Using HLS IP in Zynq SoC Design

310551050 陳聖文
Git: https://github.com/wdb870914NYCU/HLS

Introduction

This Lab is similar to Lab1 and Lab2, which is to implement an IP with vitis HLS and vivado. The difference between this Lab and Lab1, Lab2 is that we use Zynq in this Lab, while we use Pynq in Lab1, Lab2. It means we use C to control the SoC rather than Python this time. We will learn how to create Vitis HLS IP by a Tcl script, how to import HLS design as IP into IP integrator, how to connect HLS IP to a Zynq SoC by AXI interface and how to configure HLS IP/DMA in software. In this Lab, we have two labs to do, lab1 is to implement mac IP, which will communicate with CPU through AXI4-Lite, lab2 is to implement fft IP, which will communicate with CPU through AXI4-Stream.

What's Learned & Discussion

We first learned a new way to create a Vitis HLS IP Block, which is through script. Of course, we need to prepare all source files with all necessary pragmas. In the script, we use following command to finish the whole process:

Command	Explanation
open_project	Open project
set_top	Set top-level function
add_files [-tb]	Add source [testbench] files
open_solution	Open a solution
set_part	Set specified board
create_clock -period [1,2,3]	Create a clock with certain period
csim_design	Run C Simulation
csynth_design	Run C Synthesis
cosim_design	Run Co-Simulation
export_design	Export IP
put	Display certain string on terminal

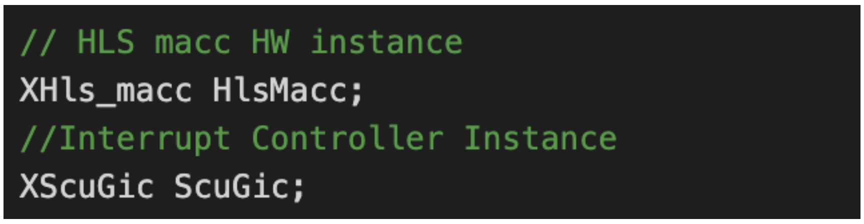
Then we know how to import the exported IP in Vivado. Through adding IP repository, create block design, custom the IPs and connect them. Which is almost the same as Lab1 and Lab2. However, we need to connect the blocks manually when implementing fft IP. When we finish the block design and bitstream generation, we will export the hardware.

Finally, we use Vitis IDE to create an application project, select the hardware we just export as the platform, choose the “Hello World” program to test the board connection, and program our design on FPGA. In this lab, we use C language to control the SoC. Vitis automatically builds the executable application with our C code, and then makes the CPU of the SoC run it.

C code in one of the most important parts in this lab. It decides how you verify, analyze, and use your design. Take lab1 (mac) for example, we declare hardware instance and interrupt controller instance first, then we define several functions. setup_interrupt() in Fig3 is used to setup the interruption routine (ISR). In this function, it will call XScuGic_Connect() to bind the interruption of one IP and the corresponding handler function together. hls_macc_init() in Fig4 is used to initialize the HLS block. Both setup_interrupt() and hls_macc_init() should run once before the usage of hardware IP. Fig5 shows how to set the input parameters of the HLS block, once we set the parameters and the block shows ready signal, we can call XHls_macc_start() to start the HLS block. Before starting the HLS block, we can choose whether to use interruption or not. If we want to use it, we should call XHls_macc_InterruptionEnable() and XHls_macc_InterruptionGlobalEnable(), which is shown in Fig6, and then we can use the interruption signal to know if the HLS block is done or not. If we don’t use interruption, we can also check the ready signal to know whether the block is IDLE or not, and it means the result is ready to fetch if we have started the HLS block before. Once the HLS block finishes the computation, we call XHls_macc_GetAccm() to fetch the result. Finally, we use C code to do the same computation, compare the result of hardware and software.

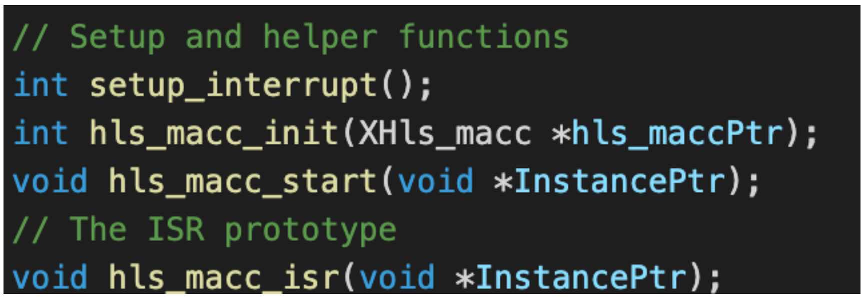
The C code of Lab2 is different from Lab1. It uses AXI4-Stream rather than AXI4-Lite. In other words, we use DMA instead of Set function and Get function to transfer the data. As a result, we need to initialize DMA first. Fig7 shows the init_dma() function. The initialization is similar to the initialization of the HLS block in Lab1. We don’t use the gather-scatter mode and interruption this time, which can be observed from init_dma(). Before sending data through DMA, we should flush the data in dcache to memory in case DMA receives dirty data. We call XAxiDma_SimpleTransfer() to transfer data. There are two directions of transfer, DEVICE to DMA and DMA to DEVICE, which DEVICE means the HLS block. Once we fetch the result through DMA, we need to invalidate the corresponding location of dcache to maintain the coherency. In the example ARM code, it will calculate the energy of the FFT result, and detect it with a certain threshold. Fig 8 shows the detail code.

Code



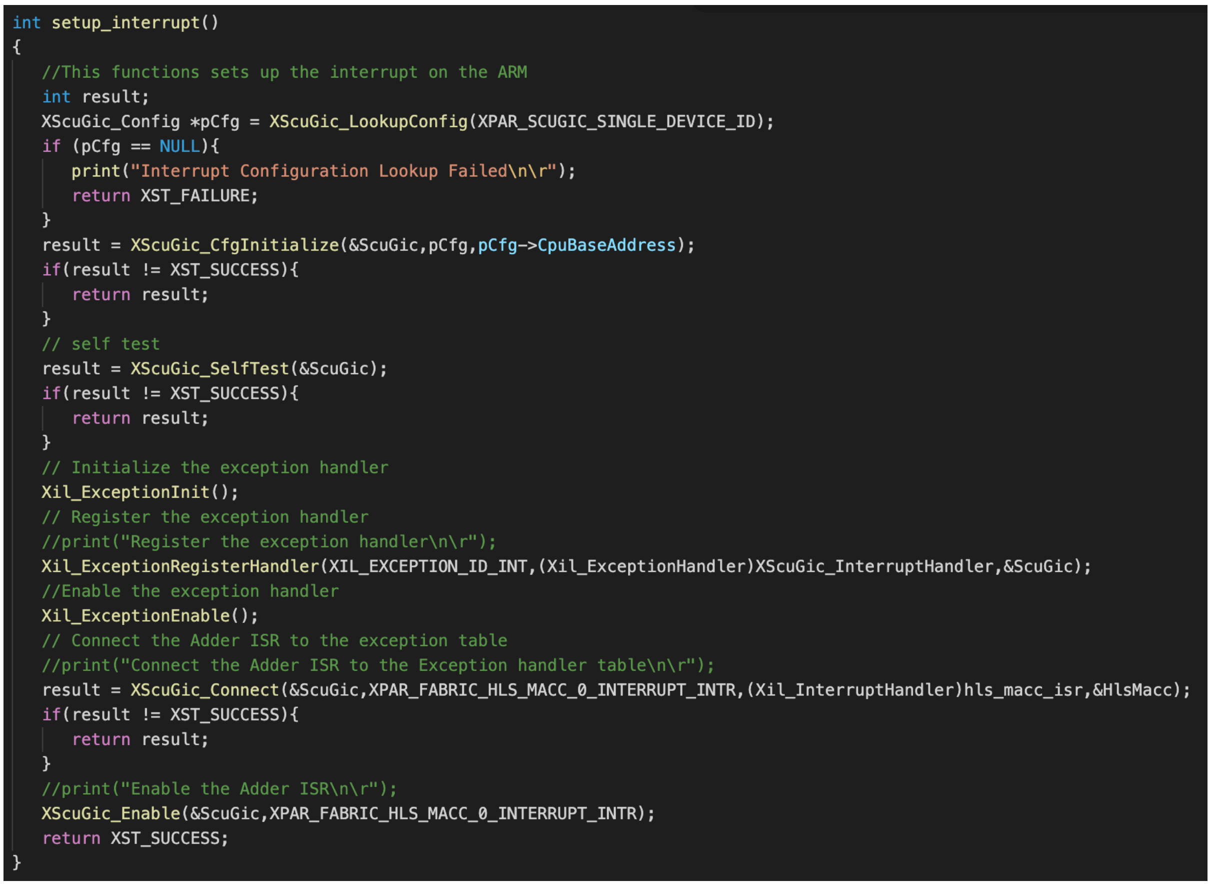
```c
// HLS macc HW instance
XHls_macc HlsMacc;
//Interrupt Controller Instance
XScuGic ScuGic;
```

Fig1, Instance Declaration



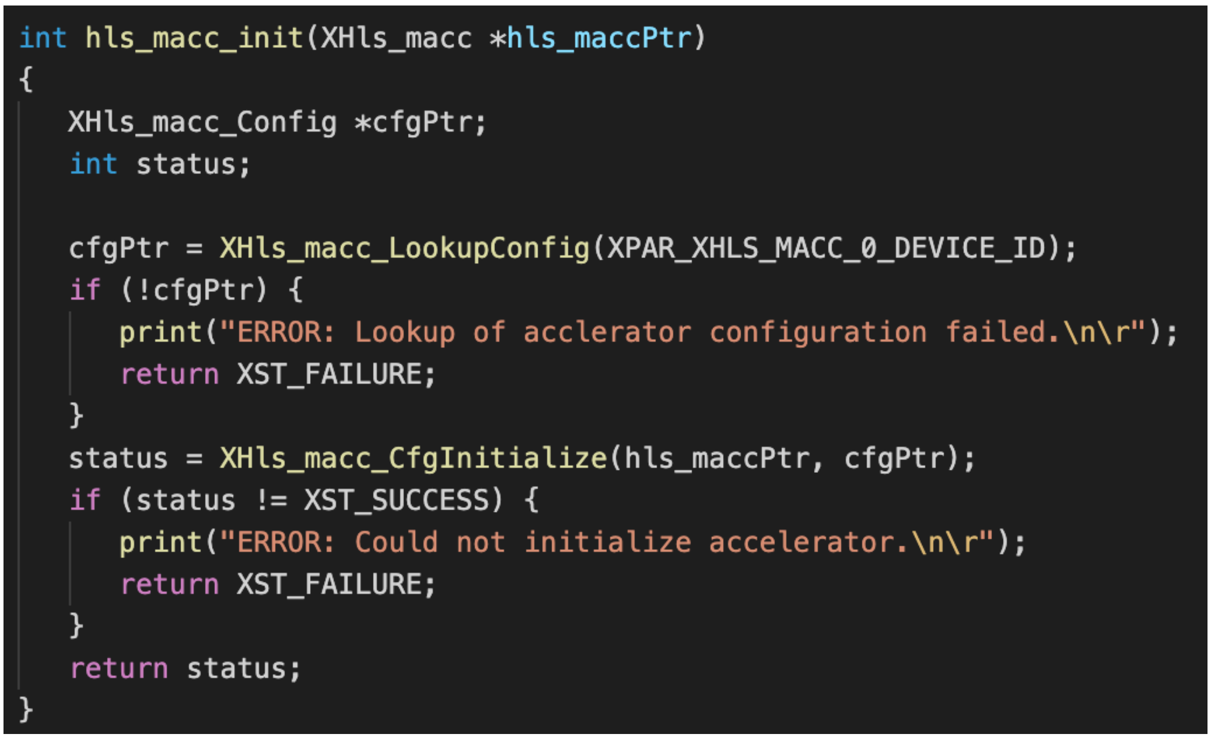
```c
// Setup and helper functions
int setup_interrupt();
int hls_macc_init(XHls_macc *hls_maccPtr);
void hls_macc_start(void *InstancePtr);
// The ISR prototype
void hls_macc_isr(void *InstancePtr);
```

Fig2, Function Declaration



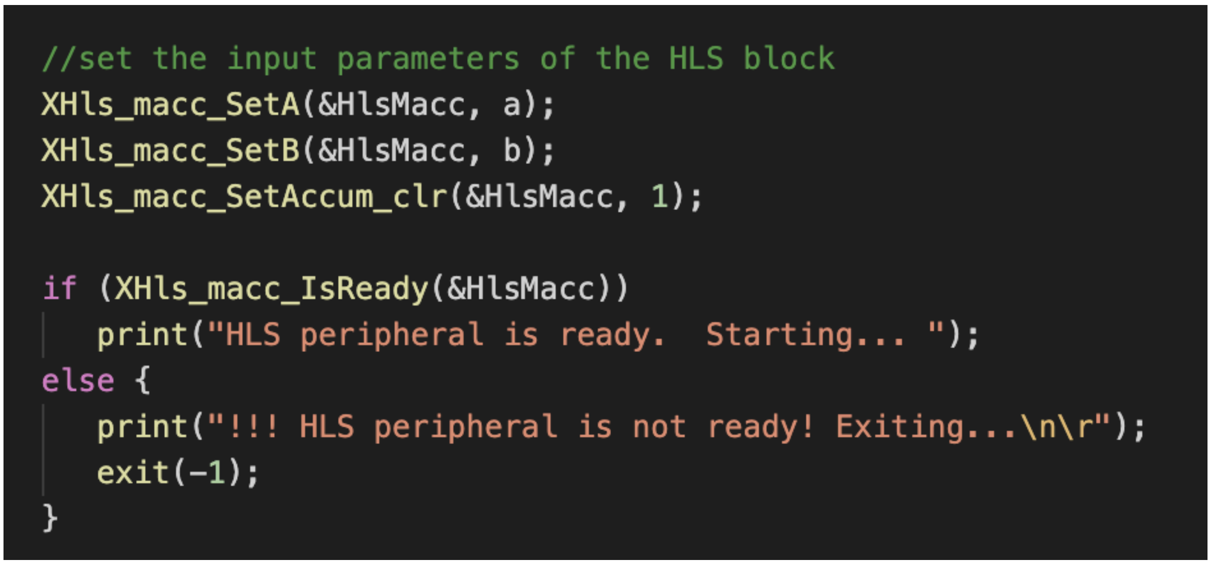
```c
int setup_interrupt()
{
   //This functions sets up the interrupt on the ARM
   int result;
   XScuGic_Config *pCfg = XScuGic_LookupConfig(XPAR_SCUGIC_SINGLE_DEVICE_ID);
   if (pCfg == NULL){
      print("Interrupt Configuration Lookup Failed\n\r");
      return XST_FAILURE;
   }
   result = XScuGic_CfgInitialize(&ScuGic,pCfg,pCfg->CpuBaseAddress);
   if(result != XST_SUCCESS){
      return result;
   }
   // self test
   result = XScuGic_SelfTest(&ScuGic);
   if(result != XST_SUCCESS){
      return result;
   }
   // Initialize the exception handler
   Xil_ExceptionInit();
   // Register the exception handler
   //print("Register the exception handler\n\r");
   Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,(Xil_ExceptionHandler)XScuGic_InterruptHandler,&ScuGic);
   //Enable the exception handler
   Xil_ExceptionEnable();
   // Connect the Adder ISR to the exception table
   //print("Connect the Adder ISR to the Exception handler table\n\r");
   result = XScuGic_Connect(&ScuGic,XPAR_FABRIC_HLS_MACC_0_INTERRUPT_INTR,(Xil_InterruptHandler)hls_macc_isr,&HlsMacc);
   if(result != XST_SUCCESS){
      return result;
   }
   //print("Enable the Adder ISR\n\r");
   XScuGic_Enable(&ScuGic,XPAR_FABRIC_HLS_MACC_0_INTERRUPT_INTR);
   return XST_SUCCESS;
}
```

Fig3, setup_interruption function



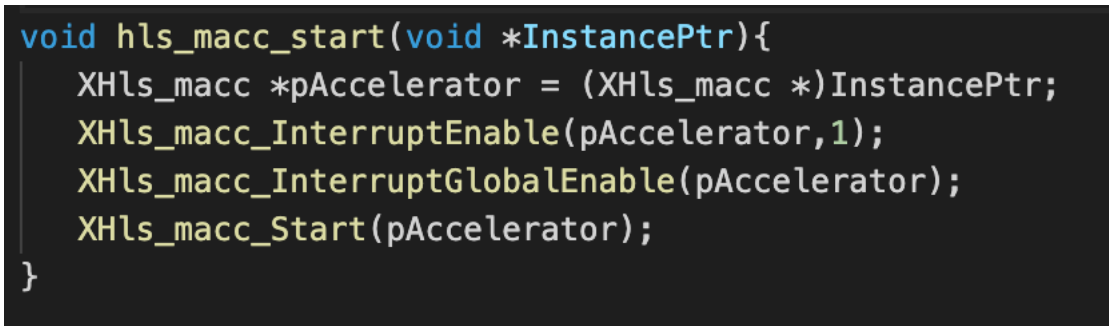
```c
int hls_macc_init(XHls_macc *hls_maccPtr)
{
    XHls_macc_Config *cfgPtr;
    int status;

    cfgPtr = XHls_macc_LookupConfig(XPAR_XHLS_MACC_0_DEVICE_ID);
    if (!cfgPtr) {
        print("ERROR: Lookup of acclerator configuration failed.\n\r");
        return XST_FAILURE;
    }
    status = XHls_macc_CfgInitialize(hls_maccPtr, cfgPtr);
    if (status != XST_SUCCESS) {
        print("ERROR: Could not initialize accelerator.\n\r");
        return XST_FAILURE;
    }
    return status;
}
```

Fig4, hls_macc_init function



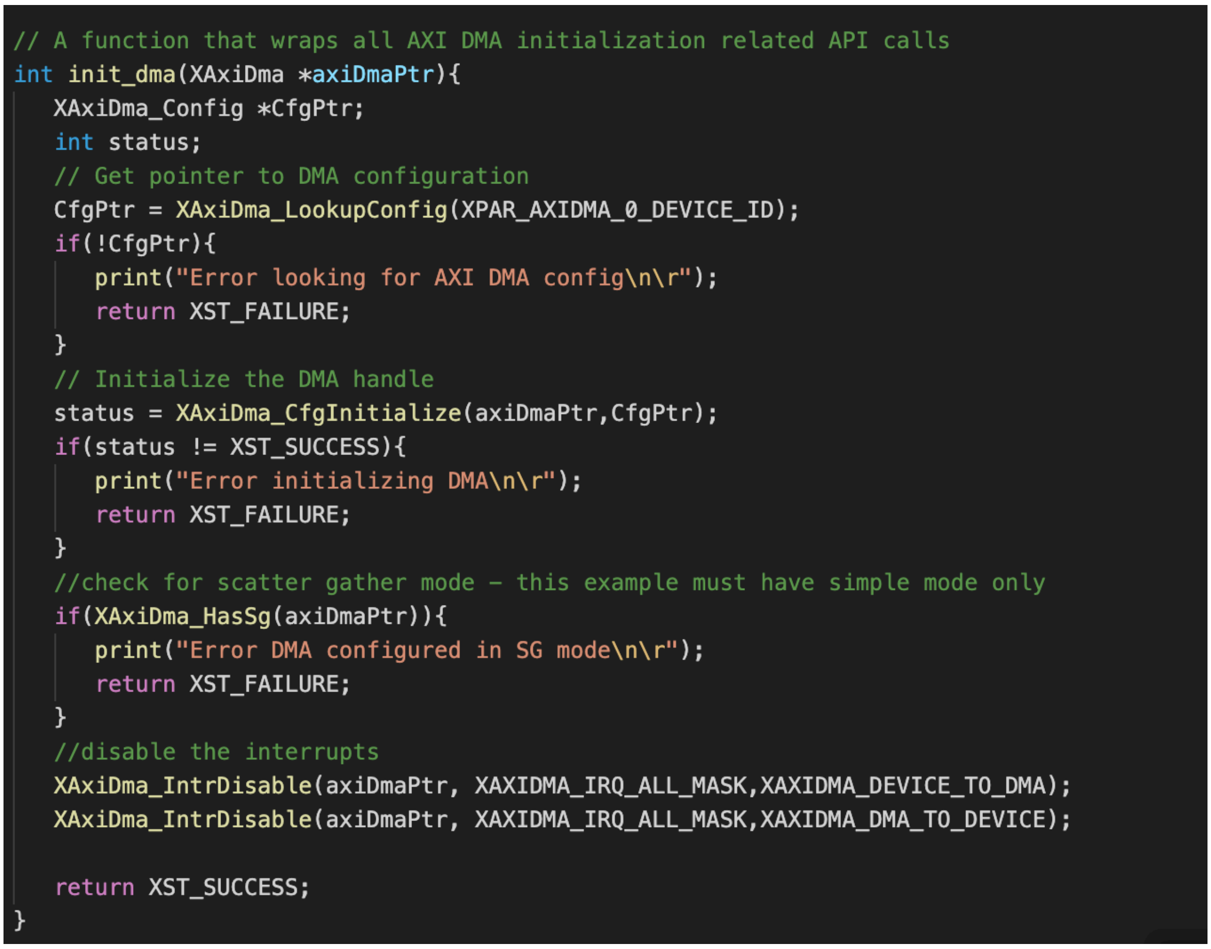
```c
//set the input parameters of the HLS block
XHls_macc_SetA(&HlsMacc, a);
XHls_macc_SetB(&HlsMacc, b);
XHls_macc_SetAccum_clr(&HlsMacc, 1);

if (XHls_macc_IsReady(&HlsMacc))
    print("HLS peripheral is ready.  Starting... ");
else {
    print("!!! HLS peripheral is not ready! Exiting...\n\r");
    exit(-1);
}
```

Fig5, set input parameters and ready check



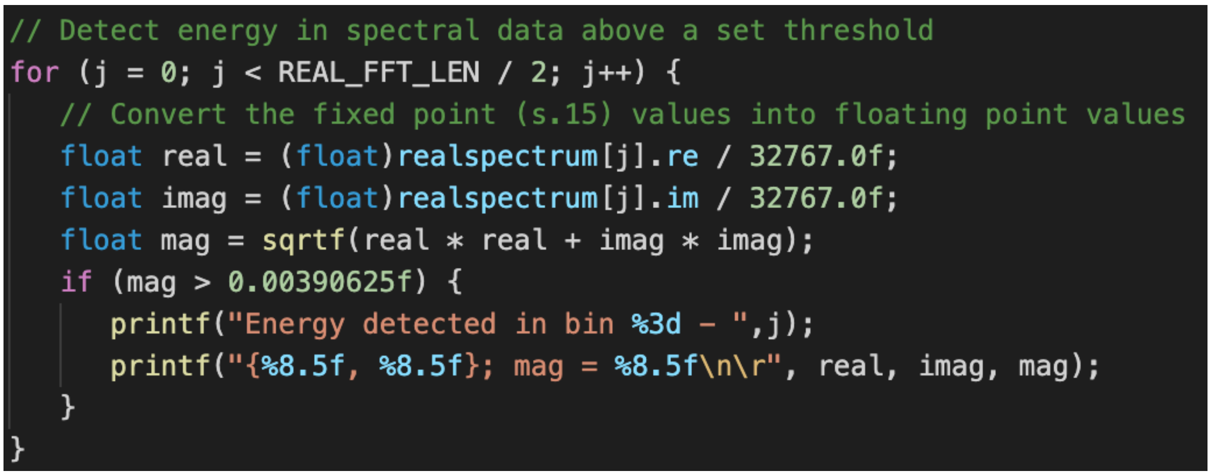
```c
void hls_macc_start(void *InstancePtr){
    XHls_macc *pAccelerator = (XHls_macc *)InstancePtr;
    XHls_macc_InterruptEnable(pAccelerator,1);
    XHls_macc_InterruptGlobalEnable(pAccelerator);
    XHls_macc_Start(pAccelerator);
}
```

Fig6, hls_macc_start function



```c
// A function that wraps all AXI DMA initialization related API calls
int init_dma(XAxiDma *axiDmaPtr){
    XAxiDma_Config *CfgPtr;
    int status;
    // Get pointer to DMA configuration
    CfgPtr = XAxiDma_LookupConfig(XPAR_AXIDMA_0_DEVICE_ID);
    if(!CfgPtr){
        print("Error looking for AXI DMA config\n\r");
        return XST_FAILURE;
    }
    // Initialize the DMA handle
    status = XAxiDma_CfgInitialize(axiDmaPtr,CfgPtr);
    if(status != XST_SUCCESS){
        print("Error initializing DMA\n\r");
        return XST_FAILURE;
    }
    //check for scatter gather mode - this example must have simple mode only
    if(XAxiDma_HasSg(axiDmaPtr)){
        print("Error DMA configured in SG mode\n\r");
        return XST_FAILURE;
    }
    //disable the interrupts
    XAxiDma_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DEVICE_TO_DMA);
    XAxiDma_IntrDisable(axiDmaPtr, XAXIDMA_IRQ_ALL_MASK,XAXIDMA_DMA_TO_DEVICE);

    return XST_SUCCESS;
}
```

Fig 7, init_dma function



```c
// Detect energy in spectral data above a set threshold
for (j = 0; j < REAL_FFT_LEN / 2; j++) {
    // Convert the fixed point (s.15) values into floating point values
    float real = (float)realspectrum[j].re / 32767.0f;
    float imag = (float)realspectrum[j].im / 32767.0f;
    float mag = sqrtf(real * real + imag * imag);
    if (mag > 0.00390625f) {
        printf("Energy detected in bin %3d - ",j);
        printf("{%8.5f, %8.5f}; mag = %8.5f\n\r", real, imag, mag);
    }
}
```

Fig 8, Energy Calculation

Screenshots

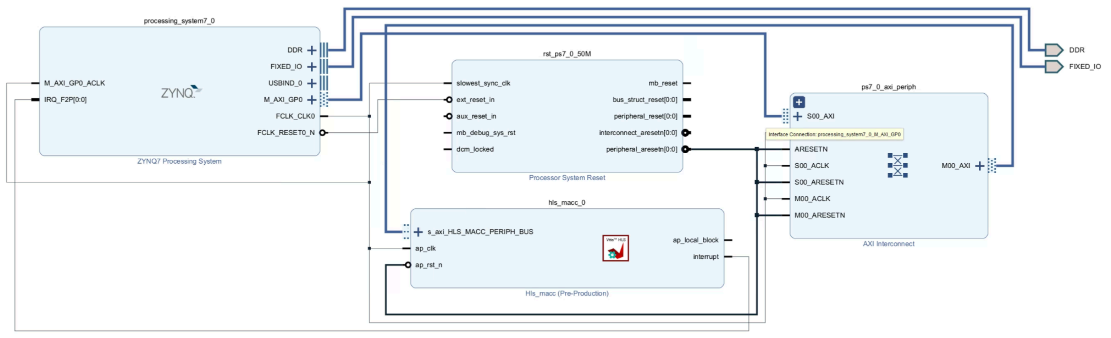


Fig 9, Block Diagram of Mac

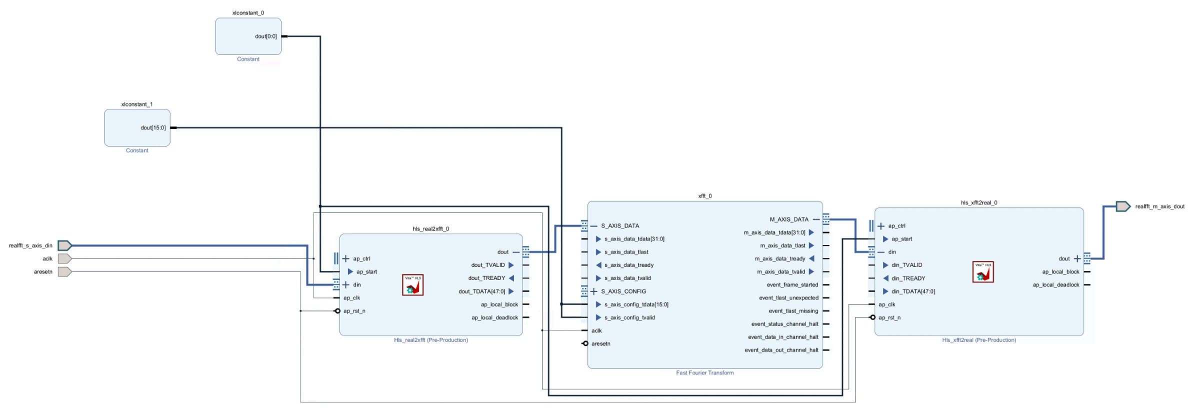


Fig 10, Block Diagram of FFT