653$
Gray-scale Mode	$49,895 + 17,503 = 67,398$
Largest Logic to fit	$75,653$

Therefore, the best lowest cost FPGA that fits our design is Artix-7 XC7A100T.

5 System Design

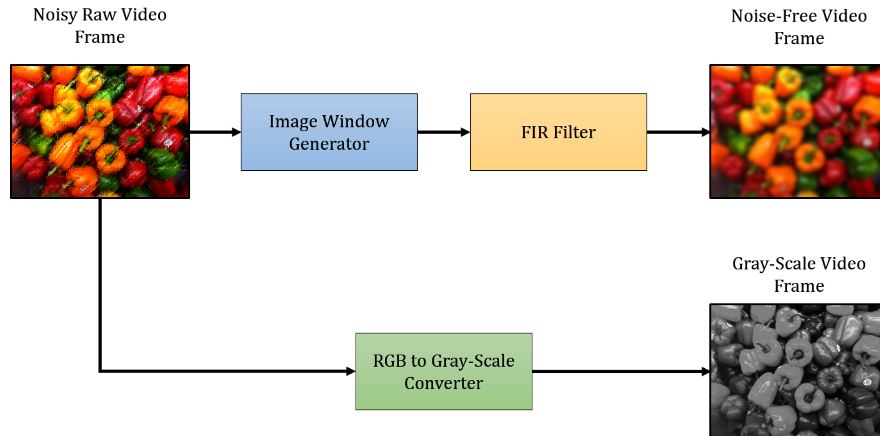


Figure 5.1: Block diagram of the proposed system with 3 major components.

A noisy raw video frame is sent to the Image Window Generator which divides the frame into specific rectangular sub-regions and allows local processing of video data within these target areas. The FIR filter designed using the window method suppresses the noise and produces a noise free video frame. The raw frame is also passed through a RGB-to-Grey scale converter which converts it to gray-scale.

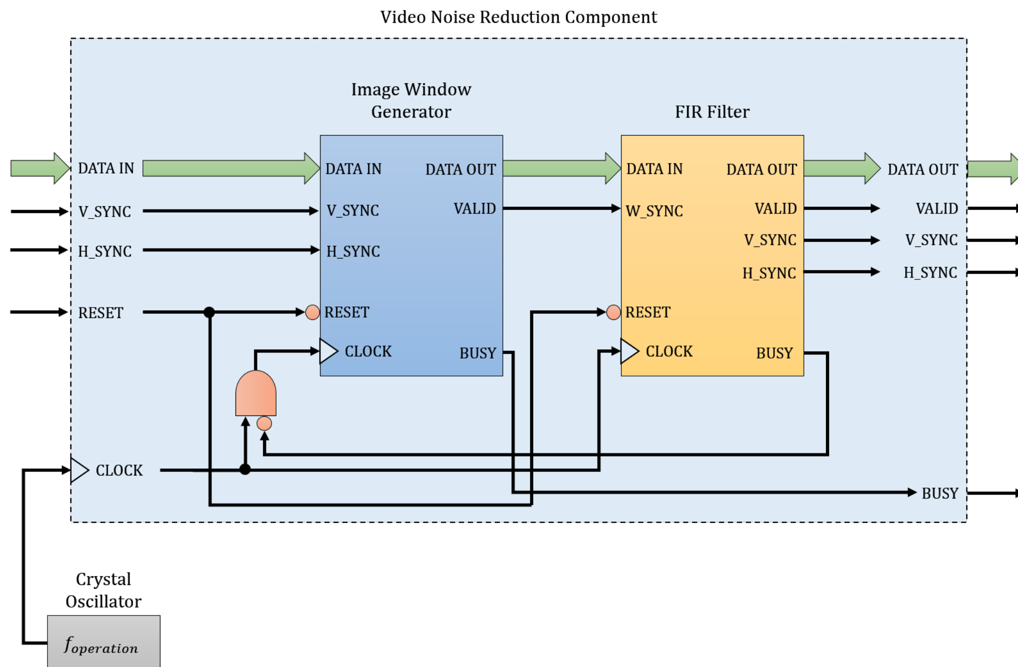


Figure 5.2: sub-components connection in video noise reduction component.

The components of the video noise reduction component are depicted in Figure 5.2. The clock is used as a strobe for R, G and B input values. The image window generator receives the input data. $V_{SYNC.}$ and $H_{SYNC.}$ on the Window Generator and FIR filter symbols are used as initiation signals.After data for the entire frame is sent, $V_{SYNC.}$ pulses to denote the end of frame. $H_{SYNC.}$ denotes the end of row when data for the entire row is sent. *BUSY* is used by the following components to ensure proper processing of data. *VALID* checks and validates data being passed on to the following sub-component. *RESET* on both blocks functions as an asynchronous termination signal.

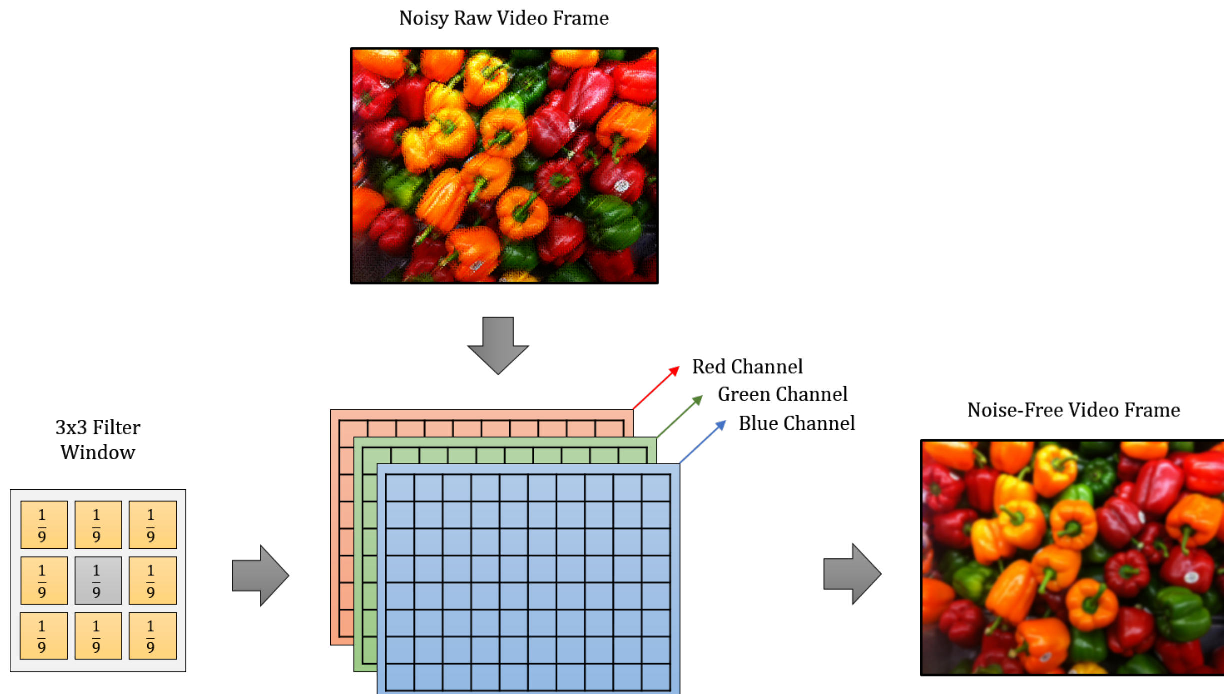


Figure 5.3: Data handling in filter.

The noisy frame is decomposed into R, G and B channels. After passing the raw frame through the window generator, local processing within the target areas involves calculations on the surrounding pixels to calculate the optimal value for the center pixel. The channels are merged for producing the noise free image.
If we are looking for a performance of 60 fps at 4K resolution:

$$d_{speed} = \frac{199,065,600 \times 60 \; bits/sec}{8 \times 1024 \times 1024} = 1423.83 \; MB/sec$$

$$d_{speed} \approx 1.42 \; GB/sec$$

6 Implementation on CPU

The following MATLAB code was written and the total time for filtering randomly generated Gaussian noise was recorded.

```
% Read 4K image
imageWithNoise = imread('imageWithNoise.jpg');
% Create a 3x3 low pass FIR filter
lpf = [1/9 1/9 1/9; 1/9 1/9 1/9; 1/9 1/9 1/9];
% Pass the image through the filter
output = imfilter(imageWithNoise, lpf
```

Profile Summary

Generated 18-Dec-2015 17:37:30 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
FIRFilter	1	0.512 s	0.006 s	
imread	1	0.447 s	0.003 s	
imagesci\private\readjpg	1	0.434 s	0.000 s	
imagesci\private\rjpg8c (MEX-file)	1	0.433 s	0.433 s	
imfilter	1	0.059 s	0.004 s	
imfilter>filterPartOrWhole	1	0.027 s	0.000 s	
images\private\imfilter_mex (MEX-file)	1	0.027 s	0.027 s	
padarray	1	0.026 s	0.000 s	
padarray>ConstantPad	1	0.024 s	0.023 s	
imagesci\private\imftype	1	0.006 s	0.002 s	
imread>parse_inputs	1	0.004 s	0.004 s	
imformats	1	0.003 s	0.000 s	
imformats>find_in_registry	1	0.003 s	0.003 s	
padarray>ParseInputs	1	0.002 s	0.001 s	
imagesci\private\isjpg	1	0.001 s	0.001 s	
imagesci\private\jpeg_depth (MEX-file)	1	0.001 s	0.001 s	
imfilter>computeSizes	1	0.001 s	0.001 s	
validatestring>checkInputs	1	0.001 s	0.001 s	
validatestring	1	0.001 s	0.000 s	
images\private\mkconstarray	1	0.001 s	0.000 s	
repmat	1	0.001 s	0.001 s	
imfilter>isSeparable	1	0.001 s	0.001 s	
imfilter>parse_inputs	1	0 s	0.000 s	
iscellstr	1	0 s	0.000 s	
validatestring>checkString	1	0 s	0.000 s	

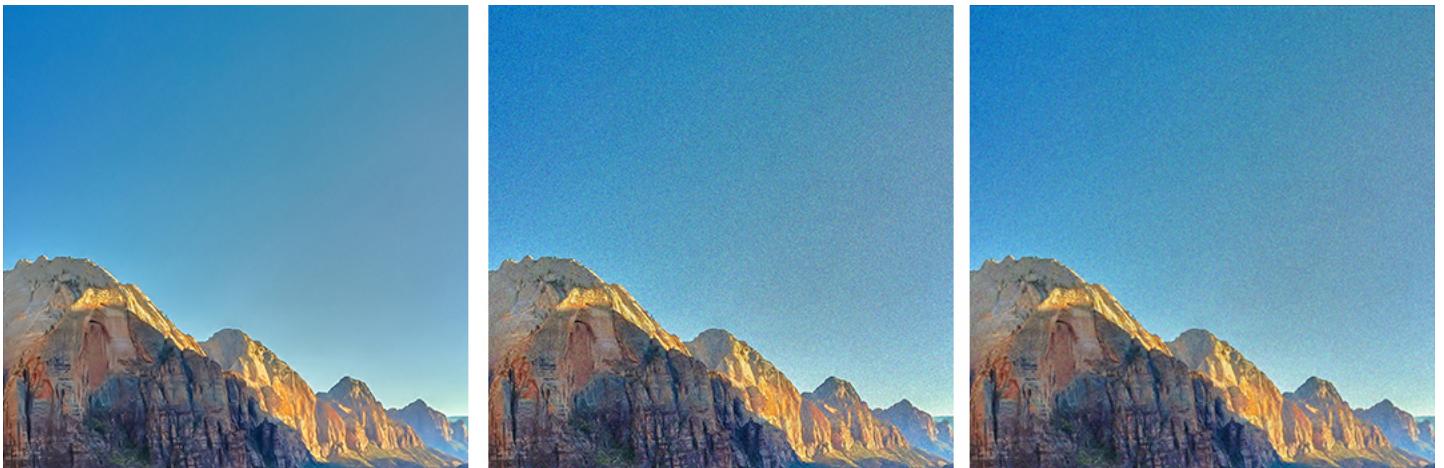


Figure 6.1: Left: Clean image without noise. Center: Image with noise. Right: Filtered image.

It takes 2.015 seconds to process one 4K resolution frame in CPU implementation. In other words, the frame rate is $\approx 0.5 fps$. Which is 12 times slower than required performance.

7 Implementation on FPGA

7.1 Non-pipelined block diagram of FIR Filter

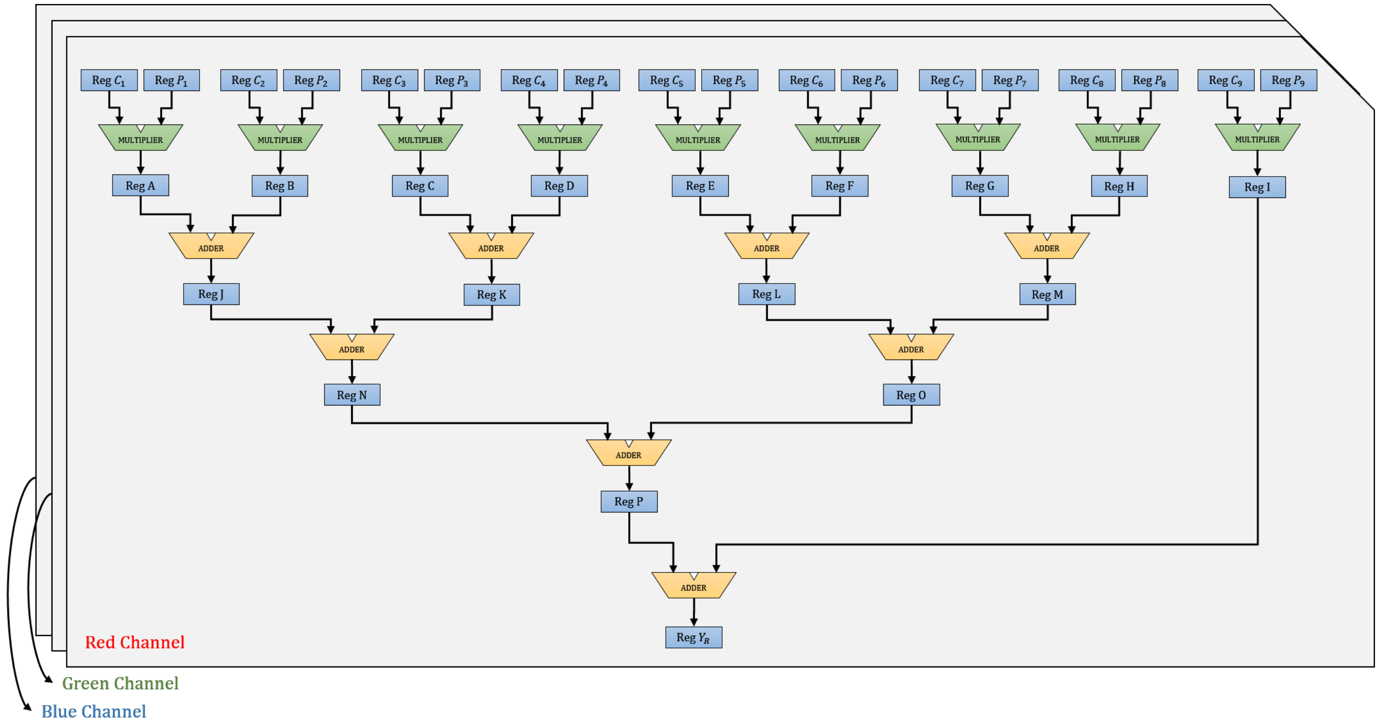


Figure 7.1: The figure above shows the detailed non-pipelined block diagram for the Red channel of the FIR filter.

There are 9 pixels (P_1,.....P_9) and each pixel is multiplied by its corresponding constant (C_1, ..., C_9) through a multiplier. The results of each of the multiplications are stored in registers (Reg A,....., Reg I). Pairs of register values (Registers A and B, C and D, E and F and G and H) are then passed through adders. The resultant values are then stored in registers (Reg J,....., Reg M). The values in these registers are then fed to adders. The values generated from addition are then stored in registers N and O. The values in registers N and O are again passed through an adder which produces a value that gets stored in register P. Finally, values in registers P and I are added together and the final resultant value for the red channel gets stored in register Y_R. The multiplication step takes 2 clock cycles and each of the addition steps takes 1 clock cycle. The green and blue channels go through the same process and processes for all 3 channels occur parallel to each other.

7.2 Non-pipelined block diagram of RGB to Gray-scale converter

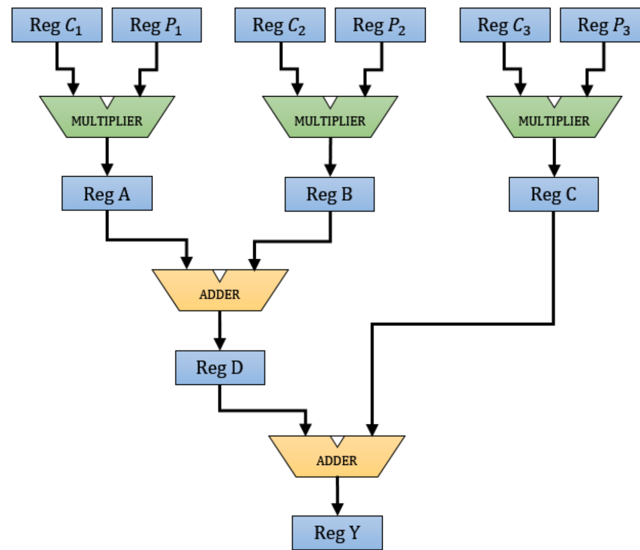


Figure 7.2: Non-pipelined block diagram for the RGB-to-Grayscale converter. It can be explained in the same way as Fig. 7.1

7.3 PIPELINED BLOCK DIAGRAM OF FIR FILTER

The adders in the hardware block diagram shown in Fig. 7.1 are not being utilized completely. When the bit values are passing through the hardware units during the calculation process, the adders which have performed their function remain unused. The pipelined block diagram solves this problem by using the technique of optimal pipelining. The pipelined solution works in the following way: Each of the 9 pixels (P_1,...., P_9) is multiplied with its corresponding constant (C_1, ..., C_9). The resulting values are stored in Registers A to I. Each of the registers is now engaged. The register values are passed through multiplexers which are again passed through adders. The resulting values which are stored in Registers J, K an L. Each of these register values is passed through demultiplexers which either pass the resulting output to the following multiplexers or send the values of registers J, K and L back to the previous multiplexers. Initially, the results from J, K and L are sent back to the multiplexers, go to the adder and given to registers J, K and L. Now, the resulting values are passed from the demultiplexers to the following multiplexers. The selected values then pass through an adder which stores the value in register M. The value in register M passes through the demultiplexer from where it first goes back to the preceding multiplexer. The new values are again passed through the adder and stored in register M. The value from register M is finally sent to register Y_R through the demultiplexer. The multiplication step in this scenario takes 2 clock cycles and the addition step takes 3 clock cycles. The green and blue channels go through the same process and processes for all 3 channels occur parallel to each other.

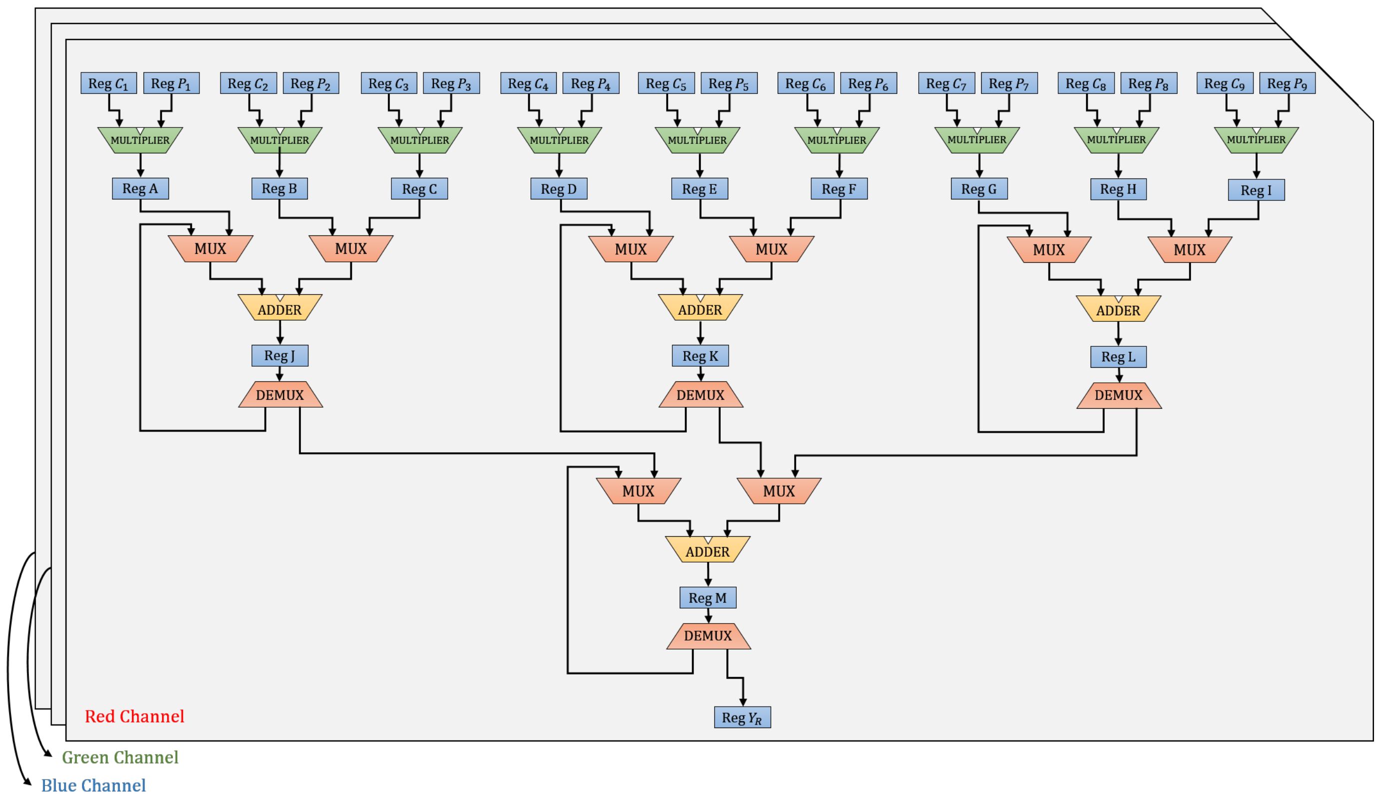


Figure 7.3: Pipelined block diagram for the Red channel of the FIR filter.

7.4 Pipelined block diagram of RGB to Gray-scale converter

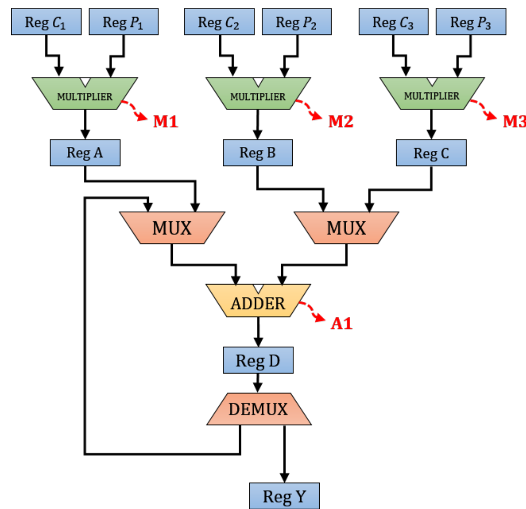


Figure 7.4: Pipelined block diagram for the RGB-to-Grayscale converter. It can be explained in the same way as Fig. 7.3.

7.5 Optimal Timing diagram of FIR Filter

From this diagram, it can be seen that the very first resulting values (Y_R, Y_B and Y_G) are released after a period of 6 clock cycles. Following that, the resultant values are obtained after every 2 clock cycles. Hence, the latency in this case is 6 clock cycles and the cycle time is 2 clock cycles.

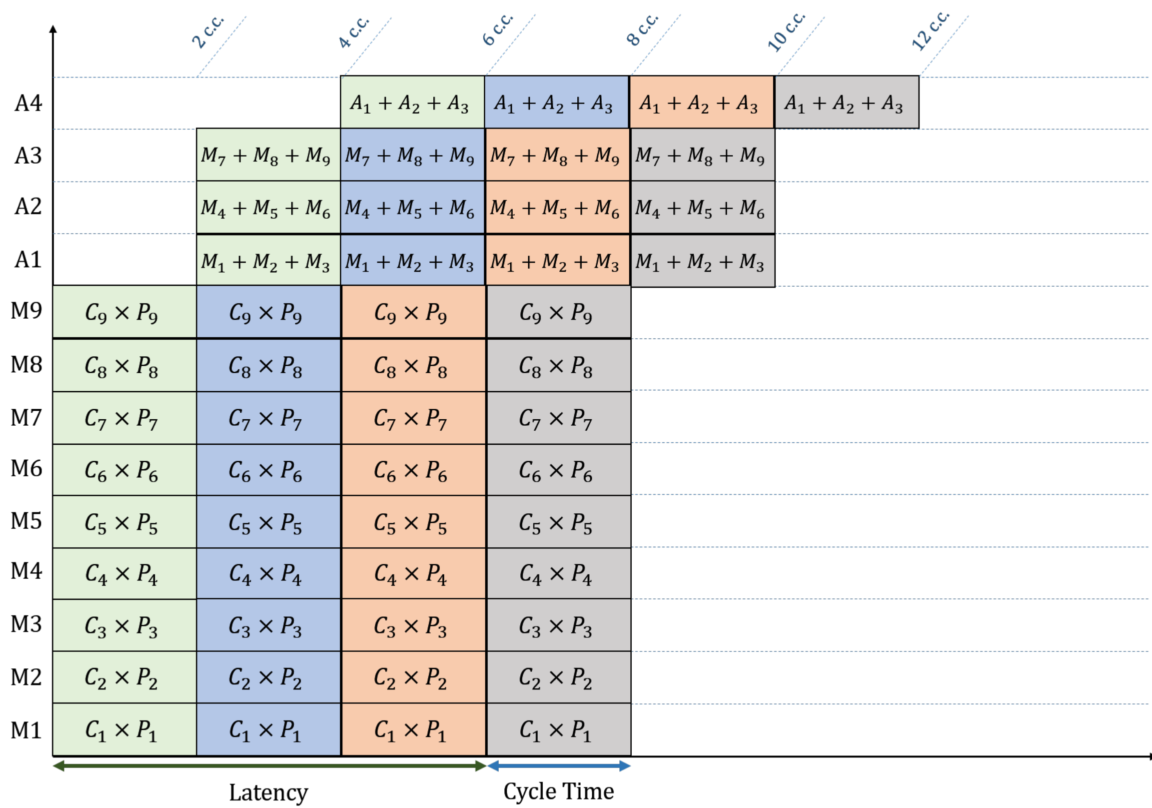


Figure 7.5: The optimal timing diagram can be seen in Fig. 7.5 and corresponds to the pipe-lined hardware system of Fig. 7.3.

7.6 Non-optimal Timing diagram of RGB to Gray-scale converter

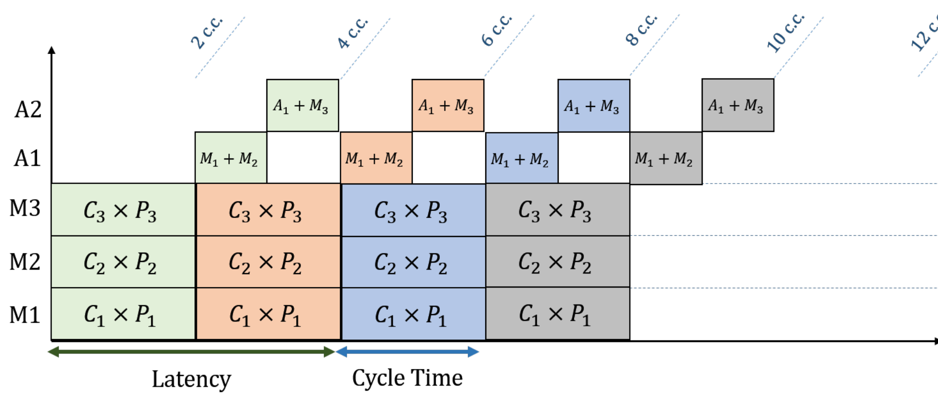


Figure 7.6: Non-optimal timing diagram of RGB to Gray-scale converter. Hardware resources are not being used optimally.

7.7 Optimal Timing diagram of RGB to Gray-scale converter

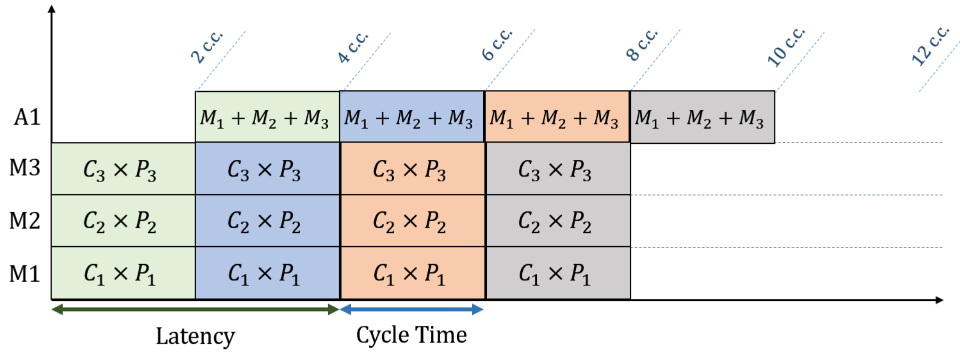


Figure 7.7: Optimal timing diagram of RGB to Gray-scale converter. Utilization of hardware resources is now better. Latency= 4 c.c. and Cycle time= 2 c.c..

8 Design

8.0.1 FIR Component Bandwidth

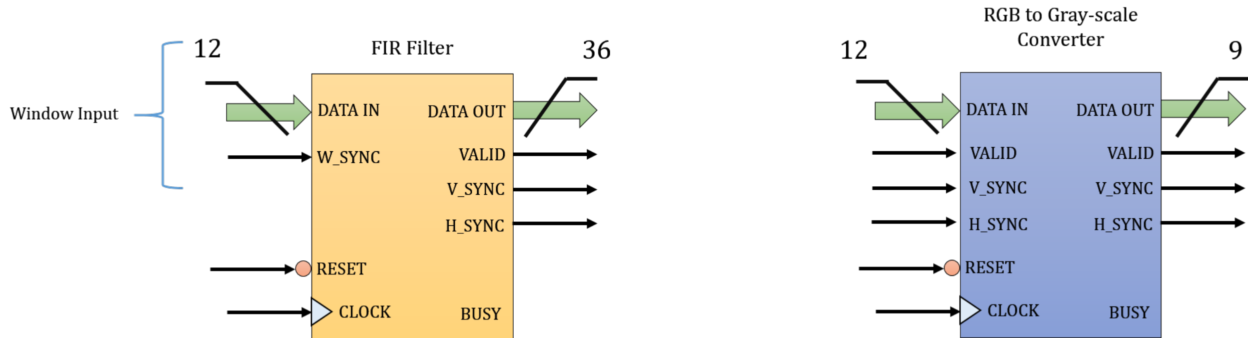
$$Input\ Bandwidth = \frac{3Bytes \times 995.33\ MHz}{2c.c. \times 1024 \times 1024} \approx 1423.831\ MB/sec \tag{8.1}$$

$$Output\ Bandwidth = \frac{72Bits \times 995.33\ MHz}{2c.c. \times 8 \times 1024 \times 1024} \approx 4271.49MB/sec \tag{8.2}$$

8.0.2 RGB to Gray-scale Component Bandwidth

$$Input\ Bandwidth = \frac{3Bytes \times 995.33\ MHz}{2c.c. \times 1024 \times 1024} \approx 1423.831\ MB/sec \tag{8.3}$$

$$Output\ Bandwidth = \frac{9Bits \times 995.33\ MHz}{2c.c. \times 8 \times 1024 \times 1024} \approx 533.94MB/sec \tag{8.4}$$



9 Comparative Analysis

The two modes of the application were compared with each other and implemented using two different methods. The first method was to compute the whole frame by using a single hardware processor. The second method involved dividing the screen into 4 equal regions and a hardware block was assigned to each of these regions running parallel to each other. This reduced the operating frequency of the hardware to 1/4th of the original value. Since FPGAs operate at a much slower frequency than ASICs, the operating frequencies that were found during analysis were too high to implement using standard FPGAs.

In the table below, comparison of non-shared and shared data workloads is provided. If a 4K resolution image is divided into 4 equal regions, there will be 12 hardware components running in parallel to calculate frame pixels.

However, it was noticed that this addition of hardware components will require extra chip area but will reduce power consumption (due to the reduced operational frequency) and increase the hardware lifetime.

Without screen splitting, it was nearly impossible to implement 4K filtering using standard FPGAs but it is possible through ASIC as shown in the table. On the other hand, by splitting the screen, achievable operation frequencies were seen and it was possible to implement.
PCR for 100 devices is highest for Reconfigurable computing implementation. For 1000 and 10,000 devices , RCS again produces the best PCR.

9.1 Speedup Calculation

Speed-up of RCS implementation compared to RISC:

$$S_{RCS-RISC} = \frac{92.06784 \times 10^9}{248,832,240} = 370\, times \tag{9.1}$$

Speed-up of RCS implementation compared to CISC:

$$S_{RCS-CISC} = \frac{246.841344 \times 10^9}{248,832,240} = 992\, times \tag{9.2}$$

Speed-up of RISC implementation compared to CISC:

$$S_{RISC-CISC} = \frac{246.841344 \times 10^9}{92.06784 \times 10^9} = 2.6812 \approx 2.7\, times \tag{9.3}$$

Implementation Type	Choosen Device	Screen Data Division	Units	Development Cost (per unit)	Device Cost (per unit)	Cost of Components	Performace (fps)	PCR (fps/$)	Operating Frequency	Possible
FPGA - Static	XC7A200T	No	100	$250	$269	$600	60	0.053619	995.33MHz	No
	XC7A200T	No	1000	$25	$229	$600	60	0.070258	995.33MHz	No
	XC7A200T	No	10,000	$2.50	$215	$600	60	0.073394	995.33MHz	No
	XC7A200T	Yes	100	$300	$269	$600	60	0.051326	248.83MHz	Yes
	XC7A200T	Yes	1000	$30	$229	$600	60	0.069849	248.83MHz	Yes
	XC7A200T	Yes	10,000	$3.00	$215	$600	60	0.073350	248.83MHz	Yes
FPGA - RCS	XC7A100T	No	100	$150	$251	$600	60	0.059940	995.33MHz	No
	XC7A100T	No	1000	$15	$213	$600	60	0.072464	995.33MHz	No
	XC7A100T	No	10,000	$1.50	$203	$600	60	0.074580	995.33MHz	No
	XC7A100T	Yes	100	$200	$251	$600	60	0.057088	248.83MHz	Yes
	XC7A100T	Yes	1000	$20	$213	$600	60	0.072029	248.83MHz	Yes
	XC7A100T	Yes	10,000	$2.00	$203	$600	60	0.074534	248.83MHz	Yes
ASIC	N/A	No	100	$20,000	$4,050	$600	60	0.002434	995.33MHz	Yes
	N/A	No	1000	$2,000	$1,050	$600	60	0.016438	995.33MHz	Yes
	N/A	No	10,000	$200	$250	$600	60	0.057143	995.33MHz	Yes
	N/A	Yes	100	$20,050	$4,050	$600	60	0.002429	248.83MHz	Yes
	N/A	Yes	1000	$2,005	$1,050	$600	60	0.016416	248.83MHz	Yes
	N/A	Yes	10,000	$200.50	$250	$600	60	0.057116	248.83MHz	Yes

Static FPGA Development Cost: $25,000
ASIC Development Cost: $2,000,000
RCS Development Cost: $15,000
Static FPGA Development Cost (With Splitting): $30,000
ASIC Development Cost (With Splitting): $2,005,000
RCS Development Cost (With Splitting): $20,000

10 Conclusion

Xilinx and Altera are pushing hard to squeeze as many logic blocks as possible onto the provided FPGA die area. The timing constraints required by 4K video processing and other DSP applications requires the system designer to exploit data parallelism and instruction parallelism to overcome these constraints and meet required specifications. Instruction parallelism was used to process complete pixel vector information of the frame.

The two modes of the application were compared with each other and implemented using two different methods. The first method was to compute the whole frame by using a single hardware processor. The second method involved dividing the screen into 4 equal regions and a hardware block was assigned to each of these regions running parallel to each other. This reduced the operating frequency of the hardware to 1/4th of the original value. Since FPGAs operate at a much slower frequency than ASICs, the operating frequencies that were found during analysis were too high to implement using standard FPGAs.

The FPGA price, configuration bits, frequency range, logic cells, etc. were gathered from renowned suppliers and data-sheets. Speed-up of 370 times was found comparing reconfigurable implementation to RISC implementation and 992 times when comparing reconfigurable implementation to CICS implementation. It was also found that it is not practical to use RISC or CISC processors like ARM for noise reduction of a 4K video stream.