LAB 4-1 report

312510154 符顥瀚 312580044 游承緯

Explanation of your firmware code

<fir.h>

```
#ifndef __FIR_H__
#define __FIR_H__

#define N 11

int taps[N] = {0,-10,-9,23,56,63,56,23,-9,-10,0};
int inputbuffer[N];
int inputsignal[N] = {1,2,3,4,5,6,7,8,9,10,11};
int outputsignal[N];
#endif
```

把 fir.c 需要用到的變數宣告在 fir.h，包括 taps coefficient、fir 的 input 訊號，最後就是 fir.c 裡的 fir 系統的 X[n]與 Y[n]，分別為 inputbuffer 及 outputsignal。

<fir.c>

```
#include "fir.h"

void __attribute__ ( ( section ( ".mprjram" ) ) ) initfir() {
    //initial your fir
    for(int i = 0 ; i < N ; i++){
        inputbuffer[i] = 0;
    }
}

int* __attribute__ ( ( section ( ".mprjram" ) ) ) fir(){
    initfir();
    //write down your fir
    for(int i = 0 ; i < N ; i++){

        for(int j = 0 ; j < N-1 ; j++)
            inputbuffer[j] = inputbuffer[j+1];
        inputbuffer[N-1] = inputsignal[i];


        int mult,acc=0;

        for(int i = 0 ; i < N ; i++){
            mult = taps[i] * inputbuffer[N-i-1];
            acc += mult;
        }

        outputsignal[i] = acc;
    }
    return outputsignal;
}
```

Initfir()裡做的事就是歸零 X[n]的所有值，section (".mprjram")是讓 initfir()這個 function 被放在 mprjram 的 section 中。fir()就是我們主要做 fir 的 function，他一樣透過 section (".mprjram")讓 function 被放在 mprjram 的 section 中，最外圍的迴圈就是我們的 fir 總共要做 N 次(因為 inputsignal 有 n 個)，再來就是將我們的 data 往前平移，並把新的 inputsignal 放在最後一個位置，接著就是做 N 次的乘加(因為 taps 跟 data 的 inputbuffer 的長度為 N)，最後就可以將乘加的結果丟進 outputsignal[i]，等到 N 次的 fir 做完之後，將 outputsignal 這個 array 的 address return 回去。

How does it execute a multiplication in assembly code

In <counter_la_fir.out>

```
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
```

<__mulsi3>這個就是乘法的 function，主要的計算就是透過 srli、slli 平移 a1 及 a2，利用累加完成乘法，最後會得出 a0 及 a1 的乘積。

In <counter_la_fir.elf-fir.s>

```
.L9:
    .loc 1 23 15 discriminator 3
    lui a5,%hi(taps)
    addi    a4,a5,%lo(taps)
    lw  a5,-32(s0)
    slli    a5,a5,2
    add a5,a4,a5
    lw  a3,0(a5)
    .loc 1 23 38 discriminator 3
    li  a4,10
    lw  a5,-32(s0)
    sub a5,a4,a5
    .loc 1 23 32 discriminator 3
    lui a4,%hi(inputbuffer)
    addi    a4,a4,%lo(inputbuffer)
    slli    a5,a5,2
    add a5,a4,a5
    lw  a5,0(a5)
    .loc 1 23 9 discriminator 3
    mv  a1,a5
    mv  a0,a3
    call    __mulsi3
    mv  a5,a0
    sw  a5,-36(s0)
    .loc 1 24 8 discriminator 3
    lw  a4,-28(s0)
    lw  a5,-36(s0)
    add a5,a4,a5
    sw  a5,-28(s0)
    .loc 1 22 29 discriminator 3
    lw  a5,-32(s0)
    addi    a5,a5,1
    sw  a5,-32(s0)
```

這段主要是在做讀 tap 與 inputbuffer 的值，並做乘加，可以看到 a5 拿到 taps 的地址之後，會存在 a3(lw a3,0(a5))，再讓 a4 去拿 inputbuffer 的 base address，再透過平移把正確的 inputbuffer 位置存在 a5，再透過 mv a1,a5 及 mv a0,a3，把 a3 a5 丟到 a0 及 a1，之後透過 call __mulsi3，把乘法計算完成結果存在 a0，最後再做累加。

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

```
mprjram : ORIGIN = 0x38000000, LENGTH = 0x00400000
```

我們可以看到 mprjram allocate 的 address 是從 0x38000000~0x38400000，我們的 firmware code 就存在這段 address 裡，我們要去<counter_la_fir.out>找 Disassembly of section .mprjram:

```
38000000 <__mulsi3>:
38000000:   00050613            mv  a2,a0
38000004:   00000513            li  a0,0
```

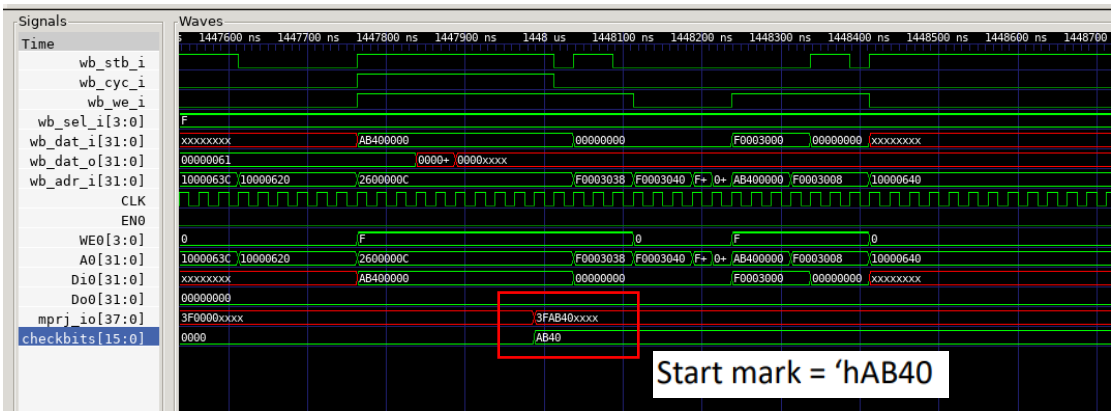
.

.

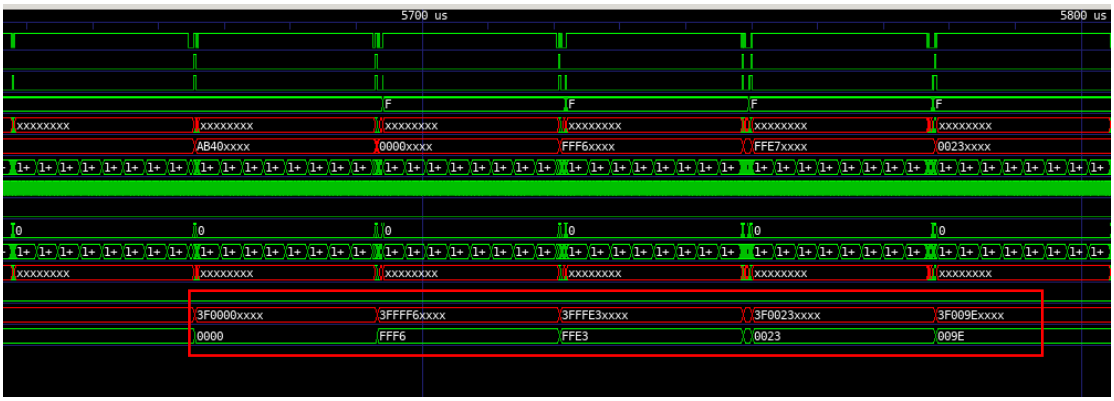
```
380001b0:   02812403            lw  s0,40(sp)
380001b4:   03010113            addi    sp,sp,48
380001b8:   00008067            ret
```

最後就可以計算出 firmware code 會被 allocate 在 0x38000000~0x380001b8 中，共 440 個 byte。

Interface between BRAM and wishbone

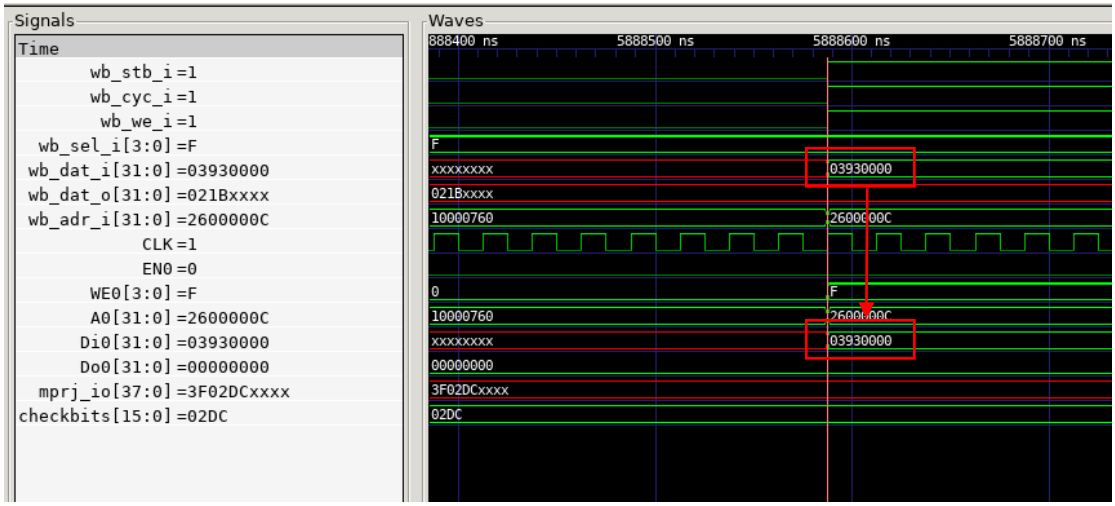


當 checkbits == 'hAB40 時，FIR 開始運作，CPU 開始執行 fir 的 firmware code。

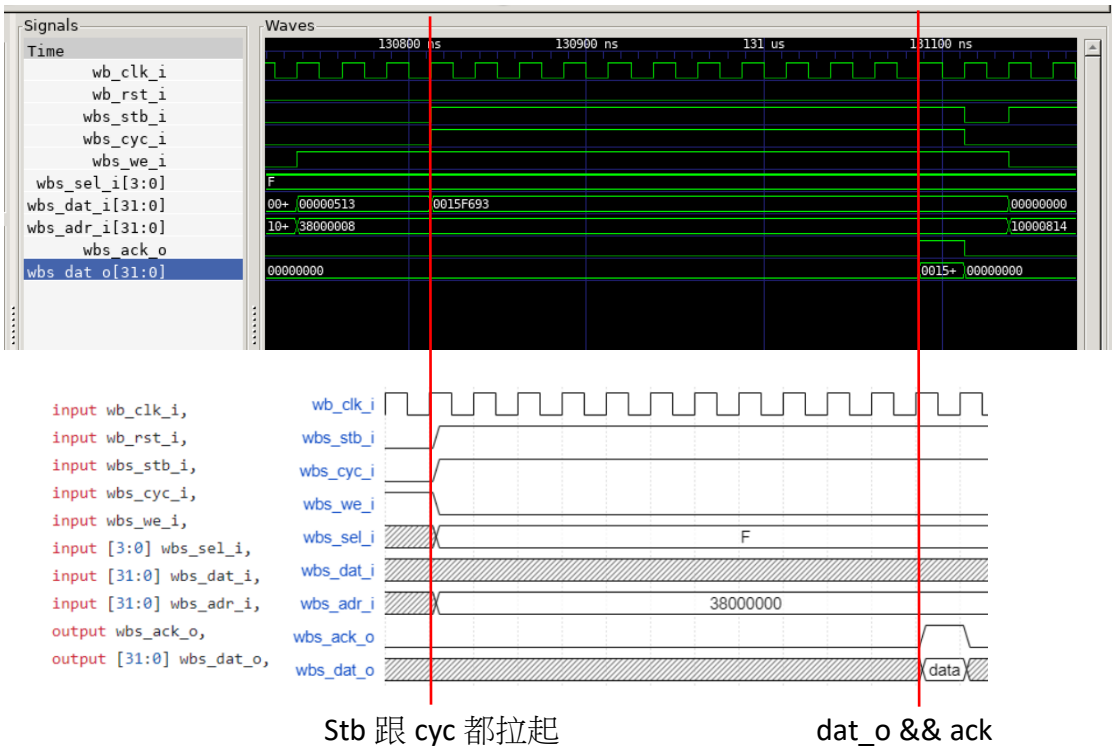


上面紅框即為每次做完 FIR 的結果，一樣會送到 mprj_io。

當 checkbits == 'hAB51 時，testbench 收到 end mark，test2 passed!



當 fir 結果算完會透過 wb_dat_i 傳給 BRAM。



最後我直接將我們的波型以及 ppt 的波型對齊，可以看到 stb&cyc 都 pull up 時，兩者 dat_o 跟 ack pull up 的 cycle 皆相同。

Synthesis report

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	13	0	0	53200	0.02
LUT as Logic	13	0	0	53200	0.02
LUT as Memory	0	0	0	17400	0.00
Slice Registers	5	0	0	106400	<0.01
Register as Flip Flop	5	0	0	106400	<0.01
Register as Latch	0	0	0	106400	0.00
F7 Muxes	0	0	0	26600	0.00
F8 Muxes	0	0	0	13300	0.00

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	4	0	0	140	2.86
RAMB36/FIFO*	4	0	0	140	2.86
RAMB36E1 only	4				
RAMB18	0	0	0	280	0.00

Design Timing Summary

WNS(ns)	TNS(ns)	TNS Failing Endpoints	TNS Total Endpoints	WHS(ns)	THS(ns)	THS Failing Endpoints	THS Total Endpoints	WPWS(ns)	TPWS(ns)	TPWS Failing Endpoints	TPWS Total Endpoints
7.365	0.000	0	9	0.146	0.000	0	9	4.500	0.000	0	14

All user specified timing constraints are met.