SOC FINAL

第 5 組
張耀明
蕭翔
陳佳詳

---

1. UART Acceleration

1. System Structure

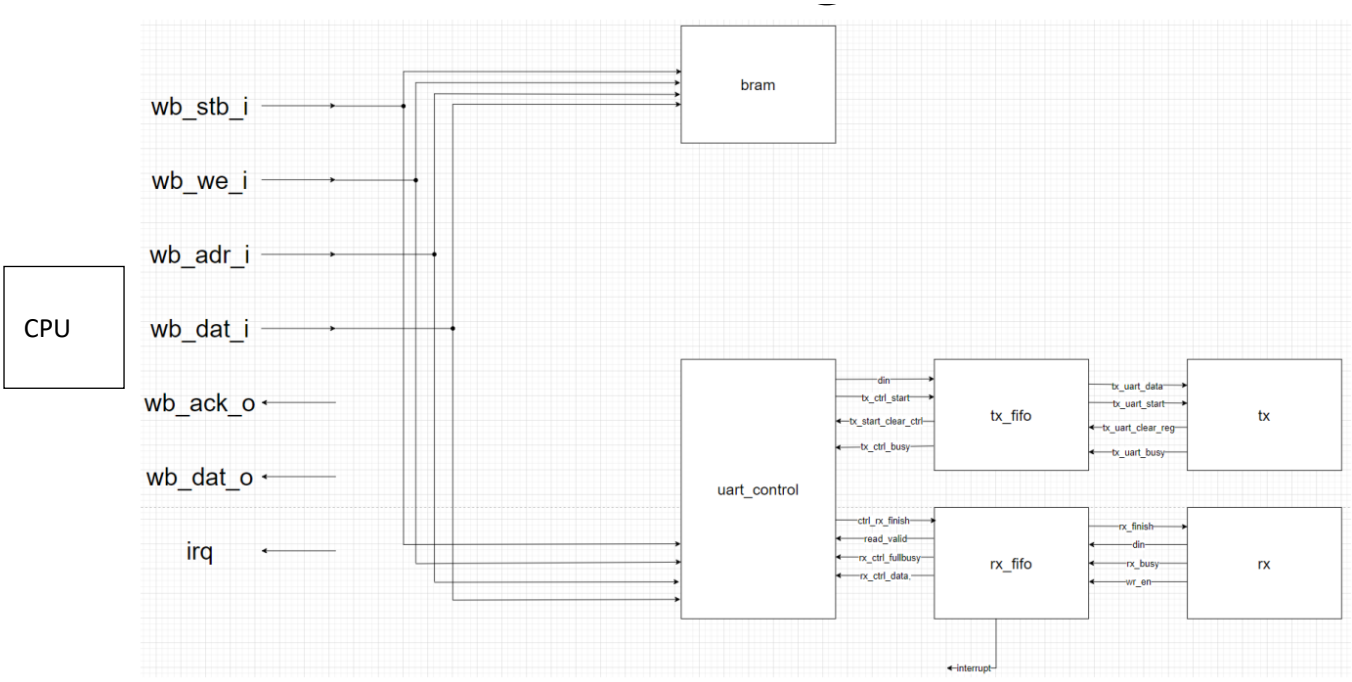


Figure 1

The Structure is shown on Figure 1. It is implemented in user-project of the Caravel soc platform which CPU utilizes wishbone bus to communicate with user project area. In the user project area, there are Bram and UART module. The BRAM stores the instructions to be executed by the CPU. On the other hand, the UART module implements the UART (Universal Asynchronous Receiver/Transmitter) to receive and transmit data with other devices.

In the lab6, the UART module is designed to receive only one byte of data and can transmit only one byte of data at a time. When the Rx module receives a byte of data, it generates an interrupt request signal, prompting the CPU to execute the interrupt service routine (ISR) to store the Rx data and then send it back through the Tx module. Therefore, the execution of Tx and Rx cannot work simultaneously, which is not efficient.

In this final project, the uart_control module and the Tx, Rx module remain unchanged. We designed and added Tx_FIFO and Rx_FIFO module between the Tx, Rx and uart_control. Additionally, we redesigned the interrupt service routing to make advantage of FIFO design.

2. Hardware design

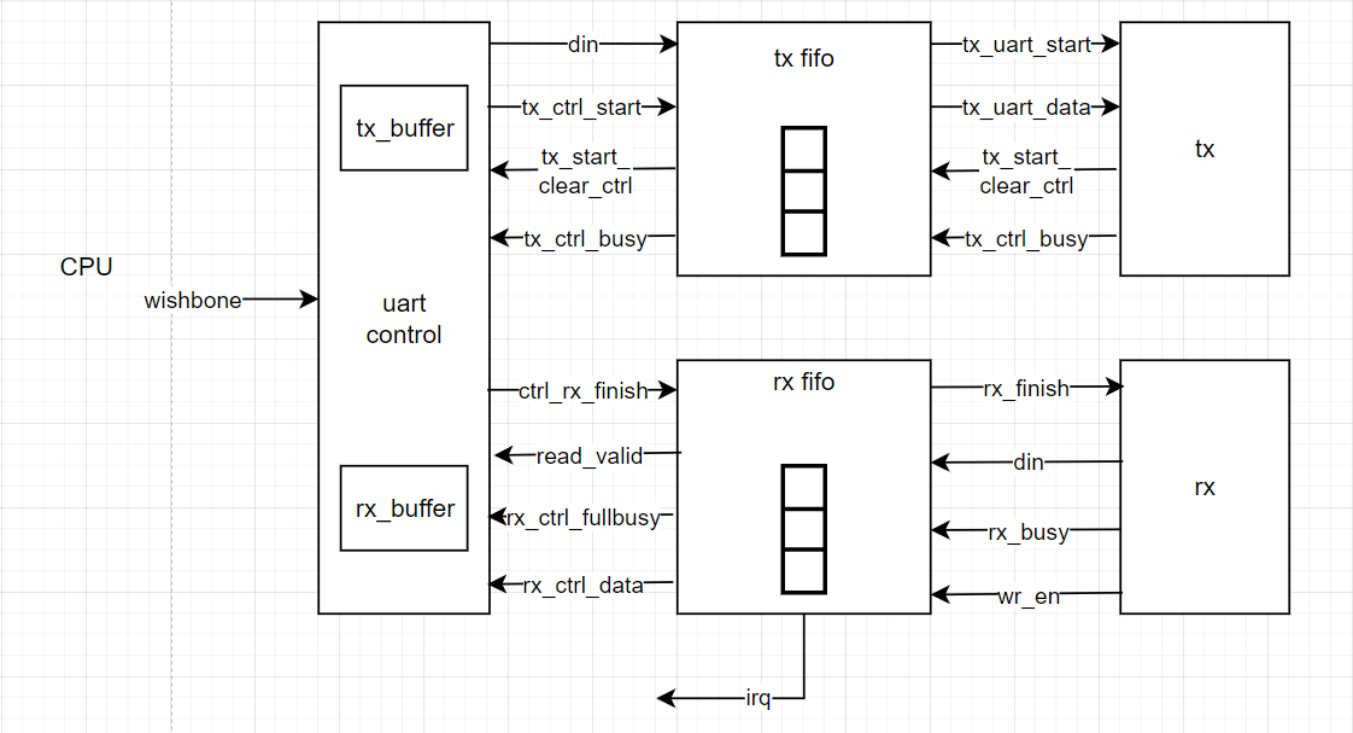
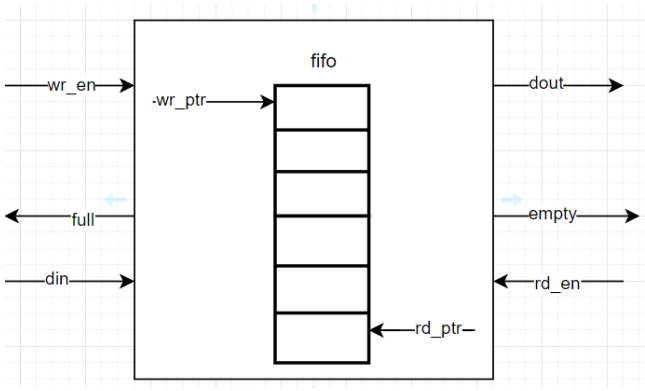


figure 2

Figure 2 shows the hardware structure of the UART module.

a. FIFO design



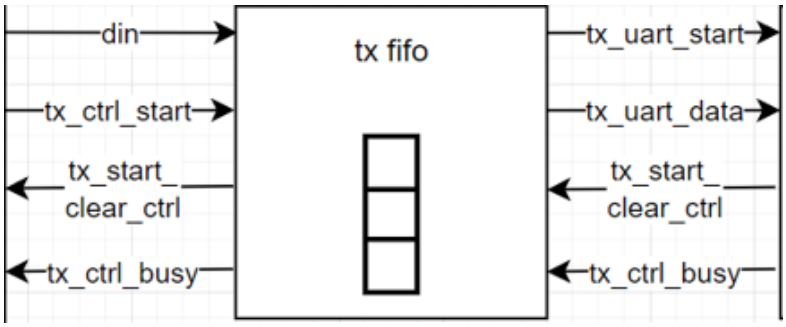

```
assign full  = (wr_ptr + 1  == rd_ptr)? 1'b1:1'b0;
assign empty = (wr_ptr == rd_ptr)? 1'b1:1'b0;
```

figure 3

The figure 3 displays the synchronized FIFO design employed in the Rx and Tx FIFO. The wr_ptr indicates to the space that the next input data from the din port will be stored, and the rd_ptr indicates to the space that stores the next data to be output.

When 'wr_en' is asserted, a new data is received, and the 'wr_ptr' is incremented by 1. Similarly, when 'rd_ptr' is asserted, a data is read, and the 'rd_ptr' is incremented by 1. Additionally, When the wr_ptr equals to rd_ptr, it indicates that the FIFO is full. When the wr_ptr equals to rd_ptr+1, it indicates that the FIFO is empty.

b. Tx FIFO

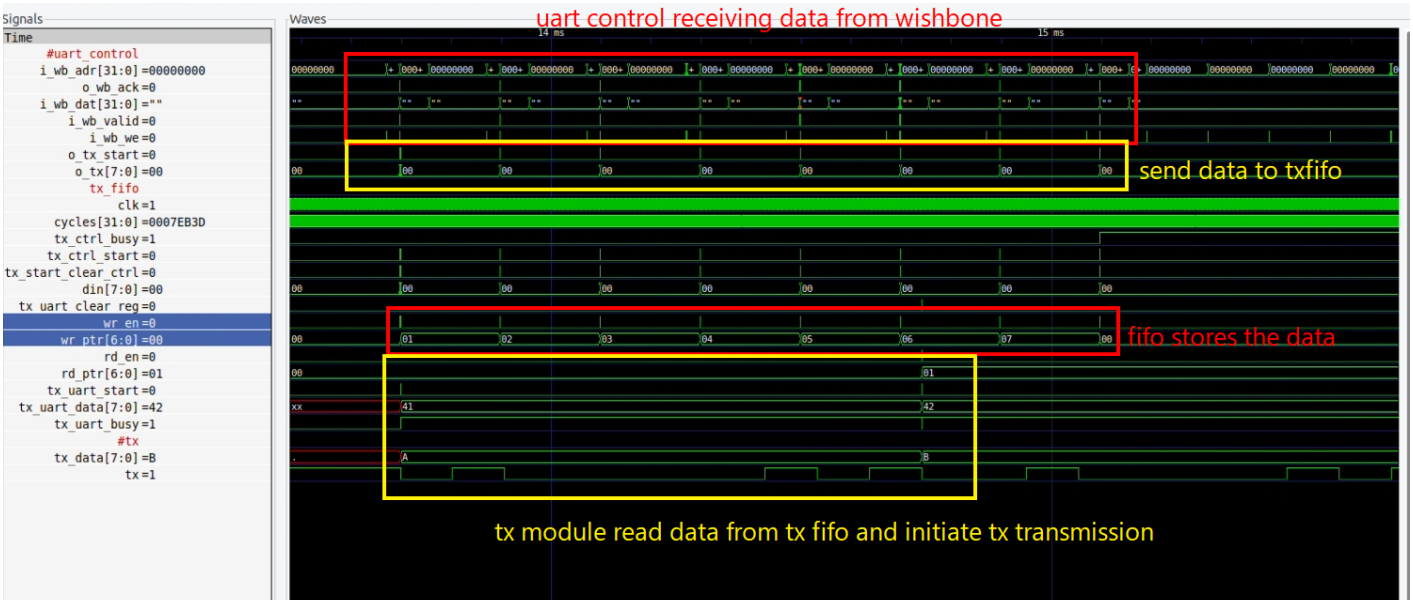


Singal:

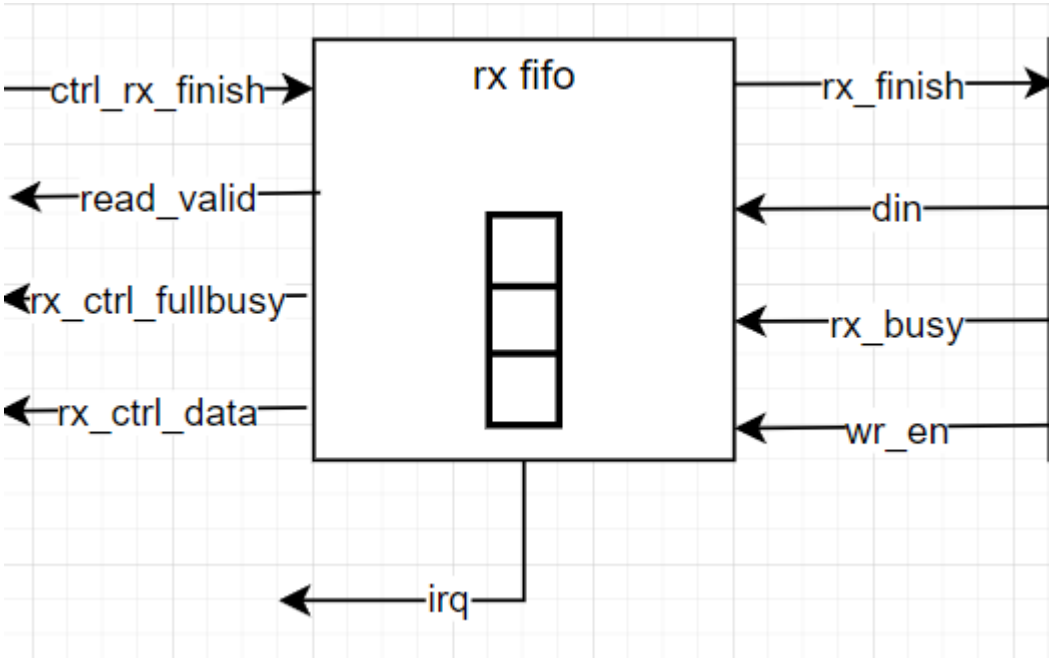
The block diagram of Tx_FIFO module is as shown above. With Tx_FIFO, CPU can send multiple data through wishbone bus to Tx to initiate the transmission. The Tx_FIFO receives data from the CPU through the din port when the tx_ctrl_start, which acts as the wr_en of the FIFO, is asserted. After receiving the data, the FIFO sends the tx_start_clear_ctrl to clear the tx_ctrl_start signal. If tx_FIFO is full, the tx_ctrl_busy is asserted to avoid the UART control sending new data.

If the FIFO is not empty, and neither tx_start_clear_ctrl nor tx_ctrl_busy is asserted, the tx_uart_start is asserted to initiate a Tx transmission util the FIFO is empty. And then, the Tx module sends the tx_start_clear_ctrl back to the tx_FIFO, which acts as the rd_en of the FIFO, to clear the tx_uart_start signal.

Waveform



c. Rx FIFO



The above displays the block diagram of Rx_FIFO

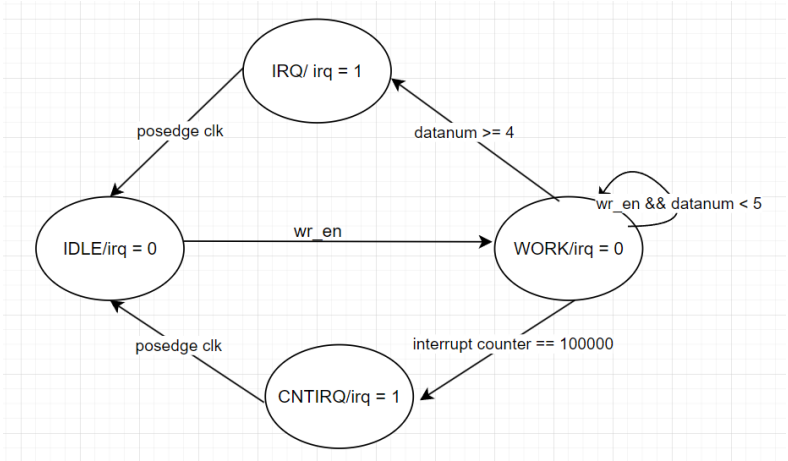
Signal

When data is being received through Rx, the Rx_busy is asserted until the handshake is finished. After receiving the data, wr_en is asserted, and the Rx_fifo receives and stores the data from din port.

In the UART control module, there is a buffer that can store one Rx_data. When the buffer is empty, the ctrl_rx_finish is asserted to request data from Rx_FIFO. If the Rx_FIFO is not empty, the read_valid is set to HIGH after the ctrl_Rx_finish is asserted, then the data stored in Rx_FIFO is sent to UART control module through Rx_ctrl_data.

Interrupt

In this Rx_FIFO module, we designed a finite state machine to generate interrupt signal to prompt the CPU to receive the data which is as shown below:



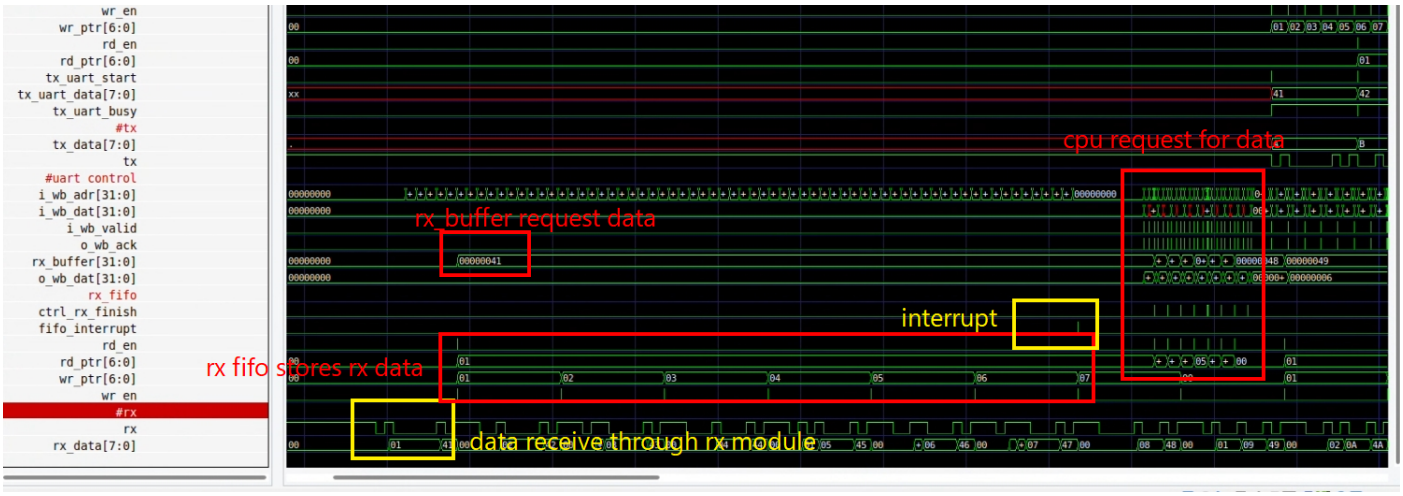
When wr_en is asserted (indicating new Rx data received), the state transitions to WORK.

In the WORK state, if the number of data stored in the FIFO reaches 5, the state transitions IRQ.

While in the WORK state, an interrupt counter is activated, and the value increases by 1 every cycle. If wr_en is asserted, the counter is reset to zero and continue incrementing. When the counter reaches to 100000, the state transitions to CNTIRQ.

In both CNTIRQ and IRQ state, the irq signal is set to 1 and the state transitions back to IDLE in the next clk cycle.

Waveform



3. Firmware design

To test the performance of UART with FIFO design, we modified the firmware design in LAB6 for this project. In LAB6, when the interrupt occurs, the CPU executes interrupt service routine to store the one byte of data received through Rx and send it back to Tx. However, in this project, Rx _FIFO generates interrupt after receiving 5 bytes of data.

isr.c:

```
void isr(void)
{

#ifndef USER_PROJ_IRQ0_EN

    irq_setmask(0);


#else
    uint32_t irqs = irq_pending() & irq_getmask();
    int buf;

    if ( irqs & (1 << USER_IRQ_0_INTERRUPT)) {
        user_irq_0_ev_pending_write(1); //Clear Interrupt Pending Event
        interrupt_flag();

    }
```

When an interrupt occured, CPU executes the interrupt_flag() function. All that function does is just set intr_flag to 1;

```
// check ebd
void __attribute__ ((section(".mprj"))) interrupt_flag(){
    intr_flag = 1;
}
```

Finally, the CPU complete the interrupt service routine. Instead of receiving UART data in interrupt service routine, we access the UART module using the polling

method. After receiving and send the end character '\n', the program ends, and send 0xAB51 to testbench

```
void __attribute__ ( ( section ( ".mprjram" ) ) ) main_loop(){
    int finish = 0;
    char buffer[10];
    int idx = 0;
    int i = 0;
    while(1){
        if(intr_flag){
            if(((reg_uart_stat) & 0x00000002 ) == 0x00000002 && idx < uart_fifo_depth){
                int data = uart_read();
                char c = (char)data;
                if(data == 0x0a) finish = 1;
                buffer[idx++] = c;
            }else{
                intr_flag = 0;
                i = 0;
            }
        }else{
            uart_write_string(buffer,idx);
            idx = 0;
        }
        if(idx ==0 && finish)break;
    }
    endflag = 1 ;
}
```

check if an interrupt has occured

store all Rx data in a char array

reset the intr_flag

send all Rx data back through Tx module

Waveform：

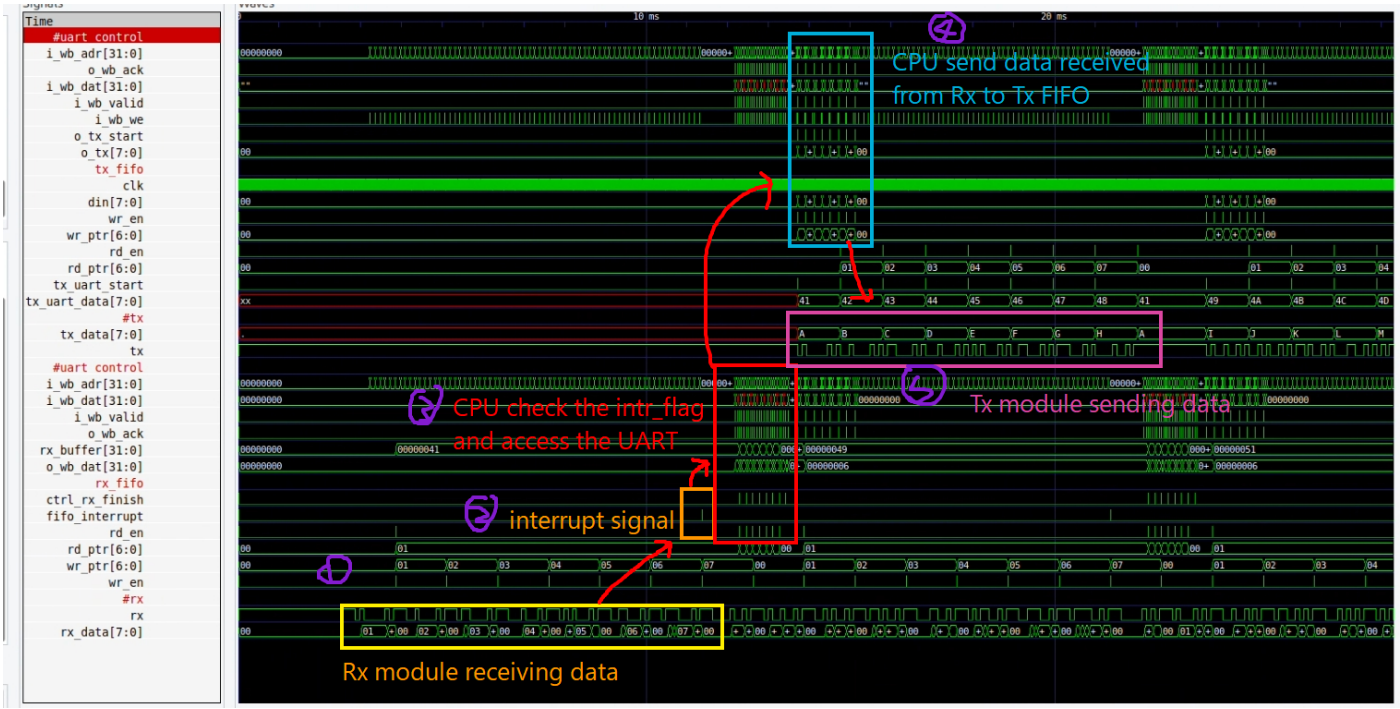
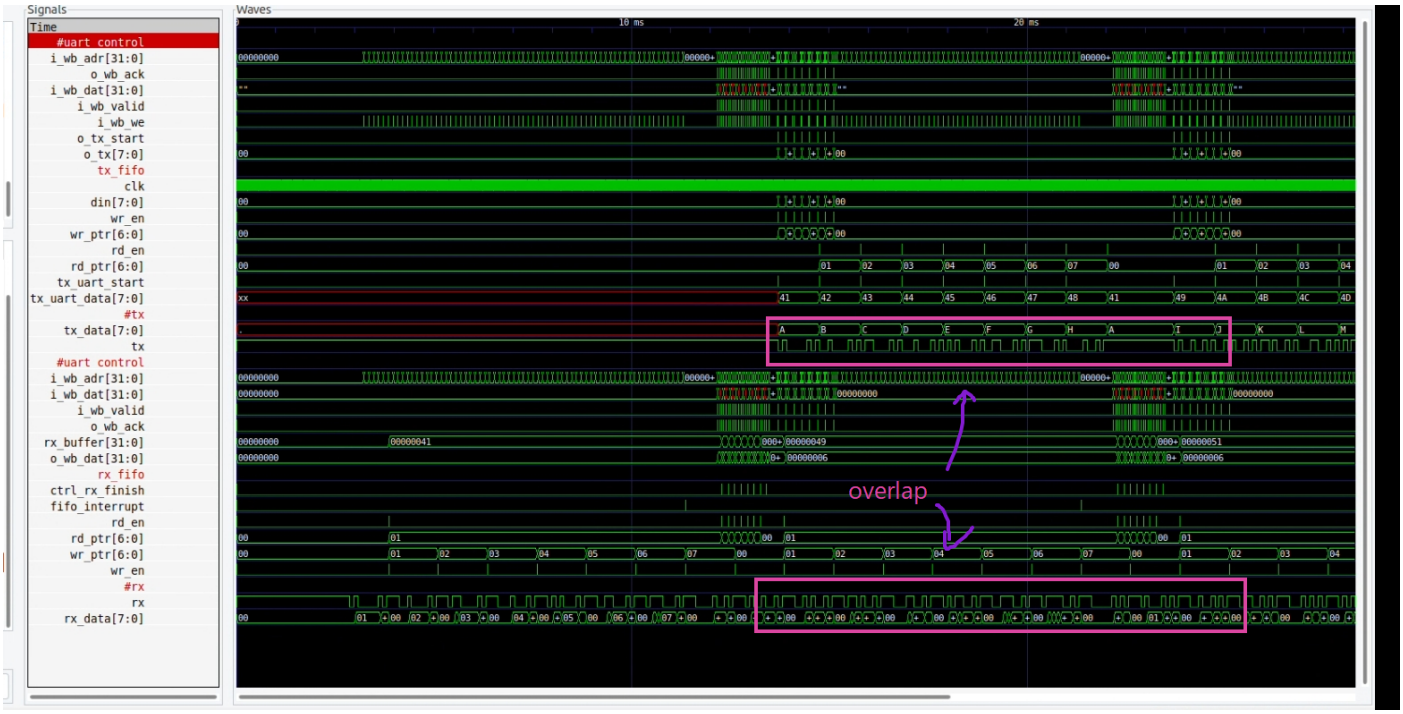


Figure 4

4. Performance

   It is evident in the waveform that the Rx and Tx modules can work simultaneously. We can observe the overlap of the Rx and Tx signal on the waveform.



In the LAB6, when a byte of data is received, the Rx_module generates interrupt. However, in our project, when 5 bytes of data are stored in the fifo, the Rx_fifo generates interrupt. With the rx_fifo, the number of interrupt signal decreases, reducing the overall interrupt overhead of system.

In this project, we've tested the cycle count of receiving and sending 512 bytes of data.

But for the UART system without FIFO, it takes too much memory space and time to execute. We tested the cycle count of receiving and sending 2 bytes of data, and multiply by 256 to calculate the total cycle count.

Uart with FIFO	Uart without fifo(LAB6)
26005948 cycles	59413248cycles

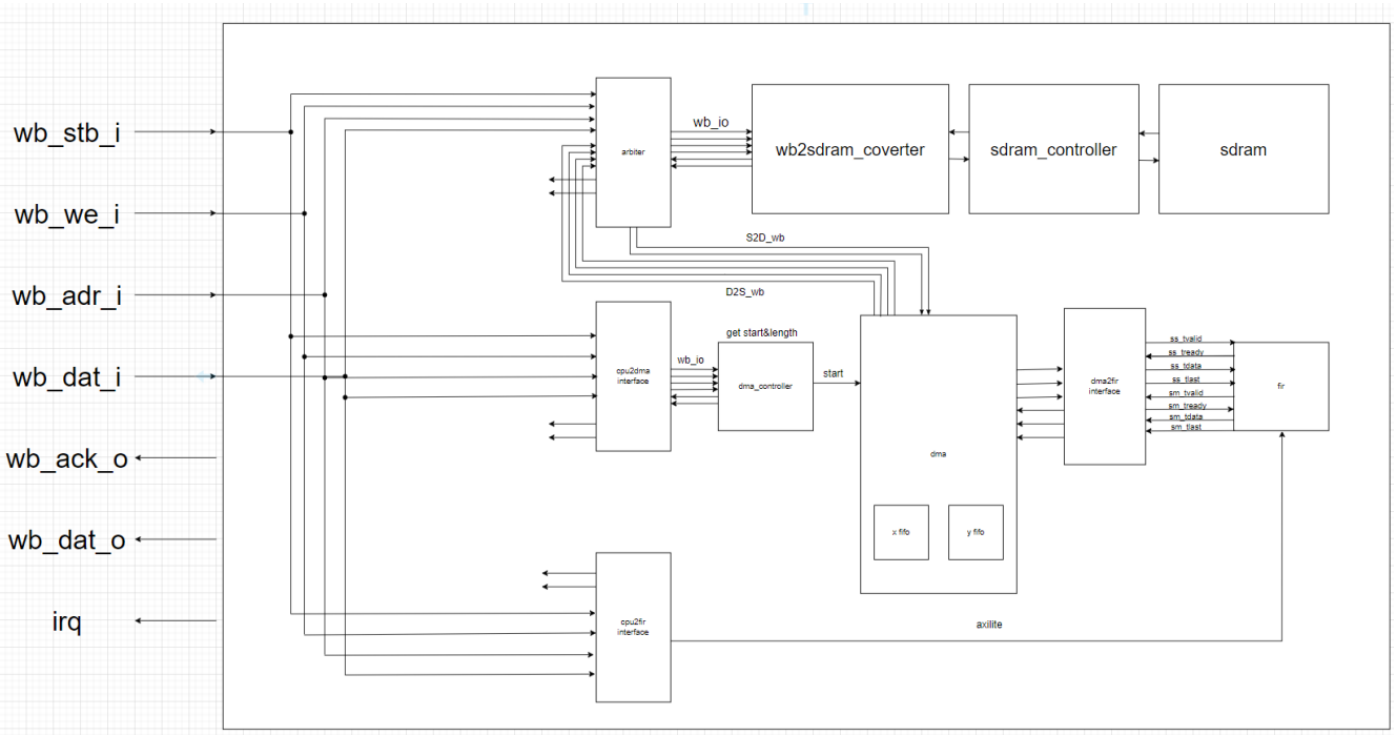
$$\frac{26005948}{59413248} * 100\% = 43.77\%$$

Compared to the UART without FIFO, our UART with FIFO utilizes only 43.77% of the cycles.

2. FIR Acceleration

1. System structure

   The system structure is as shown below.



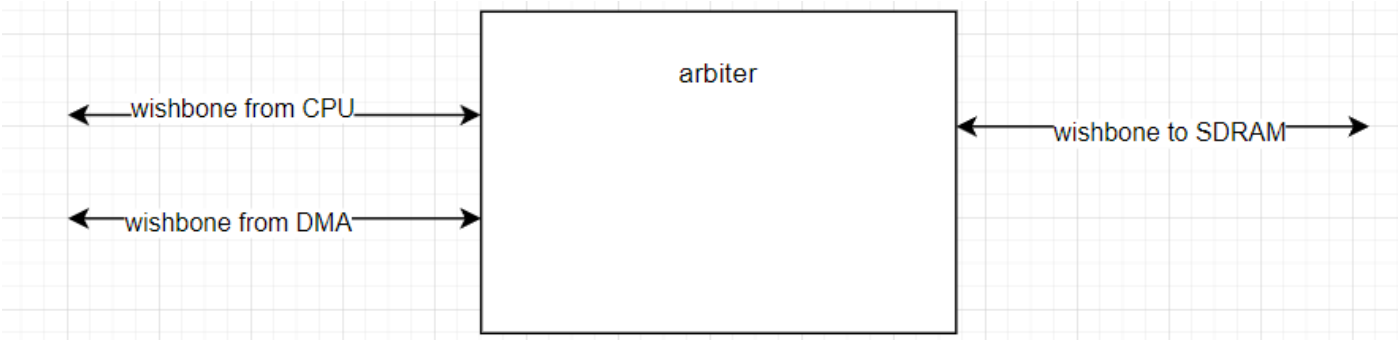
The FIR acceleration topic aims to speed up FIR execution using an FIR acceleration. In the lab4-2, we integrated the FIR accelerator into the user project area and transmit data between CPU and accelerator. However, due to the long latency in reading input data from BRAM to the CPU and sending them to the accelerator, FIR execution took a long time. In this project, our goal is to enhance execution speed by implementing DMA and improving system RAM performance by replacing the user BRAM used in lab4-2 with SDRAM.

With DMA and SDRAM, the FIR can access the system RAM directly to obtain input data. Additionally, the SDRAM design reduces the latency of receiving data and instruction from system RAM, improving the utilization of FIR accelerator and decreasing execution time. However, with DMA, there are two hardware components are accessing the system RAM, necessitating an arbiter.

2. Hardware design

Arbiter

In this system, an arbiter is essential because the CPU and DMA may access the SDRAM through wishbone bus simultaneously. The arbiter determines which wishbone source has priority for accessing the SDRAM first.

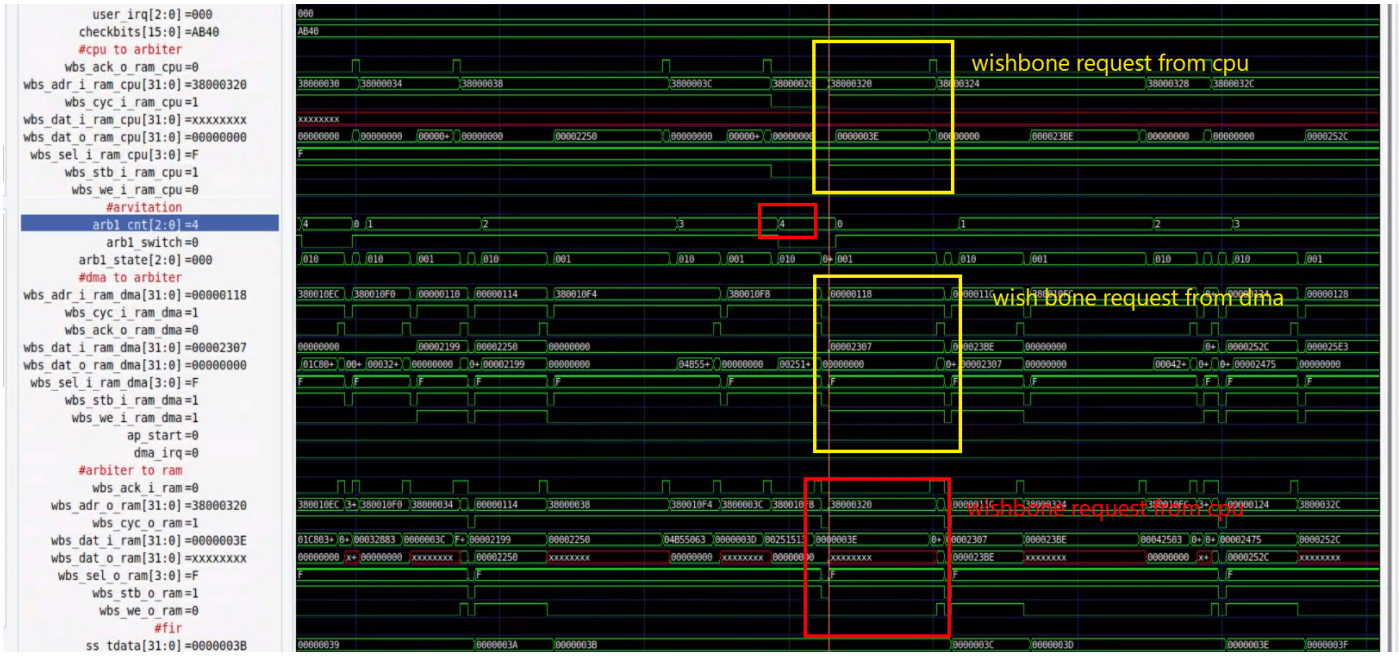


The arbiter has two Wishbone inputs: one from the CPU and one from the DMA. The arbiter prioritizes the signal that arrives first. Until the ack for this signal is received, the arbiter does not allow the other signal to intervene. That is, if the wishbone signal from CPU arrives first, the arbiter sends the wishbone signal from CPU to SDRAM. Before the SDRAM generates the wishbone ack to the CPU, the arbiter don't send the wishbone signal from DMA to the SDRAM.

If the wishbone requests from the DMA and CPU arrive at the same time, the DMA has priority for accessing the SDRAM first. However, if DMA keeps accessing the SDRAM for an extended period, the firmware runs slowly, leading to low efficiency. Therefore, we design a counter in the arbiter.

If the arbiter allows the DMA accesses the SDRAM, the counter increases by 1. When the value reach 4, the counter stop increasing, and the CPU has the priority for accessing the SDRAM first. If the wishbone request of CPU arrives and the counter value is 4, the counter reset to 0.

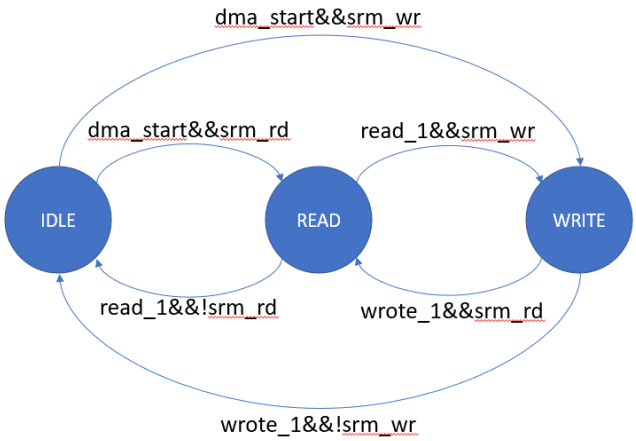
To sum up, if the counter value is less than 4, the DMA has priority for accessing the SDRAM first, if the counter value is equal to 4, the CPU has priority for accessing the SDRAM first.



The waveform shows that when the counter = 4, the CPU has the priority for accessing the SDRAM first.

DMA

State Diagram



There are three states in total, with "smr_rd" and "smr_wr" serving as transition signals for state changes.

IDLE State: In this state, no specific actions are taken.

READ State: The DMA sends a Wishbone signal to the SDRAM to read input data.

WRITE State: The DMA sends a Wishbone signal to the SDRAM to write back output data.

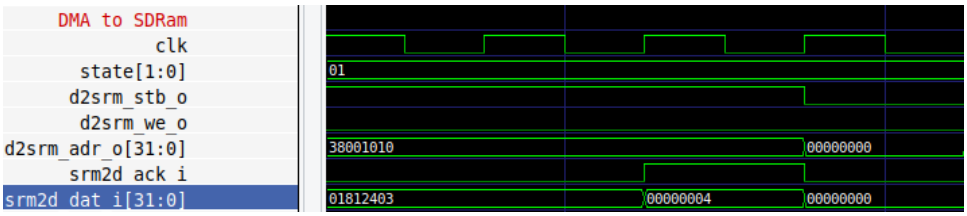
Within the DMA, there are X_fifo and Y_fifo.

When X_fifo is not full, the DMA enters the READ state. After X_fifo is full, the DMA checks whether Y_fifo is empty. If Y_fifo is not empty, the DMA enters the WRITE state. If Y_fifo is empty, the DMA returns to the IDLE state.

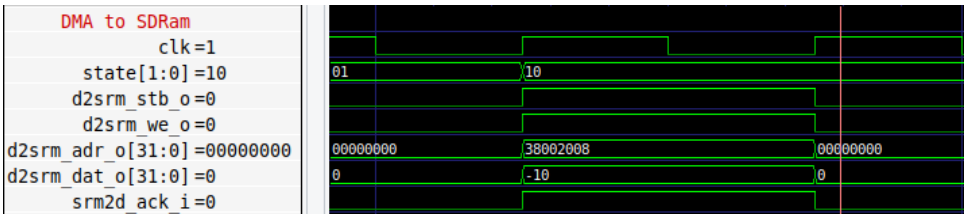
Additionally, the DMA will not enter the READ state when there is no data to be read from SDRAM, and it will not enter the WRITE state when there is no data to be written back to SDRAM.

Initially, the DMA obtains the start address, length, and save address by receiving the Wishbone signals from the CPU. Once all the necessary parameters are ready, the DMA is initiated.

Upon initiation, the DMA first enters the READ state. It starts sending Wishbone signals to the SDRAM to capture input data. The "wb_addr" is determined based on the start address and length. The data read back is stored in X_fifo. When X_fifo is not empty, the data is then sent to the FIR through AXI-Stream.



When Y_fifo is not full, FIR stores the calculated data into Y_fifo through axi-stream. After the DMA enters the WRITE state, the DMA sends the wishbone signal to the SDRAM to write the data back. The wb_addr will be determined by the save address and length.



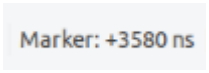
When Y_fifo is not full, the FIR stores calculated data into Y_fifo through AXI-Stream. Upon entering the WRITE state, the DMA sends a Wishbone signal to the SDRAM to write back data, with the 'wb_addr' determined by the save address and length.

When FIR sends 'sm_tlast' to DMA, it signifies the completion of all computations, and the results are returned to Y_fifo. DMA can confirm the completion of writing back all signals to SDRAM by verifying the truth of 'sm_tlast' and ensuring that Y_fifo is empty.

When DMA verifies the successful delivery of the last piece of data to SDRAM, it generates an interrupt.

Total transfer(ap_start to dma_irq)

As shown in the above diagram, when DMA writes the last data into SDRAM, it will trigger the DMA interrupt



Total transfer time:3580ns

Total cycle count:716

SDRAM

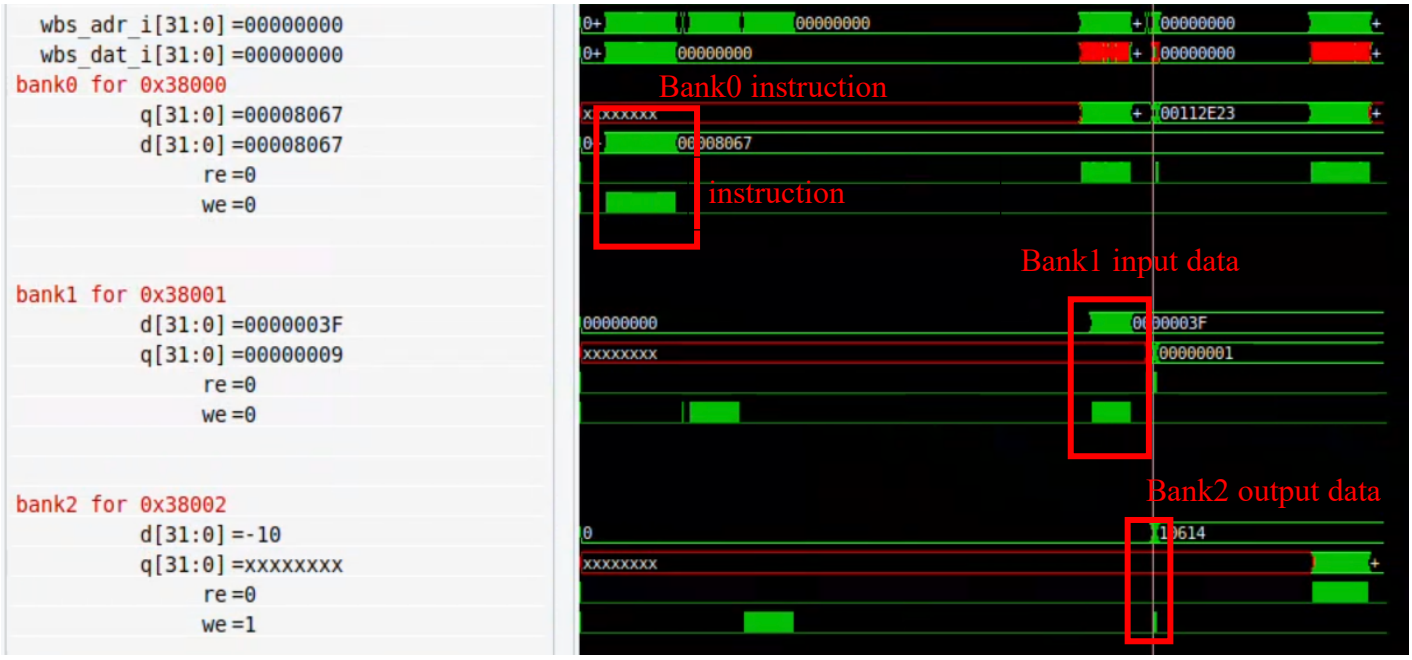
```
mprjram : ORIGIN = 0x38000000, LENGTH = 0x00001000
indata  : ORIGIN = 0x38001000, LENGTH = 0x00001000
outdata : ORIGIN = 0x38002000, LENGTH = 0x00001000
```

```
assign Mapped_RA = {user_addr[22:14],user_addr[11:8]};
assign Mapped_BA = {user_addr[13:12]};
assign Mapped_CA = {user_addr[7:0]};
assign addr = {Mapped_RA, Mapped_BA, Mapped_CA};
```

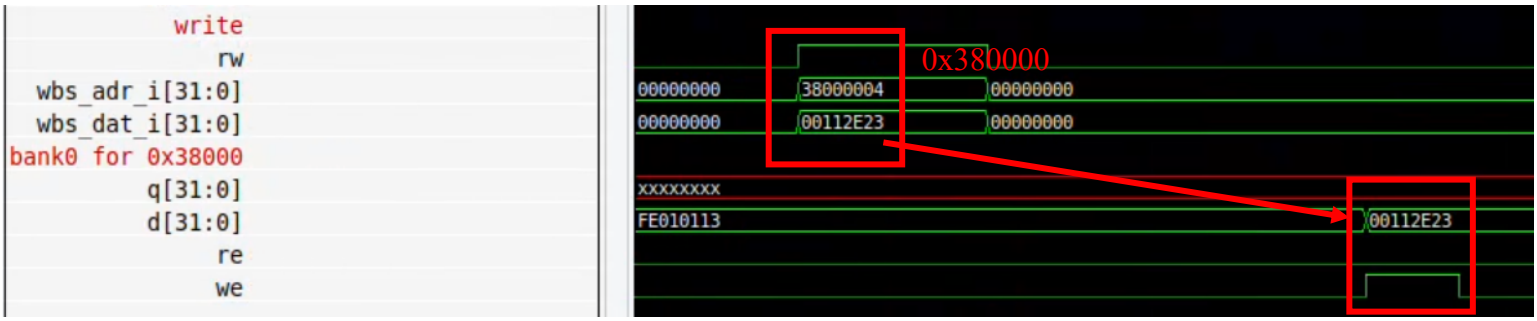
Before performing the prefetch of SDRAM, it is necessary to set the address of the section first. This step ensures that data can be distributed to different banks in the SDRAM. In this context, we have used 0x38000 for storing instructions, 0x38001 for input data, and 0x38002 for output data.

The key difference here lies in the user_addr[13:12], leading us to remap our address. This remapping allows for the data to be routed to the corresponding bank, achieving the effect of bank interleaving. The following figure illustrates the waveform diagram of our results.

Bank interleave



Bank0 for 0x380000 (instruction)



Bank1 for 0x380010 (input data)

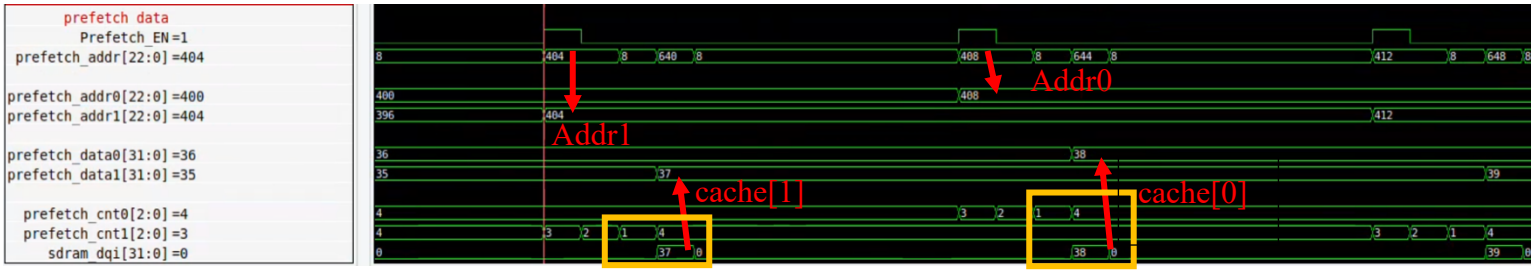
Bank2 for 0x380020 (output data)

SDRAM Prefetch

The new address is redefined as user_addr + 22'd8, termed as prefetch_addr, for the prefetch operation. This is done because when the CPU transmits an address labeled as addr, the next address it sends is generally addr + 4. Therefore, we use a cache buffer of size 2 to prefetch the data at addr + 8, storing the data from addr + 4 in cache[1] and the data from addr + 8 in cache[0].

```
//Cache
assign plus8_addr = user_addr + 22'b1000;
assign prefetch_addr = {plus8_addr[22:14], plus8_addr[11:8], plus8_addr[13:12], plus8_addr[7:0]};
```

The figure above illustrates the process for the required new address for prefetch, which is user_addr + 22'd8, defined as prefetch_addr, and then remapped.



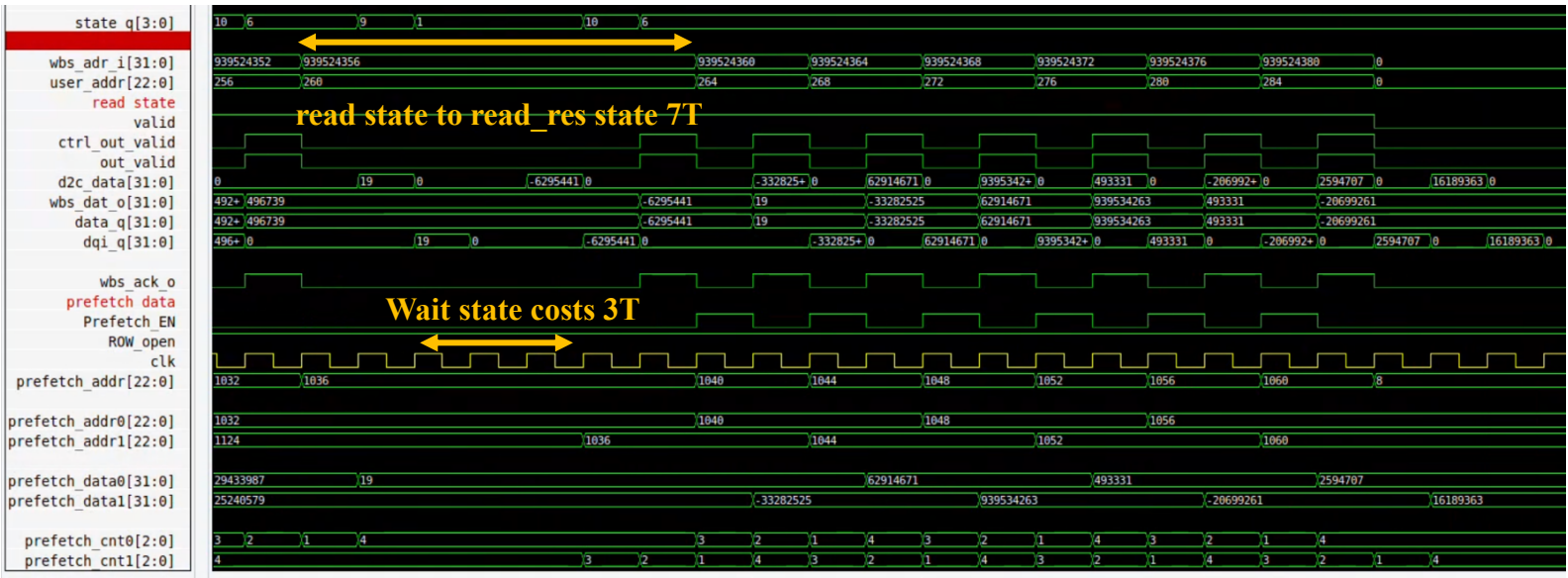
We utilize addr[2] to determine which prefetch buffer to store the data in. As depicted in the diagram below, for prefetch_addr=404, since prefetch_addr[2]=1, the data is stored in prefetch_addr1, or cache_addr[1]. Similarly, for another set, prefetch_addr=408 with prefetch_addr[2]=0, the data is stored in prefetch_addr0, i.e., cache_addr[0].

Additionally, a prefetch_en is set; when addr matches cache_addr[addr[2]], prefetch_en is asserted to 1 to initiate prefetching. It's important to note that prefetch_en also signifies a cache hit.

Furthermore, a cache_counter is configured to count down from 4 to 1. Upon reaching 1, the subsequent clock cycle will store the value from sdram_dqi into the corresponding cache, as shown in the diagram above.

addr	addr+4	addr+8	addr+12	addr+16	addr+20	addr+24	addr+28

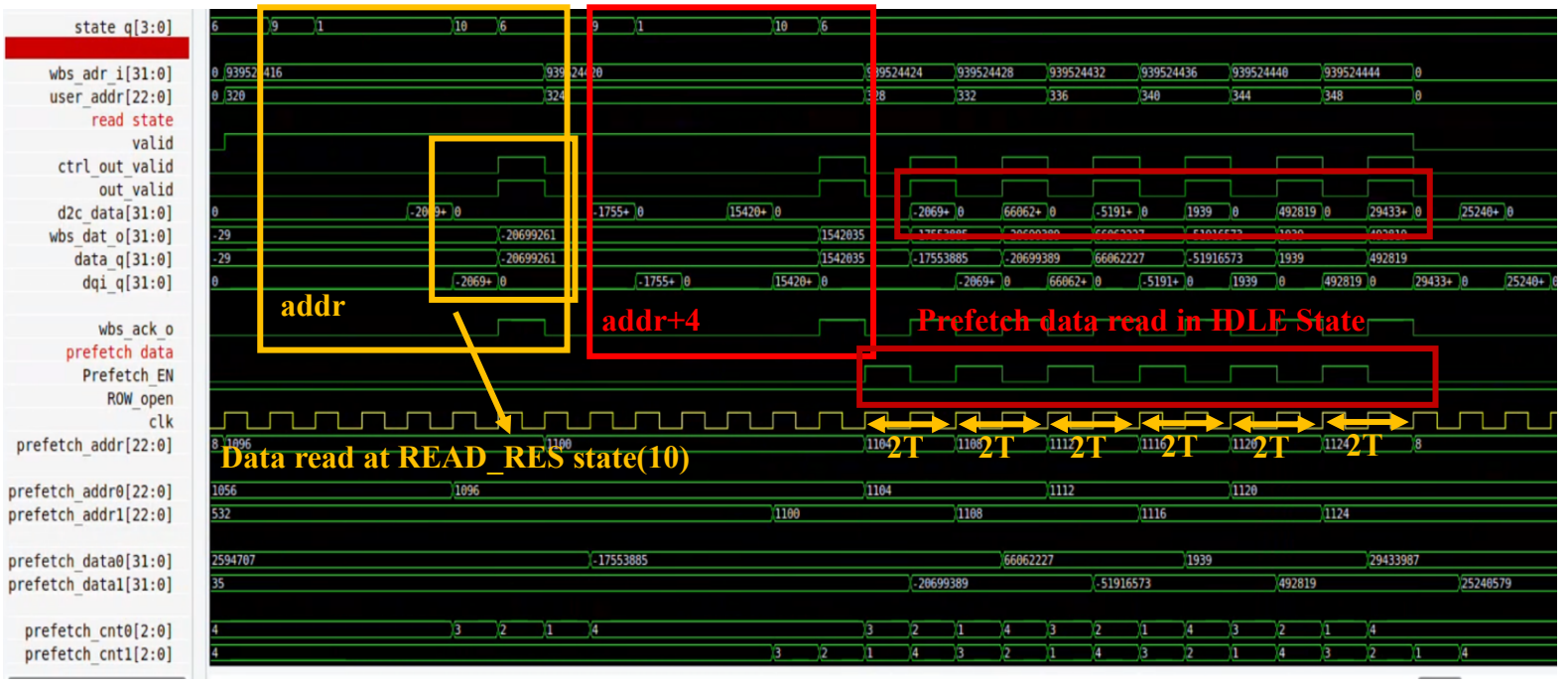
As depicted above, the blue blocks represent storage for addresses in cache[0], while the yellow blocks denote storage for addresses in cache[1].



the read_res state to retrieve the data. The time overhead associated with this process is deemed impractical.

To address this issue, a strategic prefetching mechanism has been introduced during IDLE state. Illustrated in the prefetching process, two consecutive data reads are performed, and their respective addresses undergo addr + 8 prefetch. The prefetched data is then intelligently stored in cache[0] and cache[1]. For instance, after reading data at addr+0, a prefetch operation stores data at addr+8, addr+16, and addr+24 in cache[0]. Subsequently, a prefetch operation after reading addr+4 stores data at addr+12, addr+20, and addr+28 in cache[1]. The prefetch_addr is utilized to determine which cache contains the matching address, facilitating the output of data from the respective cache.

Upon successfully prefetching six data points into the prefetch_buffer when prefetch_en is set to 1, these data are efficiently transferred to data_q, which interfaces with wb_data_o. This optimization enables the reduction of the overall data transfer time to an impressive 2T, significantly enhancing the efficiency of data transmission.



3. Firmware design

The firmware code is designed to start the FIR execution. The following table shows the MMIO address

MMIO

0x38000000	instruction
0x38001000	FIR input
0x38002000	FIR output
0x30000000	FIR ap_start, ap_done, ap_idle
0x30000040~7F	FIR taps
0x30000080	DMA, Input data ram address
0x30000084	DMA, input data length
0x30000088	DMA, output data ram address

Table, memory map io

The CPU needs to send the address and length of FIR's input data. It also needs to send the address of output data so that the DMA knows where to save the output from FIR.

Additionally, we modified the header file and linker file to specify the system RAM address to store the fir input and output data in the specified address.

Linker:

```
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
    indata : ORIGIN = 0x38001000, LENGTH = 0x00001000
    outdata : ORIGIN = 0x38002000, LENGTH = 0x00001000
    hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
    csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
```

Header file

```
#define RCV_SDRAM_INDATA __attribute__((section(".indata")))
#define RCV_SDRAM_OUTDATA __attribute__((section(".outdata")))
#include <stdint.h>

// linker memory
#define INDATA_ADR  0x38001000
#define OUTDATA_ADR  0x38002000
//wishbone operation
#define send_wb(target,data) (*(volatile uint32_t*)(target)) = data // send wishbone sign
#define read_wb(target)  (*(volatile uint32_t*)(target))// wishbone read
//data memory

//*********************************************************/
//fir accelerater                                         /
//*********************************************************/
#define N 11
#define NI 64
volatile RCV_SDRAM_INDATA int inputsignal[NI];
volatile RCV_SDRAM_OUTDATA int outputsignal[NI];

int output[NI];
int taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0};
// fir mmio
#define WB_FIR_BLK_LVL      0x30000000
#define WB_FIR_TAP_START    0x30000040
#define WB_DMA_START_ADDR   0x30000080
#define WB_DMA_LENGTH_ADDR  0x30000084
#define WB_DMA_SAVE_ADDR    0x30000088
#define data_len N
```

__attibute__ built in type to specify the memory address of variable

define input and output array

Firmware code that execute FIR :

```
void __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){
        //fir accelerater================================

        //step 1 send parameter to accelerater (taps)
        // 送tap到AXILITE
        for(int i=0;i<N;i++){
        send_wb(WB_FIR_TAP_START + i*4, taps[i]);
    }
        //step 2 allocate input and output buffer
        //把INPUT存到SDRAM
        for(int i=0;i<NI;i++){
        inputsignal[i] = i;
    }
        //啟動dma
        start_dma((uint32_t) INDATA_ADR, NI,OUTDATA_ADR);

        //step 3 start the fir accelerater
        enum BLKLVL blklvl;
        while( read_wb(WB_FIR_BLK_LVL) & (1<<ap_idle ) != 1<<ap_idle);
        send_wb(WB_FIR_BLK_LVL,  (1 << ap_start) );
```

send taps to FIR

send input data to SDRAM

send the start address, data length and save address to DMA

send ap_start to FIR

After completing the FIR execution and saving all output data, the DMA generates an interrupt signal to prompt CPU to access the output data stored in SDRAM. In the interrupt service routine, isr_dma() function is called to access the output data.

```
void isr(void)
{

#ifndef USER_PROJ_IRQ0_EN

    irq_setmask(0);


#else
    uint32_t irqs = irq_pending() & irq_getmask();
    int buf;

    if ( irqs & (1 << USER_IRQ_0_INTERRUPT)) {
        user_irq_0_ev_pending_write(1); //Clear Interrupt Pending Event
      isr_dma();
      /*
```

```
void __attribute__ ((section("mprjram")))isr_dma(){
    intr_dma = 1;
    for(int i=0;i<NI; i++)
        output[i] = outputsignal[i];
}
```

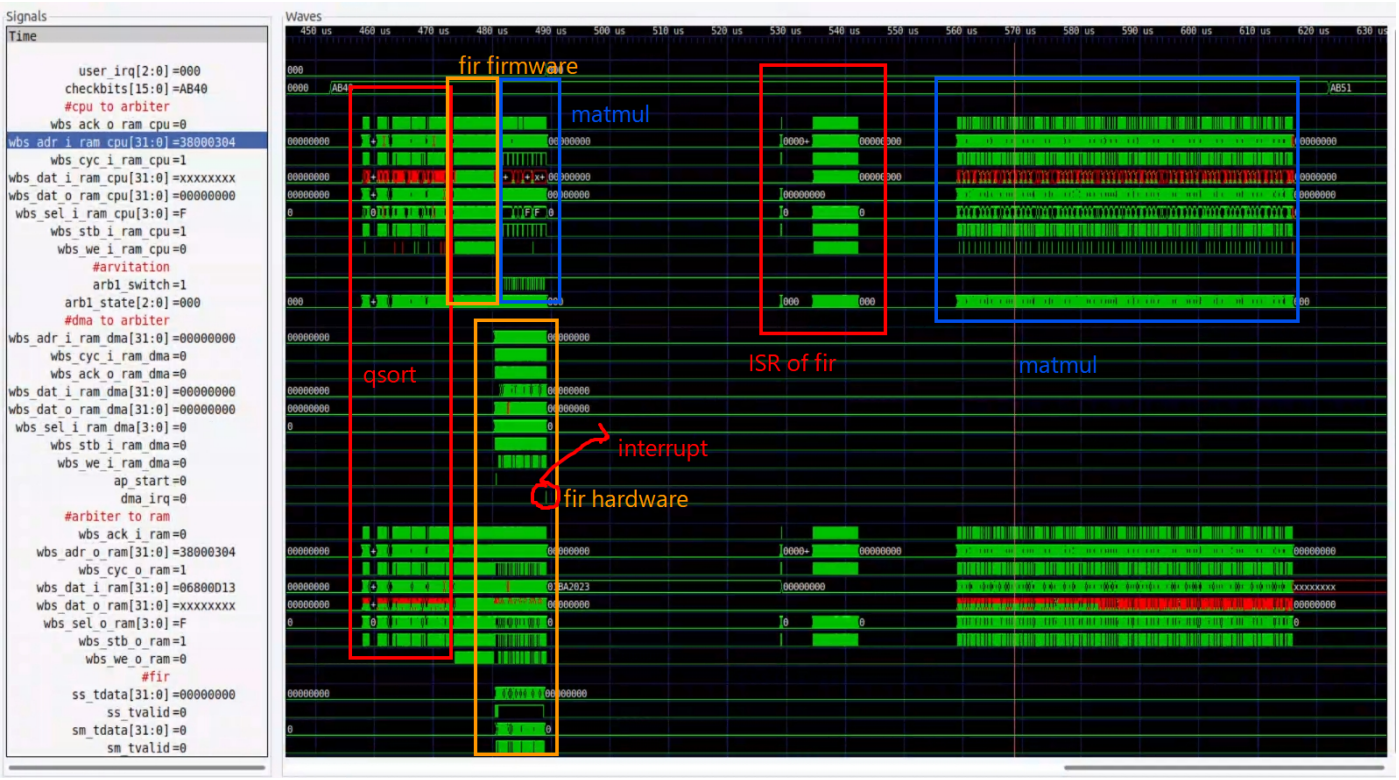
4. Testing

Finally, we tested our system by running FIR as well as two algorithms.

```
int __attribute__ ( ( section ( ".mprjram" ) ) ) main_func(){
    qsort();
    fir();
    matmul();
    //while(!intr_dma);


    return 0;
}
```

The following waveform is the result of the testing.



In the system, the CPU executes qsort first. After completing it, the CPU initializes the FIR and DMA, and starts the FIR accelerator. Then, the CPU begins executing matmul. It is evident from the waveform that the execution of the FIR accelerator and matmul runs in parallel. However, the FIR execution takes less time than matmul, the interrupt signal generated by DMA is asserted before the completion of it, leading to the interruption of matmul. The CPU execute the ISR of fir to receive the output data stored in SDRAM. Then, it continues to execute the remaining portion of matmul.

Work Assignment

	張耀明	陳佳詳	蕭翔
UART Acceleration	**1.Hardware:** Rx_fifo, Tx_fifo **2.Firmware** Firmware		
FIR Acceleration	**1.Hardware:** Arbiter, WB_Decoder, **2.Firmware:** Firmware Linker script	**Hardware:** DMA, X_FIFO, Y_FIFO, DMA_axistream_interface	**Hardware:** SDRAM SDRAM_controller

GitHub link:

We divided our work into two git branch: develop(FIR acceleration) and dev-uart(UART acceleration)

1. FIR Acceleration: https://github.com/s095339/SOC_wlos_opt/tree/develop

2. UART Acceleration: https://github.com/s095339/SOC_wlos_opt/tree/dev-uart