Department of Electrical Engineering,
National Tsing Hua University

Advance SoC

Final Project – PQC Falcon

110590022 陳冠晰

110061217 王彥智

110000107 陳柏翰

110020015 劉祐瑋

110011141 陳昇達

GitHub Link of our Project:

https://github.com/vic9112/PQC_Falcon

Table of Contents

1. Introduction

In response to the threat posed by quantum computers to existing cryptographic standards, post-quantum cryptography (PQC) has become a vital research area. The Falcon algorithm stands out in this context for its resistance to quantum attacks.

We aim to enhance the performance by transforming the Falcon algorithm from C code into synthesizable Hardware Description Language (HDL) for hardware acceleration. Initially, we utilized profiling tools to determine the most frequently executed sections of Falcon or where most of the processing time is spent. Consequently, we chose FFT, IFFT, NTT, and INTT as our core functions.

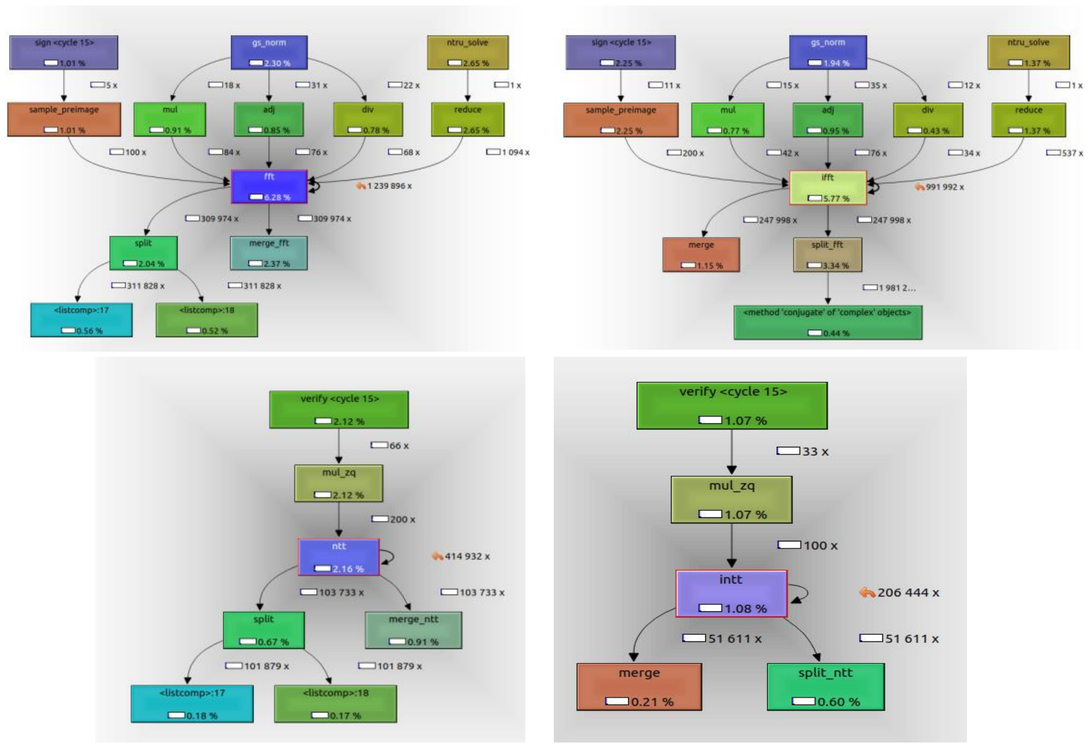
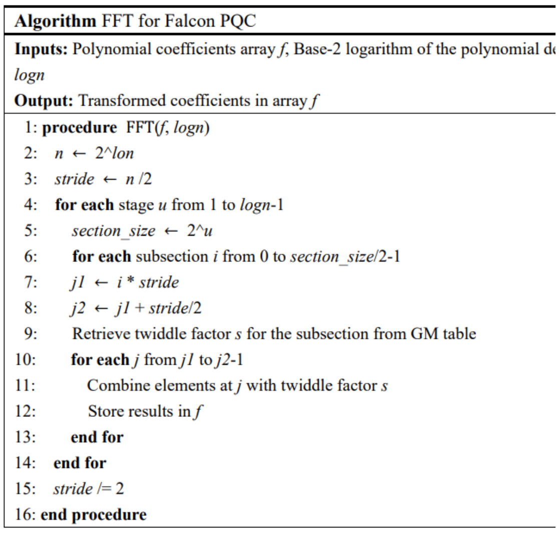


Fig. 1 Callgraph of critical functions in Falcon

2. FPGA Implementation – Vitis HLS

2-1. Initial Implementation on HLS

The iterative radix-2 FFT algorithm with bit-reversal permutation efficiently computes the Fast Fourier Transform, optimizing memory use and processing steps for real-time applications. This in-place algorithm minimizes extra memory. It breaks the FFT into binary stages, handling pairs of data points effectively when the number of data points, N, is a power of two. Over $\log_2 N$ stages, it segments the dataset, applies size-2 DFTs with twiddle factors, and combines results to complete the FFT.



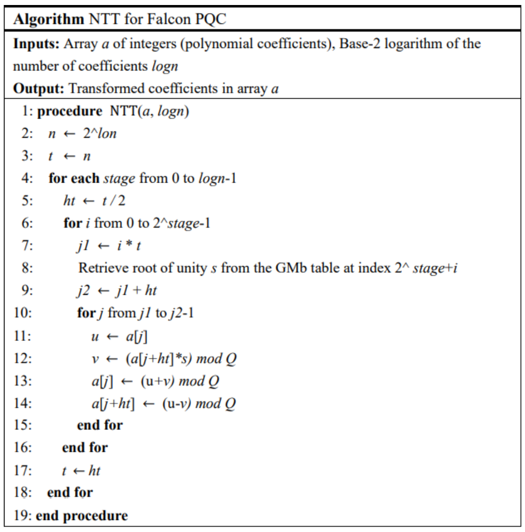
Algorithm FFT for Falcon PQC

Inputs: Polynomial coefficients array *f*, Base-2 logarithm of the polynomial d
logn

Output: Transformed coefficients in array *f*

```
1: procedure FFT(f, logn)
2:    n ← 2^lon
3:    stride ← n /2
4:    for each stage u from 1 to logn-1
5:       section_size ← 2^u
6:       for each subsection i from 0 to section_size/2-1
7:          j1 ← i * stride
8:          j2 ← j1 + stride/2
9:          Retrieve twiddle factor s for the subsection from GM table
10:         for each j from j1 to j2-1
11:            Combine elements at j with twiddle factor s
12:            Store results in f
13:         end for
14:   end for
15:   stride /= 2
16: end procedure
```

Fig. 2-1 Pseudo code of FFT



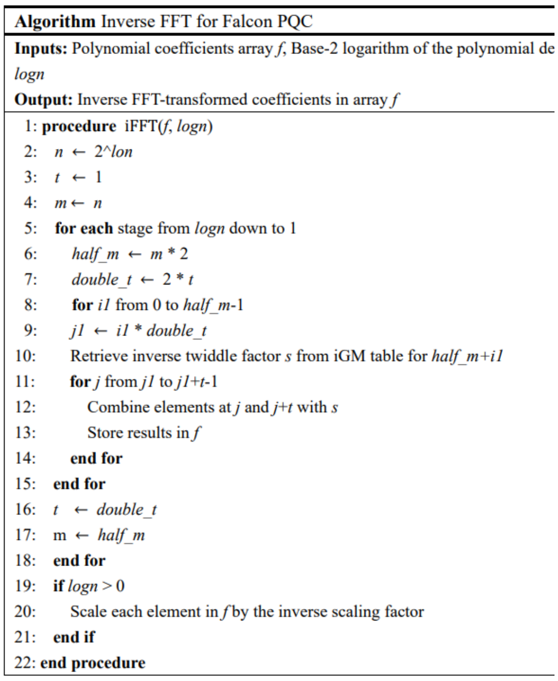
Algorithm NTT for Falcon PQC

Inputs: Array *a* of integers (polynomial coefficients), Base-2 logarithm of the number of coefficients *logn*

Output: Transformed coefficients in array *a*

```
1: procedure NTT(a, logn)
2:    n ← 2^lon
3:    t ← n
4:    for each stage from 0 to logn-1
5:       ht ← t / 2
6:       for i from 0 to 2^stage-1
7:          j1 ← i * t
8:          Retrieve root of unity s from the GMb table at index 2^ stage+i
9:          j2 ← j1 + ht
10:         for j from j1 to j2-1
11:            u ← a[j]
12:            v ← (a[j+ht]*s) mod Q
13:            a[j] ← (u+v) mod Q
14:            a[j+ht] ← (u-v) mod Q
15:         end for
16:      end for
17:      t ← ht
18:   end for
19: end procedure
```

Fig. 2-2 Pseudo code of NTT



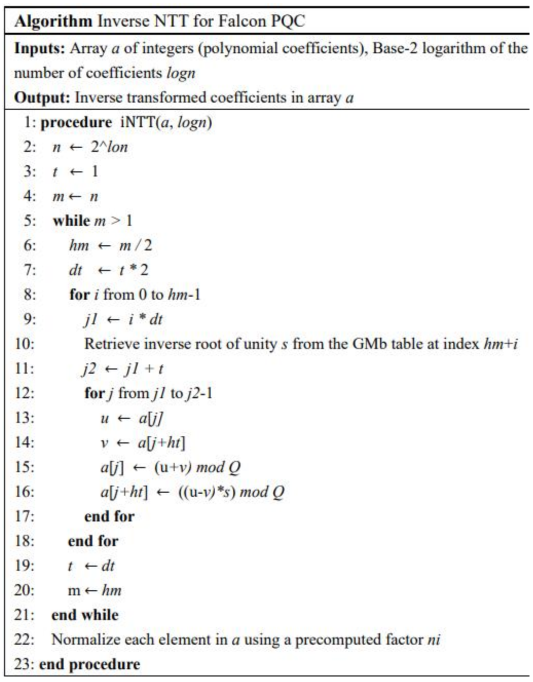
Algorithm Inverse FFT for Falcon PQC

Inputs: Polynomial coefficients array *f*, Base-2 logarithm of the polynomial de
logn

Output: Inverse FFT-transformed coefficients in array *f*

```
1: procedure iFFT(f, logn)
2:    n ← 2^lon
3:    t ← 1
4:    m ← n
5:    for each stage from logn down to 1
6:       half_m ← m * 2
7:       double_t ← 2 * t
8:       for i1 from 0 to half_m-1
9:       j1 ← i1 * double_t
10:      Retrieve inverse twiddle factor s from iGM table for half_m+i1
11:      for j from j1 to j1+t-1
12:         Combine elements at j and j+t with s
13:         Store results in f
14:      end for
15:   end for
16:   t ← double_t
17:   m ← half_m
18:   end for
19:   if logn > 0
20:      Scale each element in f by the inverse scaling factor
21:   end if
22: end procedure
```

Fig. 2-3 Pseudo code of IFFT



Algorithm Inverse NTT for Falcon PQC

Inputs: Array *a* of integers (polynomial coefficients), Base-2 logarithm of the number of coefficients *logn*

Output: Inverse transformed coefficients in array *a*

```
1: procedure iNTT(a, logn)
2:    n ← 2^lon
3:    t ← 1
4:    m ← n
5:    while m > 1
6:       hm ← m / 2
7:       dt ← t * 2
8:       for i from 0 to hm-1
9:          j1 ← i * dt
10:         Retrieve inverse root of unity s from the GMb table at index hm+i
11:         j2 ← j1 + t
12:         for j from j1 to j2-1
13:            u ← a[j]
14:            v ← a[j+ht]
15:            a[j] ← (u+v) mod Q
16:            a[j+ht] ← ((u-v)*s) mod Q
17:         end for
18:      end for
19:      t ← dt
20:      m ← hm
21:   end while
22:   Normalize each element in a using a precomputed factor ni
23: end procedure
```

Fig. 2-4 Pseudo code of INTT

The pseudocode above represents an abstraction of the kernel functions used in our hardware. Initially, we need to convert the original C code into HLS for synthesis, addressing certain non-synthesizable elements in the process. A critical modification involves transforming recursive loops into iterative loops because HLS cannot synthesize loops with undetermined endpoints, which is a major issue.

2-2. Optimization

2-2-1. Synthesizable Implementation

A good architecture will selectively expose and take advantage of parallelism and allow for pipelining. Our final architecture of FFT/iFFT/NTT/iNTT will restructure the code such that each stage is computed in a separate function or module. There will be 10 stages for the butterfly (PE) computations corresponding to the 2-point, 4-point, 8-point, 16-point, … FFT/NTT stages.

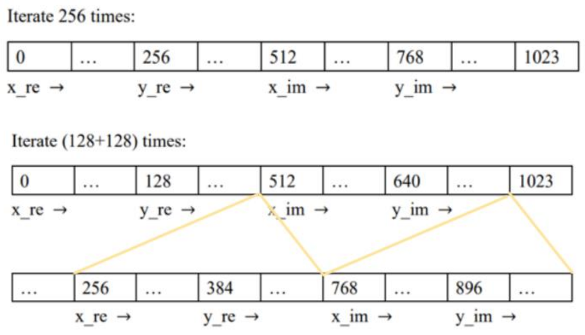


Figure 2-5 Permutation

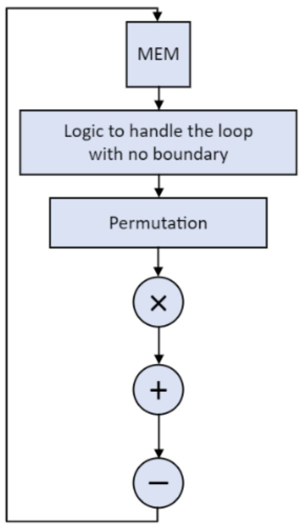


Fig. 2-6 FFT Datapath

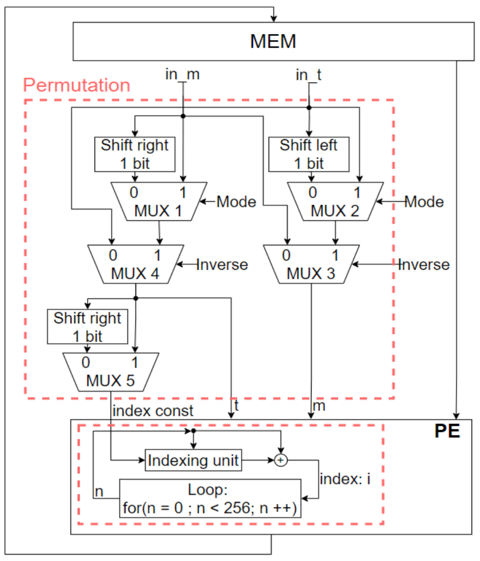


Fig. 2-7 Permutation Logics

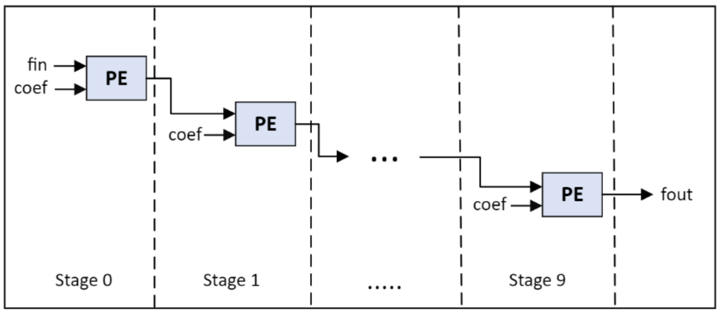
Consider the FFT as an example: initially, we allowed the computation loop boundary to be variable, similar to pseudocode. This approach requires synthesizing logic to handle these boundaries (Fig. 2-6), potentially extending execution time. Therefore, we must address the issue of the unbounded loop structure. As shown in Fig. 2-5 (first stage and second stage) using the indexing from the iterative radix-2 FFT algorithm with bit-reversal permutation, each stage loops 256 times. Therefore, we set the boundary at 256 for each stage. Following this, we outline the indexing for each stage:

・ *i = n + ((n / index_const) * index_const)*
・ *i_gm = n / index_const*

Fig. 2-7 shows our implementation of one-port memory with new custom permutation algorithm logic.

2-2-2. Datapath Restructure

In order to save the use of memory and area, we separate each stage and reuse the processing element (PE). We connect a number of functions (FFT here) in a staged fashion with arrays (fin) acting as buffers between the stages. Fig. 2-8 provides a graphical depiction of this process.



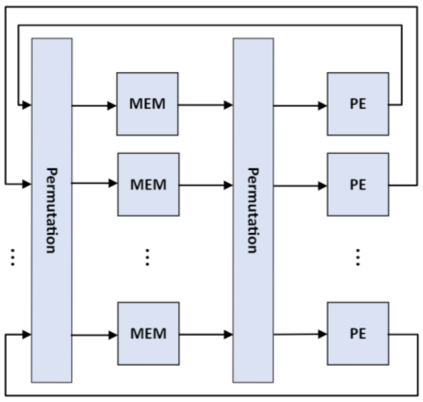


Fig. 2-8 Multiple stage PE Fig. 2-9 Reuse PE with memories

From Fig. 2-9, we can see that FFT algorithm is calculated by loading data from the memories into the PEs and storing the result again in the memories [4]. This process is iterated until the complete FFT is computed. The reuse of the PEs for different stages reduces the number of butterflies. In our project, we use one PE and one in-place memory buffer. With HLS, we can easily extend number of PEs to achieve more parallelism.

2-2-3. Complex Multiplication

In the FFT used by Falcon, floating-point (double precision) is used to represent the complex polynomial coefficients, thus requiring the use of complex multiplication. The multiplication of two complex numbers $(X_r + jX_i)$ and $(Y_r + jY_i)$ is defined as:

$$Z_r = X_r \cdot Y_r - X_i \cdot Y_i$$
$$Z_i = X_r \cdot Y_i + X_i \cdot Y_r$$

where Z_r is the real part of the result and Z_i is its imaginary part.

Fig. 2-10 demonstrates a direct implementation of a complex multiplier, which requires 4 real multipliers and 2 real adders.

Instead of using 4 real multipliers, a structure with 3 real multipliers can be obtained by rewriting as:

$$Z_r = X_r \cdot (Y_r - Y_i) + Y_i \cdot (X_r - X_i)$$
$$Z_i = X_i \cdot (Y_r + Y_i) + Y_i \cdot (X_r - X_i)$$

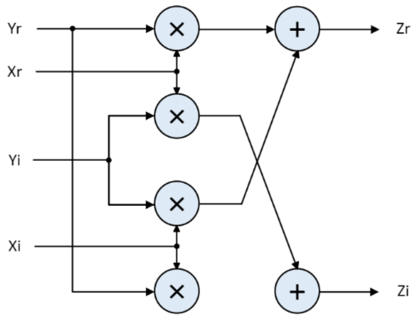


Fig. 2-10 Direct complex multiplier

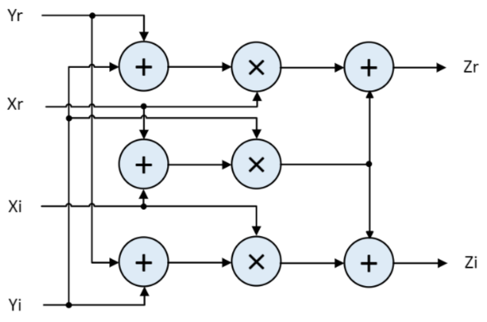


Fig. 2-11 Improved complex multiplier

Fig. 2-11 demonstrates the implementation of a complex multiplier based on it. This structure leads to area reduction although the usage of 5 adders instead of 2 adders needed in the direct implementation, since multipliers require significantly more area than adders.

2-2-4. Combine 4 Algorithms in One Hardware

- FFT & NTT

The following formula shows the algorithm of the forward transformation of DFT and the forward transformation of NTT:

$$\text{DFT}: X[k] = \sum_{i=0}^{n-1} x[i]\, e^{\frac{-j2\pi ik}{n}}, k = 0,1,2,\dots,n-1$$

$$\text{NTT}: \tilde{a}[i] = \sum_{j=0}^{n-1} x[j]\,\omega^{ij} \bmod q, \text{for } i = 0,1,\dots,n-1$$

NTT is a specialized version of the discrete Fourier transform, as we implement FFT, computing DFT in $O(n \log n)$, we can also implement NTT in $O(n \log n)$. On the left side of Fig. 2-12 shows the block diagram of forward FFT/NTT.

Because of those similarities of FFT/NTT, we try to combine them together to possibly share the logic in processing element. Table 2-1 demonstrate the resource usage of re-structured FFT/NTT, and the resource usage after combining them together.

Table 2-1

Comparison of separate FFT/NTT and combined version

Resource	**DSP**	**FF**	**LUT**
FFT	61	8414	11456
NTT	30	4016	7278
FFT_NTT	41	6841	8637

- FFT&iFFT, NTT&iNTT

Let's look at the inverse FFT:

$$x[n] = \frac{1}{N}\sum_{m=0}^{N-1} X[m]\, e^{\frac{j2\pi mn}{N}}$$

We can see that apart from the difference of the index of twiddle factors, it divided by N outside the sigma. In our Falcon project, **N** is defined to 1024, which is a power of 2, can be seen as shifting elements in binary.

Fig. 2-12 shows both the forward/inverse FFT/NTT. We can see that FFT/iFFT have the same operators, just in a different order, we can reuse the previous PE which compute FFT/NTT to share their operators. In the next part, we will explain how to also integrate the division by N outside of the inverse FFT/NTT sigma into the kernel.

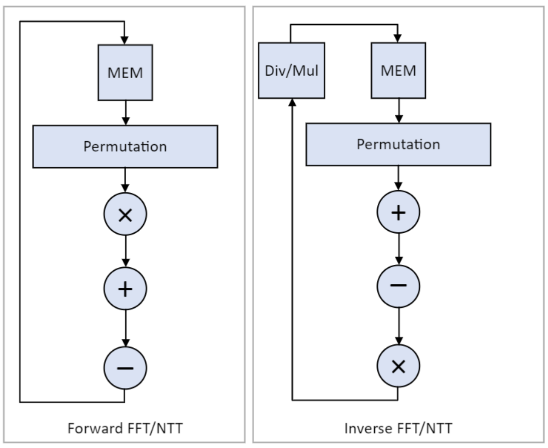


Fig. 2-12 Forward/Inverse FFT/NTT

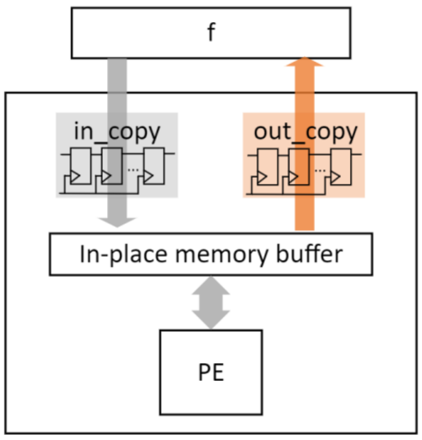


Fig. 2-13 Brief kernel structure

- Output Copy

Since we've allocated an in-place memory buffer in our kernel to calculate and store data, we need to send those data to output after finishing the computation. Fig. 2-13 shows a brief structure about our kernel.

1. *f* is the user input, which expect to return the calculated forward/inverse FFT/NTT.
2. *in_copy* is a module looping N times copying data from f to our memory buffer in-place buffer.
3. The module *out_copy* is crucial since we implement the calculation (dividing by N) here. Because the computation for this part is placed outside sigma, and the module *out_copy* is also activated only after the computation inside sigma is completed, the purpose is to transmit the computations completed within sigma from in-place buffer to f through looping N times. Therefore, we can embed the calculation of dividing by N in the outermost layer of inverse FFT/NTT right here.

2-2-5. Share the Memories

Since FFT utilizes the datatype of 64-bit floating-point(double) and NTT employs unsigned short integer, we initially use two separate buffers to store their respective computational results. We consider using a shared buffer aims to reduce our memory usage. Thus, we use a self-defined datatype **memcell** to save data.

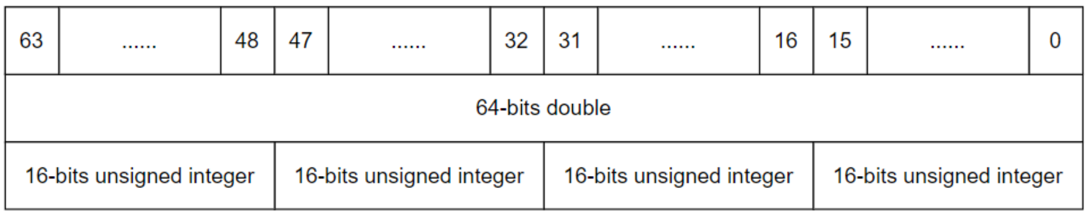


Fig. 2-14 Self defined datatype memcell

Table 2-2

Combine FFT / iFFT / NTT / iNTT (inde: separate buffers, Ver0: combined buffer)

Resource	BRAM	DSP	FF	LUT
fiFFNTT inde	32	79	13333	15919
fiFFNTT Ver0	26	78	11774	13543

From Table 2-2, we can see that the reduction of memories and area is significant. Finally, we combined 4 algorithms (FFT, iFFT, NTT, iNTT), named as **fiFFNTT (forward/inverse Fast-Fourier & Number Theoretic Transform**)

Comparing Table 2-2 and 2-1, we also discover the increasing usage of DSP, since the inverse FFT&NTT need other calculations outside the main computing loop (sigma), which are dividing by N (iFFT) and Montgomery multiplication (iNTT)

Calculating the inverse Fast Fourier Transform (iFFT) with division operations on double type elements significantly increases DSP usage. Later part will solve this issue by applying a double shifter.

2-2-6. Double Shifter & Negation

Based on IEEE-754 double precision, we implement a shifter which can shift the variable with data type double to implement dividing N in iFFT.

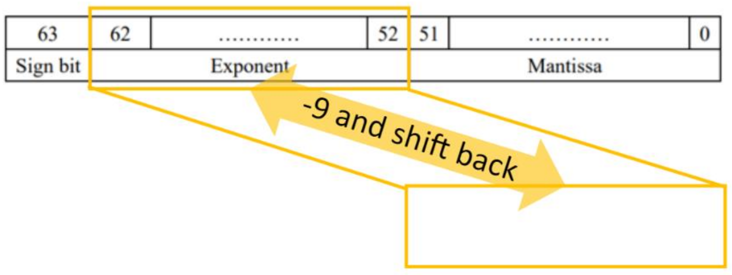


Fig. 2-15 schematic diagram of shifter

Flow:

1. Get the exponent, which located at bit 62 to 52.
2. Subtract 9 to exponent (which is dividing double by 2^9)
3. Shift back to the location from bit 62 to 52.
4. Handle underflow/overflow.

Above statement replaces the usage of DSPs (From Table 2-5, **fiFFNTT_Ver1** decrease 11 DSPs) with just some simple logic.

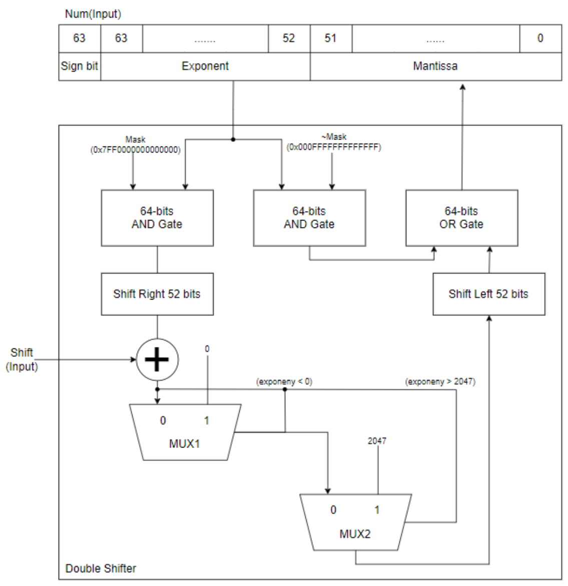


Fig. 2-16 Double Shifter

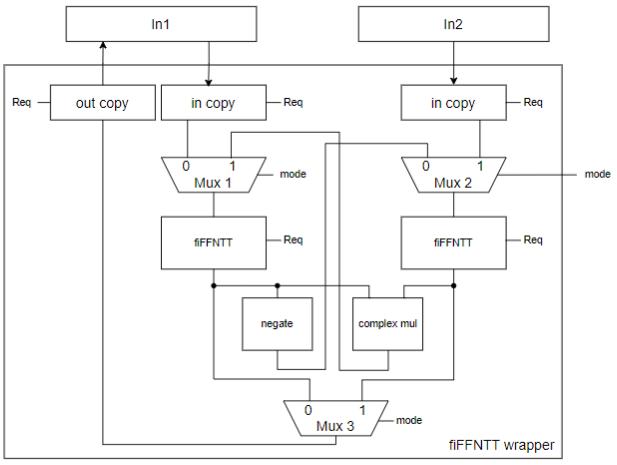


Fig. 2-17 fiFFNTT wrapper

Next, in double subtraction, we share the existing design of an adder plus negation, which can be accomplished by simply toggling the sign bit with a few logic gates, avoiding the need to create an extra subtractor and increase resource usage.

2-2-7. Share the Multiplier

To share the Multiplier, we deconstructed the complex multiplier and Montgomery multiplier in the PE, and we encapsulated the 32-bit integer, 64-bit double adder and multiplier into functions in order to limit the usage using pragma to realize sharing multiplier and adder. (**fiFFNTT_Ver2** in table 2-5)

```
#pragma HLS ALLOCATION function instances=d_mul limit=3
#pragma HLS ALLOCATION function instances=d_add limit=6
#pragma HLS ALLOCATION function instances=u_add limit=2
```

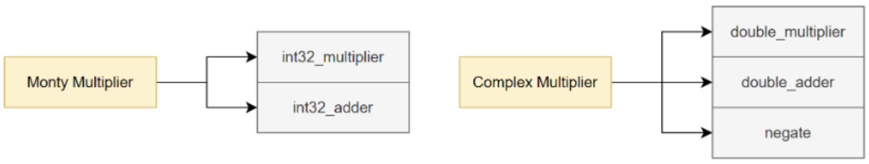
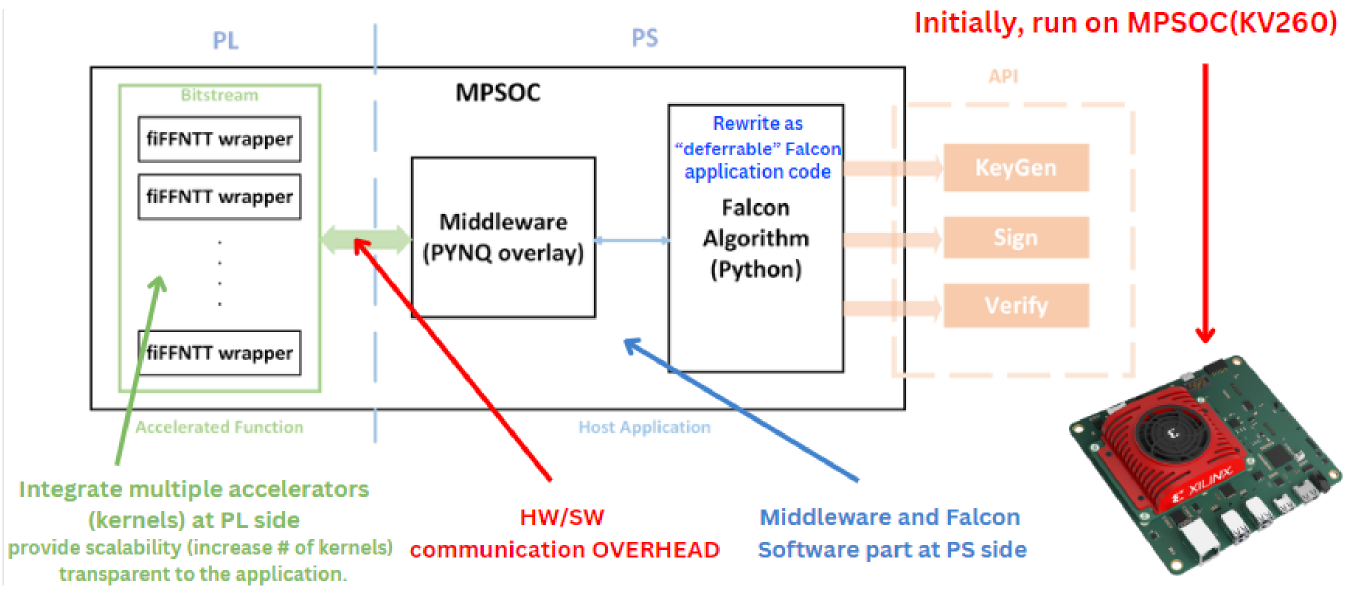


Fig. 2-18

2-3. Middleware



PL
PS
Initially, run on MPSOC(KV260)
MPSOC
API
Bitstream
fiFFNTT wrapper
fiFFNTT wrapper
fiFFNTT wrapper
Rewrite as "deferrable" Falcon application code
Falcon Algorithm (Python)
Middleware (PYNQ overlay)
KeyGen
Sign
Verify
Accelerated Function
Host Application
Integrate multiple accelerators (kernels) at PL side
provide scalability (increase # of kernels) transparent to the application.
HW/SW communication OVERHEAD
Middleware and Falcon Software part at PS side

Figure above is our hardware-software co-design block diagram. We have placed multiple hardware accelerators on the PL side to provide scalability, allowing the number of accelerators to be adjusted according to application requirements. To ensure that the Falcon code does not need to be modified based on the number of hardware accelerators, we have designed a middleware on the PS side. This middleware allocates and controls the hardware kernels and rewrites the Falcon software into deferrable functions.

The thick green double arrow represents the communication overhead between the PS side and the PL side.

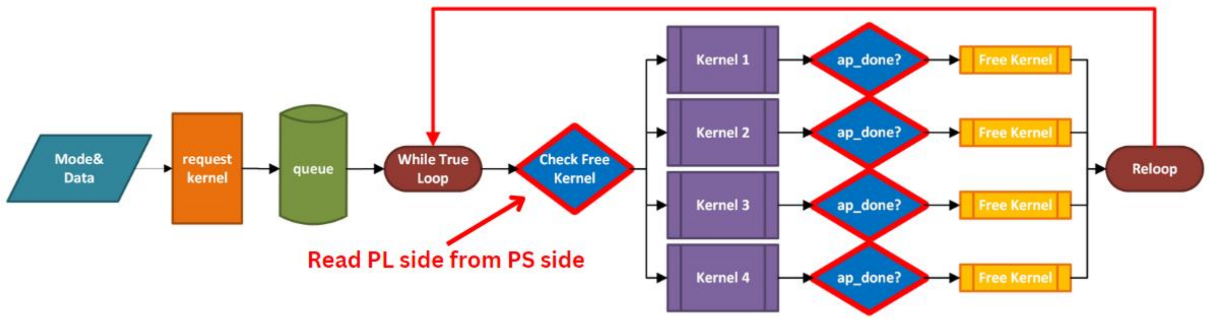
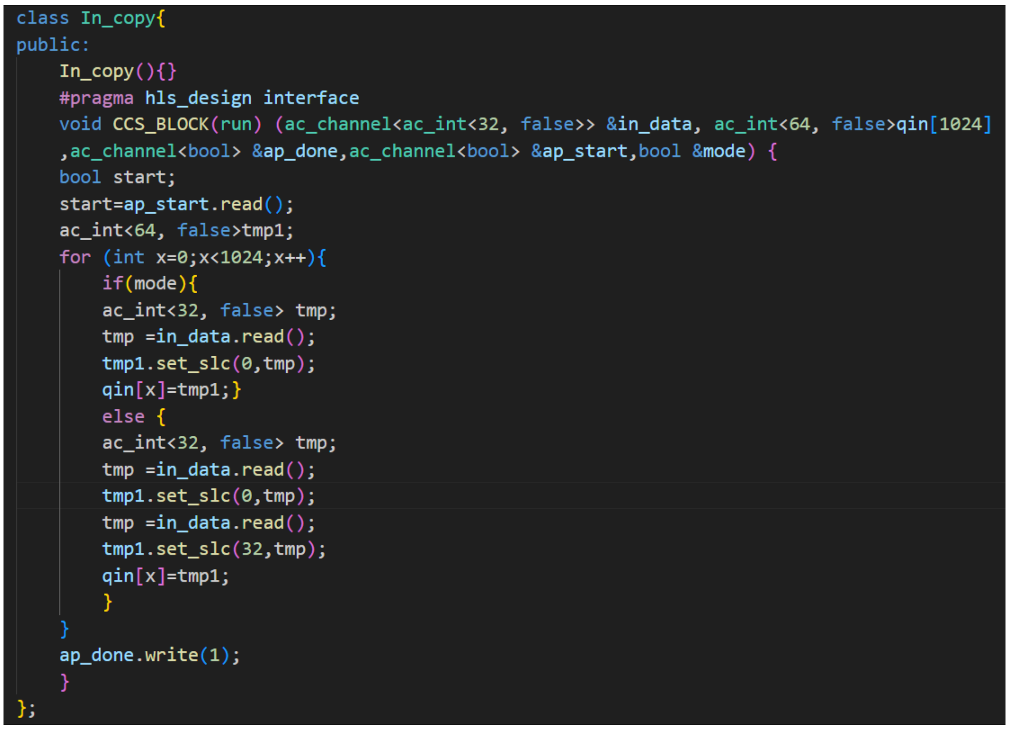


Figure above shows the flow chart of our PS side middleware. We can see there is a while loop, and the area outlined in red indicates where the PS side reads the status of the PL hardware. This section is the cause of the PS-PL overhead. The solution is to move the middleware to the PL side as well.

3. ASIC Implementation – Catapult HLS

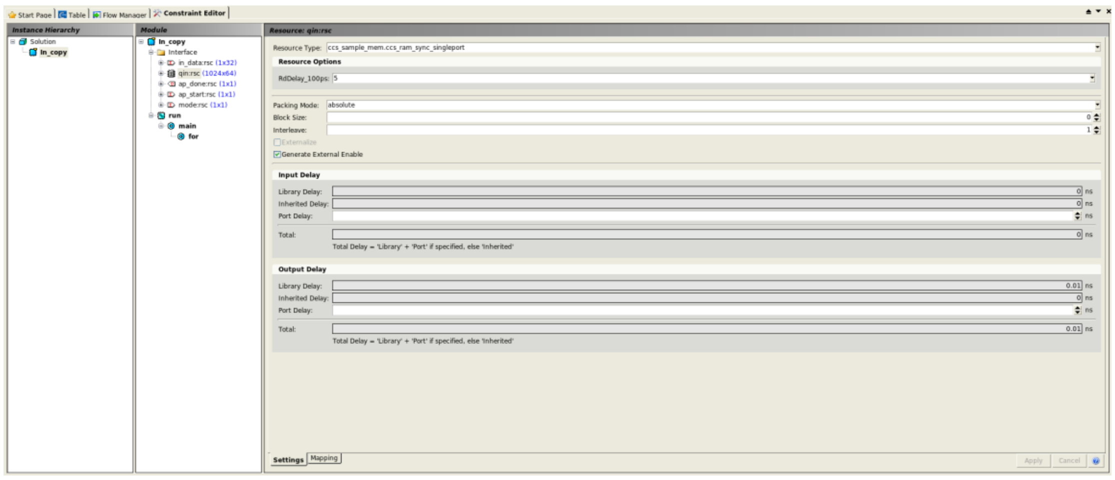
3-1. Module – in_copy

Now we need to convert the Vitis HLS code to Catapult HLS code. The first step is to design an `in_copy` module, changing the original data reading method from using AXI-Master to using ac_channel to transfer data into local SRAM. Since Catapult does not have ap_start and ap_done, we need to add these two signals ourselves to control it.



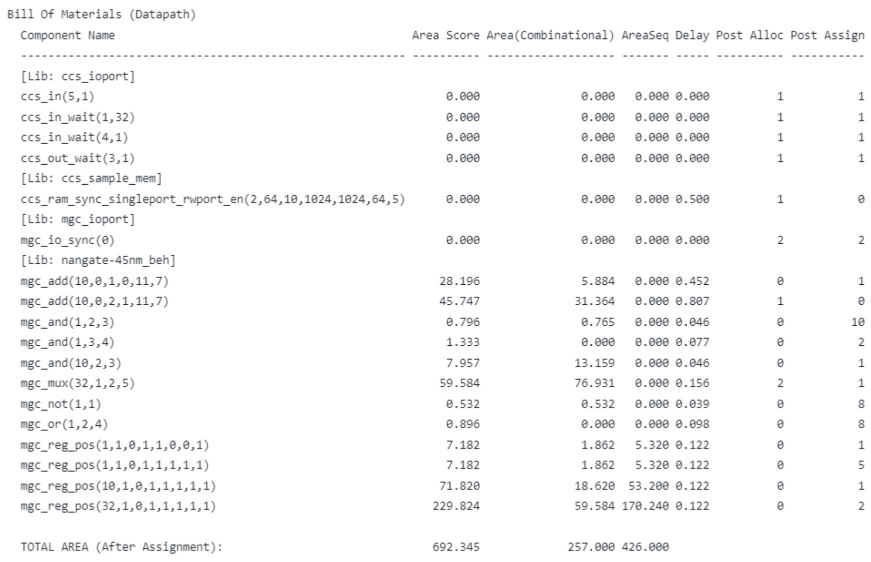
```
class In_copy{
public:
    In_copy(){}
    #pragma hls_design interface
    void CCS_BLOCK(run) (ac_channel<ac_int<32, false>> &in_data, ac_int<64, false>qin[1024]
    ,ac_channel<bool> &ap_done,ac_channel<bool> &ap_start,bool &mode) {
    bool start;
    start=ap_start.read();
    ac_int<64, false>tmp1;
    for (int x=0;x<1024;x++){
        if(mode){
        ac_int<32, false> tmp;
        tmp =in_data.read();
        tmp1.set_slc(0,tmp);
        qin[x]=tmp1;}
        else {
        ac_int<32, false> tmp;
        tmp =in_data.read();
        tmp1.set_slc(0,tmp);
        tmp =in_data.read();
        tmp1.set_slc(32,tmp);
        qin[x]=tmp1;
        }
    }
    ap_done.write(1);
    }
};
```

Additionally, during synthesis, we need to set the SRAM as external.



```
Instance Hierarchy
  Solution
    In_copy

Module
  In_copy
    Interface
      in_data:rsc (1x32)
      qin:rsc (1024x64)
      ap_done:rsc (1x1)
      ap_start:rsc (1x1)
      mode:rsc (1x1)
    run
      main
        for

Resource: qin:rsc
Resource Type: ccs_sample_mem.ccs_ram_sync_singleport
Resource Options
  RdDelay_100ps: 5
Packing Mode: absolute
Block Size: 0
Interleave: 1
[ ] Externalize
[✓] Generate External Enable
Input Delay
  Library Delay: 0 ns
  Inherited Delay: 0 ns
  Port Delay: ns
  Total: 0 ns
  Total Delay = 'Library' + 'Port' if specified, else 'Inherited'
Output Delay
  Library Delay: 0.01 ns
  Inherited Delay: 0 ns
  Port Delay: ns
  Total: 0.01 ns
  Total Delay = 'Library' + 'Port' if specified, else 'Inherited'
```

- **Synthesis result :**



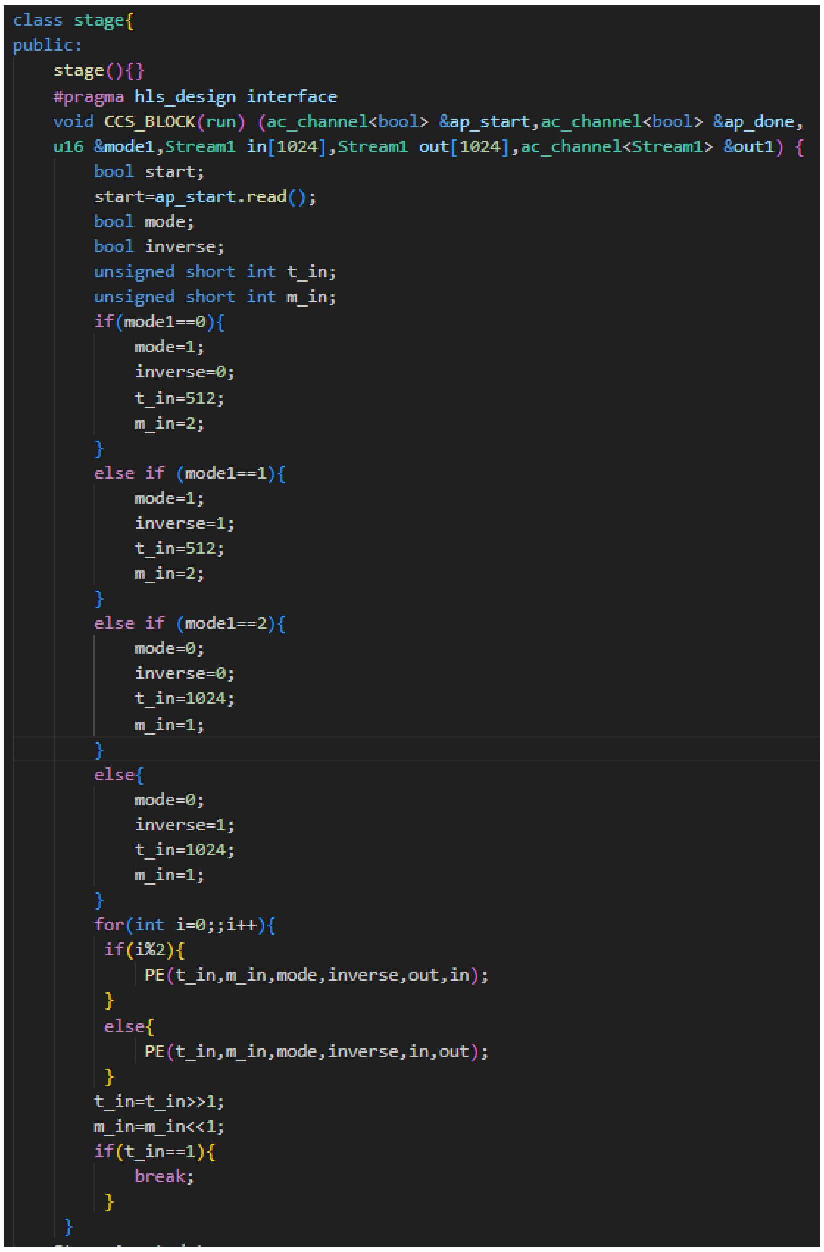
```
Bill Of Materials (Datapath)
  Component Name                                                   Area Score Area(Combinational) AreaSeq Delay Post Alloc Post Assign
  ---------------------------------------------------------------- ---------- ------------------- ------- ----- ---------- -----------
  [Lib: ccs_ioport]
  ccs_in(5,1)                                                           0.000               0.000   0.000 0.000          1           1
  ccs_in_wait(1,32)                                                     0.000               0.000   0.000 0.000          1           1
  ccs_in_wait(4,1)                                                      0.000               0.000   0.000 0.000          1           1
  ccs_out_wait(3,1)                                                     0.000               0.000   0.000 0.000          1           1
  [Lib: ccs_sample_mem]
  ccs_ram_sync_singleport_rwport_en(2,64,10,1024,1024,64,5)             0.000               0.000   0.000 0.500          1           0
  [Lib: mgc_ioport]
  mgc_io_sync(0)                                                        0.000               0.000   0.000 0.000          2           2
  [Lib: nangate-45nm_beh]
  mgc_add(10,0,1,0,11,7)                                               28.196               5.884   0.000 0.452          0           1
  mgc_add(10,0,2,1,11,7)                                               45.747              31.364   0.000 0.807          1           0
  mgc_and(1,2,3)                                                        0.796               0.765   0.000 0.046          0          10
  mgc_and(1,3,4)                                                        1.333               0.000   0.000 0.077          0           2
  mgc_and(10,2,3)                                                       7.957              13.159   0.000 0.046          0           1
  mgc_mux(32,1,2,5)                                                    59.584              76.931   0.000 0.156          2           1
  mgc_not(1,1)                                                          0.532               0.532   0.000 0.039          0           8
  mgc_or(1,2,4)                                                         0.896               0.000   0.000 0.098          0           8
  mgc_reg_pos(1,1,0,1,1,0,0,1)                                          7.182               1.862   5.320 0.122          0           1
  mgc_reg_pos(1,1,0,1,1,1,1,1)                                          7.182               1.862   5.320 0.122          0           5
  mgc_reg_pos(10,1,0,1,1,1,1,1)                                        71.820              18.620  53.200 0.122          0           1
  mgc_reg_pos(32,1,0,1,1,1,1,1)                                       229.824              59.584 170.240 0.122          0           2

  TOTAL AREA (After Assignment):                                      692.345             257.000 426.000
```

https://github.com/vic9112/PQC_Falcon/tree/main/impl_ASIC/fiFFNTT_catapult/In_copy/report

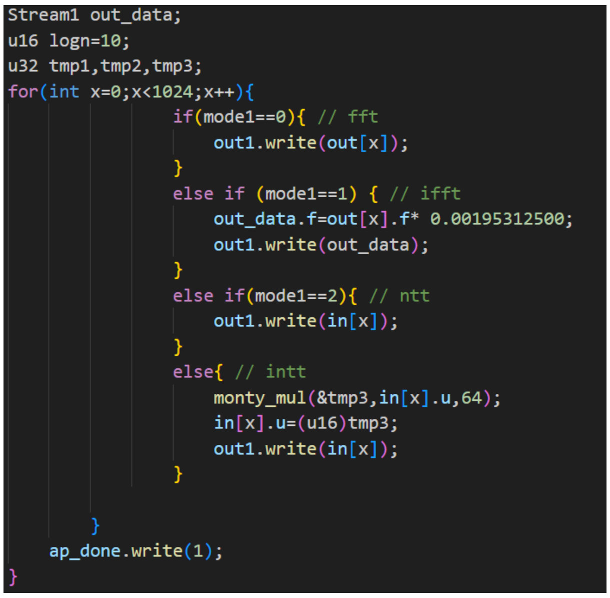
- **Cosimulation:**
  This part is simply copying data, so we do not specifically look at the cosim results. I will directly move on to verify in FSIC.

3-2. Module – out_copy

The second step is to modify the stage module and out_copy module into a coding style that catapult can accept, as follows:



```
class stage{
public:
    stage(){}
    #pragma hls_design interface
    void CCS_BLOCK(run) (ac_channel<bool> &ap_start,ac_channel<bool> &ap_done,
    u16 &mode1,Stream1 in[1024],Stream1 out[1024],ac_channel<Stream1> &out1) {
        bool start;
        start=ap_start.read();
        bool mode;
        bool inverse;
        unsigned short int t_in;
        unsigned short int m_in;
        if(mode1==0){
            mode=1;
            inverse=0;
            t_in=512;
            m_in=2;
        }
        else if (mode1==1){
            mode=1;
            inverse=1;
            t_in=512;
            m_in=2;
        }
        else if (mode1==2){
            mode=0;
            inverse=0;
            t_in=1024;
            m_in=1;
        }
        else{
            mode=0;
            inverse=1;
            t_in=1024;
            m_in=1;
        }
        for(int i=0;;i++){
         if(i%2){
            PE(t_in,m_in,mode,inverse,out,in);
         }
         else{
            PE(t_in,m_in,mode,inverse,in,out);
         }
        t_in=t_in>>1;
        m_in=m_in<<1;
        if(t_in==1){
            break;
         }
     }
```

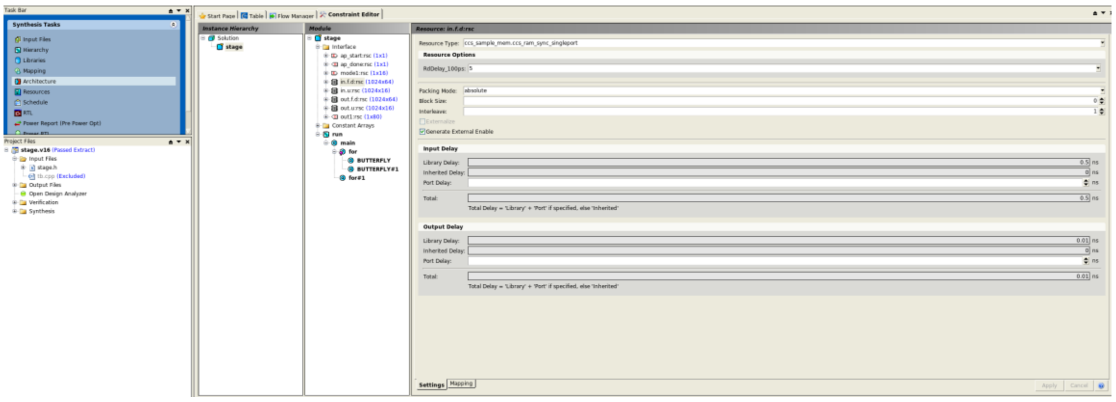


```
Stream1 out_data;
u16 logn=10;
u32 tmp1,tmp2,tmp3;
for(int x=0;x<1024;x++){
                if(mode1==0){ // fft
                    out1.write(out[x]);
                }
                else if (mode1==1) { // ifft
                    out_data.f=out[x].f* 0.00195312500;
                    out1.write(out_data);
                }
                else if(mode1==2){ // ntt
                    out1.write(in[x]);
                }
                else{ // intt
                    monty_mul(&tmp3,in[x].u,64);
                    in[x].u=(u16)tmp3;
                    out1.write(in[x]);
                }

        }
    ap_done.write(1);
}
```

The main change in the diagram above is the datatype of the SRAM. In Vitis HLS, we used a `union` to share 64-bit double and 16-bit uint16, but catapult does not support the `union` syntax. Therefore, we changed it to use a `struct`. However, since the SRAM is external, we can use the wiring part to share the SRAM. Finally, we use ac_channel to write the data from the SRAM.

```
  //typedef   ac_fixed<54,2,true> fpr;
  typedef   ac_ieee_float<binary64> fpr;
  typedef   ac_int<16, false> u16;
  typedef   ac_int<32, false> u32;
struct Stream1{
    fpr f;
    u16 u;
};
```

Figure:struct



Resource: in.f.d:rsc

Resource Type: ccs_sample_mem.ccs_ram_sync_singleport

Resource Options

RdDelay_100ps: 5

Packing Mode: absolute

Block Size: 0

Interleave: 1

☐ Externalize

☑ Generate External Enable

Input Delay

Library Delay: 0.5 ns

Inherited Delay: 0 ns

Port Delay: ns

Total: 0.5 ns

Total Delay = 'Library' + 'Port' if specified, else 'Inherited'

Output Delay

Library Delay: 0.01 ns

Inherited Delay: 0 ns

Port Delay: ns

Total: 0.01 ns

Total Delay = 'Library' + 'Port' if specified, else 'Inherited'

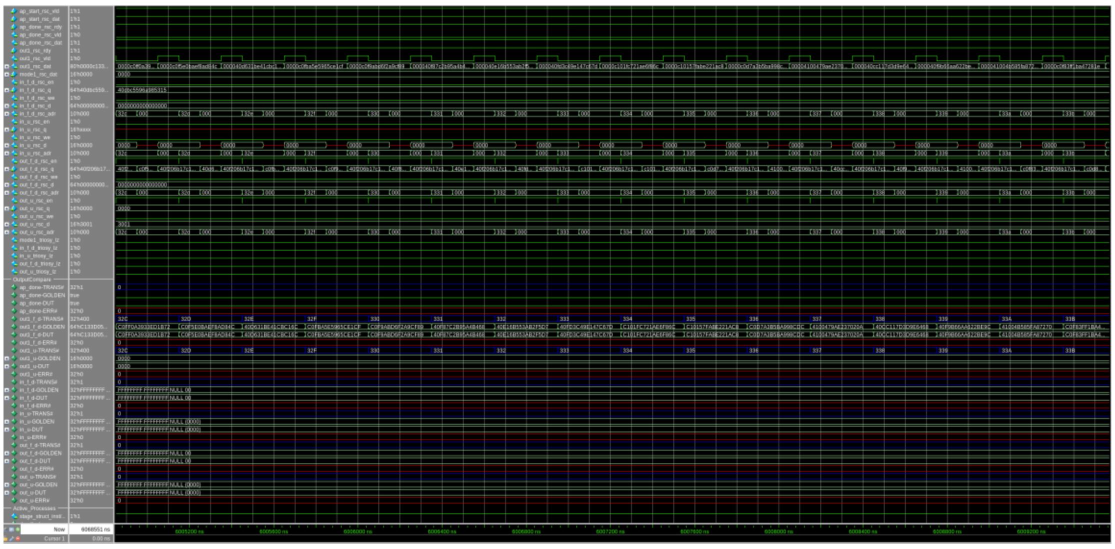
Figure. Setting as externel:

- **Synthesis result:**

```
TOTAL AREA (After Assignment):                          71106.291          65451.000 6187.000
```

https://github.com/vic9112/PQC_Falcon/tree/main/impl_ASIC/fiFFNTT_catapult/Comb_stage_out_copy/report

- **Cosimulation result:**



This meets our desired results.

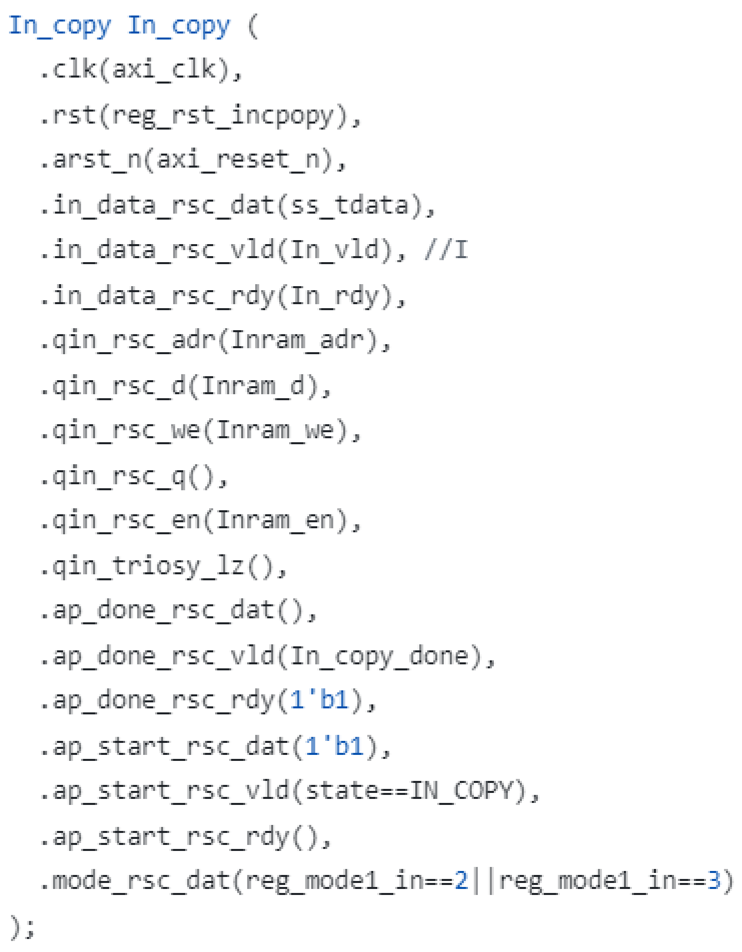
3-3. Wiring User Project

https://github.com/vic9112/PQC_Falcon/blob/main/impl_ASIC/fiFFNTT_sim_v2/vivado/vvd_srcs/caravel_soc/rtl/user/user_subsys/user_prj/user_prj2/rtl/user_prj2.v

First, we need to design a state machine to execute our different modules:

```
always@(*)begin
  case(state)
    Command:
        if(ss_tvalid && ss_tdata[3:2]==2'b01) next_state = IN_COPY;
                else next_state = Command;
    IN_COPY:
        if(In_copy_done) next_state = OUT_COPY;
                else next_state = IN_COPY;// should change
    OUT_COPY:
        if(Out_copy_done) next_state=RESET;
                else next_state=OUT_COPY;
    RESET:
        if(awvalid_in && wvalid_in &&(awaddr[11:0] == 12'h000)&&(wdata[0]==1)) next_state=Command;
        else next_state = RESET;
    default:next_state = Command;
  endcase
end
```

Initially, DMA will send the first piece of data as a command to determine which mode to execute, then enter the IN_COPY state.



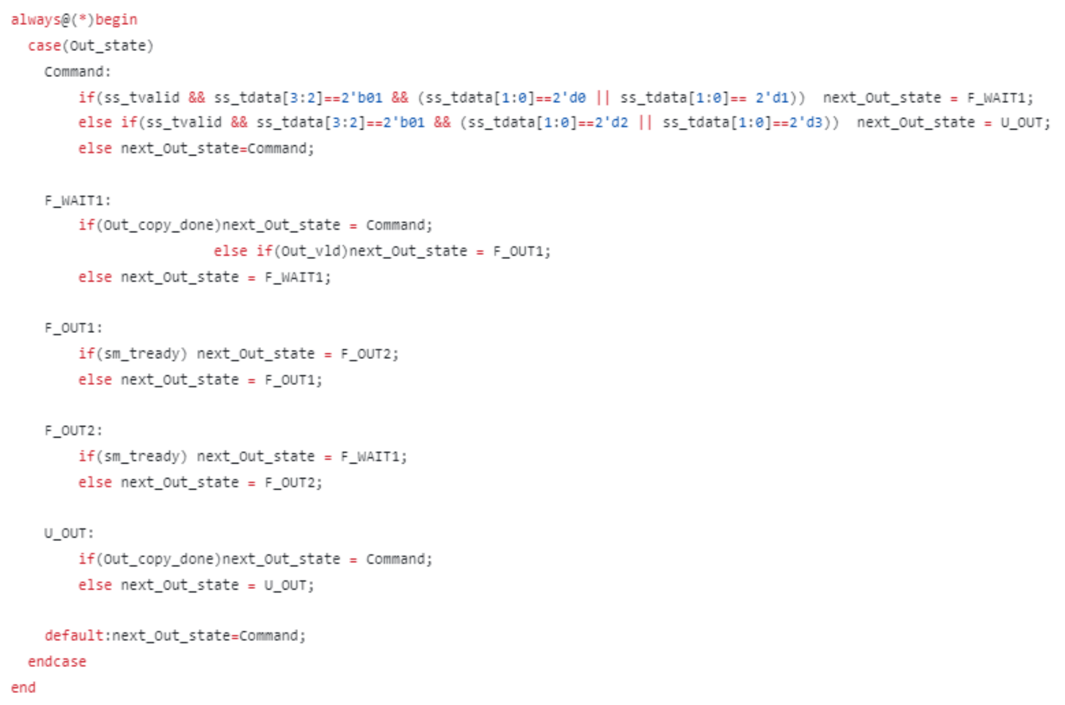
```
In_copy In_copy (
  .clk(axi_clk),
  .rst(reg_rst_incpopy),
  .arst_n(axi_reset_n),
  .in_data_rsc_dat(ss_tdata),
  .in_data_rsc_vld(In_vld), //I
  .in_data_rsc_rdy(In_rdy),
  .qin_rsc_adr(Inram_adr),
  .qin_rsc_d(Inram_d),
  .qin_rsc_we(Inram_we),
  .qin_rsc_q(),
  .qin_rsc_en(Inram_en),
  .qin_triosy_lz(),
  .ap_done_rsc_dat(),
  .ap_done_rsc_vld(In_copy_done),
  .ap_done_rsc_rdy(1'b1),
  .ap_start_rsc_dat(1'b1),
  .ap_start_rsc_vld(state==IN_COPY),
  .ap_start_rsc_rdy(),
  .mode_rsc_dat(reg_mode1_in==2||reg_mode1_in==3)
);
```

The main wiring for the IN_COPY module involves ap_start_vld. When the state is IN_COPY, this kernel will start operating. When ap_done_vld is asserted, the state_machine will transition to the next state. Additionally, the in_data channel is directly connected to ss_stream, while the qin channel is connected to our SRAM.

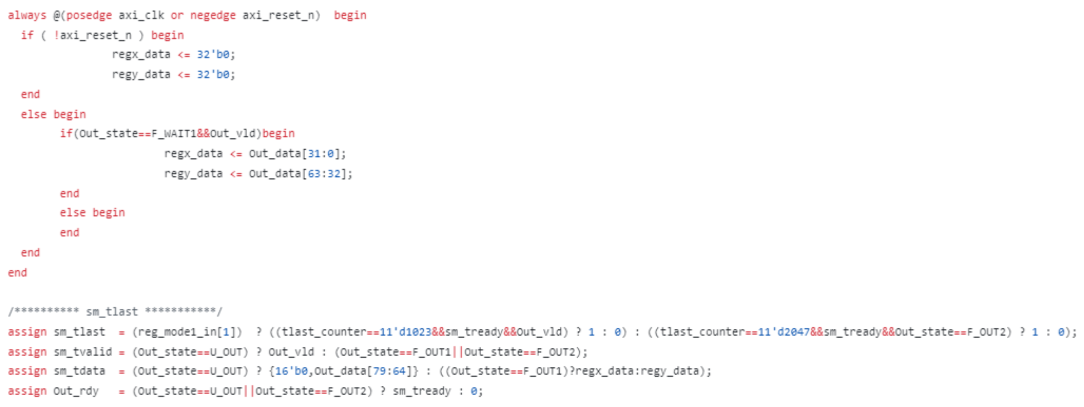
When entering the OUT_COPY state, the fiFFNTT module will execute, and after the stage is completed, it will transition to the OUT_COPY module to sequentially transmit out1_data.

```
fiFFNTT fiFFNTT(
.clk(axi_clk),          // I
.rst(reg_rst),          // I
.arst_n(axi_reset_n),   // I
.ap_start_rsc_dat(1'b1),// I
.ap_start_rsc_vld(state==OUT_COPY),    // I
.ap_start_rsc_rdy(),    // O
.ap_done_rsc_dat(),     // O
.ap_done_rsc_vld(Out_copy_done), // O
.ap_done_rsc_rdy(1'b1),     // I
.mode1_rsc_dat(reg_mode1_in),  //I 16
.mode1_triosy_lz(),
.in_f_d_rsc_adr(in_ramf_adr),  // O 10
.in_f_d_rsc_d(in_ramf_d),    // O 64
.in_f_d_rsc_we(in_ramf_we),   // O 1
.in_f_d_rsc_q(in_ramf_q),    // I 64
.in_f_d_rsc_en(in_ramf_en),   // O 1
.in_f_d_triosy_lz(),
.in_u_rsc_adr(in_ramu_adr), // O 10
.in_u_rsc_d(in_ramu_d),   // O 16
.in_u_rsc_we(in_ramu_we),  // O
.in_u_rsc_q(in_ramu_q),   // I 16
.in_u_rsc_en(in_ramu_en),  // O
.in_u_triosy_lz(),
.out_f_d_rsc_adr(out_ramf_adr),
.out_f_d_rsc_d(out_ramf_d),
.out_f_d_rsc_we(out_ramf_we),
.out_f_d_rsc_q(out_ramf_q),
.out_f_d_rsc_en(out_ramf_en),
.out_f_d_triosy_lz(),
.out_u_rsc_adr(out_ramu_adr),
.out_u_rsc_d(out_ramu_d),
.out_u_rsc_we(out_ramu_we),
.out_u_rsc_q(out_ramu_q),
.out_u_rsc_en(out_ramu_en),
.out_u_triosy_lz(),
.out1_rsc_dat(Out_data),//O,80 bit{16'b,64'b},
.out1_rsc_vld(Out_vld),//O;
.out1_rsc_rdy(Out_rdy)
);
```

The fiFFNTT module primarily involves the connection of two SRAMs (actually four SRAMs, with two being 16-bit and two being 64-bit, but we will explain how to share them later). Additionally, there is a special case where the out1 channel is actually 80-bit data. Since sm_stream can only transmit 32-bit data, if we are in FFT or IFFT mode, we need to transmit it in two parts. However, if we are in NTT or INTT mode, it can directly connect to sm_stream. Finally, when all the data has been transmitted, ap_done_vld will be asserted, allowing the state machine to transition to the RESET state.



```
always@(*)begin
  case(Out_state)
    Command:
        if(ss_tvalid && ss_tdata[3:2]==2'b01 && (ss_tdata[1:0]==2'd0 || ss_tdata[1:0]== 2'd1))  next_Out_state = F_WAIT1;
        else if(ss_tvalid && ss_tdata[3:2]==2'b01 && (ss_tdata[1:0]==2'd2 || ss_tdata[1:0]==2'd3))  next_Out_state = U_OUT;
        else next_Out_state=Command;

    F_WAIT1:
        if(Out_copy_done)next_Out_state = Command;
                        else if(Out_vld)next_Out_state = F_OUT1;
        else next_Out_state = F_WAIT1;

    F_OUT1:
        if(sm_tready) next_Out_state = F_OUT2;
        else next_Out_state = F_OUT1;

    F_OUT2:
        if(sm_tready) next_Out_state = F_WAIT1;
        else next_Out_state = F_OUT2;

    U_OUT:
        if(Out_copy_done)next_Out_state = Command;
        else next_Out_state = U_OUT;

    default:next_Out_state=Command;
  endcase
end
```



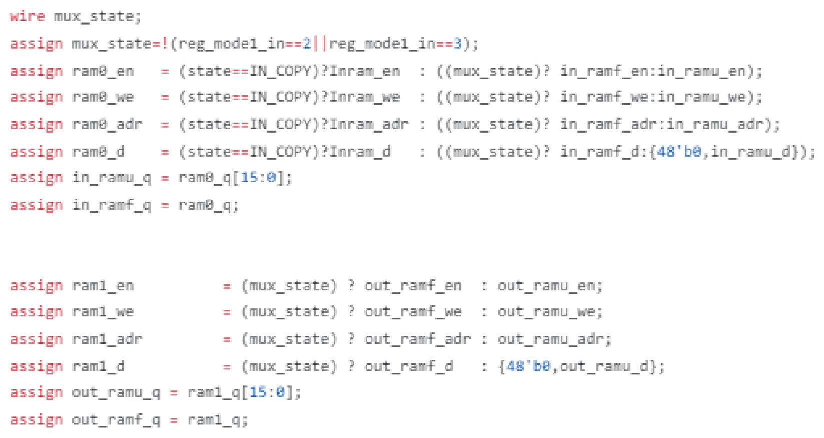
```
always @(posedge axi_clk or negedge axi_reset_n)  begin
  if ( !axi_reset_n ) begin
              regx_data <= 32'b0;
              regy_data <= 32'b0;
  end
  else begin
        if(Out_state==F_WAIT1&&Out_vld)begin
                    regx_data <= Out_data[31:0];
                    regy_data <= Out_data[63:32];
        end
        else begin
        end
  end
end

/********** sm_tlast ***********/
assign sm_tlast  = (reg_mode1_in[1])  ? ((tlast_counter==11'd1023&&sm_tready&&Out_vld) ? 1 : 0) : ((tlast_counter==11'd2047&&sm_tready&&Out_state==F_OUT2) ? 1 : 0);
assign sm_tvalid = (Out_state==U_OUT) ? Out_vld : (Out_state==F_OUT1||Out_state==F_OUT2);
assign sm_tdata  = (Out_state==U_OUT) ? {16'b0,Out_data[79:64]} : ((Out_state==F_OUT1)?regx_data:regy_data);
assign Out_rdy   = (Out_state==U_OUT||Out_state==F_OUT2) ? sm_tready : 0;
```

Finally, here is a brief introduction on how to connect the two 64-bit, 1024-depth SRAMs:

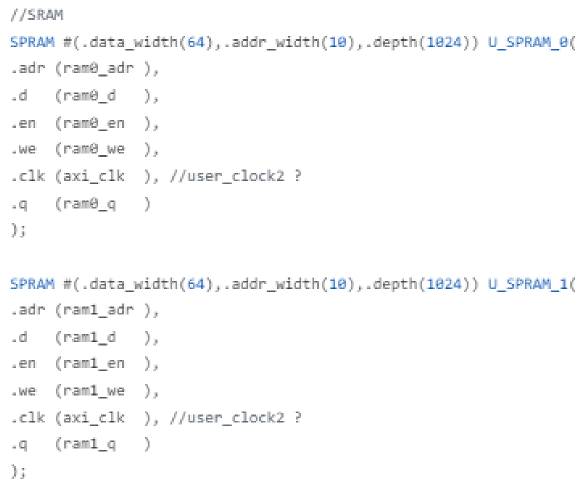
- **Connection of SRAMs**:
  - Each of the two 64-bit SRAMs will be connected to the respective data channels of the fiFFNTT module.
  - These SRAMs will be utilized for storing and retrieving data during the execution of the FFT/IFFT/NTT/INTT processes.
- **Shared Access**:
  - To enable shared access, we will use a struct to define the data types, allowing both 16-bit and 64-bit accesses within the same memory space.

- Proper control signals will be implemented to manage the access, ensuring that the correct portions of the SRAM are read or written as needed.

- **Handling of Out1 Channel**:
  - The out1 channel, which is 80-bit, will need special handling. For FFT/IFFT modes, the 80-bit data will be split and transmitted in two 32-bit parts due to the limitations of sm_stream.
  - For NTT/INTT modes, the out1 channel will directly connect to the sm_stream without splitting, as the data structure is compatible.



```
wire mux_state;
assign mux_state=!(reg_mode1_in==2||reg_mode1_in==3);
assign ram0_en   = (state==IN_COPY)?Inram_en  : ((mux_state)? in_ramf_en:in_ramu_en);
assign ram0_we   = (state==IN_COPY)?Inram_we  : ((mux_state)? in_ramf_we:in_ramu_we);
assign ram0_adr  = (state==IN_COPY)?Inram_adr : ((mux_state)? in_ramf_adr:in_ramu_adr);
assign ram0_d    = (state==IN_COPY)?Inram_d   : ((mux_state)? in_ramf_d:{48'b0,in_ramu_d});
assign in_ramu_q = ram0_q[15:0];
assign in_ramf_q = ram0_q;


assign ram1_en          = (mux_state) ? out_ramf_en  : out_ramu_en;
assign ram1_we          = (mux_state) ? out_ramf_we  : out_ramu_we;
assign ram1_adr         = (mux_state) ? out_ramf_adr : out_ramu_adr;
assign ram1_d           = (mux_state) ? out_ramf_d   : {48'b0,out_ramu_d};
assign out_ramu_q = ram1_q[15:0];
assign out_ramf_q = ram1_q;
```

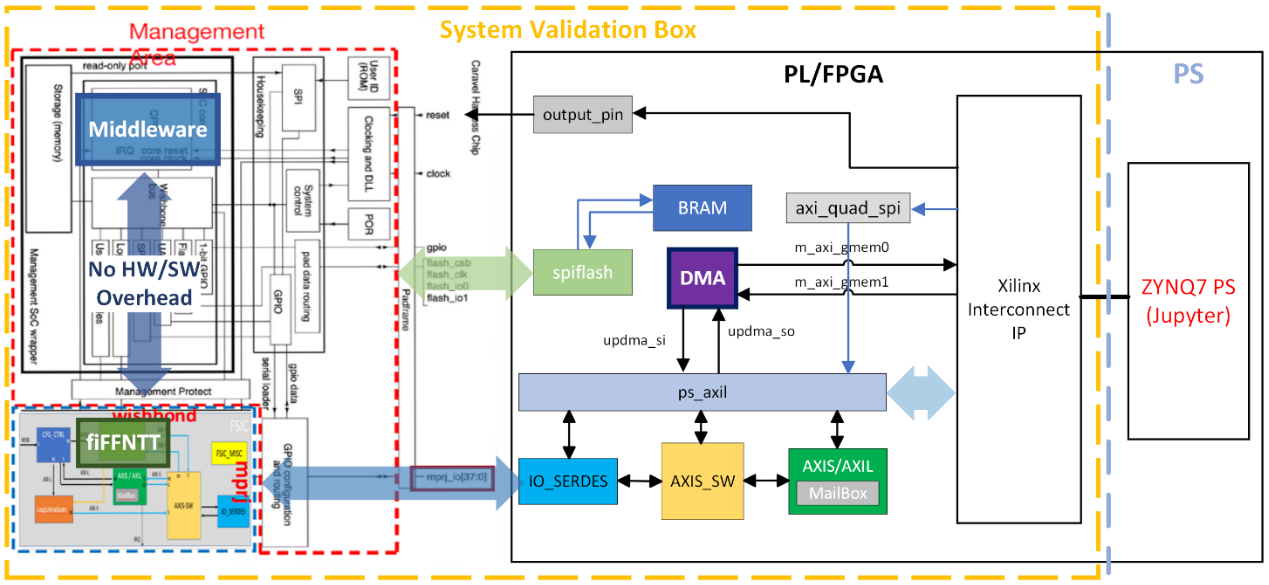


```
//SRAM
SPRAM #(.data_width(64),.addr_width(10),.depth(1024)) U_SPRAM_0(
.adr (ram0_adr ),
.d   (ram0_d   ),
.en  (ram0_en  ),
.we  (ram0_we  ),
.clk (axi_clk  ), //user_clock2 ?
.q   (ram0_q   )
);

SPRAM #(.data_width(64),.addr_width(10),.depth(1024)) U_SPRAM_1(
.adr (ram1_adr ),
.d   (ram1_d   ),
.en  (ram1_en  ),
.we  (ram1_we  ),
.clk (axi_clk  ), //user_clock2 ?
.q   (ram1_q   )
);
```

Ram0 will first switch to the qin channel in the In_copy module during the IN_COPY state. In the OUT_COPY state, RAM0 and RAM1 will determine the mode to decide whether to connect to the u channel or the f channel in the fiFFNTT module.

By setting up the SRAMs in this manner, we ensure efficient data handling and compatibility with the different modes of operation within the fiFFNTT module. This setup allows for seamless transitions between states and accurate data processing.

Figure below shows the current block diagram of our project:

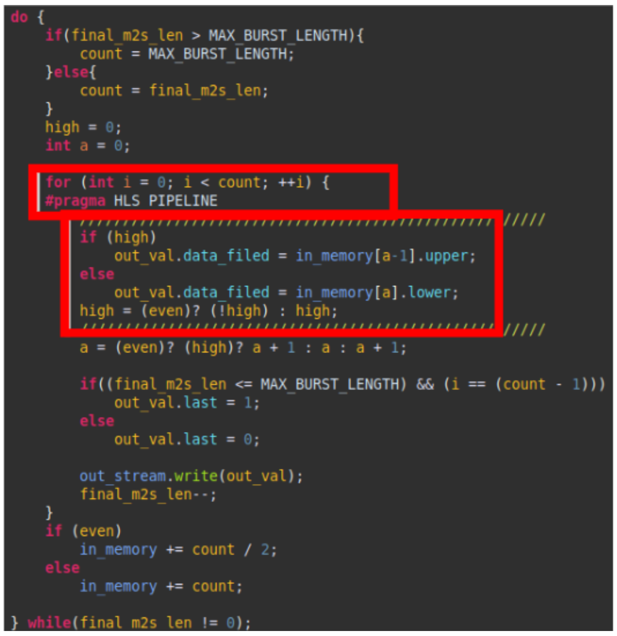


The section outlined in yellow is the system validation box. On the SOC side, we can see that we have integrated the accelerator and FSIC into the SOC user project. The middleware is also written as firmware and loaded onto the RISC-V CPU. Since everything is on the PL side, it avoids the previous PS-PL communication overhead.

On the FPGA side, we have also placed multiple Ips and designed a DMA to handle the transfer of data that the kernel needs to process. This is done through the AXI-Master, which accesses the data from the PS side's DDR memory via the interconnect, and then transfers the data to the SOC side's kernel via AXI-Stream.

4. DMA

- SPEC
  - FFT/iFFT **m2s** (DMA stream out 2049 data)
    - First data: kernel_mode
    - 2048 data: upper 32-bit and lower 32-bit of 1024 “double” data
  - NTT/iNTT **m2s** (DMA stream out 1025 data)
    - First data: kernel_mode
    - 1024 data: 16-bit of 1024 “uint_16’ data
  - FFT/iFFT **s2m** (stream in 2048 data)
    - upper 32-bit and lower 32-bit of 1024 “double” data
  - NTT/iNTT **s2m** (stream in 1024 data)
    - 1024 16-bit “uint_16” data
- AXIM to AXIS (System Memory to Device)
  - Parallel to stream (burst reading from memory)

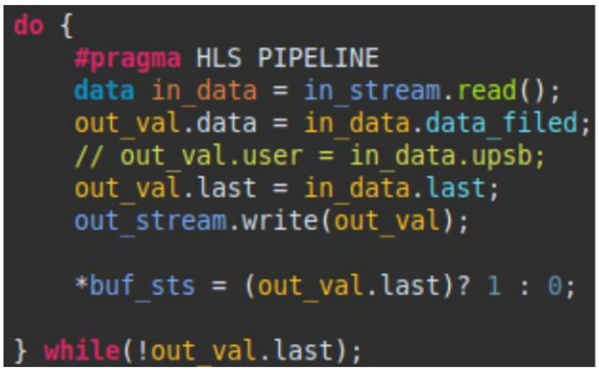


```
do {
    if(final_m2s_len > MAX_BURST_LENGTH){
        count = MAX_BURST_LENGTH;
    }else{
        count = final_m2s_len;
    }
    high = 0;
    int a = 0;

    for (int i = 0; i < count; ++i) {
    #pragma HLS PIPELINE
        /////////////////////////////////////////////////////////
        if (high)
            out_val.data_filed = in_memory[a-1].upper;
        else
            out_val.data_filed = in_memory[a].lower;
        high = (even)? (!high) : high;
        /////////////////////////////////////////////////////////
        a = (even)? (high)? a + 1 : a : a + 1;

        if((final_m2s_len <= MAX_BURST_LENGTH) && (i == (count - 1)))
            out_val.last = 1;
        else
            out_val.last = 0;

        out_stream.write(out_val);
        final_m2s_len--;
    }
    if (even)
        in_memory += count / 2;
    else
        in_memory += count;

} while(final_m2s_len != 0);
```

for-loop indicates the burst cycles

read upper/lower bit from memory

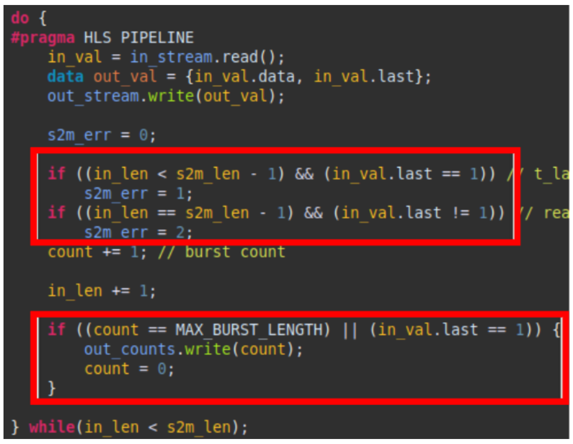
  - Stream-out



```
do {
    #pragma HLS PIPELINE
    data in_data = in_stream.read();
    out_val.data = in_data.data_filed;
    // out_val.user = in_data.upsb;
    out_val.last = in_data.last;
    out_stream.write(out_val);

    *buf_sts = (out_val.last)? 1 : 0;

} while(!out_val.last);
```

directly stream-out

- AXIS to AXIM (Device to System Memory)
  - Get in-stream from SOC side



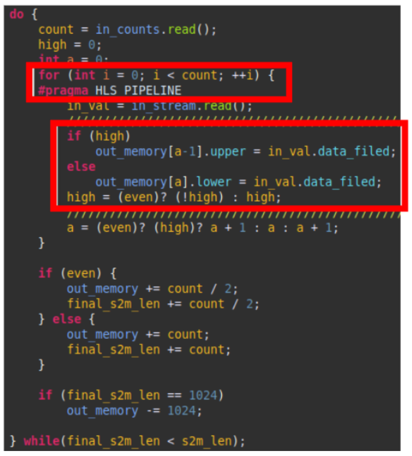
```
do {
#pragma HLS PIPELINE
    in_val = in_stream.read();
    data out_val = {in_val.data, in_val.last};
    out_stream.write(out_val);

    s2m_err = 0;

    if ((in_len < s2m_len - 1) && (in_val.last == 1)) // t_la
        s2m_err = 1;
    if ((in_len == s2m_len - 1) && (in_val.last != 1)) // rea
        s2m_err = 2;
    count += 1; // burst count

    in_len += 1;

    if ((count == MAX_BURST_LENGTH) || (in_val.last == 1)) {
        out_counts.write(count);
        count = 0;
    }

} while(in_len < s2m_len);
```

tlast asserted but DMA hasn't reach stream length

reach stream length but tlast not asserted

Count to burst length (512)
if received tlast, directly write into memory
no matter the count

  - Stream to parallel (write to memory with burst)

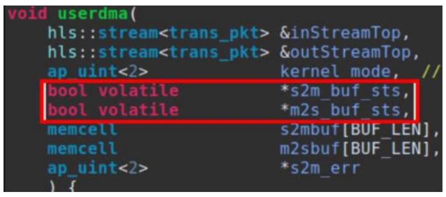


```
do {
    count = in_counts.read();
    high = 0;
    int a = 0;
    for (int i = 0; i < count; ++i) {
    #pragma HLS PIPELINE
        in_val = in_stream.read();
        ////////////////////////////////////////////
        if (high)
            out_memory[a-1].upper = in_val.data_filed;
        else
            out_memory[a].lower = in_val.data_filed;
        high = (even)? (!high) : high;
        ////////////////////////////////////////////
        a = (even)? (high)? a + 1 : a : a + 1;
    }

    if (even) {
        out_memory += count / 2;
        final_s2m_len += count / 2;
    } else {
        out_memory += count;
        final_s2m_len += count;
    }

    if (final_s2m_len == 1024)
        out_memory -= 1024;

} while(final_s2m_len < s2m_len);
```

for-loop indicates the burst cycles

write upper/lower bit to memory

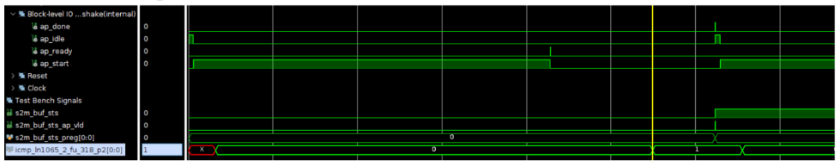
- Continuously read/update DMA status



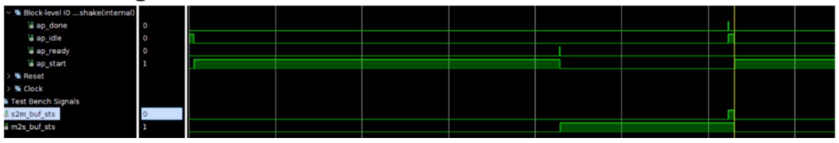
```
void userdma(
    hls::stream<trans_pkt> &inStreamTop,
    hls::stream<trans_pkt> &outStreamTop,
    ap_uint<2>             kernel_mode,  //
    bool volatile          *s2m_buf_sts,
    bool volatile          *m2s_buf_sts,
    memcell                s2mbuf[BUF_LEN],
    memcell                m2sbuf[BUF_LEN],
    ap_uint<2>             *s2m_err
    ) {
```

The keyword "**volatile**" informs the compiler that a variable's value may change at any time.

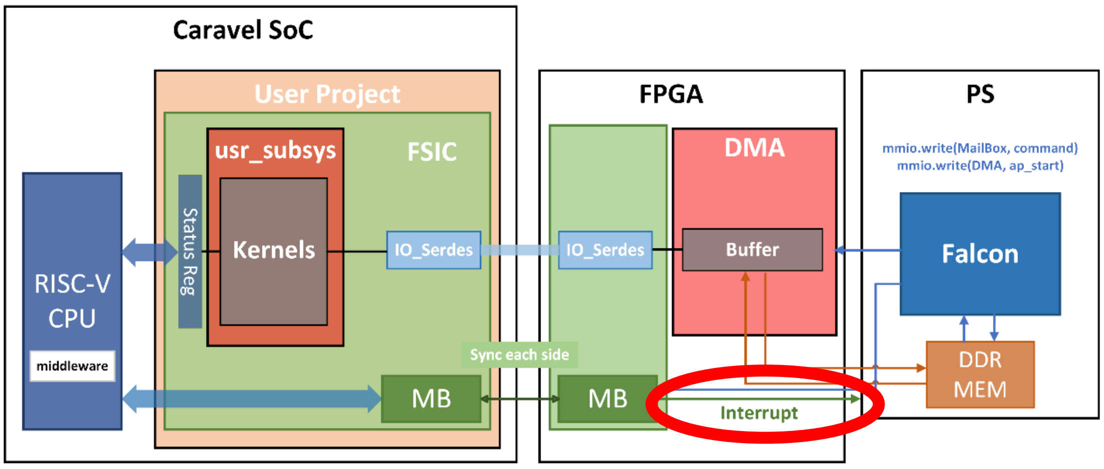
Before setting "volatile"



After setting "volatile"



5. Interrupt



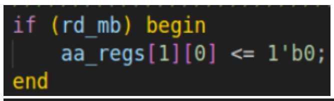
In our design, we put our kernel in user project Caravel SoC. When kernel finishing the task, we change the status register. The change in status register will trigger the middleware in RISC-V CPU to write the mailbox, which change the mailbox in FPGA correspondingly. After mailbox is written, it generates interrupt to the PS side.

To achieve this implementation, we add an interrupt controller in our original design. The interrupt is level trigger since we tried edge trigger but couldn't get the interrupt.

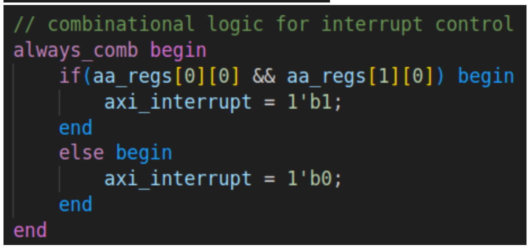
Also, we changed the design in our mailbox slightly. However, we can't have the result we expect to get.

5-1. Improving Mailbox Design

Initially, the status of interrupt (IRQ) can't be reset or de-asserted. Therefore, we improved the design of mailbox (axi_ctrl_logic.sv), such that the interrupt status will be de-asserted after a read cycle to mailbox registers.



```
if (rd_mb) begin
    aa_regs[1][0] <= 1'b0;
end
```

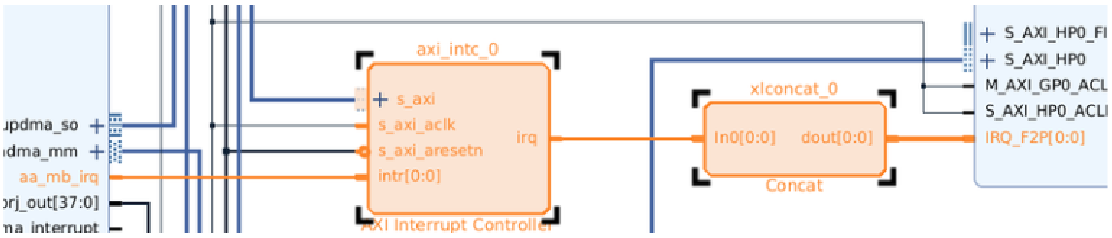


```
// combinational logic for interrupt control
always_comb begin
    if(aa_regs[0][0] && aa_regs[1][0]) begin
        axi_interrupt = 1'b1;
    end
    else begin
        axi_interrupt = 1'b0;
    end
end
```

aa_regs[0][0] stands for irq_enable
aa_regs[1][0] stands for irq trigger

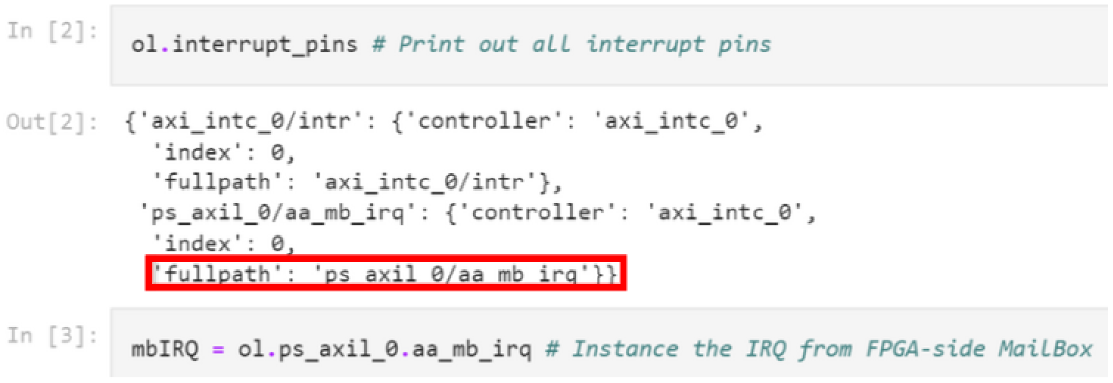
5-2. Interrupt Controller

Add AXI interrupt controller, connect pin aa_mb_irq on ps_axil to the input of AXI interrupt controller, then concatenate to pin IRQ_F2P on ZYNQ processor.



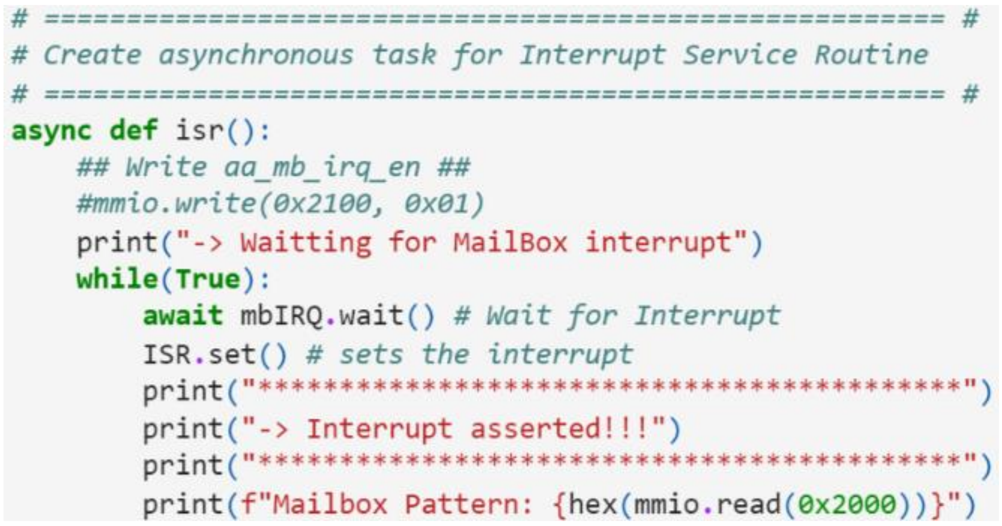
5-3. Interrupt Service Routine

- Check interrupt pins and instance it



```
In [2]:  ol.interrupt_pins # Print out all interrupt pins

Out[2]: {'axi_intc_0/intr': {'controller': 'axi_intc_0',
          'index': 0,
          'fullpath': 'axi_intc_0/intr'},
         'ps_axil_0/aa_mb_irq': {'controller': 'axi_intc_0',
          'index': 0,
          'fullpath': 'ps_axil_0/aa_mb_irq'}}

In [3]:  mbIRQ = ol.ps_axil_0.aa_mb_irq # Instance the IRQ from FPGA-side MailBox
```

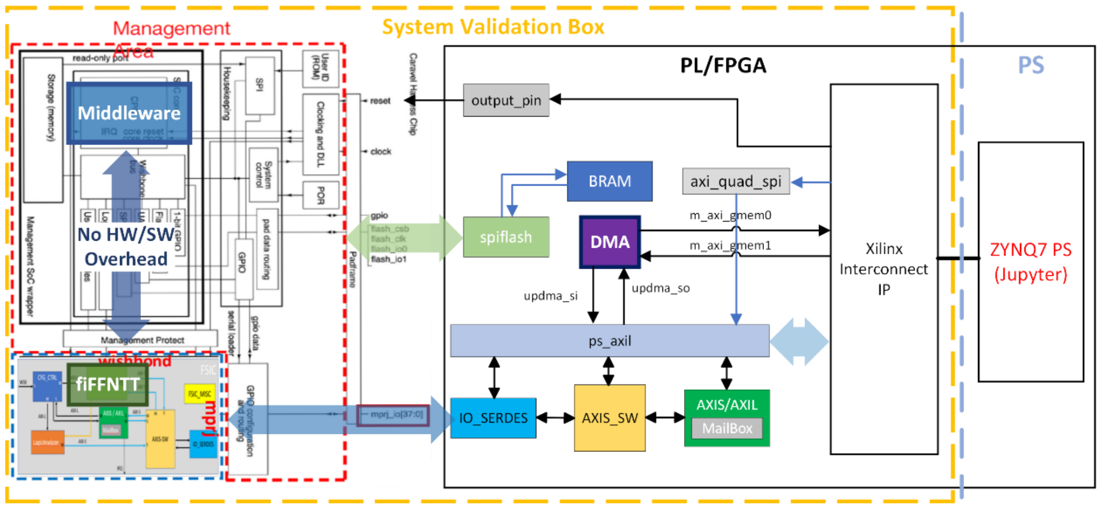
- Asynchronous interrupt service routine:



```
# ======================================================= #
# Create asynchronous task for Interrupt Service Routine
# ======================================================= #
async def isr():
    ## Write aa_mb_irq_en ##
    #mmio.write(0x2100, 0x01)
    print("-> Waitting for MailBox interrupt")
    while(True):
        await mbIRQ.wait() # Wait for Interrupt
        ISR.set() # sets the interrupt
        print("******************************************")
        print("-> Interrupt asserted!!!")
        print("******************************************")
        print(f"Mailbox Pattern: {hex(mmio.read(0x2000))}")
```

The event "ISR" will be raised after interrupt triggered.

6. Simulation & Validation



6-1. FSIC Simulation

- Simulation Flow: Take FFT for example

```
load_dma_FFT();
setting_dma_FFT();

$display($time, "=> Write PL_AA to enable IRQ");
offset = 0;
data = 32'h0000_0001;
axil_cycles_gen(WriteCyc, PL_AA, offset, data, 1);
axil_cycles_gen(ReadCyc, PL_AA, offset, data, 1);

start_algorithm();
start_DMA();

WaitIRQ(1'b1, 32'h3a3a3a3a);
CheckuserDMADone_FFT();
```

- Task : load_dma_FFT → Load data to memory

```
reg [31:0] updma_FFT_x [0:2047];
reg [31:0] updma_FFT_x_data;
task load_dma_FFT;
    begin
        $display($time, "=> load dma FFT ...");
        $display($time, "=> =====================================================================");

        $readmemh("../../../../../FFT_in.hex", updma_FFT_x);

        fd = $fopen ("../../../../../updma_FFT_input.log", "w");
        for (index = 0; index < 2048; index +=1) begin
            updma_FFT_x_data |= updma_FFT_x[index];
            slave_agent3.mem_model.backdoor_memory_write_4byte(addri+4*index,updma_FFT_x_data,4'b1111);
            updma_FFT_x_data = 0;
            $fdisplay(fd, "%08h", slave_agent3.mem_model.backdoor_memory_read_4byte(addri+4*index));
        end
        $fclose(fd);

    end
endtask
```

- Task : setting_dma_FFT → Configure kernel_mode...

```
task setting_dma_FFT;
    begin

        $display($time, "=> ----------------------------------------------------------------------");
        $display($time, "=> FpgaLocal_Write: PL_UPDMA, s2m set buffer low...");
        offset = 32'h0000_0038;
        data = 32'h4508_0000;
        axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
        //#20us
        axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 1);
        //#20us
        if(data == 32'h4508_0000) begin
            $display($time, "=> Fpga2Soc_Write PL_UPDMA offset %h = %h, PASS", offset, data);
        end else begin
            $display($time, "=> Fpga2Soc_Write PL_UPDMA offset %h = %h, FAIL", offset, data);
            ->> error_event;
        end
```

This task will configure some critical address to program dma, it will follow this sequence : 0x38 (s2m buf [31:0]) → 0x3c (s2m buf [63:32]) → 0x44 (m2s buf [31:0]) → 0x48 (m2s buf [63:32]) → 0x10 (kernel mode).

- Task: Enable IRQ

```
$display($time, "=> Write PL_AA to enable IRQ");
offset = 0;
data = 32'h0000_0001;
axil_cycles_gen(WriteCyc, PL_AA, offset, data, 1);
axil_cycles_gen(ReadCyc, PL_AA, offset, data, 1);
```

Before starting to compute, we should enable the irq first in order to check the final status of the ip.

- Task: start_algorithm & start_DMA

```
task start_algorithm;
    begin
        // Select user project2
        $display($time, "=> Fpga2Soc_Write: SOC_CC");

        offset = 0;
        data = 32'h0000_0002;
        axil_cycles_gen(WriteCyc, SOC_CC, offset, data, 1);
        axil_cycles_gen(ReadCyc, SOC_CC, offset, data, 1);

        if(data == 32'h0000_0002) begin
            $display($time, "=> Fpga2Soc_Write SOC_CC offset %h = %h, PASS", offset, data);
        end else begin
            $display($time, "=> Fpga2Soc_Write SOC_CC offset %h = %h, FAIL", offset, data);
            ->> error_event;
        end

    end
endtask

task start_DMA;
    begin
   $display($time, "=> =======================================================================");
        $display($time, "=> Start DMA streaming x in, program ap_start to DMA");
        $display($time, "=> =======================================================================");
        offset = 32'h0000_0000;
        data = 32'h0000_0001;
        axil_cycles_gen(WriteCyc, PL_UPDMA, offset, data, 1);
    end
endtask
```

These two tasks will select user project 2 and program ap_start to user DMA.

- Task : WaitIRQ → Interrupt

```
task WaitIRQ;
    input [0:0]  plaa_en;
    input [31:0] pattern;
    begin
        if (plaa_en == 1'b1) begin
            $display($time, "=> Write PL_AA to enable IRQ");
            offset = 0;
            data = 32'h0000_0001;
            axil_cycles_gen(WriteCyc, PL_AA, offset, data, 1);
            axil_cycles_gen(ReadCyc, PL_AA, offset, data, 1);
        end

        // Wait for IRQ
        keepChk = 1;
        while (keepChk) begin
        //$display($time, "=> Wating irq done...");
        #10
            if (DUT.design_1_i.ps_axil_0.aa_mb_irq == 1'b1) begin
                keepChk = 1'b0;
                $display($time, "=> ***********************************************************************");
                $display($time, "=> iNTT Interrupt Asserted!!");
                $display($time, "=> ***********************************************************************");
            end
        end
        // Read MailBox
        axil_cycles_gen(ReadCyc, PL_AA_MB, offset, data, 1);
        if(data == pattern) begin
            keepChk = 0;
            $display($time, "=> FpgaLocal_Read PL_AA_MB = %h, PASS", data);
        end

    end
endtask
```

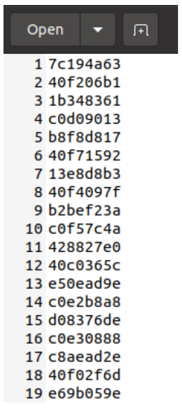
While our ip is computing, firmware keeps checking the status of ip whether the computing is finished or not. Once the computing is done, firmware will generate a pattern written to mailbox, and rise the interrupt at the same time, followed by we reading the message in the mailbox, which set the interrupt to 0.

- Task: Check DMA transfer done

```
task CheckuserDMADone_FFT;
    begin
        $display($time, "=> Starting CheckuserDMADone()...");
        $display($time, "=> ======================================================================");
        $display($time, "=> FpgaLocal_Read: PL_UPDMA");

        keepChk = 1;
        offset = 32'h0000_0018;
        $display($time, "=> Waiting buffer transfer done...");
        while (keepChk) begin
            $display($time, "=> Waiting buffer transfer done...");
            #10us
            axil_cycles_gen(ReadCyc, PL_UPDMA, offset, data, 0);
            if(data == 32'h0000_0001) begin
                $display($time, "=> Buffer transfer done. offset %h = %h, PASS", offset, data);
                keepChk = 0;

                fd = $fopen ("../../../../../updma_output_FFT.log", "w");
                for (index = 0; index < 2048; index +=1) begin
                    reg signed [31:0]anser;
                    anser=slave_agent2.mem_model.backdoor_memory_read_4byte(addro+4*index);
                    $fdisplay(fd, "%08h",anser);
                end

                $fclose(fd);
           end
        end

        ->> userdma_done;
        $display($time, "=> End FFT CheckuserDMADone()...");
        $display($time, "=> ======================================================================");

    end
endtask
```

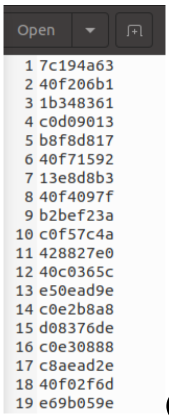
While the DMA finished transferring data, we will write the data to a log file.

- Verify the data



```
7c194a63
40f206b1
1b348361
c0d09013
b8f8d817
40f71592
13e8d8b3
40f4097f
b2bef23a
c0f57c4a
428827e0
40c0365c
e50ead9e
c0e2b8a8
d08376de
c0e30888
c8aead2e
40f02f6d
e69b059e
```

(updma_output_FFT.log)

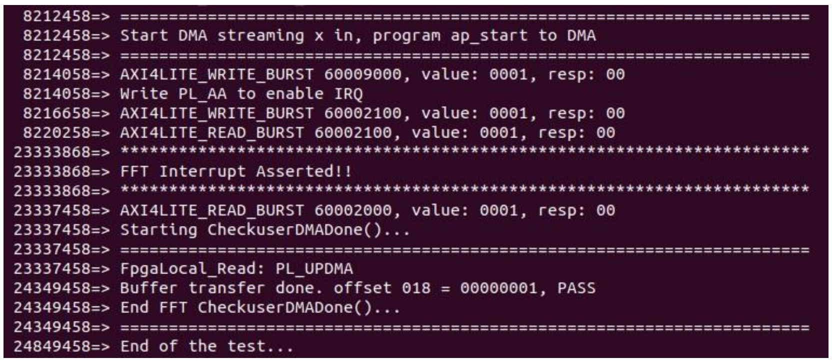


```
7c194a63
40f206b1
1b348361
c0d09013
b8f8d817
40f71592
13e8d8b3
40f4097f
b2bef23a
c0f57c4a
428827e0
40c0365c
e50ead9e
c0e2b8a8
d08376de
c0e30888
c8aead2e
40f02f6d
e69b059e
```

(FFT_out.hex)

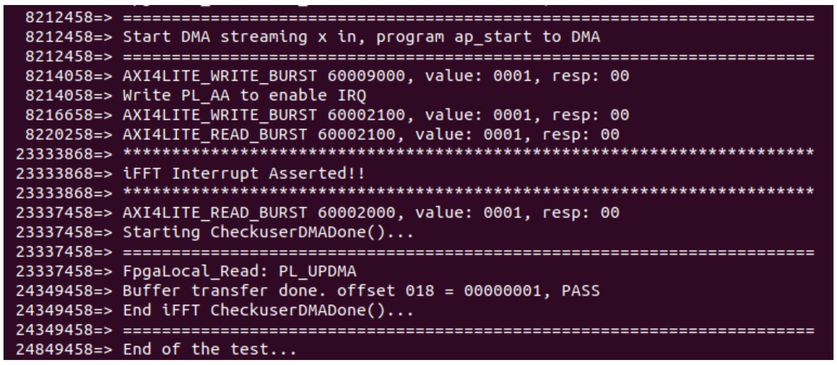
- Simulation Result

  ■ FFT:



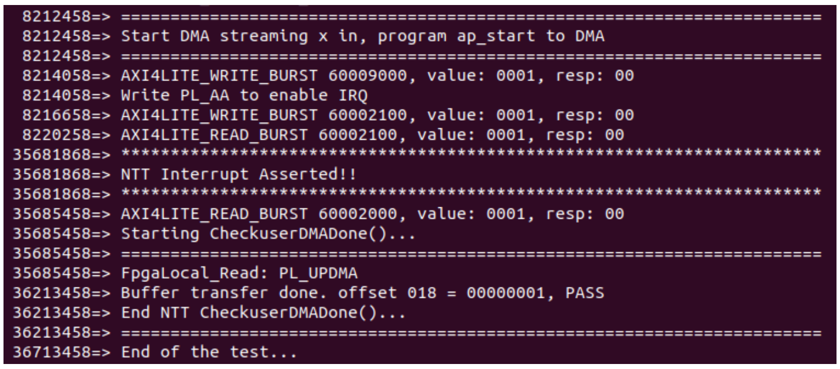
```
 8212458=> ======================================================================
 8212458=> Start DMA streaming x in, program ap_start to DMA
 8212458=> ======================================================================
 8214058=> AXI4LITE_WRITE_BURST 60009000, value: 0001, resp: 00
 8214058=> Write PL_AA to enable IRQ
 8216658=> AXI4LITE_WRITE_BURST 60002100, value: 0001, resp: 00
 8220258=> AXI4LITE_READ_BURST 60002100, value: 0001, resp: 00
23333868=> **********************************************************************
23333868=> FFT Interrupt Asserted!!
23333868=> **********************************************************************
23337458=> AXI4LITE_READ_BURST 60002000, value: 0001, resp: 00
23337458=> Starting CheckuserDMADone()...
23337458=> ======================================================================
23337458=> FpgaLocal_Read: PL_UPDMA
24349458=> Buffer transfer done. offset 018 = 00000001, PASS
24349458=> End FFT CheckuserDMADone()...
24349458=> ======================================================================
24849458=> End of the test...
```

  ■ iFFT:



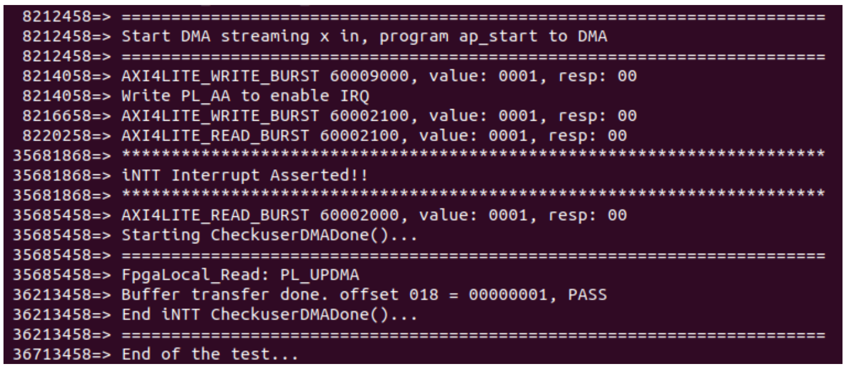
```
 8212458=> ======================================================================
 8212458=> Start DMA streaming x in, program ap_start to DMA
 8212458=> ======================================================================
 8214058=> AXI4LITE_WRITE_BURST 60009000, value: 0001, resp: 00
 8214058=> Write PL_AA to enable IRQ
 8216658=> AXI4LITE_WRITE_BURST 60002100, value: 0001, resp: 00
 8220258=> AXI4LITE_READ_BURST 60002100, value: 0001, resp: 00
23333868=> **********************************************************************
23333868=> iFFT Interrupt Asserted!!
23333868=> **********************************************************************
23337458=> AXI4LITE_READ_BURST 60002000, value: 0001, resp: 00
23337458=> Starting CheckuserDMADone()...
23337458=> ======================================================================
23337458=> FpgaLocal_Read: PL_UPDMA
24349458=> Buffer transfer done. offset 018 = 00000001, PASS
24349458=> End iFFT CheckuserDMADone()...
24349458=> ======================================================================
24849458=> End of the test...
```

  ■ NTT:



```
 8212458=> ======================================================================
 8212458=> Start DMA streaming x in, program ap_start to DMA
 8212458=> ======================================================================
 8214058=> AXI4LITE_WRITE_BURST 60009000, value: 0001, resp: 00
 8214058=> Write PL_AA to enable IRQ
 8216658=> AXI4LITE_WRITE_BURST 60002100, value: 0001, resp: 00
 8220258=> AXI4LITE_READ_BURST 60002100, value: 0001, resp: 00
35681868=> **********************************************************************
35681868=> NTT Interrupt Asserted!!
35681868=> **********************************************************************
35685458=> AXI4LITE_READ_BURST 60002000, value: 0001, resp: 00
35685458=> Starting CheckuserDMADone()...
35685458=> ======================================================================
35685458=> FpgaLocal_Read: PL_UPDMA
36213458=> Buffer transfer done. offset 018 = 00000001, PASS
36213458=> End NTT CheckuserDMADone()...
36213458=> ======================================================================
36713458=> End of the test...
```

  ■ iNTT:



```
 8212458=> ======================================================================
 8212458=> Start DMA streaming x in, program ap_start to DMA
 8212458=> ======================================================================
 8214058=> AXI4LITE_WRITE_BURST 60009000, value: 0001, resp: 00
 8214058=> Write PL_AA to enable IRQ
 8216658=> AXI4LITE_WRITE_BURST 60002100, value: 0001, resp: 00
 8220258=> AXI4LITE_READ_BURST 60002100, value: 0001, resp: 00
35681868=> **********************************************************************
35681868=> iNTT Interrupt Asserted!!
35681868=> **********************************************************************
35685458=> AXI4LITE_READ_BURST 60002000, value: 0001, resp: 00
35685458=> Starting CheckuserDMADone()...
35685458=> ======================================================================
35685458=> FpgaLocal_Read: PL_UPDMA
36213458=> Buffer transfer done. offset 018 = 00000001, PASS
36213458=> End iNTT CheckuserDMADone()...
36213458=> ======================================================================
36713458=> End of the test...
```

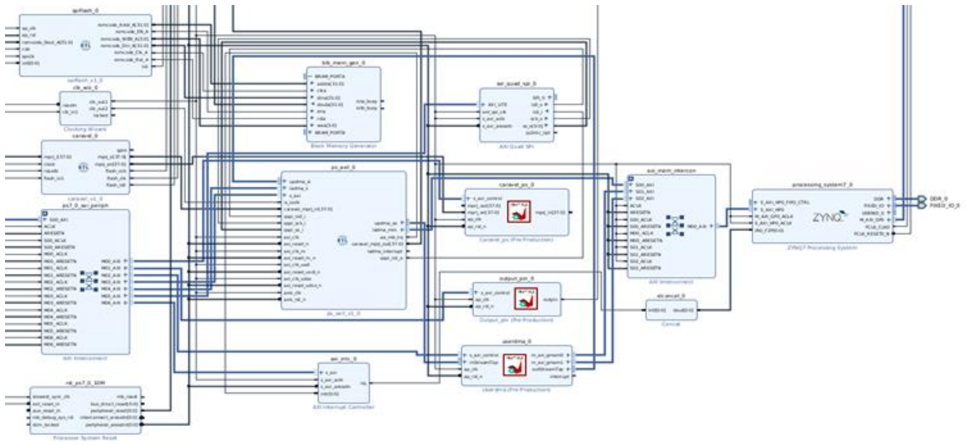
- Simulation run time:

Algorithm	Run time	Final time
FFT / iFFT	1512_1410	1512_5000
NTT / iNTT	2746_9410	2747_3000

From the table we can inspect that the time we spend on FFT/iFFT or NTT/iNTT is the same, we guess the reason being that we haven't optimize the kernel, so both algorithm uses the worst case.

6-2. On-Board Validation

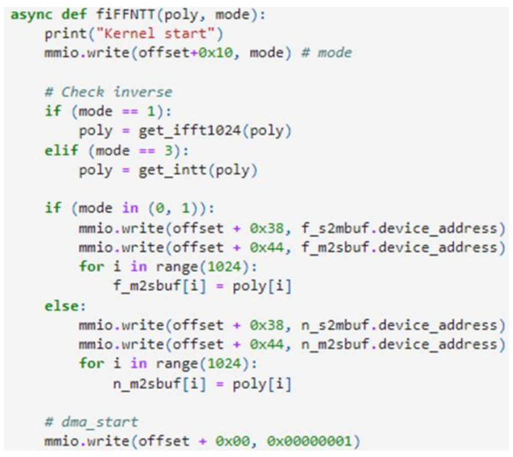
- System Block Diagram on Vivado GUI



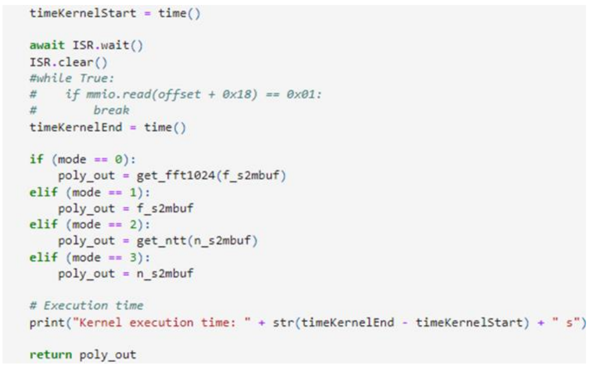
To reduce hardware & software overhead, we use the Mailbox (MB) on the FPGA side to generate an interrupt service routine to inform the PS side that our data is ready. Since we can't not add a new ISR directly in python code, we need additional procedure after running run_vivado_fsic:

1. Add AXI Interrupt Controller in FPGA block design. Choose Interrupt type - Level trigger and Interrupt Output Connection to Single.
2. The mailbox(MB) must provide a interrupt pin(aa_mb_irq) and connect to the AXI Interrupt Controller.
3. Connect the irq of Interrupt Controller to IRQ_F2P[0:0] in PS CPU.
4. Porting Jupyter Notebook Python code to import asyncio library.

- Validation Tasks
  - ■ fiFFNTT



```python
async def fiFFNTT(poly, mode):
    print("Kernel start")
    mmio.write(offset+0x10, mode) # mode

    # Check inverse
    if (mode == 1):
        poly = get_ifft1024(poly)
    elif (mode == 3):
        poly = get_intt(poly)

    if (mode in (0, 1)):
        mmio.write(offset + 0x38, f_s2mbuf.device_address)
        mmio.write(offset + 0x44, f_m2sbuf.device_address)
        for i in range(1024):
            f_m2sbuf[i] = poly[i]
    else:
        mmio.write(offset + 0x38, n_s2mbuf.device_address)
        mmio.write(offset + 0x44, n_m2sbuf.device_address)
        for i in range(1024):
            n_m2sbuf[i] = poly[i]

    # dma_start
    mmio.write(offset + 0x00, 0x00000001)
```



```python
    timeKernelStart = time()

    await ISR.wait()
    ISR.clear()
    #while True:
    #    if mmio.read(offset + 0x18) == 0x01:
    #        break
    timeKernelEnd = time()

    if (mode == 0):
        poly_out = get_fft1024(f_s2mbuf)
    elif (mode == 1):
        poly_out = f_s2mbuf
    elif (mode == 2):
        poly_out = get_ntt(n_s2mbuf)
    elif (mode == 3):
        poly_out = n_s2mbuf

    # Execution time
    print("Kernel execution time: " + str(timeKernelEnd - timeKernelStart) + " s")

    return poly_out
```

Since python has complex number data type, our reference falcon python code uses complex number to represent the polynomials used in FFT. However, we use C code (HLS) to implement kernel function fiFFNTT which does not have complex number data type, so when API falcon need to call our kernel function, we need to transfer our polynomial format so kernel function can compute. Also, the format of polynomial in NTT are different in python code and HLS code. Therefore, we used get_ifft, get_fft, get_intt, and get_ntt to change the format.

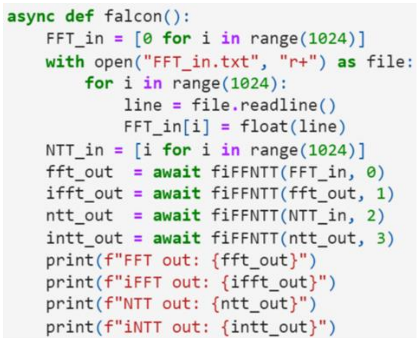
polynomial used in FFT in python

Index	0	2 3 ...	512	513 514 ... 1023
data	A0 + jB0	......	A0 - jB0	......

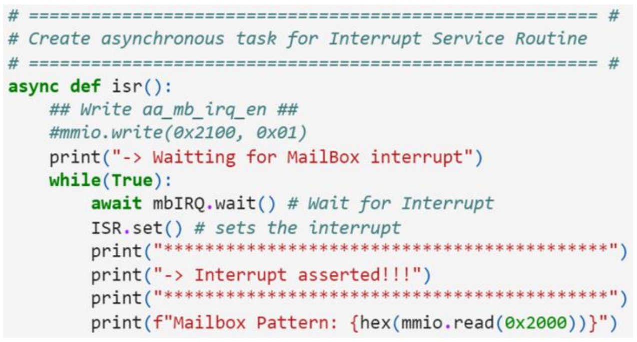
polynomial used in FFT in kernel function

Index	0	2 3 ...	512	513 514 ... 1023
data	A0	......	B0	......

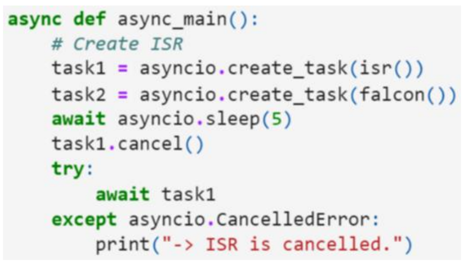
- Test fiFFNTT multiple times with ISR



```python
async def falcon():
    FFT_in = [0 for i in range(1024)]
    with open("FFT_in.txt", "r+") as file:
        for i in range(1024):
            line = file.readline()
            FFT_in[i] = float(line)
    NTT_in = [i for i in range(1024)]
    fft_out  = await fiFFNTT(FFT_in, 0)
    ifft_out = await fiFFNTT(fft_out, 1)
    ntt_out  = await fiFFNTT(NTT_in, 2)
    intt_out = await fiFFNTT(ntt_out, 3)
    print(f"FFT out: {fft_out}")
    print(f"iFFT out: {ifft_out}")
    print(f"NTT out: {ntt_out}")
    print(f"iNTT out: {intt_out}")
```



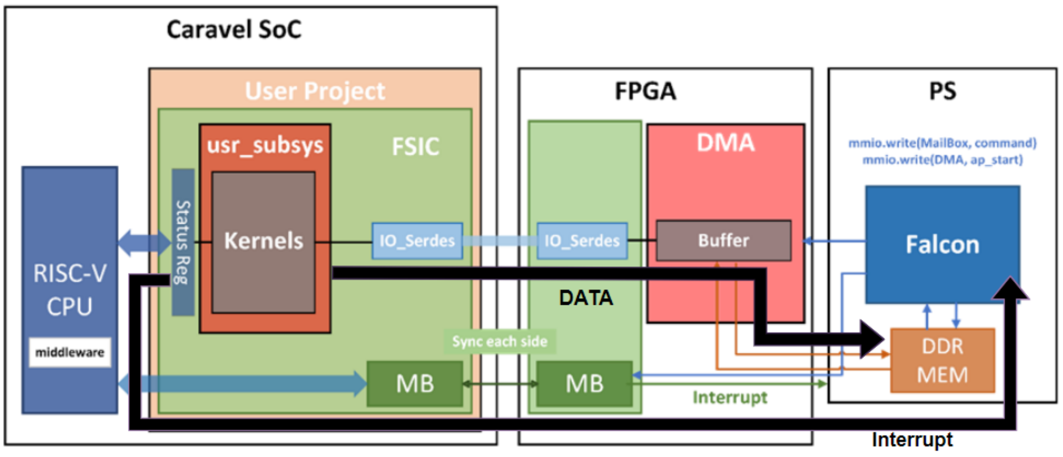
```python
# ======================================================= #
# Create asynchronous task for Interrupt Service Routine
# ======================================================= #
async def isr():
    ## Write aa_mb_irq_en ##
    #mmio.write(0x2100, 0x01)
    print("-> Waitting for MailBox interrupt")
    while(True):
        await mbIRQ.wait() # Wait for Interrupt
        ISR.set() # sets the interrupt
        print("***************************************************")
        print("-> Interrupt asserted!!!")
        print("***************************************************")
        print(f"Mailbox Pattern: {hex(mmio.read(0x2000))}")
```



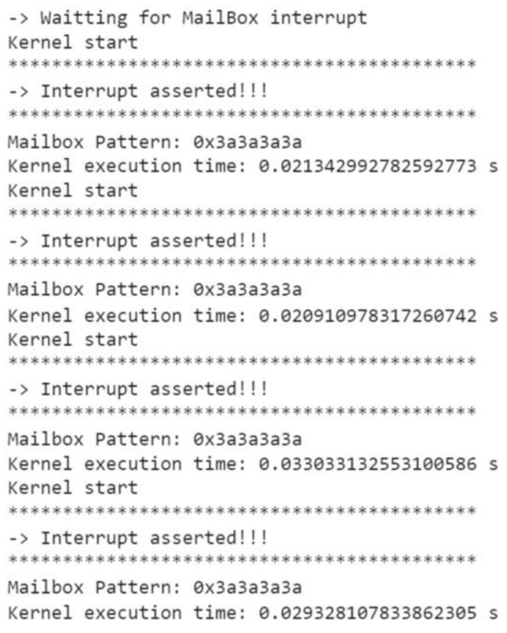
```python
async def async_main():
    # Create ISR
    task1 = asyncio.create_task(isr())
    task2 = asyncio.create_task(falcon())
    await asyncio.sleep(5)
    task1.cancel()
    try:
        await task1
    except asyncio.CancelledError:
        print("-> ISR is cancelled.")
```

We didn’t test complete falcon flow since we don’t have the library like PyCrypto , PyCryptoDome, PyCryptoDomeX in pynq jupyter notebook. If we have the permission to install all library required for falcon python code on our jupyter notebook, we can test complete falcon flow including key generation, signature and verification.

The other problem we found is that interrupt and data movement have different path, which means when assert interrupt, we cannot ensure that data has been moved to DDR memory. When simulation, we found that interrupt is asserted before DMA load all data to DDR memory. However, when on-board verification, interrupt is asserted after DMA load all data to DDR memory. We think it is because we use multi thread in jupyter notebook and when interrupt asserted, CPU need to stop other program which need some amount of time and then deal with interrupt service routine.



- Run time

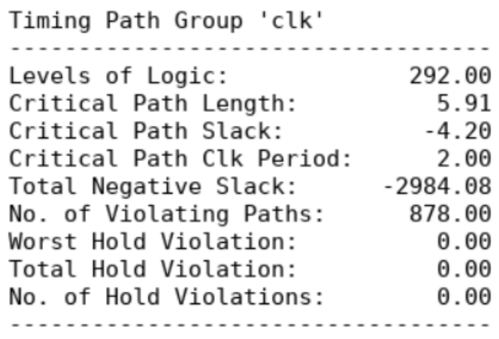

```
-> Waitting for MailBox interrupt
Kernel start
*******************************************
-> Interrupt asserted!!!
*******************************************
Mailbox Pattern: 0x3a3a3a3a
Kernel execution time: 0.021342992782592773 s
Kernel start
*******************************************
-> Interrupt asserted!!!
*******************************************
Mailbox Pattern: 0x3a3a3a3a
Kernel execution time: 0.020910978317260742 s
Kernel start
*******************************************
-> Interrupt asserted!!!
*******************************************
Mailbox Pattern: 0x3a3a3a3a
Kernel execution time: 0.033033132553100586 s
Kernel start
*******************************************
-> Interrupt asserted!!!
*******************************************
Mailbox Pattern: 0x3a3a3a3a
Kernel execution time: 0.029328107833862305 s
```

7. Synopsys IC Flow

- Design Compiler:

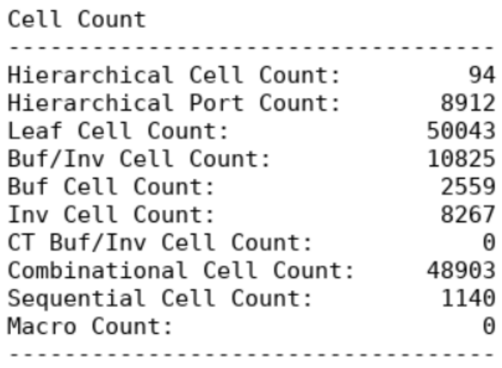
At first, we run design compiler with user project 2, but since we don’t have time to optimize our ip and using ram model, thus we only run our ‘fiFFNTT’. And since we encouter many problems in the next step “floor planning”, we end up finish design compiler synthesis only.

■ Timing:



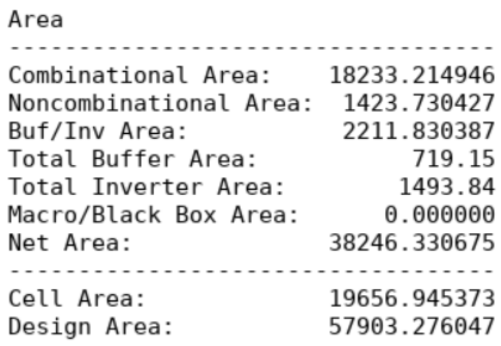
```
Timing Path Group 'clk'
-----------------------------------
Levels of Logic:            292.00
Critical Path Length:         5.91
Critical Path Slack:         -4.20
Critical Path Clk Period:     2.00
Total Negative Slack:     -2984.08
No. of Violating Paths:     878.00
Worst Hold Violation:         0.00
Total Hold Violation:         0.00
No. of Hold Violations:       0.00
-----------------------------------
```

■ Cell Count:



```
Cell Count
-----------------------------------
Hierarchical Cell Count:        94
Hierarchical Port Count:      8912
Leaf Cell Count:             50043
Buf/Inv Cell Count:          10825
Buf Cell Count:               2559
Inv Cell Count:               8267
CT Buf/Inv Cell Count:           0
Combinational Cell Count:    48903
Sequential Cell Count:        1140
Macro Count:                     0
-----------------------------------
```

■ Area:



```
Area
-----------------------------------
Combinational Area:    18233.214946
Noncombinational Area:  1423.730427
Buf/Inv Area:           2211.830387
Total Buffer Area:           719.15
Total Inverter Area:        1493.84
Macro/Black Box Area:      0.000000
Net Area:              38246.330675
-----------------------------------
Cell Area:             19656.945373
Design Area:           57903.276047
```

From the result of the report generated by the design compiler, we can see that it’s much worse than our previous ip generated by vitis HLS, so this will be one of our critical future works.