國立清華大學

系統晶片設計

SOC Design

國立清華大學

NATIONAL TSING HUA UNIVERSITY

Lab 4-1

組別: 第 12 組

學號:111063548、111061624、112501538

姓名:蕭方凱、尤弘瑋、葉承泓

指導老師:賴瑾教授

目錄

1. Explanation of firmware code

1-1. How does it execute a multiplication in assembly code?

我們在 fir.c 中寫上 FIR 運算的式子：

*outputsignal[i] ← outputsignal[i] + taps[j]*inputbuffer[data_RAM_pointer];*

其中會使用到乘法，因此在 compile 成 assembly code 後，會有一個區段(section)是負責做這個乘法運算的，底下說明如何在 assembly code 中執行到此區段以及此階段的乘法運算方式：

我們發現將 fir.c compile 後所得到的 counter_la_fir.out 檔案裡頭有清楚地寫出每個 assembly code 所對應的區段（如 Fig. 3 為<__mulsi3>區段），以及這些 code 轉換為機器語言後儲存在 configuration address map 中的哪個位址（例如 Fig. 3 中的”mv a2,a0”這個 instruction 就存在 32’h38000000 的位址）。這些 fir.c 中的 code 會存在 user project BRAM 所對應的位址(32’h38000000 附近)，是因為在 fir.c 的函數宣告時有特別加入「section (".mprjram")」，並且在 section.lds 檔案(Fig. 2)中有定義".mprjram"這個 section occupy 了從 0x38000000 開始，持續 0x00400000 長度的 memory configuration address，因此 compiler 知道要將這部分的 instruction code 存在這段區段中，也就是存到 user project BRAM 中。

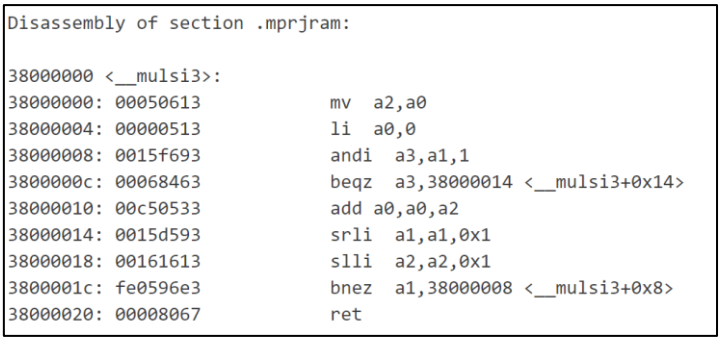
當 CPU 執行到<fir>區段（對應到 fir.c code），將要執行乘法的計算時，就會 jump（使用 jal (jump and link) instruction，如 Fig. 1 所示）到<__mulsi3>區段執行乘法運算。

```
678     3800012c:       ed5ff0ef                jal     ra,38000000 <__mulsi3>
```

Fig. 1

```
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
}
```

Fig. 2



```
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
```

Fig. 3

Fig. 3 為在 counter_la_fir.out 檔案中查找到關於乘法運算的區段[1]，可以得知 program counter (PC)在 jump 到 address 為 32'h38000000 的位置（也就是<__mulsi3>區段的開頭位址）後，將開始做乘法運算，其內部運算過程為：

首先，由於可將 jal 看作是呼叫乘法器函式，故須先將被乘數與乘數的 arguments 分別放在 a0 與 a1 兩個 register 中，然後才進入這個 section 做運算，最後的 return value 也須放在 a0 的 register 中才可被上層 function（<fir>）讀取到：

步驟	儲存的位址	instruction	說明
(1)	0x38000000	mv a2,a0	將 a0 的值複製到 a2
(2)	0x38000004	li a0,0	將"0"讀進 a0 中
(3)	0x38000008	andi a3,a1,1	將 a3 的值寫成 "a1 & 1"，也就是 a1[0]
(4)	0x3800000c	beqz a3,38000014 <__mulsi3+0x14>	若 a3 的值=0，則 jump to 0x38000014 的位址繼續執行
(5)	0x38000010	add a0,a0,a2	a0 ← a0+a2
(6)	0x38000014	srli a1,a1,0x1	a1 ← a1>>1
(7)	0x38000018	slli a2,a2,0x1	a2 ← a2<<1
(8)	0x3800001c	bnez a1,38000008 <__mulsi3+0x8>	若 a1 的值≠0，則 jump to 0x38000008 的位址繼續執行
(9)	0x38000020	ret	return to 上層函式 (<fir>)

Table 1

[1] 我們查到的參考資料(https://tomverbeure.github.io/rtl/2018/08/12/Multipliers.html)裡有寫到 RISC-V 的乘法 code 為 MUL 系列，在 counter_la_fir.out 中搜尋 MUL 只有找到<__mulsi3>這個區段有相關名稱，並且我們發現在<fir>區段中有呼叫到此區段，故認為應是乘法運算的相關區段。

由於其中有 branch 相關的 instruction，故不一定照著表中的順序執行。經過推導大致能得知它運算乘法的方式為：「**上層函式<fir>在 a0 及 a1 中放入兩個 arguments（被乘數與乘數），但由於 a0 最終需存著 return value，故先將被乘數(a0)複製到 a2 register 中，a0 則做為存乘積的 register，而 a1 則維持存著乘數。接著開始進行乘法運算：檢查乘數的 LSB 是否為 1，若是，則將此時的被乘數加至乘積 a0 中；若否，則不加至 a0 中。接著進行下一個 bit 的運算，但要先將乘數向右 shift 1 bit（代表要檢查下一個位數的值），並將被乘數向左 shift 1 bit（表示這個位數的 weighting 加了一個 order），再進行下一個 LSB 的判斷。最終當整個乘數被 shift 到變為 0 後，就表示運算完成，即可將乘數（剛好存在 return value register (a0)中）return 回上一層函式**」。底下舉個例子來說明：當被乘數 a0=11011、乘數 a1=101 時

(1) a2←a0=11011
(2) a0←0
(3) a3←a1[0]=1
(4) a3≠0→No jump
(5) a0← a0+a2=0+11011=11011（因 a1 的 LSB 為 1，所以將 11011 加至乘積中）
(6) a1←(a1>>1)=10
(7) a2 ← (a2<<1)=110110
(8) a1≠0→jump to (3)
(3) a3←a1[0]=0
(4) a3=0→jump to (6)（因 a1 的 LSB 為 0，所以不將 110110 加至乘積中）
(6) a1←(a1>>1)=1
(7) a2 ← (a2<<1)=1101100
(8) a1≠0→jump to (3)
(3) a3←a1[0]=1
(4) a3≠0→No jump
(5) a0← a0+a2=11011+1101100=10000111（因 a1 的 LSB 為 1，所以將 1101100 加至乘積中）
(6) a1←(a1>>1)=0
(7) a2 ← (a2<<1)= 11011000
(8) a1=0→No jump
(9) return

最終的乘積即為 10000111。將上述式子轉為十進位以方便觀察結果：

$$(11011)_2\times(101)_2=(27)_{10}\times(5)_{10}=(135)_{10}=(10000111)_2$$

計算結果正確無誤！

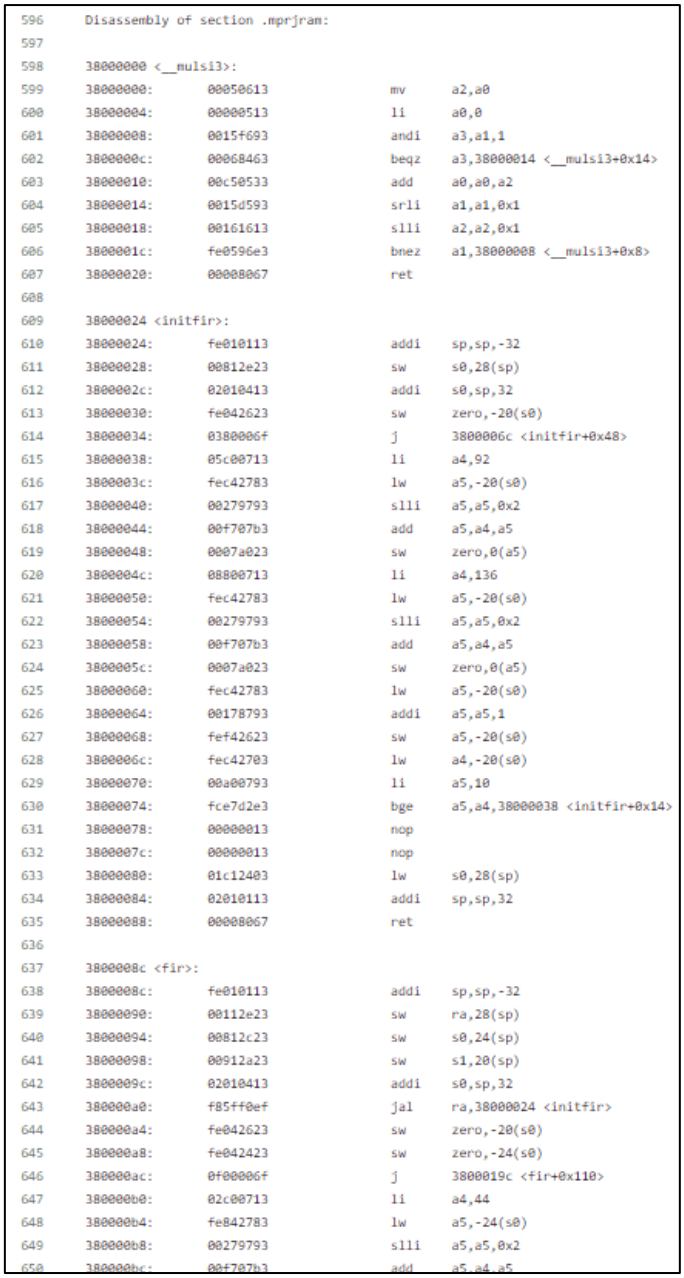
1-2. What address allocate for user project and how many space is required to allocate to firmware code?

① 在 section.lds 檔案(如下 Fig. 4)中，有定義 configuration address map，說明各個 memory address 區塊的分配，例如由圖中可知 mprjram 的 address 是定義從 0x38000000 開始，並持續 0x00400000 的長度。如此一來 wishbone 就可以知道想要 access 某處時需要給予哪段區間的 address 才能成功拿到。

```
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
}
```

Fig. 4

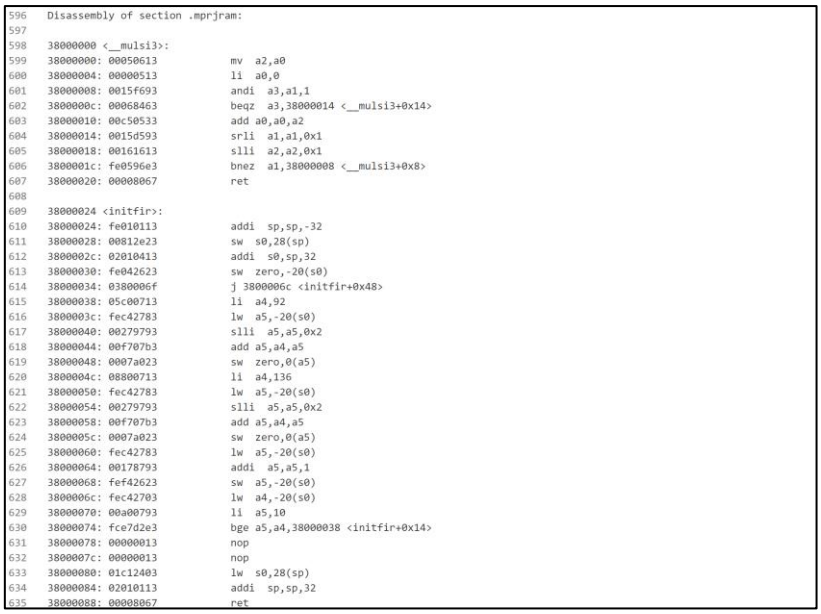
在上方 1-1.的說明中有提到在 user project 的 fir.c 中，這些我們所定義的 function 都有在函數宣告時加入「section(".mprjram")」的標記，表示我們想要將這段 code 轉換成的 assembly code（進而轉換成的機器語言.hex file）儲存到 0x38000000 的這個 memory 區塊，因此由 Fig. 4 可看出這段 section 所 allocate 給 user project 的 address space 為 mprjram 這個區塊，也就是 address 為 **0x38000000~0x383FFFFF**。（這個部分也能由底下的 Fig. 5 看出：此為 counter_la_fir.out 檔案(fir.c 經過 compile 之後的 output 檔案之一)中的一部份截圖，可看到包括乘法器的區段<__mulsi3>、fir initialization 的 function 對應的<initfir>區段，以及 fir()函數對應的區段<fir>都位在 0x3800000000 的區段中，由此可知我們所寫的這些 function 被轉成 assembly code/machine code 之後的確會被存放在這些 address space。）



```
596     Disassembly of section .mprjram:
597
598     38000000 <__mulsi3>:
599     38000000:       00050613                mv      a2,a0
600     38000004:       00000513                li      a0,0
601     38000008:       0015f693                andi    a3,a1,1
602     3800000c:       00068463                beqz    a3,38000014 <__mulsi3+0x14>
603     38000010:       00c50533                add     a0,a0,a2
604     38000014:       0015d593                srli    a1,a1,0x1
605     38000018:       00161613                slli    a2,a2,0x1
606     3800001c:       fe0596e3                bnez    a1,38000008 <__mulsi3+0x8>
607     38000020:       00008067                ret
608
609     38000024 <initfir>:
610     38000024:       fe010113                addi    sp,sp,-32
611     38000028:       00812e23                sw      s0,28(sp)
612     3800002c:       02010413                addi    s0,sp,32
613     38000030:       fe042623                sw      zero,-20(s0)
614     38000034:       0380006f                j       3800006c <initfir+0x48>
615     38000038:       05c00713                li      a4,92
616     3800003c:       fec42783                lw      a5,-20(s0)
617     38000040:       00279793                slli    a5,a5,0x2
618     38000044:       00f707b3                add     a5,a4,a5
619     38000048:       0007a023                sw      zero,0(a5)
620     3800004c:       08800713                li      a4,136
621     38000050:       fec42783                lw      a5,-20(s0)
622     38000054:       00279793                slli    a5,a5,0x2
623     38000058:       00f707b3                add     a5,a4,a5
624     3800005c:       0007a023                sw      zero,0(a5)
625     38000060:       fec42783                lw      a5,-20(s0)
626     38000064:       00178793                addi    a5,a5,1
627     38000068:       fef42623                sw      a5,-20(s0)
628     3800006c:       fec42703                lw      a4,-20(s0)
629     38000070:       00a00793                li      a5,10
630     38000074:       fce7d2e3                bge     a5,a4,38000038 <initfir+0x14>
631     38000078:       00000013                nop
632     3800007c:       00000013                nop
633     38000080:       01c12403                lw      s0,28(sp)
634     38000084:       02010113                addi    sp,sp,32
635     38000088:       00008067                ret
636
637     3800008c <fir>:
638     3800008c:       fe010113                addi    sp,sp,-32
639     38000090:       00112e23                sw      ra,28(sp)
640     38000094:       00812c23                sw      s0,24(sp)
641     38000098:       00912a23                sw      s1,20(sp)
642     3800009c:       02010413                addi    s0,sp,32
643     380000a0:       f85ff0ef                jal     ra,38000024 <initfir>
644     380000a4:       fe042623                sw      zero,-20(s0)
645     380000a8:       fe042423                sw      zero,-24(s0)
646     380000ac:       0f00006f                j       3800019c <fir+0x110>
647     380000b0:       02c00713                li      a4,44
648     380000b4:       fe842783                lw      a5,-24(s0)
649     380000b8:       00279793                slli    a5,a5,0x2
650     380000bc:       00f707b3                add     a5,a4,a5
```

Fig. 5

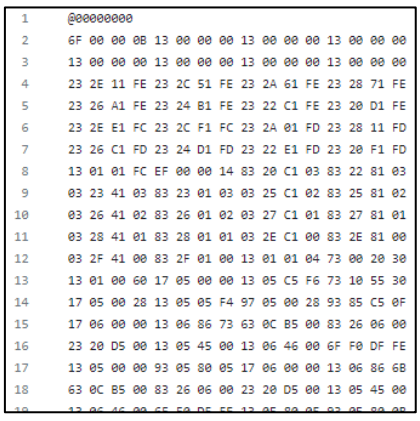
②在 counter_la_fir.out 中（如下 Fig. 6），我們可看到這些 compile 後的 assembly code 所存放的 address、instruction 的內容，以及轉換成 machine code 的 16 進位 code，例如圖中第 599 行為此 section 的第一個 instruction，為 ”mv a2,a0”，所對應的 machine code 為 “00050613” (in hexadecimal)，會存放在 “0x38000000” 的 address。



```
596  Disassembly of section .mprjram:
597
598  38000000 <__mulsi3>:
599  38000000: 00050613          mv  a2,a0
600  38000004: 00000513          li  a0,0
601  38000008: 0015f693          andi  a3,a1,1
602  3800000c: 00068463          beqz  a3,38000014 <__mulsi3+0x14>
603  38000010: 00c50533          add a0,a0,a2
604  38000014: 0015d593          srli  a1,a1,0x1
605  38000018: 00161613          slli  a2,a2,0x1
606  3800001c: fe0596e3          bnez  a1,38000008 <__mulsi3+0x8>
607  38000020: 00008067          ret
608
609  38000024 <initfir>:
610  38000024: fe010113          addi  sp,sp,-32
611  38000028: 00812e23          sw  s0,28(sp)
612  3800002c: 02010413          addi  s0,sp,32
613  38000030: fe042623          sw  zero,-20(s0)
614  38000034: 0380006f          j 3800006c <initfir+0x48>
615  38000038: 05c00713          li  a4,92
616  3800003c: fec42783          lw  a5,-20(s0)
617  38000040: 00279793          slli  a5,a5,0x2
618  38000044: 00f707b3          add a5,a4,a5
619  38000048: 0007a023          sw  zero,0(a5)
620  3800004c: 08800713          li  a4,136
621  38000050: fec42783          lw  a5,-20(s0)
622  38000054: 00279793          slli  a5,a5,0x2
623  38000058: 00f707b3          add a5,a4,a5
624  3800005c: 0007a023          sw  zero,0(a5)
625  38000060: fec42783          lw  a5,-20(s0)
626  38000064: 00178793          addi  a5,a5,1
627  38000068: fef42623          sw  a5,-20(s0)
628  3800006c: fec42703          lw  a4,-20(s0)
629  38000070: 00a00793          li  a5,10
630  38000074: fce7d2e3          bge a5,a4,38000038 <initfir+0x14>
631  38000078: 00000013          nop
632  3800007c: 00000013          nop
633  38000080: 01c12403          lw  s0,28(sp)
634  38000084: 02010113          addi  sp,sp,32
635  38000088: 00008067          ret
```

Fig. 6 counter_la_fir.out

在 fir.c 經過 compile 後也會產生 counter_la_fir.hex 檔（如下 Fig. 7），這是實際被 testbench 所引入的檔案，會實際被吃到 Caravel SoC 中（在此 lab 中這部分的 code 會存放在 user project BRAM 中），可看到其內容中 address 是從@00000000 開始計算，並只寫出 machine code，將其與 counter_la_fir.out 檔中的 machine code 對照後，我們發現此檔案中被分為 3 個部分，如 Table 2 所示。



```
1   @00000000
2   6F 00 00 0B 13 00 00 00 13 00 00 00 13 00 00 00
3   13 00 00 00 13 00 00 00 13 00 00 00 13 00 00 00
4   23 2E 11 FE 23 2C 51 FE 23 2A 61 FE 23 28 71 FE
5   23 26 A1 FE 23 24 B1 FE 23 22 C1 FE 23 20 D1 FE
6   23 2E E1 FC 23 2C F1 FC 23 2A 01 FD 23 28 11 FD
7   23 26 C1 FD 23 24 D1 FD 23 22 E1 FD 23 20 F1 FD
8   13 01 01 FC EF 00 00 14 83 20 C1 03 83 22 81 03
9   03 23 41 03 83 23 01 03 03 25 C1 02 83 25 81 02
10  03 26 41 02 83 26 01 02 03 27 C1 01 83 27 81 01
11  03 28 41 01 83 28 01 01 03 2E C1 00 83 2E 81 00
12  03 2F 41 00 83 2F 01 00 13 01 01 04 73 00 20 30
13  13 01 00 60 17 05 00 00 13 05 C5 F6 73 10 55 30
14  17 05 00 28 13 05 05 F4 97 05 00 28 93 85 C5 0F
15  17 06 00 00 13 06 86 73 63 0C B5 00 83 26 06 00
16  23 20 D5 00 13 05 45 00 13 06 46 00 6F F0 DF FE
17  13 05 00 00 93 05 80 05 17 06 00 00 13 06 86 6B
18  63 0C B5 00 83 26 06 00 23 20 D5 00 13 05 45 00
```

Fig. 7 counter_la_fir.hex

counter_la_fir.hex（Fig. 7）	**counter_la_fir.out（Fig. 6）**	**對應 software 部分**
@00000000~@000007B0	0x10000000~0x100007a8 的 section	counter_la_fir.c
@000007B0~@00000808 之間	第 544~594 行 （memory array 的部分）	fir.h (Tap[N]、inputsignal[N]…)
@00000808 以下	0x38000000 的 section	fir.c

Table 2

我們所寫的 fir.c 中的 code 對應到@00000808 以下的段落（如 Fig. 8），例如第一個 instruction 為前面有提過的 ”mv a2,a0”，所對應的 machine code 為 “00050613” (in hexadecimal)，但在這裡因為機器讀取時的 Endian 定義方式不同（little-Endian v.s. big-endian），所以需調換 byte 之間的順序為 ”13060500”，也就是 Fig. 8 圖中的第 133 行前 8 個數字。

```
37 07 51 AB 23 A0 E7 00 13 00 00 00 83 20 C1 01
03 24 81 01 13 01 01 02 67 80 00 00
@000007B0
00 00 00 00 F6 FF FF FF F7 FF FF FF 17 00 00 00
38 00 00 00 3F 00 00 00 38 00 00 00 17 00 00 00
F7 FF FF FF F6 FF FF FF 00 00 00 00 01 00 00 00
02 00 00 00 03 00 00 00 04 00 00 00 05 00 00 00
06 00 00 00 07 00 00 00 08 00 00 00 09 00 00 00
0A 00 00 00 0B 00 00 00
@00000808
13 06 05 00 13 05 00 00 93 F6 15 00 63 84 06 00
33 05 C5 00 93 D5 15 00 13 16 16 00 E3 96 05 FE
67 80 00 00 13 01 01 FE 23 2E 81 00 13 04 01 02
23 26 04 FE 6F 00 80 03 13 07 C0 05 83 27 C4 FE
93 97 27 00 B3 07 F7 00 23 A0 07 00 13 07 80 08
83 27 C4 FE 93 97 27 00 B3 07 F7 00 23 A0 07 00
83 27 C4 FE 93 87 17 00 23 26 F4 FE 03 27 C4 FE
93 07 A0 00 E3 D2 E7 FC 13 00 00 00 13 00 00 00
03 24 C1 01 13 01 01 02 67 80 00 00 13 01 01 FE
23 2E 11 00 23 2C 81 00 23 2A 91 00 13 04 01 02
EF F0 5F F8 23 26 04 FE 23 24 04 FE 6F 00 00 0F
13 07 C0 02 83 27 84 FE 93 97 27 00 B3 07 F7 00
83 A7 07 00 23 20 F4 FE 13 07 C0 05 83 27 C4 FE
93 97 27 00 B3 07 F7 00 03 27 04 FE 23 A0 E7 00
23 22 04 FE 6F 00 00 0A 13 07 80 08 83 27 84 FE
93 97 27 00 B3 07 F7 00 83 A4 07 00 13 07 00 00
83 27 44 FE 93 97 27 00 B3 07 F7 00 83 A6 07 00
13 07 C0 05 83 27 C4 FE 93 97 27 00 B3 07 F7 00
83 A7 07 00 93 85 07 00 13 85 06 00 EF F0 5F ED
93 07 05 00 33 87 F4 00 93 06 80 08 83 27 84 FE
93 97 27 00 B3 87 F6 00 23 A0 E7 00 03 27 44 FE
93 07 A0 00 63 02 F7 02 03 27 C4 FE 93 07 A0 00
63 16 F7 00 23 26 04 FE 6F 00 00 01 83 27 C4 FE
93 87 17 00 23 26 F4 FE 83 27 44 FE 93 87 17 00
23 22 F4 FE 03 27 44 FE 93 07 A0 00 E3 DE E7 F4
83 27 84 FE 93 87 17 00 23 24 F4 FE 03 27 84 FE
93 07 A0 00 E3 D6 E7 F0 93 07 80 08 13 85 07 00
83 20 C1 01 03 24 81 01 83 24 41 01 13 01 01 02
67 80 00 00
```

Fig. 8 counter_la_fir.hex

因此 firmware code 被分成 3 個區域，其各自的 size 如下：

software 部分	**counter_la_fir.out**	**counter_la_fir.hex**	**需多少 address**	**需多少 memory size**
counter_la_fir.c	0x10000000~ 0x100007a8 的 section	@00000000~@000007B0	1964 個	**1964 Bytes**
fir.h (Tap[N]、inputsignal[N]…)	第 544~594 行（memory array 的部分）	@000007B0~@00000808 之間	88 個	**88 Bytes**
fir.c	0x38000000 的 section	@00000808 以下	452 個	**452 Bytes**

Table 3

此表格中最右邊兩個 column 的計算方式如下：以前面的例子”mv a2,a0”為例，其所對應的 machine code 為 “00050613” (in hexadecimal)（在 counter_la_fir.hex 中寫作 “13 06 05 00”），由於此為 16 進位的表示方法，故「1」、「3」、「0」、「6」、「0」、「5」、「0」、「0」各自代表了 4 bits，每兩個 digit 組合起來形成 1 byte，故只需在 counter_la_fir.hex 檔案中計算各區段有幾組(每兩個 digit 一組)數字，即可知道這區域的 firmware code/machine code 需要多少 memory 來儲存，即可得到最右 column 的數值；此外，在 1-1.的 Fig. 3 中對照 instruction、machine code 及所存放之 address 可知：一行 assembly code 需要 4bytes（32bits, 4 個 address）來存，而每個 address 儲存 1byte (=8 bits)，由此可求得分別需要多少 address 來儲存這些 firmware codes。

若要計算這些所有 firmware code 所需 allocate 的 memory space，則將這 3 者加在一起即可：共需要 1964+88+452=2504 (Bytes)，若只想要看 fir.h 及 fir.c 所需使用的 memory space 量，則為 88+452=540 (Bytes)。

2. Interface between bram and wishbone

2-1. Waveform from xsim

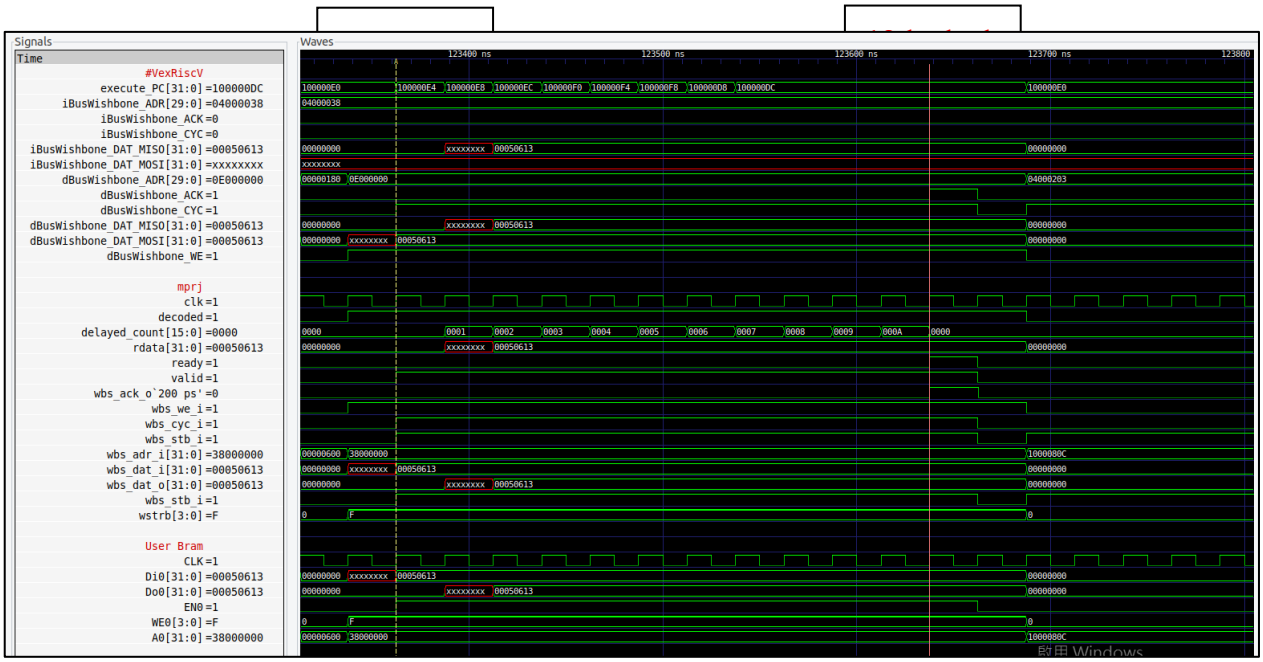


Fig 1 waveform of xsim

圖中紅色框框處為 WB 的訊號，圖中底下的部分為 BRAM 的存取訊號，訊號是按照 submission guide 中的波形操作，其操作是當 WB 的 STB 與 CYC 為 1 時，且 check address 的位置的確為分配給 user project 0x380 開頭的位址，則開始向 BRAM 要求 data(包含 CPU 要執行的 instruction 及 tap[N]、inputbuffer[N]等 data，分別存放在上方 Table 3 中的位址)。當 WE(write enable)為 1 時則寫入 BRAM；當 WE 為 0 時則從 BRAM 讀出 data，而 address 則按照 WB 所給的 address(0x380 開頭)。存取 BRAM 的過程會需要 2 個 cycle，並且按照題目的設定需再 delay 10 個 cycle 才能在 WB 回傳 ACK 及 BRAM 的 data，這 10 個 cycle 是利用「delayed_count」這個 register，它會在 reset 時回到 0，並且在存取 BRAM 時開始往上數，直到數到 10 才將 BRAM 接收的 channel ready，此時才會 ACK 讓外界能接收 wbs_dat_o，並且回到 0，為下次 access 做準備。故上圖從「STB 與 CYC 為 1」至「ACK 拉起來」共過了 11 個 cycle（第 12 個 cycle 才拉起來）。

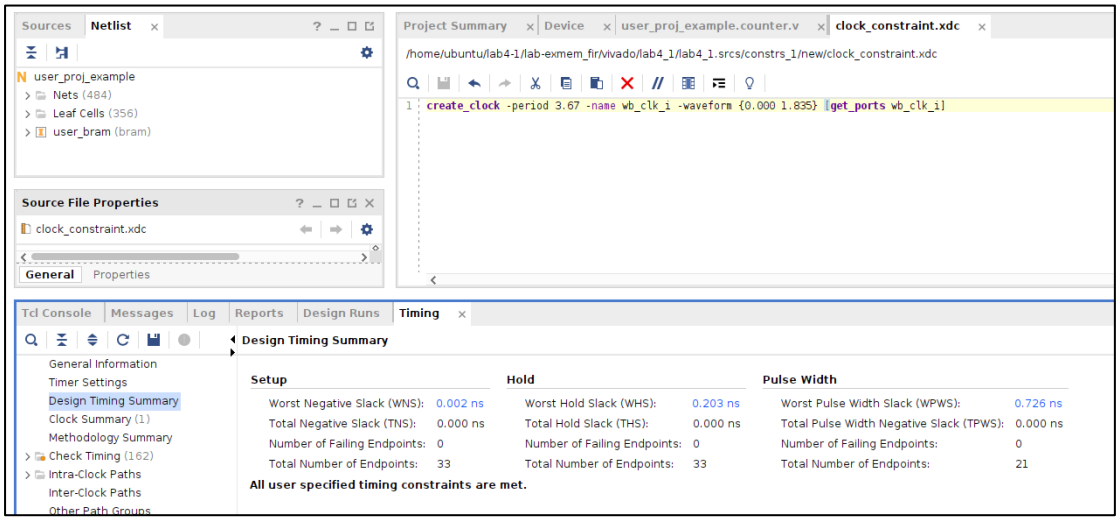
2-2. FSM

本 lab 中沒有特別使用到 FSM，主要是透過將 input 訊號以及 BRAM 的 output 訊號做邏輯閘運算 or 直接接線而得到 output 訊號，其中有將 address decode 以確認 WB 的 address 為 0x380 開頭(表示 slave 是要存取 user project 的位址中的資料)，才會輸出 ACK 與 output data 訊號，例如：

```
assign decoded = wbs_adr_i[31:20] == 12'h380 ? 1'b1 : 1'b0;
assign valid = wbs_cyc_i && wbs_stb_i && decoded;
assign wbs_dat_o = rdata;
assign wbs_ack_o = ready;
```

3. Synthesis report

我們將 design（包括 fir.c、fir.h、user_proj_example.counter.v、bram.v，以及修改 testbench 以偵測 output signal 並驗證其正確性）加入 Vivado 軟體中（simulation 是透過 run_sim 的 script 執行的，並無使用 Vivado），並加上 timing constraint 後，執行 synthesis，可得到如下圖的結果，可看到最高速度大約是 clock period≒3.67ns，對應到速度約為 272.48MHz。此時的 slack 為正值。



(1) User_proj_example_utilization_synth.rpt

執行 synthesis 後，可得到合成後的 utilization report，即 user_proj_example _utilization_synth.rpt。開啟此檔案可得到下圖的資源使用情形：

FF & LUT:

1. Slice Logic

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs*	34	0	0	53200	0.06
LUT as Logic	34	0	0	53200	0.06
LUT as Memory	0	0	0	17400	0.00
Slice Registers	17	0	0	106400	0.02
Register as Flip Flop	17	0	0	106400	0.02
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

Fig 2 Slice Logic

BRAM:

2. Memory

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	2	0	0	140	1.43
RAMB36/FIFO*	2	0	0	140	1.43
RAMB36E1 only	2				
RAMB18	0	0	0	280	0.00

Fig 3 Memory

DSP:

3. DSP

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	220	0.00

Fig 4 DSP

IO:

4. IO and GT Specific

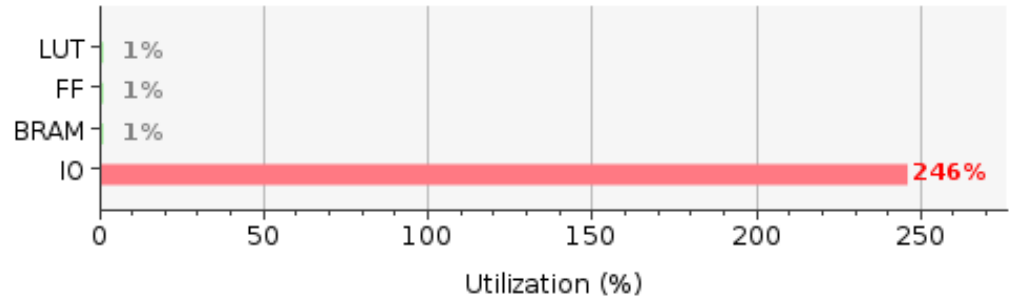
Site Type	Used	Fixed	Prohibited	Available	Util%
Bonded IOB	308	0	0	125	246.40
Bonded IPADs	0	0	0	2	0.00
Bonded IOPADs	0	0	0	130	0.00
PHY_CONTROL	0	0	0	4	0.00
PHASER_REF	0	0	0	4	0.00
OUT_FIFO	0	0	0	16	0.00
IN_FIFO	0	0	0	16	0.00
IDELAYCTRL	0	0	0	4	0.00
IBUFDS	0	0	0	121	0.00
PHASER_OUT/PHASER_OUT_PHY	0	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

Fig 5 IO and GT Specific

(2) In vivado utilization summary

另外在 Vivado 中按下”Report Utilization”後會產生下圖(更為視覺化的整理):

Resource	Utilization	Available	Utilization %
LUT	34	53200	0.06
FF	17	106400	0.02
BRAM	2	140	1.43
IO	308	125	246.40



由(1)、(2)點的 report 結果可知，此 design 合成後所使用的

1. **FF 個數為 17 個**，而此 FPGA 板中共有 106400 個，故 **utilization 為 0.02%**
2. **LUT 個數為 34 個**，而此 FPGA 板中共有 53200 個，故 **utilization 為 0.06%**
3. **BRAM 使用 2 個**，而此 FPGA 板中共有 140 個，故 **utilization 為 1.43%**

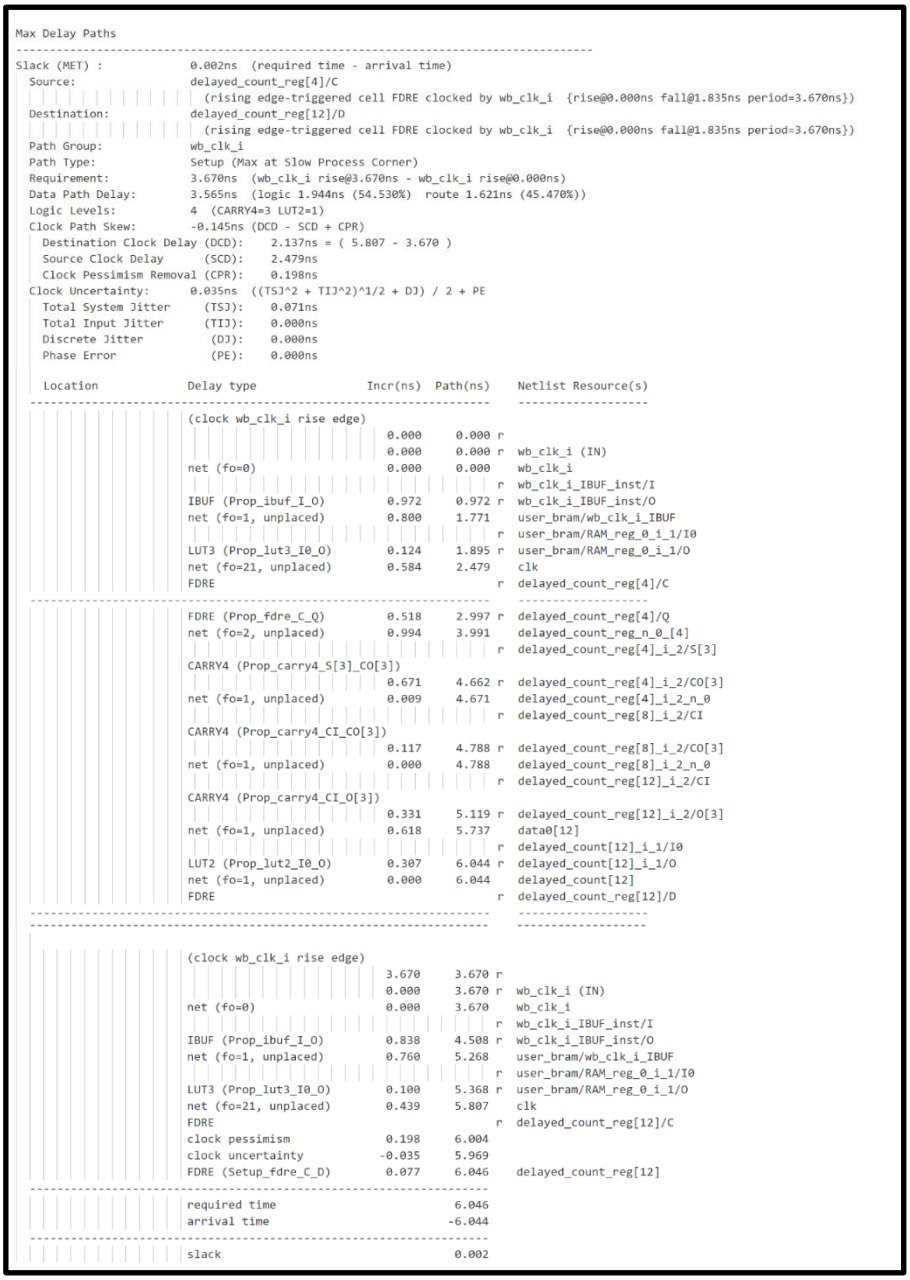
(3) User_proj_example.vds

```
Detailed RTL Component Info :
+---Adders :
      2 Input   16 Bit       Adders := 1
+---Registers :
                32 Bit    Registers := 1
                16 Bit    Registers := 1
                 1 Bit    Registers := 1
+---RAMs :
               64K Bit   (2048 X 32 bit)          RAMs := 1
+---Muxes :
      2 Input   32 Bit        Muxes := 6
      2 Input   16 Bit        Muxes := 1
      2 Input    8 Bit        Muxes := 1
      2 Input    1 Bit        Muxes := 2
```

(4) Timing_report_3_67ns.txt

使用 Vivado 匯入 design 後，執行 synthesis，並盡量設定最高速度（我們一開始先使用 clock period=10ns 合成，發現 slack 為+6 點多，因此可再加快操作速度：此 design 所能達到的最快速度約為 clock period=3.67ns，換算下來 maximum frequency 約為 272.48MHz），再按照 lab3 作業區的 SYN_Workflow.pptx 的操作流程，可得到合成後的 timing report，即 timing_report.txt，其中有一部份寫到關於 longest path：（由於文件較長，故分多次截圖）

Clock period: 3.67ns



```
Max Delay Paths
--------------------------------------------------------------------------------
Slack (MET) :             0.002ns  (required time - arrival time)
  Source:                 delayed_count_reg[4]/C
                            (rising edge-triggered cell FDRE clocked by wb_clk_i  {rise@0.000ns fall@1.835ns period=3.670ns})
  Destination:            delayed_count_reg[12]/D
                            (rising edge-triggered cell FDRE clocked by wb_clk_i  {rise@0.000ns fall@1.835ns period=3.670ns})
  Path Group:             wb_clk_i
  Path Type:              Setup (Max at Slow Process Corner)
  Requirement:            3.670ns  (wb_clk_i rise@3.670ns - wb_clk_i rise@0.000ns)
  Data Path Delay:        3.565ns  (logic 1.944ns (54.530%)  route 1.621ns (45.470%))
  Logic Levels:           4  (CARRY4=3 LUT2=1)
  Clock Path Skew:        -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD):    2.137ns = ( 5.807 - 3.670 )
    Source Clock Delay      (SCD):    2.479ns
    Clock Pessimism Removal (CPR):    0.198ns
  Clock Uncertainty:      0.035ns  ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter     (TSJ):    0.071ns
    Total Input Jitter      (TIJ):    0.000ns
    Discrete Jitter          (DJ):    0.000ns
    Phase Error              (PE):    0.000ns

    Location             Delay type                Incr(ns)  Path(ns)    Netlist Resource(s)
  -------------------------------------------------------------------    -------------------
                         (clock wb_clk_i rise edge)
                                                      0.000     0.000 r
                                                      0.000     0.000 r  wb_clk_i (IN)
                         net (fo=0)                   0.000     0.000    wb_clk_i
                                                                       r  wb_clk_i_IBUF_inst/I
                         IBUF (Prop_ibuf_I_O)         0.972     0.972 r  wb_clk_i_IBUF_inst/O
                         net (fo=1, unplaced)         0.800     1.771    user_bram/wb_clk_i_IBUF
                                                                       r  user_bram/RAM_reg_0_i_1/I0
                         LUT3 (Prop_lut3_I0_O)        0.124     1.895 r  user_bram/RAM_reg_0_i_1/O
                         net (fo=21, unplaced)        0.584     2.479    clk
                         FDRE                                           r  delayed_count_reg[4]/C
  -------------------------------------------------------------------    -------------------
                         FDRE (Prop_fdre_C_Q)         0.518     2.997 r  delayed_count_reg[4]/Q
                         net (fo=2, unplaced)         0.994     3.991    delayed_count_reg_n_0_[4]
                                                                       r  delayed_count_reg[4]_i_2/S[3]
                         CARRY4 (Prop_carry4_S[3]_CO[3])
                                                      0.671     4.662 r  delayed_count_reg[4]_i_2/CO[3]
                         net (fo=1, unplaced)         0.009     4.671    delayed_count_reg[4]_i_2_n_0
                                                                       r  delayed_count_reg[8]_i_2/CI
                         CARRY4 (Prop_carry4_CI_CO[3])
                                                      0.117     4.788 r  delayed_count_reg[8]_i_2/CO[3]
                         net (fo=1, unplaced)         0.000     4.788    delayed_count_reg[8]_i_2_n_0
                                                                       r  delayed_count_reg[12]_i_2/CI
                         CARRY4 (Prop_carry4_CI_O[3])
                                                      0.331     5.119 r  delayed_count_reg[12]_i_2/O[3]
                         net (fo=1, unplaced)         0.618     5.737    data0[12]
                                                                       r  delayed_count[12]_i_1/I0
                         LUT2 (Prop_lut2_I0_O)        0.307     6.044 r  delayed_count[12]_i_1/O
                         net (fo=1, unplaced)         0.000     6.044    delayed_count[12]
                         FDRE                                           r  delayed_count_reg[12]/D
  -------------------------------------------------------------------    -------------------
  -------------------------------------------------------------------    -------------------

                         (clock wb_clk_i rise edge)
                                                      3.670     3.670 r
                                                      0.000     3.670 r  wb_clk_i (IN)
                         net (fo=0)                   0.000     3.670    wb_clk_i
                                                                       r  wb_clk_i_IBUF_inst/I
                         IBUF (Prop_ibuf_I_O)         0.838     4.508 r  wb_clk_i_IBUF_inst/O
                         net (fo=1, unplaced)         0.760     5.268    user_bram/wb_clk_i_IBUF
                                                                       r  user_bram/RAM_reg_0_i_1/I0
                         LUT3 (Prop_lut3_I0_O)        0.100     5.368 r  user_bram/RAM_reg_0_i_1/O
                         net (fo=21, unplaced)        0.439     5.807    clk
                         FDRE                                           r  delayed_count_reg[12]/C
                         clock pessimism              0.198     6.004
                         clock uncertainty           -0.035     5.969
                         FDRE (Setup_fdre_C_D)        0.077     6.046    delayed_count_reg[12]
  -------------------------------------------------------------------
                         required time                          6.046
                         arrival time                          -6.044
  -------------------------------------------------------------------
                         slack                                  0.002
```

由上圖可知：此 **longest path 的 timing/delay 為 3.565ns**，而其 **slack 為 0.002ns**，為正的值。

4. Other discoveries

1. Run_sim 結果：

```
ubuntu@ubuntu2004:~/lab4-1/lab-exmem_fir/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
Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed,
0x044a, which is  1098 in decimal
LA Test 2 passed
```

2. 經過這次 lab 的實作，我們第一次看到 hardware 與 software co-design 的過程，原來兩者之間傳遞 data 及 instruction 可以透過像是 wishbone 這種 interface，讓 CPU 所需的 data 能從 BRAM 讀出。在研究這整個 system 運作的過程中，我們一開始感到最困惑的部分是在 lab3 中 input data 是透過 testbench 傳進來，但在這次 lab 中並沒有看到 testbench 傳進來這些 input signal，經過向同學詢問以及看了 lab4-2 的 workbook 介紹後，才得知原來這些 data 是直接存在 inputsignal 中（當作是已存放在 BRAM 中的 data），由此開始我們的進度才開始大幅前進，並且也滿順利的完成 software 的 FIR Design（使用的是 lab3 中 hardware 的運算方式，只是用 software 來達成，且 input signal 可透過 access inputsignal[] array 來取得）！另外讓我們困惑的是 WB interface 與 mprj interface 的功能，在經過 trace code 以及參考 Github 中其他資料夾（caravel-soc、caravel-soc_fpga-lab/lab-fir、caravel-soc_fpga-lab/lab-exmem）的實作方法以及老師、助教的上課講解後，我們才逐漸得知 mprj 是 CPU 與外界(testbench)的溝通橋樑，CPU 可將計算完畢的 outputsignal[] array 的位址回傳給 counter_la_fir.c 並在 reg_mprj_datal 上放入計算結果，如此一來 testbench 即可接收到這個位置的值（在 testbench 中被叫做 “checkbits”），也就可以檢查答案是否正確（如下圖）。

```
////////////////////////////////// Added ///////////////////////////////////
wait(checkbits == 16'd1098);
$display("Call function fir() in User Project BRAM (mprjram, 0x38000000) return value passed, 0x%x, which is %d in decimal", checkbits, checkbits);
////////////////////////////////////////////////////////////////////////////
```

而 WB 的 interface 則是透過 CPU 中執行 C code 的「指標運算」來達到，而 CPU 運行所需的 instruction 也是透過這個 interface 去 fetch user project BRAM 得到。

3. 關於 WB 的 address 也是個令我們困惑的問題：如何跟 user project 之間溝通。最後我們才懂原來是因為 WB 上不同 slave 之間有不同的 global address，包含 user project 的 instruction 所存放的 BRAM 位址也都有 global address，並且這些 address 的 section 是定義在 section.lds 中，slave 只會在自己的 address 被

broadcast 在 WB 上時，才會對其有反應，這也是 CPU 得以 access 到 user project BRAM 而取得 instruction/data 的方式!相信這些知識都會在未來的 lab 中使用到！