SoC Design

Lab 4-1 (Team project)

Student ID: 110590022 Name(Chinese/English): 陳冠晰 Vic
Student ID: 110000107 Name(Chinese/English): 陳柏翰 Lukas
Student ID: 110061217 Name(Chinese/English): 王彥智 Kenny

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

    Let's look at the __mulsi3 function:

```
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
```

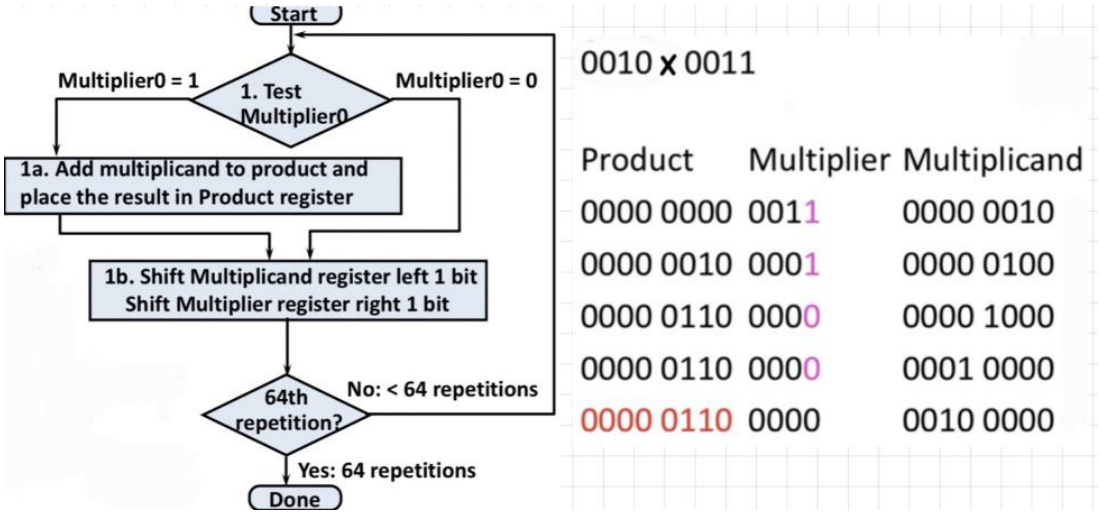
- **mv a2, a0**: This instruction moves the value from register a0 to register a2. In other words, it makes a copy of the value in a0 and stores it in a2.
- **li a0, 0**: This instruction loads an immediate value into register a0, setting it to 0. It essentially clears the content of a0.
- **andi a3, a1, 1**: This instruction performs a bitwise AND operation between the value in register a1 and the immediate value 1. The result is stored in a3. It checks the least significant bit of a1 and stores it in a3.
- **beqz a3, 38000014 <__mulsi3+0x14>**: This is a conditional branch instruction. It checks if the value in a3 (which represents the least significant bit of a1) is zero (beqz = branch if equal to zero). If it is zero, the program jumps to the address 38000014, which skips the next instruction.
- **add a0, a0, a2**: This instruction adds the value in a2 to the value in a0

and stores the result in a0. It effectively accumulates the value in a2 into a0.

- **srli a1, a1, 0x1**: This instruction performs a right shift logical operation on the value in a1, shifting it right by 1 bit position. This is effectively dividing a1 by 2.
- **slli a2, a2, 0x1**: This instruction performs a left shift logical operation on the value in a2, shifting it left by 1 bit position. This is effectively multiplying a2 by 2.
- **bnez a1, 38000008 <__mulsi3+0x8>**: This is a conditional branch instruction. It checks if the value in a1 is non-zero (bnez = branch if not equal to zero). If a1 is non-zero, the program jumps back to address 38000008(andi a3, a1, 1), effectively creating a loop that repeats until a1 becomes zero.
- **ret**: This instruction is a return instruction. It indicates the end of the <__mulsi3> function and returns control to the calling function.

That's look at the following example:

$7 * 6 = 42$
$\rightarrow 000111_2 * 000110_2$
$= 000111 * 000100 + 000111 * 000010$
$= 011100 * 1 + 001110 * 1(a2\ shift\ right, a1\ shift\ left)$
$= 011100 + 001110(accumulate\ the\ value)$
$= 101010_2$



Flow of doing multiplication

- **The way assembly code run our firmware code.**

```
637 3800008c <fir>:
638 3800008c:       fd010113                addi    sp,sp,-48
639 38000090:       02112623                sw      ra,44(sp)
640 38000094:       02812423                sw      s0,40(sp)
```

638: This instruction subtracts 48 from the stack pointer, effectively allocating space on the stack for local variables and saving space for return address (ra) and s0 register.

639-640: These instructions store the values of the ra (return address) and s0 registers onto the stack.

```
641 38000098:       03010413                addi    s0,sp,48
642 3800009c:       f89ff0ef                jal     ra,38000024 <initfir>
643 380000a0:       fe042623                sw      zero,-20(s0)
```

641: This instruction sets s0 to point to the start of the allocated stack space, effectively creating a frame pointer for local variable access.

642: This is a jump-and-link instruction, which calls a function at the address 0x38000024 (<initfir>), and stores the return address in ra.

643: These instructions initialize a value in memory at address s0-20 and then jump to label 0x380001a0.

```
644 380000a4:       0fc0006f                j       380001a0 <fir+0x114>
```

644: jump to 380001a0

```
707 380001a0:       fec42703                lw      a4,-20(s0)
708 380001a4:       00a00793                li      a5,10
709 380001a8:       f0e7d0e3                bge     a5,a4,380000a8 <fir+0x1c>
```

Probably for loop operation:

707: now a4 is zero

708: a5 = 10

709: since 10 > 0, branch to 380000a8

```
645 380000a8:       02c00713                li      a4,44
646 380000ac:       fec42783                lw      a5,-20(s0)
647 380000b0:       00279793                slli    a5,a5,0x2
648 380000b4:       00f707b3                add     a5,a4,a5
649 380000b8:       0007a703                lw      a4,0(a5)
650 380000bc:       05c00693                li      a3,92
651 380000c0:       fec42783                lw      a5,-20(s0)
652 380000c4:       00279793                slli    a5,a5,0x2
653 380000c8:       00f687b3                add     a5,a3,a5
654 380000cc:       00e7a023                sw      a4,0(a5)
655 380000d0:       fe042423                sw      zero,-24(s0)
656 380000d4:       fe042223                sw      zero,-28(s0)
```

645: a4 = 44

646: now -20(s0) = a5 = 10

647: a5 = 4 * 10

648: a5 = a4 + a5 = 84

649: a4 = 0(a5)

650: a3 = 92

651: a5 = 10

652: a5 = 40

653: a5 = a5 + a3 = 132

654: now 0(a5) = a4

655: -24(s0) = 0

656: -28(s0) = 0

```
657 380000d8:       0980006f                j       38000170 <fir+0xe4>
```

657: jump to 38000170

j loop

695: a4 = 0 (j)

696: a5 = 10 (N)

697: a5 >= 0 so jump to 380000dc (into the j loop)

```
658 380000dc:       00000713                li      a4,0
659 380000e0:       fe442783                lw      a5,-28(s0)
660 380000e4:       00279793                slli    a5,a5,0x2
661 380000e8:       00f707b3                add     a5,a4,a5
662 380000ec:       0007a783                lw      a5,0(a5)
663 380000f0:       fef42023                sw      a5,-32(s0)
664 380000f4:       fe442703                lw      a4,-28(s0)
665 380000f8:       fec42783                lw      a5,-20(s0)
666 380000fc:       02e7c263                blt     a5,a4,38000120 <fir+0x94>
```

658: a4 = 0

659: a5 = 0

660: a5 = 0

661: a5 = 0 + 0 = 0

662: a5 = 0(a5)

663: -32(s0) = a5

664: a4 = 0

…

```
675 38000120:       fec42783                lw      a5,-20(s0)
676 38000124:       00b78713                addi    a4,a5,11
677 38000128:       fe442783                lw      a5,-28(s0)
678 3800012c:       40f707b3                sub     a5,a4,a5
679 38000130:       05c00713                li      a4,92
680 38000134:       00279793                slli    a5,a5,0x2
681 38000138:       00f707b3                add     a5,a4,a5
682 3800013c:       0007a783                lw      a5,0(a5)
683 38000140:       fcf42e23                sw      a5,-36(s0)
684 38000144:       fe042583                lw      a1,-32(s0)
685 38000148:       fdc42503                lw      a0,-36(s0)
686 3800014c:       eb5ff0ef                jal     ra,38000000 <__mulsi3>
```

Jump to 38000120 to do the multiplication:

```
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
```

return the multiplication result

```
687 38000150:       00050793                mv      a5,a0
688 38000154:       00078713                mv      a4,a5
689 38000158:       fe842783                lw      a5,-24(s0)
690 3800015c:       00e787b3                add     a5,a5,a4
691 38000160:       fef42423                sw      a5,-24(s0)
692 38000164:       fe442783                lw      a5,-28(s0)
693 38000168:       00178793                addi    a5,a5,1
694 3800016c:       fef42223                sw      a5,-28(s0)
695 38000170:       fe442703                lw      a4,-28(s0)
696 38000174:       00a00793                li      a5,10
697 38000178:       f6e7d2e3                bge     a5,a4,380000dc <fir+0x50>
```

Store the maltiplication result to stack.

…… the assembly code just implement the code just like in C. However, the register, address, and stack pointer all jump instructions make it very difficult for people to understand its implementation flow (unlike high level language), but after going through the risc-v code, we still can see the implementation pattern.

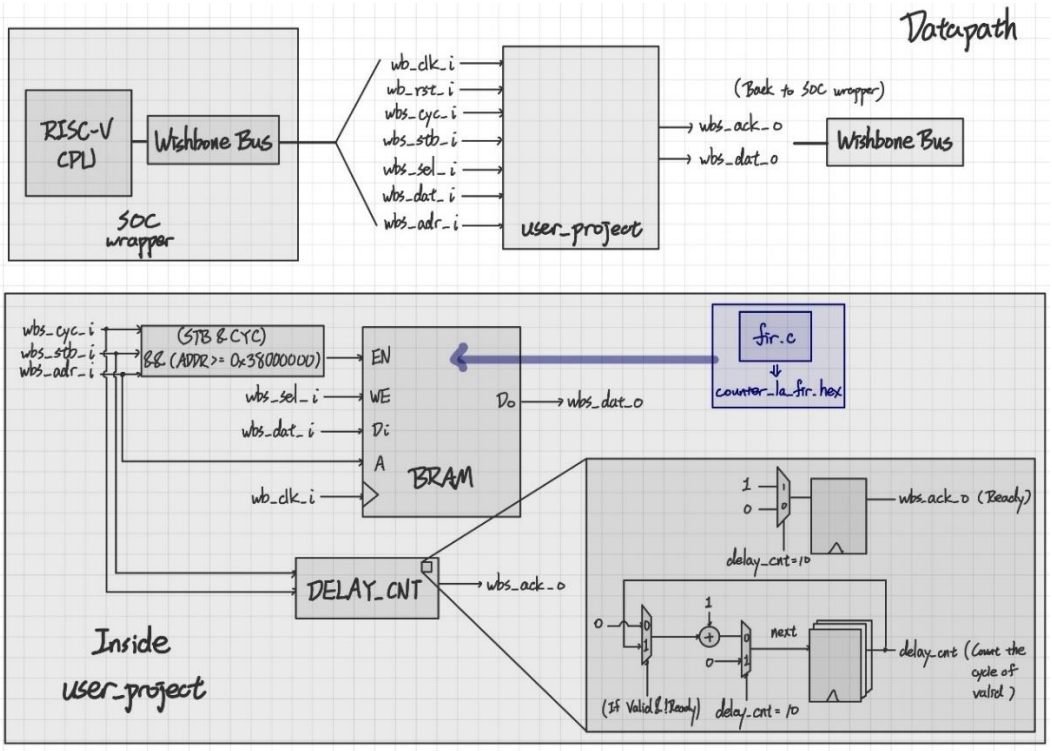
- **What address allocate for user project and how many space is required to allocate to firmware code.**

  In this Lab, spec said that it provides 4K bytes memory for our mprjram

  Our firmware code uses 0x38000000 to 0x380001c0.

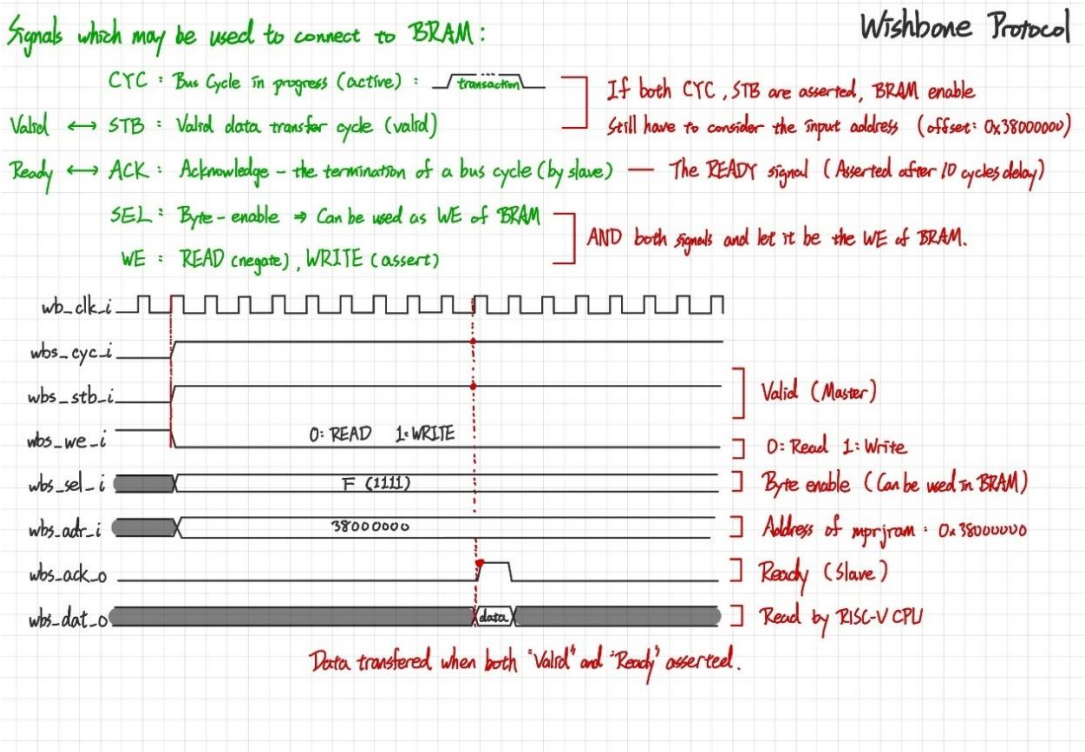
- **Interface between BRAM and wishbone**

- **Datapath**



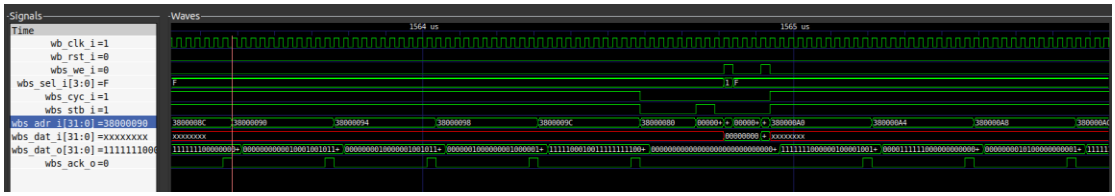
Our firmware code **fir.c** will be compiled as **counter_la_fir.hex,** and loaded into block RAM inside our user project by testbench.

- **Wishbone protocol**



- **Waveform from xsim**

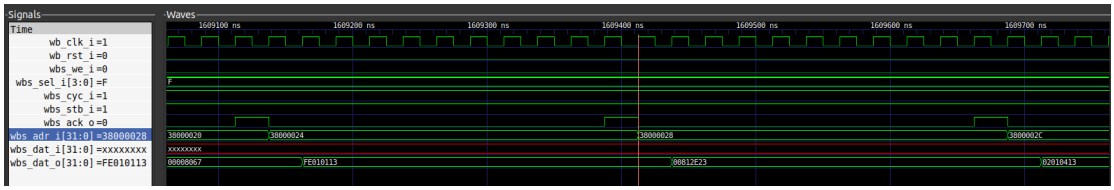
0x3800008c – 0x380001c0:



The waveform above is doing our firmware code (fir.c)

```
3800008c <fir>:
3800008c: fe010113          addi  sp,sp,-32
38000090: 00112e23          sw  ra,28(sp)
38000094: 00812c23          sw  s0,24(sp)
38000098: 02010413          addi  s0,sp,32
3800009c: f89ff0ef          jal ra,38000024 <initfir>
380000a0: fe042623          sw  zero,-20(s0)
380000a4: 0fc0006f          j 380001a0 <fir+0x114>
380000a8: 00a00793          li  a5,10
380000ac: fef42423          sw  a5,-24(s0)
380000b0: 03c0006f          j 380000ec <fir+0x60>
380000b4: fe842783          lw  a5,-24(s0)
380000b8: fff78793          addi  a5,a5,-1
380000bc: 05c00713          li  a4,92
380000c0: 00279793          slli  a5,a5,0x2
380000c4: 00f707b3          add a5,a4,a5
380000c8: 0007a703          lw  a4,0(a5)
380000cc: 05c00693          li  a3,92
380000d0: fe842783          lw  a5,-24(s0)
380000d4: 00279793          slli  a5,a5,0x2
380000d8: 00f687b3          add a5,a3,a5
380000dc: 00e7a023          sw  a4,0(a5)
380000e0: fe842783          lw  a5,-24(s0)
380000e4: fff78793          addi  a5,a5,-1
380000e8: fef42423          sw  a5,-24(s0)
380000ec: fe842783          lw  a5,-24(s0)
380000f0: fcf042e3          bgtz  a5,380000b4 <fir+0x28>
380000f4: 02c00713          li  a4,44
380000f8: fec42783          lw  a5,-20(s0)
380000fc: 00279793          slli  a5,a5,0x2
38000100: 00f707b3          add a5,a4,a5
38000104: 0007a703          lw  a4,0(a5)
38000108: 05c00793          li  a5,92
3800010c: 00e7a023          sw  a4,0(a5)
38000110: fe042223          sw  zero,-28(s0)
38000114: fe042023          sw  zero,-32(s0)
38000118: 0580006f          j 38000170 <fir+0xe4>
3800011c: 00000713          li  a4,0
38000120: fe042783          lw  a5,-32(s0)
38000124: 00279793          slli  a5,a5,0x2
38000128: 00f707b3          add a5,a4,a5
3800012c: 0007a683          lw  a3,0(a5)
38000130: 05c00713          li  a4,92
38000134: fe042783          lw  a5,-32(s0)
```

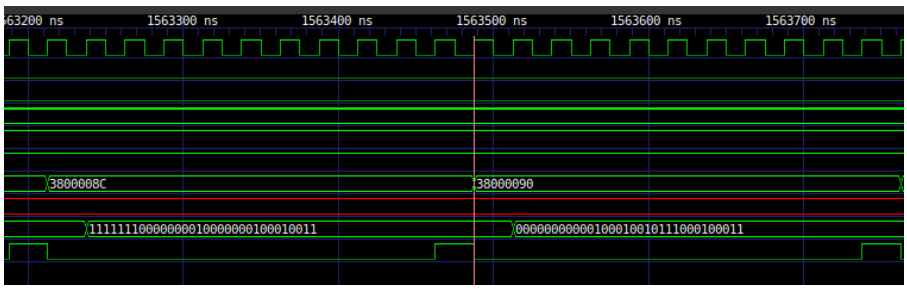
0x38000024 – 0x38000088:



Due to the assembly code, we can see the waveform above is doing the function initfir(), which be used in our firmware code to initialize the input buffer and output buffer.

```
38000024 <initfir>:
38000024: fe010113          addi  sp,sp,-32
38000028: 00812e23          sw  s0,28(sp)
3800002c: 02010413          addi  s0,sp,32
38000030: fe042623          sw  zero,-20(s0)
38000034: 0380006f          j 3800006c <initfir+0x48>
38000038: 05c00713          li  a4,92
3800003c: fec42783          lw  a5,-20(s0)
38000040: 00279793          slli  a5,a5,0x2
38000044: 00f707b3          add a5,a4,a5
38000048: 0007a023          sw  zero,0(a5)
3800004c: 08800713          li  a4,136
38000050: fec42783          lw  a5,-20(s0)
38000054: 00279793          slli  a5,a5,0x2
38000058: 00f707b3          add a5,a4,a5
3800005c: 0007a023          sw  zero,0(a5)
38000060: fec42783          lw  a5,-20(s0)
38000064: 00178793          addi  a5,a5,1
38000068: fef42623          sw  a5,-20(s0)
3800006c: fec42703          lw  a4,-20(s0)
38000070: 00a00793          li  a5,10
38000074: fce7d2e3          bge a5,a4,38000038 <initfir+0x14>
38000078: 00000013          nop
3800007c: 00000013          nop
38000080: 01c12403          lw  s0,28(sp)
38000084: 02010113          addi  sp,sp,32
38000088: 00008067          ret
```

Zoom in and we can see that ACK signal will have a delay of 10 clock cycles.



- **Synthesis report**

  1. **Utilization**

```
1. Slice Logic
--------------

+-------------------------+------+-------+------------+-----------+-------+
|        Site Type        | Used | Fixed | Prohibited | Available | Util% |
+-------------------------+------+-------+------------+-----------+-------+
| Slice LUTs*             |   15 |     0 |          0 |     53200 |  0.03 |
|   LUT as Logic          |   15 |     0 |          0 |     53200 |  0.03 |
|   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 |
+-------------------------+------+-------+------------+-----------+-------+
```

- Our user project design contains 15 LUTs and 5 registers.

2. Memory

```
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 |
+-------------------+------+-------+------------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate
only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that
tile can still accommodate a RAMB18E1
```

Since we use block RAM as ‘memory’ in our user project design.

- **Comment**

Vic: 這次要使用 wishbone 的 protocol，我認為在這個 Lab 中做起來比 AXI-lite 還簡單跟直接，訊號本身的特性就和 block RAM 需要的相關，像是 CYC 跟 STB 和 AXI 的 VALID 非常類似，老師上課有講到其實用 STB 當作 VALID，ACK 當作 READY 就可以了，但我在 user project 中還是有把 CYC 加進去，代表 processor 正在運作中。再來就是 WE 跟 SEL，分別代表 READ/WRITE 和 Byte enable，只要把 WE 延伸成四個 bit 再和 SEL 做 AND，就能當作 block RAM 的 WE 來使用了，接下來的判斷都蠻直觀的。當我完成 run_sim 階段後，我寫了一個 Makefile (附在我的 github /SOC/Lab4-1-counter_la_fir/build&xsim 中)，目的是在做 xsim，但我發現本來的 include.rtl.list 有問題，因為每一行前面

的-v，導致在嘗試做 xsim 的時候會跳出 error，於是我創了一個新檔案叫做 include.rtl.list.xsim，並把-v 都拿掉，在 Makefile 裡面也寫了以下這段指令:

xvlog -f ./include.rtl.list.xsim counter_la_fir_tb.v

代表 include 這 include.rtl.list.xsim 裡面的.v 檔，然後給 counter_la_fir_tb 做 simulation。

雖然成功讓檔案跑起來，但 simulator 回報錯誤: “port redeclaring”，於是我進去 caravel-soc 的各個.v 檔，去看裡面的 code，發現 code 都是先宣告 input/output，然後在其他地方才宣告 reg/wire，這導致 simulator 認為我們重新定義同個 port，所以產生 error。我後來將各個 module 裡面需要定義 net type 的 port 設定好，再重新跑一次 make，並成功通過 simulation。

Kenny:

在寫.c 檔時，原本想把 Lab3 的 verilog code 轉換成 c code 來寫，但其實這樣會多用到許多變數，其實直接用 software 的寫法就好。後續轉換成 assembly code c 後，我負責的是說明乘法是如何運行的部分，這部分其實我在計算機結構的課程中有學過，但看到實際運行的結果還是花了一些時間才搞懂他運行的原理。這次的 Lab 雖然 coding 的部分不多，但透過了解老師提供的 code 及檔案，我更了解整個 caravel soc 的架構及運作方式。

Lukas: 這次 lab 仍以 FIR 為主題，但有別於 lab2 的 High level synthesis 和 lab3 的 verilog implementation，這次要從 C code 開始實作 FIR，原本想說要呵 lab3 verilog implementation 一樣用 pipeline 的方式來做，結果在上次問教授的時候才恍然大悟 C code 沒有 pipeline task 的問題，因此在實作上用兩個迴圈就能達成。另外就是 wishbone interface 的實作，相對於 AXI interface，wishbone interface 可以說是直覺許多，要判斷的訊號也比較少，尤其是和 DRAM 的 interconnect. 最後，因為我的組員把 simulation 的步驟做得很完整了，在這邊就

蠻順利的，最後是 RISC-V code interpretation，我負責的是 FIR fuction 的部分，不像 high level language，risc-v code 在閱讀上非常不直覺，還有 stack, adress 和 jump instructions，而變數也沒有名稱。經過這個 lab，對於 caravel SOC 有更多的理解。