For Stage 1: Handoff, implementation.

## Vivado

1. Create blank project
2. Create Block Design
3. Add MPSoC IP
    a. Run block automation – apply board presets
4. Tie both fpd_aclk to pl_clk0
5. Validate Design
6. Create HDL Wrapper
    a. Let Vivado manage
7. Generate Bitstream
8. Export Hardware
    a. Include Bitstream

## Vitis

1. Create Platform Component
    a. Hardware Design – From the .XSA exported in Vivado step 8
        i. OS - Standalone
        ii. Processor – psu_cortexa53_0
        iii. Generate Boot Artifacts
        iv. Generate PMU Firmware
        v. Target FSBL – psu_cortexr5_0
    b. Build Platform
2. Create FSBL
    a. New Component > From Example
    b. ZynqMP FSBL
        i. Create new domain
        ii. OS – Standalone
        iii. Processor – psu_cortexr5_0
3. Create PMUFW
    a. New Component > From Example
    b. ZynqMP PMU Firmware
        i. OS – Standalone
        ii. Processor – psu_pmu_0
4. Create Test Application
    a. New Component > From Example
    b. Hello World

        i. Select new Cortex R5 standalone domain
    c. Select application in Components pane
    d. Sources > src > lscript.ld
        i. Set all memory region fields to psu_r5_0_atcm_MEM_0
5. Builds
    a. Rebuild Platform
    b. Build FSBL
    c. Build PMUFW
    d. Build Hello_world
6. Create Boot Image (note: existing .bif can be imported if this step is being repeated)
    a. From top menu bar
        i. Vitis > Create Boot Image > Zynq Ultrascale+
    b. Remove all entries
    c. Add .elfs – located in each item's 'build' folder
        i. Hello_world.elf
            1. Type - Datafile
            2. Destination – r5_0
        ii. Fsbl.elf
            1. Type - Bootloader
            2. Destination – r5_0
        iii. Pmufw.elf
            1. Type – datafile
            2. Destination - pmu
    d. Reorder items in list
        i. FSBL > PMUFW > hello_world
    e. Create BOOT.bin
7. Boot from SD card
    a. Set boot switches to off-on-off-on
    b. Place BOOT.bin on SD and insert into board
    c. Boot board with SW7. This will execute program.
    d. To verify, connect to board over UART, restart board with SW6 to view output

## Phase 2: Attaching Keys and updating PMUFW.

1. Generate AES key for bitstream encryption
    a. Create a .nky file with these contents:

```
Device        xczu1cg;
Key 0         <PASTE_YOUR_64_CHAR_KEY_HERE>;
IV 0          <PASTE_YOUR_24_CHAR_IV_HERE>;
```

    b. Create and replace 64_char_key with output of:
    openssl rand -hex 32
    c. Create and replace 24_char_IV with
    openssl rand -hex 12
2. Generate RSA-4096 Pair
    a. openssl genrsa -out device_key_4096.pem 4096
    b. openssl rsa -in device_key_4096.pem -text -noout > key_dump.txt
3. Copy U96 pmufw custom files into our pmufw src files
    a. xpfw_mod_sec.c
    b. xpfw_mod_sec.h
4. Modify xpfw_mod_sec.c with keys from key_dump.txt
    a. For these, convert “<hex>:<hex>” to “0x<hex>,0x<hex>”
    b. Replace root_sk with privateExponent
    c. Replace root_mod with modulus
    d. Leave root_pk untouched
    e. If first hex value of modulus is 0x00, key may have been padded by openssl. check number of hex values in key. If total is 513, delete the first 0x00 value.
5. Modify xpfw_mod_sec.c further
    a. #include xfpga_config.h is old vitis
        i. comment out
        ii. replace with #include “xilfpga.h”
    b. #define XPAR_XCSUDMA_0_DEVICE_ID 0
    c. Implement sec_load_bistream() fix
6. Modify xpfw_user_startup.c
    a. #include "xpfw_mod_sec.h"
    b. sec_ipi_mod_init();
        i. make that the final line in the file
7. Rebuild pmufw code, create new boot.bin, verify expected print statements from putty.

### *sec_load_bitstream() fix*

the sec_load_bistream() function in the u96 version of xpfw_mod_sec.c is incompatible with modern vitis. Replace that function entirely with this version:

```
static void sec_load_bitstream(){
    u32 status;
    s32 fpga_status;
    u8 bitstream_digest[48];
    u32 msg_buf[8];
    u32 resp_buf[2] = {0};
    u32 bytes_sent = 0;

    // NEW: Structure for 2023.2 XilFPGA API
    XFpga XFpgaInstance = {0};

    if(bitstream_size == 0 || bitstream_addr == NULL){
        XPfw_Printf(DEBUG_ERROR, "PMU: Bitstream address or size error\r\n");
        return;
    }

    // 1. Hash the bitstream
    XCsuDma_Config *csu_config;
    csu_config = XCsuDma_LookupConfig(XPAR_XCSUDMA_0_DEVICE_ID);
    if (csu_config == NULL){
        XPfw_Printf(DEBUG_ERROR, "PMU: Failed to configure CSU\r\n");
        return;
    }
    status = XCsuDma_CfgInitialize(&csu_dma, csu_config, csu_config->BaseAddress);
    if (status != XST_SUCCESS){
        XPfw_Printf(DEBUG_ERROR, "PMU: Failed to initialize CSU\r\n");
        return;
    }
    XSecure_Sha3Initialize(&secure_sha3, &csu_dma);

    // Cast to (u8*) to fix volatile warning
    XSecure_Sha3Digest(&secure_sha3, (u8*)bitstream_addr, bitstream_size, bitstream_digest);

    // 2. Load the Bitstream (FIXED FOR 2023.2 API)
    // Initialize the XilFPGA library
    fpga_status = XFpga_Initialize(&XFpgaInstance);
    if (fpga_status != XST_SUCCESS) {
         XPfw_Printf(DEBUG_ERROR, "PMU: XFpga Init Failed\r\n");
         return;
    }

    /*
     * API: XFpga_BitStream_Load(InstancePtr, Addr, KeyAddr, Size, Flags)
     * KeyAddr = 0 (We are not using an external user key here)
     * Flags = 0 (Implies XFPGA_FULL_BITSTREAM)
     */
    fpga_status = XFpga_BitStream_Load(&XFpgaInstance,
                                       (UINTPTR)bitstream_addr,
                                       0,
                                       bitstream_size,
                                       0);

    if(fpga_status == XFPGA_SUCCESS){
        XPfw_Printf(DEBUG_DETAILED, "PMU: PL Configuration successful\r\n");
    }
    else{
        XPfw_Printf(DEBUG_DETAILED, "PMU: PL Configuration failed %d\r\n", fpga_status);
    }

    // 3. Send Response
    while(bytes_sent < SHA3_SIZE){
        msg_buf[1] = IPI_BITSTREAM_HASH_MASK;
        // Cast to (u8*) to fix volatile warning
        memcpy(&msg_buf[2], &bitstream_digest[bytes_sent], 16);
```

```
        status = XPfw_IpiWriteMessage(sec_ipi_mod_ptr, IPI_PMU_0_IER_RPU_0_MASK, msg_buf, 8);
        if(status != XST_SUCCESS){ return; }

        status = XPfw_IpiTrigger(IPI_PMU_0_IER_RPU_0_MASK);
        if(status != XST_SUCCESS){ return; }

        status = XPfw_IpiPollForAck(IPI_PMU_0_IER_RPU_0_MASK, (~0));
        if(status != XST_SUCCESS){ return; }

        status = XPfw_IpiReadResponse(sec_ipi_mod_ptr, IPI_PMU_0_IER_RPU_0_MASK, resp_buf, 2);
        if(status != XST_SUCCESS){ return; }

        if(resp_buf[1] != IPI_BITSTREAM_HASH_MASK){ return; }
        bytes_sent += 16;
    }
    return;
}
```