SOC Lab 4 Report

312510145 劉峻瑋
312510147 陳建嘉
312510155 張嘉恩

4-1 Question

1. **Explanation of your firmware code**

   counter_la_fit.c:

   MMIO config:

```
//設定mmio的pin角由cpu host端或是user program端驅動
//這次lab將MMIO pin 31-16及pin 6設為由cpu host驅動，
//剩下的設為由user program驅動。
reg_mprj_io_31 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_30 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_29 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_28 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_27 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_26 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_25 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_24 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_23 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_22 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_21 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_20 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_19 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_18 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_17 = GPIO_MODE_MGMT_STD_OUTPUT;
reg_mprj_io_16 = GPIO_MODE_MGMT_STD_OUTPUT;

reg_mprj_io_15 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_14 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_13 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_12 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_11 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_10 = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_9  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_8  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_7  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_5  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_4  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_3  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_2  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_1  = GPIO_MODE_USER_STD_OUTPUT;
reg_mprj_io_0  = GPIO_MODE_USER_STD_OUTPUT;

reg_mprj_io_6  = GPIO_MODE_MGMT_STD_OUTPUT;
```

   等待MMIO設定完成:

```
//將reg_mprj_xfer拉成1開始設定config，等待reg_mprj_xfer被拉回0表示設定完成
// Now, apply the configuration
reg_mprj_xfer = 1;
while (reg_mprj_xfer == 1);
```

   透過IO傳遞 0xAB40讓通知testbench開始計時執行fir的時間:

```
// Flag start of the test
reg_mprj_datal = 0xAB400000;
```

CPU執行編譯好的fir assembly code:

```
//SW執行fir並將CPU執行結果經由MMIO傳給testbench對答案
int* tmp = fir();
reg_mprj_datal = *tmp << 16;
reg_mprj_datal = *(tmp+1) << 16;
reg_mprj_datal = *(tmp+2) << 16;
reg_mprj_datal = *(tmp+3) << 16;
reg_mprj_datal = *(tmp+4) << 16;
reg_mprj_datal = *(tmp+5) << 16;
reg_mprj_datal = *(tmp+6) << 16;
reg_mprj_datal = *(tmp+7) << 16;
reg_mprj_datal = *(tmp+8) << 16;
reg_mprj_datal = *(tmp+9) << 16;

reg mprj datal = *(tmp+10) << 16;
```

透過IO傳遞 0xAB51讓通知testbench結束計時執行fir的時間:

```
reg_mprj_datal = 0xAB510000;
```

fir.h:

define 參數的值

```
#ifndef __FIR_H__
#define __FIR_H__

#define N 64
// N stands for the number of data length
// int taps[11] = {0,-10,-9,23,56,63,56,23,-9,-10,0};
int taps_0 = 0;
int taps_1 = -10;
int taps_2 = -9;
int taps_3 = 23;
int taps_4 = 56;
int taps_5 = 63;
int taps_6 = 56;
int taps_7 = 23;
int taps_8 = -9;
int taps_9 = -10;
int taps_10 = 0;

int inputsignal[N];
int outputsignal[N];

#endif
```

fir.c:

lab4-1 的memory address起點設在0*38000000

```
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 = 0x00400000
    hk : ORIGIN = 0x26000000, LENGTH = 0x00100000
    csr : ORIGIN = 0xf0000000, LENGTH = 0x00010000
}
```

fir software code並指定將compile完的assembly code存到mprjram define的位置中

```c
//reset memory中inputbuffer及outputsignal的資料
void __attribute__ ( ( section ( ".mprjram" ) ) ) initfir() {
    //initial your fir
    for(int i=0; i<N; i=i+1){
        inputbuffer[i] = 0;
        outputsignal[i] = 0;
    }
}
//讀取memory中inputbuffer及taps的資料，並將運算結果寫入memory中outputsignal的位置，還會將結果送入MMIO的output pin
int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){
    // initfir();
    //write down your fir
    for(int idx = 0; idx < N ; idx ++){
        for(int i=N-1; i > 0;i--){ // shift the data to be calculate in fir process
            inputbuffer[i] = inputbuffer[i-1];
        }
        inputbuffer[0] = inputsignal[idx];
        for(int cnt = 0; cnt < N; cnt ++){ // fir mult
            outputsignal[idx] += inputbuffer[cnt] * taps[cnt];
        }
    }

    return outputsignal;
}
```

a. **How does it execute a multiplication in assembly code**

(file:counter_la_fir.out)

下面這段就是fir.c中乘法的assembly code

```
Disassembly of section .mprjram:

38000000 <__mulsi3>:
38000000:	00050613          	mv	a2,a0
38000004:	00000513          	li	a0,0
38000008:	0015f693          	andi	a3,a1,1
3800000c:	00068463          	beqz	a3,38000014 <__mulsi3+0x14>
38000010:	00c50533          	add	a0,a0,a2
38000014:	0015d593          	srli	a1,a1,0x1
38000018:	00161613          	slli	a2,a2,0x1
3800001c:	fe0596e3          	bnez	a1,38000008 <__mulsi3+0x8>
38000020:	00008067          	ret
```

mv:將a0的值移到a2
li:將a0歸零
andi:a3=a1[0]
beqz:if(a3==0) jump to 38000014
add:a0=a0+a2
srli:a1=a1>>1
slli:a2=a2<<1
bnez:if(a1!=0) jump to 38000008
ret:結束mulsi3

根據compile出來的assembly code可以推估出我們cpu的RISCV架構不支援mul, mulsi3會依序從LSB對每1bit作加法和左移右移來完成2個reg之間的乘法。

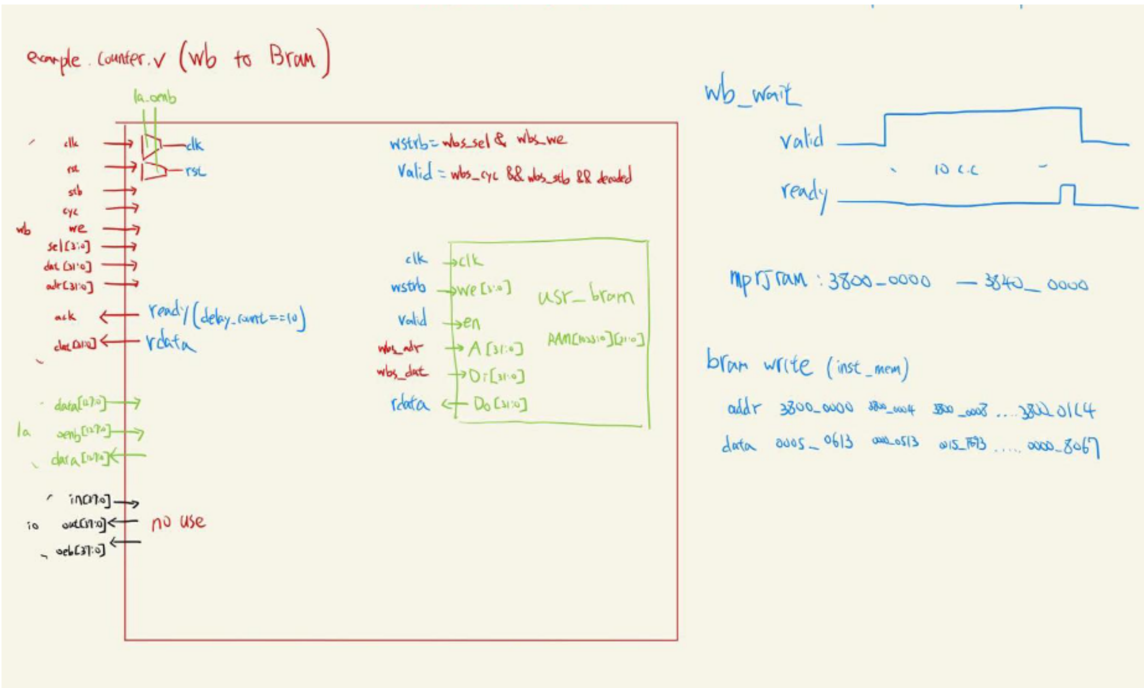
b. What address allocate for user project and how many space is required to allocate to firmware code

```
38000154:   00078593        mv   a1,a5
38000158:   00068513        mv   a0,a3
3800015c:   ea5ff0ef        jal  ra,38000000 <__mulsi3>
38000160:   00050793        mv   a5,a0
38000164:   00f48733        add  a4,s1,a5
38000168:   08800693        li   a3,136
3800016c:   fec42783        lw   a5,-20(s0)
38000170:   00279793        slli     a5,a5,0x2
38000174:   00f687b3        add  a5,a3,a5
38000178:   00e7a023        sw   a4,0(a5)
3800017c:   fe442783        lw   a5,-28(s0)
38000180:   00178793        addi     a5,a5,1
38000184:   fef42223        sw   a5,-28(s0)
38000188:   fe442703        lw   a4,-28(s0)
3800018c:   00a00793        li   a5,10
38000190:   f8e7d4e3        bge  a5,a4,38000118 <fir+0x8c>
38000194:   fec42783        lw   a5,-20(s0)
38000198:   00178793        addi     a5,a5,1
3800019c:   fef42623        sw   a5,-20(s0)
380001a0:   fec42703        lw   a4,-20(s0)
380001a4:   00a00793        li   a5,10
380001a8:   f0e7d0e3        bge  a5,a4,380000a8 <fir+0x1c>
380001ac:   08800793        li   a5,136
380001b0:   00078513        mv   a0,a5
380001b4:   01c12083        lw   ra,28(sp)
380001b8:   01812403        lw   s0,24(sp)
380001bc:   01412483        lw   s1,20(sp)
380001c0:   02010113        addi     sp,sp,32
380001c4:   00008067        ret
```

spec required : less than 4MB
our firmware code required : end address: 380001c4 -> total 456 byte

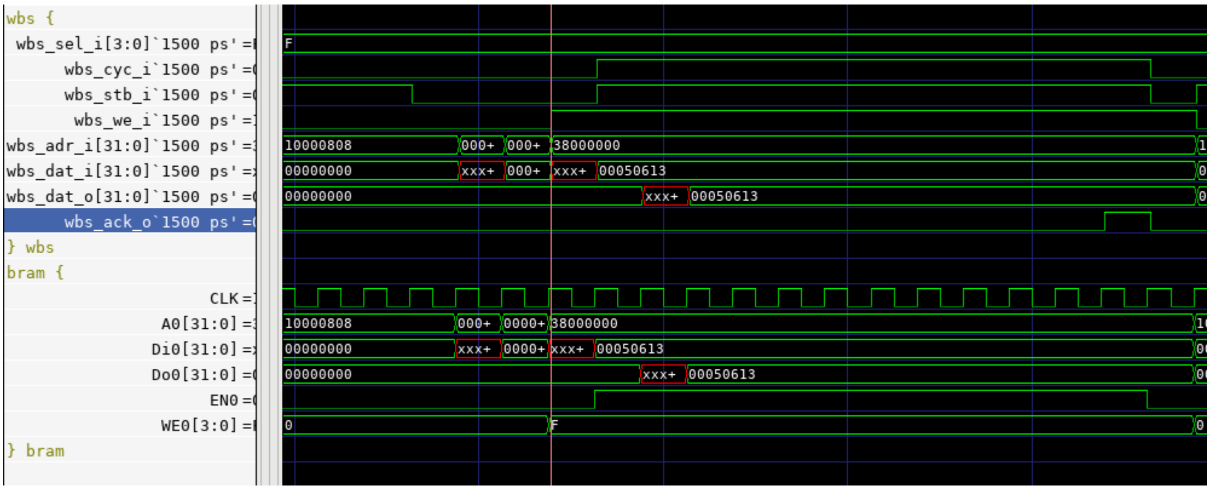
2. Interface between BRAM and wishbone

i) wb_ack在wb_cyc和wb_stb拉上來後要等10個cycle才拉起，以保證在不同design constraint下interface不會出錯
ii) mprjram的range從3800_0000給到3840_0000，但fir assembly code只會使用3800_0000到3800_01C4的adress

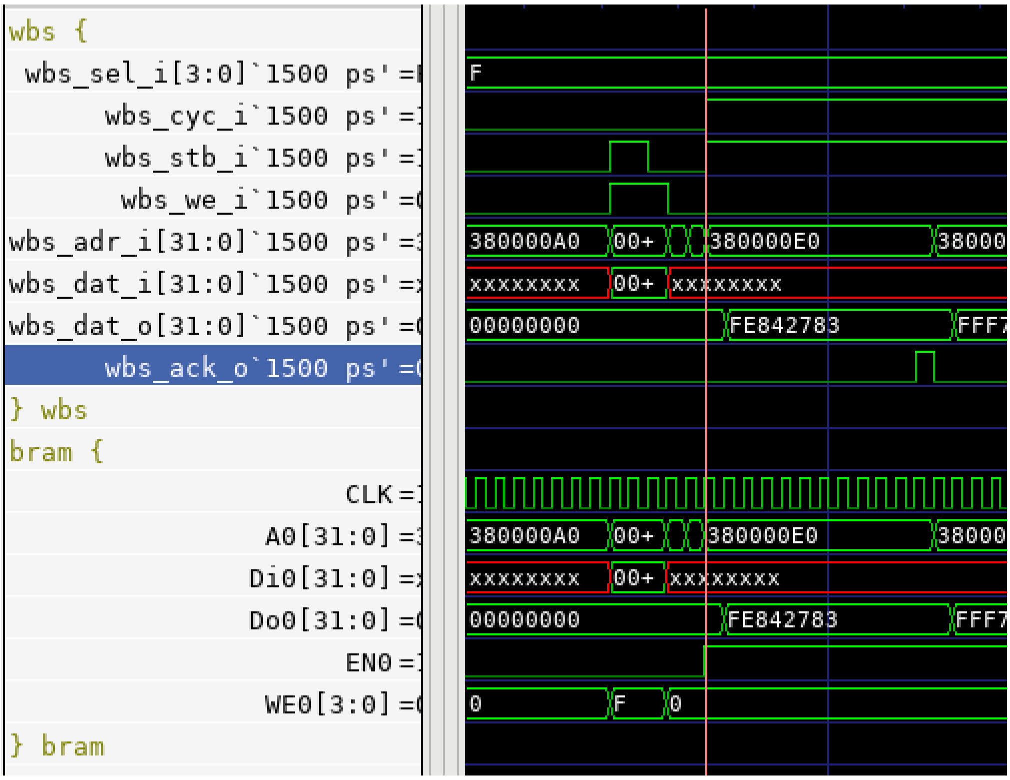


a. **Waveform from xsim**

write fir assembly code to bram:

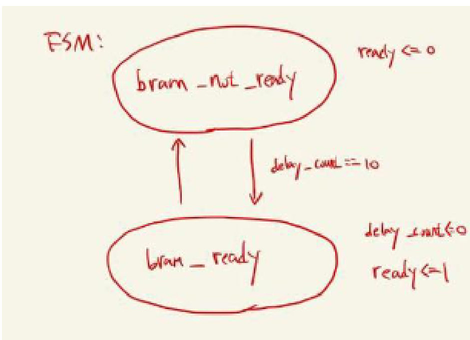


read fir assembly code to bram:



b. **FSM**

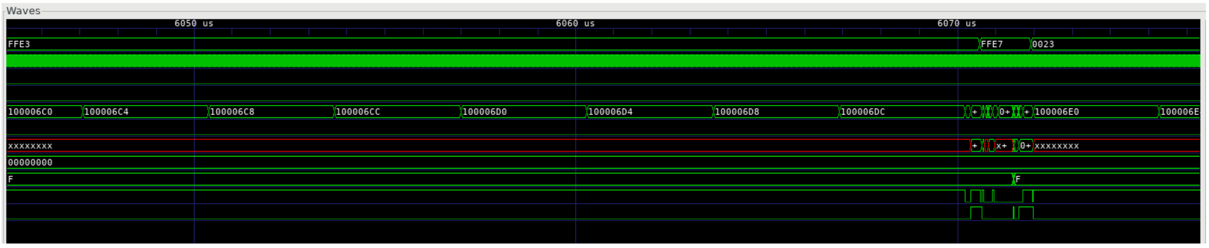
FSM其實就是看delay_counter是否數到10，將wbs_ack拉起。



3. **Synthesis report**

4-1的exmem_bram的合成放在下方4-2的synthesis report中

4. **Other discoveries**

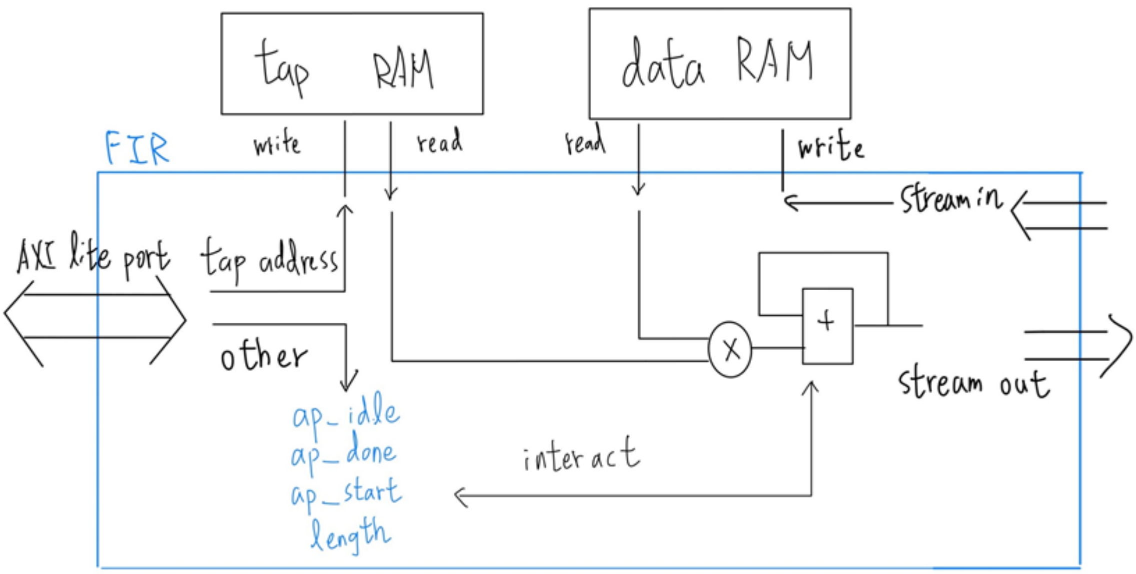


有注意到在輸出fir.c的output到reg_mprj_datal的時候，有一些特別短的訊號，這點挺有趣的，推測是跟產生assembly code相關，在加大input的數量後好像會以大概的周期出現。

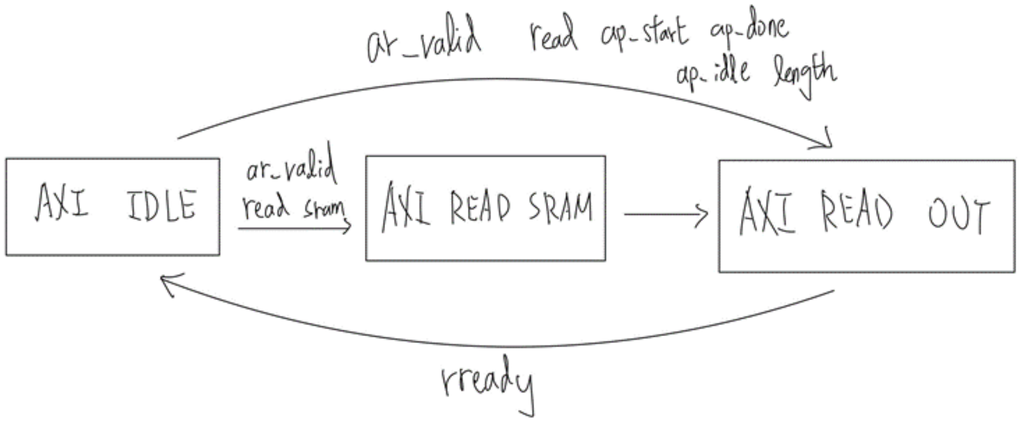
4-2 Question

1. **Design block diagram – datapath, control-path**

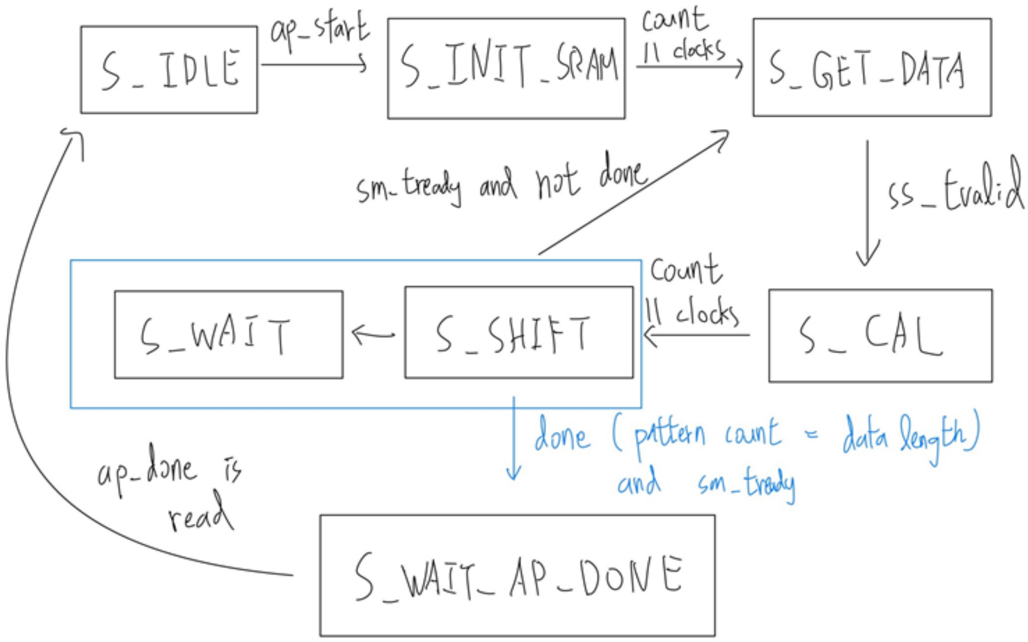
Data Path:



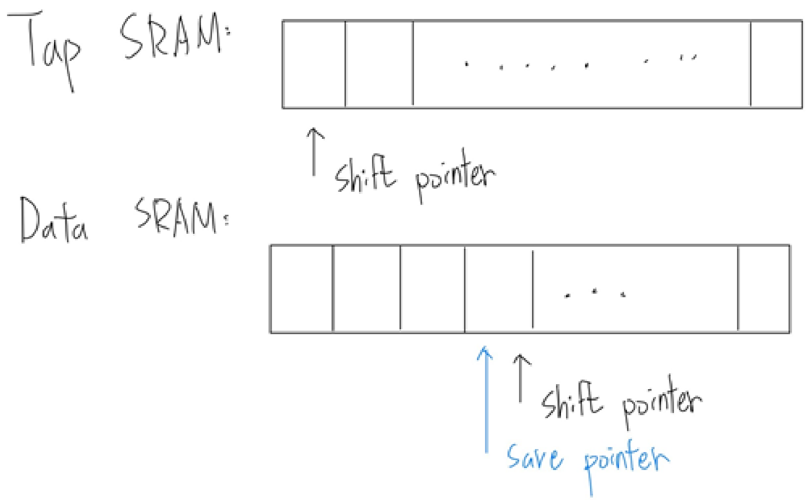
AXI lite FSM:



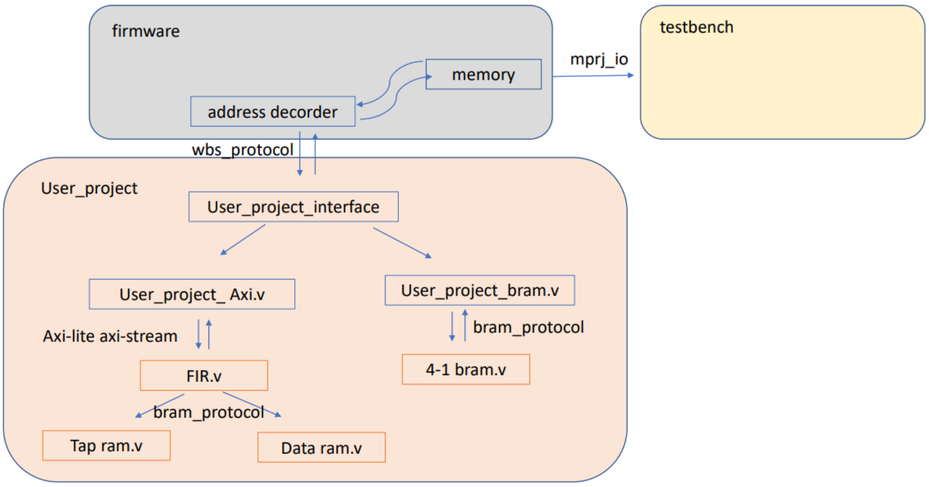
FIR FSM:



Shift SRAM control:

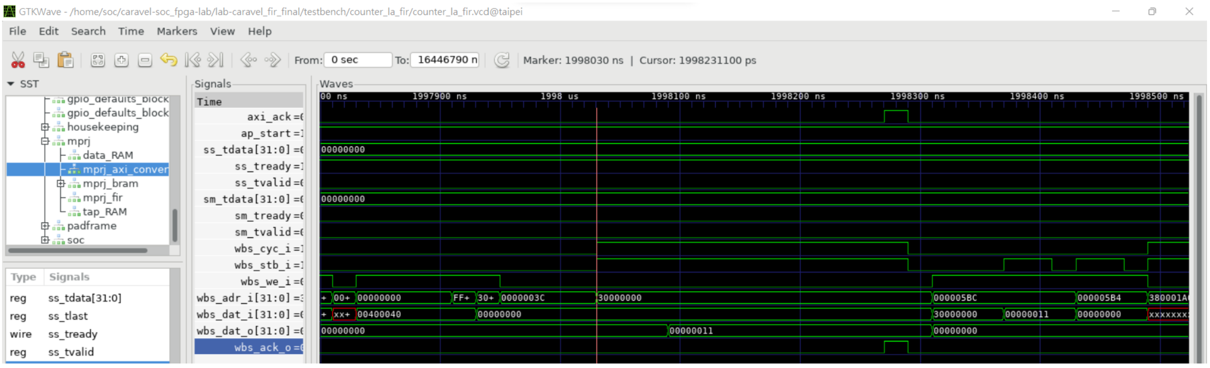


2. **The interface protocol between firmware, user project and testbench**



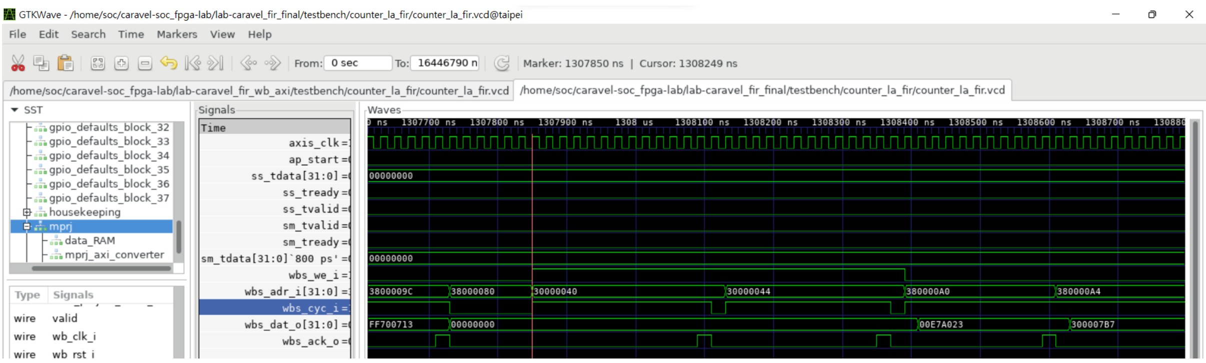
3. **Waveform and analysis of the hardware/software behavior.**

Firmware reading parameter:



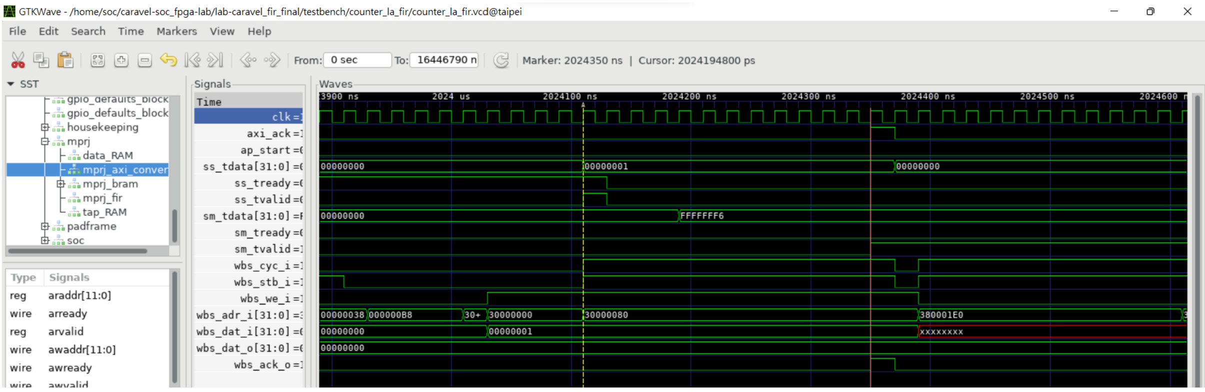
這邊可以看到fimware發送read指令給wishbone，並且在過了10個cycle delay後，收到wishbone回傳的ack以及資料，這邊read的位址是3000_0000，去讀取ap參數及幾個flag，可以看到data_out[4]為1，代表wishbone已經準備好接受X[n]輸入了

Firmware programming parameter:



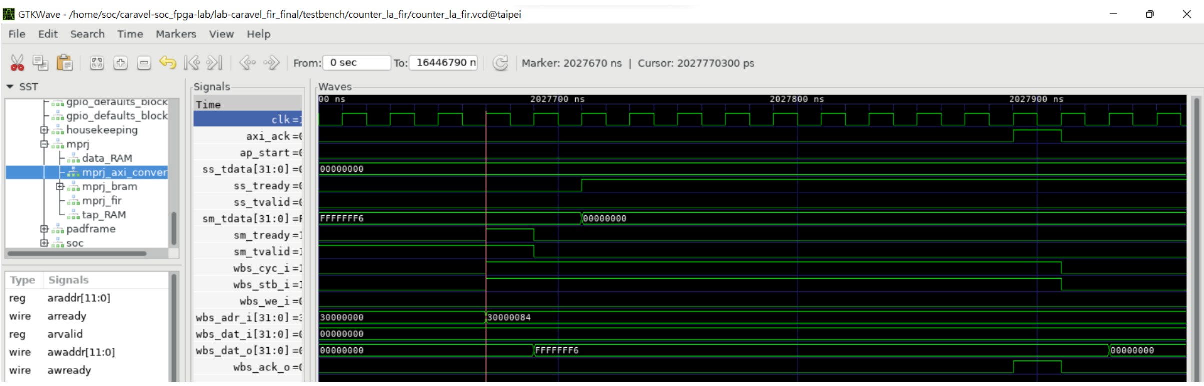
這邊可以看到firmware發送write指令給wishbone，寫入位址為3800_0040，代表正在program tap parameter

Firmware sending X[n]:



這邊可以看到firmware發送write指令給wishbone，位址為3000_0080，代表正在傳送X[n]

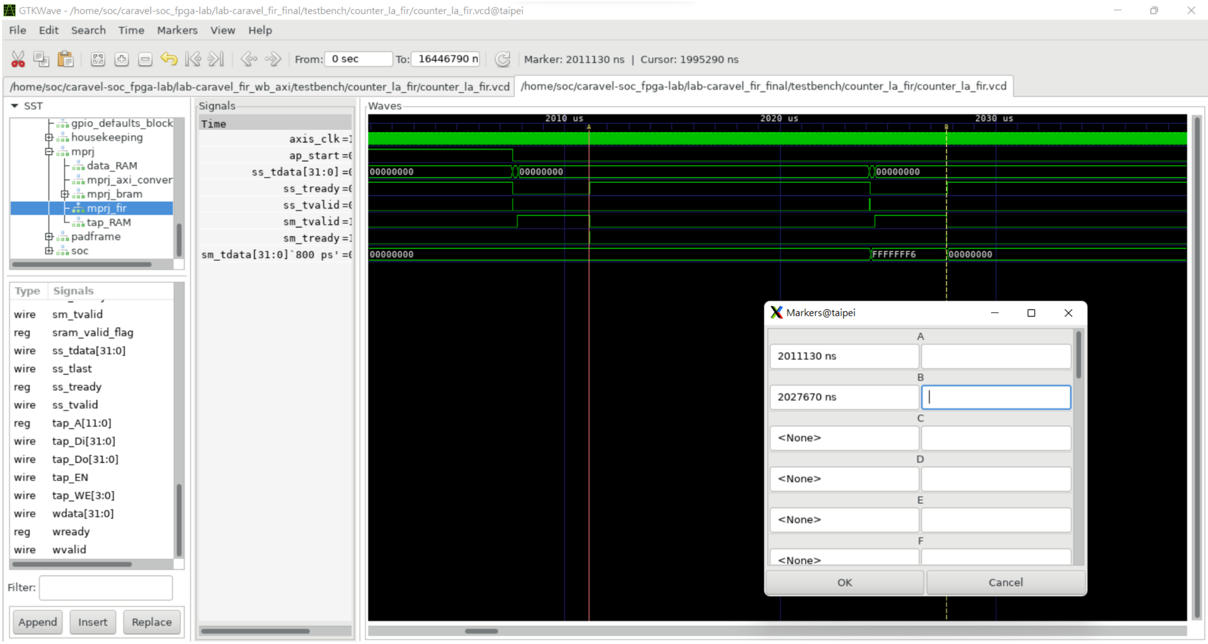
Firmware receiving Y[n]:



這邊可以看到firmware發送read指令給wishbone，位址為3000_0084，代表正在接生Y[n]

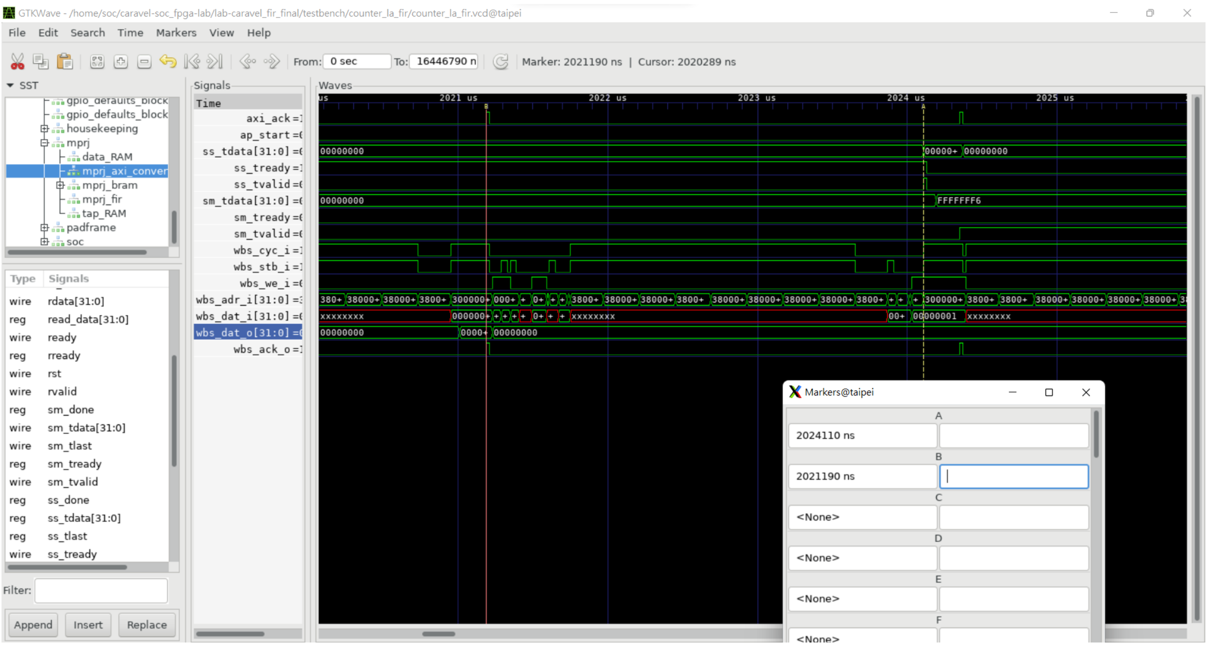
4. **What is the FIR engine theoretical throughput, i.e. data rate? Actually measured throughput?**

FIR engine theoretical throughput: 32 bits / 12 cycle = (32 bits / 120 ns)
= (8 / 15) bit/ns
因為只能用一個乘法器，根據lab3的spec，可以發現在testbench的stream interface隨時都可以送資料以及收資料的情況下，最佳的throughput就是每12個cycle算出一筆資料。
Actually measured throughput: (32 bits / 1654 cycle) = (32 bits / 16540 ns)
= (8 / 4135) bit/ns
然而在lab4-2中，資料會需要從firmware到wishbone經過decode再到user project，送資料回去也需按照此路徑傳輸，收完一筆資料才能傳下一筆資料，導致throughput會下降許多



5. **What is latency for firmware to feed data?**

Latency for firmware: 2920 ns = 292 cycle



我們的算法是計算firmware透過axi lite read讀取address 3000_0000的第四個bit，發現可以傳送資料後，到FIR透過stream interface收到資料的latency

6. **What techniques are used to improve the throughput?**

- 妥善處理不同protocol之間的轉換去減少不必要的latency跟area
- 用pointer的方式處理data bram去降低shift所帶來的cost

7. **Does bram12 give better performance, in what way?**

   就目前的硬體限制而言，tap ram改用bram12應該並不會提升throughput，反倒是area會上升，在只能使用一組乘加器進行運算的前提下我們已經可以用11cycles去完成fir的計算，因此如果要提升throughput我想重點不會是在改變ram的容量而是增加平行度。

   從另外一個設計角度去想，我們若是考慮將dataram改成bram12的話或許可以降低latency進而提高throughput，主要的原因是我們可以將多出來的位置作為input buffer來使用，在計算完第一筆input的同時，可以將等待output的時間用來計算第二筆輸入，進而將等待的時間與計算的時間overlap增加throughput。

8. **Can you suggest other methods to improve the performance?**

   - 多開硬體提升throughput(多開乘法器)
   - AXI-lite的read write channel盡量overlap來減少latency
   - FIR 本身的運算特性很適合用pipeline處理，將data使用shift register處理可以有效提升throughput

9. **Syn report:**

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	403	0	0	53200	0.76
LUT as Logic	339	0	0	53200	0.64
LUT as Memory	64	0	0	17400	0.37
LUT as Distributed RAM	64	0			
LUT as Shift Register	0	0			
Slice Registers	247	0	0	106400	0.23
Register as Flip Flop	245	0	0	106400	0.23
Register as Latch	2	0	0	106400	<0.01
F7 Muxes	10	0	0	26600	0.04
F8 Muxes	0	0	0	13300	0.00

在本次報告中總共使用了3顆bram，分別是lab3中所使用到的Dataram,tapram和lab4-1中所使用到來儲存firmware code的bram，其中在下圖中看到的會是儲存firmware code的bram，另一部分的ram則會出現在distributed ram裡面。

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	1	0	0	140	0.71
RAMB36/FIFO*	1	0	0	140	0.71
RAMB36E1 only	1				
RAMB18	0	0	0	280	0.00

3. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	3	0	0	220	1.36
DSP48E1 only	3				

4. IO and GT Specific

Site Type	Used	Fixed	Prohibited	Available	Util%
Bonded IOB	305	0	0	125	244.00
Bonded IPADs	0	0	0	2	0.00
Bonded IOPADs	0	0	0	130	0.00
PHY_CONTROL	0	0	0	4	0.00
PHASER_REF	0	0	0	4	0.00
OUT_FIFO	0	0	0	16	0.00
IN_FIFO	0	0	0	16	0.00
IDELAYCTRL	0	0	0	4	0.00
IBUFDS	0	0	0	121	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	0	16	0.00
PHASER_IN/PHASER_IN_PHY	0	0	0	16	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	0	200	0.00
ILOGIC	0	0	0	125	0.00
OLOGIC	0	0	0	125	0.00

```
7. Primitives
-------------

+----------+------+---------------------+
| Ref Name | Used | Functional Category |
+----------+------+---------------------+
| FDRE     |  243 |        Flop & Latch |
| OBUFT    |  128 |                  IO |
| OBUF     |  112 |                  IO |
| LUT5     |   95 |                 LUT |
| LUT6     |   74 |                 LUT |
| IBUF     |   65 |                  IO |
| RAMS32   |   64 |  Distributed Memory |
| LUT2     |   60 |                 LUT |
| LUT4     |   50 |                 LUT |
| LUT1     |   45 |                 LUT |
| CARRY4   |   37 |          CarryLogic |
| LUT3     |   32 |                 LUT |
| MUXF7    |   10 |               MuxFx |
| DSP48E1  |    3 |    Block Arithmetic |
| LDCE     |    2 |        Flop & Latch |
| FDSE     |    2 |        Flop & Latch |
| RAMB36E1 |    1 |        Block Memory |
| BUFG     |    1 |               Clock |
+----------+------+---------------------+
```