SOC Final Project

Hardware Accelerator, SDRAM, and UART

110590022 陳冠晰

110000107 陳柏翰

110061217 王彥智

110011141 陳昇達

110020015 劉祐瑋

Table of Contents

Part I: Introduction

We will optimize based on the workload of Lab 6, dividing the process into two parts. The first part is Computation, where we will perform computations for FIR, Quick Sort, and Matrix Multiplication. The timing will start when any of the three operations begins and will end when all three operations are completed. In Lab 6, we used software for computations, and we aim to accelerate the process using hardware techniques learned from Lab 4.

The second part is Communication, where we aim to increase the speed of UART transmission and reduce ISQ. We plan to use FIFO to achieve acceleration in this aspect.

Part II: Computation

1. **Specification**

   Computational data sizes:
   FIR - 64 data points

   Sorting - 10 data points

   Matrix Multiplication - 4x4

   Other requirements: Data and firmware code must both be stored in user_ram.

2. **The Overall Design Concept - Datapath**

   As shown in Figure 1, we've designed hardware for three accelerators. In addition, we plan to replace the original BRAM with SDRAM. In Lab4-2, we used the CPU to feed data, but for this experiment, we have already preloaded the data into SDRAM. Therefore, we will design a DMA (Direct Memory Access) to directly read from or write to our data in SDRAM. Since there are 3 DMA controllers and 1 CPU, resulting in multiple Wishbone buses, we will design arbitration logic, possibly using an arbiter, to determine which Wishbone bus will access SDRAM at any given time. This ensures proper coordination and control when multiple components are trying to access the SDRAM simultaneously.

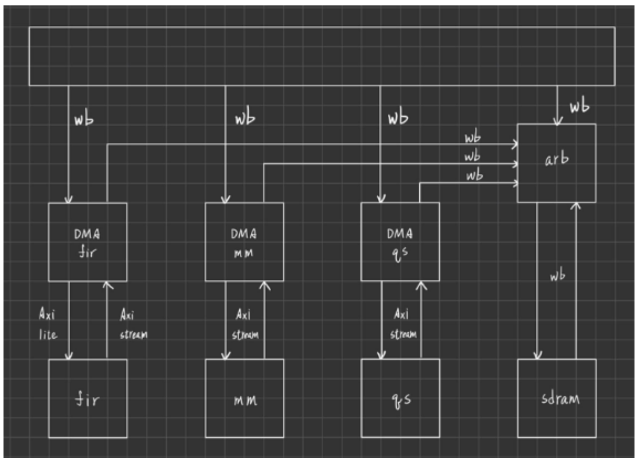


Figure. 1. Data path

3. Module Description

(1) FIR

Please refer to Lab 3 report for more details:

https://github.com/vic9112/SOC/blob/main/Lab3-FIR/Report.pdf

(2) Matrix Multiplication

In the Datapath, as shown in Figure 2, we utilize AXI-Stream to store matrices A and B in the hardware. We then employ a pipeline approach to calculate matrix C. Finally, the output is directed through AXI-Stream.

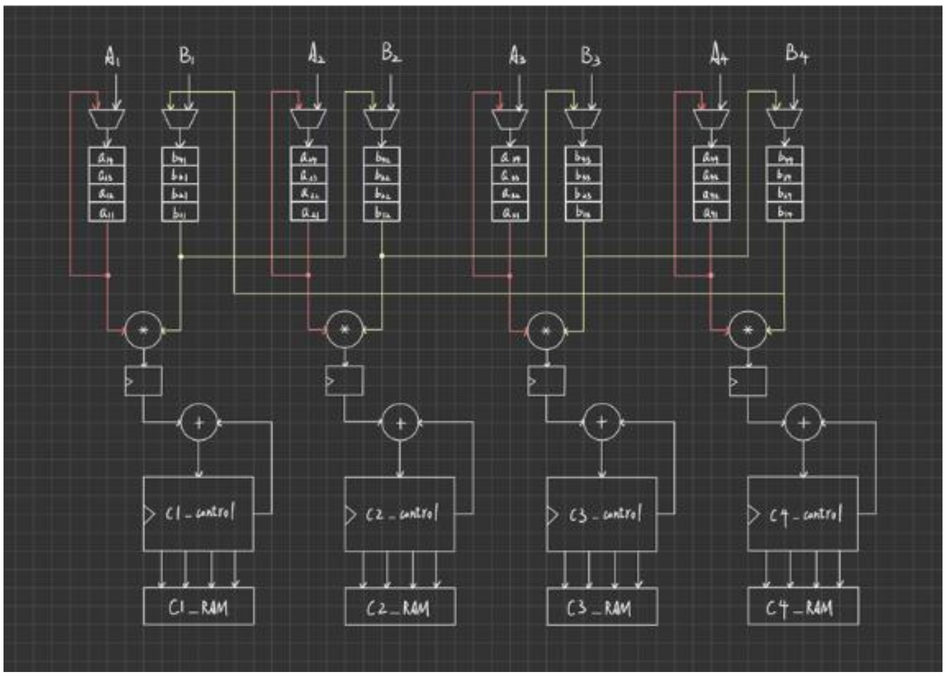


Figure. 2. Datapath of Matrix Multiplication

(3) Sorting

In Datapath, as illustrated in Figure 3, unlike Lab 6 where Quick Sort was used for sorting, in this hardware design, we utilize Insertion Sort. Upon data input, we can employ 10 comparators to identify the corresponding index for insertion. The sorting process involves utilizing shift operations to arrange the elements.

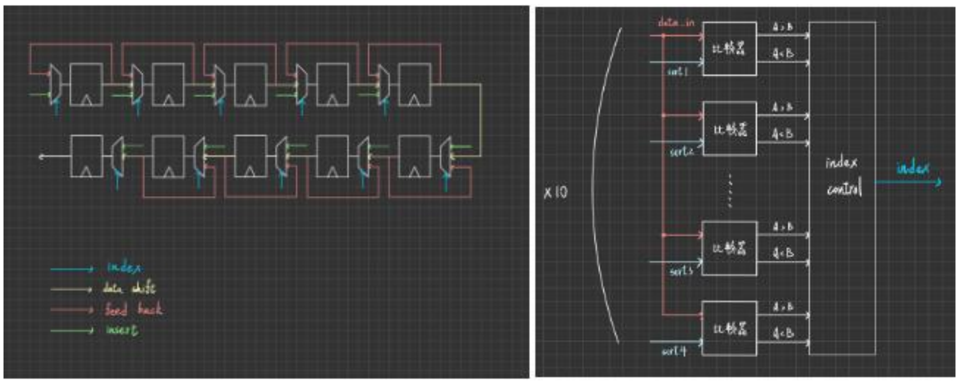


Figure. 3. Datapath of inserting sort

(4) DMA

Our DMA design is straightforward. Inside, we have two buffers for evaluation (X_FF for input and Y_FF for output). When X_FF is empty, it enters the X_addr state, issuing a wishbone (WB) with wb_addr as the address of X to read data from SDRAM. When Y_FF is full, it enters the Y_addr state, issuing a WB (with wb_addr as the address of Y and wb_we set to 1) to write data into SDRAM. Figure 4 illustrates the Finite State Machine (FSM) for the DMA.

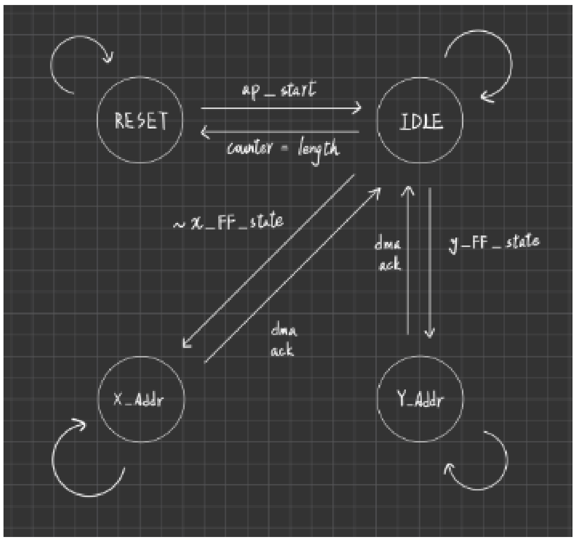
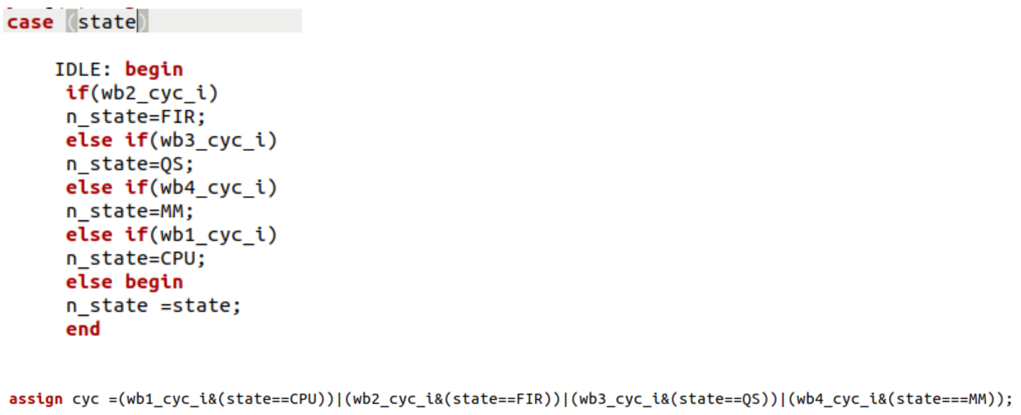


Figure. 4. FSM of DMA

(5) Arbitrary

We use a Finite State Machine to coordinate the output WB, and our priority order is FIR > Quick Sort > Matrix Multiplication > CPU.



```
case (state)

    IDLE: begin
     if(wb2_cyc_i)
     n_state=FIR;
     else if(wb3_cyc_i)
     n_state=QS;
     else if(wb4_cyc_i)
     n_state=MM;
     else if(wb1_cyc_i)
     n_state=CPU;
     else begin
     n_state =state;
     end

assign cyc =(wb1_cyc_i&(state==CPU))|(wb2_cyc_i&(state==FIR))|(wb3_cyc_i&(state==QS))|(wb4_cyc_i&(state===MM));
```

Figure. 5. FSM coding and output WB coding way。

(6) SDRAM

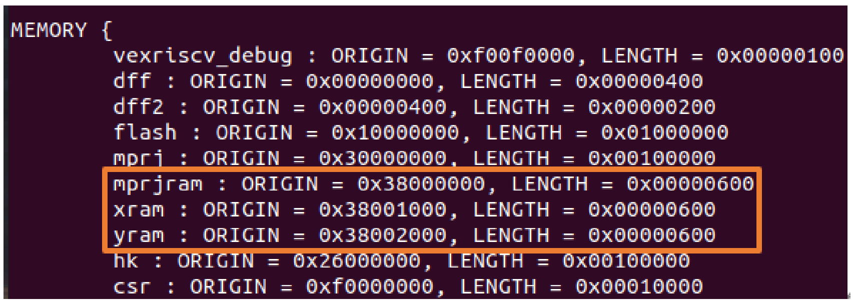
Please refer to Lab D report for some details of basic design ideals: https://github.com/vic9112/SOC/blob/main/Lab-SDRAM/Report.pdf

In this design, since we need to initialize data and place it into SDRAM, it introduces some challenges that require modifications to our original design.

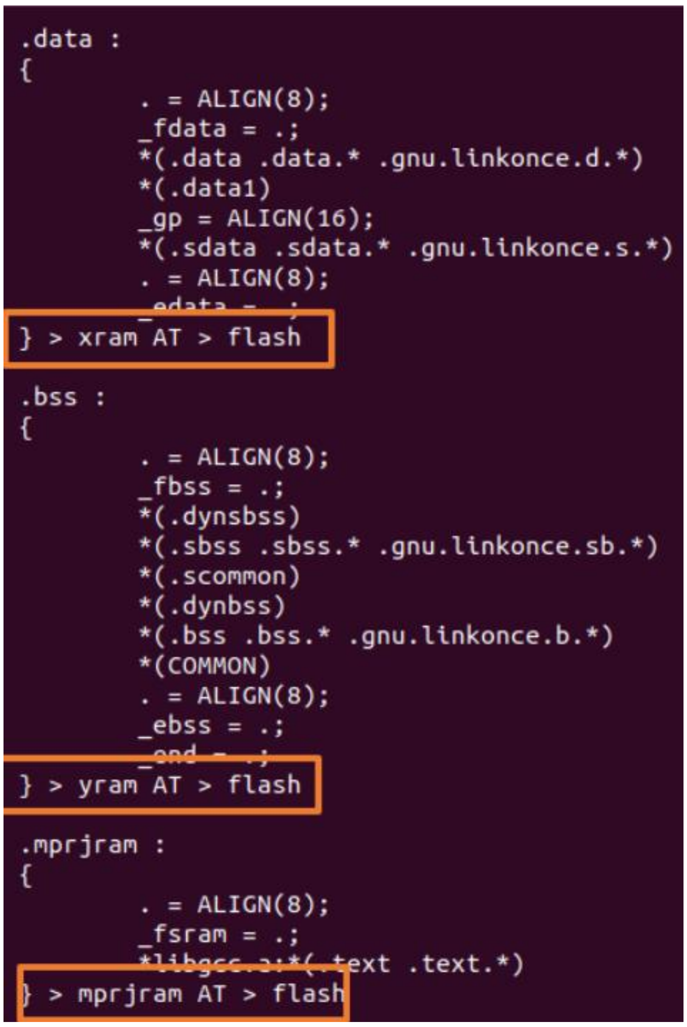
Problem 1 – Address Mapping

```
`define BA 9:8
`define RA 22:10
`define CA 7:0
```

We may encounter a situation that all data are stored in the same bank since our linker was designed as shown below:



```
MEMORY {
        vexriscv_debug : ORIGIN = 0xf00f0000, LENGTH = 0x00000100
        dff : ORIGIN = 0x00000000, LENGTH = 0x00000400
        dff2 : ORIGIN = 0x00000400, LENGTH = 0x00000200
        flash : ORIGIN = 0x10000000, LENGTH = 0x01000000
        mprj : ORIGIN = 0x30000000, LENGTH = 0x00100000
        mprjram : ORIGIN = 0x38000000, LENGTH = 0x00000600
        xram : ORIGIN = 0x38001000, LENGTH = 0x00000600
        yram : ORIGIN = 0x38002000, LENGTH = 0x00000600
        hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
        csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
```



```
.data :
{
        . = ALIGN(8);
        _fdata = .;
        *(.data .data.* .gnu.linkonce.d.*)
        *(.data1)
        _gp = ALIGN(16);
        *(.sdata .sdata.* .gnu.linkonce.s.*)
        . = ALIGN(8);
        _edata = .;
} > xram AT > flash

.bss :
{
        . = ALIGN(8);
        _fbss = .;
        *(.dynsbss)
        *(.sbss .sbss.* .gnu.linkonce.sb.*)
        *(.scommon)
        *(.dynbss)
        *(.bss .bss.* .gnu.linkonce.b.*)
        *(COMMON)
        . = ALIGN(8);
        _ebss = .;
        _end = .;
} > yram AT > flash

.mprjram :
{
        . = ALIGN(8);
        _fsram = .;
        *libgcc.a:*(.text .text.*)
} > mprjram AT > flash
```

Each data type needs at least 12-bit, but SDRAM only take 8 bit for column address, and bank_address[9:8] will restrict our size .

<u>Problem 2 – Address sends into SDRAM</u>

In SDRAM, we have 4 banks to store data, source code are shown below:

```
blkRam$(.SIZE(mem_sizes), .BIT_WIDTH(DQ_BITS))
Bank0(
    .clk(Sys_clk),
    .we(bwen[0]),
    .re(bren[0]),
    .waddr(Col_brst[9:0]),
    .raddr(Col_brst[9:0]),
    .d(bdi[0]),
    .q(bqd[0])
);
```

Only 10- bit for column address

```
READ: begin
    cmd_d = CMD_READ;
    a_d = {2'b0, 1'b0, addr_q[7:0], 2'b0};
    state_d = WAIT;
end
```

Since the address of assembly code plus 4 each time, we may not need to add the 2'b0 at the LSB.

➔ **Original memory size: 1 K bytes**

Solution:

Our design (remapping)

```
`define Ba 13:12
`define Ra 22:14
`define CA 11:0
```

Then we can get 12 bits for column address.

We also remap the **a_d** since **a_d[10]** is the precharge signal. Note that **BA** is 13:12.

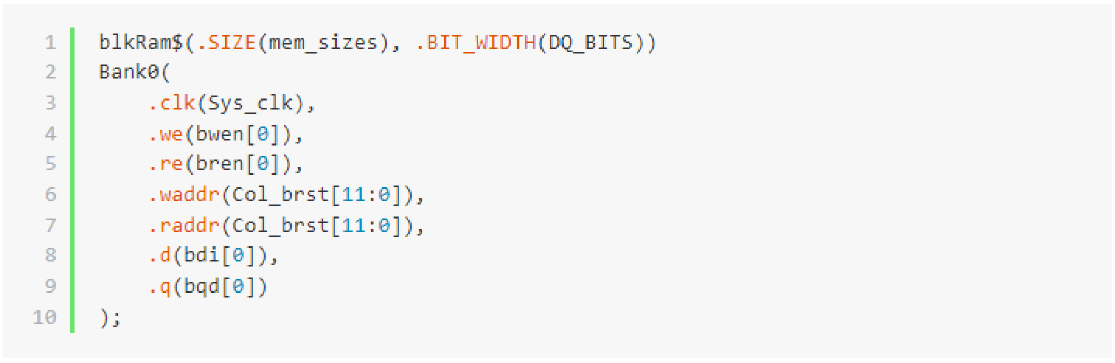
```
READ: begin
    cmd_d   = CMD_READ;
    a_d     = {addr_q[11:10], 1'b0, addr_q[9:0]};
    ba_d    = addr_q[9:8];
    state_d = WAIT;
end
```

When read:

```
If (Read_enable) begin
    Bank     <= Ba;
    Col      <= {Addr[12:11], Addr[9:0]};
    Col_brst <= {Addr[12:11], Addr[9:0]};
end
```

Decode the remapping address, prevent the **addr[10]** (precharge bit) load into block memory.

Now, address load into memory has 12 bits:



```
blkRam$(.SIZE(mem_sizes), .BIT_WIDTH(DQ_BITS))
Bank0(
    .clk(Sys_clk),
    .we(bwen[0]),
    .re(bren[0]),
    .waddr(Col_brst[11:0]),
    .raddr(Col_brst[11:0]),
    .d(bdi[0]),
    .q(bqd[0])
);
```

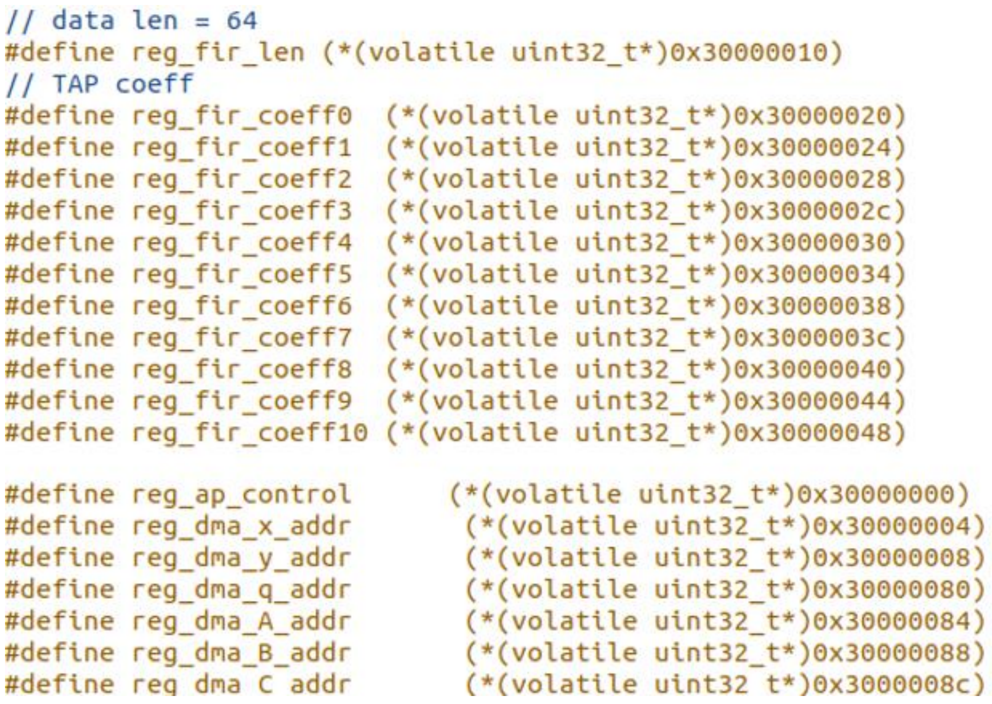
Also, seems that the block memory in the reference code don't have the row address, it may not support the on/off page characteristic.

➔ ***Memory size: 16K bytes***

4. Verification

(1) Firmware code

Configuration Register for our design as below:



```
// data len = 64
#define reg_fir_len (*(volatile uint32_t*)0x30000010)
// TAP coeff
#define reg_fir_coeff0  (*(volatile uint32_t*)0x30000020)
#define reg_fir_coeff1  (*(volatile uint32_t*)0x30000024)
#define reg_fir_coeff2  (*(volatile uint32_t*)0x30000028)
#define reg_fir_coeff3  (*(volatile uint32_t*)0x3000002c)
#define reg_fir_coeff4  (*(volatile uint32_t*)0x30000030)
#define reg_fir_coeff5  (*(volatile uint32_t*)0x30000034)
#define reg_fir_coeff6  (*(volatile uint32_t*)0x30000038)
#define reg_fir_coeff7  (*(volatile uint32_t*)0x3000003c)
#define reg_fir_coeff8  (*(volatile uint32_t*)0x30000040)
#define reg_fir_coeff9  (*(volatile uint32_t*)0x30000044)
#define reg_fir_coeff10 (*(volatile uint32_t*)0x30000048)

#define reg_ap_control      (*(volatile uint32_t*)0x30000000)
#define reg_dma_x_addr       (*(volatile uint32_t*)0x30000004)
#define reg_dma_y_addr       (*(volatile uint32_t*)0x30000008)
#define reg_dma_q_addr       (*(volatile uint32_t*)0x30000080)
#define reg_dma_A_addr       (*(volatile uint32_t*)0x30000084)
#define reg_dma_B_addr       (*(volatile uint32_t*)0x30000088)
#define reg_dma_C_addr       (*(volatile uint32_t*)0x3000008c)
```

In main() code, we will initialize the coefficients of FIR and the address of the DMA, finally, we would start when the ap_start, and end when the CPU polling ap_idle 1.

```c
// write fir data length
reg_fir_len = 64;
// write fir tap coef
reg_fir_coeff0          = 0;
reg_fir_coeff1          = -10;
reg_fir_coeff2          = -9;
reg_fir_coeff3          = 23;
reg_fir_coeff4          = 56;
reg_fir_coeff5          = 63;
reg_fir_coeff6          = 56;
reg_fir_coeff7          = 23;
reg_fir_coeff8          = -9;
reg_fir_coeff9          = -10;
reg_fir_coeff10 = 0;
 // dma address
 int32_t *addr_x=x;
 int32_t *addr_y=y;
 int32_t *addr_A=A;
 int32_t *addr_B=B;
 int32_t *addr_C=C;
 int32_t *addr_q=q;
 reg_dma_x= (int32_t)addr_x; // input dma x begin addr
 reg_dma_y= (int32_t)addr_y; // input dma y begin addr
 reg_dma_A= (int32_t)addr_A; // input dma x begin addr
 reg_dma_B= (int32_t)addr_B; // input dma y begin addr
 reg_dma_C= (int32_t)addr_C; // input dma x begin addr
 reg_dma_q= (int32_t)addr_q; // input dma y begin addr
 reg_mprj_datal = 0xAB400000;// for testbench start signal
reg_fir_control = 1;// let ap start
while(1){if(reg_fir_control==4) break;}; // when ap_idle=1, break;
reg_mprj_datal = 0xAB510000; // for testbench end signal
```

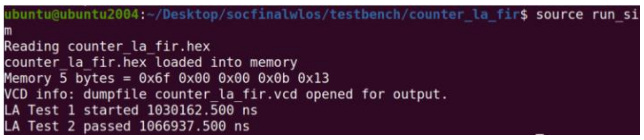
However, if we want to check whether out data load into SDRAM or not, we can define the configuration address where the data load, then by CPU polling to check.

```c
#define reg_check_fir      (*(volatile uint32_t*)0x38002100)

while(1){if(reg_check_fir==10614) break;};
```

(2) Result

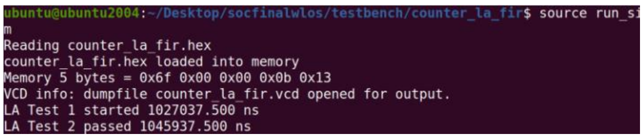
The FIR accelerator takes 1471 cycles to complete its operation.



```
ubuntu@ubuntu2004:~/Desktop/socfinalwlos/testbench/counter_la_fir$ source run_si
m
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started 1030162.500 ns
LA Test 2 passed 1066937.500 ns
```

Figure. 5. FIR simulation

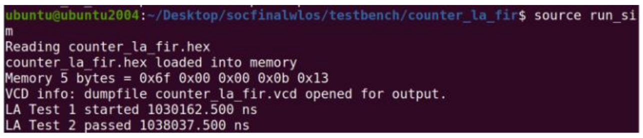
Matrix Multiplication takes 756 cycles.



```
ubuntu@ubuntu2004:~/Desktop/socfinalwlos/testbench/counter_la_fir$ source run_si
m
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started 1027037.500 ns
LA Test 2 passed 1045937.500 ns
```

Figure. 6. MM simulation

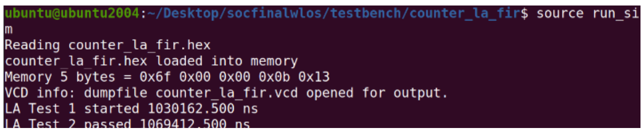
Sorting takes 315 cycles.



```
ubuntu@ubuntu2004:~/Desktop/socfinalwlos/testbench/counter_la_fir$ source run_si
m
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started 1030162.500 ns
LA Test 2 passed 1038037.500 ns
```

Figure. 7. QS simulation

Due to the presence of an Arbitration mechanism, the three hardware accelerators can operate simultaneously, achieving concurrent execution. The actual completion time for all three is 1570 cycles.



```
ubuntu@ubuntu2004:~/Desktop/socfinalwlos/testbench/counter_la_fir$ source run_si
m
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started 1030162.500 ns
LA Test 2 passed 1069412.500 ns
```

Figure. 8 Total simulation

(3) Waveform analysis

Overall view:

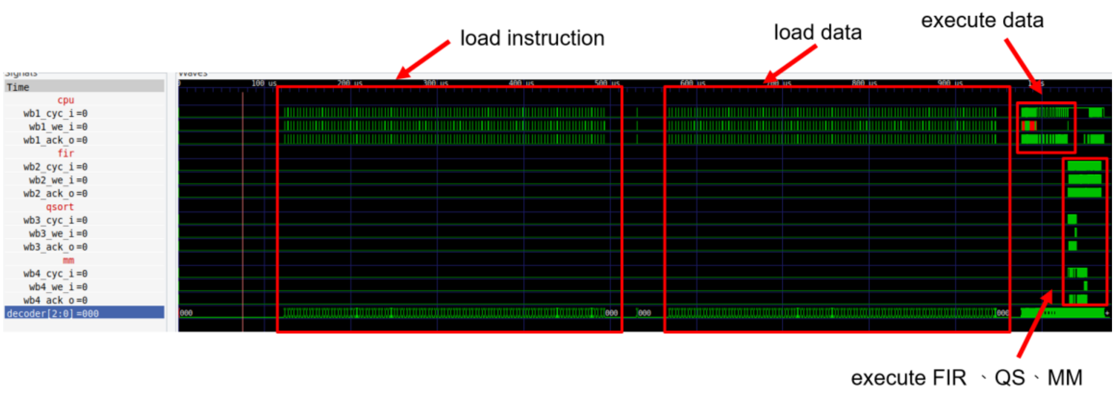


Figure. 9. Overall waveform

Execute FIR 、QS 、MM part:

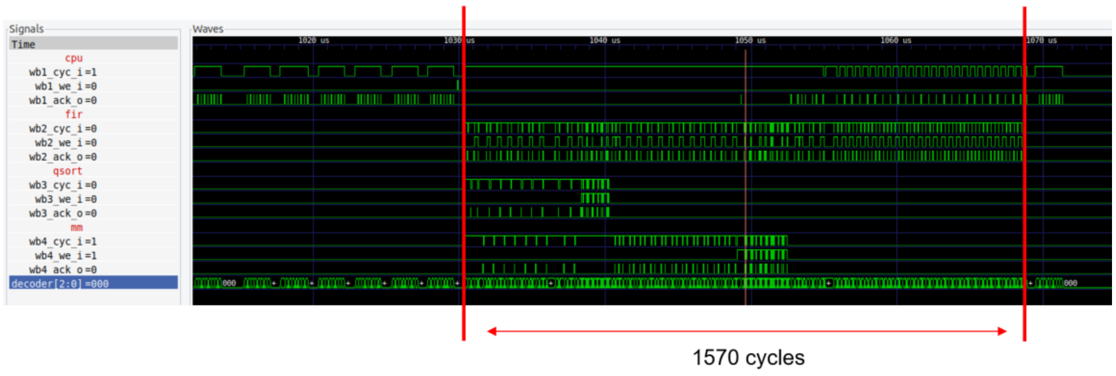


Figure. 10. the waveform of Executing FIR, QS, MM part

Now, we observe the function of arbitrary. by Figure 11 , we can find that the wishbone cycle into SDRAM from different wishbone cycle (ex: FIR 、CPU...).so, we can let the wishbone from the arbitrary continuously read data or write data to SDRAM, effectively use the SDRAM.

State IDLE = 3'b000, CPU= 3'b001, FIR = 3'b010, QS=3'b011, MM= 3'b100

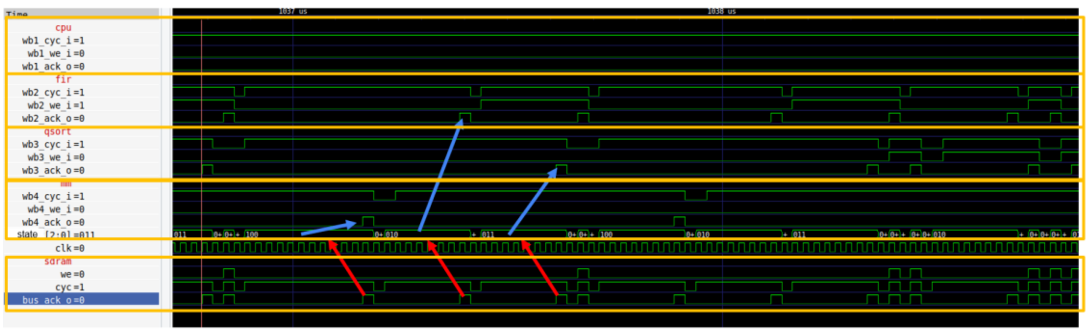


Figure. 11. The function of arbitrary.

5. Further Optimization

We know that under ideal conditions with a latency of 1 T for data reading, the required time for each of the three hardware components is as follows:

- ⇨ FIR: 64 x 11 = 704 cycles
- ⇨ QS: 10 x 2 (read data) + 10 (computation) + 10 x 2 (write data) = 50 cycles
- ⇨ MM: 32 x 2 (read data) + 16 (computation) + 16 x 2 (write data) = 102 cycles

Even in the worst-case scenario, the total time required would be 704 cycles. However, in the current implementation, it takes 1570 cycles to complete, indicating that there is still room for optimization.

(1) Why cause the cycles increase?

From Figure 12, we observe that the wishbone (WB) cycles entering SDRAM during the read data phase take 7 cycles each. This delay is attributed to the 3T internal CAS latency of SDRAM combined with 4 layers of buffers. Consequently, even with an external prefetch buffer, its effectiveness is limited as it gets emptied before it can be refilled due to the fast-reading pace.

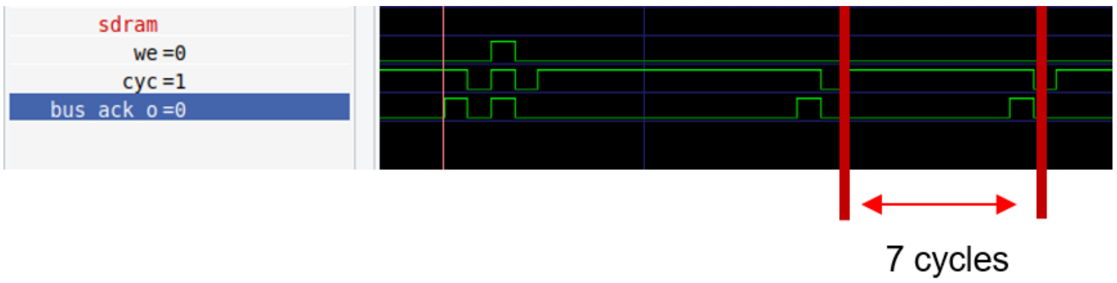


Figure. 12. Read data

(2) Solution for decrease the cycles of reading data.

Firstly, we will configure SDRAM in burst mode. As shown in Figure 13, even though the latency remains at 7 T, we can read 8 pieces of data in one burst. At this point, incorporating a prefetch buffer allows us to reduce the overall number of cycles for computation.

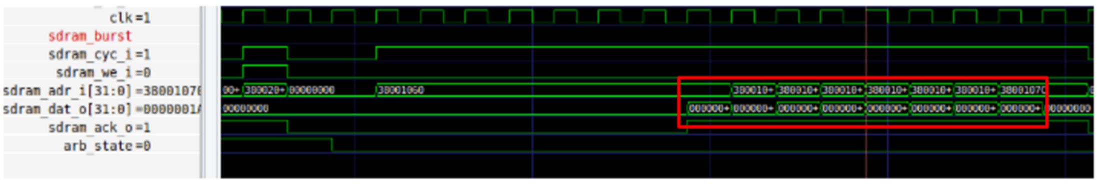


Figure. 13. burst mode of SDRAM.

(3) New design for SDRAM with burst mode.

Block Diagram:

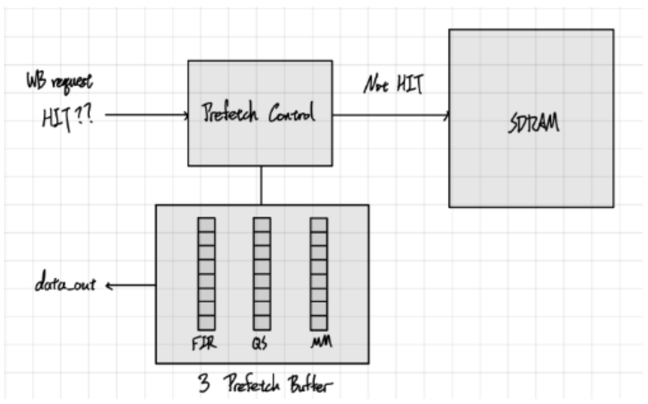


Figure. 14. Block diagram for new design

Prefetch buffer:

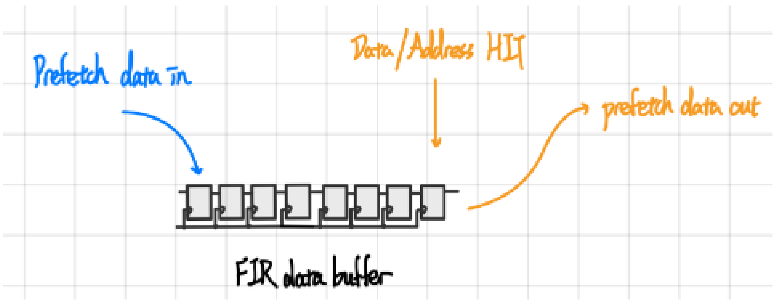
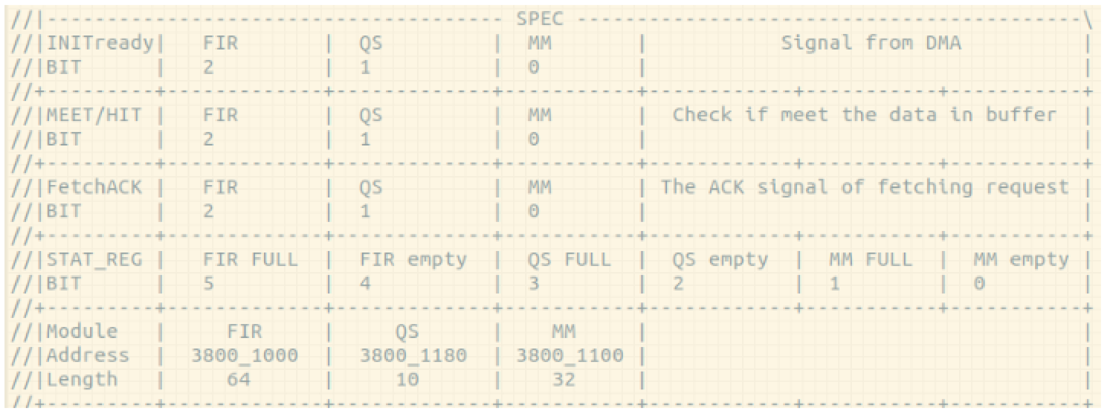


Figure.15. Prefetch buffer

We would design 3 prefetch buffer (FIR/MM/QS), here I use FIR buffer to explain our idea.

Our prefetch buffer acts like shift registers, it shifts out the stored data when the read access came. It prefetch data until all buffers are filled up before our workload start. If the buffer is empty, controller will send an Empty signal to tell the **arbiter** to let the priority of that buffer to be the last since the status of that buffer is busy, it is being filled up.

Prefetch Controller:



```
//|------------------------------------ SPEC ------------------------------------\
//|INITready|    FIR       |  QS          |  MM        |          Signal from DMA          |
//|BIT      |    2         |  1           |  0         |                                   |
//+---------+--------------+--------------+------------+-----------+-----------+-----------+
//|MEET/HIT |    FIR       |  QS          |  MM        |  Check if meet the data in buffer  |
//|BIT      |    2         |  1           |  0         |                                   |
//+---------+--------------+--------------+------------+-----------+-----------+-----------+
//|FetchACK |    FIR       |  QS          |  MM        | The ACK signal of fetching request |
//|BIT      |    2         |  1           |  0         |                                   |
//+---------+--------------+--------------+------------+-----------+-----------+-----------+
//|STAT_REG |    FIR FULL  |  FIR empty   |  QS FULL   |  QS empty |  MM FULL  |  MM empty |
//|BIT      |    5         |  4           |  3         |  2        |  1        |  0        |
//+---------+--------------+--------------+------------+-----------+-----------+-----------+
//|Module   |      FIR     |     QS       |    MM      |                                   |
//|Address  |   3800_1000  |  3800_1180   | 3800_1100  |                                   |
//|Length   |      64      |     10       |    32      |                                   |
//+---------+--------------+--------------+------------+-----------+-----------+-----------+
```

Above figure shows our design about prefetch controller.

- Set up
    - When setup(before all buffer are FULL), data will store in SDRAM first.
    - After we git the initial address (by firmware code) -> start prefetch
    - Fill the data until it buffers length, then set the state of that

buffer to “FULL”.

- After all prefetch buffers are “FULL” => START

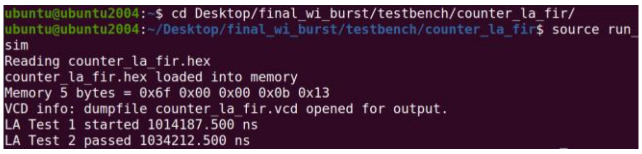
- Running
  - If input address meets the saving address -> HIT
  - If HIT, terminate the wishbone read request by send the wb_ack immediately.
  - If the buffer is Empty, start prefetch the data from SDRAM into buffer.

SDRAM burst length.

- Since our prefetch buffer have the length of 8, if we can achieve the burst length 8, it can fill up the empty buffer rapidly.

(4) Optimization result

The actual completion time for all three accelerators is 801 cycles.



```
ubuntu@ubuntu2004:~$ cd Desktop/final_wi_burst/testbench/counter_la_fir/
ubuntu@ubuntu2004:~/Desktop/final_wi_burst/testbench/counter_la_fir$ source run_
sim
Reading counter_la_fir.hex
counter_la_fir.hex loaded into memory
Memory 5 bytes = 0x6f 0x00 0x00 0x0b 0x13
VCD info: dumpfile counter_la_fir.vcd opened for output.
LA Test 1 started 1014187.500 ns
LA Test 2 passed 1034212.500 ns
```

Figure. 16. Total simulation

(5) Waveform analysis

From Figure 17, we observe that the Wishbone cycles issued by the arbiter are more frequent, allowing for faster data retrieval.

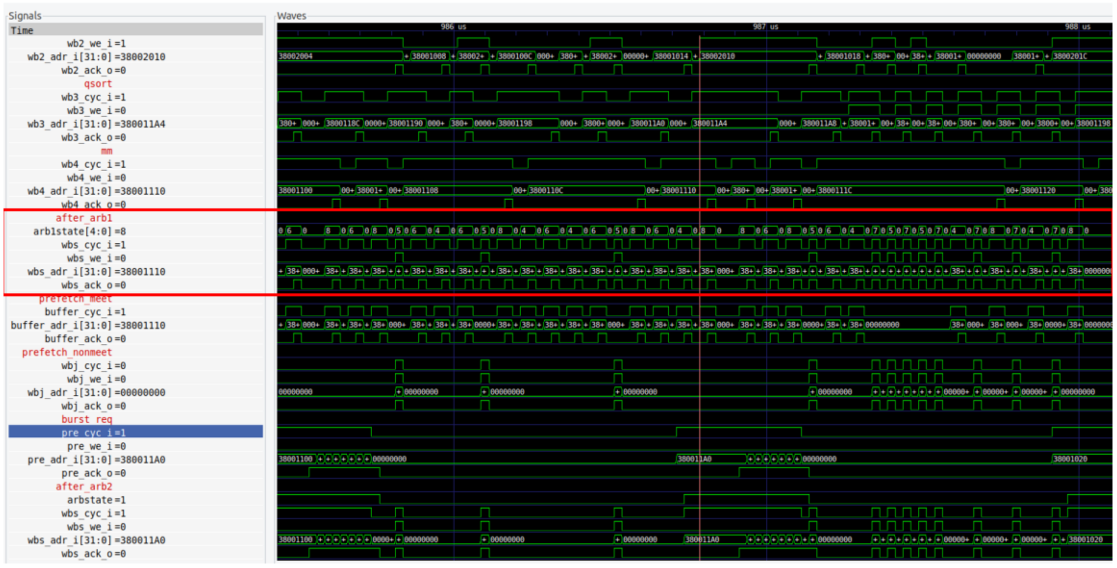
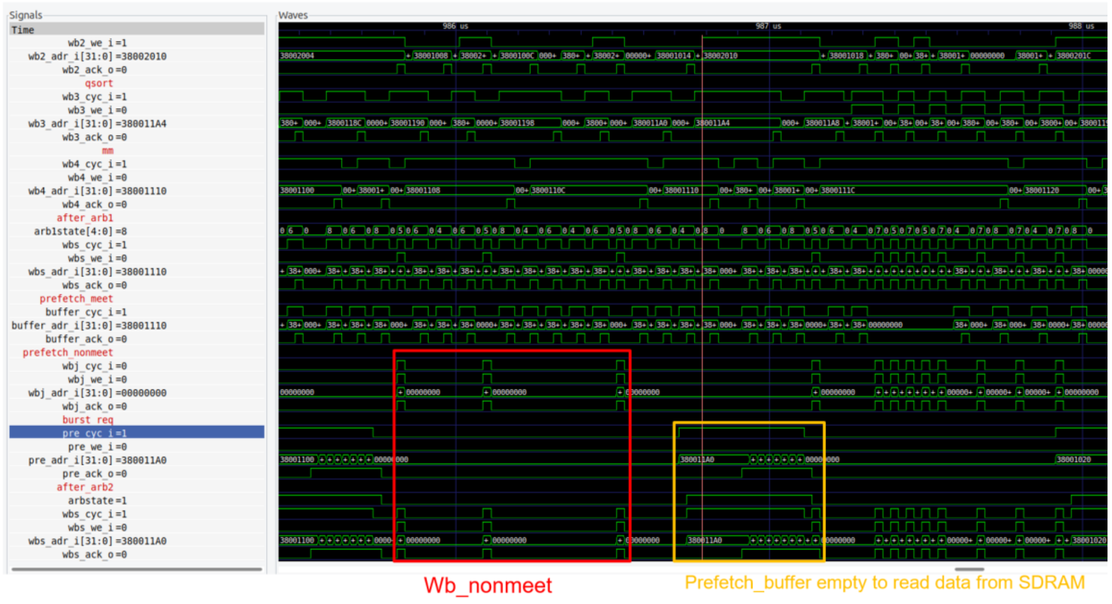


Figure. 17. WB cycles from arbiter

Additionally, from Figure 18, it is evident that entry into SDRAM only occurs when there is no "MEET" in the wishbone (WB) cycles, or when the prefetch buffer is empty during SDRAM read data. In other words, the WB cycles entering SDRAM are initiated either by the Arbiter or the Prefetch Buffer.



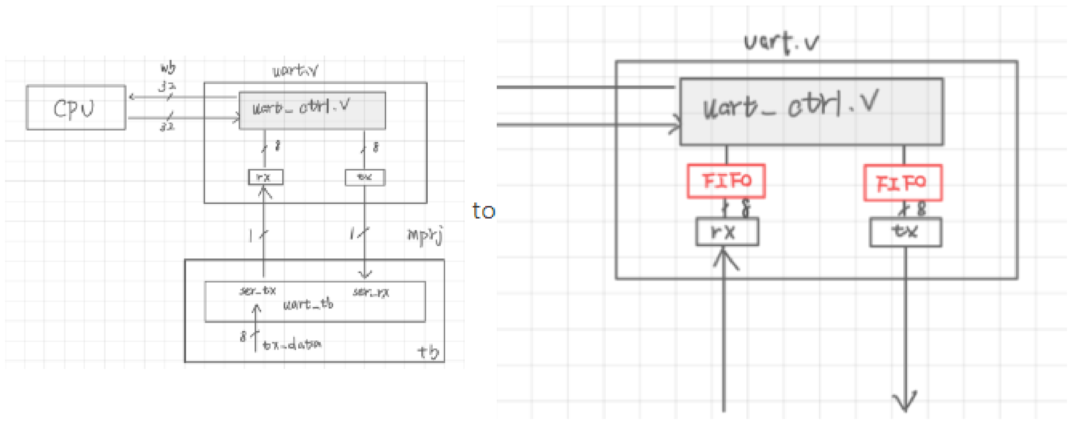
```
Signals
Time
        wb2_we_i=1
  wb2_adr_i[31:0]=38002010
       wb2_ack_o=0
           qsort
       wb3_cyc_i=1
        wb3_we_i=0
  wb3_adr_i[31:0]=380011A4
       wb3_ack_o=0
              mm
       wb4_cyc_i=1
        wb4_we_i=0
  wb4_adr_i[31:0]=38001110
       wb4_ack_o=0
      after_arb1
   arb1state[4:0]=8
       wbs_cyc_i=1
        wbs_we_i=0
  wbs_adr_i[31:0]=38001110
       wbs_ack_o=0
   prefetch_meet
    buffer_cyc_i=1
buffer_adr_i[31:0]=38001110
    buffer_ack_o=0
prefetch_nonmeet
       wbj_cyc_i=0
        wbj_we_i=0
  wbj_adr_i[31:0]=00000000
       wbj_ack_o=0
       burst_req
       pre_cyc_i=1
        pre_we_i=0
  pre_adr_i[31:0]=380011A0
       pre_ack_o=0
      after_arb2
        arbstate=1
       wbs_cyc_i=1
        wbs_we_i=0
  wbs_adr_i[31:0]=380011A0
       wbs_ack_o=0
```

Figure. 18. WB cycle to SDRAM

Part III: Communication

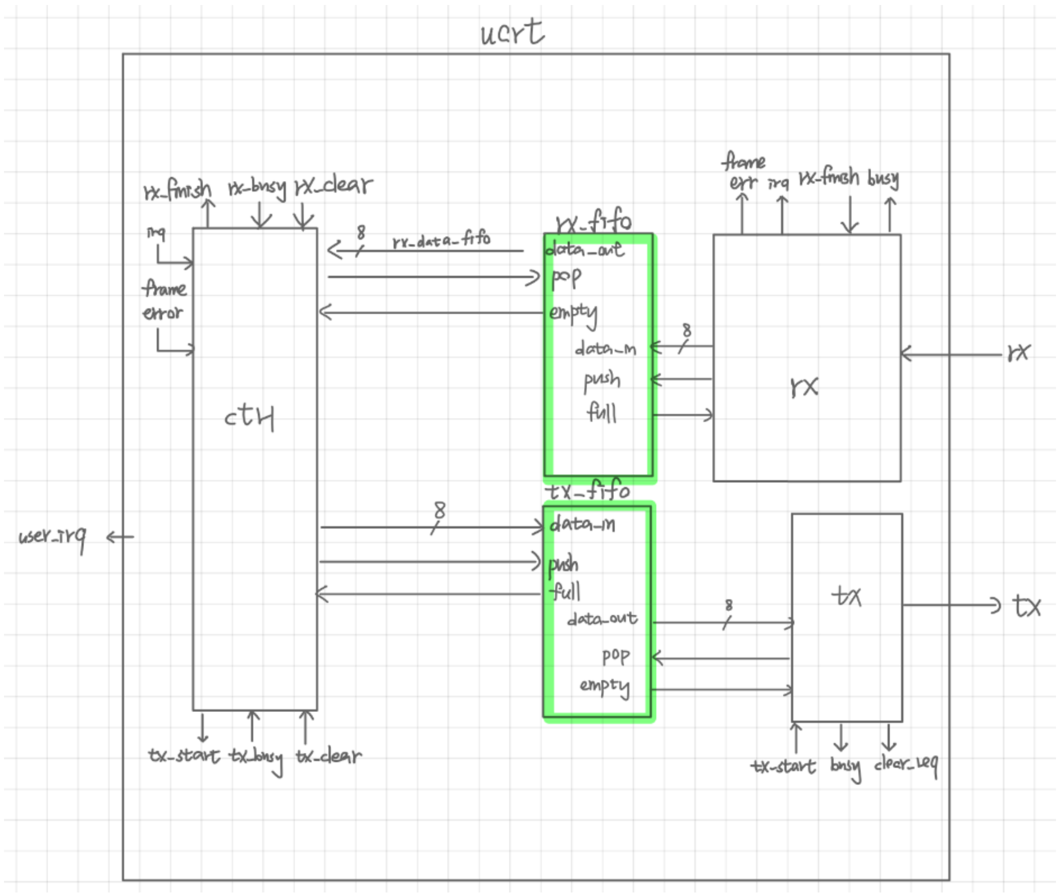
1. Design for Optimization

Original UART implementation flow vs. UART with FIFO

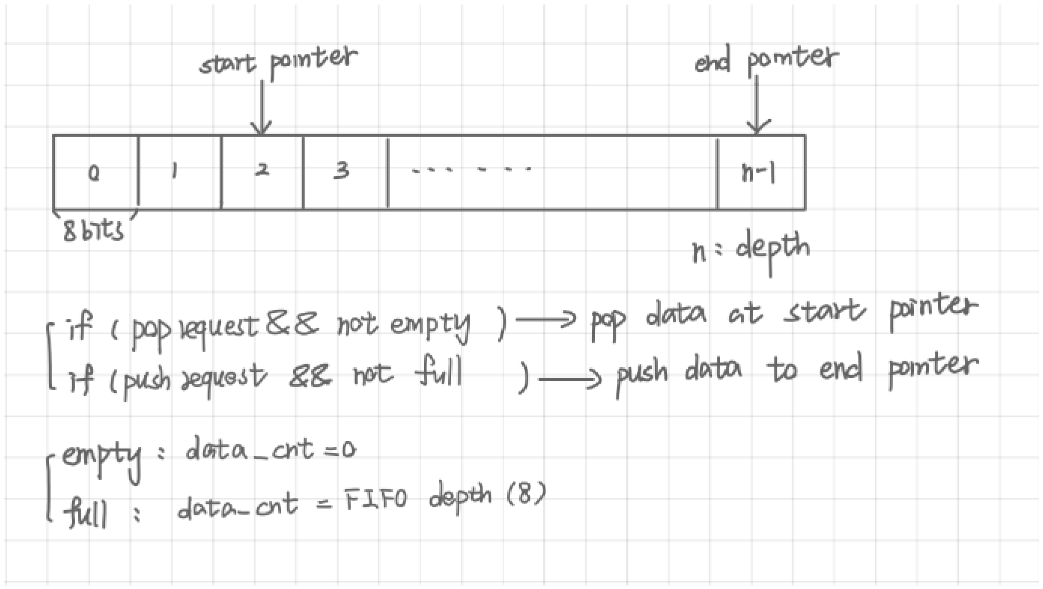


With FIFO, we can decrease the number of interrupts to make the execution faster since we can first keep the data sent in the buffer and wait until the buffer is full, then we send the data all at once.

2. How to Implement into Original Design



Signals in FIFO



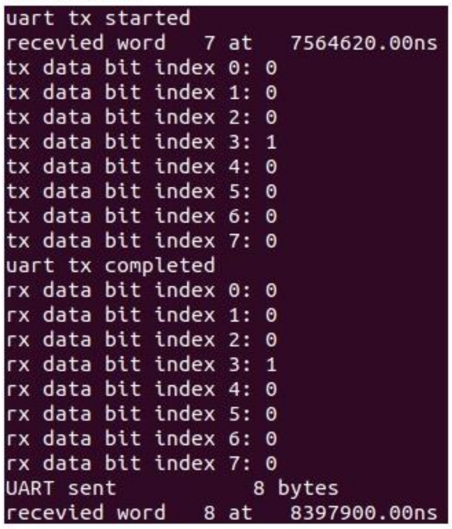
if (pop request && not empty) ⟶ pop data at start pointer
if (push request && not full) ⟶ push data to end pointer

empty : data_cnt = 0
full : data_cnt = FIFO depth (8)

3. Verification on Simulation

In simulation, we only sent 8 data.

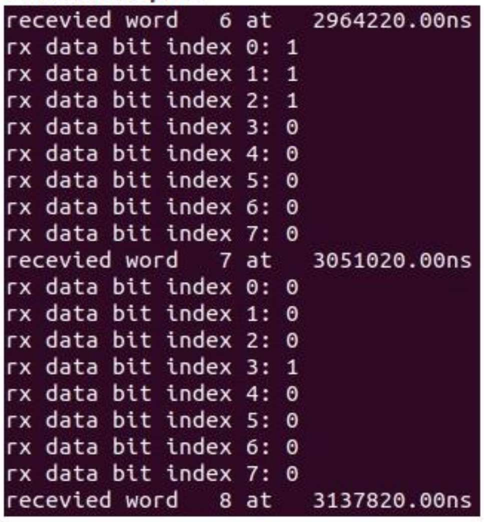
Without FIFO



```
uart tx started
recevied word   7 at   7564620.00ns
tx data bit index 0: 0
tx data bit index 1: 0
tx data bit index 2: 0
tx data bit index 3: 1
tx data bit index 4: 0
tx data bit index 5: 0
tx data bit index 6: 0
tx data bit index 7: 0
uart tx completed
rx data bit index 0: 0
rx data bit index 1: 0
rx data bit index 2: 0
rx data bit index 3: 1
rx data bit index 4: 0
rx data bit index 5: 0
rx data bit index 6: 0
rx data bit index 7: 0
UART sent           8 bytes
recevied word   8 at   8397900.00ns
```

Latnecy=8397900ns

FIFO with depth 4



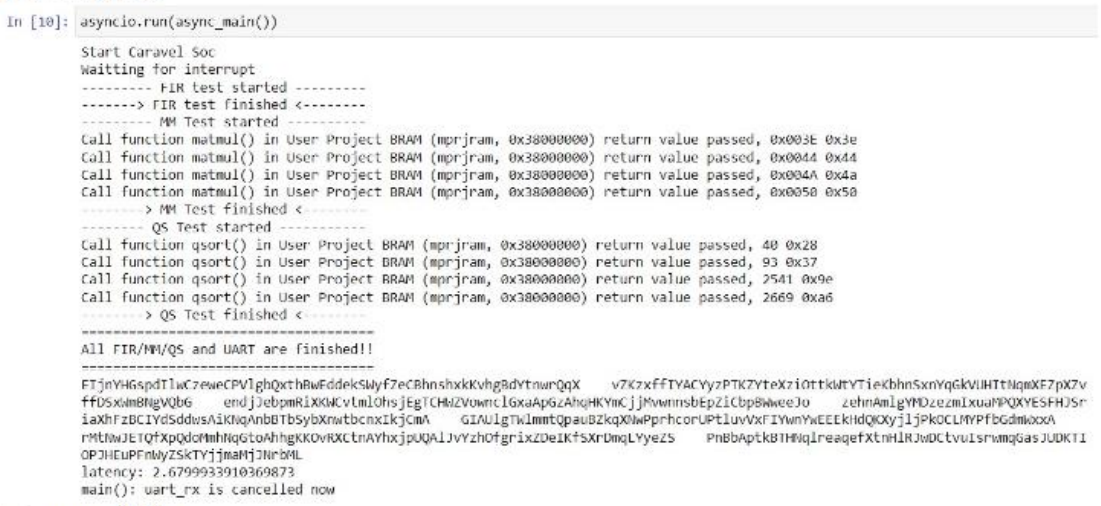
```
recevied word   6 at   2964220.00ns
rx data bit index 0: 1
rx data bit index 1: 1
rx data bit index 2: 1
rx data bit index 3: 0
rx data bit index 4: 0
rx data bit index 5: 0
rx data bit index 6: 0
rx data bit index 7: 0
recevied word   7 at   3051020.00ns
rx data bit index 0: 0
rx data bit index 1: 0
rx data bit index 2: 0
rx data bit index 3: 1
rx data bit index 4: 0
rx data bit index 5: 0
rx data bit index 6: 0
rx data bit index 7: 0
recevied word   8 at   3137820.00ns
```

Latnecy=1966020ns

4. **Verilification on FPGA**

   In FPGA, we sent 512 data.

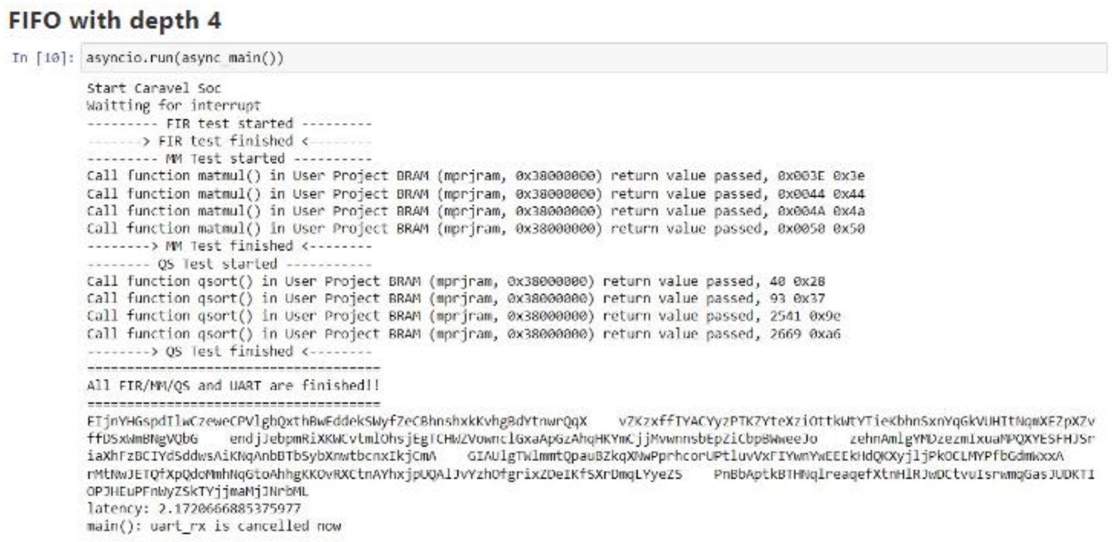
Without FIFO



```
In [10]: asyncio.run(async_main())

         Start Caravel Soc
         Waitting for interrupt
         --------- FIR test started ---------
         -------> FIR test finished <--------
         --------- MM Test started ----------
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003E 0x3e
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044 0x44
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004A 0x4a
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050 0x50
         -------> MM Test finished <--------
         -------- QS Test started -----------
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 40 0x28
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 93 0x37
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 2541 0x9e
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 2669 0xa6
         -------> QS Test finished <--------
         -------------------------------------
         All FIR/MM/QS and UART are finished!!
         -------------------------------------
         FTjnYHGspdTlwCzeweCPVlghQxthBwFddekSWyf7eCBhnshxkKvhgBdYtnwrQqX    v7KzxffTYACYyzPTKZYteXziOttkWtYTieKbhnSxnYqGkVUHTtNqmXF7pX7v
         ffDSxWmBNgVQbG    endjJebpmRiXKWCvtmlOhsjEgTCHWZVowmclGxaApGzAhqHKYmCjjMvwnnsbEpZiCbpBWweeJo    zehnAmlgYMDzezmIxuaMPQXYESFHJSr
         iaXhFzBCIYdSddwsAiKNqAnbBTbSybXnwtbcnxIkjCmA    GIAUlgTWlmmtQpauBZkqXNwPprhcorUPtluvVxFIYwnYwEEEkHdQKXyjljPkOCLMYPfbGdmWxxA
         rMtNwJETQfXpQdoMmhNqGtoAhhgKKOvRXCtmAYhxjpUQAlJvYzhOfgrixZDeIKfSXrDmqLYyeZS    PnBbAptkBTHNqlreaqefXtnHlRJwDCtvuIsrwmqGasJUDKTI
         OPJHEuPFnWyZSkTYjjmaMjJNrbML
         latency: 2.6799933910369873
         main(): uart_rx is cancelled now
```

Latency = 2.68s



FIFO with depth 4

```
In [10]: asyncio.run(async main())

         Start Caravel Soc
         Waitting for interrupt
         --------- FIR test started ---------
         -------> FIR test finished <--------
         --------- MM Test started ----------
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x003E 0x3e
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0044 0x44
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x004A 0x4a
         Call function matmul() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x0050 0x50
         --------> MM Test finished <--------
         -------- QS Test started -----------
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 40 0x28
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 93 0x37
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 2541 0x9e
         Call function qsort() in User Project BRAM (mprjram, 0x38000000) return value passed, 2669 0xa6
         --------> QS Test finished <--------
         -------------------------------------
         All FIR/MM/QS and UART are finished!!
         =====================================
         ETjnYHGspdIlwCzeweCPVlghQxthBwEddekSWyfZeCBhnshxkKvhgBdYtnwrQqX    vZKzxffIYACYyzPTKZYteXziOttkWtYTieKbhnSxnYqGkVUHTtNqmXEZpX7v
         ffDSxWmBNgVQbG    endjJebpmRiXKWCvtmlOhsjEgTCHWZVowncIGxaApGzAhqHKYmCjjMvwnnsbEpZiCbpBWweeJo    zehnAmlgYMDzezmIxuaMPQXYESFHJSr
         iaXhFzBCIYdSddwsAiKNqAnbBTbSybXnwtbcnxIkjCmA    GIAUlgTWlmmtQpauBZkqXNwPprhcorUPtluvVxFIYwnYwEEEkHdQKXyjljPkOCLMYPfbGdmkxxA
         rMtNwJETQfXpQdoMmhNqGtoAhhgKKOvRXCtnAYhxjpUQAlJvYzhOfgrixZDeIKfSXrDmqLYyeZS    PnBbAptkBTHNqlreagefXtnHlRJwDCtvuIsrwmqGasJUDKTI
         OPJHEuPFnWyZSkTYjjmaMjJNrbML
         latency: 2.1720666885375977
         main(): uart_rx is cancelled now
```

Latency = 2.17s

Part IV: Performance Summary

⇨ Computation

	Software(cycles)	Hardware without prefetch(cycles)	Hardware with prefetch (cycles)
MM	55303	756	X(no test)
FIR	65890	1471	X(no test)
QS	14394	315	X(no test)
Total	135587	1570	801

⇨ Communication

	Latency(cycle * period)	Metric(ms)	Improvement
Without FIFO	114582 * 25ns	44.78	*
FIFO depth 4	54076 * 25ns	10.48	4.27x