FPGA Acceleration of Convolutional Neural Networks

Shiwen Tang (st3510), Yufei Qian (yq2352), YuFei Jin (yj2725), Gehui Liang(gl2800), Yi Wang(yw3903)

1. Introduction

Our project is motivated by the research paper titled "Systimator: A Design Space Exploration Methodology for Systolic Array based CNNs Acceleration on the FPGA-based Edge Nodes [1]." The paper inspired us to develop a real-time system for handwriting number recognition using a Systolic Array-Based LeNet-1 Convolutional Neural Network (CNN) Accelerator on a DE-1 SoC FPGA + HPS (Hard Processor System). LeNet-1, which is a variant of the LeNet CNN, is a highly effective algorithm for recognizing patterns in handwritten data. However, its computational demands can be a challenge when running on traditional computing systems.

To address this, we harness the parallel processing capabilities of the FPGA, alongside the processing power of the HPS, to significantly reduce processing time. The result is an efficient and effective system for recognizing handwritten digits. This system has a wide range of potential real-world applications, including automated data entry, digitization of handwritten documents, and the creation of interactive educational tools.

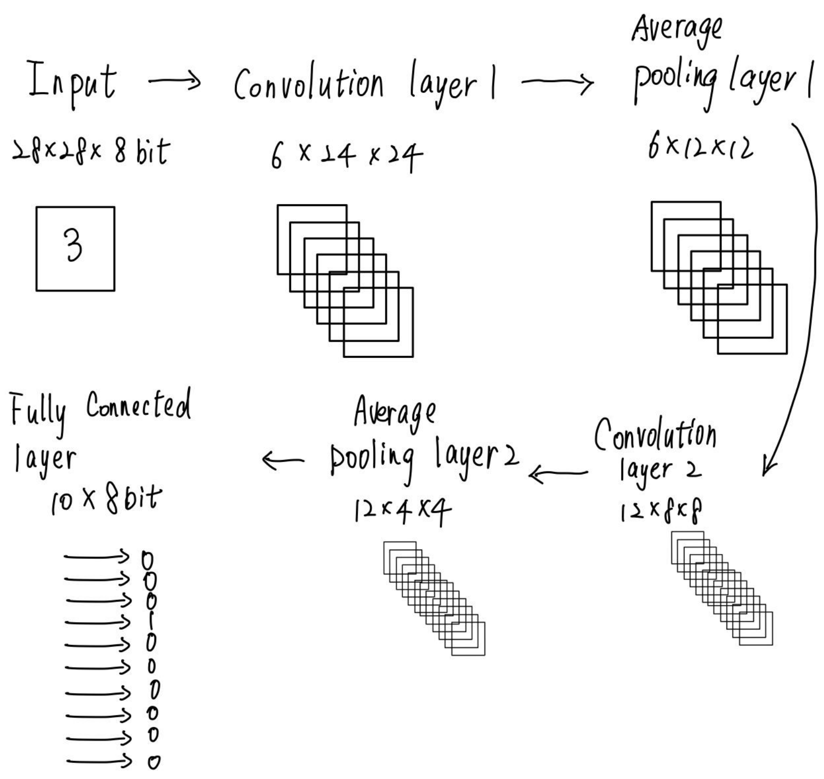
Our proposal aims to develop a U-net accelerator that employs a Max Pooling layer to reduce the input data size, followed by an Upsampling layer that increases the output size to match that of the input. Despite successful synthesis of the U-net accelerator, the number of required Programmable Logic Elements exceeded the DE1-SoC's capacity. To address this, we redesigned the accelerator's architecture by removing the Upsampling layers and reducing the input data width from 32 to 8 bits, allowing for successful deployment on the DE1-SoC.

In general, in this project, we first use a camera to capture a real-time image. Then this image will be processed to a suitable size and sent to the CNN model to handle and do the recognition. After that, the recognition result will be sent to the 7 segment display and shown to the user in the FPGA board.

2. LeNet

LeNet-1 is a simple convolutional neural network architecture introduced by Yann LeCun in 1988. It was designed for the purpose of recognizing hand-written digits and was one of the earliest applications of convolutional networks, helping pave the way for the advancements in deep learning.

The LeNet-1 architecture comprises of several layers including convolutional layers, subsampling (or average pooling) layers, and a fully connected layer at the end, all designed to reduce the complexity of the input and extract meaningful features.



2.1. LeNet Parameters

The training dataset is MNIST. The input of the data is a 28*28 image of a written number. We use the tensorflow to train the LeNet model and get the value of weights and bias. These parameters are finally converted to a 8-bit binary representation.

2.2. LeNet Data Flow

The data flow in LeNet-1 is as follows:

1. **Input Layer**: The LeNet-1 network receives a 28x28 grayscale image as input. Each pixel in this image is represented by a numerical value that indicates its intensity on a grayscale.
2. **Convolutional Layer 1**: The first layer is a convolutional layer, which uses 6 filters of size 5x5. Each of these filters is applied to the input image, and they effectively scan the image in a sliding window fashion, identifying various low-level features like edges, lines, gradients, etc. The result is 6 feature maps, each of size 24x24. Each feature map is essentially a transformed representation of the original image, with the filter's specific features highlighted. The convolution operation introduces local receptive fields, spatial parameter sharing and translation invariance.
3. **Activation Function**: After the convolutional layer, an activation function is applied elementwise. This is usually a non-linear function, which in this case is

the ReLU (Rectified Linear Unit) function. The purpose of the activation function is to introduce non-linearity into the network, which allows it to learn more complex patterns.

4. **Average Pooling Layer 1**: The next layer is an average pooling layer, which performs a down-sampling operation that reduces the dimensionality of the feature maps, resulting in size reduction to 12x12. This is done to decrease the computational complexity for the following layers, and to provide a form of translation invariance.
5. **Convolutional Layer 2**: The pooled feature maps are then passed through a second convolutional layer. This layer has 12 filters, each of size 5x5, and each filter is applied to all the feature maps from the previous layer. This results in 12 new feature maps of size 8x8. These new feature maps represent more complex features in the image, identified by combining the lower-level features from the previous layers.
6. **Activation Function**: Again, the ReLU activation function is applied to these feature maps.
7. **Average Pooling Layer 2**: Another average pooling operation is then applied, further reducing the dimensionality of the feature maps from 8x8 to 4x4.
8. **Fully Connected Layer**: The resulting 4x4x12 (192) feature maps are then flattened into a single vector and passed through a fully connected layer. This layer is a traditional Multi-Layer Perceptron that uses a softmax activation function in its output layer to return probabilities. The fully connected layer has 10 neurons, corresponding to the 10 possible classes (0-9) of the digit recognition problem. Each neuron in this layer outputs the probability that the digit in the image belongs to its class.

The overall flow of data through the LeNet-1 network thus involves a combination of convolutional layers, activation functions, pooling layers, and a fully connected layer at the end, all designed to progressively transform and reduce the complexity of the input image, while extracting and highlighting the features that are most relevant for the digit recognition task.

3. Hardware

3.1. Top Level Module

The top module (LeNet) takes in a clock and reset signal and produces an output called LeNet_Out. The module reads the input and weight values from external DRAM and uses a layer control block to manage each layer. It assigns the weights and inputs to the Conv3D module, which is then followed by an accumulation module. Next, it assigns the weights and inputs to a Full Connected module. Finally, the output is obtained from the LeNet_Out signal, with the last four digits indicating the identified numbers.

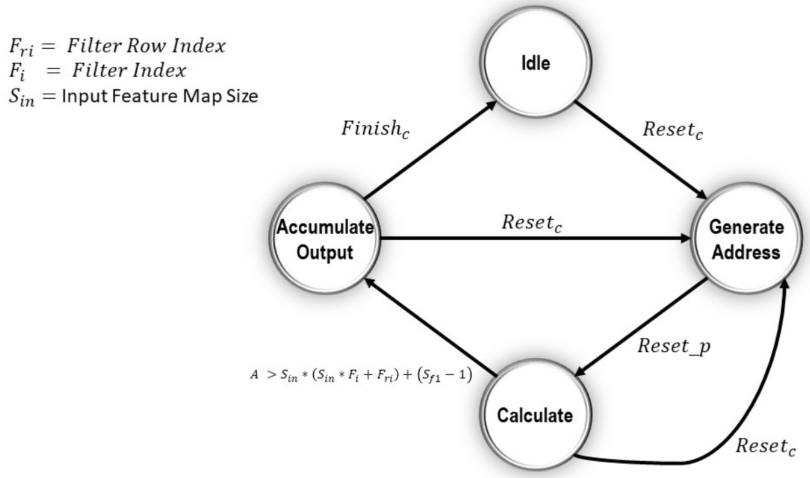
3.2 Conv3D.v

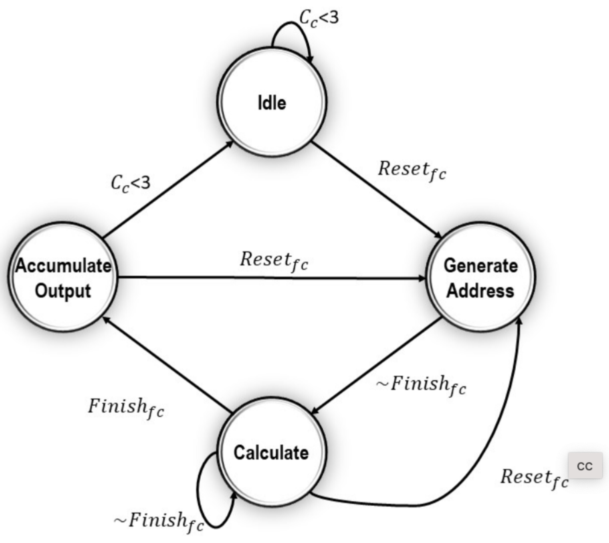
Conv3D module includes multiple blocks designed to handle different tasks such as address generation, convolution completion, filter indexing, reset control, and an instantiated PE array to efficiently process the data.

The inputs of this module include a clock and reset signal, input data 'a', a select signal 'S_in' for input data slices, the number of input data channels 'N_ch', five convolutional inputs 'I10, I11, I12, I13, I14', and thirty convolutional weight inputs. The outputs include five control signals 'Ai0-Ai4' used for selection and address generation and six output values 'P01-P06' used for activation and pooling.

The core of this convolution layer is a PE array comprising five rows and six columns. Each filter of the PE array shares the same input 'I10, I11, I12, I13, I14' but has different weight inputs, which generates a convolutional output signal. Ultimately, the module produces six output values for activation and pooling.

The figure below shows the structure of convolution layer.


F_{ri} = Filter Row Index
F_i = Filter Index
S_{in} = Input Feature Map Size
Idle
$Finish_c$
$Reset_c$
Accumulate Output
$Reset_c$
Generate Address
$Reset\_p$
$A > S_{in} * (S_{in} * F_i + F_{ri}) + (S_{f1} - 1)$
Calculate
$Reset_c$



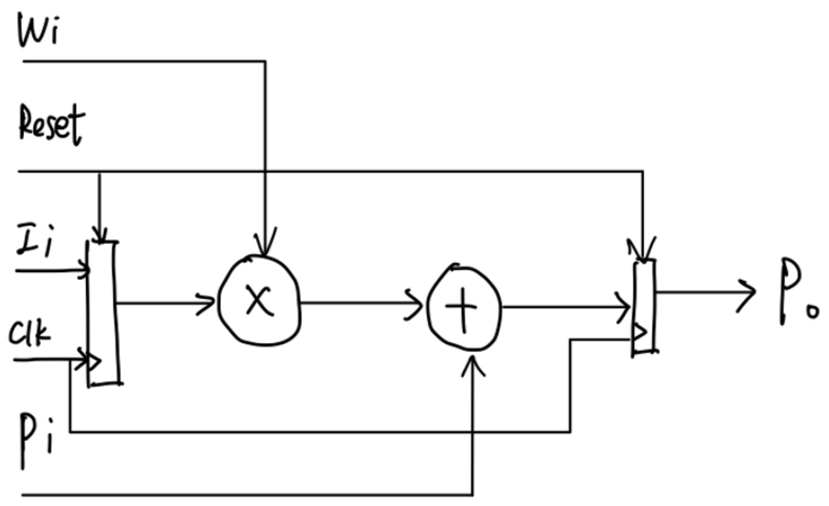
3.2.1 PE and PE_array Module

PE array module is one of the key submodules of the Conv3D module. The PE module takes in four input signals: `rst`, `clk`, `W`, and `Ii`. The `rst` signal is used to reset the module, `clk` is the clock signal, `W` is the weight value, and `Ii` is the input value. The module also has two output signals: `Po` and `Io`. `Po` represents the product of `W` and `Ii` added to a previous value `Pi`, and `Io` represents the input value `Ii`. It performs multiplication between the input data and weights, adds the result to the previous output, and produces the output sum and output data. The calculations occur on the positive edge of the clock signal when the reset signal is inactive.

The `always` block in the module is triggered on the rising edge of the clock signal. When the reset signal is asserted, the output `Io` is set to the input `Ii`, and `Po` is set to the sum of the input `Pi` and the product `P` calculated in the multiplication block.

This process is repeated multiple times to process all the rows of the PE. Once the partial sums through each column of the PE array are available at the output of its last row, they are fed into the accumulation block.

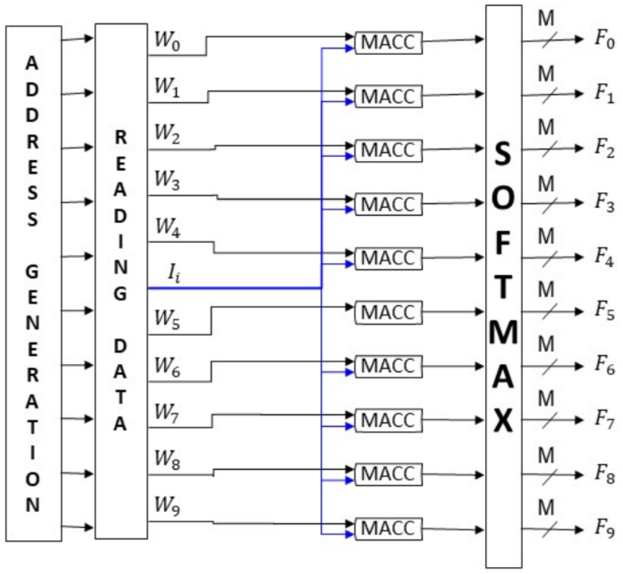
The figure below shows the structure of PE module.



3.3 Fully Connected (FC) Module

The input ports of FC module include the clock signal, reset signal, weight matrices, and the input image. The output ports include the output matrices and the output digit value. There are 10 Po signals, which are the output signals from each of the 10 MACC (Multiply-Accumulate) modules. These signals are then input to the softmax module to calculate the probability of each digit. There are 10 MACC modules in total, each with its own weight matrix, input, and output signals. The FC module also includes several control signals and registers to manage the operation of the MACC and softmax modules. The Fi and FCi signals are used to control the column and row index of the weight and input matrices, respectively. Finally, the output digit value is calculated using the 10 Ps signals, which represent the probability of each digit. The value of LeNet_Out is set to -1 if the finish signal is not high, indicating that the computation is not yet complete.

The figure below shows the structure of Fully Connected layer.

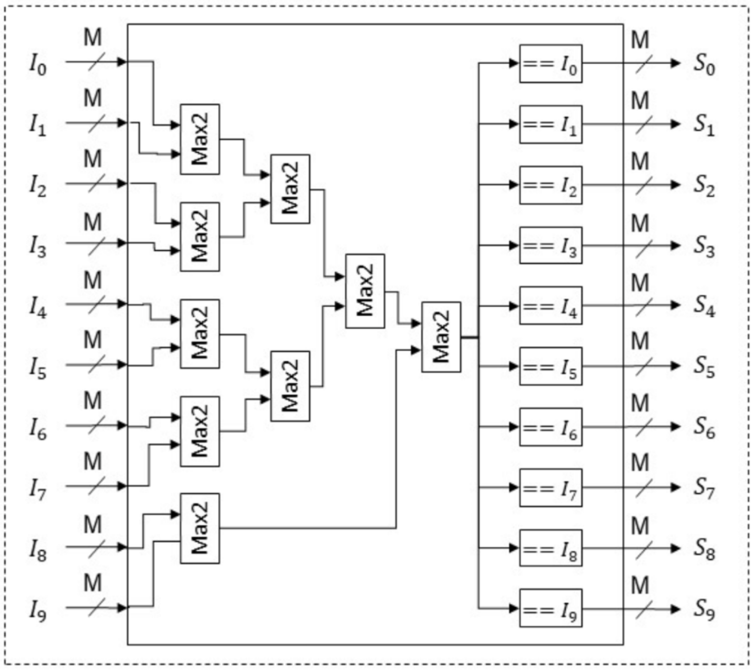


3.3.2 Softmax Module

Softmax module is the most significant submodule of The FC Module. It takes in 10 input values `A0` to `A9`. These input values are compared with each other to find the maximum value, which is then used to calculate the output signals `O0` to `O9`.

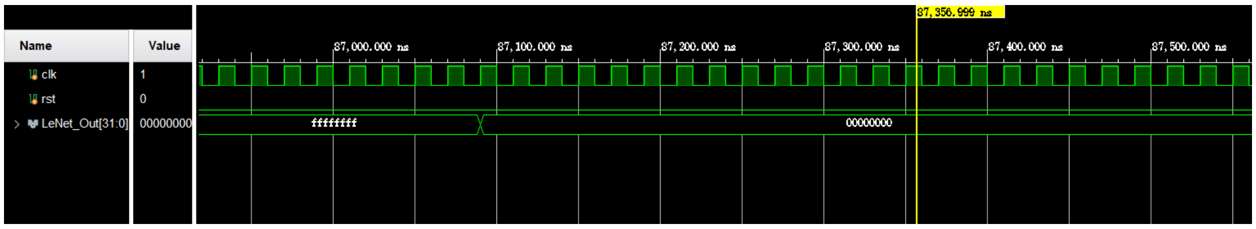
The parameter M determines the width of the input values. There are 9 `max2` sub-modules, each of which compares two inputs and outputs the maximum value. The outputs of these `max2` sub-modules are then used as inputs to other `max2` sub-modules in order to calculate the maximum value of all 10 input values. The output of the last `max2` sub-module is compared with each of the 10 input values, and the output signals `O0` to `O9` are set to 1 if the corresponding input value is equal to the maximum value, and 0 otherwise.

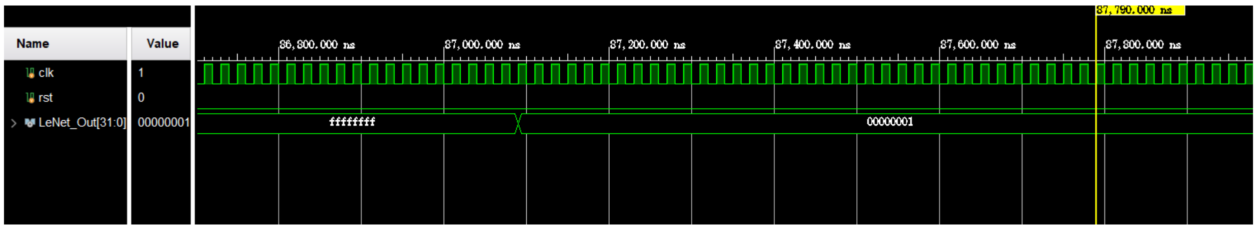
The figure below shows the structure of Softmax layer.

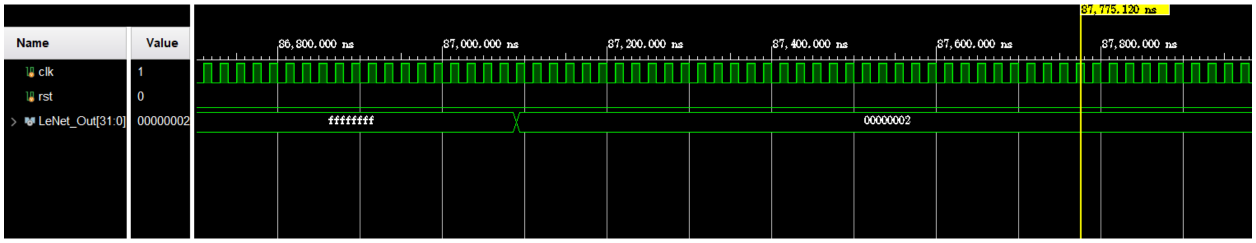


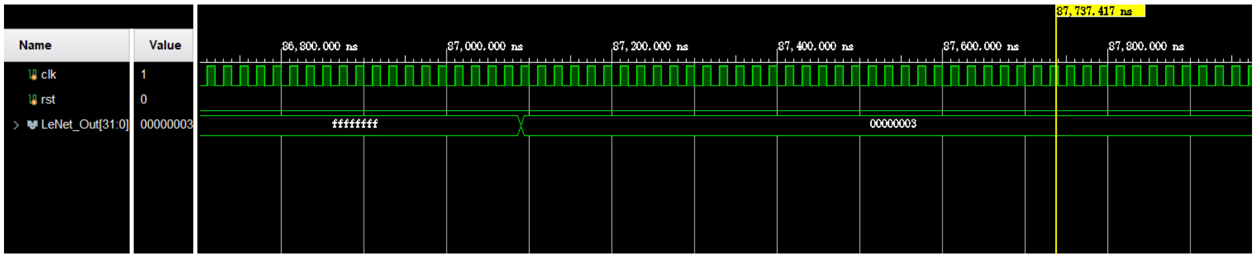
3.4 Simulation Results

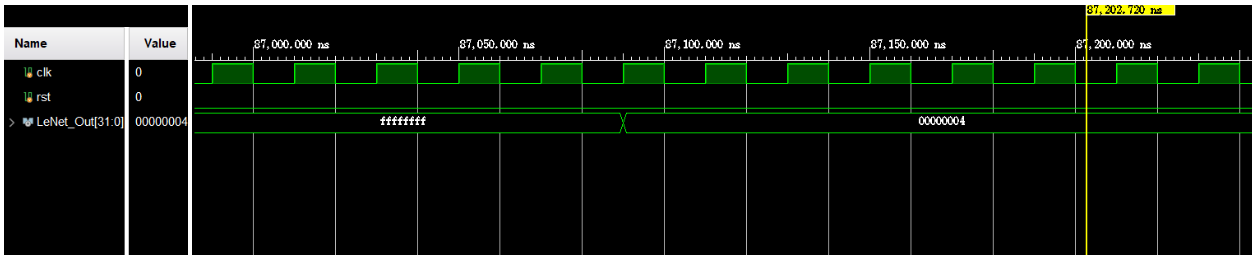
The simulation results presented below demonstrate LeNet's ability to recognize different input images. We tested the model's performance by feeding it with input images of numbers ranging from 0 to 9 and monitored the output signal LeNet_Out. Our results indicate that LeNet was able to correctly recognize numbers 0, 1, 2, 3, 4, 6, 7, and 9. However, it misclassified number 5 as 3 and number 8 as 1, resulting in an accuracy rate of approximately 80%.

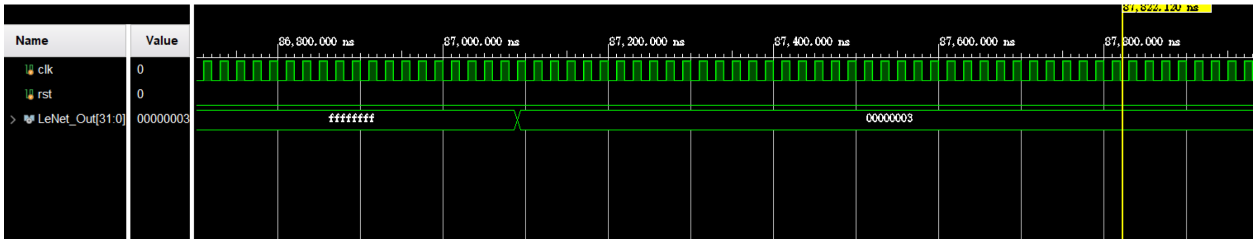


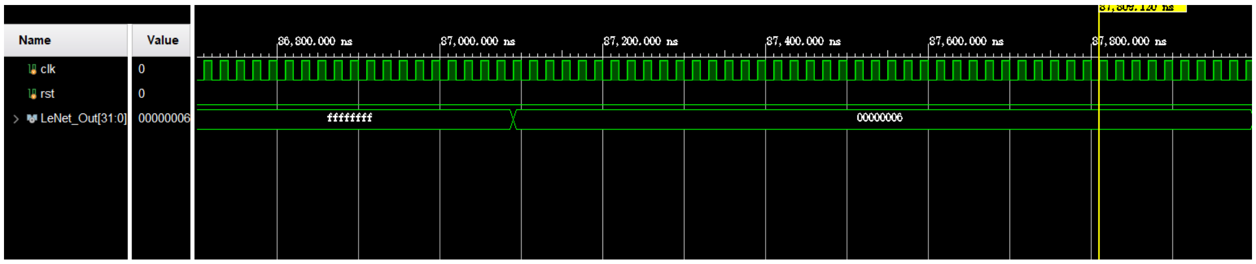


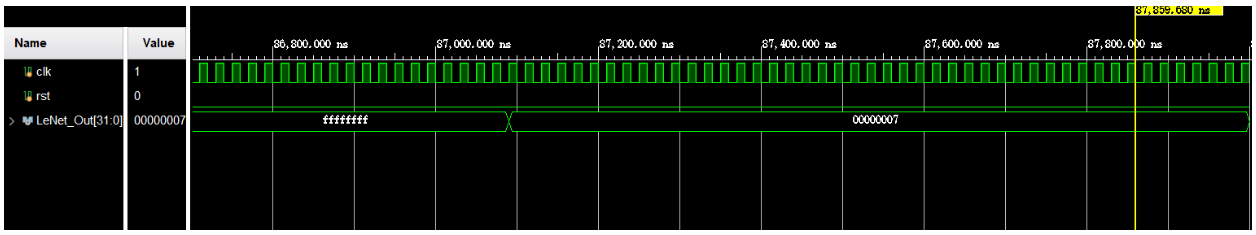

87,775.120 ns
Name
Value
clk 1
rst 0
LeNet_Out[31:0] 00000002
86,800.000 ns
87,000.000 ns
87,200.000 ns
87,400.000 ns
87,600.000 ns
87,800.000 ns
ffffffff
00000002

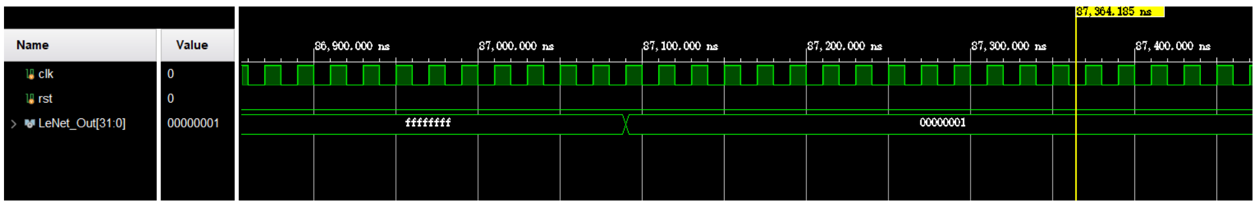

87,737.417 ns
Name
Value
clk 1
rst 0
LeNet_Out[31:0] 00000003
86,800.000 ns
87,000.000 ns
87,200.000 ns
87,400.000 ns
87,600.000 ns
87,800.000 ns
ffffffff
00000003

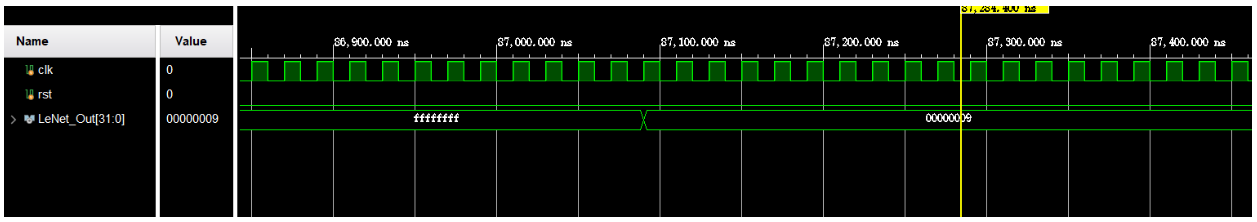

87,202.720 ns
Name
Value
clk 0
rst 0
LeNet_Out[31:0] 00000004
87,000.000 ns
87,050.000 ns
87,100.000 ns
87,150.000 ns
87,200.000 ns
ffffffff
00000004


87,822.120 ns
Name
Value
clk 0
rst 0
LeNet_Out[31:0] 00000003
86,800.000 ns
87,000.000 ns
87,200.000 ns
87,400.000 ns
87,600.000 ns
87,800.000 ns
ffffffff
00000003

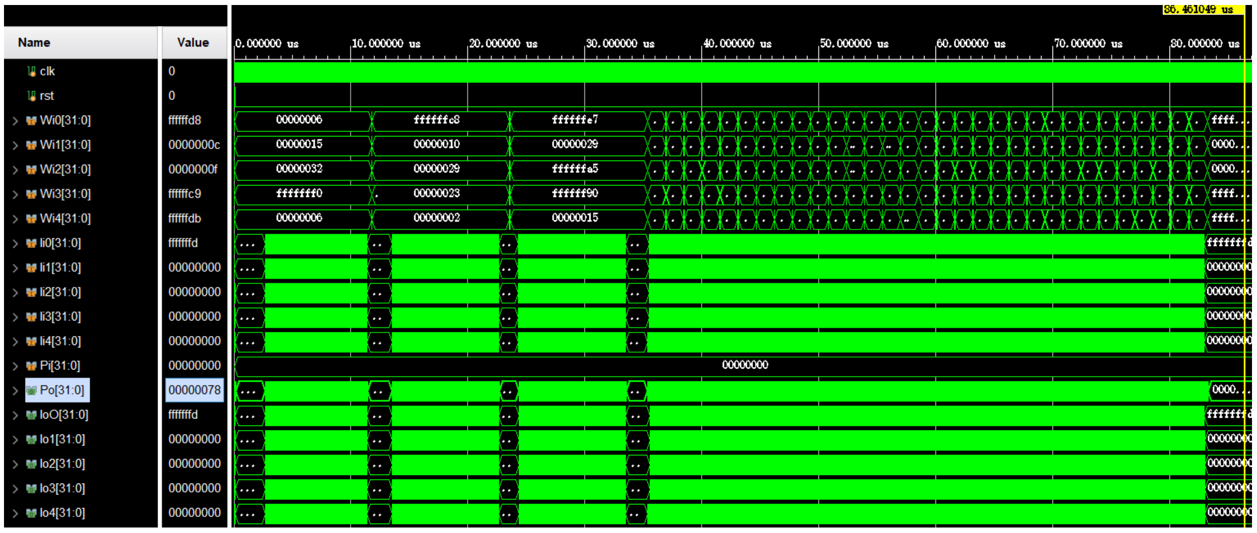

87,809.120 ns
Name
Value
clk 0
rst 0
LeNet_Out[31:0] 00000006
86,800.000 ns
87,000.000 ns
87,200.000 ns
87,400.000 ns
87,600.000 ns
87,800.000 ns
ffffffff
00000006


87,859.680 ns
Name
Value
clk 1
rst 0
LeNet_Out[31:0] 00000007
86,800.000 ns
87,000.000 ns
87,200.000 ns
87,400.000 ns
87,600.000 ns
87,800.000 ns
ffffffff
00000007

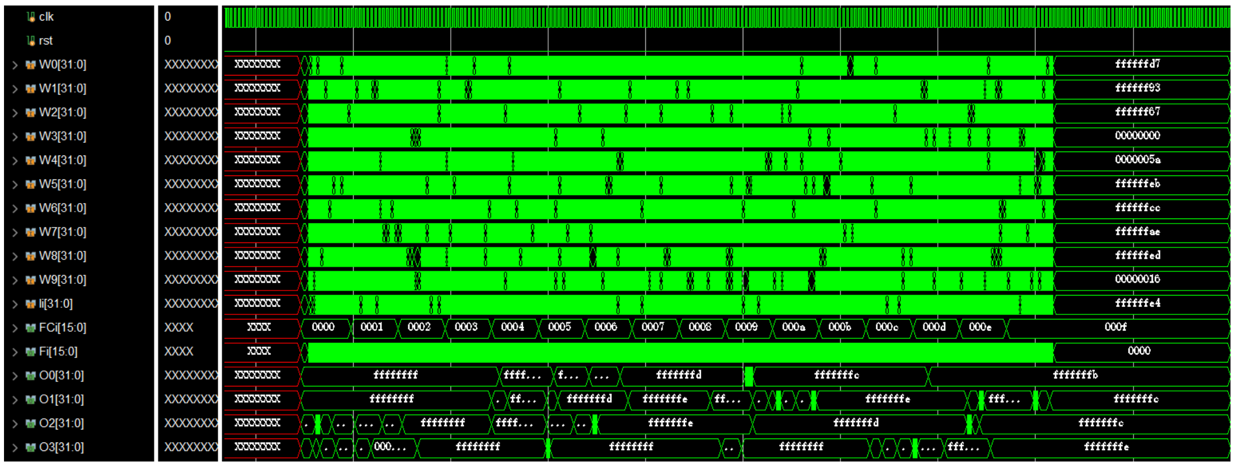




Moreover, all the aforementioned components have been subjected to rigorous simulation testing. The simulation outcomes of crucial elements such as the convolution layer and the fully connected layer demonstrate their functionality and are provided in the subsequent sections.



Simulation result for conv3D layer

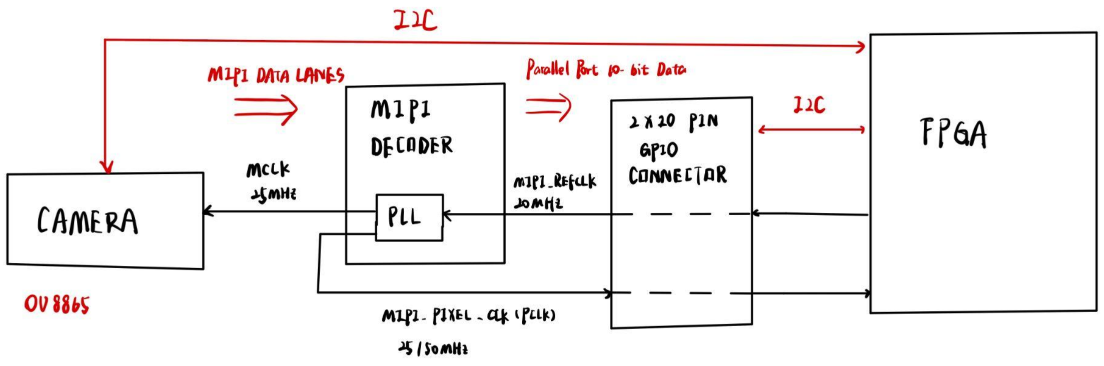


Simulation result for fully connected layer

3.5 Camera 2 VGA module

We use a D8M(ov8865) camera to capture real-time image, the D8M provides the Camera Module with high speed MIPI interface, which contributes to the simple 10Bit Parallel Bayer Data solution after the MIPI Decoder conversation. There are also lot of functions like auto-focus, zoom in and out provided by Terasic(http://www.terasic.com)

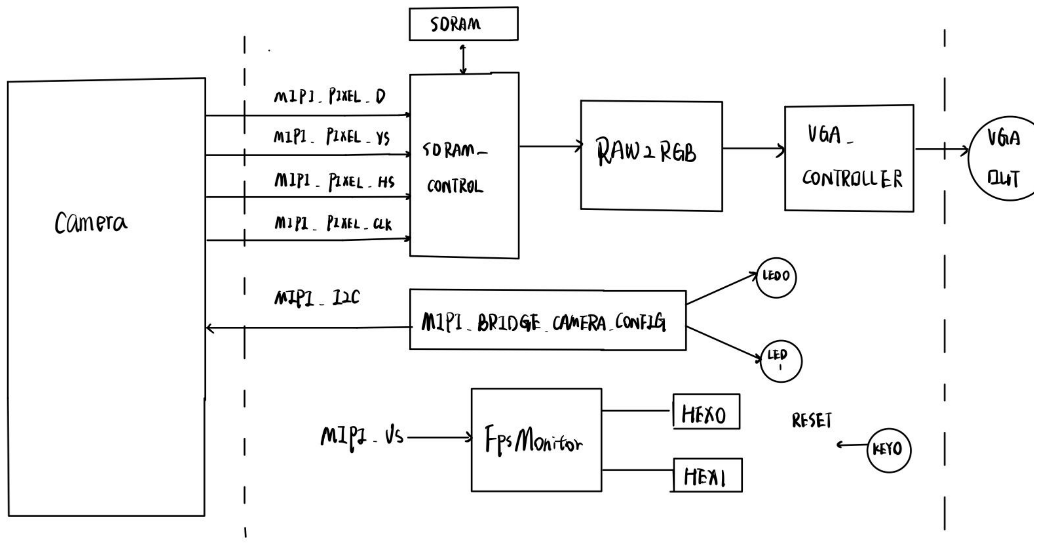
3.5.1 Camera module


I2C
MIPI DATA LANES
Parallel Port 10-bit Data
MIPI DECODER
2 X 20 PIN GPIO CONNECTOR
I2C
FPGA
MCLK 25MHz
MIPI_REFCLK 20MHz
PLL
CAMERA
OV8865
MIPI_PIXEL_CLK (PCLK) 25/50MHz

The diagram illustrates the clock tree for the D8M board. The MIPI Decoder Phase Locked Loop (PLL) accepts the FPGA Reference Clock (MIPI_REFCLK) and sends a clock signal to the camera sensor (MCLK). Concurrently, the MIPI Decoder PLL also generates a parallel port clock (MIPI_PIXEL_CLK) that is sent back to the FPGA for managing parallel data operations.

3.5.2 Overview of diagram(Top level)

The figure below shows the block diagram from D8M camera to VGA monitor:



The captured D8M data will be first written into SDRAM. When the frame buffer is filled, SDRAM_control module will read out the data from SDRAM to RAW2RGB module, the RAW2RGB module as it is named will convert RAW data to RGB data. VGA_controller will generate the signal timing to allow the output from the former module to display on the VGA monitor.

3.5.3 Modules

MIPI_BRIDGE_CAMERA_Config: The D8M I2C setting controller, which is employed to configure the D8M to deliver output at a resolution of 640x480 pixels with a refresh rate of 60Hz. This operation primarily involves writing corresponding I2C parameters into the registers of the D8M MIPI decoder IC and the Camera Sensor IC, utilizing their individual I2C buses.

The MIPI_I2C bus is utilized to transmit data to the MIPI decoder IC, which has an I2C slave address of 0x1c. Similarly, the CAMERA_I2C bus is used for communication with the Camera Sensor, which has an I2C slave address of 0x6c.

SDRAM_control: This module can general external SDRAM Memory and read/write image data. We create two write and read FIFOs, after finishing writing a frame, sdram_control will read out data from SDRAM to the next module. This part is quite significant but also complicated, we have references from the demo provided by Terasic and write emails to get technical support from support@terasic.com.

RAW2RGB：This module is responsible for converting raw sensor data into an RGB format. Sensors often output data in a format known as a Bayer pattern, which is a grid of pixels where each pixel only captures one color component: red, green, or blue.

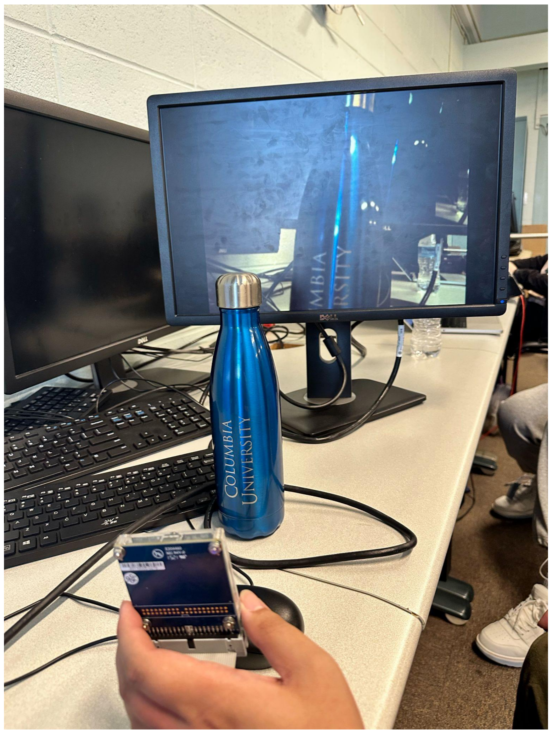
The RAW2RGB module would interpolate the missing color components for each pixel, resulting in an image where each pixel has red, green, and blue components.

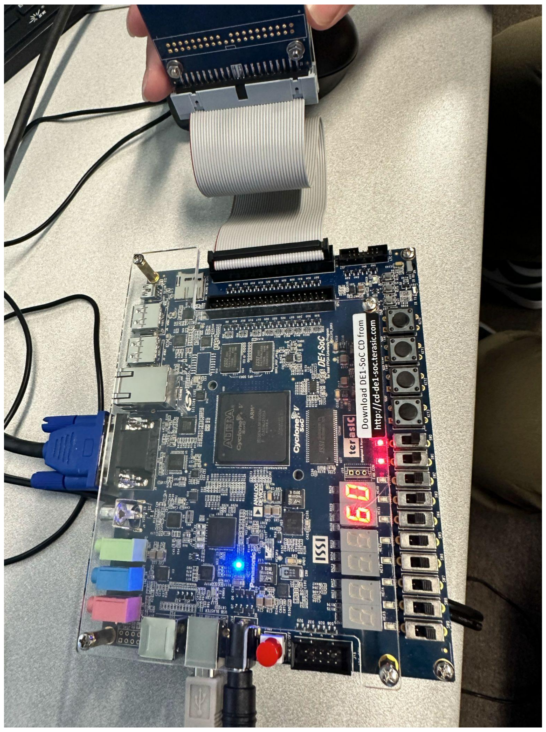
VGA_Controler: This module is a VGA signal timing generator. It is specifically designed to generate a signal timing pattern suitable for a 640x480 resolution display operating at a refresh rate of 60Hz.

The VGA signal timing involves specific timing parameters to ensure proper display functioning. These parameters include horizontal sync, vertical sync, front porch, back porch, and display area for both horizontal and vertical directions.

FpsMonitor: This module is designed to tally the pulses from the D8M MIPI_PIXEL_VS signal over a one-second interval, representing the frames per second (fps) count. It then transforms this tally into a decimal number, which is subsequently displayed on a pair of seven-segment LEDs.

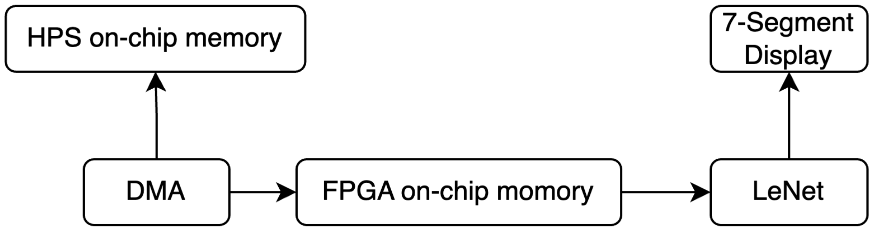
3.5.4 Results





We successfully accomplished the functionality of real-time camera capture and VGA monitor display. Hex[0] and Hex[1] show the frame rate of our frame steaming, there's a '60' shown on fpga since we set that in the SDRAM module and could fluctuate a bit due to jitter. Press key[0] can send reset signal that sleep the camera and monitor for a sec. LED[0] and LED[1] light up imply that camera_config and MIPI_bridge function well.

3.6 Data Transfer

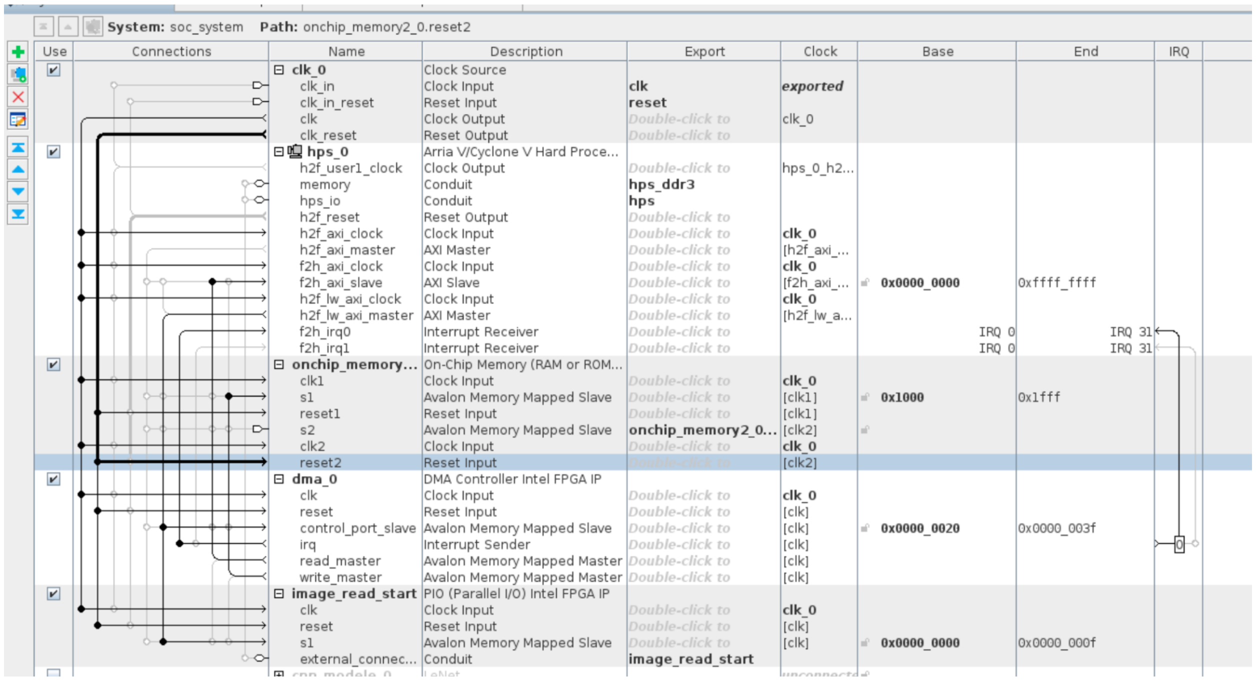


Directory memory access (DMA) is a feature of computer system which allows data to be transferred between device. We use the DMA to transfer data from HPS to FPGA.

The AXI slave on the HPS is able to access the HPS address space and retrieve the image data. For the DMA controller part, the read-master port is connected to the AXI-slave and the write-master port is connected to the on-chip memory block. The h2f_lw_axi_master in hps block is connected to the control port in the DMA block to set up the DMA transfer.

In the software part, a memory mapping (mmap) is first created, establishing a virtual memory address range for the HW_REGS spanning a specific size HW_REGS_SPAN. `void *virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );`The memory region is set to be both read and write. Then the lw_dma_addr is set to the calculated address by adding the offset ALT_LWFPGASLVS_OFST and the DMA_0_BASE value to the virtual_base address. `void *virtual_base = mmap( NULL, HW_REGS_SPAN, ( PROT_READ | PROT_WRITE ), MAP_SHARED, fd, HW_REGS_BASE );` The ALT_LWFPGASLVS_OFST is set as the 0xff200000 based on the document for the Cyclone V HPS Register Address Map. The DMA_0_BASE is accessed in hps_0.h file, which is generated through run the command `sopc-create-header-files "soc_system.sopcinfo" --single hps_0.h --module hps_0`.

Then the status, read, write, length, and control registers of DMA are configured to the corresponding value to prepare for the transfer. For example, for the DMA read address, we do `#define _DMA_REG_READ_ADDR(BASE_ADDR) *((uint32_t *)BASE_ADDR+1)` and`_DMA_REG_READ_ADDR(lw_dma_addr) = physical_addr; `.



System: soc_system **Path:** onchip_memory2_0.reset2

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
✓		**clk_0**	Clock Source					
		clk_in	Clock Input	**clk**	***exported***			
		clk_in_reset	Reset Input	**reset**				
		clk	Clock Output	*Double-click to*	clk_0			
		clk_reset	Reset Output	*Double-click to*				
✓		**hps_0**	Arria V/Cyclone V Hard Proce...					
		h2f_user1_clock	Clock Output	*Double-click to*	hps_0_h2...			
		memory	Conduit	**hps_ddr3**				
		hps_io	Conduit	**hps**				
		h2f_reset	Reset Output	*Double-click to*				
		h2f_axi_clock	Clock Input	*Double-click to*	**clk_0**			
		h2f_axi_master	AXI Master	*Double-click to*	[h2f_axi_...			
		f2h_axi_clock	Clock Input	*Double-click to*	**clk_0**			
		f2h_axi_slave	AXI Slave	*Double-click to*	[f2h_axi_...	**0x0000_0000**	0xffff_ffff	
		h2f_lw_axi_clock	Clock Input	*Double-click to*	**clk_0**			
		h2f_lw_axi_master	AXI Master	*Double-click to*	[h2f_lw_a...			
		f2h_irq0	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31	
		f2h_irq1	Interrupt Receiver	*Double-click to*		IRQ 0	IRQ 31	
✓		**onchip_memory...**	On-Chip Memory (RAM or ROM...					
		clk1	Clock Input	*Double-click to*	**clk_0**			
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk1]	**0x1000**	0x1fff	
		reset1	Reset Input	*Double-click to*	[clk1]			
		s2	Avalon Memory Mapped Slave	**onchip_memory2_0...**	[clk2]			
		clk2	Clock Input	*Double-click to*	**clk_0**			
		reset2	Reset Input	*Double-click to*	[clk2]			
✓		**dma_0**	DMA Controller Intel FPGA IP					
		clk	Clock Input	*Double-click to*	**clk_0**			
		reset	Reset Input	*Double-click to*	[clk]			
		control_port_slave	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0020**	0x0000_003f	
		irq	Interrupt Sender	*Double-click to*	[clk]			0
		read_master	Avalon Memory Mapped Master	*Double-click to*	[clk]			
		write_master	Avalon Memory Mapped Master	*Double-click to*	[clk]			
✓		**image_read_start**	PIO (Parallel I/O) Intel FPGA IP					
		clk	Clock Input	*Double-click to*	**clk_0**			
		reset	Reset Input	*Double-click to*	[clk]			
		s1	Avalon Memory Mapped Slave	*Double-click to*	[clk]	**0x0000_0000**	0x0000_000f	
		external_connec...	Conduit	**image_read_start**				

After all image data is successfully write to the on-chip memory, we set the start_read signal to 1. ` void *start_read = virtual_base+((unsigned long)(ALT_LWFPGASLVS_OFST +IMAGE_READ_START_BASE)&( unsigned long)(HW_REGS_MASK));*((uint32_t *)start_read) = 1;`.

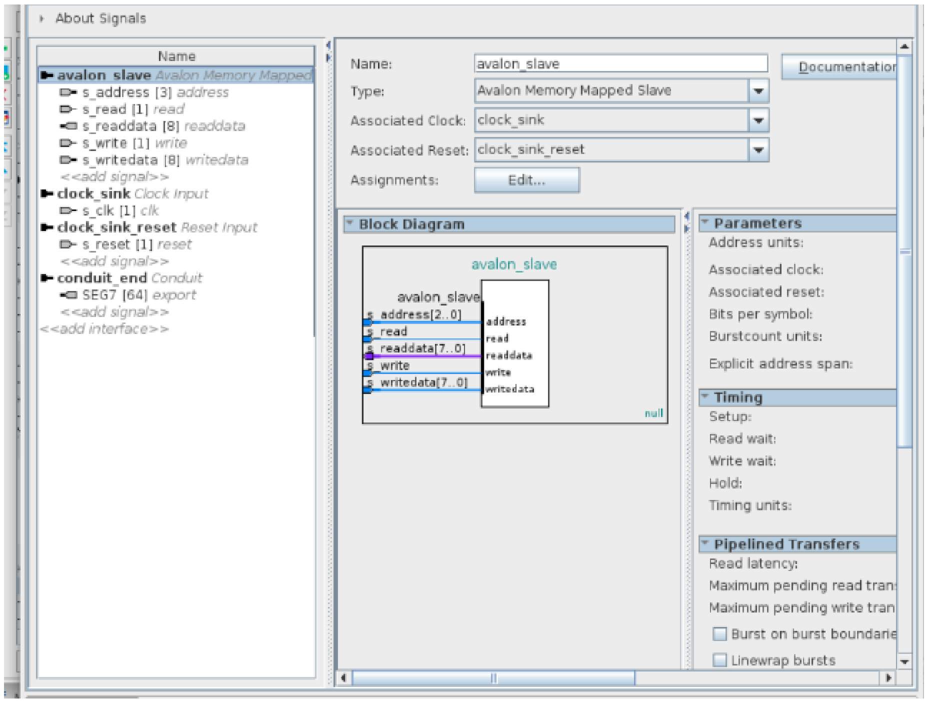
4. 7 Segment display the result:

4.1. Implementation of a 7-Segment Block in Quartus Platform Designer for an Electrical Engineering Project

In this section, we will discuss the steps taken to implement a 7-segment block in Quartus Platform Designer for an electrical engineering project. The goal of this project was to create a functional 7-segment display using FPGA technology.

4.2. Component Creation

The first step in the implementation process was to create a component in Quartus Platform Designer to represent the 7-segment block. And the block signals and interfaces are shown in figure 1. Once the component was created, a related TCL file was generated.



```
Name
avalon_slave Avalon Memory Mapped
    s_address [3] address
    s_read [1] read
    s_readdata [8] readdata
    s_write [1] write
    s_writedata [8] writedata
    <<add signal>>
clock_sink Clock Input
    s_clk [1] clk
clock_sink_reset Reset Input
    s_reset [1] reset
    <<add signal>>
conduit_end Conduit
    SEG7 [64] export
    <<add signal>>
<<add interface>>
```

```
Name:               avalon_slave
Type:               Avalon Memory Mapped Slave
Associated Clock:   clock_sink
Associated Reset:   clock_sink_reset
Assignments:        Edit...
```

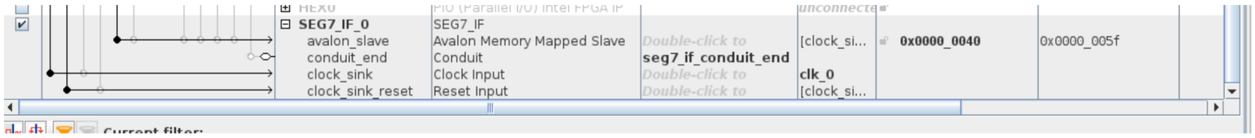
```
Block Diagram
avalon_slave
s_address[2..0]
s_read
s_readdata[7..0]
s_write
s_writedata[7..0]
```

```
Parameters
Address units:
Associated clock:
Associated reset:
Bits per symbol:
Burstcount units:
Explicit address span:
Timing
Setup:
Read wait:
Write wait:
Hold:
Timing units:
Pipelined Transfers
Read latency:
Maximum pending read tran
Maximum pending write tran
☐ Burst on burst boundarie
☐ Linewrap bursts
```

Fig1: The signals and interfaces of the 7-segment block.

4.3. Block Connection

After creating the component, it was added to the system contents and connected with the clock and hps. Specifically, the avalon_slave was connected with h2f_axi_master in hps0, clock_sink was connected with clk in clk_0, and clock_sink_reset was connected with clk_reset in clk_0. After successful connection, the system contents looked like figure 2.



```
⊞ HEX0                 PIO (Parallel I/O) Intel FPGA IP                          unconnecte
⊟ SEG7_IF_0            SEG7_IF
    avalon_slave       Avalon Memory Mapped Slave   Double-click to        [clock_si...   0x0000_0040   0x0000_005f
    conduit_end        Conduit                      seg7_if_conduit_end
    clock_sink         Clock Input                  Double-click to        clk_0
    clock_sink_reset   Reset Input                  Double-click to        [clock_si...
Current filter:
```

Fig2: QSys System Components and Connections of 7 segment block

4.4. Top SV File Assignment and Definition

According to this project's requirement, it only need to display a hexadecimal number in one of the 7 segment displays. So HEX0 has been chosen. Thus, the other 7 segment displays' default active has been set as 0 (which will not display any light) by using the code below:

```
  soc_system soc_system0(
        .seg7_if_conduit_end_export     ({ HEX1,HEX2,HEX3,HEX4,HEX5}),
)
```

Then, read the result from LeNet using the code below:

```
  LeNet LeNet_1 (.clk(CLOCK_50), .rst(rst), .LeNet_Out(LeNet_Out1));
```

Next, using these code to send the last 4 bits of LeNet as the input to the model hex7seg:

```
        hex7seg h1(.a(LeNet_Out1[3:0]), .y(HEX0));
```

The last 4 bits of LeNet_Out stored the binary type of final result needed to be output. The code of model hex7seg is shown below. In this model, the binary type input will be translated to the related array represents the array that can display the result in hexadecimal type on FPGA board.

```
  case(a)
        4'b0000: y = 7'h40;
        4'b0001: y = 7'h79;
        4'b0010: y = 7'h24;
        4'b0011: y = 7'h30;
        4'b0100: y = 7'h19;
        4'b0101: y = 7'h12;
        4'b0110: y = 7'h02;
        4'b0111: y = 7'h78;
        4'b1000: y = 7'h00;
        4'b1001: y = 7'h10;
        default: y = 7'd00;
  endcase
```

To give more details about how to define these output numbers, here will introduce the principle of the DE1-SoC 7 segment displays. The example of how 7 segments consists of a “HEX” is shown in figure3 below. Each segment is linked to an individual pin. These segment signals have active-low behavior, meaning that a "0" activates them. The rightmost bit (bit 0) in each 7-bit segment vector corresponds to the "a" segment, bit 1 corresponds to the "b" segment, and so on, with the leftmost bit (bit 6) controlling the "g" segment.

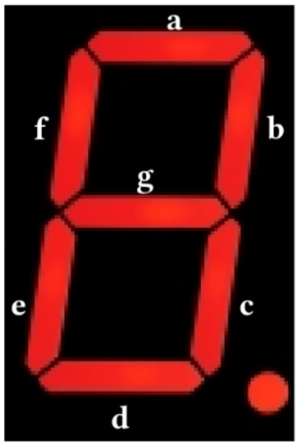


Fig 3: a hex module in DE1-SoC board consists of 7 segments.

Since the result could only be an integer from 0 to 9, the SV was written in 9 cases. Depending on the input binary numbers, the corresponding case was used to give the hexadecimal values for HEX0. This generated the correct result in the FPGA 7-segment screen.

4.5. The result:

By implement these in real FPGA board, its output is shown as below:

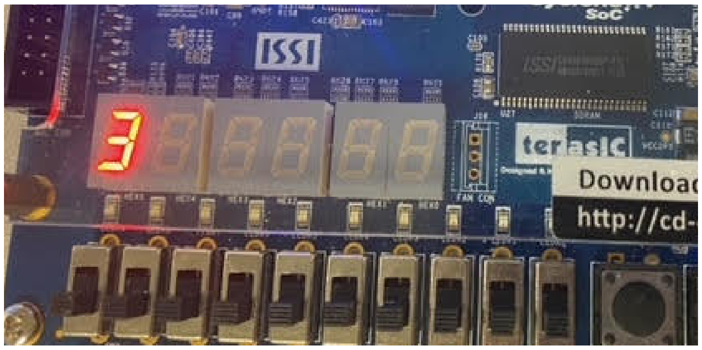


Fig 4: The FPGA hex output example.

4.6. Future improvement on the 7 segment display

Based on the 4.2 and 4.3, the hardware (HPS-FPGA) architecture has already been connected in the qsys file. In the future work, the controlling of the 7 segment display can be implemented in the C program(software part). It can be done by using a related bridge to map the SEG7_IF block's port.Then create a thread and write the related function so that the content of the src address can be copied to dest and then put in use. Also, it modularizes the 7 segment part, so that it is easier for future work if it is needed to be wired up with any other part of the FPGA i.e. memory, dma and so on.

Reference

[1] Ahmad, Hazoor & Tanvir, Muhammad & Abdullah, Muhammad & Javed, Muhammad Usama & Hafiz, Rehan & Shafique, Muhammad. (2018). Systimator: A Design Space Exploration Methodology for Systolic Array based CNNs Acceleration on the FPGA-based Edge Nodes.

[2] rmcbarreto. DE1-SoC-pipeline-haddoc2-rebuild. GitHub. Available at: https://github.com/rmcbarreto/DE1-SoC-pipeline-haddoc2-rebuild (Accessed: May 11, 2023).

[3] Terasic Technologies. (n.d.). DE10-Nano Kit. Retrieved May 11, 2023, from https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=68&No=1011&PartNo=4#contents