ASIC & FPGA Chip Design

Spring 2024

Dr. Mahdi Shaabani

Project Assignment: Stage 1

:Implementation overview

Arman yazdani 400102255

Zahra Mojtahedin 99102167

Intro

The final project for this course centers around the application of signal processing techniques using Zynq Boards.

In the initial phase of the project, the primary aim is to implement an FFT (Fast Fourier Transform) block to analyze and manipulate the provided signal data.

Subsequently, the project will expand to incorporate additional components, such as the integration of visual displays to showcase the processed results on a monitor through the HDMI port. This comprehensive approach will enable a thorough exploration of signal processing capabilities within the context of Zynq Boards. As outlined in the given manual, we commence the process by generating a variety of waveforms (Simple Pulse, Sawtooth, Triangular, and Sinusoidal) in the PS section of the Zynq board. One of the selected waveforms will be transferred to the PL section using proper configuration and will be channeled into the FFT block to perform the desired operation on the received signal. By doing so, we would have two signals ready for display on the monitor: the initial waveform, followed by the processed signal that has passed through the FFT Block.

It's worth noting that our given structure differs from the one provided in the manual, as we return the FFT-applied signal to the PS and follow the rest of the flow from that point.

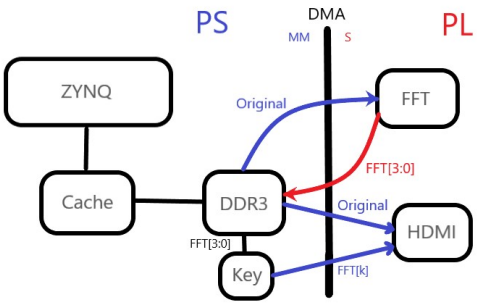

Figure 1: General flow

Programming logic(PL)

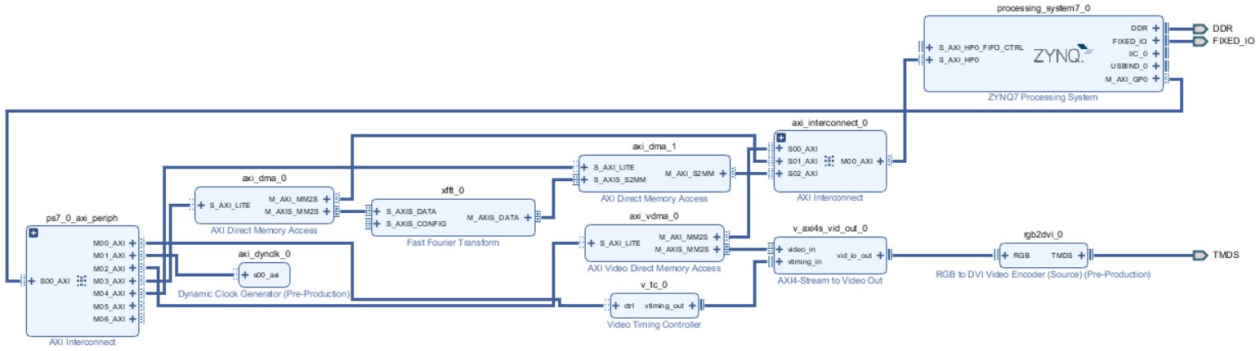

Figure 2: The final block design of the stage 1

The provided block design would produce a visual display showcasing both the initial signal and a secondary signal that has been processed through the FFT block. The output would be presented on the connected monitor via the HDMI port. It's important to note that we were given two distinct project files, each containing the essential blocks required for the first phase of the project. However, rather than relying solely on the provided resources, we elected to start from scratch and meticulously assemble the blocks to ensure they function cohesively. This approach allowed us to gain a deeper understanding of the underlying architecture and signal processing principles at play. By constructing the system from the ground up, we were able to customize and fine-tune the components to meet the specific requirements of our project. The decision to forgo the pre-configured project files and instead build the system manually was a strategic one. It not only reinforced our comprehension of the technical concepts but also enabled us to address any potential limitations or discrepancies in the provided resources.

Through this hands-on process, we were able to develop a robust and tailored solution that not only meets the project's objectives but also showcases our proficiency in signal processing and Zynq Board integration. The ability to create and configure the system from scratch has been a valuable learning experience, equipping us with the skills and confidence to tackle future signal processing challenges with a deeper understanding of the underlying principles.

A closer look to flow of data

- Different Waveforms are generated in the ZYNQ7 Processing System (PS) and will be stored in the DDR Memory respectively. PS Block is connected to peripheral block to be properly configured, and it's also connected to an AXI Interconnect to transmit and receive the desired data.
- AXI Peripheral Block is responsible for proper configuration of other blocks, so that they

can function properly.

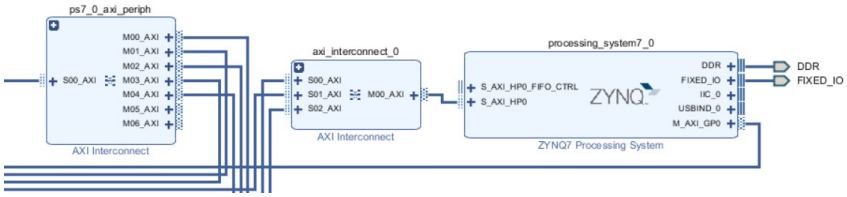

Figure 3: ZYNQ7 Processing System

- As we have our waveforms ready to be used, they'll get passed to a DMA Block to be read and converted to Stream.
- The FFT block will receive a stream of the selected waveform and applies FFT on the received signal.
- The final signal (which is FFT of the primary received waveform) should be stored in the memory again. To do so, the FFT-applied signal will be transferred to another DMA block to convert the Received stream to Memory Map and store the Secondary signal in DDR Memory respectively.

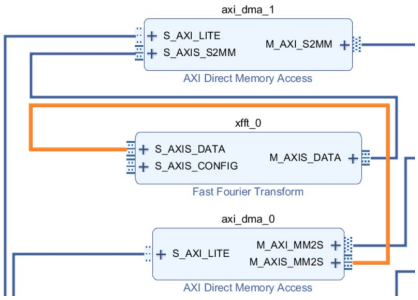

Figure 4: Reading and Writing via DMA

- At this point, we have both signals in the DDR Memory, so we can proceed with displaying section.
- Both stored signals (Main and FFT-applied) will be passed to an AXI Video DMA Block to be converted from Memory Map to Stream.
- This DMA block will pass the signal to AXI4-Stream to Video Out to convert the stream to a suitable format for display
- The Video-out block in the mentioned schematic requires two different clock signals, so a Dynamic Clock Generator is placed in the design to handle these signals.

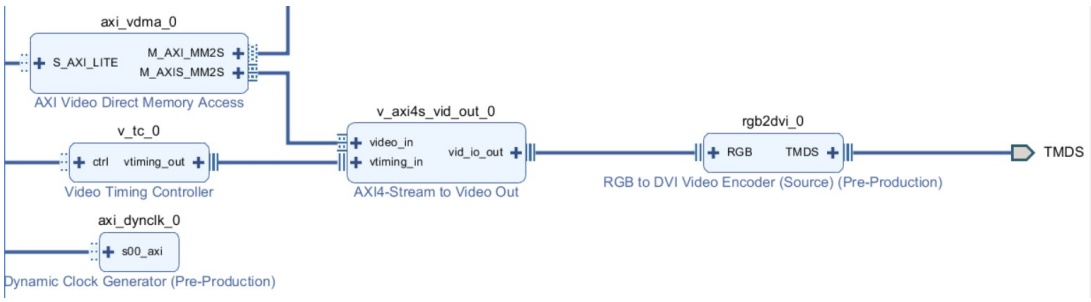

Figure 5: Display Process

Configurations

There has been number of configurations needed to be done during this stage. Number of these critical configurations are listed below for more clarification of what has been done:

- FFT Block needs to be configured properly in order to function smoothly. Configuration of other blocks are handled using peripheral block. However, FFT Block couldn't be configured using mentioned block. As a result, two Constant 1s were connected to the config. ports of the FFT block to ensure its functionality.

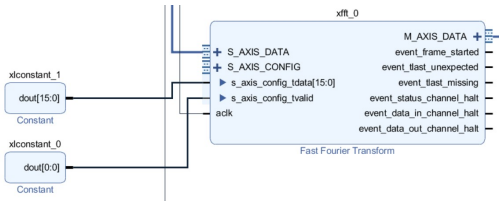

Figure 6: FFT configuration

- There has been number of challenges with the width of Input and Output Data. According to the setup of DMA block, it's working with 32-bit input/output data, while we've stored 16-bit data in the PS (DDR Memory). To ensure the proper functionality, we need to convert the stored 16-bit data to 32-bit, considering that the FFT block interpret the first half of the data (LSB Side) as the Real component, and the MSB Side as the Imaginary Component. By adding zeros in between our stored data, we can ensure that proper data is delivered to the FFT block, and the interpretation is correct.
- During the Synthesis process, we've encountered a critical warning as a result of different clocks we have in our setup. These different clocks were only used to ensure the functionality of last block (RGB to DVI Video Encoder) and thus, we forced the whole system to work with it's clock.

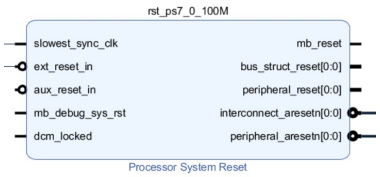

Figure 7: Clock&Reset configuration

- interrupts should be made and sent to PS for further consideration and mangement.

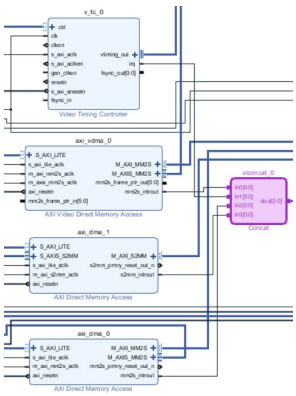

Figure 8: Interrupt vector

- Video Timing Controller Block should be configured properly with respect to the LCD (Monitor) we use.

Processing System(PS)

We have provided enough describtion of the PS part in the presentation session, More over we mention the key changes made to template projects for our use here.

Definitions

```
#define MAX_DMA_LEN         0x400      /* DMA max length in byte */
#define DMA_0_DEV_ID        XPAR_AXIDMA_0_DEVICE_ID
#define DMA_1_DEV_ID        XPAR_AXIDMA_1_DEVICE_ID
#define S2MM_INTR_ID        XPAR_FABRIC_AXI_DMA_1_S2MM_INTROUT_INTR
#define MM2S_INTR_ID        XPAR_FABRIC_AXI_DMA_0_MM2S_INTROUT_INTR
#define KEY_INTR_ID         XPAR_XGPIOPS_0_INTR
```

Figure 9: adma_ctrl.h:defining read&write&key Intrr

```
#define CANVAS_LEN          1920*1080*3    /* Canvas total length in byte */
#define WAVE_START_ROW1     50             /* Grid and Wave1 start row in frame */
#define WAVE_START_ROW2     300            /* Grid and Wave2 start row in frame */
#define WAVE_START_COLUMN   0              /* Grid and Waves start column in frame */
#define WAVE_HEIGHT         256            /* Grid and Wave height */
```

Figure 10: adma_ctrl.h:waves positioning

```
volatile int s2mm_flag ;
volatile int mm2s_flag ;
volatile int key_flag ;
```

Figure 11: adma_ctrl.c:defining read&write&key flags

```
int XAxiDma_Write_Initial(u16 DeviceId, u16 IntrID, XAxiDma *XAxiDma, XScuGic *InstancePtr) ;
int XAxiDma_Read_Initial(u16 DeviceId, u16 IntrID, XAxiDma *XAxiDma, XScuGic *InstancePtr) ;
void Dma_Read_Interrupt_Handler(void *CallBackRef);
void Dma_Write_Interrupt_Handler(void *CallBackRef);
void frame_copy(u32 width, u32 height, u32 stride, int hor_x, int ver_y, u8 *frame, u8 *CanvasBufferPtr) ;
int KeySetup(XScuGic *InstancePtr, u16 IntrID, XGpioPs *GpioInstancePtr) ;
void GpioHandler(void *CallbackRef);
```

Figure 12: adma_ctrl.c:declaring read&write&key Intrr initializers and ISRs

```
int XAxiDma_Adc_Wave(u32 width, u8 *frame, u32 stride, XScuGic *InstancePtr)
{
    int Status;
    u32 wave_width = width ;

    s2mm_flag = 1 ;
    mm2s_flag = 1 ;

    XAxiDma_Write_Initial(DMA_1_DEV_ID, S2MM_INTR_ID, &AxiDma_1, InstancePtr) ;
    XAxiDma_Read_Initial(DMA_0_DEV_ID, MM2S_INTR_ID, &AxiDma_0, InstancePtr) ;
```

Figure 13: adma_ctrl.c:instancing read&write Intrr initializers inside *while(1)*

```
if (s2mm_flag)
{
    /* clear s2mm_flag */
    s2mm_flag = 0 ;

    double real, imag;
    for (int i = 0; i < MAX_DMA_LEN; i++) {
        real = (double)((DmaRxBuffer[i] & 0xFFFF) / 128.0);
        imag = (double)((DmaRxBuffer[i] & 0xFFFF0000) / 128.0);
        AmpBuffer[i] = sqrt(real * real + imag * imag);
    }

    /* Grid Overlay */
    draw_grid(wave_width, WAVE_HEIGHT, CanvasBuffer) ;
    /* wave Overlay */
    draw_wave(wave_width, WAVE_HEIGHT, (void *)RealBuffer, CanvasBuffer, UNSIGNEDCHAR, ADC_BITS, YELLOW
    draw_wave(wave_width, WAVE_HEIGHT, (void *)DmaRxBuffer, CanvasBuffer, UNSIGNEDCHAR, ADC_BITS, YELLO
    /* Copy Canvas to frame buffer */
    frame_copy(wave_width, WAVE_HEIGHT, stride, WAVE_START_COLUMN, WAVE_START_ROW1, frame, CanvasBuffer
    frame_copy(wave_width, WAVE_HEIGHT, stride, WAVE_START_COLUMN, WAVE_START_ROW2, frame, CanvasBuffer
    /* delay 100ms */
    usleep(100000) ;
    /* start DMA translation from FFT to DDR3 */
    Status = XAxiDma_SimpleTransfer(&AxiDma_1,(UINTPTR) DmaRxBuffer,
            MAX_DMA_LEN, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
```

Figure 14: adma_ctrl.c:handling write flag

```
if (mm2s_flag)
{
    mm2s_flag = 0;
}
if (key_flag)
{
    switch(wave_sel)
    {
    case 0 : GetSquareWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    case 1 : GetTriangleWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    case 2 : GetSawtoothWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    case 3 : GetSubSawtoothWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    case 4 : GetSinWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    default: GetSinWave(MAX_DMA_LEN, MAX_AMP_VAL, AMP_VAL, WaveBuffer) ; break ;
    }

    memcpy(RealBuffer, WaveBuffer, MAX_DMA_LEN) ;
    for (int i = 0; i < MAX_DMA_LEN; i++) {
        DmaTxBuffer[i] = (u32)RealBuffer[i];
    }

    Xil_DCacheFlushRange((UINTPTR)DmaTxBuffer, MAX_DMA_LEN);

    if (wave_sel == 4)
        wave_sel = 0 ;
    else
        wave_sel++ ;

    /* Clear flag */
    key_flag = 0 ;
```

Figure 15: adma_ctrl.c:handling read&key flag